Reasoning over Relaxed Shared Memory Models: A Tutorial*

Brijesh Dongol

University of Surrey, Guildford, UK

Abstract. The notion of a *relaxed* (aka *weak*) *memory model* is well known, given that such models are implemented by almost all hardware vendors and embedded within the concurrency semantics of many major programming languages. However, given the sheer volume of work on consistency models, formalisations and tools, the area can be both confusing (and intimidating) for newcomers to get into, with even experts missing new developments. In this paper, we coalesce a recent line of work that has focussed on developing *reasoning principles* for relaxed memory into a single reference, extrapolating their key ideas. This line of work aims to reuse (standard) verification techniques for concurrent programs such as Owicki-Gries, rely-guarantee and refinement that were established in the 1970s and 80s, which are well known to most formal methods researchers. We aim to explain these ideas in simple terms, explain some of the main developments and discuss open problems and opportunities for further research. Throughout the paper, we will focus on the RC11 memory model, but discuss how these reasoning principles apply to other models. Instead of focussing on relaxed memory litmus tests (which non-experts may not appreciate), we use a novel proof of a non-trivial buffer developed by Lamport as a running example.

1 Introduction

Hardware vendors and language developers have, for many years, forgone Lamport's principle of *sequential consistency (SC)*, where

> *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* — Lamport [40]

Instead, most architectures implement *relaxed execution models*, which permit greater levels of hardware and compiler optimisation [26]. The (excessive) synchronisation induced by SC architectures was already anticipated by Lamport: "*The requirements needed to guarantee SC rule out some techniques which can be used to speed up individual sequential processors.*", which was later validated by

* The author is supported by VeTSS and EPSRC grants EP/X037142/1, EP/X015149/1 and EP/R025134/2.

empirical studies (e.g., [26]). As such, relaxed memory hardware has been available since the IBM System/370 Model 158 developed in 1972 [8], while high-level programming languages supporting relaxed behaviours (e.g., causal consistency) have been proposed since the early 1990s [4]. (The latter have inspired the design of concurrency models in modern languages such as Java [42] and C++ [9].)

Interestingly, in the same article [40], Lamport also anticipated that the loss of SC *may be necessary for some applications*, but that these would lead to challenges in reasoning. As he states, without SC:

1. "*designing multiprocess algorithms cannot be relied upon to produce correctly executing programs*" and
2. "*verifying their correctness becomes a monumental task*".

Both of these predictions have turned out to be true.

On design, there is still no clear consensus on the semantics of many high-level languages. For example, current formal models for both C11 (the 2011 C/C++ standard) [38,45] and Java [3,50] are known to have unexpected or even unwanted behaviours, despite the fact that the semantics and formal models of relaxed memory hardware are well established [5,6,48,51]. For this tutorial, we focus on a memory model known as repaired C11 (RC11) [38] which has been designed from the ground up to be well-behaved (e.g., does not allow unintuitive behaviours such as thin-air reads), but is still non-trivial to reason about. The model supports four types of ordering guarantees on memory accesses (i.e., reads and writes), but we only consider the two most interesting ones: relaxed and release-acquire, eschewing non-atomic and SC memory accesses.

On verification, early attempts aimed to bypass the problem altogether by relying on data-race freedom (DRF) guarantees [25] — if a program could be shown to be free from data-races, provided that the memory model of the language supported particular desirable properties (e.g., DRF-SC), one could simply reason about the program as if it were executing on an SC memory model. However, many programs are correct in spite of data races and moreover, reasoning about the absence of data races itself is a non-trivial task.

Our point of departure is the desire to support reasoning about a program that may have data races to avoid restricting a programmer to a specific design paradigm. Here, one line of work is on specialised logics for particular weak memory models, e.g., RC11 [32,56] and Rust [21]. While these are impressive efforts that are supported by (automated) verification tools, their reasoning principles enforce use of particular separation logics.

In this tutorial, we focus on an alternative line of work [10,11,18,20,37,49,61], which have aimed to link verification of relaxed memory programs with classical reasoning frameworks such as Owicki-Gries [44] and rely-guarantee [31], both of which are already well-established within the formal methods community. These frameworks stem from the observation that many modern approaches to formalising relaxed memory use complex state models (e.g., graphs of read/write events, or "views") accompanied by *operational semantics* in which the traces of a program are obtained by *interleaving* the program's threads. This therefore provides an opportunity to use classical reasoning techniques as desired.

We note that there are complex models such as the full C11 model, in which compiler optimisations result in intra-thread reorderings [45], requiring consideration of more complex techniques that additionally interleave operations within a thread [60, 61], but we do not consider them in this paper.

Our running example is a buffer developed by Lamport [39], which uses reads and writes on two shared variables to synchronise a producer that executes concurrently with a consumer. Interestingly, the algorithm does not require any additional synchronisation primitives such as locks or read-modify-write instructions (e.g., CAS or FAA). As such, our presentation only comprises reads and writes, further simplifying the memory model, helping keep the paper focussed on reasoning techniques. It is straightforward to extend models with reads and writes to accommodate read-modify-write instructions (e.g., see [17, 19, 20, 37]).

Contributions. This paper aims to be instructive, rather than presenting any significant new results. Nevertheless, we do develop some new material. **(1)** We present a model of RC11 (with relaxed and release-acquire accesses) based on an earlier encoding of the release-acquire model [14]. We consider this to be one of the simplest possible encodings, reducing the encoding of the memory model to its essential components. **(2)** We present a new proof of Lamport's buffer in RC11 and show that the properties defined in the original paper can be guaranteed even when using relaxed accesses, which provide very weak ordering guarantees. **(3)** We extend the model and proof of the buffer to model the data written by the producer and read by the consumer, which we show induces release-acquire synchronisation. **(4)** We develop a generalised set of assertions that allow us to reason about "view-transfer" during release-acquire synchronisation, as required in the proof of the full buffer.

Overview. We present background to this work in §2, where we introduce Lamport's buffer and semi-formally discuss the RC11 memory model and correctness of the buffer under this relaxed memory model. In §3, we present the operational semantics for the relaxed fragment of RC11 reads and writes as well as the assertions introduced in §2. We then present the full buffer in §4, the problems with using relaxed accesses to transfer data between threads and how release-acquire accesses are used to ensure inter-thread memory ordering. Additionally, we introduce the formal model and our new generalised assertions. In §5, we conclude with some discussions on related work and future challenges.

2 Background

In this section, we motivate this paper and present the informal background to this work. In §2.1, we present our running example, which is a buffer developed by Lamport in 1977 designed to support message passing from a producer thread to a concurrently executing consumer thread. Interestingly, synchronisation between these threads is achieved without using expensive primitives such as compare-and-swap or locks. Instead, the algorithm uses two shared variables with only increment operations on each variable (see §2.1 for details). We in-

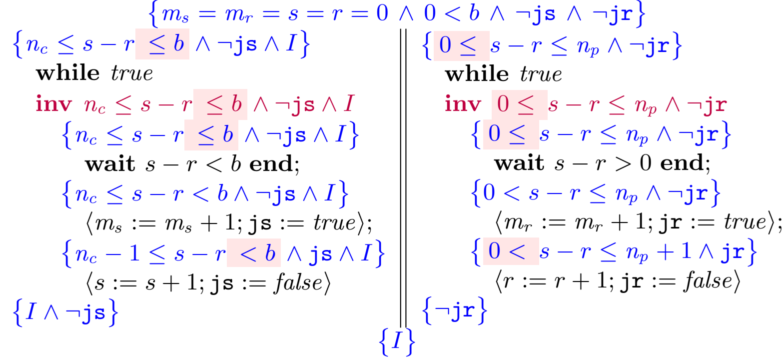

Fig. 1. Correctness proof of Lamport's buffer, where $n \mathrel{\widehat{=}} m_s - m_r$, $I \mathrel{\widehat{=}} 0 \leq n \leq b$, $n_p \mathrel{\widehat{=}} \textbf{if } \texttt{js} \textbf{ then } n-1 \textbf{ else } n$ and $n_c \mathrel{\widehat{=}} \textbf{if } \texttt{jr} \textbf{ then } n+1 \textbf{ else } n$. The purpose of the algorithm is to ensure that I is an invariant of the program. The highlighted conjuncts are present in Lamport's proof, but can be omitted.

formally introduce the RC11 memory model in §2.2, then the behaviour and correctness of the buffer under RC11 in §2.3.

2.1 Lamport's Buffer under SC

Lamport's algorithm [39] (given in Fig. 1), assumes a single producer (left thread) and a single consumer (right thread) that communicate through a shared buffer[1]. As in the original paper [39], we only present the *intent* of the buffer and (for now) do not model the reads and writes on the buffer by the producer and consumer threads. In particular, we use counters m_s and m_r (both initially set to 0) to model the number of messages sent and received, respectively. Later (in §4) we will discuss how the buffer may be implemented and discuss the complications that this induces in the RC11 memory model.

The threads are synchronised using the shared variables s and r, which are modified by the producer and consumer, respectively. In particular, after sending a message (modelled by the increment to m_s), the producer increments s. After receiving a message (modelled by incrementing m_r), the consumer increments r. Note that s (resp. r) is only modified by the producer (resp. consumer). We assume that at most b messages may be in-flight (i.e., sent but not yet received) at any time, thus the producer only attempts to send a message to the buffer if $s - r < b$. Similarly, the consumer only attempts to read from the buffer if an item is available, i.e., $s - r > 0$. Variables $\texttt{js}$ and $\texttt{jr}$ are auxiliary variables [44], which are only used in the proof; we discuss these in more detail below. We use

[1] To simplify the presentation, we elide a modulus parameter, $k > b$. Lamport's algorithm replaces $s-r$ by $(s-r) \mathsf{mod} k$, and the increments to s and r by $(s+1) \mathsf{mod} k$ and $(r+1) \mathsf{mod}\, k$, respectively. But as Lamport discusses, an instance of this parameter is $k = \infty$, which results in the algorithm in Fig. 1.

$\langle a_1; a_2 \rangle$ a statement that executes a_1 followed by a_2 as a single *atomic* step. The statement **wait** *grd* **end** is used to block the respective thread until the guard *grd* holds.

Memory model. Like Lamport [39], the program in Fig. 1 assumes a strong SC memory model [40], i.e., each modification to each shared variable, is visible to all other threads immediately after the modification takes place. This means that we can view the execution of the program in Fig. 1 as an *interleaving* of the two threads. Like Lamport, we assume that each line of the program is atomic since the variables modified by each thread are not modified by the other.

Correctness. Lamport also gives a proof of correctness of his algorithm by modelling each thread as a flowchart with nodes representing (atomic) statements and transitions labelled with assertion that holds after each statement has been executed. This is then accompanied by an informal proof discussing why the main invariant, I, holds. I states that the difference between the number of messages sent and the number of messages received is at least 0, and at most b.

Note that both of the increments in the producer (resp. consumer) are accompanied by an update to `js` (resp. `jr`) to denote that a message has just been sent (resp. received). We convert Lamport's proof to an Owicki-Gries [44] proof outline, which we then encode and verify in Isabelle/HOL's built-in encoding of the Owicki-Gries framework [43]. The Owicki-Gries framework is well known, so we only describe it informally here. A proof outline is *valid* [44] iff each of the following holds.

1. The precondition of the program implies the initial assertion of each thread. We refer to this proof obligation as *initialisation*.
2. The conjunction of the final assertions of all threads implies the postcondition of the program. We refer to this proof obligation as *finalisation*.
3. For each atomic statement a in each thread, the Hoare triple $\{P\}a\{Q\}$ from the proof outline holds. This proof obligation is known as *local correctness*.
4. Each assertion P of each thread is stable against the execution of the other thread, i.e., for each $\{Q\}a$ from the other thread (where Q is the precondition of the atomic statement a), $\{P \wedge Q\}a\{P\}$ holds. This proof obligation is known as *global correctness*.

Lemma 1. *The proof outline in Fig. 1 is valid.*

This lemma has been verified using Isabelle/HOL using the encoding of Owicki-Gries framework that is part of the standard Isabelle/HOL distribution [43]. This encoding is powerful and is supported by tactics that allow the proof outline to be validated fully automatically in under 1 second. This speed has enabled us to experiment with the assertions, and we have discovered redundant conjuncts in Lamport's original proof [39], which are highlighted in Fig. 1 and have been omitted in our proof.

2.2 The RC11 Memory Model

Although Lamport's algorithm and its proof of correctness (as presented in §2.1) are straightforward, this simplicity stems from the strong assumption of an SC memory model, which is difficult to achieve in practice. As already discussed, most modern hardware (and programming languages) implement a relaxed memory model whose effects invalidate the proof outline in Fig. 1.

In this paper, we will consider the RC11 memory model in detail, which is a restricted version of the C/C++ 2011 standard [38]. Given that this is a tutorial on relaxed memory *reasoning*, we only discuss the minimal fragments of the model that are needed to address each example. Moreover, we focus on *principles* for building logics to support formal proofs under relaxed memory, whereby frameworks such as Owicki-Gries can be re-used.

Variables in RC11 may either be *non-atomic* or *atomic*. The former type results in undefined behaviour if there is a data race (a read or write on one thread occurring concurrently with a write on another thread). However, we are interested in the behaviour of racy programs such as Lamport's buffer. For this reason, we will assume that all shared variables are atomic. Under RC11, one can use a range of *memory-ordering*[2] restrictions, describing the observation guarantees that different threads can make. We will make use of three of these:

- *relaxed* reads and writes, which does not impose ordering among concurrent accesses,
- *releasing* writes, which induce a happens-before ordering with acquiring reads (see below), and
- *acquiring* reads, which induce a happens-before ordering whenever the read reads from a releasing write. That is, any happens-before information known by the thread executing the releasing write is transferred to the thread that executes the acquiring read.

We will formalise each of these in the upcoming sections, and discuss how they impact Lamport's buffer.

An important feature of the RC11 memory model is that *program order* [5,38] is maintained, i.e., the order of operations within a thread is unchanged. Thus, as in SC, we can think of the execution of an RC11 program as an *interleaving* of the program's atomic statements. The difference is that, due to relaxed memory effects, the modifications in one thread are not guaranteed to be seen by the other threads, unless dictated by the memory-order restrictions.

2.3 Lamport's Buffer under RC11

We now consider the behaviour of Lamport's buffer under RC11, where (like Lamport [39]) we ignore the reads and writes to the buffer and simply use m_s and m_r to count the number of sends and receives. We will discuss the complications induced by RC11 when modelling the full buffer in the upcoming sections.

[2] See `https://en.cppreference.com/w/c/atomic/memory_order.html`.

$$\left\{ s \overset{p}{=} \mathsf{sv} \wedge r \overset{c}{=} \mathsf{rv} \wedge V_{pc} \wedge b > 0 \wedge \mathsf{sv} - \mathsf{rv} = n \wedge \neg \mathtt{js} \wedge \neg \mathtt{jr} \wedge I \right\}$$

Producer thread:

$P_1 : \left\{ \begin{array}{l} s \overset{p}{=} \mathsf{sv} \wedge V_p \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathsf{sv} - \mathsf{rv}) \end{array} \right\}$

$\mathtt{s}_p :\overset{\mathtt{X}}{=} s;$

$P_2 : \left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge V_p \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathsf{rv}) \end{array} \right\}$

do $\mathtt{r}_p :\overset{\mathtt{X}}{=} r;$

$P_3 : \left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge V_p \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathsf{rv} \wedge \mathtt{r}_p \leq \mathsf{rv}) \end{array} \right\}$

until $\mathtt{s}_p - \mathtt{r}_p < b;$

$P_4 : \left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge (\exists \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv}) \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathsf{rv} < b) \end{array} \right\}$

$\langle m_s := m_s + 1; \mathtt{js} := \mathit{true} \rangle;$

$P_5 : \left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c - 1 \leq \mathsf{sv} - \mathsf{rv}) \end{array} \right\}$

$\langle s :\overset{\mathtt{X}}{=} \mathtt{s}_p + 1, \mathtt{js} := \mathit{false} \rangle$

$P_6 : \left\{ s \overset{p}{=} \mathtt{s}_p + 1 \wedge \neg \mathtt{js} \wedge I \right\}$

Consumer thread:

$\left\{ \begin{array}{l} r \overset{c}{=} \mathsf{rv} \wedge V_c \wedge \neg \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathsf{sv} - \mathsf{rv} \leq n_p) \end{array} \right\}$

$\mathtt{r}_c :\overset{\mathtt{X}}{=} r;$

$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge V_c \wedge \neg \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathsf{sv} - \mathtt{r}_c \leq n_p) \end{array} \right\}$

do $\mathtt{s}_c :\overset{\mathtt{X}}{=} s$

$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge V_c \wedge \neg \mathtt{js} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathtt{s}_c \leq \mathsf{sv} \wedge \mathsf{sv} - \mathtt{r}_c \leq n_p) \end{array} \right\}$

until $\mathtt{s}_c - \mathtt{r}_c > 0;$

$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge \neg \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow 0 < \mathsf{sv} - \mathtt{r}_c \leq n_p) \end{array} \right\}$

$\langle m_r := m_r + 1; \mathtt{jr} := \mathit{true} \rangle;$

$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathsf{sv} - \mathtt{r}_c \leq n_p + 1) \end{array} \right\}$

$\langle r :\overset{\mathtt{X}}{=} \mathtt{r}_c + 1, \mathtt{jr} := \mathit{false} \rangle$

$\left\{ s \overset{p}{=} \mathtt{s}_p + 1 \wedge \neg \mathtt{jr} \right\}$

$$\{I\}$$

where

$$V_p \mathrel{\widehat{=}} (\forall \mathsf{rv}, \mathsf{v}.\ r \overset{c}{=} \mathsf{rv} \wedge \mathsf{v} \in |p\rangle r \Rightarrow \mathsf{v} \leq \mathsf{rv}) \wedge (\exists \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv})$$
$$V_c \mathrel{\widehat{=}} (\forall \mathsf{sv}, \mathsf{v}.\ s \overset{p}{=} \mathsf{sv} \wedge \mathsf{v} \in |c\rangle s \Rightarrow \mathsf{v} \leq \mathsf{sv})$$
$$V_{pc} \mathrel{\widehat{=}} V_p \wedge V_c$$

Fig. 2. Lamport's buffer under RC11, where n_c, n_p, n and I are defined as in Fig. 1. The assertions are named $P_1 \ldots P_6$ to simplify referencing.

Given the range of memory-ordering options, the first question is: which of these is sufficient to correctly implement the buffer from Fig. 1? It turns out that, because each shared variable is only modified by one of the threads, it is sufficient to use *relaxed* accesses everywhere. This results in the program and correctness proof in Fig. 2. We use highlights to indicate the assertions that change or are introduced due to the execution of each atomic command.

RC11-based algorithm. To simplify the discussion, we only consider a single iteration of the loop, but it is straightforward to generalise the program to the looping version since a loop invariant is maintained and the precondition does not over-constrain the initial values of s and r. There are three main differences in the RC11 version of the producer code compared to the SC version (the consumer has similar differences):

1. The read accesses of the shared variables s and r within the guard of the **wait** is replaced by explicit *relaxed reads* ($\mathtt{s}_p :\overset{\mathtt{X}}{=} s$ and $\mathtt{r}_p :\overset{\mathtt{X}}{=} r$) that store the values read into local variables $\mathtt{s}_p$ and $\mathtt{r}_p$, respectively.

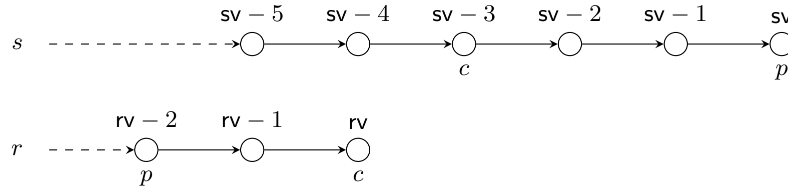

Fig. 3. A possible state of Lamport's buffer under RC11, with each arrow representing the *modification order* [5, 38].

2. The **wait** is replaced by an equivalent **do-until** loop that re-reads r whenever the program is due to continue waiting. Note that we do not re-read s since it is never modified by the consumer thread.
3. The update to s is replaced by a *relaxed write* ($s :\stackrel{\mathtt{X}}{=} \mathsf{s}_p+1$), which increments s. Again, because s is only modified by the producer, we do not need to re-read s, i.e., it is sufficient to use the local copy s_p read in the first line of the producer thread.

Example 1. For the program in Fig. 2, because all accesses are relaxed, a possible snapshot of the program state is depicted in Fig. 3. The top and bottom chains represents the ordering of values of the shared variables s and r, respectively generated by the writes to each variable. Each arrow represents the *modification order* [5, 38], which is an ordering of the values of each variable as the program executes. The modification order is a total order on each variable. The values of s depicted are $\mathsf{sv}-5$, $\mathsf{sv}-4$, ..., sv, whereas the values of r are $\mathsf{rv}-2$, $\mathsf{rv}-1$ and rv. In Fig. 3, we assume that $\mathsf{rv} = \mathsf{sv}-4$.

The value(s) that a thread may read are dictated by the thread's current view. Since s is only modified by the producer thread, p, its view of s is the latest (i.e., most up-to-date) value sv. Thus, any read (including a relaxed read) of s performed by p is guaranteed to return sv. However, the consumer thread c's view of s may be *stale*, as is the case in Fig. 3. Here, since c's view is currently $\mathsf{sv}-3$, a read performed by c may return any of the values in $\{\mathsf{sv}-3, \mathsf{sv}-2, \mathsf{sv}-1, \mathsf{sv}\}$. Similarly, c's view of r is the latest value rv, whereas the producer thread p's view is the stale value $\mathsf{rv}-2$. Thus, any read of r by c may only return rv, whereas a read of p may return values in $\{\mathsf{rv}-2, \mathsf{rv}-1, \mathsf{rv}\}$.

The challenge therefore (which was solved in prior works [17, 20, 37, 61]) is to develop a set of *assertions* that capture the phenomena in Fig. 3 at a high level. We discuss these below.

Correctness under RC11. Validity of the proof outline in Fig. 2 is defined using Owicki-Gries' definition, exactly as in §2.1, i.e., we check the standard proof obligations: initialisation, finalisation, local correctness and global correctness. The only difference is that the assertions used describe the relaxed memory RC11 state. The proof in Fig. 2 only requires two weak memory assertions:

- The *definite value* assertion, $x \stackrel{\tau}{=} \mathsf{xv}$, which indicates that thread τ's view of x is of the last value, xv.
- The *possible values* function, $\langle \tau \rangle x$, which returns the set of values that τ may read for x.

Example 2. For the state in Fig. 3 (described in Example 1), we have $s \stackrel{p}{=} \mathsf{sv}$ and $r \stackrel{c}{=} \mathsf{rv}$. Moreover, $\langle p \rangle r = \{\mathsf{rv} - 2, \mathsf{rv} - 1, \mathsf{rv}\}$ and $\langle c \rangle s = \{\mathsf{sv} - 3, \mathsf{sv} - 2, \mathsf{sv} - 1, \mathsf{sv}\}$.

In general, if $x \stackrel{\tau}{=} \mathsf{xv}$, then $\mathsf{xv} \in \langle \tau \rangle x$ and, in fact, $\langle \tau \rangle x = \{\mathsf{xv}\}$. Note however that $\langle \tau \rangle x = \{\mathsf{xv}\}$ does *not* imply $x \stackrel{\tau}{=} \mathsf{xv}$ since the definite-view assertion also requires that τ's view of x is the last-written value.

With this understanding, it is possible to prove the following lemma. We have, in fact, encoded and verified validity in Isabelle/HOL using the encoding by Dalvandi et al [20].

Lemma 2. *The proof outline in Fig. 1 is valid.*

The main components of the proof outline in Fig. 2 are derived from the SC proof in Fig. 1. The main difference is in the supporting assertions that are used to establish the invariant, I.

Example 3. Consider the assertion P_1. The conjunct $\forall \mathsf{rv}.\ r \stackrel{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathsf{sv} - \mathsf{rv}$ is used directly to establish I. Note the similarity of this conjunct with the corresponding assertion $n_c \leq s - r$ in Fig. 1. The difference in RC11 is that we cannot directly use the values s and r since these values may be different for different threads.

The value sv is the definite value that p observes of s, which is obtained from the conjunct $s \stackrel{p}{=} \mathsf{sv}$ in P_1 itself. For rv, we cannot fix the value of r using a corresponding conjunct of the form $r \stackrel{c}{=} \mathsf{rv}$ since such an assertion would *not* be stable under the update to r in thread c, i.e., the global correctness proof would fail. Instead, we obtain the value rv using universal quantification, which states that for *any* definite value rv of r as seen by thread c, we have $n_c \leq \mathsf{sv} - \mathsf{rv}$.

As in Fig. 1, the conjunct $\neg \mathtt{js}$ supports the global correctness proof of thread c (establishing a value for n_p. Finally, V_p is used within thread p to establish the conjunct $\mathtt{r}_p \leq \mathsf{rv}$ in P_3 and the conjunct $\exists \mathsf{rv}.\ r \stackrel{c}{=} \mathsf{rv}$ in P_4. The latter is straightforward since it follows directly from V_p, but the former is interesting.

Essentially, we require a condition that allows us to strengthen the for-all condition in P_3, which describes the value read by p in relation to the last value written by c. As depicted in Fig. 3, this value is at most rv (i.e., the value that c can see). To support establishing this conjunct (i.e., $\mathtt{r}_p \leq \mathsf{rv}$) in P_3, we use the first conjunct of V_p, which states that, if the definite value of r is rv (according to c) and p can read a value v, then v must be at most rv. Under this assumption, the read of r by thread p within the **do** loop is guaranteed to establish $\mathtt{r}_p \leq \mathsf{rv}$.

3 Relaxed Reads/Writes

We now discuss the formal background to support the claims in §2. We start by introducing the fragment of RC11 comprising only *relaxed* read and write

accesses to keep the discourse as simple as possible. We introduce the operational semantics in §3.1, then in §3.2 formalise the "value" assertions from Fig. 2. Later, in §4, we consider the more complex release-acquire fragment and the assertions that can be used to reason about message passing under such semantics.

3.1 Operational Semantics

The semantics of the RC11 was originally defined in a graph-based declarative (aka axiomatic [5, 38]) style, which provide descriptions of the allowable behaviours of a program in a given memory model. These semantics consider *complete* executions of a program to determine whether or not each execution is allowed by (i.e., possible in) the memory model using axioms over the resulting execution graph.

However, such semantics are not amenable to stepwise verification, as required by frameworks such as Owicki-Gries and rely/guarantee. However, for many axiomatic models (including RC11), it is possible to develop equivalent operational semantics [23, 32, 33, 46, 48]. These models can either build directly on the axiomatic graphs [23, 34, 41] or use an equivalent encoding based on "views" [11, 15, 32, 46, 61]. The assertions needed for relaxed memory verification can easily be defined in both operational encodings (e.g., see [60]). In this paper, we opt to present the view-based semantics, which are closer to the assertions that we use, and hence easier to understand.

Language. We assume the extended while language below. We assume Var denotes the set of all variables, which are partitioned into local variables, L, and global variables, G. We further partition the local variables into registers, R_L, and auxiliary variables, A_L. Registers appear in programs, whereas auxiliary variables are only used to support proofs. We also assume a set of values, Val, and a set of thread identifiers, Tid.

Suppose $\mathsf{v} \in Val$, $\mathtt{r} \in R_L$, $\mathtt{a} \in A_L$, $e, e_1, e_2 \in Exp$, $f, f_1, f_2 \in Aux$, $x \in G$, $a \in Asgn$, $c, c_1, c_2 \in Com$ and $b \in Exp$ is boolean valued. Let $\mathtt{M} \in \{\mathtt{X}, \mathtt{R}, \mathtt{A}\}$ define the memory access type. Then the syntax of programs is given by:

$$Exp ::= \mathsf{v} \mid \mathtt{r} \mid \ominus e \mid e_1 \oplus e_2 \qquad\qquad Aux ::= \mathsf{v} \mid \mathtt{r} \mid \mathtt{a} \mid \ominus f \mid f_1 \oplus f_2$$

$$Asgn ::= \mathtt{r} := e \mid \mathtt{r} \overset{\mathtt{M}}{:=} x \mid x \overset{\mathtt{M}}{:=} e$$

$$Com ::= a \mid \langle a; \overline{\mathtt{a}} := \overline{f} \rangle \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{do } c \textbf{ until } b$$

$$Prog ::= \lambda \tau \in Tid.\ c$$

Note that all expressions are over local registers only. We have three types of assignments: local assignments ($\mathtt{r} := e$), reads ($\mathtt{r} \overset{\mathtt{M}}{:=} x$) and writes ($x \overset{\mathtt{M}}{:=} e$). The remaining commands are standard.

For simplicity, we assume top-level parallelism though it is possible to support dynamic thread creation [37]. A concurrent program is a mapping from thread identifiers to commands. We assume that the set of registers are further partitioned into disjoint sets of local registers for each thread.

Modularising the Memory Semantics. The state is split into two components: a local component (which has type $\Sigma \mathrel{\widehat{=}} L \to Val$ as in standard SC semantics [43, 44]) and a relaxed-memory state (whose type Γ can be instantiated in different ways). We discuss the view-based instantiation of Γ below.

Let $\sigma \in \Sigma$, $\gamma \in \Gamma$ and $\Pi \in Prog$. At the top level, program execution is defined by the following rule:

$$\textsc{Prog}\ \frac{\langle \sigma, \gamma, \Pi(\tau) \rangle \to_\tau \langle \sigma', \gamma', c' \rangle}{\langle \sigma, \gamma, \Pi \rangle \to \langle \sigma', \gamma', \Pi[\tau \mapsto c'] \rangle}$$

Here the program takes a step if some thread τ takes a step. The execution of non-atomic commands is standard. As an example, we present the rule for atomics with auxiliary variables, where $[\![e]\!]_\sigma$ denotes the value of e in the state σ.

$$\textsc{Aux}\ \frac{\langle \sigma, \gamma, a \rangle \to_\tau \langle \sigma', \gamma', \bot \rangle \qquad \bar{\mathtt{a}} = \mathtt{a}_1, \ldots, \mathtt{a}_n \qquad \overline{f} = f_1, \ldots, f_n}{\langle \sigma, \gamma, \langle a, \bar{\mathtt{a}} := \overline{f} \rangle \rangle \to_\tau \langle \sigma'[\mathtt{a}_1 \mapsto [\![f_1]\!]_\sigma, \ldots, \mathtt{a}_n \mapsto [\![f_n]\!]_\sigma], \gamma', \bot \rangle}$$

Note that because the auxiliary update must take place in parallel with the execution of a, we evaluate e in the initial local state σ. We use $\bot$ to determine that the command under consideration has terminated.

Finally, we have rules for the three assignment statements. The rule for local assignment $\mathtt{r} := e$ is as standard and updates the local state σ. For the reads and writes, we have the following rules, which are parameterised by the memory ordering restriction $\mathtt{M}$. The premises of both rules for different instances of $\mathtt{M}$ are defined in Figs. 4 and 7 and will be discussed below.

$$\textsc{Rd}\ \frac{\gamma \xrightarrow{\langle x, \mathsf{read}(v), \mathtt{M} \rangle}_\tau \gamma'}{\langle \sigma, \gamma, \mathtt{r} \overset{\mathtt{M}}{:=} x \rangle \to_\tau \langle \sigma[\mathtt{r} \mapsto v], \gamma', \bot \rangle} \qquad \textsc{Wr}\ \frac{\gamma \xrightarrow{\langle x, \mathsf{write}([\![e]\!]_\sigma), \mathtt{M} \rangle}_\tau \gamma'}{\langle \sigma, \gamma, x \overset{\mathtt{M}}{:=} e \rangle \to_\tau \langle \sigma, \gamma', \bot \rangle}$$

The Rd read rule presumes that it is possible to transition from γ to γ' using the rule $\gamma \xrightarrow{\langle x, \mathsf{read}(v), \mathtt{M} \rangle}_\tau \gamma'$, where the read returns the value v. Note that the reads may have a side effect of updating the view and hence γ' may not be the same as γ. Similarly, the write rule presumes $\gamma \xrightarrow{\langle x, \mathsf{write}([\![e]\!]_\sigma), \mathtt{M} \rangle}_\tau \gamma'$ and introduces a new write with value $[\![e]\!]_\sigma$ to the weak memory state.

View-Based Relaxed Read/Write Semantics. We provide an instantiation of Γ based on the formalisation in [14]. Each $\gamma \in \Gamma$ is a pair of the form $\langle m, \mathcal{V} \rangle$.

- m is a set of *messages* comprising *all* previously executed writes. Each message is a tuple of the form $\mu = \langle x, \mathsf{v}, t, V, \mathtt{M} \rangle$ where: $x \in Var$ is the variable of the message, $\mathsf{v} \in Val$ is the written value, $t \in \mathbb{Q}$ is the *timestamp*, $V \in \mathsf{View} \mathrel{\widehat{=}} Var \to \mathbb{Q}$ is the message's *view*, and $\mathtt{M}$ is the memory ordering. Timestamps are used to order messages of the same variable, as depicted in Fig. 3. A message's view is used in the release-acquire fragment, and can be ignored for now since it has no use for relaxed reads/writes.

$$\frac{\begin{array}{c}\text{READ-RLX}\\ e = \langle x, \mathsf{read}(v), \mathtt{X}\rangle \quad \langle x, v, t, _, _\rangle \in m \\ \mathcal{V}(\tau)(x) \leq t \quad V' = \mathcal{V}(\tau)[x \mapsto t]\end{array}}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}
\qquad
\frac{\begin{array}{c}\text{WRITE-RLX-REL}\\ e = \langle x, \mathsf{write}(v), \mathtt{M}\rangle \quad V' = \mathcal{V}(\tau)[x \mapsto t] \\ \mathcal{V}(\tau)(x) < t \quad t \notin \mathsf{ts}(m_{|x}) \\ \mu = \langle x, v, t, V', \mathtt{M}\rangle\end{array}}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m \cup \{\mu\}, \mathcal{V}[\tau \mapsto V']\rangle}$$

Fig. 4. Relaxed reads and relaxed/releasing writes in RC11

- $\mathcal{V} : Tid \to \mathsf{View}$ is a mapping from each thread to its view.

The operational semantics of relaxed reads/writes on the weak memory state are given by the transitions in Fig. 4. We use the functions $\mathsf{var}(\mu)$, $\mathsf{val}(\mu)$, $\mathsf{ts}(\mu)$, $\mathsf{view}(\mu)$ and $\mathsf{ord}(\mu)$ to retrieve the variable (x), value (v), timestamp (t), view (V) and memory order ($\mathtt{M}$) of a message μ. We lift these projection functions to sets of messages pointwise (e.g., $\mathsf{ts}(S) \mathrel{\widehat{=}} \{\mathsf{ts}(\mu) \mid \mu \in S\}$), and write $m_{|x}$ for the set of all messages $\mu \in m$ with $\mathsf{var}(\mu) = x$.

A relaxed read by thread τ on shared variable x may read from any message $\langle x, v, t, _, _\rangle$ whose timestamp t is at least as large as τ's view of x. The read ignores the view and memory-ordering constraints of the message, but retrieves the value v. As a side effect of this read, the view of τ for x is updated to t.

A relaxed write by thread τ writes to a variable x, adds a corresponding message μ to m with some *fresh* timestamp t that is larger than $\mathcal{V}(\tau)(x)$, the timestamp of the latest message to x that was observed by τ. Then, the thread τ updates its view of x to t. Note that the write rule in Fig. 4 also encodes releasing writes, which we discuss in §4. Such writes make use of the component V' in each message, which is ignored by relaxed reads/writes.

3.2 Value Assertions and Owicki-Gries

With an operational semantics in place, we now formalise the value assertions used in Fig. 2. The view function and definite-view assertion are now straightforward to define:

$$\begin{aligned}
\mathsf{obs}(\tau) &\mathrel{\widehat{=}} \lambda\alpha.\ \textbf{let}\ \alpha = \langle _, \langle m, \mathcal{V}\rangle\rangle\ \textbf{in}\ \{\mu \in m \mid \mathsf{ts}(\mu) \geq \mathcal{V}(\tau)(\mathsf{var}(\mu))\} \\
[\tau\rangle x &\mathrel{\widehat{=}} \lambda\alpha.\ \mathsf{val}(\mathsf{obs}(\tau)(\alpha)_{|x}) \\
x\langle\!\langle\tau &\mathrel{\widehat{=}} \lambda\alpha.\ \textbf{let}\ \alpha = \langle _, \langle m, \mathcal{V}\rangle\rangle\ \textbf{in}\ \mathcal{V}(\tau)(x) = \max(\mathsf{ts}(m_{|x})) \\
x \stackrel{\tau}{=} e &\mathrel{\widehat{=}} \lambda\alpha.\ \textbf{let}\ \alpha = \langle \sigma, _\rangle\ \textbf{in}\ (x\langle\!\langle\tau)(\alpha) \wedge ([\tau\rangle x)(\alpha) = \{[\![e]\!]_\sigma\}
\end{aligned}$$

Thus $\mathsf{obs}(\tau)$ determines the set of messages that τ can observe in the given state. These are the messages whose timestamp is at least the timestamp of τ's view for $\mathsf{var}(\mu)$. $[\tau\rangle x$ is a function on the state that returns the set of values of x that τ may read. $x\langle\!\langle\tau$ is a predicate that holds iff τ's view of x is the message over x with the highest timestamp. $x \stackrel{\tau}{=} \mathsf{xv}$ holds iff τ is viewing the highest-timestamp message on x and this message has the value xv.

The operational semantics also leads to a natural definition of a Hoare-triple, where $\tau \rhd c$ denotes the command c executed by thread τ.

Definition 1. *Given assertions $P, Q : \Sigma \times \Gamma \to \mathbb{B}$, thread $\tau \in Tid$, command $c \in Com$ and program $\Pi \in Prog$:*

$$\{P\}\tau \rhd c\{Q\} \mathrel{\widehat{=}} \forall \sigma, \gamma.\ P(\langle \sigma, \gamma \rangle) \wedge \langle \sigma, \gamma, c\rangle \to_\tau^* \langle \sigma', \gamma', \bot \rangle \Rightarrow Q(\langle \sigma', \gamma' \rangle)$$
$$\{P\}\Pi\{Q\} \mathrel{\widehat{=}} \forall \sigma, \gamma.\ P(\langle \sigma, \gamma \rangle) \wedge \langle \sigma, \gamma, \Pi\rangle \to^* \langle \sigma', \gamma', \lambda\tau.\ \bot \rangle \Rightarrow Q(\langle \sigma', \gamma' \rangle)$$

This definition naturally supports decomposition rules from Hoare logic for compound statements, e.g., sequential composition is covered by the following rule.

$$\textsc{seq}\ \frac{\{P\}\tau \rhd c_1\{Q\} \qquad \{Q\}\tau \rhd c_2\{R\}}{\{P\}\ \tau \rhd c_1; c_2\ \{R\}}$$

The decomposition of proof outlines of concurrent programs is also the same as in Owicki-Gries' framework (see §2.1), except that each triple must include information about the executing thread since we use thread-specific assertions.

- Local correctness holds if, for each atomic statement a in each thread τ, the Hoare triple $\{P\}\tau \rhd a\{Q\}$ from the proof outline holds.
- Global correctness holds if, for each assertion P of each thread τ, for each $\{Q\}\tau' \rhd a$ in another thread τ', we have $\{P \wedge Q\}\tau' \rhd a\{P\}$.

The main difficulties stem from the proof rules for atomic commands (e.g., relaxed reads and writes). Here, it is necessary to describe how these statements interact with the view-based assertions above [20,61]. We give some examples of such rules below.

$$\textsc{rd-poss}\ \frac{}{\{[\tau\rangle x = S\}\tau \rhd \mathtt{r} :\overset{\mathtt{M}}{=} x\{\mathtt{r} \in S\}} \qquad \textsc{rd-def}\ \frac{}{\{x \overset{\tau}{=} \mathsf{xv}\}\tau \rhd \mathtt{r} :\overset{\mathtt{M}}{=} x\{\mathtt{r} = \mathsf{xv}\}}$$

$$\textsc{rd-stable}\ \frac{}{\{x \overset{\tau}{=} \mathsf{xv}\}\tau' \rhd \mathtt{r} :\overset{\mathtt{M}}{=} y\{x \overset{\tau}{=} \mathsf{xv}\}} \qquad \textsc{wr-latest}\ \frac{}{\{x\langle\!\langle\tau\}\tau \rhd x :\overset{\mathtt{M}}{=} e\{x \overset{\tau}{=} e\}}$$

Rule RD-POSS states that, if the view of τ on variable x is such that it sees the set of values S, then loading the value of x into the local register $\mathtt{r}$ results in a post state where $\mathtt{r} \in S$. By RD-DEF, if the view of τ on x is the last value xv, then the effect of loading x into $\mathtt{r}$ results in a post state where $\mathtt{r}$ has the value xv. Rule RD-STABLE states that a definite-view assertion is stable under the read (by any thread). Note that x and y as well as τ and τ' are not necessarily different. Finally, WR-LATEST states that if τ sees the last value on x, then storing e in x results in a state where, from the perspective of τ, the value of x is definitely e.

These can be extended to cover stability of assertions, e.g., due to assignments on other variables and statements executed by other threads. Soundness of each of these rules is proved by unfolding Definition 1. Such proofs are generally straightforward to encode and prove in theorem provers such as Isabelle/HOL [11,20,61].

4 Modelling and Verifying the Full Buffer

Although Lemma 2 establishes correctness of the main invariant I, there are additional subtleties if the correctness proof includes details about the buffer itself. As Lamport discusses (though does not prove), under SC, the buffer can be modelled and verified if the algorithm includes an array $B[0..b-1]$, where the producer puts each new message into $B[s \text{ mod } b]$, and the consumer takes each message from $B[r \text{ mod } b]$. To simplify the assertions and avoid some tedious modulo calculations, we assume that B is an infinite-sized array. With some additional work, the program and its proof outline can be extended to accommodate a wraparound buffer.

4.1 Synchronising the Buffer in RC11

Under RC11, each $B[i]$ is a shared location, and thus is affected by the relaxed memory semantics. In particular, it may be possible for the consumer to read a stale value of $B[i]$, which is undesirable. To prevent this, we must strengthen the memory-ordering guarantees.

In an implementation, updates to B may involve a large payload, so it is undesirable to introduce extra memory orderings on B directly. Instead, we use the index updates on s to induce *release-acquire* synchronisation, which enables transfer of happens-before information from the producer to the consumer. We require that, upon reading a value k of s, that allows the consumer to read from B (i.e., exit its **do-until** loop), the consumer thread c must see the latest value of $B[k]$. However, the semantics of reads (cf Fig. 4) is that the reader may read from *any* later write. Thus, it may be the case that c jumps ahead by several steps when reading s, i.e., updates its value from k_0 to k where $k - k_0 > 1$. Since it will read from the buffer in order, i.e., $B[k_0]$, then $B[k_0 + 1]$, up to $B[k-1]$, we require that upon reading k for s, the consumer thread sees the latest values of each of the entries $B[k_0], B[k_0 + 1], \ldots, B[k-1]$.

Release-acquire synchronisation provides precisely this guarantee, which we describe informally using our view model using the example state in Fig. 5.

Example 4. Fig. 5 aims to capture the effect of an acquiring read of s (returning the value $\mathsf{sv} - 1$) by thread c. Here, we not only consider the writes on s and r, but also on the buffer indices. We assume that all writes are relaxed (`X`) except those on s, which are always releasing (`R`).

The view of c before and after this read are highlighted in red and green, respectively. Notice how the effect of the read is to not only update c's view on s but also its views on the important buffer indices $B[\mathsf{sv}-2]$ and $B[\mathsf{sv}-1]$. Thread c's view of $B[\mathsf{sv}]$ is unchanged. To achieve this in the view model, the key idea is to *store the writing thread's view within each releasing write at the time of writing*. In Fig. 5, when the write of value sv occurs, the view of this write on B is as indicated by the blue dashed arrows. Now, if c reads sv for s using an acquiring read, c not only updates its view of s (cf Fig. 4), it also *inherits* the views of this write.

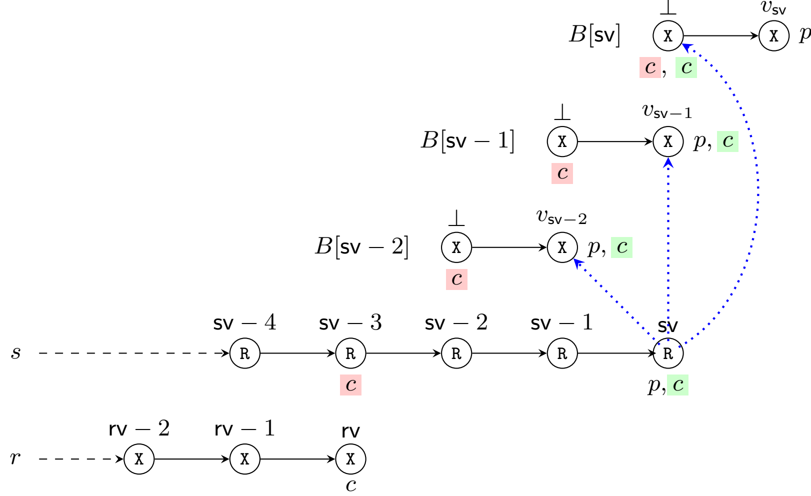

Fig. 5. A possible state of Lamport's buffer under RC11, with the solid arrows representing the *modification order* [5,38] and the dashed arrows representing the message view. In this state p has written the value v_{sv} in $B[\mathsf{sv}]$, but it has not yet updated s.

These observations lead to the program and proof in Fig. 6. To model the send-receive behaviour, we assume an (infinite) array of values, v, that p wishes to send (via B) and that these are read into an array of values w by c. Thus main invariant is thus I_{pc} (see Fig. 6). Given that rv is the value of r definitely seen by c, for all all indices i less than rv, each value received w_i must equal the value sent (v_i).

The proof I_{pc} is supported by new assertions W_p and W_c. Here, W_c describes *view-transfer* during release-acquire synchronisation via its consequent, which is a predicate of the form $\pi \overset{\tau}{\rightsquigarrow} \varphi$, stating that, if thread τ performs an *acquiring read* such that π holds, then the state *after* the read is guaranteed to satisfy φ.

Example 5. Consider the state in Fig. 5 before c performs the read of s, i.e., c's view of B is as indicated in red. Let $(x = \mathsf{xv}) \overset{\tau}{\rightsquigarrow} \varphi$ be a special case of $\pi \overset{\tau}{\rightsquigarrow} \varphi$, where the transition to φ is only guaranteed if τ reads the value xv for x (see (1) for the formal definition). Then in the state before c performs its read, we have:

$$(s = \mathsf{sv}) \overset{c}{\rightsquigarrow} (\forall k < \mathsf{sv}.\ B[k] \overset{c}{=} v_k)$$

i.e., if c were to read the value sv for s using an acquiring read, then after performing this read, its view would be updated so that $(\forall k < \mathsf{sv}.\ B[k] \overset{c}{=} v_k)$ holds, as indicated by the green views.

4.2 Release-Acquire Semantics Proof

We now present the formal definitions that establish the intuitions over view transfer from §4.1.

$$\{Init \wedge I_{pc} \wedge W_c\}$$

$P'_1 : \left\{\begin{array}{l} P_1 \wedge (\forall i.\ B[i]\langle\langle p\rangle \wedge \\ (\forall i < \mathsf{sv}.\ B[i] \overset{p}{=} v_i) \end{array}\right\}$
$\mathtt{s}_p :\overset{\mathtt{X}}{=} s;$
$P'_2 : \{P_2 \wedge W_p\}$
do $\mathtt{r}_p :\overset{\mathtt{X}}{=} r;$
$P'_3 : \{P_3 \wedge W_p\}$
until $\mathtt{s}_p - \mathtt{r}_p < b;$
$P'_4 : \{P_4 \wedge W_p\}$
$\langle B[\mathtt{s}_p] :\overset{\mathtt{X}}{=} v_{\mathtt{s}_p};$
$\mathtt{ms}, \mathtt{js} := \mathtt{ms} + 1, true\rangle;$
$P'_5 : \left\{P_5 \wedge W_p \wedge B[\mathtt{s}_p] \overset{c}{=} v_{\mathtt{s}_p}\right\}$
$\langle s :\overset{\mathtt{R}}{=} \mathtt{s}_p + 1; \mathtt{js} := false\rangle$
$P'_6 : \left\{\begin{array}{l} P_6 \wedge (\forall i.\ B[i]\langle\langle p\rangle \wedge \\ (\forall i < \mathsf{sv}+1.\ B[i] \overset{p}{=} v_i) \end{array}\right\}$

$Q'_1 : \{Q_1 \wedge I_{pc} \wedge W_c\}$
$\mathtt{r}_c :\overset{\mathtt{X}}{=} r;$
$Q'_2 : \{Q_2 \wedge I_{pc} \wedge W_c\}$
do $\mathtt{s}_c :\overset{\mathtt{A}}{=} s$
$Q'_3 : \left\{Q_3 \wedge I_{pc} \wedge W_c \wedge (\forall k < \mathtt{s}_c.\ B[k] \overset{c}{=} v_k)\right\}$
until $\mathtt{s}_c - \mathtt{r}_c > 0;$
$Q'_4 : \left\{Q_4 \wedge I_{pc} \wedge W_c \wedge (\forall k < \mathtt{s}_c.\ B[k] \overset{c}{=} v_k)\right\}$
$\langle w_{\mathtt{r}_c} :\overset{\mathtt{X}}{=} B[\mathtt{r}_c];$
$\mathtt{mr}, \mathtt{jr} := \mathtt{mr} + 1, true\rangle;$
$Q'_5 : \{Q_5 \wedge I_{pc} \wedge W_c \wedge w_{\mathtt{r}_c} = v_{\mathtt{r}_c}\}$
$\langle r :\overset{\mathtt{X}}{=} \mathtt{r}_c + 1; \mathtt{jr} := false\rangle$
$Q'_6 : \{Q_6 \wedge I_{pc} \wedge W_c\}$

$$\{I \wedge I_{pc}\}$$

where
$$W_p \mathrel{\widehat{=}} (\forall i.\ B[i]\langle\langle p\rangle \wedge (\forall i < \mathtt{s}_p.\ B[i] \overset{p}{=} v_i)$$
$$W_c \mathrel{\widehat{=}} (\forall \mathsf{sv}.\ s \overset{p}{>} \mathsf{sv} \Rightarrow s = \mathsf{sv} \overset{c}{\rightsquigarrow} (\forall k < \mathsf{sv}.\ B[k] \overset{c}{=} v_k))$$
$$I_{pc} \mathrel{\widehat{=}} (\forall \mathsf{rv}.\ s \overset{c}{=} \mathsf{rv} \Rightarrow w_0 w_1 \ldots w_{\mathsf{rv}-1} = v_0 v_1 \ldots v_{\mathsf{rv}-1})$$

Fig. 6. Full model of Lamport's buffer under RC11, where *Init* is the initial precondition in Fig. 2, and $Q_1 \ldots Q_6$ are the counterparts to $P_1 \ldots P_6$ in the consumer thread.

ACQ-RF-RLX
$$\frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \quad \langle x, v, t, _, \mathtt{X}\rangle \in m \quad \mathcal{V}(\tau)(x) \le t \quad V' = \mathcal{V}(\tau)[x \mapsto t]}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_\tau \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

ACQ-RF-REL
$$\frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \quad \langle x, v, t, V, \mathtt{R}\rangle \in m \quad \mathcal{V}(\tau)(x) \le t \quad V' = \mathcal{V}(\tau) \sqcup V}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_\tau \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

Fig. 7. Transitions of acquiring reads

Release-Acquire Semantics. Recall that the semantics of releasing writes is already covered by the WRITE-RLX-REL rule in Fig. 4. This time we take note of the fact that the message μ corresponding to the write records the *view (V') of the executing thread* (τ). Roughly speaking, this view represents the happens-before information of τ, and by storing it in μ, we allow future releasing reads to learn this information.

The semantics of acquiring reads is given in Fig. 7. There are two rules for acquiring reads, depending on whether the write that it reads from is releasing.

– When the write is not releasing, the transition is covered by the rule ACQ-RF-RLX. The effect of this rule on the memory state is identical to the rule for relaxed reads (rule READ-RLX in Fig. 4).
– When the write is releasing, we use the rule ACQ-RF-REL. The difference here is that the read causes the reading thread's view to synchronise with

the view of the write that it reads from, i.e., the new view is determined by $\mathcal{V}(\tau) \sqcup V$, where

$$V_1 \sqcup V_2 \mathrel{\widehat{=}} \lambda x \in \mathit{Var}.\, \max\{V_1(x), V_2(x)\}.$$

Given $V_1, V_2 \in \mathsf{View}$, $V_1 \sqcup V_2$ defines a new view mapping each variable x to the larger timestamp between $V_1(x)$ and $V_2(x)$.

View-Transfer Assertions. With the release-acquire semantics in place, we formalise the view-transfer assertions. We first define a function filt that filters messages that τ can observe according to a predicate π over messages:

$$\mathsf{filt}(\tau, \pi) \mathrel{\widehat{=}} \lambda\alpha.\ \{\mu \in \mathsf{obs}(\tau)(\alpha) \mid \pi(\mu)\}$$

Thus, for example $\mathsf{filt}(\tau, (\lambda\mu.\ \mathsf{var}(\mu) = x \wedge \mathsf{val}(\mu) = v))$ returns the set of all messages on variable x that thread τ can observe whose value is v. Filters allow one to define assertions such as:

$$x \overset{\tau}{>} \mathsf{xv} \mathrel{\widehat{=}} x\langle\!\langle\tau \wedge \mathsf{filt}(\tau, (\lambda\mu.\ \mathsf{var}(\mu) = x \wedge \mathsf{val}(\mu) > \mathsf{xv})) \neq \emptyset$$

which is used in W_c in Fig. 6. This assertion states that τ's view of x is maximal, and that the value of this last message is greater than xv. Note that if $x\langle\!\langle\tau$, then $|\tau\rangle x$ is a singleton set since τ can read from exactly one message (which is a write on x). If $\mathsf{filt}(\tau, (\lambda\mu.\ \mathsf{var}(\mu) = x \wedge \mathsf{val}(\mu) > \mathsf{xv}))$, then this last message on x has a value greater that xv, precisely when the set returned by filt is non-empty.

The view-transfer assertion is then defined as follows:

$$\begin{aligned}\pi \overset{\tau}{\rightsquigarrow} \varphi \mathrel{\widehat{=}} \lambda\alpha.\ &\textbf{let } \alpha = \langle\sigma, \langle m, \mathcal{V}\rangle\rangle \textbf{ in} \\ &\forall\mu \in \mathsf{filt}(\tau, \pi)(\alpha). \\ &\qquad \mathsf{ord}(\mu) = \mathtt{R} \wedge \varphi(\langle\sigma, \langle m, \mathcal{V}[\tau \mapsto \mathcal{V}(\tau) \sqcup \mathsf{view}(\mu)]\rangle\rangle)\end{aligned}$$

First we filter the messages observable to τ satisfying π, and require that all of these are releasing. Moreover, the effect of reading these via an acquiring read results in a state that satisfies φ. In particular, in $\pi \overset{\tau}{\rightsquigarrow} \varphi$, we require that φ holds in the state *after* τ reads a value satisfying π, rather than in the current state. A special case of the view-transfer predicate is the following:

$$(x = \mathsf{xv}) \overset{\tau}{\rightsquigarrow} \varphi \mathrel{\widehat{=}} (\lambda\mu.\ \mathsf{var}(\mu) = x \wedge \mathsf{val}(\mu) = \mathsf{xv}) \overset{\tau}{\rightsquigarrow} \varphi \qquad (1)$$

which ensures view transfer whenever τ reads the value xv for x. An instance of this predicate appears in the consequent of W_c in Fig. 6.

Establishing and Using View-Transfer. We make use of view transfer to establish the new assertion φ as a postcondition using the following rule:

ACQ-VIEW-TRANSFER

$$\frac{}{\left\{x = \mathsf{xv} \overset{\tau}{\rightsquigarrow} \varphi\right\} \tau \rhd \mathtt{r} \overset{\mathtt{A}}{:=} x \left\{\mathtt{r} = \mathsf{xv} \Rightarrow \varphi\right\}}$$

Given the precondition $x = \mathsf{xv} \overset{\tau}{\rightsquigarrow} \varphi$ and an acquiring read of x into $\mathtt{r}$, if the value of $\mathtt{r}$ in the post state is xv then φ must hold.

$$\{d\langle\!\langle\tau_1 \wedge 1 \notin |\tau_2\rangle f\}$$

$$\left.\begin{array}{l}\{1 \notin |\tau_2\rangle f \wedge d\langle\!\langle\tau_1\}\\ d :\overset{\mathtt{X}}{=} 1;\\ \{1 \notin |\tau_2\rangle f \wedge d \overset{\tau_1}{=} 1\}\\ f :\overset{\mathtt{R}}{=} 1\\ \{true\}\end{array}\right\| \begin{array}{l}\{f = 1 \overset{\tau_2}{\rightsquigarrow} (d \overset{\tau_2}{=} 1)\}\\ \mathtt{r} :\overset{\mathtt{A}}{=} f;\\ \{\mathtt{r} = 1 \Rightarrow d \overset{\tau_2}{=} 1\}\\ \mathtt{s} :\overset{\mathtt{X}}{=} d\\ \{\mathtt{r} = 1 \Rightarrow \mathtt{s} = 1\}\end{array}$$

$$\{\mathtt{r} = 1 \Rightarrow \mathtt{s} = 1\}$$

Fig. 8. Proof of the message passing litmus test

Although ACQ-VIEW-TRANSFER is generic, establishing it is non-trivial. We make use of the following rule, which establishes a generalised form of the view-transfer assertion in our earlier work [17,20,23].

REL-DEF-VIEW-SET

$$\frac{\forall i \in S.\ x \neq y_i}{\{\mathsf{xv} \notin |\tau'\rangle x \wedge (\forall i \in S. y_i \overset{\tau}{=} \mathsf{yv}_i)\}\tau \rhd x :\overset{\mathtt{R}}{=} \mathsf{xv}\{x = \mathsf{xv} \overset{\tau'}{\rightsquigarrow} (\forall i \in S. y_i \overset{\tau'}{=} \mathsf{yv}_i)\}}$$

The rule REL-DEF-VIEW-SET is used to establish $x = \mathsf{xv} \overset{\tau'}{\rightsquigarrow} (\forall i. y_i \overset{\tau'}{=} \mathsf{yv}_i)$ in the post state via the execution of $x :\overset{\mathtt{R}}{=} \mathsf{xv}$ by τ where τ' is an arbitrarily chosen thread (and may include $\tau' = \tau$). Let ψ be the postcondition. If $\tau = \tau'$, then because $(\forall i. y_i \overset{\tau}{=} \mathsf{yv}_i)$ holds in the pre-state and the assignment does not modify any y_i, $(\forall i. y_i \overset{\tau}{=} \mathsf{yv}_i)$ holds in the post-state, which implies ψ. The more interesting case is when $\tau \neq \tau'$. Here, we make use of the assumption that τ' cannot read xv for x. Then, because $(\forall i. y_i \overset{\tau}{=} \mathsf{yv}_i)$ holds, after τ executes a releasing write on x with value xv, if thread τ' then reads xv for x, its state must change so that $(\forall i. y_i \overset{\tau'}{=} \mathsf{yv}_i)$ holds.

Note that the overall transfer of the assertion $(\forall i. y_i \overset{\tau}{=} \mathsf{yv}_i)$ (in thread τ) to the assertion $(\forall i. y_i \overset{\tau'}{=} \mathsf{yv}_i)$ (in thread τ') is via *two* transitions: the first is the releasing write on x (of value xv) and the second is the acquiring read of x by τ' (of the same value xv).

Example 6. We demonstrate the view transfer rules on a simpler message passing litmus test shown in Fig. 8, where the data d written by the writer thread (τ_1) is transferred to the reader thread (τ_2) via a release-acquire synchronisation on the flag (f). This is captured by the postcondition $\mathtt{r} = 1 \Rightarrow \mathtt{s} = 1$, i.e., if the value of f read by the reader is 1, then it also later reads the data d to be 1.

The proof in Fig. 8 is almost identical to our prior work [17,20], but makes use of some weaker assertions that we have introduced in this paper. For instance, the initial condition and precondition of $d :\overset{\mathtt{X}}{=} 1$ only require $d\langle\!\langle\tau_1$ instead of the stronger condition $d \overset{\tau_1}{=} 0$ in prior work.

Lemma 3. *The proof outline in Fig. 8 is valid.*

We focus on the local and global correctness of the conditional view assertion:

$$\psi \mathrel{\widehat{=}} f = 1 \overset{\tau_2}{\rightsquigarrow} (d \overset{\tau_2}{=} 1)$$

in thread τ_2. For local correctness, if $1 \notin |\tau_2\rangle f$ then ψ is trivially true (since τ_2 could never read the value 1 for f). Hence, the initial condition implies ψ. For the same reason, ψ is stable under $\{1 \notin |\tau_2\rangle f\}\tau_1 \rhd d :\overset{\mathtt{X}}{=} 1$, which establishes the stronger assertion $1 \notin |\tau_2\rangle f$ as a postcondition. Finally, ψ is stable under $\left\{1 \notin |\tau_2\rangle f \wedge d \overset{\tau_1}{=} 1\right\}\tau_1 \rhd f :\overset{\mathtt{R}}{=} 1$ by REL-DEF-VIEW-SET discussed above.

With this formal framework in place, we now state our main result:

Lemma 4. *The proof outline in Fig. 6 is valid.*

Note that this has only been checked manually as it relies on a generalised set of assertions which have not yet been encoded in Isabelle/HOL. We leave this as future work.

5 Related Work and Discussion

There has been immense progress in relaxed memory models in the past few decades, from formal models and compilation correctness results to library abstractions and verification frameworks and tools. This has helped uncover numerous bugs in real software as well as inconsistencies in architecture and language specifications. We present some related works below and discuss some key challenges and opportunities for future work. Given the scale of works on this topic, the review below is almost certainly incomplete. More comprehensive reviews may be found elsewhere, e.g., a recent survey by Su and Colvin [55].

Other Memory Models. This paper has focussed on reasoning techniques for relaxed memory models, whereby the principles for reasoning about inter-thread interference stem from existing frameworks for SC memory (cf [37]). Although this tutorial has focussed on the RC11 memory model, our methods have been applied more generally to other memory models, e.g., total store order (TSO) [22], persistent x86-TSO [11], strong release-acquire [37] and C11 with relaxed dependencies [60,61]. There have also been some attempts to unify reasoning principles across different memory models [22,37].

Yet, there is more work to do as the memory models increase in complexity, e.g., to support future architectures comprising heterogeneous computing [27] or distributed shared memory [6,7]. Here, the types of assertions that are likely to be useful for reasoning are currently unclear. Moreover, when memory models become too relaxed (i.e., they lose program order within a thread), they induce "interference" from within the same thread, which induces many additional complications [60,61]. Exactly how intra-thread interference can be efficiently handled is currently unknown. One possible approach is a generalisation of the technique by Coughlin et al [16], which uses rely conditions to describe

weak memory effects. Another is the development of new separation logics over mixed semantics comprising both axiomatic and operational principles [29].

Refinement and Library Abstraction. Several works have consider the problem of developing abstractions of common data structures and algorithms, over both declarative [7, 47, 54] and operational [10, 19, 52] semantics. While the former supports modular reasoning, the latter can be trivially linked to stepwise refinement and standard simulation-based proof techniques.

One difficulty here is that the precise notions of correctness for relaxed libraries (e.g., linearizability [30]) can be unclear since library programs not only affect the happens-before relations within the library, but also the client program's context [24, 28, 47, 54]. Moreover, a total order of operation invocations and responses, as used in the definition of linearizability, has less meaning here since it is possible for an operation to be after another (according to this total order), but still be executed concurrently since the effects of the first are not seen by the second. As such, an alternative set of works have taken (contextual) refinement directly as the correctness condition [13, 19, 52], *without* a detour through linearizability. Like linearizability, these approaches *use* an atomic sequential specification, but define the happens-before interactions between different libraries and the client context at this abstract level.

More recently, approaches to library construction have placed *compositionality* at the forefront [59], which have discovered that compositionality and linearizability are intrinsically linked. It would be interesting to study these ideas through the lens of relaxed memory models.

Logics. We have focussed on a particular line of work that is based on Owicki-Gries reasoning. Other works have considered generic methods for building rely-guarantee frameworks in causally consistent memory models [37]. Another line of work is based on separation logic, which has been applied to several memory models [21, 29, 32, 56–58]. Although the approaches are different, a common feature between all of these frameworks use bespoke (non-standard) assertions to support reasoning about relaxed memory effects. There are some efforts aimed at unifying reasoning across multiple memory models [22], but these techniques still assume certain restrictions such as causal consistency, which not all memory models satisfy.

Work is therefore needed on verifying more examples to generate libraries of proofs across different memory models. These will, in particular, shed more light into the types of synchronisation patterns that are most common amongst different algorithms, allowing one to develop more efficient strategies for their verification. Such work could also involve development of decidable logics that sacrifice completeness to boost scalability and automation.

Tool Support. The complexity of reasoning about relaxed memory behaviours has meant that developing tool support is critical. However, building tools is challenging since checking safety problems is often undecidable [2, 12, 36, 53]. There are now several model checkers built specifically for program executing over a relaxed memory semantics [28, 35, 41]. However, even when implementing optimal model checking algorithms based on dynamic partial order reduc-

tion [34] or using techniques such as stateless model checking [1], such provers remain bounded in the number of threads, locations and/or values, i.e., are not fully generic. Of course, their push-button (automated verification) capabilities makes them valuable tools in the verification toolchain, so continued research into further optimisations remains critical.

Full correctness proofs have been supported by a wide range of theorem proving tools such as Isabelle/HOL [19,61], Iris [29] and Coq [57], as well as bespoke tools such as Viper [56] that support automatic deductive verification. As with the logics discussed above, further experience with proof tools is required to enable verification to scale, e.g., through more efficient encodings and/or the development or tactics that allow proof obligations to be automatically discharged. Our experiences with Isabelle/HOL [17,20,61] have shown that once the Hoare triples for (atomic) assignment commands have been established as Isabelle lemmas, proofs of Hoare triples from a program's proof outline can typically be automatically discharged using calls to Isabelle's `sledgehammer` tool. Note that `sledgehammer` itself calls a number of internal tactics and SMT solvers to discharge proof obligations. As these tactics and SMT solvers improve, so will our ability to verify relaxed memory programs. However, this experience still pales in comparison to the speed at which Isabelle is able to discharge proofs under SC (see Lemma 1).

Acknowledgements We thank the anonymous reviewers for suggestions that have helped improve the paper.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 353–367. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_28
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: A load-buffer semantics for total store ordering. Log. Methods Comput. Sci. **14**(1) (2018). https://doi.org/10.23638/LMCS-14(1:9)2018
3. Adve, S.V., Boehm, H.J.: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM **53**(8), 90–101 (Aug 2010). https://doi.org/10.1145/1787234.1787255
4. Ahamad, M., Hutto, P.W., John, R.: Implementing and programming causal distributed shared memory. In: ICDCS. pp. 274–281. IEEE Computer Society (1991). https://doi.org/10.1109/ICDCS.1991.148677
5. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014). https://doi.org/10.1145/2627752
6. Ambal, G., Dongol, B., Eran, H., Klimis, V., Lahav, O., Raad, A.: Semantics of remote direct memory access: Operational and declarative models of RDMA on TSO architectures. Proc. ACM Program. Lang. **8**(OOPSLA2), 1982–2009 (2024). https://doi.org/10.1145/3689781

7. Ambal, G., Hodgkins, G., Madler, M., Chockler, G.V., Dongol, B., Izraelevitz, J., Raad, A., Vafeiadis, V.: A verified high-performance composable object library for remote direct memory access. Proc. ACM Program. Lang. **10**(POPL), 2051–2082 (2026). https://doi.org/10.1145/3776713
8. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP. Lecture Notes in Computer Science, vol. 9032, pp. 283–307. Springer (2015). https://doi.org/10.1007/978-3-662-46669-8_12
9. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL. pp. 55–66. ACM (2011). https://doi.org/10.1145/1926385.1926394
10. Bila, E.V., Dongol, B.: A verified durable transactional mutex lock for persistent x86-TSO. Formal Methods Syst. Des. **64**(1), 237–282 (2024). https://doi.org/10.1007/S10703-024-00462-1
11. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based Owicki-Gries reasoning for persistent x86-TSO. In: Sergey, I. (ed.) ESOP. Lecture Notes in Computer Science, vol. 13240, pp. 234–261. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_9
12. Bouajjani, A., Enea, C., Mutluergil, S.O., Tasiran, S.: Reasoning about TSO programs using reduction and abstraction. In: Chockler, H., Weissenbacher, G. (eds.) CAV. Lecture Notes in Computer Science, vol. 10982, pp. 336–353. Springer (2018). https://doi.org/10.1007/978-3-319-96142-2_21
13. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) ESOP. Lecture Notes in Computer Science, vol. 7211, pp. 87–107. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_5
14. Castañeda, A., Chockler, G., Dongol, B., Lahav, O.: What Cannot Be Implemented on Weak Memory? In: Alistarh, D. (ed.) DISC. Leibniz International Proceedings in Informatics (LIPIcs), vol. 319, pp. 11:1–11:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.DISC.2024.11
15. Cho, K., Lee, S., Raad, A., Kang, J.: Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 16–31. ACM (2021). https://doi.org/10.1145/3453483.3454027
16. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. Formal Aspects Comput. **35**(2), 8:1–8:30 (2023). https://doi.org/10.1145/3574137
17. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries Reasoning for C11 RAR. In: Hirschfeld, R., Pape, T. (eds.) ECOOP. Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 11:1–11:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.11
18. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries reasoning for C11 RAR (artifact). Dagstuhl Artifacts Ser. **6**(2), 15:1–15:2 (2020). https://doi.org/10.4230/DARTS.6.2.15
19. Dalvandi, S., Dongol, B.: Implementing and verifying release-acquire transactional memory in C11. Proc. ACM Program. Lang. **6**(OOPSLA2), 1817–1844 (2022). https://doi.org/10.1145/3563352

20. Dalvandi, S., Dongol, B., Doherty, S., Wehrheim, H.: Integrating Owicki-Gries for C11-Style memory models into Isabelle/HOL. J. Autom. Reason. **66**(1), 141–171 (2022). https://doi.org/10.1007/S10817-021-09610-2
21. Dang, H., Jourdan, J., Kaiser, J., Dreyer, D.: Rustbelt meets relaxed memory. Proc. ACM Program. Lang. **4**(POPL), 34:1–34:29 (2020). https://doi.org/10.1145/3371102
22. Doherty, S., Dalvandi, S., Dongol, B., Wehrheim, H.: Unifying operational weak memory verification: An axiomatic approach. ACM Trans. Comput. Log. **23**(4), 27:1–27:39 (2022). https://doi.org/10.1145/3545117
23. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) PPoPP. pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702
24. Dongol, B., Jagadeesan, R., Riely, J., Armstrong, A.: On abstraction and compositionality for weak-memory linearisability. In: Dillig, I., Palsberg, J. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 10747, pp. 183–204. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_9
25. Gharachorloo, K., Adve, S.V., Gupta, A., Hennessy, J.L., Hill, M.D.: Programming for different memory consistency models. J. Parallel Distributed Comput. **15**(4), 399–407 (1992). https://doi.org/10.1016/0743-7315(92)90052-O
26. Gharachorloo, K., Gupta, A., Hennessy, J.L.: Performance evaluation of memory consistency models for shared memory multiprocessors. In: Patterson, D.A., Rau, B. (eds.) ASPLOS. pp. 245–257. ACM Press (1991). https://doi.org/10.1145/106972.106997
27. Goens, A., Chakraborty, S., Sarkar, S., Agarwal, S., Oswald, N., Nagarajan, V.: Compound memory models. Proc. ACM Program. Lang. **7**(PLDI), 1145–1168 (2023). https://doi.org/10.1145/3591267
28. Golovin, P., Kokologiannakis, M., Vafeiadis, V.: RELINCHE: automatically checking linearizability under relaxed memory consistency. Proc. ACM Program. Lang. **9**(POPL), 2090–2117 (2025). https://doi.org/10.1145/3704906
29. Hammond, A., Liu, Z., Pérami, T., Sewell, P., Birkedal, L., Pichon-Pharabod, J.: An axiomatic basis for computer programming on the relaxed arm-a architecture: The axsl logic. Proc. ACM Program. Lang. **8**(POPL), 604–637 (2024). https://doi.org/10.1145/3632863
30. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
31. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (Oct 1983). https://doi.org/10.1145/69575.69577
32. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: Müller, P. (ed.) ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPICS.ECOOP.2017.17
33. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) POPL. pp. 175–189. ACM (2017). https://doi.org/10.1145/3009837.3009850
34. Kokologiannakis, M., Majumdar, R., Vafeiadis, V.: Enhancing genmc's usability and performance. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. Lecture Notes in Computer Science, vol. 14571, pp. 66–84. Springer (2024). https://doi.org/10.1007/978-3-031-57249-4_4

35. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL), 1–28 (2022). https://doi.org/10.1145/3498711
36. Lahav, O., Boker, U.: What's decidable about causally consistent shared memory? ACM Trans. Program. Lang. Syst. **44**(2), 8:1–8:55 (2022). https://doi.org/10.1145/3505273
37. Lahav, O., Dongol, B., Wehrheim, H.: Rely-guarantee reasoning for causally consistent shared memory. In: Enea, C., Lal, A. (eds.) CAV. Lecture Notes in Computer Science, vol. 13964, pp. 206–229. Springer (2023). https://doi.org/10.1007/978-3-031-37706-8_11
38. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. SIGPLAN Not. **52**(6), 618–632 (Jun 2017). https://doi.org/10.1145/3140587.3062352
39. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **3**(2), 125–143 (1977). https://doi.org/10.1109/TSE.1977.229904
40. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
41. de León, H.P., Furbach, F., Heljanko, K., Meyer, R.: BMC with memory models as modules. In: Bjørner, N.S., Gurfinkel, A. (eds.) FMCAD. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603021
42. Manson, J., Pugh, W.W., Adve, S.V.: The Java memory model. In: Palsberg, J., Abadi, M. (eds.) POPL. pp. 378–391. ACM (2005). https://doi.org/10.1145/1040305.1040336
43. Nipkow, T., Nieto, L.P.: Owicki/Gries in Isabelle/HOL. In: Finance, J. (ed.) FASE. vol. 1577, pp. 188–203. Springer (1999). https://doi.org/10.1007/978-3-540-49020-3_13
44. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta informatica **6**(4), 319–340 (1976)
45. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: Müller, P. (ed.) ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 599–625. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_22
46. Pulte, C., Pichon-Pharabod, J., Kang, J., Lee, S., Hur, C.: Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In: McKinley, K.S., Fisher, K. (eds.) PLDI. pp. 1–15. ACM (2019). https://doi.org/10.1145/3314221.3314624
47. Raad, A., Doko, M., Rozic, L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. Proc. ACM Program. Lang. **3**(POPL), 68:1–68:31 (2019). https://doi.org/10.1145/3290381
48. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL), 11:1–11:31 (2020). https://doi.org/10.1145/3371079
49. Semenyuk, M., Batty, M., Dongol, B.: Verifying read-copy update under RC11. In: Ferreira, C., Willemse, T.A.C. (eds.) SEFM. Lecture Notes in Computer Science, vol. 14323, pp. 301–319. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_17
50. Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP. pp. 27–51. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

51. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010). https://doi.org/10.1145/1785414.1785443
52. Singh, A.K., Lahav, O.: An operational approach to library abstraction under relaxed memory concurrency. Proc. ACM Program. Lang. **7**(POPL), 1542–1572 (2023). https://doi.org/10.1145/3571246
53. Singh, A.K., Lahav, O.: Decidable verification under localized release-acquire concurrency. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. Lecture Notes in Computer Science, vol. 14572, pp. 235–254. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_12
54. Stefanesco, L., Raad, A., Vafeiadis, V.: Specifying and verifying persistent libraries. In: ESOP (2). Lecture Notes in Computer Science, vol. 14577, pp. 185–211. Springer (2024)
55. Su, R.C., Colvin, R.J.: Weak memory model formalisms: Introduction and survey. Concurrency and Computation: Practice and Experience **38**(2), e70484 (2026). https://doi.org/10.1002/cpe.70484
56. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs. In: Beyer, D., Huisman, M. (eds.) TACAS. Lecture Notes in Computer Science, vol. 10805, pp. 190–209. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_11
57. Tassarotti, J., Dreyer, D., Vafeiadis, V.: Verifying read-copy-update in a logic for weak memory. In: Grove, D., Blackburn, S.M. (eds.) PLDI. pp. 110–120. ACM (2015). https://doi.org/10.1145/2737924.2737992
58. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: Black, A.P., Millstein, T.D. (eds.) OOPSLA. pp. 691–707. ACM (2014). https://doi.org/10.1145/2660193.2660243
59. Vale, A.O., Shao, Z., Chen, Y.: A compositional theory of linearizability. Proc. ACM Program. Lang. **7**(POPL), 1089–1120 (2023). https://doi.org/10.1145/3571231
60. Wright, D., Batty, M., Dongol, B.: Owicki-Gries reasoning for C11 programs with relaxed dependencies. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) FM. Lecture Notes in Computer Science, vol. 13047, pp. 237–254. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_13
61. Wright, D., Dalvandi, S., Batty, M., Dongol, B.: Mechanised operational reasoning for C11 programs with relaxed dependencies. Formal Aspects Comput. **35**(2), 10:1–10:27 (2023). https://doi.org/10.1145/3580285