On Library Correctness under Weak Memory Consistency

Specifying and Verifying Concurrent Libraries under Declarative Consistency Models

AZALEA RAAD, MPI-SWS, Germany
MARKO DOKO, MPI-SWS, Germany
LOVRO ROŽIĆ, MPI-SWS, Germany
ORI LAHAV, Tel Aviv University, Israel
VIKTOR VAFEIADIS, MPI-SWS, Germany

Concurrent libraries are the building blocks for concurrency. They encompass a range of abstractions (e.g. locks, exchangers, stacks, queues, sets) built in a layered fashion: more advanced libraries are built out of simpler ones. While there has been a lot of work on verifying such libraries in a sequentially consistent (SC) environment, little is known about how to specify and verify them under weak memory consistency (WMC).

We propose a general declarative framework that allows us to specify concurrent libraries declaratively, and to verify library implementations against their specifications compositionally. Our framework is sufficient to encode standard models such as SC, (R)C11 and TSO. Additionally, we specify several concurrent libraries, including mutual exclusion locks, reader-writer locks, exchangers, queues, stacks and sets. We then use our framework to verify multiple weakly consistent implementations of locks, exchangers, queues and stacks.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Theory of computation → Semantics and reasoning**; **Concurrency**;

Additional Key Words and Phrases: Weak memory consistency, concurrent libraries, linearisability

ACM Reference Format:
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, Article 68 (January 2019), 31 pages. https://doi.org/10.1145/3290381

1 INTRODUCTION

Large software systems are typically structured as layers of abstractions, where higher-level abstractions are constructed using lower-level ones. This layered approach is also prevalent in concurrent programs, whose abstraction layers are *concurrent libraries*. At the lowest level are the atomic operations such as reads, writes and compare-and-swaps (CAS). These are used to build synchronisation primitives (e.g. locks); synchronisation primitives are used to build concurrent containers (e.g. queues); containers are then used to implement higher-level algorithms (e.g. concurrent graph traversal), which may be a component of the concurrent program.

For better scalability, concurrent systems are often verified *compositionally*: each constituent library of the system is specified separately, and each library implementation is verified against its specification. This approach has been studied extensively in the context of interleaving concurency—a.k.a. *sequential consistency* (SC) [Lamport 1979]. The existing work includes the correctness criteria

Authors' addresses: Azalea Raad, MPI-SWS, Saarland Informatics Campus, Germany; Marko Doko, MPI-SWS, Saarland Informatics Campus, Germany; Lovro Rožić, MPI-SWS, Saarland Informatics Campus, Germany; Ori Lahav, Tel Aviv University, Israel; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany.

2475-1421/2019/1-ART68
https://doi.org/10.1145/3290381

for libraries (most notably *linearisability* [Herlihy and Wing 1990], and many variants thereof [Castañeda et al. 2015; Hemed et al. 2015; Neiger 1994; Sergey et al. 2015]), program logics [Dinsdale-Young et al. 2010; Krebbers et al. 2017; Nanevski et al. 2014; Raad et al. 2015], and automated tools for checking or proving library correctness [Bouajjani et al. 2017; Vafeiadis 2010; Zhu et al. 2015].

Unfortunately, however, most of this work is detached from practice, where *weak memory consistency* (WMC) has become the de facto paradigm for shared-memory concurrency. The semantics of primitive atomic operations is governed by a weak memory model both at the hardware architecture level [Alglave et al. 2014; Owens et al. 2009; Pulte et al. 2018] and the programming language level [Alglave et al. 2018; Batty et al. 2011; Lahav et al. 2017; Manson et al. 2005], which allows behaviours (e.g. "store buffering") disallowed by SC. Similar weak behaviours are typically exposed at higher-level abstractions. For instance, consider a concurrent queue library with methods *enq* and *try-deq*, and a concurrent stack library with methods *push* and *try-pop*, where *try-deq* (resp. *try-pop*) returns *empty* if the queue (resp. stack) is empty. Efficient implementations exhibit the following weak behaviour on a queue at location q and a stack at s:

$$\begin{array}{l|\!|l} e : enq(q, 1); & a : push(s, 2); \\ r : a = try\text{-}pop(s) \;/\!/ \textit{returns empty} & d : b = try\text{-}deq(q) \;/\!/ \textit{returns empty} \end{array} \qquad \text{(SB-lib)}$$

Although it is possible for individual libraries to introduce sufficient memory fences to prevent such weak behaviours, they typically eschew this for better performance. We thus seek to *specify and verify* concurrent libraries in a general fashion, agnostic to the underlying memory model (SC or WMC). To this end, in §4 we propose a unifying general framework that allows us (1) to *specify* concurrent libraries *declaratively*, in the existing style of declarative models (e.g. RC11 [Lahav et al. 2017]); and (2) to *verify* library implementations against their specifications *compositionally*. In our framework the underlying memory model is simply a concurrent library. As such, our framework allows us to encode language-level memory models simply as concurrent libraries. In particular, we can encode RC11 [Lahav et al. 2017], TSO [Owens et al. 2009], and SC [Lamport 1979], as well as all other memory models that do not allow the load buffering behaviour, and thus, do not suffer from the "out-of-thin-air" problem [Boehm and Demsky 2014; Vafeiadis and Narayan 2013], known to "confound compositionality" [Batty et al. 2013].

To demonstrate the generality of our framework for library specification, in §5 we specify several concurrent libraries, including C11-style atomic memory accesses, mutual exclusion locks, reader-writer locks, exchangers, queues, stacks and sets. In several cases (e.g. queues and exchangers), we demonstrate that existing linearisability-style approaches are not suitable for specification under WMC. For a few libraries, we present multiple specifications and relate them to one another.

Later in §6, we showcase the application of our framework for compositional verification of library implementations. In particular, we verify the correctness of two WMC variants of the Herlihy-Wing queue implementation [Herlihy and Wing 1990], originally presented to demonstrate that statically determined linearisation points are insufficient for verifying linearisability. We then verify an implementation of the exchanger library, which is known to lack a linearisability-style specification [Hemed et al. 2015]. To illustrate the compositionality of our approach, we verify the correctness of an elimination stack implementation, represented as an internal stack together with an array of exchangers. We verify further implementations of mutual exclusion locks, reader-writer locks and queues in the technical appendix [Raad et al. 2018]. Our framework and several simple verification proofs are mechanised in Coq, and are available as auxiliary material [Raad et al. 2018].

Outline. In §2 we present our programming language and provide an overview of our contributions. In §3 we describe the semantics of our language. In §4 we present our framework. In §5 we specify several libraries. In §6 we verify several implementations. In §7 we discuss related work.

Expressions

$$\begin{aligned}\text{Exp} \ni \mathtt{e} ::= {} & v \mid x \mid m(x_1, \ldots, x_n) \\ & \mid \mathtt{let}\ x = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \\ & \mid \mathtt{if}\ x\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \mid \mathtt{e}_1 \| \mathtt{e}_2 \\ & \mid \mathtt{loop}\ \mathtt{e} \mid \mathtt{break}_n\ x\end{aligned}$$

Basic domains

$n \in \mathbb{N}$	Natural numbers
$v \in \text{Val}$	Values
$x \in \text{Var}$	Variables
$m \in \text{Method}$	Method names

Fig. 1. Our ANF expression language (left); and the basic domains used (right)

2 OVERVIEW OF MAIN IDEAS

Programming Language. To keep our presentation concrete and concise, we employ a simple first-order concurrent programming language of expressions in administrative normal form (ANF) [Sabry and Felleisen 1993], as presented in Fig. 1. We assume countably infinite sets Val of values with $\mathbb{N} \cup \{\bot\} \subseteq \text{Val}$; Var of program variables; and Method of method names. We use n and its variants (e.g. n_1, n') as metavariables for natural numbers; v and its variants for values; x, y, z and their variants for variables; and m and its variants for method names. Expressions contain the standard constructs of integer values, variables, method calls, let-bindings (sequential compositions), conditionals, and parallel compositions. Methods include standard arithmetic operators and user-defined library methods. Our language additionally includes the infinite loop construct, `loop e`, executing `e` ad infinitum; and the $\mathtt{break}_n\ x$ construct, which exits n levels of nested `loop` blocks and returns x. We refer to n as the *break number*. These somewhat unusual looping constructs are also present in CompCert Cminor [Leroy 2009], and can be used to encode the conventional `while`, `for` and `repeat-until` loops.

As is standard, we do not always follow the ANF constraints in our examples and write e.g. $m(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ for $\mathtt{let}\ x_1 = \mathtt{e}_1\ \mathtt{in}\ \ldots\ \mathtt{let}\ x_n = \mathtt{e}_n\ \mathtt{in}\ m(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are assumed to be fresh. We write $\mathtt{e}_1; \mathtt{e}_2$ for $\mathtt{let}\ x = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$ for a fresh variable x.

Sequential Specifications under WMC. A common approach for specifying the behaviour of *concurrent* libraries is to first specify their behaviour in a *sequential* setting, and then extend it to concurrency. Concretely, the sequential specification of a library can be defined as the set of method call sequences it accepts. For instance, the sequential specification of a queue would contain the sequence $\mathsf{enq}(q, 1); \mathsf{deq}(q, 1)$, but not $\mathsf{enq}(q, 1); \mathsf{deq}(q, 2)$.

In a concurrent setting, the method calls of different threads may not be ordered with respect to one another. A concurrent execution is thus represented as a partially ordered set $G{=}\langle E, \mathsf{hb}\rangle$, where E denotes the set of method calls and hb denotes the *happens-before* relation, a partial order on E. As such, given a sequential specification of library L, to describe the L behaviour *concurrently*, we can require that the method calls in G can be totally ordered to form a sequence that is allowed by the sequential specification. That is, for each execution $G{=}\langle E, \mathsf{hb}\rangle$, *there exists* a strict total order to on E that agrees with hb ($\mathsf{hb} \subseteq \mathsf{to}$) and meets the sequential specification.

This generic lifting of sequential specifications yields concurrent specifications akin to those in the *linearisability* literature [Herlihy and Wing 1990], and can be used to specify a number of libraries (e.g. queues and stacks). However, linearisability-style specifications are often too restrictive due to three main limitations. First, as Hemed et al. [2015] demonstrate, several concurrent libraries such as *exchangers* (in `java.util.concurrent`) do not have a sequential specification (we elaborate on this shortly) and thus one cannot build a concurrent specification by lifting the sequential one.

Linearisability was initially introduced in the context of *sequential consistency* (SC). As such, as we describe in the next two limitations, it is not always suitable in *weak memory concurrency* (WMC) settings. Second, the existentially quantified total order to is often not present in WMC

```
new-queue() ≜
  let q = alloc(+∞) in q

enq(q, v) ≜
  let i = fetch-add(q, 1, rel) in
  store(q + i + 1, v, rel)

deq(q) ≜
  loop
    let range = load(q, acq) in
    for i = 1 to range do
      let x = atomic-xchg(q + i, 0, acq) in
      if x ≠ 0 then break_2 x
```

Fig. 2. The (weak) Herlihy-Wing queue implementation [Herlihy and Wing 1990] under WMC; the strong variant is obtained by replacing the highlighted mode with the (stronger) acquire-release mode acqrel.

implementations, leading to the challenging task of inferring to. Moreover, the existential quantification of to is not conducive to existing verification techniques such as model checking [Abdulla et al. 2018; Kokologiannakis et al. 2018], which would have to enumerate *all* possible total orders.

Third, as we discovered whilst trying to verify the correctness of the Herlihy-Wing queue [Herlihy and Wing 1990] under WMC, the total order to *may not exist at all*. That is, WMC implementations often satisfy *weaker* specifications, in keeping with the weaker guarantees of WMC.

For the third limitation, consider the Herlihy-Wing implementation in Fig. 2. The queue is represented as a zero-initialised infinite array (from $q+1$ onwards) with the index of the last array element (the queue tail) stored at q. A call to $enq(q, v)$ reads the current value i of the tail, increments it by one thus reserving the slot immediately after the tail (at $q{+}i{+}1$), and inserts v at the reserved slot. A call to $deq(q)$ traverses the queue searching for a non-zero value. To do this, the value of each array entry is atomically exchanged (via *atomic-xhcg*) with zero, and the exchanged value is returned in x. If x is non-zero, then it is returned and the call terminates; otherwise, the search is repeated until a non-zero value is found. Note that the dequeue implementation is blocking: it does not terminate until it succeeds to dequeue a value.

The underlying memory model of the original implementation [Herlihy and Wing 1990] is SC. Here, we develop a WMC variant by using C11 release-acquire (RA) registers. In Fig. 2 we also present a *strong* and perhaps less efficient variant of the implementation by using the C11 'acquire-release' mode in lieu of the access highlighted. As we discuss in §6.2, the strong implementation satisfies the strong linearisability-style specification. However, rather counter-intuitively, given an execution of the weak implementation, it is not always possible to construct a total order to of the queue method calls as described above. To see this, consider the following program where the $/\!/ v$ annotation denotes that value v is dequeued, and $l :$ labels the method call:

$$\begin{array}{l|l|l|l} a : enq(q, v_1); & b' : deq(q); \; /\!/ v_2 & c' : deq(q); \; /\!/ v_3 & d : enq(q, v_4); \\ b : enq(q, v_2) & c : enq(q, v_3) & d' : deq(q) \; /\!/ v_4 & a' : deq(q) \; /\!/ v_1 \end{array} \qquad \text{(W-HWQ)}$$

We now demonstrate that it is not possible to construct a strict total order to for the above execution. Given the program order in the first (left-most) thread, we know that v_1 is enqueued before v_2. Similarly, from the second thread we know that v_2 is dequeued before v_3 is enqueued. As v_2 is enqueued before it is dequeued, we know that v_2 is enqueued before v_3. Lastly, from the third thread we know that v_3 is dequeued before v_4. Given the first-in-first-out (FIFO) property of queues, we thus know that v_3 is enqueued before v_4. The enqueue operations are then ordered as: $a \xrightarrow{\text{to}} b \xrightarrow{\text{to}} c \xrightarrow{\text{to}} d$ and the dequeue operations as $a' \xrightarrow{\text{to}} b' \xrightarrow{\text{to}} c' \xrightarrow{\text{to}} d'$, to maintain the FIFO paradigm. From the last thread we have $d \xrightarrow{\text{to}} a'$, from which we deduce that $c \xrightarrow{\text{to}} b'$. On the other hand, from the second thread we have $(b', c) \in \mathsf{hb} \subseteq \text{to}$, leading to a cycle in to.

Although it is not possible to construct a total order to for the annotated behaviour in (W-HWQ), it can be produced by the weak implementation in Fig. 2—we have confirmed this both by hand and

via the RCMC model-checking tool [Kokologiannakis et al. 2018]. Hence, this weak implementation does *not* satisfy the strong linearisability-style specification. Nevertheless, the weak implementation is a natural WMC adaptation of the original SC implementation. In particular, in the original implementation, each $enq(q, v)$ *synchronises* with its matching $deq(q)$ removing v. To ensure such synchronisation using RA registers, the natural choice is to use release (`rel`) writes in $enq(q, v)$, and acquire (`acq`) reads in $deq(q)$. As such, we found the weak behaviour in (W-HWQ) rather surprising.

The absence of a total order to does not however render this weak implementation useless; rather, the implementation provides *weaker* guarantees to facilitate a more efficient implementation. As such, one can weaken the library specification (library guarantees) whilst staying within the spirit of the definitions of weak memory models. In §5 we thus develop alternative weaker specifications for several libraries (including queues). In general, linearisability-style specifications are suitable in the SC setting where the total order to can be inferred from the total execution order afforded by SC. In a WMC setting however, such total execution order does not generally exist, and so concurrent libraries need not enforce a total order amongst their method calls.

In this article, we thus develop a *general framework* for library *specification* and *verification* that is: (1) agnostic to the underlying memory model and can be used in both SC and WMC settings; and (2) moves away from the linearisability-style specifications, allowing for direct specifications that avoid the total order (to) quantification. Note that the latter does not preclude developing specifications that are as strong as those in the linearisability style. As we describe later in §5, in several cases (e.g. locks and queues) we also develop *equivalent* specifications with the same strong guarantees, whilst avoiding the total order quantification. For instance, in case of queues we show that the lack of certain cycles in an execution ensures the existence of to order and vice versa. This in turn makes it easier to establish the correctness of candidate implementations, and to employ existing WMC model checking techniques [Abdulla et al. 2018; Kokologiannakis et al. 2018].

Representation of Executions. In the linearisability literature [Herlihy and Wing 1990], given a program execution, the method calls are typically represented by a *pair* of events denoting the method *invocation* (initiating the call) and its *response* (returning from the call). For instance, in the queue execution $G_1 = \mathsf{enq}(q, 1); \mathsf{deq}(q, 1)$, rather than a single event per method (e.g. $\mathsf{enq}(q, 1)$ for enqueuing 1), one would instead have $G_1' = inv(\mathsf{enq}, q, 1); res(\mathsf{enq}, q); inv(\mathsf{deq}, q); res(\mathsf{deq}, q, 1)$. As each call is represented by two events, the hb order amongst events is no longer simply determined by their position in the sequence. Instead, a method call m_1 is said to happen before m_2 iff the response of m_1 appears before the invocation of m_2 in the sequence. For instance, in G_1' the call for enqueuing 1 happens before that of dequeuing 1. In an SC concurrency setting however, the invocations and responses of methods in different threads may be arbitrarily interleaved, and thus the method calls of different threads may not be hb-ordered with respect to one another. An execution is *sequential* if the invocation and response of a call are not interleaved by others. For instance, G_1' is sequential, whilst $inv(\mathsf{enq}, q, 1); inv(\mathsf{deq}, q); res(\mathsf{enq}, q); res(\mathsf{deq}, q, 1)$ is not.

In our framework here, rather than representing each call with a pair of invocation and response events (as in G_1'), we opt instead for a *single* event per call (as in G_1). We made this design choice for three reasons. First, capturing each call with a single event allows for a *simpler and cleaner formalism*. Second, we argue that the pair representation in the style of linearisability specifications is an *artefact of SC*. In particular, associating each call with an event pair allows one to determine the hb order amongst interleaving calls of different threads, as described above; the pair representation is thus helpful in the SC setting. However, in the WMC setting, the hb order is generally determined as a transitive closure of other orders, including the program order. As such, the pair representation is of little use in the general context of WMC. As our aim is to develop a general framework agnostic to the concurrency model (SC or WMC), we move away from the pair representation.

Third and most importantly, the singleton representation is more *WMC-friendly* and more *suitable for declarative models*. In particular, in the literature of declarative concurrency models, executions are represented as a number of *partial orders* over events rather than *sequences* of events; and each execution event (e.g. a write to memory) is represented as a *single* event rather than a *pair*. As our aim here is to develop a specification framework in the style of existing declarative models, the singleton representation is more suitable. In particular, as we discuss below, we provide a general specification framework in which many existing declarative models (e.g. the WMC C11 model[1]) can be formalised. As such, by continuing the trend of singleton representation, we can employ and adapt the existing verification tools for declarative concurrency models and WMC, such as those for model checking [Abdulla et al. 2018; Kokologiannakis et al. 2018].

Benign Synchronisation Cycles. By opting for singleton events, we encounter interesting challenges in library specification. In the concurrency literature, the hb relation is described as a *strict* partial order over events and is typically defined to include the program order po (the control flow in each thread) and the *synchronisation order* so: $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{so})^+$.[2] However, abstracting away from event pairs introduces a *benign* kind of so cycles that by extension ($\mathsf{so} \subseteq \mathsf{hb}$) violates the strictness condition on hb. We refer to these cycles as *benign* as they are naturally present in the associated libraries due to the bidirectional synchronisation amongst methods.

To understand this, consider an *exchanger* object as in the `java.util.concurrent` library, exposing the $\mathit{exchange}(g, v)$ method. Exchangers allow threads to pair up and *atomically* swap values; that is, either *both* threads succeed to exchange their values with one another or *neither* thread does. Given an exchanger object at location g, a call to $v' = \mathit{exchange}(g, v)$ allows the calling thread to offer value v in exchange for the return value v'. We represent such a call by the singleton event $\mathsf{exchange}(g, v, v')$. As such, a call event $m_1 : \mathsf{exchange}(g, v, v')$ (offering v in exchange for v') *synchronises with* its matching symmetric call $m_2 : \mathsf{exchange}(g, v', v)$, and vice versa. That is, synchronisation between m_1 and m_2 is *bidirectional* as each reads the value offered by the other. In other words, we have $(m_1, m_2), (m_2, m_1) \in \mathsf{so}$, leading to a cycle in so (and hb).

To account for such benign so cycles inherent to certain libraries, we require that the only cycles present in hb be those comprising solely so edges: no hb cycle may use po edges. Note that were we to represent each call as a pair of events, such cycles would be pre-empted as the so edge would be between the symmetric invocation and response events. For instance, if we represent m_1 as the pair $i_1 : \mathit{inv}(\mathsf{exchange}, v)$, $r_1 : \mathit{res}(\mathsf{exchange}, v')$, and m_2 as $i_2 : \mathit{inv}(\mathsf{exchange}, v')$, $r_2 : \mathit{res}(\mathsf{exchange}, v)$, we then have $(i_1, r_2), (i_2, r_1) \in \mathsf{so}$, averting an so cycle. However, due to reasons discussed above, we opt for the singleton representation and allow instead for such benign so cycles.

Lastly, recall from earlier that the authors in [Hemed et al. 2015] identify the exchanger as an example of a library without a sequential specification. Indeed, the bidirectional synchronisation between matching exchanges is the very reason behind this. In particular, in the example above it is not possible to construct a sequential execution that agrees with hb: both sequential execution candidates, $i_1; r_1; i_2; r_2$ and $i_2; r_2; i_1; r_1$, violate the hb constraint $(i_1, r_2), (i_2, r_1) \in \mathsf{so} \subseteq \mathsf{hb}$. Put differently, in our singleton representation it is not possible to construct a strict total order on m_1, m_2 that agrees with hb, as we have both $(m_1, m_2), (m_2, m_1) \in \mathsf{so} \subseteq \mathsf{hb}$.

Connection to Existing WMC Specifications. As discussed briefly in §1, we can specify many existing WMC language- and hardware-level declarative models as instances of our framework.

[1] Throughout this article we refer to the RC11 model of Lahav et al. [2017] simply as the C11 model.

[2] The $(\mathsf{po} \cup \mathsf{so})^+$ denotes the transitive closure of $\mathsf{po} \cup \mathsf{so}$. The so order is determined by libraries and their guarantees. For instance, in case of a queue library, an enqueue event adding value v synchronises with the dequeue event that removes v.

For instance, we can specify the C11 model as a library in our framework. This is because existing C11 specifications (e.g. [Lahav et al. 2017]) are specified in our declarative style and can be directly ported to our framework. In particular, the various relations and concepts of the C11 model (e.g. release sequences and modification order) can all be formalised via the components of our framework. Indeed, in our Coq formalism we have specified the C11 model of Lahav et al. [2017] as a library.[3] We demonstrate this in §4 via several examples. Analogously, existing hardware specifications such as the TSO model by Owens et al. [2009] can be directly ported to our framework. As such, our formalism is a general unifying framework for existing and future WMC models.

Justifying Weak Specifications. As discussed, in §5 we present several *weak* specifications with weaker guarantees than their strong (linearisability-style) counterparts. For instance, we develop a weak queue specification against which the weak Herlihy-Wing queue implementation in Fig. 2 can be verified. To demonstrate the suitability of such weak specifications, we use several criteria to gauge their fitness: (1) implementability; (2) utility; and (3) feasibility.

For (1), we verify several well-known implementations against our weak specifications in §6 .

For (2), we argue that often weaker specifications provide sufficient guarantees without being too restrictive. For instance, our weak queue specification is strong enough to provide the necessary guarantees when the queue is used in the single-producer single-consumer pattern, while the strong specification is *too strong* and provides additional guarantees not necessary for that usage.

For (3), we demonstrated that for several libraries (e.g. exchangers), weak specifications are the only viable option as these libraries do not lend themselves to strong linearisability-style specifications. Moreover, we showed that WMC adaptations of existing SC implementations may not satisfy the same strong specifications. For instance, as discussed above, the weak Herlihy-Wing implementation in Fig. 2 is a natural WMC adaptation of the original implementation in [Herlihy and Wing 1990], even though it does not satisfy the same strong specification.

In general, while strong specifications are more intuitive, they place an undue burden on library implementers, leading to substantial performance loss. As Shavit [2011] observes, to support scalability, the consistency requirements (specification) of data structures need to be *relaxed* (weakened).

Compositional Verification. Thus far we have only considered executions in which all constituent method calls are to the same library, e.g. the queue library. However, concurrent programs often comprise method calls of different libraries. For instance, the (SB-lib) program in §1 comprises calls to the queue and stack libraries. When this is the case, the correctness condition in linearisability-style approaches is adapted accordingly as follows. For the overall execution to be correct, there must exist a strict total order to on *all* execution events, such that restricting the events to those of each library L and subsequently ordering them by to yields a sequential execution allowed by L. That is, to must be first constructed for all events, and then checked for each constituent library. However, it is possible that such a to cannot be constructed for all events, even though the events on each library can be totally ordered. For instance, to produce the annotated behaviour of (SB-lib), the dequeue event (d) must be ordered before the enqueue event (e) since otherwise the dequeue cannot return empty; i.e. $(d, e) \in \mathsf{to}$. Analogously for the stack events we must have $(r, a) \in \mathsf{to}$. As such, the events of the queue and stack libraries can each be totally ordered. However, as the events of each thread are hb-ordered by the program order, $(e, r), (a, d) \in \mathsf{hb}$, and any candidate to must agree with hb ($\mathsf{hb} \subseteq \mathsf{to}$), we then have $e \xrightarrow{\mathsf{to}} r \xrightarrow{\mathsf{to}} a \xrightarrow{\mathsf{to}} d \xrightarrow{\mathsf{to}} e$, violating the strictness condition on to. That is, we cannot construct a total order to for the overall execution.

To remedy this, given an overall execution comprising the events of different libraries, in our framework we first restrict the execution events to those of individual libraries, and then check

[3] See the 'wmcLibrary_coq/wlib/LibC11.v' Coq file in the accompanying artefact.

whether the projected execution satisfies the specification of its associated library. For instance, in (SB-lib) we check if (1) the execution comprising the $\{e, d\}$ events satisfies the queue specification; and (2) the execution comprising the $\{a, r\}$ events satisfies the stack specification.

This *per-library* validation is inspired by the *per-location* definitions in declarative WMC models. In particular, the local (per-location) versus global (all locations) dichotomy is one of the differentiating factors between WMC and SC. For instance, in the (WMC) RA fragment of C11 [Lahav et al. 2016], execution consistency is checked for *each* memory location separately. This is in contrast to SC, where consistency is checked for *all* locations at once. As our aim is to develop a framework in the style of declarative models, we opt for the weaker local (per-library) validation. As discussed above, the stronger behaviour can be encoded by inserting e.g. C11 fences to enforce the desired ordering between the events of *different* libraries (as discussed shortly, C11 itself can be formalised in our framework). By contrast, had we opted for the stronger global (all libraries) validation, we would have precluded many (valid) weak behaviours, including that of (SB-lib).

Note that our per-library validation does not preclude different libraries from introducing synchronisation constraints on one another, just as per-location validation in WMC does not prevent different locations from inducing synchronisation constraints on each other, e.g. in the "message-passing" (MP) litmus test. To see this, consider the following library variant of MP:

$$\begin{array}{l} e : enq(q, 1); \\ a : push(s, 2) \end{array} \;\Big\|\; \begin{array}{l} r : \texttt{let } b = try\text{-}pop(s) \texttt{ in} \\ \quad \texttt{if } b = 2 \texttt{ then} \\ \quad\quad d : try\text{-}deq(q) \;\; \textit{// returns 1} \end{array} \qquad \text{(MP-lib)}$$

When the condition of the `if` statement is satisfied ($b = 2$), the $try\text{-}pop(s)$ call reads the value pushed by $push(s, 2)$, and thus their associated events synchronise, i.e. $(a, r) \in \mathsf{so}$. As such, since $e \xrightarrow{\mathsf{po}} a \xrightarrow{\mathsf{so}} r \xrightarrow{\mathsf{po}} d$ and $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{so})^+$, we have $e \xrightarrow{\mathsf{hb}} d$, and thus $try\text{-}deq(q)$ must return 1.

In other words, the $e \xrightarrow{\mathsf{hb}} d$ edge between the queue library events is brought about in part due to the synchronisation edge $a \xrightarrow{\mathsf{so}} r$ of the stack library. Were we to restrict the execution events to only those of the queue library $\{e, d\}$, without taking into account the hb edges induced by the stack library, the presence of the $e \xrightarrow{\mathsf{hb}} d$ edge could not be ascertained. To this end, per-library validation is carried out with respect to the hb relation calculated for the overall execution. That is, the hb relation is first calculated for the entire execution and then restricted to events of each individual library, whereupon per-library validation is carried out as discussed above.

Towers of Abstraction. As we demonstrate in §6, we can use our framework to build *abstraction towers*, allowing us to verify the correctness of library implementations *compositionally*.

For instance, we first specify a (fragment) of the C11 library. Using C11 operations, we implement a mutual exclusion (mutex) lock library. We then appeal to our C11 specification to verify the correctness of our mutex implementation against its specification (also developed in our framework). Fig. 3 illustrates the library implementations we specify and verify in our framework. HW-Queue denotes the two Herlihy-Wing queue implementations discussed earlier; MRSW-Lock denotes two implementations of multiple-readers-single-writer locks; ExchArray denotes an exchanger array; and Weak-Stack denotes a stack where push and pop operations may fail (see §6).

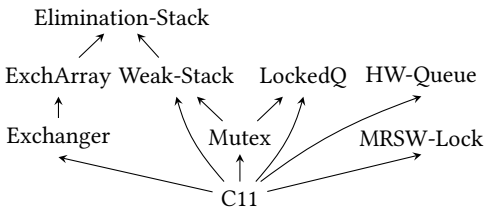

Fig. 3. Towers of abstraction built in our framework

Our framework allows for both *horizontal* and *vertical* composition. The (SB-lib) program is an example of horizontal composition where we compose the stack and queue libraries to develop (SB-lib). In Fig. 3 we illustrate several examples of vertical composition. For instance, we combine the ExchArray and Weak-Stack specifications to implement a new layer: the elimination stack.

3 SEMANTICS

We proceed with the semantics of our expression language presented in Fig. 1.

Notation. Given a set A, we write $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$. Given a relation r, we write $r|_A$ for $r \cap (A \times A)$; and r^{-1} for the inverse of r. We write $r^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closure of r, respectively. We write $dom(r)$ for the domain of r (i.e. $\{a \mid \exists b.\ (a,b) \in r\}$) and $rng(r)$ for its range (i.e. $dom(r^{-1})$). We write $r(a)$ for $\{b \mid (a,b) \in r\}$ and $r(A)$ for $\bigcup_{a \in A} r(a)$. Given two relations r_1 and r_2, we write $r_1; r_2$ for their relational composition: $\{(a,b) \mid \exists c.\ (a,c) \in r_1 \wedge (c,b) \in r_2\}$. We write $r|_{\text{imm}}$ for the *immediate* edges in r, i.e. $r \setminus (r; r)$. Following Cerone et al. [2015], r is *prefix-finite* if $r^{-1}(b)$ is finite for every $b \in rng(r)$.

Events and Plain Executions. We define the semantics of programs in terms of *plain executions*. A plain execution, $G = \langle E, \mathrm{po} \rangle$, is a (partially) ordered set of *events* E, where the order po represents whether one event precedes another in the control flow of the program. An *event* 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 $m(v_1, \ldots, v_n, v)$ and represent a method invocation with arguments $v_1, \ldots, v_n$ and return value v. We typically use a, b and e to range over events. The function lab(.) projects the label of an event.

We write $\emptyset_{\mathrm{G}} \triangleq \langle \emptyset, \emptyset \rangle$ for the empty execution and $\{a\}_{\mathrm{G}} \triangleq \langle \{a\}, \emptyset \rangle$ for the execution with a single event a. Given two executions, $G_1 = \langle E_1, \mathrm{po}_1 \rangle$ and $G_2 = \langle E_2, \mathrm{po}_2 \rangle$, with disjoint sets of events ($E_1 \cap E_2 = \emptyset$), we define their sequential composition, $G_1; G_2$, by ordering all G_1 events before those of G_2: $G_1; G_2 \triangleq \langle E_1 \cup E_2, \mathrm{po}_1 \cup \mathrm{po}_2 \cup (E_1 \times E_2) \rangle$. Similarly, we define their parallel composition, $G_1 \| G_2$, by placing no additional order between events of G_1 and G_2: $G_1 \| G_2 \triangleq \langle E_1 \cup E_2, \mathrm{po}_1 \cup \mathrm{po}_2 \rangle$.

Definition 1 (Events and plain executions)**.** The set of *event labels* is $\mathsf{Lab} \triangleq \mathsf{Method} \times \mathsf{Val}^* \times \mathsf{Val}$. The set of *events* is $\mathsf{Event} \triangleq \mathbb{N} \times \mathsf{Lab}$. A *plain execution*, $G \in \mathsf{PExec}$, is a tuple $G = \langle E, \mathrm{po} \rangle$, where E is a set of *events* with distinct identifiers and $\mathrm{po} \subseteq E \times E$ is a prefix-finite strict partial order denoting the *program order* relation.

Expression Semantics. Expressions are interpreted with respect to an *environment* $\Gamma \in \mathsf{Env}$, which maps variables to their values. The interpretation of an expression e with respect to Γ, written $[\![\mathtt{e}]\!](\Gamma)$, generates a set of pairs of the form (r, G), where r denotes the expression *outcome* returned by e, and G denotes the corresponding plain execution leading to r. The outcome r may in turn be either $\bot$, when the computation has not yet terminated; or a pair (v, n), where $v \in \mathsf{Val}$ denotes the return value and $n \in \mathbb{N}$ denotes the break number, i.e. the number of `loop` blocks to exit. Note that a non-zero break number is applicable only when returned from within a loop.

The interpretation function $[\![.]\!]$ is given in Fig. 4, and is defined by induction over the expression syntax. Interpreting value v yields outcome $\langle v, 0 \rangle$ with the empty execution $\emptyset_{\mathrm{G}}$; interpreting variable x looks up x in the environment Γ, and thus returns outcome $\langle \Gamma(x), 0 \rangle$ with empty execution $\emptyset_{\mathrm{G}}$. Interpreting a method call $m(x_1, \ldots, x_n)$ adds a singleton event with label $m(v_1, \ldots, v_n, v)$, denoting a call to method m with arguments $v_1, \ldots, v_n$ (where $v_i = \Gamma(x_i)$) and return value v.

The interpretation of a conditional is determined by the value of the condition in the standard fashion. When $\langle r_1, G_1 \rangle \in [\![\mathtt{e}_1]\!](\Gamma)$ and $\langle r_2, G_2 \rangle \in [\![\mathtt{e}_2]\!](\Gamma)$, the interpretation of `let` x `=` $\mathtt{e}_1$ `in` $\mathtt{e}_2$ captures the sequential composition of $\mathtt{e}_1$ and $\mathtt{e}_2$ and comprises two cases depending on the outcome of $\mathtt{e}_1$. When the computation of $[\![\mathtt{e}_1]\!](\Gamma)$ terminates with a zero break number, as expected

$$\Gamma \in \mathsf{Env} \triangleq \mathsf{Var} \to \mathsf{Val} \qquad \mathsf{RV} \triangleq \mathcal{P}((\mathsf{Val} \times \mathbb{N})_\perp \times \mathsf{PExec})$$

$$[\![\_]\!] \;:\; \mathsf{Exp} \to \mathsf{Env} \to \mathsf{RV}$$

$$[\![v]\!](\Gamma) \triangleq \{\langle\langle v, 0\rangle, \emptyset_{\mathrm{G}}\rangle\} \qquad [\![x]\!](\Gamma) \triangleq \{\langle\langle \Gamma(x), 0\rangle, \emptyset_{\mathrm{G}}\rangle\}$$

$$[\![m(x_1, \dots, x_n)]\!](\Gamma) \triangleq \left\{\langle\langle v, 0\rangle, \{\langle n, m(\Gamma(x_1), \dots, \Gamma(x_n), v)\rangle\}_{\mathrm{G}}\rangle \;\middle|\; \begin{array}{l} v \in \mathsf{Val}, \\ n \in \mathbb{N} \end{array}\right\} \cup \{\langle \perp, \emptyset_{\mathrm{G}}\rangle\}$$

$$[\![\texttt{let } x = \mathtt{e}_1 \texttt{ in } \mathtt{e}_2]\!](\Gamma) \triangleq \left\{\langle r_2, G_1; G_2\rangle \;\middle|\; \langle\langle v_1, 0\rangle, G_1\rangle \in [\![\mathtt{e}_1]\!](\Gamma) \wedge \langle r_2, G_2\rangle \in [\![\mathtt{e}_2]\!](\Gamma[x \mapsto v_1])\right\}$$
$$\cup \left\{\langle r_1, G_1\rangle \;\middle|\; \langle r_1, G_1\rangle \in [\![\mathtt{e}_1]\!](\Gamma) \wedge \nexists v.\ r_1 = \langle v, 0\rangle\right\}$$

$$[\![\mathtt{e}_1 \| \mathtt{e}_2]\!](\Gamma) \triangleq \left\{\mathsf{par}(r_1, G_1, r_2, G_2) \;\middle|\; \langle r_1, G_1\rangle \in [\![\mathtt{e}_1]\!](\Gamma) \wedge \langle r_2, G_2\rangle \in [\![\mathtt{e}_2]\!](\Gamma)\right\}$$

$$[\![\texttt{if } x \texttt{ then } \mathtt{e}_1 \texttt{ else } \mathtt{e}_2]\!](\Gamma) \triangleq \begin{cases} [\![\mathtt{e}_1]\!](\Gamma) & \text{if } \Gamma(x) \neq 0 \\ [\![\mathtt{e}_2]\!](\Gamma) & \text{if } \Gamma(x) = 0 \end{cases} \qquad [\![\mathtt{break}_n\ x]\!](\Gamma) \triangleq \{\langle\langle \Gamma(x), n\rangle, \emptyset_{\mathrm{G}}\rangle\}$$

$$[\![\texttt{loop } \mathtt{e}]\!](\Gamma) \triangleq \bigcup_{n \in \mathbb{N}} \left\{\langle\langle v, k\rangle, G_1; \dots; G_n\rangle \;\middle|\; \begin{array}{l} \forall i < n.\ \langle\langle \_, 0\rangle, G_i\rangle \in [\![\mathtt{e}]\!](\Gamma) \\ \wedge\ \langle\langle v, k{+}1\rangle, G_n\rangle \in [\![\mathtt{e}]\!](\Gamma) \end{array}\right\}$$
$$\cup \bigcup_{n \in \mathbb{N}} \left\{\langle \perp, G_1; \dots; G_n\rangle \;\middle|\; \begin{array}{l} \forall i < n.\ \langle\langle \_, 0\rangle, G_i\rangle \in [\![\mathtt{e}]\!](\Gamma) \\ \wedge\ \langle \_, G_n\rangle \in [\![\mathtt{e}]\!](\Gamma) \end{array}\right\}$$

$$\mathsf{par}(r_1, G_1, r_2, G_2) \triangleq \begin{cases} \langle\langle 0, 0\rangle, G_1 \| G_2\rangle & \text{if } \exists v_1, v_2.\ r_1 = \langle v_1, 0\rangle \wedge r_2 = \langle v_2, 0\rangle \\ \langle \perp, G_1 \| G_2\rangle & \text{otherwise} \end{cases}$$

Fig. 4. The semantics of our ANF expression language

the resulting outcome is that of $\mathtt{e}_2$ (i.e. r_2) and the resulting execution is obtained from the sequential composition of executions $(G_1; G_2)$. On the other hand, when $[\![\mathtt{e}_1]\!](\Gamma)$ does not terminate, or terminates with a non-zero (invalid) break number, the interpretation yields $\langle r_1, G_1\rangle$.

Analogously, when $\langle r_1, G_1\rangle \in [\![\mathtt{e}_1]\!](\Gamma)$ and $\langle r_2, G_2\rangle \in [\![\mathtt{e}_2]\!](\Gamma)$, interpretation of $\mathtt{e}_1 \| \mathtt{e}_2$ captures the parallel composition of $\mathtt{e}_1$ and $\mathtt{e}_2$ via the par function. The definition of par (at the bottom of Fig. 4) comprises two cases depending on the $\mathtt{e}_1$ and $\mathtt{e}_2$ outcomes. When both computations terminate with a zero break number, the outcome is $\langle 0, 0\rangle$ (parallel composition does not return a meaningful value). Otherwise, the computation is marked as non-terminating ($\perp$). In both cases, the resulting execution is obtained from the parallel composition of the constituent executions $(G_1 \| G_2)$.

Recall that our looping construct `loop e` models the infinite execution of `e`, and may be terminated only when a break expression is executed within it. Interpreting $[\![\texttt{loop } \mathtt{e}]\!](\Gamma)$ thus comprises two cases. The first captures the case when the computation of $[\![\texttt{loop } \mathtt{e}]\!](\Gamma)$ terminates after n iterations. That is, computing the first $n{-}1$ iterations of `e` yield $\langle -, 0\rangle$ (with a zero break number) and thus do not trigger loop termination, whilst the n^{th} iteration of `e` yields $(v, k{+}1)$, with the non-zero break number $(k{+}1)$ indicating loop termination. As such, the loop is exited with return value v and the break number is decremented by one (k). The resulting execution is that of the n iterations composed sequentially. The second captures the ongoing computation of $[\![\texttt{loop } \mathtt{e}]\!](\Gamma)$ after n iterations, and thus the returned outcome is $\perp$. As before, the resulting execution is obtained from the sequential composition of the n executions accumulated thus far. Interpreting $\mathtt{break}_n\ x$ simply returns the value of x and the indicated break number n, (i.e. $\langle \Gamma(x), n\rangle$), with empty execution $\emptyset_{\mathrm{G}}$.

We write $[\![\mathtt{e}]\!]$ for $[\![\mathtt{e}]\!](\Gamma_0)$ where Γ_0 assigns 0 to all variables. Note that the executions generated by $[\![.]\!]$ are *prefix-closed*: for all $\mathtt{e}, E', \Gamma$ and $\langle r, \langle E, \mathrm{po}\rangle\rangle \in [\![\mathtt{e}]\!](\Gamma)$, if $E' \subset E$ and E' is prefix-closed on po (i.e. $\{a \mid \exists b \in E'.\ (a, b) \in \mathrm{po}\} \subseteq E'$), then $\langle \perp, \langle E', \mathrm{po}|_{E'}\rangle\rangle \in [\![\mathtt{e}]\!](\Gamma)$, where $\mathrm{po}|_{E'} \triangleq \mathrm{po} \cap (E' \times E')$.

4 LIBRARY SPECIFICATION AND VERIFICATION FRAMEWORK

We describe our formal declarative framework for specifying and verifying concurrent libraries.

4.1 Specifying Concurrent Libraries

The formal framework presented here is for specifying a general library L. Note that our formal development does not depend on any pre-existing libraries in that even the most fundamental operations can be formulated as library operations. In particular, as we demonstrate shortly, we can formalise the *C11 library* in our framework, providing us with read, write and atomic update operations with various access modes. Similarly, although we appeal to standard arithmetic operators, these themselves can be formalised as operations of an arithmetic library in our framework. Lastly, our framework is agnostic to the underlying concurrency model; as such, all specifications developed in our framework are usable under both SC and WMC settings.

Note that a prefix-finite strict partial order r is well-founded and satisfies $r = r|_{\text{imm}}{}^{+}$.

Library Interfaces. To define a concurrent library formally, we first formalise the notion of *library interfaces*. The interface of a concurrent library L is a tuple $\langle \mathcal{M}, \mathcal{M}_c, \texttt{loc} \rangle$, where $\mathcal{M} \subseteq \mathsf{Label}$ denotes the library *labels* (Def. 1), $\mathcal{M}_c$ denotes the library *constructor labels* and $\texttt{loc}$ is a *location function*. The $\mathcal{M}$ component tracks the labels of the library methods. We require that $\mathcal{M}$ methods be associated with a function $\texttt{arity}(.)$, mapping each method onto its arity (the number of its arguments). As such, we preclude duplicate method names of different arities for simplicity.

The next two components, $\mathcal{M}_c$ and $\texttt{loc}$, are used to formalise *encapsulation*. When specifying a concurrent library, it is crucial to ascertain execution encapsulation. For instance, if location x has been designated as a lock location accessed by a mutual exclusion (mutex) library L^{MX}, it is important to ensure that x is *owned* by the L^{MX} library in that it is accessed (read and written) solely by L^{MX}. Were this not the case, location x may be accessed and modified e.g. as a regular heap location, thus violating the mutual exclusion properties guaranteed by L^{MX}. To formalise the notions of ownership and encapsulation, we designate a subset of labels in $\mathcal{M}$ as *constructor labels*: $\mathcal{M}_c \subseteq \mathcal{M}$. An event with a constructor label is one that allocates and claims ownership of the relevant memory locations. In the example above, the constructor of the mutex at x claims ownership of x. To this end, we assume a set of memory *locations*, $\mathsf{Loc} \subseteq \mathsf{Val}$. The *location function*, $\texttt{loc}(.) : \mathcal{M} \to \mathcal{P}(\mathsf{Loc})$, returns the set of locations of a label. When $l_c \in \mathcal{M}_c$ and $l_m \in \mathcal{M} \setminus \mathcal{M}_c$, then $\texttt{loc}(l_c)$ denotes the set of locations *owned* by the constructor, whilst $\texttt{loc}(l_m)$ denotes the set of locations *accessed* by l_m. We revisit the notion of encapsulation shortly and describe it formally.

Library Executions. In the literature of declarative concurrency models, the traces of memory events generated by concurrent program are commonly represented as a set of execution graphs, where each graph G comprises: (1) a set of events denoting the graph nodes; and (2) a number of relations on events, denoting the sundry graph edges. Similarly, given the library L interface $\langle \mathcal{M}, \mathcal{M}_c, \texttt{loc} \rangle$, we describe the behaviour of L as a set of *library executions*. [4] An execution G of library L is a tuple of the form $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb} \rangle$. The set E denotes the execution *events* (see Def. 1), with each $a \in E$ denoting a call to an L method; that is, $\texttt{lab}(a) \in \mathcal{M}$. The po relation is the *program order* (as before); the com relation denotes the *communication order*. Intuitively, com relates those events in E that exchange information. For instance, when formalising the C11 library registers, the com relation describes the 'reads-from' relation, where $(w, r) \in \mathsf{com}$ denotes that event r reads a value written by event w. In case of a queue library, com relates matching enqueue and dequeue operations; i.e. $(e, d) \in \mathsf{com}$ denotes that d dequeues a value enqueued by e.

[4] This is analogous to specification in terms of valid sequences of method calls in the formal definition of linearisability.

The so relation denotes the *synchronisation order*. Intuitively, so denotes those $(\mathsf{po} \cup \mathsf{com})^+$ paths that contribute to the 'happens-before' relation: $\mathsf{so} \subseteq (\mathsf{po} \cup \mathsf{com})^+$. For instance, in case of a release-acquire register, all com edges are also so edges. By contrast, in case of a relaxed register, com edges do not contribute to the 'happens-before' order at all.

The last component, lhb, denotes the *local-happens-before* relation. Intuitively, lhb captures causality between library events.[5] As is standard practice, we require that lhb be transitive and that $\mathsf{po} \cup \mathsf{so} \subseteq \mathsf{lhb}$. Recall from §2 that to enable the specification of certain libraries such as exchangers, we allow for benign so cycles. As such, since $\mathsf{so} \subseteq \mathsf{lhb}$, rather than requiring that lhb be a strict partial order, we require that lhb be acyclic except for cycles comprising solely so edges. That is, lhb should have no cycles involving at least one po edge: po; lhb is irreflexive.

The set $\mathcal{G}_L$ denotes the *execution set of* L, consisting of all executions of L.

Concurrent Library Specification. Given the library L interface $\langle \mathcal{M}, \mathcal{M}_c, \mathtt{loc} \rangle$ and its execution set $\mathcal{G}_L$, we define the *concurrent library* L as the tuple $\langle \mathcal{M}, \mathcal{M}_c, \mathtt{loc}, \mathcal{G}_c, \mathcal{G}_{wf} \rangle$. The set $\mathcal{G}_c \subseteq \mathcal{G}_L$ denotes the set of *consistent library executions*, i.e. those deemed valid by the library. We require that consistent executions be monotonic with respect to lhb: extending lhb must not introduce additional behaviours.

The set $\mathcal{G}_{wf} \subseteq \mathcal{G}_L$ denotes the set of *well-formed* executions. The library guarantees afforded to its clients are subject to the proviso that clients use the library in a *well-formed* fashion. For instance, in case of a mutex library, we expect that clients acquire the mutex prior to releasing it. Such library-specific conditions are captured by the $\mathcal{G}_{wf}$ component. When the library imposes no well-formedness conditions on clients, one defines $\mathcal{G}_{wf}$ as $\mathcal{G}_L$.

Definition 2 (Libraries)**.** Assume a set of *memory locations* $\mathsf{Loc} \subseteq \mathsf{Val}$. An *interface* of a concurrent library L is a tuple $\langle \mathcal{M}, \mathcal{M}_c, \mathtt{loc} \rangle$, where $\mathcal{M} \subseteq \mathsf{Label}$ denotes the *library labels* (methods), $\mathcal{M}_c \subseteq \mathcal{M}$ denotes the library *constructor labels*, and $\mathtt{loc}(.) : \mathcal{M} \to \mathcal{P}(\mathsf{Loc})$ denotes the *location function*, such that $\mathtt{loc}(l) \neq \emptyset$ for $l \in \mathcal{M}_c$. The set of *library* L *executions*, denoted $\mathcal{G}_L$, consists of all tuples $G = \langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb} \rangle$, where:

- E is a set of *events* with distinct identifiers over the $\mathcal{M}$ labels: $\forall e \in E.\ \mathtt{lab}(e) \in \mathcal{M}$;
- $\mathsf{po} \subseteq E \times E$ is the *program order*;
- $\mathsf{com} \subseteq E \times E$ is the *communication order*;
- $\mathsf{so} \subseteq (\mathsf{po} \cup \mathsf{com})^+$ is the *synchronisation order*; and
- $\mathsf{lhb} \subseteq E \times E$ is the *local-happens-before* relation, defined as a prefix-finite transitive relation extending program and synchronisation orders ($\mathsf{po} \cup \mathsf{so} \subseteq \mathsf{lhb}$), such that po; lhb is irreflexive.

A *concurrent library* L is a tuple $\langle \mathcal{M}, \mathcal{M}_c, \mathtt{loc}, \mathcal{G}_c, \mathcal{G}_{wf} \rangle$, where:

- $\langle \mathcal{M}, \mathcal{M}_c, \mathtt{loc} \rangle$ is an interface;
- $\mathcal{G}_c \subseteq \mathcal{G}_L$ denotes the set of *consistent executions*; it is required to be *monotonic* with respect to lhb: if $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb} \rangle \in \mathcal{G}_c$ and $\mathsf{po} \cup \mathsf{so} \subseteq \mathsf{lhb}' \subseteq \mathsf{lhb}$, then $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb}' \rangle \in \mathcal{G}_c$;
- $\mathcal{G}_{wf} \subseteq \mathcal{G}_L$ denotes the set of *well-formed executions*.

Two libraries $\langle \mathcal{M}^1, \mathcal{M}^1_c, \mathtt{loc}^1, \mathcal{G}^1_c, \mathcal{G}^1_{wf} \rangle$, $\langle \mathcal{M}^2, \mathcal{M}^2_c, \mathtt{loc}^2, \mathcal{G}^2_c, \mathcal{G}^2_{wf} \rangle$ are *compatible* if they have disjoint label sets: $\mathcal{M}^1 \cap \mathcal{M}^2 = \emptyset$. A *collection* Λ is a set of pairwise compatible concurrent libraries.

Given a library L, we use the 'L.' prefix to project its components (e.g. $L.\mathcal{M}$). We write $\mathtt{lab}(L)$ for the L labels: $\mathtt{lab}(L) \triangleq L.\mathcal{M}$. In the context of a collection Λ with $L \in \Lambda$, for $E \subseteq \mathsf{Event}$ we define $E_L \triangleq \{e \in E \mid \mathtt{lab}(e) \in \mathtt{lab}(L)\}$; and $E^c \triangleq \{e \in E \mid \exists L \in \Lambda.\ \mathtt{lab}(e) \in L.\mathcal{M}_c\}$; for $e \in E$, we write $\mathtt{lib}(e)$ for its library: $\mathtt{lib}(e){=}L \Leftrightarrow e \in E_L$; we write $\mathtt{loc}(e)$ for $L.\mathtt{loc}(\mathtt{lab}(e))$ when $e \in E_L$;

[5] As discussed in §2 and formalised shortly, the lhb relation of a library L execution corresponds to the restriction of the happens-before relation associated with the overall execution (comprising events of several libraries) to library L events.

and define $E_x \triangleq \{e \in E \mid x \in \mathtt{loc}(e)\}$; lastly, for $\mathsf{r} \subseteq E \times E$, we define $\mathsf{r}_L \triangleq \mathsf{r} \cap (E_L \times E_L)$. Given an execution G, we use the '$G.$' prefix to project its components (e.g. $G.E$). For $G \in \mathcal{G}_L$ with $E' \subseteq G.E$, we write $G|_{E'}$ for $\langle E', \mathsf{po}|_{E'}, \mathsf{com}|_{E'}, \mathsf{so}|_{E'}, \mathsf{lhb}|_{E'}\rangle$; and write G_x for $G|_{G.E \cap E_x}$.

We present several examples of simple well-known libraries. Further examples are given in §5.

Example 1 (RA registers)**.** We define the *release-acquire (RA) library* L^{RA} as follows. The *RA interface* is $\langle \mathcal{M}, \mathcal{M}_{\mathsf{c}}, \mathtt{loc}\rangle$, where $\mathcal{M}_{\mathsf{c}} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathcal{M}_{\mathsf{c}}^x$ with $\mathcal{M}_{\mathsf{c}}^x \triangleq \{\mathtt{alloc}(x,0)\}$; $\mathcal{M} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathcal{M}^x$ with $\mathcal{M}^x \triangleq \mathcal{M}_{\mathsf{c}}^x \cup \{\mathtt{load}(x,v), \mathtt{store}(x,v) \mid v \in \mathsf{Val}\}$; and $\forall l \in \mathcal{M}^x.\ \mathtt{loc}(l) = \{x\}$. We then define:

$$\mathcal{R}^{x,v} \triangleq \{e \mid \mathtt{lab}(e)=\mathtt{load}(x,v)\} \qquad \mathcal{R}^x \triangleq \bigcup_{v \in \mathsf{Val}} \mathcal{R}^{x,v} \qquad \text{read events}$$

$$\mathcal{W}^{x,v} \triangleq \{e \mid \mathtt{lab}(e)=\mathtt{store}(x,v) \vee (v{=}0 \wedge \mathtt{lab}(e) \in \mathcal{M}_{\mathsf{c}}^x)\} \qquad \mathcal{W}^x \triangleq \bigcup_{v \in \mathsf{Val}} \mathcal{W}^{x,v} \qquad \text{write events}$$

A tuple $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb}\rangle$ is *RA-consistent on* x if $\mathsf{com} \subseteq \bigcup_{v \in \mathsf{Val}} \mathcal{W}^{x,v} \times \mathcal{R}^{x,v}$, $\mathit{rng}(\mathsf{com}) = E \cap \mathcal{R}^x$, $\mathsf{so} = \mathsf{com}$, and there exists a total order mo on $E \cap \mathcal{W}^x$ such that $\mathsf{lhb} \cup \mathsf{com} \cup \mathsf{mo} \cup (\mathsf{com}^{-1}; \mathsf{mo})$ is acyclic. Given the *RA execution set* $\mathcal{G}_{L^{\mathsf{RA}}}$, let $\mathcal{G}_{\mathsf{c}} \triangleq \{G \in \mathcal{G}_{L^{\mathsf{RA}}} \mid \forall x.\ G_x \text{ is RA-consistent on } x\}$ denote the set of *RA-consistent executions*. The *RA library* is the tuple $L^{\mathsf{RA}} \triangleq \langle \mathcal{M}, \mathcal{M}_{\mathsf{c}}, \mathtt{loc}, \mathcal{G}_{\mathsf{c}}, \mathcal{G}_{L^{\mathsf{RA}}}\rangle$. □

Example 2 (Relaxed registers)**.** The library for a (strong) relaxed register can be defined as in Example 1, with the only difference being $\mathsf{so} = \emptyset$ (instead of $\mathsf{so} = \mathsf{com}$). □

Example 3 (SC memory)**.** Let $\mathcal{M}_{\mathsf{c}}$, $\mathcal{M}$, $\mathtt{loc}$, $\mathcal{R}^x$, $\mathcal{W}^x$, $\mathcal{R}^{x,v}$ and $\mathcal{W}^{x,v}$ be as defined in Example 1. The L^{SC} *interface* is $\langle \mathcal{M}, \mathcal{M}_{\mathsf{c}}, \mathtt{loc}\rangle$. A tuple $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb}\rangle$ is *SC-consistent* if $\mathsf{so} = \mathsf{com} \subseteq \bigcup_{x \in \mathsf{Loc},\, v \in \mathsf{Val}} \mathcal{W}^{x,v} \times \mathcal{R}^{x,v}$; $\mathit{rng}(\mathsf{com}) = E \cap \bigcup_{x \in \mathsf{Loc}} \mathcal{R}^x$; and there exist relations $\{\mathsf{mo}_x\}_{x \in \mathsf{Loc}}$, such that each mo_x is a total order on $E \cap \mathcal{W}^x$, and $\mathsf{lhb} \cup \mathsf{mo} \cup (\mathsf{com}^{-1}; \mathsf{mo})$ is acyclic, where $\mathsf{mo} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{mo}_x$. Let $\mathcal{G}_{\mathsf{c}} \triangleq \{G \in \mathcal{G}_{L^{\mathsf{SC}}} \mid G \text{ is SC-consistent}\}$. The L^{SC} *library for a sequentially consistent memory* is the tuple $L^{\mathsf{SC}} \triangleq \langle \mathcal{M}, \mathcal{M}_{\mathsf{c}}, \mathtt{loc}, \mathcal{G}_{\mathsf{c}}, \mathcal{G}_{L^{\mathsf{SC}}}\rangle$. □

Example 4 (SC fences)**.** We define the *SC fence* library L^{SCF} as follows. The L^{SCF} *interface* is $\langle \mathcal{M}, \emptyset, \mathtt{loc}\rangle$, where $\mathcal{M} \triangleq \{\mathtt{sc\text{-}fence}\}$ and $\mathtt{loc}(\mathtt{sc\text{-}fence}) = \emptyset$. A tuple $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb}\rangle$ is *SCF-consistent* if $\mathsf{so} = \mathsf{com}$ and com is a strict total order on E. Let $\mathcal{G}_{\mathsf{c}} \triangleq \{G \in \mathcal{G}_{L^{\mathsf{SCF}}} \mid G \text{ is SCF-consistent}\}$. The *SC fence library* is the tuple $L^{\mathsf{SCF}} \triangleq \langle \mathcal{M}, \emptyset, \mathtt{loc}, \mathcal{G}_{\mathsf{c}}, \mathcal{G}_{L^{\mathsf{SCF}}}\rangle$. □

Remark 1. In order to keep our presentation simple, in the examples above we formalise different fragments of C11 as *separate* instances of our framework. However, it is possible to formalise the entire C11 specification as a *single* library in our framework. In particular, we can directly port existing C11 specifications (e.g. [Lahav et al. 2017]) to our framework with minimal change, namely by renaming the rf relation as com.

Program Executions. A concurrent program e typically comprises calls to several concurrent libraries constituting a collection. As we demonstrated in §3, we describe the semantics of a given concurrent program e as a set of plain execution graphs by the interpretation function $[\![.]\!]$ presented in Fig. 4. Recall that each plain execution of e is a pair $\langle E, \mathsf{po}\rangle$. We next define the notion of execution graphs, as an extension of plain execution graphs with additional components. More concretely, each execution of e is a tuple $G{=}\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$, where E and po denote the events and the program order as before; and com and so respectively denote the *communication* and *synchronisation* orders.

We also define the notion of execution *prefixes* on the $(G.\mathsf{po} \cup G.\mathsf{com})^+$ order.

Definition 3 (Program executions)**.** A *program execution* is a tuple $G{=}\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$, where:

- $\langle E, \mathsf{po}\rangle$ is a plain execution (see Def. 1);
- $\mathsf{com} \subseteq E \times E$ is the *communication order* relation; and
- $\mathsf{so} \subseteq (\mathsf{po} \cup \mathsf{com})^+$ is the *synchronisation order* relation.

The *happens-before* relation of a program execution is defined as $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{so})^+$.
A program execution $G'{=}\langle E', \mathsf{po}', \mathsf{com}', \mathsf{so}'\rangle$ is a *prefix* of $G{=}\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$ if:
$E' \subseteq E$, $\mathsf{po}' = \mathsf{po}|_{E'}$, $\mathsf{com}' = \mathsf{com}|_{E'}$, $\mathsf{so}' = \mathsf{so}|_{E'}$ and $\mathit{dom}((\mathsf{po} \cup \mathsf{com})^+; [E']) \subseteq E'$
The *executions* of a program $\mathtt{e}$ are defined as $\left\{\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle \mid \exists v.\ \langle\langle v, 0\rangle, \langle E, \mathsf{po}\rangle\rangle \in [\![\mathtt{e}]\!]\right\}$.

For $G{=}\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$, we define $G|_{E'} \triangleq \langle E', \mathsf{po}|_{E'}, \mathsf{com}|_{E'}, \mathsf{so}|_{E'}\rangle$ and $G_L \triangleq \langle E_L, \mathsf{po}_L, \mathsf{com}_L, \mathsf{so}_L, \mathsf{hb}_L\rangle$.

Consistency. The set of plain executions associated with a program is almost unrestricted as there are very few constraints on its components. Such restrictions and thus the permitted behaviours of a program are determined by defining the set of *consistent* executions (Def. 4 below).

Given a collection Λ, a program execution G is Λ-*consistent* if: (1) its nodes and edges are those of the libraries in Λ; (2) G is prefix-finite and has no cycles of a certain shape; and (3) for each library $L \in \Lambda$, restricting the nodes and edges of G to those of L yields a consistent library L execution. In particular, as we allow for benign so cycles, and thus $\mathsf{po} \cup \mathsf{com}$ cycles ($\mathsf{so} \subseteq (\mathsf{po} \cup \mathsf{com})^+$), we cannot simply require that $\mathsf{po} \cup \mathsf{com}$ be acyclic. Instead, similar to the case of lhb above, we require that $\mathsf{po} \cup \mathsf{com}$ be acyclic except for cycles comprising only com edges: $\mathsf{po}; \mathsf{com}^+$ is acyclic.

Definition 4 (Consistency)**.** A program execution $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$ is Λ-*consistent* if:

(1) $E = \bigcup_{L\in\Lambda} E_L$; $\mathsf{com} = \bigcup_{L\in\Lambda} \mathsf{com}_L$; $\mathsf{so} = \bigcup_{L\in\Lambda} \mathsf{so}_L$;
(2) $\mathsf{po}; \mathsf{com}^+$ is acyclic and $(\mathsf{po} \cup \mathsf{com})^+$ is prefix-finite; and
(3) $\forall L \in \Lambda.\ (E_L, \mathsf{po}_L, \mathsf{com}_L, \mathsf{so}_L, \mathsf{hb}_L) \in L.\mathcal{G}_\mathtt{c}$.

A plain execution $\langle E, \mathsf{po}\rangle$ is Λ-*consistent* if $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$ is Λ-consistent for some com and so.
Given a program $\mathtt{e}$, we define $\mathsf{outcomes}_\Lambda(\mathtt{e}) \triangleq \{v \mid \exists G.\ \langle\langle v, 0\rangle, G\rangle \in [\![\mathtt{e}]\!] \wedge G \text{ is } \Lambda\text{-consistent}\}$.

As discussed in §2, prior to per-library validation (consistency), the happens-before relation is first calculated for the overall program execution, and then restricted to the events of each library. This is captured by the hb_L projection in (3) above, where $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{so})^+$ denotes the happens-before relation of the program execution. Per-library consistency is then carried out by checking $(E_L, \mathsf{po}_L, \mathsf{com}_L, \mathsf{so}_L, \mathsf{hb}_L) \in L.\mathcal{G}_\mathtt{c}$ for each $L \in \Lambda$.

Encapsulation. The specification of a concurrent library L and its guarantees are typically subject to certain 'usage conditions'. One such condition is that of *encapsulation*: if a client accesses the locations owned by L outside the purview of its methods, then its guarantees are no longer ensured as the client has broken its 'usage conditions'. We formalise the notion of encapsulation in Def. 5 below. In order to ensure encapsulation of an execution $G = \langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$, we require that (1) the locations owned by different constructors be disjoint; and (2) when $e \in E$ accesses location x, we require that x be owned by the constructor of e. That is, each event is associated with a (unique) constructor c (of the same library) that precedes e in hb order and owns the locations accessed by e. Intuitively, each constructor event c of library L allocates an *instance* of L, and each event e with constructor c denotes a library call on the same instance. The two conditions together ensure that encapsulated executions may not have events of different libraries accessing the same location.

Definition 5 (Encapsulation)**.** An execution $G{=}\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$ is Λ-*encapsulated* if:

(1) for all $c, c' \in E^\mathtt{c}$, if $c \neq c'$, then $\mathtt{loc}(c) \cap \mathtt{loc}(c') = \emptyset$; and
(2) $\forall e \in E \setminus E^\mathtt{c}.\ \mathtt{loc}(e) \neq \emptyset \Rightarrow \exists c \in E^\mathtt{c}.\ \mathtt{lib}(e) = \mathtt{lib}(c) \wedge \mathtt{loc}(e) \subseteq \mathtt{loc}(c) \wedge (c, e) \in \mathsf{hb}$

Remark 2. We note that our treatment of constructors is simplistic in that memory allocation is only performed by constructors, and once a memory block is allocated by a library L constructor, its ownership forever remains with L and cannot be transferred to others. While it is possible to generalise our formalism to facilitate such ownership transfer, our chosen approach simplifies the

definition of library encapsulation. As the problem of encapsulation is orthogonal to that of library specification, we opt for the simpler approach.

Well-formedness. Another typical 'usage condition' stipulated by libraries pertains to the shape of client programs. For instance, in case of a mutex library, it is reasonable to expect that clients acquire the mutex prior to releasing it. Such 'usage conditions' are *local* (i.e. library-specific) and are delineated as part of its specification (the $\mathcal{G}_{\mathsf{wf}}$ component). By contrast, encapsulation is a *global* condition of the entire execution (Def. 3), requiring disjointness amongst libraries. As we describe shortly, we refer to these local and global conditions collectively as *well-formedness* conditions.

Note that execution consistency (Def. 4) does not imply its well-formedness and vice versa. This dichotomy allows us to separate library guarantees from client obligations and to lay blame where it is due. *Consistency* denotes that libraries fulfil their guarantees as described by their specifications; *well-formedness* denotes that clients adhere to their obligations in using the libraries correctly. It is thus reasonable for libraries to guarantee consistency *only* for *well-formed* client programs. If an execution of a non-well-formed program is inconsistent (i.e. the library fails to deliver its guarantees), then the blame lies with the client due to incorrect use. If however an execution of a well-formed program is inconsistent, then the blame lies with the library for failing its guarantees.

We must next formalise well-formed execution. Given a collection Λ, as a first attempt we can describe an execution G as Λ-well-formed if: (1) G is encapsulated; and (2) $G_L \in L.\mathcal{G}_{\mathsf{wf}}$ for $L \in \Lambda$. Requiring all executions of a program to meet this condition is however too *strong*. Consider the following client programs:

$$\texttt{let } x = \mathit{new\text{-}mutex}() \texttt{ in } \{\mathit{lock}(x); \mathtt{e}; \mathit{unlock}(x)\} \| a = \mathit{load}(x) \quad \text{(P1)}$$

$$\texttt{let } x = \mathit{new\text{-}mutex}(); y = \mathit{alloc}() \texttt{ in } \mathit{store}(y, 1); \texttt{if } \mathit{load}(y) == 0 \texttt{ then } a = \mathit{load}(x) \quad \text{(P2)}$$

Observe that *syntactically*, both programs violate encapsulation as location x is both owned by the mutex library and accessed directly via $\mathit{load}(x)$. As such, we may be inclined to deem both programs non-well-formed. However, whilst the $\mathit{load}(x)$ in (P1) is always reachable, the $\mathit{load}(x)$ in (P2) constitutes 'dead code' as the condition of the if statement is never satisfied. In other words, the only executions in which the $\mathit{load}(x)$ in (P2) is reachable, are those in which value 0 is read for y (despite the previous write of 1 to y); i.e. those executions that are *inconsistent*.

We may then be inclined to require an execution G to be well-formed only when G is also consistent. This notion of well-formedness is however too *weak*. More concretely, given an execution G of (P1), as part of the consistency guarantee for register x, we must show that the read event associated with $\mathit{load}(x)$ *reads from* a corresponding write event on x; i.e. the event of $\mathit{load}(x)$ has a suitable incoming com edge. However, assuming that the code in e does not access x, no execution G of (P1) contains a write event on x form which the event of $\mathit{load}(x)$ can read. That is, *all* executions of (P1) are inconsistent. As such, if we require an execution G to be well-formed *only* when G is also consistent, the (P1) would be vacuously well-formed, despite violating encapsulation.

Our notion of well-formedness must thus identify (P1) as non-well-formed, whilst identifying (P2) as well-formed. To this end, we check the well-formedness of an execution G *incrementally* for all its prefixes as follows. Given a prefix G'' of G, let a_{max} denote an event in G that can be added to G'' to grow the prefix. That is, adding a_{max} to G'' yields G' such that G' is also a prefix of G and has a_{max} as a maximal event in $\mathsf{r} \triangleq (G'.\mathsf{po} \cup G'.\mathsf{com})^+$. Given a collection Λ, for an execution G to be Λ-well-formed, we require that for each prefix G'' of G, if G'' is consistent, then for each a_{max} and G' constructed as above: (1) G' is encapsulated; and (2) $G'_L \in L.\mathcal{G}_{\mathsf{wf}}$ for all $L \in \Lambda$.

Recall that our library executions allow benign so cycles and thus potentially com cycles ($\mathsf{so} \subseteq (\mathsf{po} \cup \mathsf{com})^+$). When an execution contains com cycles, it is not possible to identify a *single* maximal event a_{max} in $\mathsf{r} \triangleq (G'.\mathsf{po} \cup G'.\mathsf{com})^+$. To remedy this, at each step we also allow the addition of a *set*

of maximal events that form a com cycle. That is, at each step we either add a single event a_{max} that is maximal in r ($\mathsf{r}(a_{max}) = \emptyset$); or we add a set of events A forming a com cycle ($A \times A \subseteq G'.\mathsf{com}^+$), with A events being maximal in r with respect to all other events not in A ($\mathsf{r}(A) \setminus A = \emptyset$).

Note that checking well-formedness for all prefixes of G ensures that G itself is also well-formed. Moreover, starting with a consistent prefix G'' ensures that our definition is not too strong in that executions are not considered non-well-formed due to inconsistency, as in (P2). For all executions G of (P2), any prefix of G that contains the event associated with $\mathit{load}(x)$, also contains the event of $\mathit{load}(y)$ reading zero, and is thus inconsistent. As such, these inconsistent prefixes will not be considered and G is deemed well-formed. Conversely, checking the well-formedness of prefixes after each step ensures that our definition is not too weak in that we do not ignore executions in which non-well-formedness causes inconsistency, as in (P1). For all executions G of (P1), any prefix G'' of G without the event of $\mathit{load}(x)$ is consistent. However, adding the event of $\mathit{load}(x)$ to G'' yields a non-well-formed execution, thus rendering G non-well-formed as required.

Definition 6 (Well-formedness)**.** An execution G is Λ-*well-formed* if for all prefixes $G' = \langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}\rangle$ of G and for all $A \in \max(G')$, if $G'|_{E\setminus A}$ is Λ-consistent, then:

(1) G' is Λ-encapsulated (Def. 5); and (2) $\forall L \in \Lambda.\ G'_L \in L.\mathcal{G}_{\mathsf{wf}}$

where $\max(G') \triangleq \left\{\{a\} \subseteq E \;\middle|\; \mathsf{r}(a) = \emptyset\right\} \cup \left\{A \subseteq E \;\middle|\; \mathsf{r}(A) \setminus A = \emptyset \wedge A \times A \subseteq \mathsf{com}^+\right\}$ and $\mathsf{r} \triangleq (\mathsf{po} \cup \mathsf{com})^+$.
A program $\mathtt{e}$ is Λ-*well-formed*, written $\mathsf{wf}(\mathtt{e})$, if all its executions are Λ-well-formed.

4.2 Verifying Library Implementations

Library Implementations. Recall that we are interested not only in *specifying* concurrent libraries, but also in *verifying* their implementations. To this end, we formally define the notion of a library implementation in Def. 8. An implementation I of library L is a function that maps L methods to their implementation code. The $I(m) = (x_1, \cdots, x_n, \mathtt{e})$ entry corresponds to the L method m of arity n, where $x_1, \cdots, x_n$ are placeholder variables denoting the method arguments and are used in the implementation body $\mathtt{e}$. As such, we require that for each method m: (1) the domain of I consists of all L methods with the appropriate arities; when $I(m) = (\vec{x}, \mathtt{e})$ then (2) $\mathtt{e}$ contains no free program variables other than those of $\vec{x}$; and (3) $\mathtt{e}$ is encapsulated in that it only accesses locations allocated in $\mathtt{e}$ or those passed as arguments ($\vec{x}$).

To understand this last condition, recall the (SB-lib) program from §1, and consider a (malicious) queue implementation that accesses and mutates location s which is owned by the stack library. To rule out such malicious behaviour, we first define the notion of *location maps*, relating specification and implementation locations. For instance, given a queue at location q and an enqueue event e with label $\mathsf{enq}(q, v)$, the only (specification) location accessed by e is q: $\mathtt{loc}(e)=\{q\}$. However, the implementation of a queue at q may allocate and access several locations. For example, the Herlihy-Wing implementation in Fig. 2 represents a queue as an infinite array at q, and thus the locations accessed by the implementation are in the set $\{q+i \mid i \in \mathbb{N}\}$. A location map f captures this correspondence. In particular, a function $f : \mathsf{Loc} \rightharpoonup \mathcal{P}(\mathsf{Loc})$ is a location map of a library L against an implementation I, if given a label $l{=}m(\vec{v}, v)$ of L: (a) if m is not a constructor method, then the implementation of m only accesses either the locations it allocates itself internally, or those passed as method arguments, which are included in the locations of m when mapped via f; and (b) if m is a constructor method, then its locations, when mapped via f, are allocated by the implementation; and the implementation of m itself only accesses the locations it allocates.

Translation. We next formalise the notion of *translation*, and given a program $\mathtt{e}$, we write $[\![\mathtt{e}]\!]_{L:I}$ to denote the program obtained from $\mathtt{e}$ by replacing every call to a library L method with its implementation in I. It is defined by straightforward induction on the $\mathtt{e}$ structure.

Definition 7 (Translations)**.** Given a function I : Method $\rightharpoonup$ (Var* × Exp), a program $\mathtt{e}$ and a library L, the *I-translation of* $\mathtt{e}$ *for* L, written $[\![\mathtt{e}]\!]_{L:I}$, is defined as follows, where $[x/y]$ denotes capture-avoiding substitution of x for y:

$$[\![v]\!]_{L:I} \triangleq v \qquad [\![x]\!]_{L:I} \triangleq x \qquad [\![\mathtt{loop}\ \mathtt{e}]\!]_{L:I} \triangleq \mathtt{loop}\ [\![\mathtt{e}]\!]_{L:I} \qquad [\![\mathtt{break}_n\ x]\!]_{L:I} \triangleq \mathtt{break}_n\ x$$

$$[\![m(x_1,\cdots,x_n)]\!]_{L:I} \triangleq \begin{cases} \mathtt{e}[x_1/y_1,\cdots,x_n/y_n] & \text{if } \exists y_1,\cdots,y_n,\mathtt{e}.\ I(m)=(y_1,\cdots,y_n,\mathtt{e}) \\ m(x_1,\cdots,x_n) & \text{otherwise} \end{cases}$$

$$[\![\mathtt{e}_1 \| \mathtt{e}_2]\!]_{L:I} \triangleq [\![\mathtt{e}_1]\!]_{L:I} \,\|\, [\![\mathtt{e}_2]\!]_{L:I} \qquad [\![\mathtt{let}\ x = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2]\!]_{L:I} \triangleq \mathtt{let}\ x = [\![\mathtt{e}_1]\!]_{L:I}\ \mathtt{in}\ [\![\mathtt{e}_2]\!]_{L:I}$$

$$[\![\mathtt{if}\ x\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2]\!]_{L:I} \triangleq \mathtt{if}\ x\ \mathtt{then}\ [\![\mathtt{e}_1]\!]_{L:I}\ \mathtt{else}\ [\![\mathtt{e}_2]\!]_{L:I}$$

Definition 8 (Implementations)**.** Given a library L, an *implementation* of L is a function, I : Method $\rightharpoonup$ (Var* × Exp), such that for all $m \in$ Method:

(1) $\exists v, v_1, \cdots, v_n.\ m(v_1,\cdots,v_n,v) \in \mathsf{lab}(L) \Leftrightarrow \exists \mathtt{e}, x_1, \cdots, x_n.\ I(m) = (x_1,\cdots,x_n,\mathtt{e})$
(2) $\forall \vec{x}, \mathtt{e}.\ I(m) = (\vec{x},\mathtt{e}) \Rightarrow \mathsf{fv}(\mathtt{e}) \subseteq \vec{x}$
(3) there exists f:Loc $\rightharpoonup \mathcal{P}$(Loc) such that for all l=$m(\vec{v},v) \in L.\mathcal{M}$, $\langle -, \langle E, \mathrm{po}\rangle\rangle \in [\![[\![m(\vec{v})]\!]_{L:I}]\!]$, $e \in E$, and C=$\{c \in E^{\mathsf{c}} \mid (c,e) \in \mathrm{po}\}$ the following hold, where $f(S)$=$\bigcup_{l \in S} f(l)$ for $S \subseteq$ Loc:
   (a) $l \in L.\mathcal{M} \setminus L.\mathcal{M}_{\mathsf{c}} \Rightarrow \mathtt{loc}(e) \subseteq \bigcup_{c \in C} \mathtt{loc}(c) \cup f(\mathtt{loc}(l))$; and
   (b) $l \in L.\mathcal{M}_{\mathsf{c}} \Rightarrow f(\mathtt{loc}(l)) \subseteq \bigcup_{c \in E^{\mathsf{c}}} \mathtt{loc}(c) \wedge \mathtt{loc}(e) \subseteq \bigcup_{c \in C} \mathtt{loc}(c)$

where fv(e) denotes the free program variables of e (those outside let-bindings).

Implementation Soundness. We next formalise what it means for a library implementation to be *sound* with respect to its specification. Intuitively, an implementation I of library L is sound if for all well-formed programs e, replacing the L calls in e with their implementations in I does not introduce additional behaviours. That is, for all client programs e, the outcomes of $[\![\mathtt{e}]\!]_{L:I}$ are included in those of e. Recall that by focusing on well-formed clients only, we can assign blame duly. If implementation soundness cannot be established for well-formed clients, then the implementation is at fault as it fails to deliver the specified guarantees. By contrast, if implementation soundness cannot be established for non-well-formed clients, then the client is at fault through incorrect library use. As such, soundness of non-well-formed clients is not a proof obligation.

Definition 9 (Soundness)**.** An implementation I of library L is *sound* if for all collections Λ and $\Lambda' = \Lambda \uplus \{L\}$, and all Λ'-well-formed programs e: $\mathsf{outcomes}_\Lambda([\![\mathtt{e}]\!]_{L:I}) \subseteq \mathsf{outcomes}_{\Lambda'}(\mathtt{e})$.

Verifying Implementations. To show an implementation I of library L is sound, we must show that given a well-formed program e with calls to libraries in Λ'=$\Lambda \uplus \{L\}$, and a Λ-consistent G of $[\![\mathtt{e}]\!]_{L:I}$, we can construct a Λ'-consistent G' of e with the same outcome. To show that a candidate G' is Λ'-consistent, we must show (see Def. 4) that: (i) $G'_L \in L.\mathcal{G}_{\mathsf{c}}$; and (ii) $G'_{L'} \in L'.\mathcal{G}_{\mathsf{c}}$ *for all* $L' \in \Lambda$. However, since the implementation execution G is consistent, and intuitively G and G' must be the same up to library L events, if G' has a certain *shape*, then (ii) follows immediately. That is, if I is *locally sound on* L (for all G there exists G' of a certain shape such that (i) holds), then I is sound ((ii) also holds). This allows for a *compositional* proof by showing consistency *only* for the library implemented (L). We thus developed the meta theory for identifying such shape properties, thereby enabling compositional reasoning and reducing proof overhead significantly.

The desired shape property requires that G and G' be the same up to library L events: if we exclude L events from G' and their corresponding implementation events from G, then the remaining graphs must be identical. To capture this, we define the notion of an *abstraction function*, relating the G events in the implementation body of each L method to the corresponding L event in G'. Given an implementation I of library L, and plain executions G'=$\langle E', \mathrm{po}'\rangle$ (of the specification) and G=$\langle E, \mathrm{po}\rangle$ (of the implementation), a function $f : E \to E'$ *abstracts* G to G' on (L, I) if: (1) G' only

comprises library L events ($G'.E = G'.E_L$); f is surjective (onto); po' is a lifting of po via f; and (2) for all library L events $e' \in E'$ with label $m(\vec{v}, v)$ (e' is an m call in G' returning v), executing the implementation of m also returns v, together with $G_{e'}$, where $G_{e'}$ denotes limiting G to the implementation events associated with e' (those mapped on to e' via f).

Definition 10. Given an implementation I of a library L and plain executions $G{=}\langle E, \mathrm{po}\rangle$ and $G'{=}\langle E', \mathrm{po}'\rangle$, a function $f : E \to E'$ *abstracts* G *to* G' *on* (L, I), written $\mathrm{abs}_{L,I}(f, G, G')$, if:

(1) $E'_L = E'$; f is surjective; $\mathrm{po}' = \left\{(f(a), f(b)) \,\middle|\, (a, b) \in \mathrm{po} \wedge f(a) \neq f(b)\right\}$; and
(2) $\forall e' \in E'.\ \forall m, \vec{v}, v.\ \mathtt{lab}(e') = m(\vec{v}, v) \Rightarrow \langle\langle v, -\rangle, G|_{\{e \mid f(e)=e'\}}\rangle \in [\![\lfloor m(\vec{v}) \rfloor_{L:I}]\!]$.

We next formalise the notion of *local soundness*. An implementation of library L is locally sound on L if given a consistent and well-formed program execution $G{=}\langle E, \mathrm{po}, \mathrm{com}, \mathrm{so}\rangle$ of the implementation, and a function f abstracting $\langle E, \mathrm{po}\rangle$ to $\langle E', \mathrm{po}'\rangle$, then there exists com', so' such that $G'{=}\langle E, \mathrm{po}', \mathrm{com}', \mathrm{so}', \mathrm{lhb}'\rangle \in L.\mathcal{G}_{\mathrm{c}}$ (G' is a consistent library L execution). The (specification) lhb' relation denotes the lifting of the (implementation) $G.\mathrm{hb}$ relation (defined in Def. 11).

Definition 11. An implementation I is *locally sound* on L if for all $\Lambda, f, G{=}\langle E, \mathrm{po}, \mathrm{com}, \mathrm{so}\rangle, E', \mathrm{po}'$:

if $\quad G$ is Λ-consistent and Λ-well-formed $\wedge\ \mathrm{abs}_{L,I}(f, \langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$
then $\quad \exists \mathrm{com}', \mathrm{so}'.\ \langle E', \mathrm{po}', \mathrm{com}', \mathrm{so}', \mathrm{lhb}'\rangle \in L.\mathcal{G}_{\mathrm{c}}$
with $\quad \mathrm{lhb}' \triangleq \left(\{(a', b') \mid \forall a, b.\ f(a) = a' \wedge f(b) = b' \Rightarrow (a, b) \in G.\mathrm{hb}\} \cup \mathrm{so}'\right)^+$

We next formulate the *modularity* theorem below, stating that to show the soundness of an implementation I of library L, it suffices to show its local soundness on L. That is, the local soundness of I ensures its soundness 'for free', thus streamlining the soundness proof significantly.

Theorem 1 (Modularity). *If I is locally sound on L, then I is a sound implementation of L.*

Proof. A slightly simplified variant of this theorem is mechanised in Coq and is available as auxiliary material [Raad et al. 2018]. □

5 SPECIFYING CONCURRENT LIBRARIES IN OUR FRAMEWORK

We present several examples of concurrent libraries specified in our framework. In the technical appendix [Raad et al. 2018], we present additional library specifications, including a set library specification and two (one strong and one weak) specifications for a reader-writer lock library.

5.1 Mutual Exclusion Lock (Mutex) Library Specification

We consider a mutual exclusion (mutex) lock library with three methods: *new-mutex*( ), for constructing a new mutex; *lock*(x) and *unlock*(x), for acquiring and releasing the mutex at x, respectively. The *mutex interface* is $\langle \mathcal{M}^{\mathsf{MX}}, \mathcal{M}_{\mathrm{c}}^{\mathsf{MX}}, \mathtt{loc}^{\mathsf{MX}}\rangle$, where $\mathcal{M}_{\mathrm{c}}^{\mathsf{MX}} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathcal{M}_{\mathrm{c}}^{x}$ with $\mathcal{M}_{\mathrm{c}}^{x} \triangleq \{\mathtt{new\text{-}mutex}(x)\}$; $\mathcal{M}^{\mathsf{MX}} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathcal{M}^{x}$ with $\mathcal{M}^{x} \triangleq \mathcal{M}_{\mathrm{c}}^{x} \cup \{\mathtt{lock}(x), \mathtt{unlock}(x)\}$; and $\forall l \in \mathcal{M}^{x}.\ \mathtt{loc}^{\mathsf{MX}}(l) = \{x\}$.
For a mutex lock at location x, we then define the following event sets:

$$\mathcal{C}^x \triangleq \left\{e \,\middle|\, \mathtt{lab}(e){=}\mathtt{new\text{-}mutex}(x)\right\} \quad \mathcal{L}^x \triangleq \left\{e \,\middle|\, \mathtt{lab}(e){=}\mathtt{lock}(x)\right\} \quad \mathcal{U}^x \triangleq \{e \mid \mathtt{lab}(e){=}\mathtt{unlock}(x)\}$$

A tuple $\langle E, \mathrm{po}, \mathrm{com}, \mathrm{so}, \mathrm{lhb}\rangle$ is *MX-consistent* on x if:

(1) there is at most one constructor event: $E^{\mathrm{c}} = \emptyset \vee \exists c \in \mathcal{C}^x.\ E^{\mathrm{c}} = \{c\}$;
(2) com *matches* mutex unlock and lock events: $\mathrm{com} \subseteq (\mathcal{U}^x \cup \mathcal{C}^x) \times \mathcal{L}^x$;
(3) each lock is matched by at most one event and vice versa: com, com^{-1} are functional;
(4) all lock events are matched: $E \cap \mathcal{L}^x = \mathit{rng}(\mathrm{com})$; and
(5) every matching edge is synchronising: $\mathrm{so} = \mathrm{com}$.

Intuitively, com describes the order of mutex acquisition. For each $l \in \mathcal{L}^x$ with $(e, l) \in \mathsf{com}$, when $e \in \mathcal{U}^x$ then e denotes the unlock event releasing the mutex immediately before it is acquired by l; when $e \in C^x$ then e denotes the constructor event initialising the mutex, i.e. l corresponds to the very first *lock*(x) call. As such, all lock events are matched by com in a one-to-one fashion.

A tuple $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb}\rangle$ is *MX-well-formed on x* if:

$$\mathsf{min}(\mathsf{po}) \subseteq \mathcal{L}^x \cup C^x \quad \text{and} \quad \mathsf{po}|_{\mathsf{imm}}(E^{\mathsf{c}}) \subseteq \mathcal{L}^x \quad \text{and} \quad [\mathcal{L}^x]; \mathsf{po}|_{\mathsf{imm}} = \mathsf{po}|_{\mathsf{imm}}; [\mathcal{U}^x]$$

where $\mathsf{min}(\mathsf{po})$ denotes the set of po-minimal events in E. Intuitively, well-formedness requires that the first call in each thread be to either *new-mutex*( ) or *lock*(x); a *new-mutex*( ) call be immediately followed (in po) by a *lock*(x) call; and each *unlock*(x) call be immediately preceded (in po) by a *lock*(x) call and vice versa.

Definition 12 (Mutex library)**.** The *mutex library* is $L^{\mathsf{MX}} \triangleq \langle \mathcal{M}^{\mathsf{MX}}, \mathcal{M}_{\mathsf{c}}^{\mathsf{MX}}, \mathtt{loc}^{\mathsf{MX}}, \mathcal{G}_{\mathsf{c}}^{\mathsf{MX}}, \mathcal{G}_{\mathsf{wf}}^{\mathsf{MX}}\rangle$, where $\mathcal{G}_{\mathsf{c}}^{\mathsf{MX}} \triangleq \left\{G \in \mathcal{G}_{L^{\mathsf{MX}}} \middle| \forall x.\, G_x \text{ MX-consistent on } x\right\}$ and $\mathcal{G}_{\mathsf{wf}}^{\mathsf{MX}} \triangleq \left\{G \in \mathcal{G}_{L^{\mathsf{MX}}} \middle| \forall x.\, G_x \text{ MX-well-formed on } x\right\}$.

When the mutex on x is used in a well-formed manner by the clients, then the mutex guarantees the desired mutual exclusion properties. That is, when an execution G_x is both consistent and well-formed on x, then lhb constitutes a strict *total* order on all mutex events in $G_x.E$ such that when $G_x.E$ is non-empty then enumerating $G_x.E$ according to lhb corresponds to a *prefix* of the regular expression $C^x.(\mathcal{L}^x.\mathcal{U}^x)^*$. Each $(\mathcal{L}^x.\mathcal{U}^x)$ interval describes a critical section, guaranteeing mutual exclusion. This is formalised in Thm. 2 below. In particular, the second property states that when thread τ contains a lock event l, then each mutex event e of another thread proceeding l in lhb, (i.e. $(l, e) \in \mathsf{lhb} \setminus \mathsf{po}$), is interleaved by an unlock event u by the same thread τ (i.e. $(l, u) \in \mathsf{po}$ and $(u, e) \in \mathsf{lhb}$). In other words, the (l, u) interval describes a mutually-excluded critical section.

Theorem 2. *For all x and G, if $G \in \mathcal{G}_{\mathsf{c}}^{MX} \cap \mathcal{G}_{\mathsf{wf}}^{MX}$, then $G_x.\mathsf{lhb}$ is a strict total order on $G_x.E$ such that:*

- $G_x.\mathsf{lhb}|_{imm} \subseteq (C^x \times \mathcal{L}^x) \cup (\mathcal{L}^x \times \mathcal{U}^x) \cup (\mathcal{U}^x \times \mathcal{L}^x)$*; and*
- $[\mathcal{L}^x]; (G_x.\mathsf{lhb} \setminus G_x.\mathsf{po}); [\mathcal{L}^x \cup \mathcal{U}^x] \subseteq G_x.\mathsf{po}; [\mathcal{U}^x]; G_x.\mathsf{lhb}$.

Proof. The full proof is given in the technical appendix [Raad et al. 2018]. □

5.2 Exchanger Library Specification

We consider an exchanger library with two methods: *new-exchanger*( ), for constructing a new exchanger; and *exchange*(g, v), for exchanging value v. Recall from §2 that it is not possible to develop a useful sequential specification for exchangers in the linearisability style [Hemed et al. 2015]. The authors in [Hemed et al. 2015] present an exchanger specification under SC by generalising the notion of linearisability. By contrast, we develop an exchanger specification that is agnostic to the underlying memory model and is thus usable under both SC and WMC.

We define the *exchanger interface* as $\langle \mathcal{M}^{\mathsf{X}}, \mathcal{M}_{\mathsf{c}}^{\mathsf{X}}, \mathtt{loc}^{\mathsf{X}}\rangle$, where $\mathcal{M}_{\mathsf{c}}^{\mathsf{X}} \triangleq \bigcup_{g \in \mathsf{Loc}} \mathcal{M}_{\mathsf{c}}^{g}$ with $\mathcal{M}_{\mathsf{c}}^{g} \triangleq \{\mathtt{new\text{-}exchanger}(g)\}$; $\mathcal{M}^{\mathsf{X}} \triangleq \bigcup_{g \in \mathsf{Loc}} \mathcal{M}^{g}$ with $\mathcal{M}^{g} \triangleq \mathcal{M}_{\mathsf{c}}^{g} \cup \left\{\mathtt{exchange}(g, v_1, v_2) \,\middle|\, v_1, v_2 \in \mathsf{Val}\right\}$; and $\forall l \in \mathcal{M}^{g}.\ \mathtt{loc}^{\mathsf{MX}}(l) = \{g\}$. For an exchanger at location g, we define the following sets of events:

$$C^{g} \triangleq \{e \mid \mathtt{lab}(e) = \mathtt{new\text{-}exchanger}(g)\} \qquad \mathcal{X}^{g, v_1, v_2} \triangleq \left\{e \,\middle|\, \mathtt{lab}(e) = \mathtt{exchange}(g, v_1, v_2) \wedge v_1 \neq \bot\right\}$$

Let $\mathcal{X}^{g} \triangleq \bigcup_{v_1, v_2 \in \mathsf{Val}} \mathcal{X}^{g, v_1, v_2}$. A tuple $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb}\rangle$ is *exchanger-consistent* on g if:

(1) there is at most one constructor event: $E^{\mathsf{c}} = \emptyset \vee \exists c \in C^{g}.\ E^{\mathsf{c}} = \{c\}$;
(2.a) com is symmetric and irreflexive;
(2.b) com relates matching events: $\mathsf{com} \subseteq \bigcup_{v_1, v_2 \in \mathsf{Val}} \mathcal{X}^{g, v_1, v_2} \times \mathcal{X}^{g, v_2, v_1}$;
(3) every exchange event is matched by at most one exchange event: com is functional;
(4) every unmatched exchange returns $\bot$: $E \cap \mathcal{X}^{g} \setminus \mathit{dom}(\mathsf{com}) \subseteq \bigcup_{v \in \mathsf{Val}} \mathcal{X}^{g, v, \bot}$; and

(5) every matching edge is synchronising: so = com.

Given an event e with label $\mathtt{exchange}(g, v_1, v_2)$, if $v_2{=}\bot$ then e denotes a failed exchange. Moreover, only valid values can be offered for exchange, i.e. $v_1{\neq}\bot$. Intuitively, $(e_1, e_2) \in \mathsf{com}$ denotes that e_1 and e_2 successfully exchange their values. As such, com is defined to be *symmetric* ($\mathsf{com} = \mathsf{com}^{-1}$) to capture the bidirectional information flow between e_1 and e_2.

Definition 13 (Exchanger library)**.** The *exchanger library* is $L^{\mathsf{X}} \triangleq \langle \mathcal{M}^{\mathsf{X}}, \mathcal{M}_{\mathsf{c}}^{\mathsf{X}}, \mathtt{loc}^{\mathsf{X}}, \mathcal{G}_{\mathsf{c}}^{\mathsf{X}}, \mathcal{G}_{L^{\mathsf{X}}} \rangle$, where $\mathcal{G}_{\mathsf{c}}^{\mathsf{X}} \triangleq \left\{ G \in \mathcal{G}_{\mathsf{c}}^{\mathsf{X}} \;\middle|\; \forall g.\ G_g \text{ exchanger-consistent on } g \right\}$.

5.3 Queue Library Specification

We consider a queue library with three methods: *new-queue*( ), for constructing a new queue; *enq*(q, v) for enqueuing v to the queue at q; and *deq*(q) for dequeuing a value from the queue at q.

In what follows, we first present a *strong* queue specification, which requires the existence of a total order on the set of queue events, determining the execution order. As our first attempt, we present our strong specification in the style of linearisability specifications: we sequentially enumerate the queue events in accordance with to and produce a *history*; we then ensure that the result is a legal queue history, i.e. it satisfies the first-in-first-out (FIFO) paradigm.

As discussed in §2, verifying library implementations under linearisability-style specifications is not straightforward as it requires the construction of the existentially quantified to order. We thus develop an alternative *equivalent* specification for queues that forgoes the existentially quantified to. In particular, we demonstrate that the lack of certain cycles in an execution ensures the existence of a to order and vice versa. Consequently, in order to establish consistency, it suffices to ascertain the absence of such cycles in the execution graph. By reducing the consistency problem to searching for cycles, we can employ and adapt existing verification techniques in the literature, such as those based on model-checking [Kokologiannakis et al. 2018].

Lastly, as we demonstrate in §6, our first queue specification (in both styles) is *too* strong. In particular, as discussed in §2, this strong specification renders the weak implementation of the Herlihy-Wing queue [Herlihy and Wing 1990] in Fig. 2 unsound. We thus develop a weaker queue specification that does not guarantee the existence of a total execution order.

We define the *queue interface* as the tuple $\langle \mathcal{M}^{\mathsf{Q}}, \mathcal{M}_{\mathsf{c}}^{\mathsf{Q}}, \mathtt{loc}^{\mathsf{Q}} \rangle$, where $\mathcal{M}_{\mathsf{c}}^{\mathsf{Q}} \triangleq \bigcup_{q \in \mathsf{Loc}} \mathcal{M}_{\mathsf{c}}^{q}$ with $\mathcal{M}_{\mathsf{c}}^{q} \triangleq \{\mathtt{new\text{-}queue}(q)\}$; $\mathcal{M}^{\mathsf{Q}} \triangleq \bigcup_{q \in \mathsf{Loc}} \mathcal{M}^{q}$ with $\mathcal{M}^{q} \triangleq \mathcal{M}_{\mathsf{c}}^{q} \cup \left\{ \mathsf{enq}(q, v), \mathsf{deq}(q, v) \;\middle|\; v \in \mathsf{Val} \right\}$; and $\forall l \in \mathcal{M}^{q}.\ \mathtt{loc}^{\mathsf{MX}}(l){=}\{q\}$. For a queue at location q, we define the following sets of events:

$$\mathcal{C}^{q} \triangleq \{e \mid \mathtt{lab}(e){=}\mathtt{new\text{-}queue}(q)\} \quad \mathcal{E}^{q,v} \triangleq \{e \mid \mathtt{lab}(e){=}\mathsf{enq}(q, v) \wedge v{\neq}\bot\} \quad \mathcal{D}^{q,v} \triangleq \{e \mid \mathtt{lab}(e){=}\mathsf{deq}(q, v)\}$$

Let $\mathcal{E}^{q} \triangleq \bigcup_{v \in \mathsf{Val}} \mathcal{E}^{q,v}$ and $\mathcal{D}^{q} \triangleq \bigcup_{v \in \mathsf{Val}} \mathcal{D}^{q,v}$.

A tuple $(E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb})$ is *strongly queue-consistent* on q if:

(1) there is at most one constructor event: $E^{\mathsf{c}} = \emptyset \vee \exists c \in \mathcal{C}^{q}.\ E^{\mathsf{c}} = \{c\}$;
(2) com relates matching enqueue and dequeue events: $\mathsf{com} \subseteq \bigcup_{v \in \mathsf{Val}} \mathcal{E}^{q,v} \times \mathcal{D}^{q,v}$;
(3) every enqueue is matched by at most one dequeue and vice versa: com, com^{-1} are functional;
(4) every unmatched dequeue returns $\bot$: $E \cap \mathcal{D}^{q} \setminus \mathit{rng}(\mathsf{com}) \subseteq \mathcal{D}^{q,\bot}$
(5) dequeues with previous unmatched enqueues cannot return $\bot$: $[\mathcal{E}^{q} \setminus \mathit{dom}(\mathsf{com})]; \mathsf{lhb}; [\mathcal{D}^{q,\bot}]{=}\emptyset$;
(6) every matching edge is synchronising: so = com; and
(7) there exists a total order to on $E \setminus \mathcal{C}^{q}$ such that: (i) $\mathsf{lhb} \subseteq \mathsf{to}$; and (ii) enumerating $E \setminus \mathcal{C}^{q}$ according to to yields a sequence H where $\mathsf{fifo}(\epsilon, H)$ holds, with:

$$\begin{aligned} \mathsf{fifo}(h, H) \overset{\text{def}}{\Leftrightarrow} & H = \epsilon \vee (\exists e, H'.\ e \in \mathcal{E}^{q} \wedge H{=}e; H' \wedge \mathsf{fifo}(h; e, H')) \\ & \vee (\exists e, d, h', H'.\ h{=}e; h' \wedge H{=}d; H' \wedge (e, d) \in \mathsf{com} \wedge \mathsf{fifo}(h', H')) \\ & \vee (\exists d, H'.\ h{=}\epsilon \wedge H{=}d; H' \wedge d \in \mathcal{D}^{q,\bot} \wedge \mathsf{fifo}(h, H')) \end{aligned}$$

Intuitively, $(e, d) \in \mathsf{com}$ denotes that d dequeues a value enqueued by e. An event labelled $\mathsf{deq}(q, \bot)$ denotes a failed dequeue (when the queue is empty). As such, only valid values can be enqueued: for all events labelled $\mathsf{enq}(q, v)$ we have $v \neq \bot$. Lastly, as the name suggests (7) ensures that sequential enumeration of queue events by to produces a history that respects the FIFO property.

Definition 14 (Strong queue library)**.** The *strong queue library* is $L^{\mathrm{SQ}} \triangleq \langle \mathcal{M}^{\mathrm{Q}}, \mathcal{M}_{\mathrm{c}}^{\mathrm{Q}}, \mathtt{loc}^{\mathrm{Q}}, \mathcal{G}_{\mathrm{c}}^{\mathrm{SQ}}, \mathcal{G}_{L^{\mathrm{Q}}} \rangle$ where $\mathcal{G}_{\mathrm{c}}^{\mathrm{SQ}} \triangleq \left\{ G \in \mathcal{G}_{L^{\mathrm{Q}}} \;\middle|\; \forall q.\ G_q \text{ strongly queue-consistent on } q \right\}$.

Alternative Strong Specification for Queues. We next demonstrate how we move away from the existentially quantified to order and arrive at an alternative strong specification for queues that guarantees the existence of to by requiring the absence of certain cycles. Note that when two enqueue events are ordered by lhb, their matching dequeues must be accordingly ordered by a candidate to to ensure the FIFO property; that is, (i) $\mathsf{com}^{-1}; \mathsf{lhb}; \mathsf{com} \subseteq \mathsf{to}$. Dually, when two dequeues events are ordered by lhb, their matching enqueues must be accordingly ordered: (ii) $\mathsf{com}; \mathsf{lhb}; \mathsf{com}^{-1} \subseteq \mathsf{to}$. Moreover, observe that a candidate to must satisfy: (iii) $\mathsf{lhb} \subseteq \mathsf{to}$; (iv) $\mathsf{to}; \mathsf{to} \subseteq \mathsf{to}$ (transitivity); and (v) to is irreflexive. By iteratively replacing the left-hand side of (i), (ii) and (iii) for to in the left-hand side of (iv), and subsequently checking the irreflexivity of to as per (v), we arrive at a fixed point. In particular, thus checking the irreflexivity of to prohibits all cycles comprising an *equal number of* A *and* B *edges*, where $\mathsf{A} \triangleq \mathsf{com}^{-1}; \mathsf{lhb}$ and $\mathsf{B} \triangleq \mathsf{com}; \mathsf{lhb}$.

We write $\mathsf{C}^{i,j}$ for a path comprising i edges of A and j edges of B. In Thm. 3 below we show that when $\mathsf{C}^{n,n}$ is irreflexive for $n \in \mathbb{N}^+$, then we can construct a total order to. That is, an execution is consistent if it satisfies (1)-(6) above, and $\mathsf{C}^{n,n}$ is irreflexive for $n \in \mathbb{N}^+$ (in lieu of (7)).

Theorem 3. *Given a relation* r*, let* r^0 *denote the identity relation* id*, and let* $\mathsf{r}^{n+1} \triangleq \mathsf{r}; \mathsf{r}^n$*, when* $n \geq 0$*. For a given tuple* $(E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb})$*, condition* (7) *above holds iff* $\mathsf{C}^{n,n}$ *is irreflexive for all* $n \in \mathbb{N}^+$*, where for all* $i, j \in \mathbb{N}^+$ *and* $k \in \mathbb{N}$*:*

$$\mathsf{C}^{i,j} \triangleq (\mathsf{A}; \mathsf{C}^{i-1,j}) \cup (\mathsf{B}; \mathsf{C}^{i,j-1}) \qquad \mathsf{A} \triangleq \mathsf{com}^{-1}; \mathsf{lhb} \qquad \mathsf{B} \triangleq \mathsf{com}; \mathsf{lhb} \qquad \mathsf{C}^{k,0} \triangleq \mathsf{A}^k \qquad \mathsf{C}^{0,k} \triangleq \mathsf{B}^k$$

Proof. The full proof is given in the technical appendix [Raad et al. 2018]. □

Weaker Queue Specification. We develop a weaker queue specification that does not require the total order to. In particular, instead of requiring the existence of to, we require that *ordered* (by lhb) enqueued values not be dequeued out of order: if $(e_1, d_1), (e_2, d_2) \in \mathsf{com}$ and $(e_1, e_2) \in \mathsf{lhb}$, then $(d_2, d_1) \notin \mathsf{lhb}$. That is, $\mathsf{C}^{1,1} = \mathsf{com}^{-1}; \mathsf{lhb}; \mathsf{com}; \mathsf{lhb}$ is irreflexive. As we discuss in §6, this weaker specification allows us to verify the weak implementation of the Herlihy-Wing queue in Fig. 2.

Definition 15 (Queue library)**.** A tuple $(E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb})$ is *queue-consistent* on q if: (1)-(6) as above; and (7) $\mathsf{C}^{1,1}$ ($\mathsf{com}^{-1}; \mathsf{lhb}; \mathsf{com}; \mathsf{lhb}$) is irreflexive. The queue library is $L^{\mathrm{Q}} \triangleq \langle \mathcal{M}^{\mathrm{Q}}, \mathcal{M}_{\mathrm{c}}^{\mathrm{Q}}, \mathtt{loc}^{\mathrm{Q}}, \mathcal{G}_{\mathrm{c}}^{\mathrm{Q}}, \mathcal{G}_{L^{\mathrm{Q}}} \rangle$, where $\mathcal{G}_{\mathrm{c}}^{\mathrm{Q}} \triangleq \left\{ G \in \mathcal{G}_{L^{\mathrm{Q}}} \;\middle|\; \forall q.\ G_q \text{ is queue-consistent on } q \right\}$.

The absence of $\mathsf{C}^{1,1}$ cycles in the weak specification simply states that two values enqueued in (lhb) order, cannot be dequeued in the reverse order, reinforcing a particular case of the FIFO paradigm. For strong queues, the absence of $\mathsf{C}^{n,n}$ cycles (for all n) enforces the full FIFO paradigm and is perhaps less intuitive. However, we propose the $\mathsf{C}^{i,j}$ specification for strong queues not as an intuitive alternative to the linearisability-style specification, but rather as an equivalent formalism better suited to existing techniques such as model checking [Kokologiannakis et al. 2018].

Recall from §2 that the weak implementation in Fig. 2 is *not* a sound implementation of the strong queue library due to the counter example in (W-HWQ). We revisit (W-HWQ), and this time show the absence of a total order to by appealing to Thm. 3. Let us write $(l_1, l_2) \in \mathsf{r}$ to denote that the event of the call labelled l_1 is r-ordered before that of l_2. We thus have $(a, b), (b', c), (c', d'), (d, a') \in \mathsf{po} \subseteq \mathsf{lhb}$; given the values dequeued, we also have $(a, a'), (b, b'), (c, c'), (d, d') \in \mathsf{com}$. We then have

$d' \overset{\mathsf{com}^{-1}}{\to} d \overset{\mathsf{lhb}}{\to} a' \overset{\mathsf{com}^{-1}}{\to} a \overset{\mathsf{lhb}}{\to} b \overset{\mathsf{com}}{\to} b' \overset{\mathsf{lhb}}{\to} c \overset{\mathsf{com}}{\to} c' \overset{\mathsf{lhb}}{\to} d'$. That is, $(d', d') \in \mathrm{C}^{2,2}$. From Thm. 3 we then know it is not possible to construct a to, and thus the annotated behaviour is not allowed by L^{SQ}.

Note that the program in (W-HWQ) constitutes a *minimal* counter example demonstrating the unsoundness of the weak implementation: the weak implementation prohibits $\mathrm{C}^{1,1}$ cycles and $n = 2$ is the smallest n for which the weak implementation admits a $\mathrm{C}^{n,n}$ cycle.

5.4 Stack Library Specification

We consider a stack library with three methods: *new-stack*( ) for constructing a new stack; *push*(s, v) for pushing v on to the stack at s; and *pop*(s) for popping a value from the stack at s.

As we demonstrated with the queue example in §5.3, strong specifications with a total execution order are not always suitable for the WMC setting. As such, we move away from this strong specification style and develop a weaker specification for stacks. Nevertheless, it is straightforward to develop a strong specifications for stacks, analogous to that of strong queues.

We define the *stack interface* as the tuple $\langle \mathcal{M}^{\mathrm{S}}, \mathcal{M}^{\mathrm{S}}_{\mathrm{c}}, \mathtt{loc}^{\mathrm{S}} \rangle$, where $\mathcal{M}^{\mathrm{S}}_{\mathrm{c}} \triangleq \bigcup_{s \in \mathsf{Loc}} \mathcal{M}^{s}_{\mathrm{c}}$ with $\mathcal{M}^{s}_{\mathrm{c}} \triangleq \{\mathtt{new\text{-}stack}(s)\}$; $\mathcal{M}^{\mathrm{S}} \triangleq \bigcup_{s \in \mathsf{Loc}} \mathcal{M}^{s}$ with $\mathcal{M}^{s} \triangleq \mathcal{M}^{s}_{\mathrm{c}} \cup \{\mathtt{push}(s, v), \mathtt{pop}(s, v) \mid v \in \mathsf{Val}\}$; and $\forall l \in \mathcal{M}^{s}.\ \mathtt{loc}^{\mathrm{S}}(l) = \{s\}$. For a stack at location s, we define the following sets of events:

$$\mathcal{C}^{s} \triangleq \{e \mid \mathtt{lab}(e) = \mathtt{new\text{-}stack}(s)\} \qquad \mathcal{A}^{s,v} \triangleq \{e \mid \mathtt{lab}(e) = \mathtt{push}(s, v) \wedge v \neq \bot\} \qquad \mathcal{R}^{s,v} \triangleq \{e \mid \mathtt{lab}(e) = \mathtt{pop}(s, v)\}$$

Let $\mathcal{A}^{s} \triangleq \bigcup_{v \in \mathsf{Val}} \mathcal{A}^{s,v}$ and $\mathcal{R}^{s} \triangleq \bigcup_{v \in \mathsf{Val}} \mathcal{R}^{s,v}$. A tuple $\langle E, \mathsf{po}, \mathsf{com}, \mathsf{so}, \mathsf{lhb} \rangle$ is *stack-consistent on* s if:

(1) there is at most one constructor event: $E^{\mathrm{c}} = \emptyset \vee \exists c \in \mathcal{C}^{s}.\ E^{\mathrm{c}} = \{c\}$;
(2) com relates matching push and pop events: $\mathsf{com} \subseteq \bigcup_{v \in \mathsf{Val}} \mathcal{A}^{s,v} \times \mathcal{R}^{s,v}$;
(3) every push is matched by at most one pop and vice versa: com, com^{-1} are functional;
(4) every unmatched pop returns $\bot$: $E \cap \mathcal{R}^{s} \setminus \mathit{rng}(\mathsf{com}) \subseteq \mathcal{R}^{s,\bot}$
(5) a pop with a previous unmatched push cannot return $\bot$: $[\mathcal{A}^{s} \setminus \mathit{dom}(\mathsf{com})]; \mathsf{lhb}; [\mathcal{R}^{s,\bot}] = \emptyset$.
(6) every matching edge is synchronising: so = com; and
(7) pushed values cannot be popped out of order:
$\forall a_1, a_2, r_1, r_2.\ (a_1, r_1), (a_2, r_2) \in \mathsf{com} \wedge (a_1, a_2), (r_1, r_2) \in \mathsf{lhb} \Rightarrow (a_2, r_1) \notin \mathsf{lhb}$

Intuitively, $(a, r) \in \mathsf{com}$ denotes that r pops a value pushed by a. Note that an event with label $\mathtt{pop}(s, \bot)$ denotes a failed pop (i.e. when the stack is empty). As such, only valid (non-$\bot$) values can be pushed on to the stack. Observe that to ensure the first-in-last-out (LIFO) property, pushed values must be popped in the reverse order, *unless* the first value is popped before the second value is pushed. In other words, if two ordered pushes are popped in the same order (rather than in the LIFO order), then the second push must not happen before the first pop. This constitutes a particular case of the LIFO property and is captured by condition (7) above.

Definition 16 (Stack library)**.** The stack library is $L^{\mathrm{S}} \triangleq \langle \mathcal{M}^{\mathrm{S}}, \mathcal{M}^{\mathrm{S}}_{\mathrm{c}}, \mathtt{loc}^{\mathrm{S}}, \mathcal{G}^{\mathrm{S}}_{\mathrm{c}}, \mathcal{G}_{L^{\mathrm{S}}} \rangle$, where $\mathcal{G}^{\mathrm{S}}_{\mathrm{c}} \triangleq \{G \in \mathcal{G}_{L^{\mathrm{S}}} \mid \forall s.\ G_s \text{ is stack-consistent on } s\}$.

Strong Stack Specification. Note that as with the queue specification, it is straightforward to develop a *strong* stack specification by replacing condition (7) with a strong LIFO condition described via a strict total order to and a lifo predicate, defined analogously to that of strong queues in Def. 14. We next demonstrate that our stack specification in Def. 16 is indeed weaker than this strong specifications described via a total order to. Consider the following program:

$$\begin{array}{l|l} a : \mathit{push}(s, 4); & c : \mathit{push}(s, 2); \\ b : \mathit{push}(s, 1); & d : \mathit{push}(s, 3); \\ c' : \mathit{pop}(s);\ /\!/2 & a' : \mathit{pop}(s);\ /\!/4 \\ b' : \mathit{pop}(s)\ /\!/1 & d' : \mathit{pop}(s)\ /\!/3 \end{array} \qquad \text{(W-stack)}$$

It is straightforward to demonstrate that the annotated outcome is allowed by our specification. However, this outcome is *not* allowed by the strong specification. That is, for all executions of (W-stack), no total order to on the events of G respects the strong LIFO property. This is because the program order on the pop calls ($c' \overset{\text{po}}{\longrightarrow} b'$ and $a' \overset{\text{po}}{\longrightarrow} d'$) requires that their associated push calls be ordered conversely, i.e. $b \overset{\text{to}}{\longrightarrow} c$ and $d \overset{\text{to}}{\longrightarrow} a$. This leads to the cycle $a \overset{\text{po}\subseteq\text{to}}{\longrightarrow} b \overset{\text{to}}{\longrightarrow} c \overset{\text{po}\subseteq\text{to}}{\longrightarrow} d \overset{\text{to}}{\longrightarrow} a$, thus violating the requirement that to is a strict total order.

5.5 Weak Stack Library Specification

We consider a weak stack library with three methods: *new-wstack*( ) for constructing a new weak stack; *try-push*(s, v) for attempting to push v onto the weak stack at s; and *try-pop*(s) for attempting to pop a value from the weak stack at s. The weak stack library is similar to the stack library in §5.4, except that push and pop operations may non-deterministically fail to perform their operations. This is to allow for implementations with better performance. For instance, in an implementation of the weak stack library, the push and pop operations may fail whenever there is contention over the stack top. As before, a pop operation also fails whenever the stack is empty.

The label of an event associated with a *try-push*(s, v) call is of the form $\texttt{try-push}(s, v, o)$, where $o \in \{\top, \bot\}$ denotes the operation outcome, i.e. whether the push was successful. Similarly, the label of an event associated with a *try-pop*(s) call is of the form $\texttt{try-pop}(s, v, o)$, where v denotes the value popped (if any) and $o \in \{\top, \bot\}$ denotes whether the pop was successful. Note that successful *try-pop*(s) calls may only pop valid (non-$\bot$) values; and failed *try-pop*(s) calls may only return $\bot$. That is, for all events e with label $\texttt{try-pop}(s, v, o)$: either $v \neq \bot$ and $o = \top$, or $v = \bot$ and $o = \bot$.

We thus define the *weak stack interface* as $\langle \mathcal{M}^{\text{WS}}, \mathcal{M}_{\text{c}}^{\text{WS}}, \texttt{loc}^{\text{WS}} \rangle$, where $\mathcal{M}_{\text{c}}^{\text{WS}} \triangleq \bigcup_{s \in \text{Loc}} \mathcal{M}_{\text{c}}^{s}$ with $\mathcal{M}_{\text{c}}^{s} \triangleq \{\texttt{new-wstack}(s)\}$; $\forall l \in \mathcal{M}^{s}.\ \texttt{loc}^{\text{WS}}(l) = \{s\}$; and $\mathcal{M}^{\text{WS}} \triangleq \bigcup_{s \in \text{Loc}} \mathcal{M}^{s}$ with $\mathcal{M}^{s} \triangleq \mathcal{M}_{\text{c}}^{s} \cup \left\{ \texttt{try-push}(s, v, o), \texttt{try-pop}(s, v, o) \;\middle|\; v \in \text{Val} \wedge o \in \{\top, \bot\} \right\}$.

For a weak stack at location s, we define the following sets of events:

$$C^{s} \triangleq \left\{ e \;\middle|\; \texttt{lab}(e) = \texttt{new-wstack}(s) \right\}$$

$$\mathcal{A}^{s,v,o} \triangleq \left\{ e \;\middle|\; \texttt{lab}(e) = \texttt{try-push}(s, v, o) \right\} \qquad \mathcal{A}^{s} \triangleq \bigcup_{v \in \text{Val} \setminus \{\bot\}, o \in \{\top, \bot\}} \mathcal{A}^{s,v,o}$$

$$\mathcal{R}^{s,v,o} \triangleq \left\{ e \;\middle|\; \texttt{lab}(e) = \texttt{try-pop}(s, v, o) \right\} \qquad \mathcal{R}^{s} \triangleq \bigcup_{v \in \text{Val} \setminus \{\bot\}} \mathcal{R}^{s,v,\top} \cup \mathcal{R}^{s,\bot,\bot}$$

A tuple $\langle E, \text{po}, \text{com}, \text{so}, \text{lhb} \rangle$ is *weak-stack-consistent* on s if:

(1) there is at most one constructor event: $E^{\text{c}} = \emptyset \vee \exists c \in C^{s}.\ E^{\text{c}} = \{c\}$;
(2) com relates matching push and pop events: $\text{com} \subseteq \bigcup_{v \in \text{Val} \setminus \{\bot\}} \mathcal{A}^{s,v,\top} \times \mathcal{R}^{s,v,\top}$;
(3) every push is matched by at most one pop and vice versa: com, com^{-1} are functional;
(4) every unmatched pop returns $\bot$: $E \cap \mathcal{R}^{s} \setminus \mathit{rng}(\text{com}) \subseteq \mathcal{R}^{s,\bot,\bot}$
(5) every matching edge is synchronising: so = com; and
(6) pushed values cannot be popped out of order:
$\forall a_1, a_2, r_1, r_2.\ (a_1, r_1), (a_2, r_2) \in \text{com}' \wedge (a_1, a_2), (r_1, r_2) \in \text{hb}' \Rightarrow (a_2, r_1) \notin \text{hb}'$.

As before, $(a, r) \in \text{com}$ denotes that r (successfully) pops a value (successfully) pushed by a. Conditions (1)-(6) are analogous to their counterparts of the stack specification in §5.4. Note that the weak stack specification does not include condition (7) of strong stacks. This is to capture the non-deterministic failure of *try-pop*(s) calls, as described above. That is, a *try-pop*(s) call may fail to pop a value *despite* the existence of a value on the stack.

Definition 17 (Weak stack library)**.** The *weak stack library* is $L^{\text{WS}} \triangleq \langle \mathcal{M}^{\text{WS}}, \mathcal{M}_{\text{c}}^{\text{WS}}, \texttt{loc}^{\text{WS}}, \mathcal{G}_{\text{c}}^{\text{WS}}, \mathcal{G}_{L^{\text{WS}}} \rangle$, where $\mathcal{G}_{\text{c}}^{\text{WS}} \triangleq \left\{ G \in \mathcal{G}_{L^{\text{WS}}} \;\middle|\; \forall s.\ G_s \text{ is weak-stack-consistent on } s \right\}$.

```
new-exchanger() ≜ let g = alloc(+∞) in store(g, 1, rlx); g
exchange(g, v) ≜ let i = load(g, rlx) in // next exchange slot
                 if compare-set(g+i, 0, v, rel) then // init
                   sleep(50); let b = compare-set(g+i+1, 0, ⊥, rlx) in // try exchange
                   if b then ⊥ else let v′ = load(g+i+1, acq) in v′
                 else let b = compare-set(g+i+1, 0, v, rel) in // try exchange
                   compare-set(g, i, i+2, rlx); // clean
                   if b then let v′ = load(g+i, acq) in v′ else ⊥
```

Fig. 5. The exchanger implementation

6 VERIFYING CONCURRENT LIBRARY CLIENTS

As discussed in §2, our framework allows for both vertical (using libraries to implement clients that are themselves libraries) and horizontal composition (where clients do not form a library, e.g. (SB-lib)). Here, we explore client verification by focussing on vertical composition as these examples are more challenging. However, the overall approach and proof structure presented applies to examples of both horizontal and vertical composition. For instance, we can easily verify that the annotated behaviour in (SB-lib) is possible, and more generally reason about litmus tests in the WMC literature (e.g. message-passing) and their variants in the library setting.

We verify the correctness of several library implementations. In the technical appendix [Raad et al. 2018], we verify several additional implementations including a mutex library implementation, two reader-writer lock library implementations, and an additional queue library implementation.

6.1 Exchanger Implementation

In Fig. 5 we present a simplified implementation of the exchanger object in `java.util.concurrent`. We represent an exchanger at location g as an infinite zero-initialised array (from g+1 onwards), with two adjacent cells denoting an exchange *slot*. The next free slot is stored at g, initially set to 1. That is, when g stores $i \geq 1$, the next available exchange slot is the adjacent cells at $g{+}i$ and $g{+}i{+}1$.

When the next free slot is at $g{+}i$ (g stores i), a thread calling *exchange*(g, v) may exchange value v in two ways. The first is when the value at $g{+}i$ is zero (no existing offers). The thread then sets $g{+}i$ to its value (line annotated *init*); waits for a partner thread, and upon awakening checks whether it was paired with another thread (at $g{+}i{+}1$) via an atomic *compare-set* (CAS) operation. If the CAS succeeds, (the value at $g{+}i{+}1$ is zero) then no match occurred, and thus $\bot$ is returned; otherwise, the value at $g{+}i{+}1$ is returned. By setting $g{+}i{+}1$ to $\bot$ via a successful CAS, the thread indicates that it is no longer interested in a match and thus future threads should not offer a value at $g{+}i{+}1$. The second way is when the value at $g{+}i$ is non-zero (there is an existing offer). The thread then attempts to match this offer by setting $g{+}i{+}1$ to v (when it is zero) via a CAS. If the CAS succeeds then the match is successful and the value at $g{+}i$ is returned; otherwise, another thread has already matched with the value at $g{+}i$ (by offering a value at $g{+}i{+}1$) and thus $\bot$ is returned. In both cases, the thread advances the next free slot (stored at g) by incrementing it by two (line annotated *clean*).

Soundness of the Exchanger Implementation. Let I_{x} denote the implementation in Fig. 6. As formalised in Thm. 4 below, we show that I_{x} is a sound implementation of the exchanger library L^{X}. That is, for all well-formed programs $\mathtt{e}$, and every consistent execution G_i of $[\![\mathtt{e}]\!]_{L^{\mathsf{X}}:I_{\mathsf{x}}}$, there exists a consistent execution G_s of $\mathtt{e}$ with the same outcome. We refer the reader to [Raad et al. 2018] for the full proof; we proceed with an informal account of how we construct G_s for a given G_i.

Given an arbitrary execution graph G_i of the implementation, note that every successful invocation of *exchange*(g, v) with return value v' comprises (amongst other events) either: (1) an event o with label $\texttt{compare-set}(g{+}i, 0, v, \texttt{rel})$ and an event r with label $\texttt{load}(g{+}i{+}1, v', \texttt{acq})$, for some i; or (2) an event o with label $\texttt{compare-set}(g{+}i{+}1, 0, v, \texttt{rel})$ and an event r with label $\texttt{load}(g{+}i, v', \texttt{acq})$ for some i. To construct the specification graph G_s, we associate each (o, r) pair described above with a single exchanger event e with the appropriate label: $\texttt{lab}(e) = \texttt{exchange}(g, v, v')$, and add e to $G_s.E$. For every failed invocation of *exchange*(g, v) we add an event e with label: $\texttt{lab}(e) = \texttt{exchange}(g, v, \bot)$ to $G_s.E$. All other non-exchanger events (those not corresponding to the events of the I_x implementation), are simply added to the $G_s.E$ unchanged. Constructing the po relation is straightforward; we next describe how we construct com and so. For each pair of successful exchanger events e_1, e_2 of the specification graph, we add $(e_1, e_2), (e_2, e_1)$ to $G_s.\text{com}$ and $G_s.\text{so}$ iff $(o_1, r_2), (o_2, r_1) \in G_i.\text{com}$, where o_1, r_1, o_2 and r_2 denote the corresponding implementation events as described above. For all other non-exchanger events a and b in G_s, (in $G_s.E \cap G_i.E$), we keep their edges unchanged: $(a, b) \in G_s.\text{com} \Leftrightarrow (a, b) \in G_i.\text{com}$ and $(a, b) \in G_s.\text{so} \Leftrightarrow (a, b) \in G_i.\text{so}$. As we show in [Raad et al. 2018], it is then straightforward to show that G_s is exchanger-consistent on g.

Theorem 4. *The exchanger implementation is a sound implementation of the exchanger library* L^{X}.

Proof. The full proof is given in the technical appendix [Raad et al. 2018]. □

6.2 Herlihy-Wing Queue Implementations

Recall our two implementations of the Herlihy-Wing blocking queue in Fig. 2 (§2). As discussed, the underlying memory model of the original implementation [Herlihy and Wing 1990] is SC. Here, we develop our two WMC variants by using the C11 release-acquire (RA) registers. As demonstrated in Example 1, the RA registers can be formalised as a library in our framework.

Soundness of the Strong Implementation. Let I_{sq} denote the strong implementation obtained from Fig. 2 by replacing the highlighted mode with acqrel. As formalised in Thm. 5 below, we show that I_{sq} is sound with respect to the strong (linearisability-style) queue specification. That is, for all well-formed programs e, and for every consistent execution G_i of $[\![\texttt{e}]\!]_{L^{\text{SQ}}:I_{\text{sq}}}$, there exists a consistent execution G_s of e with the same outcome. We refer the reader to [Raad et al. 2018] for the full proof. We proceed with an informal account of how we construct G_s for a given G_i.

Given an arbitrary execution graph G_i of the implementation, note that every invocation of *enq*(q, v) comprises exactly two events: e_1 with label $\texttt{fetch-add}(q, i, i{+}1, \texttt{rel})$ and e_2 with label $\texttt{store}(q{+}i{+}1, v, \texttt{rel})$ with $(e_1, e_2) \in \text{po}$. Similarly, every invocation of *deq*(q) returning v contains (amongst others) two events d_1 and d_2 such that $(d_1, d_2) \in \text{po}$, $\texttt{lab}(d_1) = \texttt{load}(q, \mathit{range}, \texttt{acq})$, and $\texttt{lab}(d_2) = \texttt{atomic-xchg}(q{+}i, v, 0, \texttt{acqrel})$ for some i and *range* with $i < \mathit{range}$. In other words, the d_1 and d_2 are the events of the final for loop iteration.

To construct the specification graph G_s, we associate each (e_1, e_2) pair described above with a single enqueue event e with the appropriate label: $\texttt{lab}(e) = \texttt{enq}(q, v)$, and add e to $G_s.E$. Similarly, we associate each (d_1, d_2) pair described above with a single dequeue event d with $\texttt{lab}(d) = \texttt{deq}(q, v)$, and add d to $G_s.E$. All other non-queue events (those not corresponding to the events of I_{sq}), are simply added to G_sE unchanged. Constructing the po relation is straightforward; we next describe how we construct the com and so relations. For each enqueue event e and dequeue event d of the specification graph, we add (e, d) to $G_s.\text{com}$ and $G_s.\text{so}$ iff $(e_2, d_2) \in G_i.\text{com}$, where e_2 and d_2 denote the corresponding implementation events as described above. For all other non-queue events a and b in G_s, (in $G_s.E \cap G_i.E$), we keep their edges unchanged.

Lastly, we have to demonstrate that G_s is strongly queue-consistent on q by establishing the

```
new-wstack() ≜
  let s = alloc(+∞) in
  store(s, 0, rel); store(s+1, 1, rlx); s

try-push(s, v) ≜
  let lock = compare-set(s, 0, 1, acqrel) in
  if (!lock) then ⊥
  else let top = load(s+1, rlx) in
    store(s+top+1, v, rlx);
    store(s+1, top+1, rlx); store(s, 0, rel); v

try-pop(s) ≜
  let lock = compare-set(s, 0, 1, acqrel) in
  if (!lock) then ⊥
  else let top = load(s+1, rlx) in
    if (top == 1) then // empty stack
      store(s, 0, rel); ⊥ // unlock and return ⊥
    else let v = load(s+top, rlx) in
      store(s+top, 0, rlx); store(s+1, top−1, rlx);
      store(s, 0, rel); v // unlock and return v
```

```
new-stack() ≜
  let s = alloc(2) in
  let ws = new-wstack() in
  let ea = new-elim-array(k) in
  store(s, ws, rlx); store(s+1, ea, rlx); s

new-elim-array(k) ≜
  let a = alloc(k) in
  for i = 0 to k−1 do
    let x = new-exchanger() in
    store(a+i, x, rlx);
  a

elim-exchange(a, v) ≜
  let slot = random(0, k−1) in
  let r = exchange(a[slot], v) in r

push(s, v) ≜
  let ws = load(s, rlx) in
  let ea = load(s+1, rlx) in
  loop
    if (try-push(ws, v)) then break1 ()
    let v′ = elim-exchange(ea, v) in
    if (v′ == POP) then break1 ()

pop(s) ≜
  let ws = load(s, rlx) in
  let ea = load(s+1, rlx) in
  loop
    let v = try-pop(ws) in
    if v then break1 v
    let v = elim-exchange(ea, POP) in
    if (v ≠ ⊥ && v ≠ POP) then break1 v
```

Fig. 6. The elimination stack implementation (below); and its weak stack implementation (above)

(1)-(7) conditions outlined on page 20. Showing conditions (1)-(6) is straightforward; to show (7), we appeal to Thm. 3 and demonstrate the absence of $C^{n,n}$ cycles for all $n \in \mathbb{N}^+$.

Soundness of the Weak Implementation. Let I_{wq} denote the *weak* implementation in Fig. 2. As stated in Thm. 5 below, I_{wq} is sound with respect to the (weak) queue specification. Given a consistent execution G_i of the weak implementation, construction of the corresponding specification execution G_s is analogous to that of the strong implementation outlined above. As such, establishing the (1)-(6) conditions on page 20 is straightforward. To show that G_s is queue-consistent, we additionally show that $C^{1,1}$ is irreflexive. We refer the reader to [Raad et al. 2018] for the full proof.

Theorem 5. *The strong Herlihy-Wing implementation is a sound implementation of the strong queue library L^{SQ}; the weak implementation is a sound implementation of the queue library L^{Q}.*

Proof. The full proof is given in the technical appendix [Raad et al. 2018]. □

6.3 Elimination Stack Implementation

Elimination stack [Hendler et al. 2004] is a scalable concurrent stack implemented using two components: a *weak stack*, *ws*, which implements the internal stack data structure, and an *elimination array*, *ea*, emulating an exchanger, implemented as an array of exchangers to reduce contention.

In Fig. 6 we present the elimination stack (below) and a simplified variant of its internal (weak) stack (above). The weak stack is implemented as an infinite array (from *ws*+2 onwards) and is

protected by a mutex at location *ws*, with the stack top stored at *ws*+1. The weak stack exposes the *try-push* and *try-pop* methods that *attempt* to perform their operations by acquiring the mutex, and fail if the mutex is already taken. The *try-pop* further fails if the stack is empty. The elimination array (of length k) at *ea* exposes *elim-exchange* for exchanging a value. A call to *elim-exchange*(ea, v) randomly selects an array entry within its range and attempts to exchange v.

A call to *push*(s, v) or *pop*(s) first attempts to perform its operation on the (weak) internal stack at *ws*. If this fails, it uses its elimination array at *ea* to directly exchange a value with a concurrently executing thread by calling *elim-exchange*. A pushing thread thus offers the value being pushed (v), whilst a popping thread offers the designated value POP. The pushing thread then checks if the return value matches POP; dually, a popping thread checks if the return value is non-POP and not $\perp$ (failed exchange). Note that the exchange operation may fail either because no exchange took place, or because the exchange was performed between two threads executing the same operation (two pushes or two pops). When this is the case, the operation is simply retried.

Soundness of the Weak Stack Implementation. Let I_{ws} denote the weak stack implementation in Fig. 6. As stated in Thm. 6 below (see [Raad et al. 2018] for the full proof), we show that I_{ws} is a sound implementation of the weak stack library L^{WS}. Lastly, our soundness proof is *compositional* in that it appeals to the specification of the C11 and mutex libraries. That is, we do not consider the implementations of the mutex methods (e.g. *lock*), or the C11 operations (e.g. *load*). Rather, we treat them as abstract library events and use the guarantees offered by their specifications.

Soundness of the Elimination Stack. Let I_{es} denote the elimination stack implementation in Fig. 6. As formalised in Thm. 6 below, we show that I_{es} is a sound implementation of the stack library L^{S}. That is, for all well-formed programs e, and every consistent execution G_i of $[\![\mathsf{e}]\!]_{L^{\mathsf{S}}:I_{\mathsf{es}}}$, there exists a consistent execution G_s of e with the same outcome. We present the full soundness proof of I_{es} in the technical appendix [Raad et al. 2018]; we proceed with a proof sketch here.

Given an arbitrary execution graph G_i of the implementation, note that each *push*(s, v) call produces (amongst others): either an event a_w with label try-push$(ws, v, \top)$ (when v is pushed on the internal stack); or an event a_e with label exchange$(ea{+}i, v, \mathtt{POP})$ for some i (when v is exchanged on the elimination array). Similarly, each *pop*(s) call returning v contains (amongst others) either an event r_w with label try-pop$(ws, v, \top)$; or an event r_e with label exchange$(ea{+}i, \mathtt{POP}, v)$ for some i.

To construct the specification graph G_s, we associate each a_w or a_e event with a single push event a with the appropriate label: $\mathtt{lab}(a) = \mathsf{push}(s, v)$, and add a to $G_s.E$. Similarly, we associate each r_w or r_e event with a single pop event r where $\mathtt{lab}(r) = \mathsf{pop}(s, v)$, and add r to $G_s.E$. Constructing the po relation is straightforward; to construct the com and so relations, for each push event a and pop event r of the specification graph, we add (a, r) to $G_s.\mathsf{com}$ and $G_s.\mathsf{so}$ iff $(a_w, r_w) \in G_i.\mathsf{com}$ *or* $(a_e, r_e) \in G_i.\mathsf{com}$, where a_w, a_e, r_w and r_e denote the corresponding implementation events as described above. As before, all other non-stack events and their edges remain unchanged. As we demonstrate in [Raad et al. 2018], it is then straightforward to show that G_s is stack-consistent on s.

Finally, note that our soundness proof of I_{es} is *compositional* in that it appeals to the specifications of the exchanger, the weak stack and C11 libraries.

Theorem 6. *The weak stack implementation is a sound implementation of the weak stack library L^{WS}; the elimination stack implementation is a sound implementation of the stack library L^{S}.*

Proof. The full proof is given in the technical appendix [Raad et al. 2018]. □

Lastly, we believe the elimination stack implementation in Fig. 6 to be sound against the strong stack specification discussed in §5.4 (page 22). We have eschewed a formal proof in order to dedicate additional space to more challenging examples.

7 RELATED WORK

We review the specification and verification work targeting weak memory models.

Burckhardt et al. [2012] initiated the study of correctness criteria for concurrent objects under WMC. Their work concerned only the x86-TSO model [Owens et al. 2009], following its store buffer operational account. As such, it is based on notions that are internal to this model (such as "flush" of the local store buffer to the main memory). By contrast, we are interested in a more general specification framework for declaratively specified models.

Batty et al. [2013] extend the C11 model with an 'atomic block' construct with higher-level atomicity guarantees, and use this construct for specifying concurrent libraries. This approach is useful for specifying strongly synchronising data structure implementations, such as the Treiber stack, but not for weakly synchronising implementations such as the elimination stack, where the usual atomic specification of a concurrent stack synchronises more than the implementation.

Dongol et al. [2018] develop a linearisability notion for general WMC specified as per the framework of Alglave et al. [2014]. Their object specifications are total orders, describing valid method call sequences, augmented with an so component that, as in our framework, specifies the synchronisation induced by concurrent objects. Nevertheless, as we demonstrated in §2, we consider the use of total orders for specifications under WMC as overly restrictive.

Doherty et al. [2018] present a generalisation of linearisability that can be applied for Lamport's execution structures [Lamport 1986]. They show that, unlike naive applications of linearisability to partial orders, their definition is compositional: Lamport's execution structures can be restricted to each object, and linearisability of all such restrictions implies linearisability of the full structure. Their notion, however, is too strong for our purposes, as it does not hold for various implementations under WMC (the authors identify the non-blocking Treiber stack under RA as such an example).

Burckhardt et al. [2014] present a declarative specification framework based on "abstract executions", using visibility and arbitration relations, which is quite different from the declarative style of existing WMC models using e.g. reads-from and happens-before relations. The authors show that *some* fragments of C11 can be described in their framework. This however requires non-trivial correspondence proofs and still cannot model the full C11, whilst our framework can. In general, existing language- and hardware-level WMC models cannot be directly ported to [Burckhardt et al. 2014] without establishing the correspondence between the two, whilst no such correspondence is required in our framework. Moreover, the work in [Burckhardt et al. 2014] is centred around verifying implementations of replicated objects given as operational message-passing distributed algorithms. By contrast, we use the *same* framework for both specification and implementation verification, and thus, unlike [Burckhardt et al. 2014], our framework allows for compositional verification of "towers of abstraction" built from different libraries.

Perrin et al. [2015] discuss a particular consistency model based on *sequential* object specifications, obtained by strengthening eventual-consistency. The resulting consistency criterion can be expressed in our framework.

Chakraborty et al. [2015] present a formalism specifically tailored for the queue library, and propose the same specification as our weak-queue specification in §5. In the (SC) setting of [Chakraborty et al. 2015], that weak-queue specification is equivalent to its usual ADT-style specification. However, in our more general setting the specifications differ – see (W-HWQ) on page 4.

On the program logic side, there is a number of logics for reasoning about different fragments of the (R)C11 memory model [Doko and Vafeiadis 2016, 2017; Kaiser et al. 2017; Lahav and Vafeiadis 2015; Svendsen et al. 2018; Tassarotti et al. 2015; Turon et al. 2014; Vafeiadis and Narayan 2013], the x86-TSO model [Ridge 2010; Sieczkowski et al. 2015], as well as a logic that is parametrised over the memory model [Alglave and Cousot 2017]. While most of these logics provide facilities

for writing abstract specifications of concurrent libraries, their specification language is not rich enough to express the functional correctness and atomicity specification of a concurrent stack or a queue. As such, the proofs in Turon et al. [2014] provide the *same* (very weak) specification for both stacks and queues, which does not account for ordering constraints between operations concerning different values.

Finally, whilst there is quite some work on the automated verification side, none has yet considered functional correctness of atomic libraries under WMC. On the one hand, there are stateless model checking tools for WMC programs, such as CDSchecker [Norris and Demsky 2016], RCMC [Kokologiannakis et al. 2018], Tracer [Abdulla et al. 2018] and Nidhugg/TSO [Abdulla et al. 2017], that check for memory errors and assertion violations. On the other hand, there are tools for checking *robustness* of a concurrent program [Bouajjani et al. 2013, 2011] (i.e. whether it exhibits non-SC behaviours) and automatically insert fences to enforce robustness. The benefit of robustness is that robust programs can be specified and verified using SC-based techniques, such as linearisability [Herlihy and Wing 1990]. The downside is that enforcing robustness has a significant performance cost; as such, it cannot be used for libraries that intentionally exhibit weak behaviour for better performance.

ACKNOWLEDGMENTS

We thank the POPL 2019 reviewers for their constructive feedback. We thank Derek Dreyer and Michalis Kokologiannakis for their helpful suggestions and feedback. The first and second authors were supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", under the European Union Horizon 2020 Framework Programme (grant agreement number 683289). The fourth author was supported by the Israel Science Foundation (grant number 5166651), and by Len Blavatnik and the Blavatnik Family foundation.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless model checking for TSO and PSO. *Acta Inf.* 54, 8 (2017), 789–818. https://doi.org/10.1007/s00236-016-0275-0

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: An invariance proof method for weak consistency models. In *POPL 2017*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/3009837.3009883

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *ASPLOS 2018*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/3173162.3177156

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *POPL 2013*. ACM, New York, NY, USA, 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, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Hans-J. Boehm and Brian Demsky. 2014. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC 2014*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and enforcing robustness against TSO. In *ESOP (LNCS)*, Vol. 7792. Springer, Heidelberg, Germany, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

Ahmed Bouajjani, Constantin Enea, and Chao Wang. 2017. Checking linearizability of concurrent priority queues. In *CONCUR 2017 (LIPIcs)*, Vol. 85. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 16:1–16:16. https://doi.org/10.4230/LIPIcs.CONCUR.2017.16

Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *ICALP (2) (LNCS)*, Vol. 6756. Springer, Heidelberg, Germany, 428–440. https://doi.org/10.1007/978-3-642-22012-8_34

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

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: Specification, verification, optimality. In *POPL 2014*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2015. Specifying concurrent problems: Beyond linearizability and up to tasks. In *DISC 2015 (LNCS)*, Vol. 9363. Springer, Heidelberg, Germany, 420–435. https://doi.org/10.1007/978-3-662-48653-5_28

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *CONCUR 2015 (LIPIcs)*, Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:20)2015

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP 2010 (LNCS)*, Vol. 6183. Springer, Heidelberg, Germany, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2018. Making linearizability compositional for partially ordered executions. In *IFM 2018 (LNCS)*, Vol. 11023. Springer, Heidelberg, Germany, 110–129. https://doi.org/10.1007/978-3-319-98938-9_7

Marko Doko and Viktor Vafeiadis. 2016. A program logic for C11 memory fences. In *VMCAI 2016 (LNCS)*, Vol. 9583. Springer, Heidelberg, Germany, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *ESOP 2017 (LNCS)*, Vol. 10201. Springer, Heidelberg, Germany, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

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

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular verification of concurrency-aware linearizability. In *DISC 2015 (LNCS)*, Vol. 9363. Springer, Heidelberg, Germany, 371–387. https://doi.org/10.1007/978-3-662-48653-5_25

Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *SPAA 2004*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1007912.1007944

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP 2017 (LIPIcs)*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *ESOP 2017 (LNCS)*, Vol. 10201. Springer, Heidelberg, Germany, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL 2016*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries reasoning for weak memory models. In *ICALP 2015 (LNCS)*, Vol. 9135. Springer, Heidelberg, Germany, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

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

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Leslie Lamport. 1986. On interprocess communication. *Distributed Computing* 1, 2 (01 Jun 1986), 77–85. https://doi.org/10.1007/BF01786227

Xavier Leroy. 2009. A formally verified compiler back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL'05*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *ESOP 2014 (LNCS)*, Vol. 8410. Springer, Heidelberg, Germany, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Gil Neiger. 1994. Set-Linearizability. In *PODC 1994*. ACM, New York, NY, USA, 396. https://doi.org/10.1145/197917.198176

Brian Norris and Brian Demsky. 2016. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 10 (May 2016), 51 pages. https://doi.org/10.1145/2806886

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

Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. 2015. Update consistency for wait-free concurrent objects. In *IPDPS 2015*. IEEE Computer Society, Piscataway, NJ, USA, 219–228. https://doi.org/10.1109/IPDPS.2015.39

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

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2018. Technical appendix. http://plv.mpi-sws.org/yacovet/

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent local subjective logic. In *ESOP 2015 (LNCS)*, Vol. 9032. Springer, Heidelberg, Germany, 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

Tom Ridge. 2010. A rely-guarantee proof system for x86-TSO. In *VSTTE 2010 (LNCS)*, Vol. 6217. Springer, Heidelberg, Germany, 55–70. https://doi.org/10.1007/978-3-642-15057-9_4

Amr Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation* 6, 3 (01 Nov 1993), 289–360. https://doi.org/10.1007/BF01019462

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP 2015 (LNCS)*, Vol. 9032. Springer, Heidelberg, Germany, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Nir Shavit. 2011. Data structures in the multicore age. *Commun. ACM* 54, 3 (March 2011), 76–84. https://doi.org/10.1145/1897852.1897873

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A separation logic for fictional sequential consistency. In *ESOP 2015 (LNCS)*, Vol. 9032. Springer, Heidelberg, Germany, 736–761. https://doi.org/10.1007/978-3-662-46669-8_30

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP 2018 (LNCS)*, Vol. 10801. Springer, Heidelberg, Germany, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *PLDI 2015*. ACM, New York, NY, USA, 110–120. https://doi.org/10.1145/2737924.2737992

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA 2014*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis. 2010. Automatically proving linearizability. In *CAV (LNCS)*, Vol. 6174. Springer, Heidelberg, Germany, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

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

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT aided linearizability proofs. In *CAV 2015 (LNCS)*, Vol. 9207. Springer, Heidelberg, Germany, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1