Relinche: Automatically Checking Linearizability under Relaxed Memory Consistency

PAVEL GOLOVIN, MPI-SWS, Germany
MICHALIS KOKOLOGIANNAKIS, ETH Zurich, Switzerland
VIKTOR VAFEIADIS, MPI-SWS, Germany

Concurrent libraries implement standard data structures, such as stacks and queues, in a thread-safe manner, typically providing an atomic interface to the data structure. They serve as building blocks for concurrent programs, and incorporate advanced synchronization mechanisms to achieve good performance.

In this paper, we are concerned with the problem of verifying correctness of such libraries under weak memory consistency in a fully automated fashion. To this end, we develop Relinche, a model checker that verifies atomicity and functional correctness of a concurrent library implementation in any client program that invokes the library methods up to some bounded number of times. Our tool establishes refinement between the concurrent library implementation and its atomic specification in a fully parallel client, which it then strengthens to capture all possible other (more constrained) clients of the library.

Relinche scales sufficiently to verify correctness of standard concurrent library benchmarks for all client programs with up to 7–9 library method invocations, and finds minimal counterexamples with 4–7 method calls of non-trivial linearizability bugs due to weak memory consistency.

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

Additional Key Words and Phrases: Model Checking, Weak Memory Consistency, Linearizability

ACM Reference Format:
Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. Relinche: Automatically Checking Linearizability under Relaxed Memory Consistency. *Proc. ACM Program. Lang.* 9, POPL, Article 70 (January 2025), 28 pages. https://doi.org/10.1145/3704906

1 Introduction

Despite the abundant opportunities for parallel execution in modern computing platforms, exploiting them correctly and efficiently remains a big challenge for programmers. A partial answer to this challenge is given by *concurrent libraries*, such as `java.util.concurrent` [Lea 2005], `oneTBB` [Reinders 2007], `libcds` [Khizhinsky n.d.], and `ckit` [Bahra n.d.]. These libraries implement standard data structures like stacks and queues in a "thread-safe" manner, encapsulating most of the intricacies of inter-thread synchronization, and enabling ordinary programmers to write correct concurrent programs without having to worry about such details. Achieving correct and efficient synchronization is left to the expert implementers of these libraries, who may use a collection of sophisticated mechanisms for

- fine-grained synchronization (e.g., optimistic traversals, lazy updates, helping [Herlihy and Shavit 2008]),

Authors' Contact Information: Pavel Golovin, MPI-SWS, Kaiserslautern, Germany, pgolovin@mpi-sws.org; Michalis Kokologiannakis, ETH Zurich, Zurich, Switzerland, michalis.kokologiannakis@inf.ethz.ch; Viktor Vafeiadis, MPI-SWS, Kaiserslautern, Germany, viktor@mpi-sws.org.

ACM 2475-1421/2025/1-ART70
https://doi.org/10.1145/3704906

- contention reduction (e.g., elimination [Hendler et al. 2004]), and
- memory reclamation (e.g., hazard pointers [Michael 2004], RCU [Desnoyers et al. 2012]),

and often even optimize the code for a specific architecture.

As these libraries are essential building blocks for concurrent programs, verifying their correctness has been an active area of research over the past 20-30 years. Most research has assumed the context of *sequential consistency* (SC) [Lamport 1979], where the threads execute in an interleaving fashion and no out-of-order execution effects are observable. In the SC setting, the correctness of a concurrent library is given by *linearizability* [Herlihy and Wing 1990], which requires each library method to behave as if it were executed in a single atomic step.

In this paper, we focus on verifying correctness of concurrent libraries in a *weak memory consistency* setting, where a concurrent program can observe the effects of out-of-order execution. In this setting, classical linearizability is no longer a suitable correctness condition because efficient concurrent library implementations do not fully encapsulate the effects of weak memory, and often expose them at the library interface. For example, the stack and queue libraries of `libcds` (and similarly, of `libvsync`), if used together, can exhibit the following *"store buffering"* behavior:

$$\begin{array}{l||l} \text{T1: } x.\text{push}(1) & \text{T2: } y.\text{enqueue}(2) \\ \quad a := y.\text{dequeue}() \ /\!/\ \textit{returns } \perp & \quad b := x.\text{pop}() \ /\!/\ \textit{returns } \perp \end{array} \qquad \text{(SB)}$$

Consequently, several papers have introduced correctness notions for concurrent libraries in the weak memory setting. While some authors, e.g., Batty et al. [2013] and Burckhardt et al. [2012], have defined adaptations of linearizability suitable for specific weak memory consistency models, we prefer to follow Raad et al. [2019], who provide a generic framework for defining the correctness of concurrent libraries in a weak memory setting, that can express linearizability.

An important aspect of these works is that a library, besides having to ensure atomicity of its methods, must also define the synchronization that its operations induce. For example, a queue library must stipulate that matching enqueue and dequeue operations synchronize in order to support the *"message passing"* idiom, where the following weakly consistent behavior is forbidden.

$$\begin{array}{l||ll} \text{T1: } x.\text{push}(1) & \text{T2: } a := y.\text{dequeue}() & /\!/\ \textit{returns } 2 \\ \quad y.\text{enqueue}(2) & \quad b := x.\text{pop}() & /\!/\ \textit{returns } \perp \end{array} \qquad \text{(MP)}$$

Despite a large body of work on verifying concurrent libraries, however, there exists no fully automated, sound verification approach for concurrent libraries under weak memory consistency. Most works are largely manual proof efforts, often with the support of an interactive proof assistant and/or a program logic. The closest work would be the pioneering work of Burckhardt et al. [2007], which establishes classical linearizability specifications for specific library client programs, and thus cannot distinguish between queue implementations that differ in their support of the MP and SB patterns.

In this paper, we present Relinche, the first widely-applicably, fully-automated, sound relaxed linearizability checker for verifying concurrent libraries. In Relinche, we avoid making assumptions about the implementations of the libraries subject to verification, such as having fixed linearization points, because such assumptions would restrict the applicability of our approach. Since the general verification problem is undecidable, however, we do restrict our attention to *bounded linearizability*, namely that the library is correct in all clients that invoke at most K library operations, which is a parameter of our verification approach. Relinche is based on the following three key ideas.

First, following Raad et al. [2019], we represent program executions as a set of consistent *execution graphs* with events representing method invocations and responses. The graph representation enables us to distinguish plain communication edges from synchronization edges, and to thus specify the synchronization induced by a library execution. Following Singh and Lahav [2023]. we write specifications as code using library-local mutexes along with C/C++11 primitives to specify the desired synchronization.

Second, we use state-of-the-art stateless model checkers [Kokologiannakis et al. 2022, 2019] to generate the consistent execution graphs of a program. Similar to Burckhardt et al. [2007], for a given library client, we collect the set of all possible outcomes of the specification and check that each implementation execution produces an outcome allowed by the specification and induces at least as much synchronization as required by the specification.

Third, to make our verification applicable to *all* library clients with up to K method invocations (and not a single client), we introduce the notion of the *most parallel client* (MPC), which consists of K parallel threads, each invoking one library method. While the MPC fully explores the state-space of the library, on its own, it does not suffice for checking library correctness, as it allows the methods to be linearized in any order (and so it cannot check e.g., that a queue implementation follows a FIFO order or that a stack follows LIFO order). As such, along with the possible outcomes of the specification, for each possible outcome we also calculate and record the set of *minimal happens-before extensions* that would render the outcome impossible. Then, for each execution of the implementation, we check that its outcome is allowed by the specification and that the corresponding recorded happens-before extensions also render it inconsistent.

We have implemented RELINCHE as a new verification tool, and show that (a) it scales sufficiently well for verifying relaxed linearizability of concurrent libraries for all clients with up to 7–9 operations, and (b) it quickly finds errors in incorrect library implementations: RELINCHE was able to show for the first time that well-known, SC-linearizable data structures are not linearizable under weak memory; the minimal linearizability violations are non-trivial and involve 4–7 threads. More impressively, RELINCHE was able to find an unknown correctness problem in an implementation of Michael's list [Michael 2002] used in `libcds` [Khizhinsky n.d.], a C++ library of concurrent data structures that has more than 2.5k stars on Github.

Our contributions can be summarized as follows:

§2 We start with an informal overview of our approach.
§3 We introduce a semantic framework for specifying and verifying concurrent libraries in a weak memory setting.
§4 We present RELINCHE in detail.
§5,§6 We discuss RELINCHE's implementation, evaluate it thoroughly, and discuss the linearizability violations found.

We conclude with a discussion of related work §7.

2 Overview

In this section, we present an overview of our verification approach with the help of the Herlihy-Wing queue [Herlihy and Wing 1990] below. (Our approach of course extends to other data structures; we use a queue only for demonstrational purposes.)

```
type HWQueue ≜ {
    int[]   array
    int     back
}
```

```
q.enqueue(v) ≜
    i := fetch_add^rel(q.back, 1)
    store^rel(q.array[i], v)
```

```
q.dequeue() ≜
    b := load^acq(q.back)
    i := 0; v := ⊥
    while (v = ⊥ ∧ i < b)
        v := xchg^acqrel(q.array[i], ⊥)
        i := i + 1
    return v
```

The queue is implemented as an index *back* over an infinite array, denoting the next free position for an item to be enqueued. The enqueue(v) method atomically reads and increments *back* (recording the value read just before the increment into the local variable i), and stores the value v at index i of the array. The dequeue method scans the array from 0 to *back* until it finds a non-empty value; once it does, it atomically exchanges the value with $\bot$.

Our implementation incorporates the C/C++11 access mode annotations suggested by Raad et al. [2019]: the enqueue accesses are marked as *release* (`rel`), the load of dequeue is marked as *acquire* (`acq`), while the atomic exchange as a combined *acquire-release* accesses (`acqrel`). The effect of these annotations is that whenever an acquire operation reads from a release operation, then the two operations synchronize and thus everything executed before the release is observed by the thread performing the acquire. By contrast, if some operation is marked as *relaxed* (`rlx`), then it does not synchronize with other operations and so does not induce any ordering constraints.

Although simple, this example will help us demonstrate several important points.

2.1 Queue Specification

Let us begin by discussing the specification of the Herlihy-Wing queue. To devise a suitable concurrent specification, we could start with a standard sequential queue implementation, and then wrap each method in an atomic block, as in the example below.

Example 2.1 (Global-lock queue specification) We implement the queue as a pair of indices *front* and *back* over an infinite array *array*, and make enqueue and dequeue acquire a mutual exclusion lock *lock*.

```
type Queue ≜ {
   int[]   array
   int     front
   int     back
   lock    lock
}
```

```
q.enqueue(v) ≜
   lock(q.lock)
   q.array[q.back] := v
   q.back := q.back + 1
   unlock(q.lock)
```

```
q.dequeue() ≜
   lock(q.lock)
   if (q.back ≤ q.front) v := ⊥
   else v := q.array[q.front]
        q.front := q.front + 1
   unlock(q.lock)
   return v
```

While holding the lock, enqueue simply appends an item at the back of the array, while dequeue checks that the queue is non-empty and if so, removes an item from the front. Observe that this implementation is not really similar to the Herlihy-Wing one, but that should not preclude us from using it as a basis for a specification.

One may be tempted to use this globally locked queue implementation as a reasonable specification of a concurrent queue. Although this works in a sequentially consistent setting, it unfortunately does not under weak memory consistency. The issue is that the contention on the global lock forces all operations to synchronize with one another, even if they logically commute. Efficient concurrent queue implementations—and the Herlihy-Wing queue in particular—avoid such excessive synchronization, and thus cannot be shown to refine the aforementioned queue specification.

One therefore needs to specify which queue operations are meant to synchronize with one another. At the very least, we would need to enforce that the n^{th} successful dequeue synchronizes with the n^{th} enqueue, i.e., with the enqueue whose value it returns. This constraint ensures that the message passing idiom (the program MP from §1) works as expected. (If the enqueue and the successful dequeue do not synchronize, then the push might not have been propagated to T2, and so T2 can observe the stack being empty.)

Stronger specifications may additionally require that all enqueues synchronize (which guarantees that if a dequeue sees an enqueue, then it is also aware of all previous enqueues) or, symmetrically, that all dequeues synchronize.

The question now is: How shall we write such a specification? The literature contains two different approaches for doing so: either to write the specification directly in logic [Dongol et al. 2018; Raad et al. 2019] or as code using C/C++11 constructs (and possibly some additional specification constructs) [Batty et al. 2013; Singh and Lahav 2023]. We prefer the latter approach, i.e., to specify the desired synchronization *directly inside the specification program*, because we think it is more understandable to the developers of concurrent libraries.

Specifying the desired synchronization only using C/C++11 constructs, however, can be a challenge! As the following example demonstrates, specifying that only matching enqueue and dequeue operations synchronize requires a fair amount of ingenuity and can be as difficult as designing an efficient queue implementation.

Example 2.2 (Lock-free queue specification)

```
type Queue ≜ {
   int[]   $array$
   int     $fb$
}
```

```
$q$.enqueue($v$) ≜
   $k$ := fetch_add$^{\mathsf{rlx}}$($q.fb$, 1)
   $b$ := $k \bmod 2^{32}$
   store$^{\mathsf{rel}}$($q.array[b], v$)
```

```
$q$.dequeue() ≜
   do $k$ := load$^{\mathsf{rlx}}$($q.fb$)
      $\langle f, b\rangle := \langle k \div 2^{32}, k \bmod 2^{32}\rangle$
      if ($f \geq b$) return $\bot$
   while ($\neg$CAS$^{\mathsf{rlx}}$($q.fb, k, k + 2^{32}$))
   do $v$ := load$^{\mathsf{acq}}$($q.array[b]$) while ($v = \bot$)
   return $v$
```

This lock-free queue specification uses a single field *fb* to record both pointers as $2^{32} \times \mathit{front} + \mathit{back}$. The enqueue method uses a fetch-and-add instruction to atomically increment the *back* pointer, while dequeue consists of a do-while loop that tries to advance *front*, and another loop that waits for the enqueued item to appear in *array*.

The reason why this implementation achieves the desired synchronization is that all accesses apart from the ones to *array* are relaxed. Since each index of *array* can be accessed by at most one enqueue/dequeue pair, the only synchronization that is ever created takes place during item dequeueing.

We argue that the lock-free specification is so complex because the code implementing the sequential data structure semantics is entangled with the code enforcing atomicity and the code enforcing the appropriate synchronization. To simplify writing specifications, we thus need to disentangle these three different aspects. To that end, we employ an idea by Singh and Lahav [2023], and extend the C++ memory model with a special "partial lock" that provides mutual exclusion and synchronization internally, but whose synchronization is discounted when computing the induced synchronization of the library specification. With this construct, we can specify a concurrent queue much more easily as in the example below.

Example 2.3 (Partial-lock queue specification)

```
type Queue ≜ {
   int[]   $array$
   int     $front$
   int     $back$
   plock   $lock$
}
```

```
$q$.enqueue($v$) ≜
   plock($q.lock$)
   store$^{\mathsf{rel}}$($q.array[q.back], v$)
   $q.back := q.back + 1$
   punlock($q.lock$)
```

```
$q$.dequeue() ≜
   plock($q.lock$)
   if ($q.back \leq q.front$) $v := \bot$
   else $v$ := load$^{\mathsf{acq}}$($q.array[q.front]$)
          $q.front := q.front + 1$
   punlock($q.lock$)
   return $v$
```

Note that, since we discount the synchronization due to the library locks, the only synchronization left is between matching enqueue and dequeue operations, which is induced by the corresponding write and read of the queue *array* (these are implemented using release/acquire accesses, respectively).

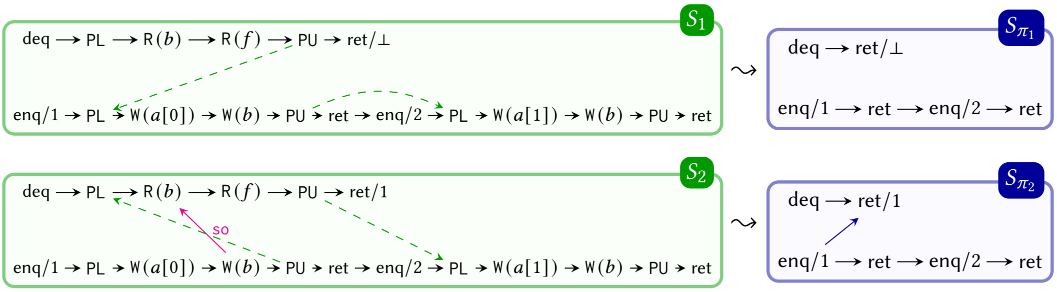

S_1
deq → PL → R(b) → R(f) → PU → ret/⊥
enq/1 → PL → W(a[0]) → W(b) → PU → ret → enq/2 → PL → W(a[1]) → W(b) → PU → ret
S_{π_1}
deq → ret/⊥
enq/1 → ret → enq/2 → ret
S_2
deq → PL → R(b) → R(f) → PU → ret/1
so
enq/1 → PL → W(a[0]) → W(b) → PU → ret → enq/2 → PL → W(a[1]) → W(b) → PU → ret
S_{π_2}
deq → ret/1
enq/1 → ret → enq/2 → ret

Fig. 1. Specification graphs for FIFO-CLIENT (left) and their projections (right)

This partial-lock queue specification is just a bit more complex than the global-lock queue, and so we think is easier to write and understand. Our verification methodology is actually orthogonal to the choice of specification, but for simplicity, we will use the partial-lock queue specification when discussing the verification of the Herlihy-Wing queue.

2.2 Showing Correctness for One Client

Next, let us consider verifying the correctness of the Herlihy-Wing queue for a single concurrent client, such as the following:

$$\begin{array}{r||l} \text{T1: } q.\text{enqueue}(1); & \text{T2: } a := q.\text{dequeue}(); \\ q.\text{enqueue}(2); & \end{array} \qquad \text{(FIFO-CLIENT)}$$

A good way of doing this automatically is via model checking. Model checkers with built-in support for weak memory consistency (e.g., GenMC [Kokologiannakis et al. 2019]), verify a concurrent program by representing its executions as a set of graphs, and then enumerating this set.

If, while enumerating all the execution graphs of the Herlihy-Wing queue on the FIFO-CLIENT, we find a graph containing the outcome $a = 2$, then we clearly have found a violation of the FIFO property of the queue specification. But it is worth wondering: Is $a = 2$ the only invalid outcome?

To answer this question, we can use the partial-lock queue specification from §2.1 to generate all valid outcomes of FIFO-CLIENT. Indeed, similarly to how we run a model checker on FIFO-CLIENT applied to the Herlihy-Wing implementation, we can also run it on the client applied to the partial-lock queue specification. Some graphs generated during the verification of FIFO-CLIENT applied to the partial-lock specification can be seen in Fig. 1 (left), while graphs generated when applying the Herlihy-Wing implementation can be seen in Fig. 2 (left).

Looking at Fig. 1 more closely, the nodes of these graphs correspond to the instructions of the queue specification with T2 above and T1 below. The graph edges denote relations among the instructions: the horizontal edges denote *program order* (i.e., relate instructions of the same thread), while the *synchronization* edges, so, show where reads are reading from in a synchronizing fashion (from a *release* write to an *acquire* read). The remaining dashed edges demonstrate the lock acquisition order: in S_1, T2 acquires the partial lock first, and then T1 reads from the unlock of T2.

At this point, we have seen that model checking can be used to enumerate the graphs of a client applied to the specification and the implementation, respectively. We avoid discussing how the model checker works internally to achieve this, as the inner workings of the model checker are not relevant for our approach.

As explained in §1, to show that the Herlihy-Wing implementation is linearizable, we can simply show *refinement* between the graphs of the implementation and the specification, i.e., show that

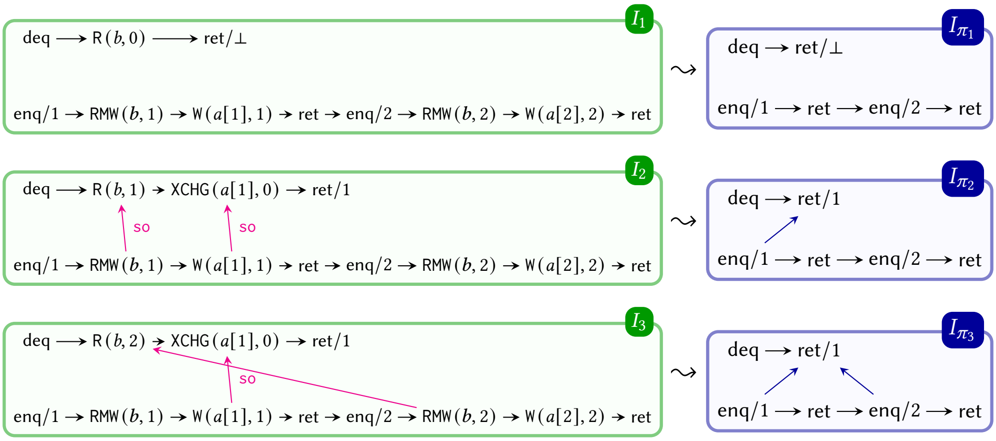

I_1
deq ⟶ R(b, 0) ⟶ ret/⊥
enq/1 → RMW(b, 1) → W(a[1], 1) → ret → enq/2 → RMW(b, 2) → W(a[2], 2) → ret
I_{π_1}
deq ⟶ ret/⊥
enq/1 ⟶ ret ⟶ enq/2 ⟶ ret
I_2
deq ⟶ R(b, 1) → XCHG(a[1], 0) ⟶ ret/1
so
so
enq/1 → RMW(b, 1) → W(a[1], 1) → ret → enq/2 → RMW(b, 2) → W(a[2], 2) → ret
I_{π_2}
deq ⟶ ret/1
enq/1 ⟶ ret ⟶ enq/2 ⟶ ret
I_3
deq ⟶ R(b, 2) → XCHG(a[1], 0) ⟶ ret/1
so
enq/1 → RMW(b, 1) → W(a[1], 1) → ret → enq/2 → RMW(b, 2) → W(a[2], 2) → ret
I_{π_3}
deq ⟶ ret/1
enq/1 ⟶ ret ⟶ enq/2 ⟶ ret

Fig. 2. Implementation graphs for FIFO-client (left) and their projections (right)

each implementation graph refines some graph of the specification. But how can we check for refinement given that the execution graphs of the specification and the implementation are quite different?

Relinche does this by generating what we call *projection graphs*. Such graphs are obtained by erasing events related to queue internals, and only recording stub events for the beginning and end of each operation. The projection graphs of the specification and implementation graphs of FIFO-client can be seen in Figures 1 and 2, respectively.

Let us now examine these projection graphs in more detail, focusing on the projections of the implementation (Fig. 2, right). These graphs do not record rf, as we do not care about where reads are reading in general, but rather about synchronization among methods. As such, we only record *happens-before* paths between these events, where happens-before hb ≜ $(\mathsf{po} \cup \mathsf{so})^{+}$po is the smallest transitive relation containing po and so edges. Observe that, despite having the same events, graphs I_2 and I_3 have different projections: the dequeue in graph I_{π_3} also synchronizes with enqueue(2) because it reads the updated queue size.

Given a projection graph of the implementation I_π, we can checks refinement by finding a corresponding projection graph of the specification S_π. S_π might be the same as graph I_π itself (as is the case with graphs I_{π_2} and S_{π_2}), or a graph with less synchronization (as is the case with I_{π_3} and S_{π_2}).

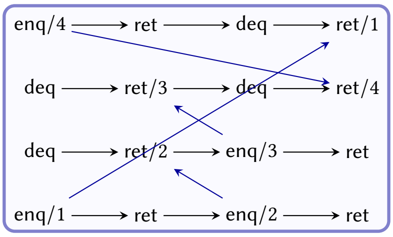
Allowing the specification projection to induce less synchronization than the implementation projection is important. Suppose, for example, that we want to show refinement between a naive queue implementation that acquires a global lock at each method call, and our queue specification. The method invocations in its projected implementation graphs are totally ordered by synchronization, unlike the specification ones, which are only partially ordered.

2.3 Showing Correctness for All Bounded Clients

We move on to the more challenging problem of checking correctness of the Herlihy-Wing queue implementation for all clients of the queue with at most K queue method invocations. The motivation for this problem stems from the fact that, as the following example demonstrates, it can be quite challenging to devise client programs that fully check the correctness of a data structure.

Example 2.4 (Non-linearizability in Herlihy-Wing) In this example, we consider a variant of the Herlihy-Wing implementation that uses an "acquire" exchange in the dequeue method instead of an "acquire-release" exchange. While this variant is correct on FIFO-client, it is incorrect on the client below that has 4 threads and 8 operations [Raad et al. 2019]. (The comments in the code below indicate the dequeued values.)

```
T1: q.enqueue(1)  ||  T2: q.dequeue() // 2  ||  T3: q.dequeue() // 3  ||  T4: q.enqueue(4)
    q.enqueue(2)  ||      q.enqueue(3)      ||      q.dequeue() // 4  ||      q.dequeue() // 1
```



The projection graph of a non-linearizable behavior can be seen on the right. For this behavior, it is impossible to totally order the method invocations in accordance with the synchronization obtained by the implementation. Indeed, the hb obtained by the implementation methods implies that the enqueues are ordered as enqueue(1) → enqueue(2) → enqueue(3) → enqueue(4): the first pair due to po, the second due to T2 dequeueing 2 before enqueueing 2, and the third due to T3 dequeueing first 3 and then 4. The issue is that this order among the enqueues also implies an ordering among the dequeues (due to FIFO), and in particular that enqueue(4) is ordered before enqueue(2) (due to T4's dequeue being FIFO-ordered before enqueue(2)), which is in turn ordered before enqueue(4) (and hence T4's dequeue, due to po).

As the example above demonstrates, while FIFO-client does check the FIFO property, it does not do so completely.

Relinche checks refinement for all clients by considering a *most parallel client* (MPC) with at most K operations. MPCs consist of K threads each one invoking a single method call. For instance, an MPC with 3 operations (two enqueues and one dequeue), can be seen below.

```
T1: q.enqueue(1) || T2: q.enqueue(2) || T3: q.dequeue()          (mpc-client)
```

Applying the method of § 2.2, we can easily show that the Herlihy-Wing queue is correct on this client. This is unsurprising and not very useful, since all three possible return values for the dequeue (⊥, 1, and 2) are allowed by the queue specification. What we need to do now is extend this verification result to arbitrary other clients with two calls to enqueue and one to dequeue.

In turns out we can extend the verification result to arbitrary other clients that use the same type of method invocations by making the following key observation: the executions of every other such client are *partial linearizations* of the executions of mpc-client, where some of the method invocations are ordered by the client, and may additionally contain some other client events. For instance, FIFO-client can be thought of as a specialization of mpc-client where the two enqueue calls are ordered, while MP can be considered a specialization of mpc-client where each execution further contains events of the stack library.

However, assuming that the return values of the library calls and the induced synchronization among them fully capture the library semantics, we can safely *ignore* additional client events: it suffices to only consider clients that provide additional ordering among library method invocations. In turn, all such clients are specializations (i.e., partial linearizations) of an MPC. Since there are finitely many partial linearizations of an MPC, one can, in principle, generate all possible bounded clients of the queue (each one corresponds to a different linearization), and check that the Herlihy-Wing queue is correct in each one.

The key idea behind Relinche is that we do not even have to explicitly generate all bounded clients corresponding to an MPC. Instead, Relinche utilizes the observation that the executions of any partially linearized client can be obtained by taking the executions of the MPC, and then adding hb edges between the appropriate method invocations. To see this, take an arbitrary execution of a client that specializes the MPC. Since the MPC is less constrained (in terms of synchronization) than the client, the same execution can be obtained by taking an MPC execution and adding hb edges to it. As such, Relinche is able to check linearizability of all clients by taking each projection graph of the MPC applied to the implementation, adding hb edges to it (as long as no hb cycle is created), and checking that there exists corresponding projection graph of the MPC applied to the specification that has less synchronization.

The procedure above suffices to verify the correctness of all bounded clients without actually generating them and re-running the model checker. However, Relinche does not stop there. Given a projection of the implementation, we observe that blindly checking all hb extensions is inefficient.

First, it suffices to only consider extensions that violate the corresponding specification projection. So, for example, we will never consider the hb extension $\text{enqueue}(1) \longrightarrow \text{enqueue}(2)$ for implementation executions where the dequeue returns $\bot$ because it does not invalidate the corresponding specification projection.

Second, due to monotonicity, it suffices to only consider *minimal* extensions that violate the corresponding specification projection. For example, we will never consider the hb extension $\text{enqueue}(1) \longrightarrow \text{enqueue}(2) \longrightarrow \text{dequeue}()$ as a violation of the projection where dequeue returns $\bot$ because the smaller extension $\text{enqueue}(1) \longrightarrow \text{dequeue}()$ also violates the same projection.

Finally, by the principle of symmetry reduction [Clarke et al. 1996], we can avoid exploring multiple minimal extensions that are symmetric to one another by declaring one of the symmetric extensions as the representative, and only considering symmetric linearizations. With this final optimization, we also need not consider the extension $\text{enqueue}(2) \longrightarrow \text{dequeue}()$ because it is symmetric to the extension $\text{enqueue}(1) \longrightarrow \text{dequeue}()$.

As such, our final algorithm works in two phases. First, Relinche calculates all specification outcomes, as well as the minimal hb extensions that violate each one. Then, for each projection graph of the implementation, it checks that (1) the projection is allowed by the specification and (2) that every minimal hb extension that violates the corresponding specification projection, also renders the implementation execution inconsistent. We mention in passing that such extensions can be generated and checked in a very compact way, but defer the presentation of our full algorithm to §4.

3 Contextual Refinement for Execution Graphs

In this section, we introduce execution graphs (§3.1) as a representation of program executions. We then define library specifications (§3.2), library implementations (§3.3), along with the semantics of programs (§3.4), and the correctness of a library implementation in terms of refinement (§3.5).

3.1 Executions

We assume domains of values (Val) and method names (Method), ranged over by v and m respectively. We further assume that each method expects a fixed number of arguments and that there is a function, $\text{arity}\colon \text{Method} \rightarrow \mathbb{N}$, that maps each method to the number of arguments that it expects.

A *library interface*, $A \subseteq \text{Method}$, is a set of method names belonging to the library.

We define four important semantic domains:

- *Events*, $e \in \text{Event}$, which represent either a single method invocation or the return of such an invocation.

- *Plain executions* (Pexec), which are partially ordered sets of events with the partial order po representing the *program order*, i.e., the order of events according to the program control flow.
- *Full executions* (Exec) extend plain executions with a *synchronization order* so, due to orderings by individual libraries.
- *Library executions* (Lexec_A) over a library interface, which are sets of events corresponding to methods in A along two kinds of orderings among them: synchronization orderings due to the library itself (so), and due to its environment (hbC).

We use G to range over all types of executions, since the type of execution will be clear from the context.

Definition 3.1 (Events and labels). An *event* $e \in \mathsf{Event} \triangleq \mathbb{N} \times \mathsf{Lab}$ is a tuple $\langle n, l \rangle$, where $n \in \mathbb{N}$ is an *event identifier* and l is an *event label.* Event labels are of the form $\mathsf{C}^m(v_1, \dots, v_n)$ and $\mathsf{R}^m(v)$, and represent a method invocation with arguments $v_1, \dots, v_n$ and a method response with return value v, respectively.

Definition 3.2 (Plain Executions). A *plain execution*, $G \in \mathsf{Pexec}$, is a tuple $G = \langle E, po \rangle$, where:

- $E = C \uplus R$ is a set of *events* comprising a set C of invocation events with distinct identifiers and a set R of matching response events with the same identifiers; and
- $po \subseteq E \times E$ is a strict partial order denoting the *program order* relation, such that each invocation event is *po*-before its matching response.

Definition 3.3 (Full Executions). A *full execution*, $G \in \mathsf{Exec}$, is a tuple $G = \langle E, po, \mathsf{so} \rangle$, where $\langle E, po \rangle$ is a plain execution and $\mathsf{so} \subseteq E \times E$ is the *synchronization order*, relating invocation events and response events of the same library.

Definition 3.4 (Library Executions). A *library execution* G of a library interface A is a tuple $\langle E, rf, so, hbC \rangle$:

- $E = C \uplus R$ is a set of events consisting of invocation events (C) of methods in A and matching responses (R);
- $so \subseteq C \times R$ is the *library synchronization* order, denoting ordering induced by the library; and
- $hbC \subseteq E \times E$ is the *client happens-before* order, denoting ordering induced by the environment of the library.

We use dot notation (e.g., $G.\mathsf{E}$, $G.\mathsf{so}$, etc) to project the various components of an execution G.

3.2 Library Specifications

A *specification* of a library interface, A, is a set of consistent library executions that is closed under prefixes, induced-ordering strengthenings, and client ordering relaxations.

Definition 3.5 (Library specification). A *specification* of a library interface A is a set $S \subseteq \mathsf{Lexec}_A$ that is closed in two ways:

Prefixes: If $\langle E, so, hbC \rangle \in S$ and $E' \subseteq E$ such that $dom((hbC \cup \mathsf{so}); [E']) \subseteq E'$, then
$\langle E', so \cap (E' \times E'), hbC \cap (E' \times E') \rangle \in S$.
Ordering: If $\langle E, so, hbC \rangle \in S$ and $so' \supseteq so$ and $hbC' \subseteq hbC$, then $\langle E, so', hbC' \rangle \in S$.

A *library* $L \triangleq \langle A, S \rangle$ is a pair of a library interface and a specification for that interface. Given a library L, we write $L.\mathsf{A}$ and $L.\mathsf{S}$ to project to its components. We say that two libraries L_1 and L_2 are *compatible* if their interfaces are disjoint ($L_1.\mathsf{A} \cap L_2.\mathsf{A} = \emptyset$). We write $\mathcal{L}$ for a set of pairwise compatible libraries.

The notion of libraries provided above does not assume any pre-existing libraries in the program. Indeed, even fundamental operations like reads and writes can be formulated as libraries in the framework above, and we make use of that fact when defining client executions below. As examples, we present three register libraries: one inducing no synchronization (relaxed register), one inducing synchronization along the information flow from a write to a read (release-acquire register), and one inducing synchronization for all non-commuting operations (SC register).

Example 3.1 (Relaxed Register) The relaxed register interface, $A_{\mathtt{rlx}}$, two comprises two methods: $\mathtt{store}^{\mathtt{rlx}}$ and $\mathtt{load}^{\mathtt{rlx}}$, the former taking a value argument and returning always $\bot$ and the latter taking no arguments but returning a value.

The relaxed register specification contains all executions $\langle E, \mathsf{so}, hbC\rangle$ of $A_{\mathtt{rlx}}$ such that there exists a total order $<_T$ over E extending hbC with the following properties:

- Every invocation event in E immediately precedes its matching response in $<_T$.
- Every load response event r returns the value written by the last write invocation event w that precedes it in $<_T$, or 0 if there is no previous store invocation event.

(Note that so is unconstrained because a relaxed register does not induce any synchronization.)

Example 3.2 (Release-Acquire Register) A release-acquire register is defined analogously with the additional constraint about the total order $<_T$ that:

- so contains all the $\langle w, r\rangle$ pairs, such that w is a write-invocation, r is a read-response that appears later in $<_T$ and there are no other write-invocation events between w and r in $<_T$.

Example 3.3 (SC Register) A sequentially consistent register further constrains so and $<_T$ so that:

- so contains all the $\langle e, e'\rangle$ pairs, such that $e <_T e'$, e is an invocation event, e' is a response event, and they are both read events.

3.3 Client Programs and Library Implementations

We define a simple language for implementing libraries and their clients. Expressions, $e \in \mathsf{Exp}$, comprise values $v \in \mathsf{Val}$, variables $x \in \mathsf{Var}$, and arithmetic and logical operations (without any side-effects), whereas programs $P \in \mathsf{Prog}$ can also define and invoke library methods and contain control-flow constructs (conditionals, sequencing, and parallel composition):

$$\begin{aligned} e &::= v \mid x \mid e_1 \oplus e_2 \quad \text{for } \oplus \in \{+, -, \times, \div, =, \neq, <, \leq, >, \geq, \ldots\} \\ P &::= e \mid m(e_1, \ldots, e_{\mathsf{arity}(m)}) \mid \mathbf{if}\ e\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid \mathbf{let}\ x = P_1\ \mathbf{in}\ P_2 \mid P_1 \parallel P_2 \\ &\quad \mid \mathbf{let}\ m(x_1, \ldots, x_{\mathsf{arity}(m)}) = P_1\ \mathbf{in}\ P_2 \end{aligned}$$

A *closed program*, $C \triangleq (\mathcal{L} \vdash P)$, is a program together with the set of libraries that it uses.

A *client program*, $C \triangleq (\mathcal{L} \vdash P)$, of a library interface A, is a program together with the set of libraries that it uses other than A. We write $C \triangleright S$ for the closed program $\mathcal{L} \cup \{\langle A, S\rangle\} \vdash P$, resulting from composing a client program C of A with a specification S of A.

An *implementation* of a library interface provides definitions for all the methods of the interface.

Definition 3.6. An *implementation*, I, of a library interface A, comprises a set of pairwise compatible libraries $\mathcal{L}_{\text{internal}}$ and a method definition $m(\vec{x}) = P$ for all methods $m \in A$, such that all the method names appearing in P belong to $\mathcal{L}_{\text{internal}}$, the free variables of P are included in $\vec{x}$, and $|\vec{x}| = \mathsf{arity}(m)$.

$$\llbracket e \rrbracket(\eta, \theta) \triangleq \{\langle \eta(e), G_\emptyset \rangle\}$$

$$\llbracket m(\vec{e}) \rrbracket(\eta, \theta) \triangleq \begin{cases} \left\{\langle v, G_c \,;\, G \,;\, G_r \rangle \;\middle|\; \langle v, G \rangle \in \theta(m)(\eta(\vec{e})), n \in \mathbb{N}\right\} & \text{if } m \in dom(\theta) \\ \left\{\langle v, G_c \,;\, G_r \rangle \;\middle|\; v \in \mathsf{Val}, n \in \mathbb{N}\right\} & \text{if } m \notin dom(\theta) \end{cases}$$

$$\text{where } c = \langle n, \mathsf{C}^m(\eta(e_1), \dots, \eta(e_n)) \rangle$$
$$r = \langle n, \mathsf{R}^m(v) \rangle$$

$$\llbracket \mathtt{let}\ m(\vec{x}) = P_1\ \mathtt{in}\ P_2 \rrbracket(\eta, \theta) \triangleq \llbracket P_2 \rrbracket(\eta, \theta[m \mapsto \lambda \vec{v}.\, \llbracket P_1 \rrbracket(\eta[\vec{x} \mapsto \vec{v}], \theta)])$$

$$\llbracket \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \rrbracket(\eta, \theta) \triangleq \begin{cases} \llbracket P_1 \rrbracket(\eta, \theta) & \text{if } \eta(e) \neq 0 \\ \llbracket P_2 \rrbracket(\eta, \theta) & \text{if } \eta(e) = 0 \end{cases}$$

$$\llbracket \mathtt{let}\ x = P_1\ \mathtt{in}\ P_2 \rrbracket(\eta, \theta) \triangleq \left\{\langle v_2, G_1 \,;\, G_2 \rangle \;\middle|\; \langle v_1, G_1 \rangle \in \llbracket P_1 \rrbracket(\eta, \theta) \wedge \langle v_2, G_2 \rangle \in \llbracket P_2 \rrbracket(\eta[x \mapsto v_1], \theta)\right\}$$

$$\llbracket P_1 \parallel P_2 \rrbracket(\eta, \theta) \triangleq \left\{\langle 0, G_1 \parallel G_2 \rangle \;\middle|\; \langle v_1, G_1 \rangle \in \llbracket P_1 \rrbracket(\eta, \theta) \wedge \langle v_2, G_2 \rangle \in \llbracket P_2 \rrbracket(\eta, \theta)\right\}$$

Fig. 3. Mapping of programs to plain execution graphs, $\llbracket \_ \rrbracket : \mathsf{Prog} \to \mathsf{Lenv} \to \mathsf{Menv} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Pexec})$.

Definition 3.7. The *composition* of a client program $C = (\mathcal{L} \vdash P)$ and an implementation I of a library interface $A = \langle \mathcal{L}_{\text{internal}}, m_1(\vec{x}) = P_1, \dots, m_n(\vec{x}) = P_n \rangle$, written $C \triangleright I$, is the client program:

$$\begin{aligned} (\mathcal{L}_{\text{internal}} \uplus \{L \in \mathcal{L} \mid L.\mathsf{A} \neq A\}) \vdash\ & \mathtt{let}\ m_1(\vec{x}) = P_1\ \mathtt{in} \\ & \dots \\ & \mathtt{let}\ m_n(\vec{x}) = P_n\ \mathtt{in}\ P \end{aligned}$$

3.4 Semantics of Client Programs and Library Implementations

We next describe the semantics of programs. The semantics of expressions is determined by a *variable environment*, $\eta \in \mathsf{Lenv} \triangleq \mathsf{Var} \to \mathsf{Val}$, which maps variables to their values, and a *method environment*, $\theta \in \mathsf{Menv}$, which maps method names to their semantic representations (which will be defined shortly). By abusing notation, we extend the domain of variable environments to expressions so that $\eta(v) = v$ and $\eta(e_1 \oplus e_2) = \eta(e_1) \oplus \eta(e_2)$.

The semantics of programs is given in two steps. In the first step, we define a function $\llbracket . \rrbracket$ that maps programs into sets of executions. Then, we check that the projection of these executions to the methods of each library satisfies the specification of that library.

In more detail, in the first step, the mapping of programs is given with respect to environments η, θ and returns a set of tuples $\langle v, G \rangle$, where v is the value returned by the program and G is the corresponding *plain execution*, recording the methods invoked by the program. Method environments have the following type:

$$\theta \in \mathsf{Menv} \triangleq (m : \mathsf{Method}) \rightharpoonup \mathsf{Val}^{\mathsf{arity}(m)} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Pexec})$$

To define the program mapping, we introduce some constructions on plain executions. We write $G_\emptyset \triangleq \langle \emptyset, \emptyset \rangle$ for the empty execution and $G_a \triangleq \langle \{a\}, \emptyset \rangle$ for the execution with a single event a.

Given two executions, $G_1 = \langle E_1, po_1 \rangle$ and $G_2 = \langle E_2, po_2 \rangle$, with disjoint sets of events, we define:

- their sequential composition, $G_1 \,;\, G_2 \triangleq \langle E_1 \cup E_2, po_1 \cup po_2 \cup (E_1 \times E_2) \rangle$, by ordering all G_1 events before those of G_2, and
- their parallel composition, $G_1 \parallel G_2 \triangleq \langle E_1 \cup E_2, po_1 \cup po_2 \rangle$, by placing no additional order between events of G_1 and G_2.

With these constructions, we can now define the mapping $\llbracket P \rrbracket$ by structural induction in Fig. 3. An expression returns the empty execution graph, $G_\emptyset$. A method call $m(e_1, \dots, e_n)$ creates two events with labels $\mathsf{C}^m(v_1, \dots, v_n)$ and $\mathsf{R}^m(v)$, denoting a call to method m with arguments $v_1, \dots, v_n$ (where $v_i = \eta(e_i)$) and its return with value v, respectively. In the case where the implementation of

m is known (part of θ), it generates the events of the implementation as well. The interpretation of a method definition extends θ by mapping the method name to the interpretation of its body. The interpretation of a conditional, **if** e **then** P_1 **else** P_2, returns $[\![P_1]\!]$ or $[\![P_2]\!]$ depending on the value of the condition. The interpretation of a let-expression, **let** $x = P_1$ **in** P_2, composes the execution graphs corresponding to P_1 and to P_2 sequentially, whereas that of a parallel command, $P_1 \parallel P_2$ composes them in parallel. The interpretation of a top-level program is performed with $\eta = \emptyset$.

We move on to the second step. The *consistent executions* of a closed program are the set of all full executions generated by the program that satisfy the specifications of all the libraries it uses.

Definition 3.8 (Consistent executions). A full execution G is a *consistent execution* of a closed program $C = (\mathcal{L} \vdash P)$ if

- $\langle v, \langle G.\mathsf{E}, G.\mathsf{po} \rangle \rangle \in [\![P]\!]$ for some value v, and
- for each $L \in \mathcal{L}$, it is $\langle G.\mathsf{E}|_{L.\mathsf{A}}, G.\mathsf{so}|_{L.\mathsf{A}}, (G.\mathsf{po} \cup G.\mathsf{so})^+|_{L.\mathsf{A}} \rangle \in L.\mathsf{S}$.

where $S|_A$ restricts a set S to events concerning methods in A, and similarly $r|_A$ similarly restricts a relation r.

We write $\mathsf{Execs}(C)$ for the set all consistent executions of a closed program C.

3.5 Library Implementation Correctness

We define the correctness of a library implementation as a refinement of its specification. To do so, we first need to define the observations for a library interface as the set of library events of that interface along with the ordering between those events.

Definition 3.9 (Observations). The *observation* of a full execution G for a library interface A is $\mathsf{Obs}_A(G) \triangleq \langle G.\mathsf{E}|_A, G.\mathsf{so}|_A \rangle$. The *observations* of a closed program C for a library interface A are $\mathsf{Obs}_A(C) \triangleq \{\mathsf{Obs}_A(G) \mid G \in \mathsf{Execs}(C)\}$, i.e., the set of all observations of all consistent executions of C.

A *bound* is a function $K\colon \mathsf{Method} \rightharpoonup \mathbb{N}$ that maps some methods to a number, indicating that at most that many invocations of the method are allowed. A client program $\mathcal{L} \vdash P$ *respects* a bound K if for all $\langle v, G \rangle \in [\![P]\!]$, and all $m \in \mathit{dom}(K)$, $G.\mathsf{E}$ contains at most $K(m)$ invocations of m. (G can contain any number of invocations to methods $m \notin \mathit{dom}(K)$.)

With these definitions, we can then define (bounded) contextual refinement in a standard fashion.

Definition 3.10. Let A be a library interface, I be an implementation of A and S be a specification of A. Then:

- *I refines S* for a client program C over A if $\mathsf{Obs}_A(C \triangleright I) \subseteq \mathsf{Obs}_A(C \triangleright S)$.
- *I refines S* up to a bound K if it refines S for all client programs C of A that respect the bound K.

4 Checking Library Implementation Correctness

We move on the problem of checking that a library implementation I of a library interface $A = \{m_1, \ldots, m_n\}$ refines a library specification S up to some bound K.

To simplify the presentation, we assume that all the methods of A do not take any arguments. Methods that take arguments from a finite domain can always be transformed to multiple methods with no arguments, while methods that draw their arguments from an infinite domain but use them in a (partially) data-independent fashion are handled explicitly in §4.5.

Relinche consists of two phases: the first analyzes the specification and the second checks conformance of the implementation. We start by defining the most parallel client (§4.1), and then describe the two phases of Relinche (§4.2, §4.3) and establish its correctness (§4.4).

4.1 Most Parallel Client

The key idea behind Relinche is that we can show that I refines S by employing the most parallel client of A with bound K.

Definition 4.1 (Most parallel client). The most parallel client of a library interface $A = \{m_1, \dots, m_n\}$ with bound $K : A \to \mathbb{N}$ is the program:

$$\mathsf{MPC}_K \triangleq \underbrace{m_1() \parallel \dots \parallel m_1()}_{K(m_1)\text{ times}} \parallel \underbrace{m_2() \parallel \dots \parallel m_2()}_{K(m_2)\text{ times}} \parallel \dots \parallel \underbrace{m_n() \parallel \dots \parallel m_n()}_{K(m_n)\text{ times}}$$

To see why we can employ MPC_K to check refinement, consider an arbitrary execution of a K-bounded client of A. By construction, its projection to the events of A will have at most $K(m_i)$ invocations of method m_i and $K(m_i)$ matching response events, and will be a (prefix) execution of MPC_K with some additional client happens-before edges. We can thus obtain all such projections by generating all executions of MPC_K and considering all consistent happens-before strengthenings of these executions.

In turn, given a consistent execution G of $\mathsf{MPC}_K \triangleright I$, we can check whether its A-observation is valid by comparing it with the A-observations of $\mathsf{MPC}_K \triangleright S$. If the observation is allowed, we further need to consider any strengthenings of MPC_K that render the observation invalid according to the specification. If no such strengthenings invalidate the observation, then we are done. If, however, there is some more constrained client C such that $C \triangleright S$ forbids the observation, we also need to make sure that $C \triangleright I$ also forbids the observation, or else we have found an error. Therefore for all the minimal hb-strengthenings of MPC_K that render the observation invalid according to the specification, we need to check that these hb-extensions also make the implementation execution inconsistent.

4.2 Phase 1: Specification Analysis

We next describe the first phase of our verification procedure: Relinche's *specification analysis* (Algorithm 1). This phase uses a model checker to enumerate all specification graphs arising from the most parallel client applied to the library specification, collects the set of allowed library observations, and for each observation generates a set of *happens-before extensions* for it.

This first phase takes as input only the library interface A, the bound K, and the specification code *Spec*. As such, it can run "offline", since there are typically only a few specifications of interest for each type of data structure.

In more detail, Relinche first generates all the executions of the most parallel client MPC_K applied to the specification, and groups them according to their observations. To that end, it initializes a map *Lins* (Line 2) that maps each observation to the set of linearizations of the atomic library methods resulting in this observation. The library observation is simply the projection of a given graph G to the events of the library and the associated synchronization order. Here, we take as the induced synchronization order of the specification the happens-before order of the specification code *without* any edges due to the partial lock library.

Then, for each library observation, Relinche calculates the set of minimal additional happens-before edges, which contradict all the linearizations generating that outcome. As such, it initializes (Line 2) and populates (Line 6) another map, *HBexts*, which maps each allowed observation to the set of all minimal hb-extensions that render the observation invalid.

Calculating such extensions is done via CalcExts. Given an hb-extension h, a set of edges $S \subseteq \mathsf{E} \times \mathsf{E}$, and a set $\mathbb{L}$ of linearizations, a call to $\textsc{CalcExts}(h, S, \mathbb{L})$ recursively calculates all possible minimal edge additions from S to h that render each graph in $\mathbb{L}$ invalid. Initially (at Line 6), CalcExts is called with $h = \emptyset$, S containing all possible edges from A-response events to A-invocation events,

Algorithm 1 RELINCHE: Specification analysis

```
1: function ANALYZESPEC($A, K, \mathit{Spec}$)
2:     local $(\mathit{Lins}, \mathit{HBexts}) \leftarrow (\emptyset, \emptyset)$
3:     for all $G \in \mathsf{Execs}(\mathsf{MPC}_K \triangleright \mathit{Spec})$ do        ▷ Collect specification executions
4:         $\mathit{Lins}[\langle G.\mathsf{E}|_A, G.\mathsf{hb}^{abs}|_A\rangle] \leftarrow \mathit{Lins}[\langle G.\mathsf{E}|_A, G.\mathsf{hb}^{abs}|_A\rangle] \cup \{G.\mathsf{hb}|_A\}$
5:     for all $O \in \mathit{dom}(\mathit{Lins})$ do        ▷ Compute happens-before extensions
6:         $\mathit{HBexts}[O] \leftarrow$ CALCEXTS$(\emptyset, O.\mathsf{E} \times O.\mathsf{E}, \bigcup\{\mathit{Lins}[O'] \mid O'.\mathsf{E} = O.\mathsf{E} \land O'.\mathsf{so} \subseteq O.\mathsf{so}\})$
7:     return $\mathit{HBexts}$
8: function CALCEXTS($h, S, \mathbb{L}$)        ▷ Return minimal extensions of $h$ invalidating $\mathbb{L}$
9:     if $\mathbb{L} = \emptyset$ then return $\{h\}$
10:    $S \leftarrow \{\langle e, e'\rangle \in S \mid \langle e, e'\rangle \notin h^+ \land \langle e', e\rangle \notin h^+\}$
11:    $S \leftarrow \{\langle e, e'\rangle \in S \mid \exists l \in \mathbb{L}.\ \langle e', e\rangle \in l\}$
12:    local $\mathit{Res} \leftarrow \emptyset$
13:    for all $\langle e, e'\rangle \in S$ do $\mathit{Res} \leftarrow \mathit{Res} \cup$ CALCEXTS$(h \cup \{\langle e, e'\rangle\}, S, \{l \in \mathbb{L} \mid \langle e', e\rangle \notin l\})$
14:    return $\mathit{Res}$
```

and $\mathbb{L}$ containing all recorded linearizations with weaker observations: the same set events and possible less induced synchronization.

Let us now see how CALCEXTS$(h, S, \mathbb{L})$ works in more detail. If $\mathbb{L}$ is empty, then CALCEXTS simply returns $\{h\}$ (Line 9). Otherwise, it first removes from S all edges that are either already included in h or contradict it (Line 10). It also removes from S all edges that do not eliminate any linearization in $\mathbb{L}$ (Line 11). Then, for each remaining edge in S, CALCEXTS calls itself recursively adding the edge to h and removing from $\mathbb{L}$ all linearizations eliminated by the edge (Line 13). Observe that the edge will be discounted in the recursive call, and that at each recursive call the set $\mathbb{L}$ decreases.

4.3 Phase 2: Implementation Conformance Checking

We move on to the second phase of RELINCHE, which checks for *implementation conformance* with respect to the computed set of allowed outcomes and their associated happens-before extensions (Algorithm 2).

Algorithm 2 RELINCHE: Implementation conformance

```
1: procedure CHECKCONFORMANCE($A, K, I, \mathit{HBexts}$)
2:     for all $G \in \mathsf{Execs}(\mathsf{MPC}_K \triangleright I)$ do        ▷ Analyze implementation executions
3:         $\mathbb{O} \leftarrow \{O \in \mathit{dom}(\mathit{HBexts}) \mid O.\mathsf{E} = G.\mathsf{E}|_A \land O.\mathsf{so} \subseteq G.\mathsf{hb}|_A\}$
4:         if $\mathbb{O} = \emptyset$ then REPORTERROR()
5:         $H \leftarrow \{\emptyset\}$        ▷ Calculate relevant happens-before extensions
6:         for all $O \in \mathbb{O}$ such that $\nexists O' \in \mathbb{O}.\ O \subset O'$ do
7:             $H \leftarrow \{h \cup h' \mid h \in H, h' \in \mathit{HBexts}[O]\}$
8:         if $\exists h \in H.$ CONSISTENT$(\langle G.\mathsf{E}, G.\mathsf{po} \cup h, G.\mathsf{so}\rangle)$ then REPORTERROR()
```

The conformance phase largely follows the description of §4.1. For each execution of the most parallel client applied to the library implementation, RELINCHE calculates in $\mathbb{O}$ the set of recorded specification observations that match the implementation execution, namely those that have the same library events and require one to induce less or equal synchronization than the implementation provides (Line 3). If this set is empty, RELINCHE reports a refinement violation (Line 4). Otherwise, Relinche looks up the set of recorded happens-before extensions that need to be considered. In the common cases, $\mathbb{O}$ is a singleton set, and so we can just read off the appropriate entry from *HBexts*.

In the general case, however, $\mathbb{O}$ could contain multiple elements, say $\{O_1, \dots, O_n\}$. If so, we need to pick extensions that violate all the observations in $\mathbb{O}$. A simple choice is to let $H = \{h_1 \cup \dots \cup h_n \mid h_1 \in \mathit{HBexts}[O_1], \dots, h_n \in \mathit{HBexts}[O_n]\}$, which is what the loop on Lines 6 and 7 achieves. The loop contains a small optimization: we can ignore any observation $O \in \mathbb{O}$ that is strictly weaker than some other observation $O' \in \mathbb{O}$ because the linearizations leading to O were taken into account when computing the hb-extensions of O' in the specification analysis phase.

Finally, for each happens-before extension in H, Relinche checks that it violates consistency. If any of them do not, it reports a refinement violation (Line 8).

Remark 4.2 (Employing Symmetry Reduction). We note that *symmetry reduction* [Clarke et al. 1996] can easily be applied to make conformance checking faster (by a factorial factor in the bound K). On Line 2, instead of returning all consistent executions of $\mathsf{MPC}_K \triangleright I$, it is sound to use an approach such as Spore [Kokologiannakis et al. 2024] to return all consistent executions up to symmetry. Moreover, if we use a fixed way of resolving symmetries (as, e.g., in the Spore algorithm), then we can also optimize the set of generated happens-before extensions to remove ones that would contradict the resolution of symmetries.

We can also employ *symmetry reduction* to speed up the specification analysis. We can treat observations up to symmetry by sorting the threads invoking the same method according to their return values. We can then only calculate hb-extensions for these equivalence classes, and make sure that the conformance checking phase sorts the threads of the G in the same way before checking for its relevant hb-extensions.

4.4 Correctness

We next state and prove the correctness of Relinche.

Theorem 4.3. *CheckConformance*$(A, K, I, \mathit{AnalyzeSpec}(A, K, S))$ *reports an error if and only if there is a library client C that respects K such that I does not refine S for C.*

Proof sketch. In the forward direction, if Relinche reports an error at Line 4, then the execution G witnesses a A-observation of $\mathsf{MPC}_K \triangleright I$ that is not allowed by $\mathsf{MPC}_K \triangleright S$. If it reports an error at Line 8, then consider the client C obtained by sequencing the method invocations in MPC_K following the edges of h. Then, $\langle G.\mathsf{E}, G.\mathsf{po} \cup h, G.\mathsf{so}\rangle$ witnesses a A-observation of $C \triangleright I$ that is not allowed by $C \triangleright S$.

In the backward direction, let G be an execution of $C \triangleright I$, whose A-observation O is not allowed by $C \triangleright S$. By definition, $\langle G.\mathsf{E}, \emptyset, G.\mathsf{so}\rangle$ is an execution of $\mathsf{MPC}_K \triangleright I$ with the same observation O. Since $C \triangleright S$ does not allow O, there has to be a minimal $h' \subseteq h$ such that such that $C' \triangleright S$ does not allow O, where C' is obtained by adding h' po-edges to MPC_K. If $h' = \emptyset$, then O is not allowed by $\mathsf{MPC}_K \triangleright I$ and so an error will be reported at Line 4. Otherwise, $\exists O' \in \mathit{dom}(\mathit{HBexts})$ and $h'' \in \mathit{HBexts}[O']$ such that $O' \subseteq O$ and $h'' \subseteq h'$, and so Relinche will report an error at Line 8. □

4.5 Handling Methods with Arguments Using Data Independence

We now extend the basic approach to handle methods with arguments drawn from an infinite domain, assuming that the library implementation uses these arguments in a data-independent fashion. We consider three degrees of data independence in increasing level of complexity.

Libraries with Full Data Independence. The library implementation treats the arguments to the various methods as completely abstract: it does not inspect nor perform any computation on the passed arguments. For a library implementation with full data independence and a multiset of method names $m_1, m_2, \ldots, m_N$ (i.e., a sorted sequence of method names that can contain duplicates), we construct the most parallel client as the program that calls each method in a separate thread with a fresh argument: $m_1(arg_1) \parallel \ldots \parallel m_N(arg_N)$ where $arg_j \triangleq j$ if M_j takes an argument and $\perp$ otherwise.

Equality-checking Library Implementations. Whenever the library implementation may check for equality between two arguments to its methods, we additionally need to consider executions that have multiple method invocations with the same argument. Starting with the most parallel client for the fully data-independent case, where each method is invoked with a different argument, we further consider all partitions of the arguments into equivalence classes. For each partitioning, we generate one most parallel client replacing each argument value with its equivalence class representative.

Ordering-checking Implementations. Some library implementation may also check for ordering ($<$) between method arguments, e.g., to insert an element into a binary tree. For such ordering-checking libraries, we additionally have to consider all orderings between the different equivalence classes. Thus, for every partitioning and for every total ordering among the partitions, we generate a most parallel client that invokes each method M_j that takes an argument on the index of the equivalence class of j in the total order.

5 Implementation

We have implemented Relinche as an extension of the open-source GenMC stateless model checker [Kokologiannakis and Vafeiadis 2021]. Our tool supports two modes of operation corresponding to the two phases of Relinche (Algorithms 1 and 2):

Specification analysis: The tool generates all specification graphs of a program (the most parallel client applied to the library specification), groups them according to outcome, calculates the relevant happens-before extensions, and stores the generated *HBexts* map into a file.

Conformance checking: The tool generates all implementation graphs of a program (the most parallel client applied to the library implementation), and checks them against the *HBexts* map, which is provided as an additional input.

Note that since the invoked library operations are determined by the client program, Relinche represents specification graphs in a very compressed fashion. Concretely, we only need to store: (1) the results of the library methods that return results, and (2) the induced synchronization edges between the library events. To enable quick access to all matching outcomes in *HBexts* modulo synchronization, we represent *HBexts* as a hashtable using as the key the method results (and not the induced synchronization edges).

6 Evaluation

In this section, we evaluate Relinche aiming to answer the following questions:

Q1: How well does Relinche scale (in terms of maximum MPC size)?

Q2: How do different specifications affect its performance?

Q3: Can Relinche verify library implementations that lie beyond the state of the art (even of manual verification techniques), or find linearizability bugs for which it may be difficult to devise suitable clients?

To answer these questions, we conducted three case studies on a set of challenging concurrent data structures.

To answer Q1, assuming that the specification analysis has been performed offline, we measure the maximum MPC size for which Relinche can establish bounded linearizability within a given time budget, for various benchmarks and specification types.

To answer Q2, we measure how the performance of Relinche's specification analysis is affected by the specification size and type. In line with §2, we use three different types of specifications:

R: a specification where the only synchronizing operation is reading a value,

RI: a specification where, in addition to (R), all insertions synchronize with each other, and

RID: a specification where, in addition to (RI), all deletions that return an element synchronize with each other.

To answer Q3, we evaluate Relinche on a set of concurrent data structures injected with bugs that are difficult to express as memory or assertion violations (i.e., they need elaborate clients to manifest).

Our experimental results suggest that

(1) Relinche is fast enough to be used in practice, and scales to MPCs that use up to 7–9 operations,
(2) Relinche is faster for stronger specifications than for weaker ones (as weaker specifications have more linearizations),
(3) Relinche is able to find subtle linearizability bugs for which constructing a suitable client is challenging.

Notably, Relinche was able to prove for the first time that well-known data structures that are linearizable under SC are not linearizable under weak memory, even if we use the strongest non-SC access modes, and even under the weakest specification (R).

Experimental Setup. We conducted all experiments on a Dell PowerEdge R6525 system running a custom Debian-based distribution with 2 AMD EPYC 7702 CPUs (256 cores @ 2.80 GHz) and 2TB of RAM. We set the timeout limit to 30 minutes for the conformance checking phase, and to 60 minutes for the specification analysis phase (denoted by 🕒 in tables). All times are in seconds.

6.1 Implementation Conformance

Starting with the implementation conformance phase, the time Relinche takes to verify (bounded) linearizability for various lock-free concurrent data structures can be seen in Figures 4 and 5. For all benchmarks, we used a "50-50" workload where $\lceil N/2 \rceil$ threads insert an item to the data structure, and $\lfloor N/2 \rfloor$ threads read an item[1], and all benchmarks were implemented using the weakest access modes for which we could prove refinement of the corresponding specifications. (We discuss benchmarks that are not linearizable under weak memory in §6.3.)

Starting with Fig. 4, an observation we can make right off the bat about is that the time Relinche needs to enumerate the executions of a given implementation does not vary among different specifications. This is of course expected since enumerating the executions of an implementation only depends on the characteristics of the implementation itself (and not on the specification checked).

Enumeration time aside, we can draw three major conclusions from Fig. 4.

First, the time spent in checking whether an implementation outcome refines the specification is typically much less than the time spent by the model checker enumerating executions. This is due to the fact that the cost per execution is typically much higher than the cost of checking one hb-extension, since enumerating a single execution involves running the program. As such,

[1] Relinche would scale much better with a "100-0" workload as it would leverage the full symmetry among threads, but such a workload is uninteresting for proving linearizability.

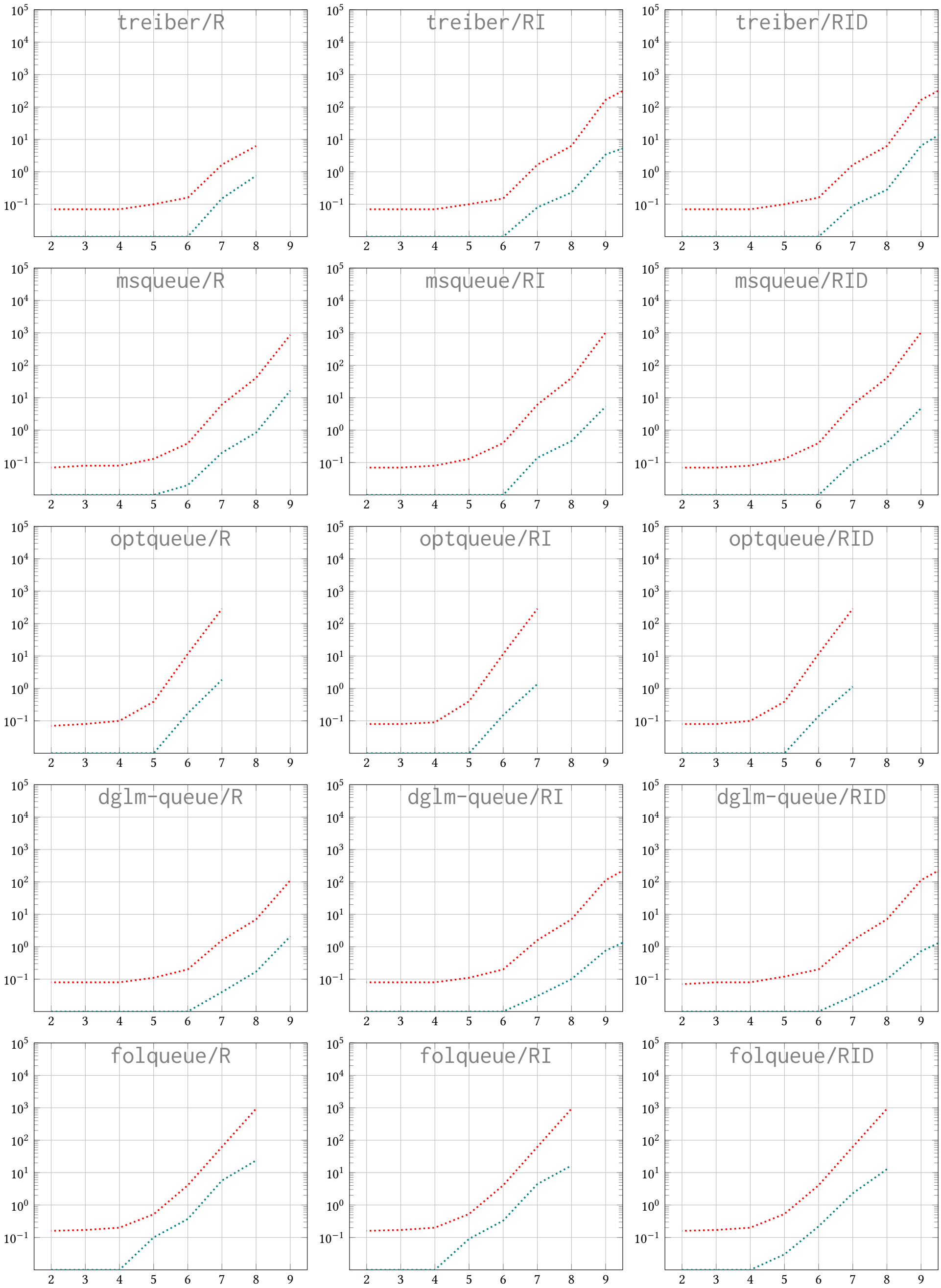


Fig. 4. Implementation conformance: enumeration/refinement time (Y-axis) per input parameter (X-axis)

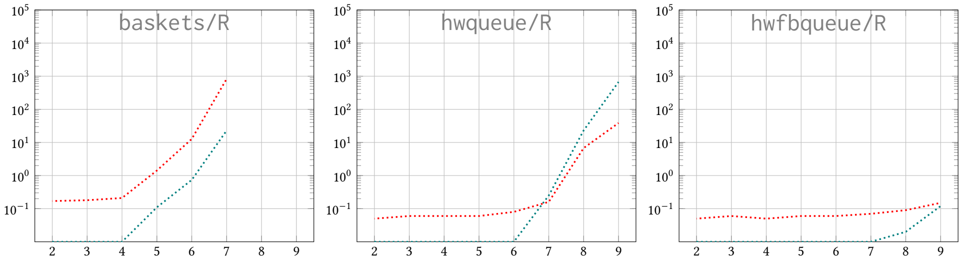


Fig. 5. Implementation conformance: benchmarks linearizable only under a single specification

the total time spent checking hb-extensions is less than the time spent enumerating executions, even in cases where the number of extensions is much greater than the number of executions: in `msqueue/RI(9)`, for instance, Relinche is able to check 358560 extensions in less time than what it takes it to enumerate 15120 executions.

Second, Relinche scales gracefully to 7–9 method invocations. The scalability bottleneck is largely the model checking time to enumerate all executions of the given benchmark, and we can see that this time varies from benchmark to benchmark. For `optqueue`, Relinche scales only to 7 threads, whereas for the other benchmarks it scales at least until 9 threads. The reason why scalability stops at 8 threads for the weakest specification type (R) is that the specification analysis phase for 9 threads timed out, and so we did not run the implementation conformance phase for 9 threads. (We could have used a larger timeout for specification analysis since it is performed offline, but, as we explain in §6.2, we preferred to set a similar timeout for both phases to see how well Relinche scales within a small time frame.)

Third, Relinche typically spends a bit more time checking refinement for weaker specifications than for stronger specifications. This is because weaker specifications can be invalidated in more ways than stronger specifications, and so specification analysis phase (Algorithm 1) generates more hb-extensions for weaker specification, which in turn makes refinement checking slower. As an example, Relinche checks 60234 hb-extensions for the (R) spec of `msqueue(8)`, as opposed to 31488 and 30384 extensions for the (RI) and (RID) versions of the same benchmark, respectively.

Moving on to Fig. 5, we see benchmarks that are linearizable under one specification only. Indeed, since these benchmarks are only "loosely" synchronized when using release-acquire accesses, such implementations do not satisfy the guarantees required by the stronger specifications (RI) and (RID). As with Fig. 4, we report scalability up to 8 threads because we do not have hb-extensions for 9 threads. Similar to `optqueue`, the `baskets` benchmark has a large state space and scales only to 7 threads. By contrast, the Herlihy-Wing queue (`hwqueue`) and the lock-free queue specification from §2 (`hwfbqueue`) have a much smaller state space and scale very well. Their scalability can be largely attributed to these benchmarks representing the queue as an array (and thus having very little sharing among the threads), which makes their model checking very efficient.

6.2 Specification Analysis

Moving on to the specification analysis phase, the time Relinche needs to analyze the respective queue, stack and set specifications can be seen in Fig. 6. The workload used for analyzing specifications is the same as the one of §6.1.

Similarly to the implementation conformance phase, the enumeration time does not vary among different specifications and grows exponentially with the number of method invocations, and

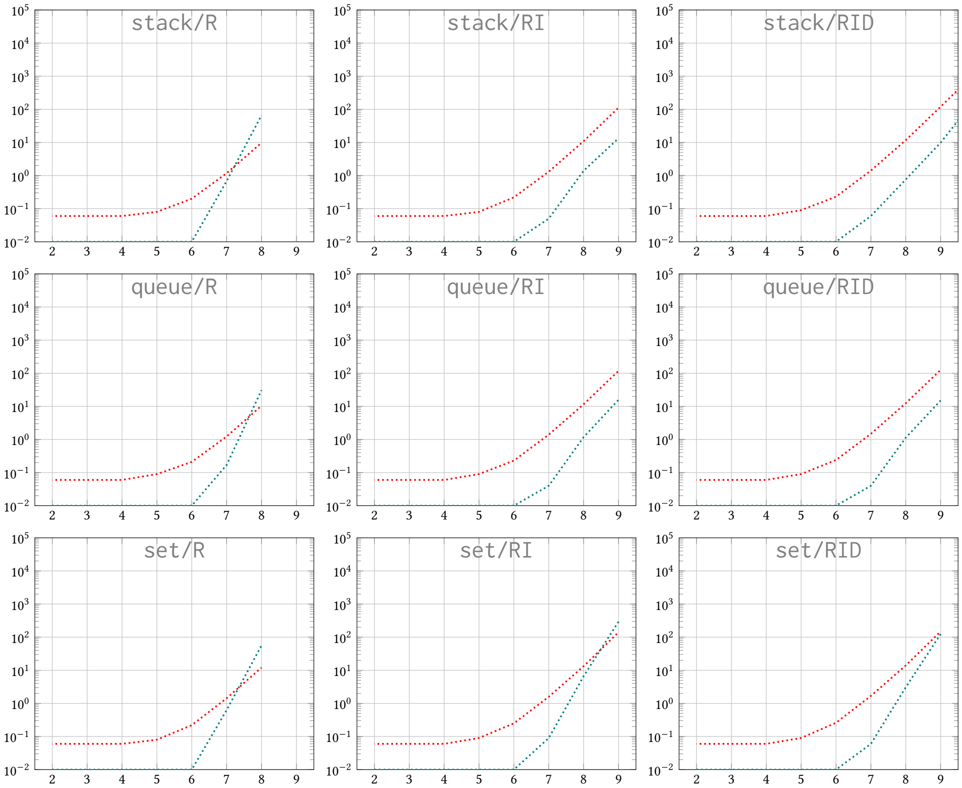


Fig. 6. Specification analysis: enumeration/analysis time (Y-axis) per input parameter (X-axis)

Relinche needs more time to analyze weaker specifications because such specifications generate more happens-before extensions.

In contrast to the implementation conformance phase, however, the time Relinche needs to analyze a specification can be greater than the time it needs to enumerate its executions. The reason for that is that generating hb-extensions has exponential complexity. A naive calculation requires $O(2^{N})$ steps (where N is the maximum size) and, even though `CalcExts` does drastically reduce this number, the generation still consumes a significant portion of Relinche's runtime. This number could further be reduced by taking symmetry into account, but our current implementation does not do so. That said, we do not consider this phase to be a bottleneck, as it is typically performed offline, and only once for each specification.

6.3 Non-linearizable Data Structures

So far, we have only discussed how Relinche can be used to establish bounded linearizability. During our development, however, we also found that Relinche is quite useful in detecting subtle linearizability violations.

One such violation concerned `timestamped-stack` [Dodds et al. 2015]. `timestamped-stack` is a lock-free data structure that is based on the observation that enforcing a total ordering on the memory layout of a data

	MPC size	*Conf. check time*
`timestamped-stack/R`	5	0.22
`lf-lazy-list/R`	5	2.2
`harris-list/R`	5	109
`libcds-michael/R`	5	42
`baskets-queue/RI`	4	0.1
`hwqueue/R`	7	0.09
`hwfbqueue-buggy`	4	0.07

Fig. 7. Linearizability violations found

structure is inefficient and unnecessary, and a partial ordering on the layout is sufficient to establish linearizability. While this is indeed the case under SC, we demonstrate `timestamped-stack` implemented solely using non-SC accesses is not linearizable, even under our weakest specification (R). The counterexample produced by Relinche is quite complex, as it involves an MPC with 5 operations.

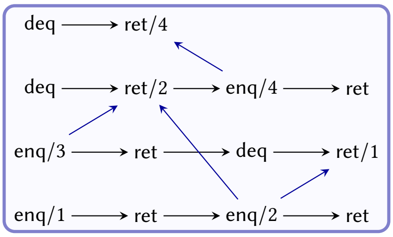
In total, we were able to show that 6 well-known data structures [Dodds et al. 2015; Harris 2001; Herlihy and Shavit 2008; Herlihy and Wing 1990; Hoffman et al. 2007; Michael 2002] are not linearizable when using non-SC atomics; our results are summarized in Fig. 7. The table shows the weakest specification and the minimum MPC size for which a violation was found. The time reported is the time that Relinche's conformance phase (both enumeration and refinement time) required to expose the violation. Note that the tests do not employ relaxed accesses at all; apart from `hwqueue` (which requires a weakening of an exchange to "acquire" for the violation to manifest), all benchmarks have been implemented using the strongest non-SC access modes everywhere.

The counterexamples reported by Relinche are quite interesting. For Herlihy-Wing, Relinche was able to find a (minimal) linearizability counterexample that is smaller than the previously proposed one by Raad et al. [2019]. The counterexample can be seen below.

Example 6.1 (Non-linearizability in Herlihy-Wing) Recall from §2 that the Herlihy-Wing queue is not linearizable under weak memory if we use an "acquire" exchange in the dequeue method instead of an "acquire-release" exchange. The counterexample below is obtained by analyzing the most parallel client with 4 enqueues and 3 dequeues applied to the Herlihy-Wing queue: we group the method calls to threads following the reported happens-before extension that witnesses the refinement violation.

$$\begin{array}{l} q.\mathrm{enqueue}(1)\\ q.\mathrm{enqueue}(2)\end{array} \;\Big\|\; \begin{array}{l} q.\mathrm{enqueue}(3)\\ q.\mathrm{dequeue}()\ \textit{// returns 1}\end{array} \;\Big\|\; \begin{array}{l} q.\mathrm{dequeue}()\ \textit{// returns 2}\\ q.\mathrm{enqueue}(4)\end{array} \;\Big\|\; q.\mathrm{dequeue}()\ \textit{// returns 4}$$

The projection graph of a non-linearizable behavior with 7 operations can be seen on the right. As was the case with the counterexample in §2, it is impossible to totally order the method invocations in accordance with the synchronization obtained by the implementation. The hb obtained by the implementation methods implies that the enqueues are ordered as enqueue (1) → enqueue (2) → enqueue (4) → enqueue (3): the first pair due to po, the second pair due to T3 dequeueing 2 before enqueueing 4, and the third pair due to 3 not being dequeued. The corresponding dequeue order that respects FIFO (dequeue (1) → dequeue (2) → dequeue (4)) creates a cycle, since dequeue (1) → dequeue (2), and enqueue (4) → enqueue (3).



More interestingly, Relinche also finds counterexamples for various lock-free list implementations, thereby indicating that release-acquire accesses are insufficient for implementing linearizable versions of these data structures. Relinche's counterexample for `lf-lazy-list` can be seen below (the counterexamples for others are similar).

Example 6.2 (Non-linearizability in lock-free lazy list) The counterexample consists of three threads and has a similar flavor to SB: T1 and T3 both attempt to remove an item after inserting one, and both are unsuccessful.

$$\begin{array}{l||c||l} \mathsf{insert}(1) & & \mathsf{insert}(3) \\ & \mathsf{insert}(2) & \\ \mathsf{remove}(3)\ /\!/\ \textit{returns } \bot & & \mathsf{remove}(1)\ /\!/\ \textit{returns } \bot \end{array}$$

Intuitively, this behavior can arise because operations involving different elements 1 operate on different parts of the list and so their lock acquisitions and their updates do not synchronize. Without SC accesses and fences, the program naturally exhibits "store buffering" behaviors.

We conclude our evaluation with some notes regarding the violation found for `libcds-michael`. The code for this benchmark was ported from the popular C++ concurrency library `libcds` [Khizhinsky n.d.], which in turns implements Michael's list [Michael 2002] using weak atomics.

Relinche was able to find two different errors for that implementation. The first error manifested as a memory issue (access to uninitialized, dynamically allocated memory), which was a result of a compare-and-exchange (CAS) instruction being performed with an "acquire" mode (instead of an "acquire/release" mode). This CAS was used to "publish" certain changes performed to the data structure, but an "acquire" mode does not provide synchronization guarantees to threads reading from the CAS, leading to consumers not seeing the results being published.

The second error manifested after strengthening the implementation's access modes. Similarly to `lf-lazy-list`, release-acquire synchronization was insufficient to guarantee linearizability, and Relinche was able to find a minimal counterexample involving 5 threads[2].

All in all, this last example demonstrates the usefulness of tools like Relinche: even though some linearizability bugs readily manifest as memory/assertion violations (and therefore can be detected by model checking tools by having a simple client calling the library methods without any additional assertions), others are much more subtle, and require the use of more sophisticated techniques to be detected.

7 Related Work

We classify related work into three categories: (1) library correctness notions for weak memory consistency, (2) verification works for linearizability, and (3) model checking for weak memory models.

Relaxed Linearizability Definitions. Various extensions of linearizability [Herlihy and Wing 1990] have been proposed: some of them directly handle specific memory models like TSO [Owens et al. 2009] and (R)C11 [Batty et al. 2011; Lahav et al. 2017], while others apply to a range of models. In the first category, Burckhardt et al. [2012] develop a notion of linearizability for TSO, by extending traces with markers denoting notions specific to TSO (namely, the begin and end of store-buffer flushing). Batty et al. [2013] propose a notion for library abstraction under C/C++11 that generalizes linearizability based on atomic blocks that have transactional semantics. Singh and Lahav [2023] develop an operational account of RC11 along with a correctness criterion for concurrent libraries. From this work, we adopt the idea of "partial locks" as a means to write atomic library specifications in a weak memory setting.

In the second category, most papers follow a declarative semantics. Dongol et al. [2018] and Emmi and Enea [2019b] propose relaxations of linearizability that use total orders to specify valid call sequences of library methods, while Raad et al. [2019] present a more general framework that allows for even weaker specifications, where such total orders might not even exist. Our model of concurrent libraries largely follows that of Raad et al. [2019] with a key difference/simplication: we

[2] The counterexample is the same as for `lf-lazy-list`.

split each method call event into two events—one for the invocation and one for the response—which enables us to avoid `rf`-cycles in execution graphs, and streamlines the development of the refinements.

Linearizability Verification Approaches. There are many different approaches to verifying linearizability in the literature. On the one end of the spectrum, there are several manual techniques based on *program logics*, such as [Khyzha et al. 2016; Liang and Feng 2013], which require labor-intensive proof efforts and are prone to human error. Some of the more recent works encode their proofs in a proof assistant thereby ruling out the possibility of error in the proofs, but significantly increasing the cost of carrying out those proofs.

On the other end of the spectrum, there are *linearizability checkers*, such as [Koval et al. 2023; Lowe 2017; Ou and Demsky 2017], that can check particular executions for linearizability but require the user to write an appropriate client invoking the library methods and do not provide any coverage guarantees, and tools like `Violat` [Emmi and Enea 2019a] that generate tests exposing ADT-violations.

Somewhere in the middle lie heuristic techniques for proving linearizability either fully automatically [Vafeiadis 2010] or with some user input to annotate the linearization points of methods [Amit et al. 2007; Vafeiadis 2009]. These techniques are based on a "shape analysis" (a type of static analysis that describes dynamically allocated data structures with pointers). They typically apply only to SC, and cannot handle implementations such as the Herlihy-Wing queue [Herlihy and Wing 1990], which have non-fixed linearization points.

Later works have extended the applicability of these techniques to handle some common cases of non-fixed linearization points. For example, O'Hearn et al. [2010] introduce a "hindsight" lemma for verifying optimistic traversals of linked list data structures, Feldman et al. [2020] present a more general approach for verifying optimistic traversals of search data structures, while Dragoi et al. [2013] reduce cooperating updates in algorithms like the elimination stack of Hendler et al. [2004] by rewriting the program. Other works develop verification techniques for specific data structures. For example, Henzinger et al. [2013] show that a concurrent queue is linearizable if and only if it does not contain certain patterns of violations, which can be checked with standard verification tools. Abdulla et al. [2013] extend this result to stacks.

On the more automated side, `CheckFence` [Burckhardt et al. 2007] uses a SAT solver to verify linearizability of concurrent libraries under a very weak relaxed memory model for a specific concurrent client program. This approach provides completeness guarantees albeit only for the provided client program. Thus, to provide confidence for the correctness of a library, the user has to devise multiple client programs. which is by no means an easy task. In fact, although the authors provide multiple such clients, none would be sufficient to expose the bug of the Herlihy-Wing queue discussed earlier. In a precursor to `CheckFence`, Burckhardt et al. [2006] develop a similar tool that additionally requires linearization point annotations.

`Line-Up` [Burckhardt et al. 2010] verifies linearizability for all clients of a bounded size by enumerating all such clients and calling a stateless model checker for carrying out the verification. This simpler approach, sadly, does not scale and so the authors resolve to random sampling of executions instead of exhaustive verification.

Model Checking for Weak Memory Models. As far as (bounded) model checking for weak memory consistency is concerned, we can broadly classify existing approaches into enumerative (e.g., [Abdulla et al. 2015; Bui et al. 2021; Burckhardt and Musuvathi 2008; Chalupa et al. 2017; Kokologiannakis et al. 2017, 2019; Norris and Demsky 2013]), SAT-based (e.g., [Clarke et al. 2004; Gavrilenko et al. 2019]), and combinations thereof (e.g., [Demsky and Lam 2015]). Most approaches in either category only support one or a few closely related memory models; the only tools that are parametric in the choice of the memory model are GenMC [Kokologiannakis and Vafeiadis 2021] and Dartagnan [Gavrilenko et al. 2019]. None of these tools, however, supports the verification of safety properties across all possible program clients, like linearizability.

There has been a lot of work in optimizing the model checking algorithms in the last few years. In particular, the most recent version of GenMC that we call from Relinche incorporates optimizations from multiple works, e.g., to ensure polynomial space complexity [Kokologiannakis et al. 2022], to reduce the number of blocked executions [Kokologiannakis et al. 2023], and to support symmetry reduction [Kokologiannakis et al. 2024], which translate to better scalability of the model checking. To a large extent, the scalability of Relinche can be attributed to these works.

8 Conclusion

We have presented the first fully automated technique for verifying lineariability of concurrent libraries under weak memory models for all clients with a bounded number of library invocations, and have demonstrated that our Relinche implementation scales well enough to verify standard concurrent data structures with 7–9 operations.

We note that Relinche can also be used in combination with randomized testing or context-bounded model checking to test implementations, whose full verification is infeasible, by checking for refinement of a subset of the executions of the most parallel client program applied to the library implementation. We leave the exploration of the tradeoffs between soundness and scalability as future work. Similarly, Relinche could be used as the verification component of higher-level tools like VSync [Oberhauser et al. 2021] that relax barriers in a sound way to potentially increase performance.

A second item for future work is to consider the automated verification of concurrent libraries containing methods that can only be invoked in certain contexts, and so our most parallel client does not constitute a valid client. For example, the client thread of a locking library may invoke the unlock operation only if it holds the lock, while a client thread of a transactional library may invoke transactional accesses only within a scope of a transaction.

Finally, Relinche can also be used to inform users of overly strong specifications in cases where a library can have two outcomes with the same results of the operations but with different synchronization. The extent to which such warnings would be useful remains to be seen.

Acknowledgments

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

Data-Availability Statement

The benchmarks and tools used to produce the results of this paper can be found at [Golovin et al. 2024].

References

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

Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt Jonsson, and Ahmed Rezine. 2013. “An integrated specification and verification technique for highly concurrent data structures.” In: *TACAS 2013* (LNCS). Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Springer, 324–338.

Daphna Amit, Noam Rinetzky, Tom Reps, Mooly Sagiv, and Eran Yahav. 2007. "Comparison under abstraction for verifying linearizability." In: *CAV 2007* (LNCS). Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49.

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

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. "Library abstraction for C/C++ concurrency." In: *POPL 2013.* ACM, 235–248. https://doi.org/10.1145/2429069.2429099.

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

Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. Oct. 2021. "The Reads-from Equivalence for the TSO and PSO Memory Models." *Proc. ACM Program. Lang.*, 5, OOPSLA, (Oct. 2021). https://doi.org/10.1145/3485541.

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2006. "Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study." In: *CAV 2006* (LNCS). Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Springer, 489–502. https://doi.org/10.1007/11817963_45.

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. "CheckFence: Checking consistency of concurrent data types on relaxed memory models." In: *PLDI 2007.* Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/1250734.1250737.

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. "Line-Up: A complete and automatic linearizability checker." In: *PLDI 2010.* Ed. by Benjamin G. Zorn and Alexander Aiken. ACM, 330–340. https://doi.org/10.1145/1806596.1806634.

Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. "Concurrent Library Correctness on the TSO Memory Model." In: *ESOP 2012* (LNCS). Ed. by Helmut Seidl. Vol. 7211. Springer, 87–107. https://doi.org/10.1007/978-3-642-28869-2_5.

Sebastian Burckhardt and Madanlal Musuvathi. 2008. "Effective Program Verification for Relaxed Memory Models." In: *CAV 2008* (LNCS). Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. Springer, 107–120. https://doi.org/10.1007/978-3-540-70545-1_12.

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

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

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

Brian Demsky and Patrick Lam. 2015. "SATCheck: SAT-directed stateless model checking for SC and TSO." In: *OOPSLA 2015.* ACM, Pittsburgh, PA, USA, 20–36. https://doi.org/10.1145/2814270.2814297.

Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. "User-Level Implementations of Read-Copy Update." *IEEE Trans. Parallel Distrib. Syst.*, 23, 2, 375–382. https://doi.org/10.1109/TPDS.2011.159.

Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. "A Scalable, Correct Time-Stamped Stack." In: *POPL 2015.* ACM, Mumbai, India, 233–246. ISBN: 9781450333009. https://doi.org/10.1145/2676726.2676963.

Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. "On abstraction and compositionality for weak-memory linearisability." In: *VMCAI 2018* (LNCS). Ed. by Isil Dillig and Jens Palsberg. Vol. 10747. Springer, 183–204. https://doi.org/10.1007/978-3-319-73721-8_9.

Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. 2013. "Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates." In: *CAV 2013* (LNCS). Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Springer, 174–190.

Michael Emmi and Constantin Enea. 2019a. "Violat: Generating Tests of Observational Refinement for Concurrent Objects." In: *CAV 2019* (LNCS). Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Springer, 534–546. https://doi.org/10.1007/978-3-030-25543-5_30.

Michael Emmi and Constantin Enea. 2019b. "Weak-consistency specification via visibility relaxation." *Proc. ACM Program. Lang.*, 3, POPL, 60:1–60:28. https://doi.org/10.1145/3290373.

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. "Proving highly-concurrent traversals correct." *Proc. ACM Program. Lang.*, 4, OOPSLA, 128:1–128:29. https://doi.org/10.1145/3428196.

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

Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. Oct. 2024. *RELINCHE: Automatically Checking Linearizability under Relaxed Memory Consistency (Replication Package)*. (Oct. 2024). https://doi.org/10.5281/zenodo.13992580.

Timothy L. Harris. 2001. "A Pragmatic Implementation of Non-blocking Linked-Lists." In: *DISC 2001*. Springer-Verlag, Berlin, Heidelberg, 300–314. ISBN: 3540426051. https://doi.org/10.5555/645958.676105.

Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. "A Scalable Lock-Free Stack Algorithm." In: *SPAA 2004*. ACM, Barcelona, Spain, 206–215. https://doi.org/10.1145/1007912.1007944.

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. "Aspect-Oriented Linearizability Proofs." In: *CONCUR 2013* (LNCS). Ed. by Pedro R. D'Argenio and Hernán C. Melgratti. Vol. 8052. Springer, 242–256.

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

Maurice Herlihy and Jeannette M. Wing. 1990. "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Program. Lang. Syst.*, 12, 3, 463–492. https://doi.org/10.1145/78969.78972.

Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. "The Baskets Queue." In: *OPODIS 2007*. Springer Berlin Heidelberg, Berlin, Heidelberg, 401–414. https://doi.org/10.1007/978-3-540-77096-1_29.

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

Artem Khyzha, Alexey Gotsman, and Matthew J. Parkinson. 2016. "A Generic Logic for Proving Linearizability." In: *FM 2016* (LNCS). Ed. by John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou. Vol. 9995. Springer, 426–443. https://doi.org/10.1007/978-3-319-48989-6_26.

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

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

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

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

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

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. "GenMC: A model checker for weak memory models." In: *CAV 2021* (LNCS). Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Springer, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20.

Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. 2023. "Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM." In: *CAV 2023*. Ed. by Constantin Enea and Akash Lal. Springer, Cham, 156–169. https://doi.org/10.1007/978-3-031-37706-8_8.

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

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

Doug Lea. 2005. "The java.util.concurrent synchronizer framework." *Sci. Comput. Program.*, 58, 3, 293–309. Special Issue on Concurrency and synchonization in Java programs. https://doi.org/10.1016/j.scico.2005.03.007.

Hongjin Liang and Xinyu Feng. 2013. "Modular verification of linearizability with non-fixed linearization points." In: *PLDI 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 459–470. https://doi.org/10.1145/2491956.2462189.

Gavin Lowe. 2017. "Testing for linearizability." *Concurr. Comput. Pract. Exp.*, 29, 4. https://doi.org/10.1002/CPE.3928.

Maged M. Michael. June 2004. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects." *IEEE Trans. Parallel Distrib. Syst.*, 15, 6, (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8.

Maged M. Michael. 2002. "High performance dynamic lock-free hash tables and list-based sets." In: *SPAA 2002*. ACM, Winnipeg, Manitoba, Canada, 73–82. ISBN: 1581135297. https://doi.org/10.1145/564870.564881.

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

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. "Verifying linearizability with hindsight." In: *PODC 2010*. Ed. by Andréa W. Richa and Rachid Guerraoui. ACM, 85–94.

Jonas Oberhauser et al.. 2021. "VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models." In: *ASPLOS 2021*. ACM, Virtual, USA, 530–545. https://doi.org/10.1145/3445814.3446748.

Peizhao Ou and Brian Demsky. 2017. “Checking Concurrent Data Structures Under the C/C++11 Memory Model.” In: *PPoPP 2017*. ACM, 45–59. https://doi.org/10.1145/3018743.3018749.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. “A better x86 memory model: x86-TSO.” In: *TPHOLs 2009*. Springer, Munich, Germany, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27.

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, 68:1–68:31. https://doi.org/10.1145/3290381.

James Reinders. 2007. *Intel Threading Building Blocks*.

Abhishek Kr Singh and Ori Lahav. Jan. 2023. “An Operational Approach to Library Abstraction under Relaxed Memory Concurrency.” *Proc. ACM Program. Lang.*, 7, POPL, (Jan. 2023). https://doi.org/10.1145/3571246.

Viktor Vafeiadis. 2010. “Automatically Proving Linearizability.” In: *CAV 2010* (LNCS). Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. Springer, 450–464.

Viktor Vafeiadis. 2009. “Shape-Value Abstraction for Verifying Linearizability.” In: *VMCAI 2009* (LNCS). Ed. by Neil D. Jones and Markus Müller-Olm. Vol. 5403. Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27.

Received 2024-07-11; accepted 2024-11-07