A separation logic for a promising semantics

Kasper Svendsen, Jean Pichon-Pharabod[1], Marko Doko[2], Ori Lahav[3], and Viktor Vafeiadis[2]

[1] University of Cambridge
[2] MPI-SWS
[3] Tel Aviv University

Abstract. We present SLR, the first expressive program logic for reasoning about concurrent programs under a weak memory model addressing the out-of-thin-air problem. Our logic includes the standard features from existing logics, such as RSL and GPS, that were previously known to be sound only under stronger memory models: (1) separation, (2) per-location invariants, and (3) ownership transfer via release-acquire synchronisation—as well as novel features for reasoning about (4) the absence of out-of-thin-air behaviours and (5) coherence. The logic is proved sound over the recent "promising" memory model of Kang et al., using a substantially different argument to soundness proofs of logics for simpler memory models.

1 Introduction

Recent years have seen the emergence of several program logics [2, 6, 8, 16, 23, 24, 26–28] for reasoning about programs under weak memory models. These program logics are valuable tools for structuring program correctness proofs, and enabling programmers to reason about the correctness of their programs without necessarily knowing the formal semantics of the programming language. So far, however, they have only been applied to relatively strong memory models (such as TSO [19] or release/acquire consistency [15] that can be expressed as a constraint on individual candidate program executions) and provide little to no reasoning principles to deal with C/C++ "relaxed" accesses.

The main reason for this gap is that the behaviour of relaxed accesses is notoriously hard to specify [3, 5]. Up until recently, memory models have either been too strong (e.g., [5, 14, 17]), forbidding some behaviours observed with modern hardware and compilers, or they have been too weak (e.g., [4]), allowing so-called out-of-thin-air (OOTA) behaviour even though it does not occur in practice and is highly problematic.

One observable behaviour forbidden by the strong models is the load buffering behaviour illustrated by the example below, which, when started with both locations x and y containing 0, can end with both r_1 and r_2 containing 1. This behaviour is observable on certain ARMv7 processors after the compiler optimises $r_2 + 1 - r_2$ to 1.

$$\begin{array}{l} r_1 := [x]_{\texttt{rlx}};\ \textit{// reads 1} \\ [y]_{\texttt{rlx}} := r_1 \end{array} \;\Bigg\|\; \begin{array}{l} r_2 := [y]_{\texttt{rlx}};\ \textit{// reads 1} \\ [x]_{\texttt{rlx}} := r_2 + 1 - r_2 \end{array} \qquad \text{(LB+data+fakedep)}$$

However, one OOTA behaviour they should not allow is the following example by Boehm and Demsky [5]. When started with two completely disjoint lists a and b, by updating them separately in parallel, it should not be allowed to end with a and b pointing to each other, as that would violate physical separation (for simplicity, in these lists, a location just holds the address of the next element):

$$\begin{array}{l\|l} r_1 := [a]_{\texttt{rlx}};\ \text{// reads } b & r_2 := [b]_{\texttt{rlx}};\ \text{// reads } a \\ [r_1]_{\texttt{rlx}} := a & [r_2]_{\texttt{rlx}} := b \end{array} \qquad \text{(Disjoint-Lists)}$$

Because of this specification gap, program logics either do not reason about relaxed accesses, or they assume overly strengthened models that disallow some behaviours that occur in practice (as discussed in Sect. 5).

Recently, there have been several proposals of programming language memory models that allow load buffering behaviour, but forbid obvious out-of-thin-air behaviours [10, 13, 20]. This development has enabled us to develop a program logic that provides expressive reasoning principles for relaxed accesses, without relying on overly strong models.

In this paper, we present SLR, a separation logic based on RSL [27], extended with strong reasoning principles for relaxed accesses, which we prove sound over the recent "promising" semantics of Kang et al. [13]. SLR features per-location invariants [27] and physical separation [22], as well as novel assertions that we use to show the absence of *OOTA behaviours* and to reason about various *coherence* examples. (Coherence is a property of memory models that requires the existence of a per-location total order on writes that reads respect.)

There are two main contributions of this work.

First, SLR is the first logic which can prove absence of OOTA in all the standard litmus tests. As such, it provides more evidence to the claim that the promising semantics solves the out-of-thin-air problem in a satisfactory way. The paper that introduced the promising semantics [13] comes with three DRF theorems and a simplistic value logic. These reasoning principles are enough to show absence of some simple out-of-thin-air behaviours, but it is still very easy to end up beyond the reasoning power of these two techniques. For instance, they cannot be used to prove that $r_1 = 0$ in the following "random number generator" litmus test[4], where both the x and y locations initially hold 0.

$$\begin{array}{l\|l} r_1 := [x]_{\texttt{rlx}}; & r_2 := [y]_{\texttt{rlx}}; \\ [y]_{\texttt{rlx}} := r_1 + 1 & [x]_{\texttt{rlx}} := r_2 \end{array} \qquad \text{(RNG)}$$

The subtlety of this litmus test is the following: if the first thread reads a certain value v from x, then it writes $v + 1$ to y, which the second thread can read, and write to x; this, however, does not enable the first thread to read $v + 1$. SLR features novel assertions that allow it to handle those and other examples, as shown in the following section.

[4] The litmus test is called this way because some early attempts to solve the OOTA problem allowed this example to return arbitrary values for x and y.

The second major contribution is the proof of soundness of SLR over the promising semantics [13][5]. The promising semantics is an operational model that represents memory as a collection of timestamped write messages. Besides the usual steps that execute the next command of a thread, the model has a non-standard step that allows a thread to promise to perform a write in the future, provided that it can guarantee to be able to fulfil its promise. After a write is promised, other threads may read from that write as if it had already happened. Promises allow the load-store reordering needed to exhibit the load buffering behaviour above, and yet seem, from a series of litmus tests, constrained enough so as to not introduce out-of-thin-air behaviour.

Since the promising model is rather different from all other (operational and axiomatic) memory models for which a program logic has been developed, none of the existing approaches for proving soundness of concurrent program logics are applicable to our setting. Two key difficulties in the soundness proof come from dealing with promise steps.

1. Promises are very non-modular, as they can occur at every execution point and can affect locations that may only be accessed much later in the program.
2. Since promised writes can be immediately read by other threads, the soundness proof has to impose the same invariants on promised writes as the ones it imposes on ordinary writes (e.g., that only values satisfying the location's protocol are written). In a logic supporting ownership transfer,[6] however, establishing those invariants is challenging, because a thread may promise to write to x even without having permission to write to x.

To deal with the first challenge, our proof decouples promising steps from ordinary execution steps. We define two semantics of Hoare triples—one "promising", with respect to the full promising semantics, and one "non-promising", with respect to the promising semantics without promising steps—and prove that every Hoare triple that is correct with respect to its non-promising interpretation is also correct with respect to its promising interpretation. This way, we modularise reasoning about promise steps. Even in the non-promising semantics, however, we do allow threads to have outstanding promises. The main difference in the non-promising semantics is that threads are not allowed to issue new promises.

To resolve the second challenge, we observe that in programs verified by SLR, a thread may promise to write to x only if it is able to acquire the necessary write permission before performing the actual write. This follows from promise certification: the promising semantics requires all promises to be certifiable; that is, for every state of the promising machine, there must exist a non-promising execution of the machine that fulfils all outstanding promises.

We present the SLR assertions and rules informally in Sect. 2. We then give an overview of the promising semantics of Kang et al. [13] in Sect. 3, and use it

[5] As the promising semantics comes with formal proofs of correctness of all the expected local program transformations and of compilation schemes to the x86-TSO, Power, and ARMv8-POP architectures [21], SLR is sound for these architectures too.

[6] Supporting ownership transfer is necessary to provide useful rules for C11 release and acquire accesses.

$$
\begin{array}{rlll}
e \in \mathit{Expr} ::= & n & \text{integer} & s \in \mathit{Stm} ::= \textbf{skip} \mid s_1; s_2 \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \\
\mid & r & \text{register} & \quad\quad\quad \mid \textbf{while } e \textbf{ do } s \mid r := e \mid r := [e]_{\mathtt{rlx}} \\
\mid & e_1 \mathbin{op} e_2 & \text{arithmetic} & \quad\quad\quad \mid r := [e]_{\mathtt{acq}} \mid [e_1]_{\mathtt{rlx}} := e_2 \mid [e_1]_{\mathtt{rel}} := e_2
\end{array}
$$

Fig. 1. Syntax of the programming language.

in Sect. 4 to explain the proof of soundness of SLR. We discuss related work in Sect. 5. *Details of the rules of SLR and its soundness proof can be found in our technical appendix [1].*

2 Our logic

The novelty of our program logic is to allow non-trivial reasoning about relaxed accesses. Unlike release/acquire accesses, relaxed accesses do not induce synchronisation between threads, so the usual approach of program logics, which relies on ownership transfer, does not apply. Therefore, in addition to reasoning about ownership transfer like a standard separation logic, our logic supports reasoning about relaxed accesses by collecting information about what reads have been observed, and in which order. When combined with information about which writes have been performed, we can deduce that certain executions are impossible.

For concreteness, we consider a minimal "WHILE" programming language with expressions, $e \in \mathit{Expr}$, and statements, $s \in \mathit{Stm}$, whose syntax is given in Fig. 1. Besides local register assignments, statements also include memory reads with relaxed or acquire mode, and memory writes with relaxed or release mode.

2.1 The assertions of the logic

The SLR assertion language is generated by the following grammar, where N, l, v, t, π and X all range over a simply-typed term language which we assume includes booleans, locations, values and expressions of the programming language, fractional permissions, and timestamps, and is closed under pairing, finite sets, and sequences. By convention, we assume that l, v, t, π and X range over terms of type location, value, timestamp, permission and sets of pairs of values, and timestamps, respectively.

$$
\begin{aligned}
P, Q \in \mathit{Assn} ::= {} & \bot \mid \top \mid P \lor Q \mid P \land Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid N_1 = N_2 \mid \phi(N) \\
& \mid P * Q \mid \mathsf{Rel}(l, \phi) \mid \mathsf{Acq}(l, \phi) \mid \mathsf{O}(l, v, t) \mid \mathsf{W}^{\pi}(l, X) \mid \nabla P \\
\phi \in \mathit{Pred} ::= {} & \lambda x.\, P
\end{aligned}
$$

The grammar contains the standard operators from first order logic and separation logic, the Rel and Acq assertions from RSL [27], and a few novel constructs.

$\mathsf{Rel}(l, \phi)$ grants permission to perform a release write to location l and transfer away the invariant $\phi(v)$, where v is the value written to that location. Conversely, $\mathsf{Acq}(l, \phi)$ grants permission to perform an acquire read from location l and gain access to the invarant $\phi(v)$, where v is the value returned by the read.

The first novel assertion form, $\mathsf{O}(l, v, t)$, records the fact that location l was observed to have value v at timestamp t. The timestamp is used to order it with other reads from the same location. The information this assertion provides is very weak: it merely says that the owner of the assertion has observed that value, it does not imply that any other thread has ever observed it.

The other novel assertion form, $\mathsf{W}^{\pi}(l, X)$, asserts ownership of location l and records a set of writes X to that location. The fractional permission $\pi \in \mathbb{Q}$ indicates whether ownership is shared or exclusive. Full permission, $\pi = 1$, confers exclusive ownership of location l and ensures that X is the set of all writes to location l; any fraction, $0 < \pi < 1$, confers shared ownership and enforces that X is a lower-bound on the set of writes to location l. The order of writes to l is tracked through timestamps; the set X is thus a set of pairs consisting of the value and the timestamp of the write.

In examples where we only need to refer to the order of writes and not the exact timestamps, we write $\mathsf{W}^{\pi}(x, \ell)$, where $\ell = [v_1, ..., v_n]$ is a list of values, as shorthand for $\exists t_1, ..., t_n.\, t_1 > t_2 > \cdots > t_n * \mathsf{W}^{\pi}(x, \{(v_1, t_1), ..., (v_n, t_n)\})$. The $\mathsf{W}^{\pi}(x, \ell)$ assertion thus expresses ownership of location x with permission π, and that the writes to x are given by the list ℓ in order, with the most recent write at the front of the list.

Relation between reads and writes. Records of reads and writes can be confronted by the thread owning the exclusive write assertion: all reads must have read values that were written. This is captured formally by the following property:

$$\mathsf{W}^1(x, X) * \mathsf{O}(x, a, t) \Rrightarrow \mathsf{W}^1(x, X) * \mathsf{O}(x, a, t) * (a, t) \in X \quad \text{(Reads-from-Write)}$$

Random number generator. These assertions allow us to reason about the "random number generator" litmus test from the Introduction, and to show that it cannot read arbitrarily large values. As discussed in the Introduction, capturing the set of values that are written to x, as made possible by the "invariant-based program logic" of Kang et al. [13, §5.5] and of Jeffrey and Riley [10, §6], is not enough, and we make use of our stronger reasoning principles. We use $\mathsf{O}(x, a, t)$ to record what values reads read from each location, and $\mathsf{W}^1(x, \ell)$ to record what sequences of values were written to each location, and then confront these records at the end of the execution. The proof sketch is then as follows:

$$\begin{array}{l||l}
\{\mathsf{W}^1(y, [0]) * \ldots\} & \{\mathsf{W}^1(x, [0]) * \ldots\} \\
r_1 := [x]_{\mathtt{rlx}}; & r_2 := [y]_{\mathtt{rlx}}; \\
\{\mathsf{W}^1(y, [0]) * \mathsf{O}(x, r_1, \_) * \ldots\} & \{\mathsf{W}^1(x, [0]) * \mathsf{O}(y, r_2, \_) * \ldots\} \\
[y]_{\mathtt{rlx}} := r_1 + 1 & [x]_{\mathtt{rlx}} := r_2 \\
\{\mathsf{W}^1(y, [r_1 + 1; 0]) * \mathsf{O}(x, r_1, \_) * \ldots\} & \{\mathsf{W}^1(x, [r_2; 0]) * \mathsf{O}(y, r_2, \_) * \ldots\}
\end{array}$$

At the end of the execution, we are able to draw conclusions about the values of the registers. From $\mathsf{W}^1(x, [r_2; 0])$ and $\mathsf{O}(x, r_1, \_)$, we know that $r_1 \in \{r_2, 0\}$ by rule Reads-from-Write. Similarly, we know that $r_2 \in \{r_1 + 1, 0\}$, and so we can conclude that $r_1 = 0$. We discuss the distribution of resources at the beginning of a program, and their collection at the end of a program, in Theorem 2. Note that we are unable to establish what values the reads read before the end of the litmus test. Indeed, before the end of the execution, nothing enforces that there are no further writes that reads could read from.

2.2 The rules of the logic for relaxed accesses

We now introduce the rules of our logic by focusing on the rules for *relaxed* accesses. In addition, we support the standard rules from separation logic and Hoare logic, rules for release/acquire accesses (§2.4), and the following consequence rule:

$$\frac{P \Rrightarrow P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \Rrightarrow Q}{\vdash \{P\}\, c\, \{Q\}} \qquad \text{(CONSEQ)}$$

which allows one to use “view shifting” implications to strengthen the precondition and weaken the postcondition.

The rules for relaxed accesses are adapted from the rules of RSL [27] for release/acquire accesses, but use our novel resources to track the more subtle behaviour of relaxed accesses. Since relaxed accesses do not introduce synchronisation, they cannot be used to transfer ownership; they can, however, be used to transfer information. For this reason, as in RSL [27], we associate a predicate ϕ on values to a location x using paired $\mathsf{Rel}(x, \phi)$ and $\mathsf{Acq}(x, \phi)$ resources, for writers and readers, respectively. To write v to x, a writer has to provide $\phi(v)$, and in exchange, when reading v from x, a reader obtains $\phi(v)$. However, here, relaxed writes can only send *pure* predicates (i.e., ones which do not assert ownership of any resources), and relaxed reads can only obtain the assertion from the predicate guarded by a modality ∇[7] that only pure assertions filter through: if P is pure, then $\nabla P \Longrightarrow P$. All assertions expressible in first-order logic are pure.

Relaxed write rule. To write value v (to which the value expression e_2 evaluates) to location x (to which the location expression e_1 evaluates), the thread needs to own a write permission $\mathsf{W}^\pi(x, X)$. Moreover, it needs to provide $\phi(v)$, the assertion associated to the written value, v, to location x by the $\mathsf{Rel}(x, \phi)$ assertion. Because the write is a relaxed write, and therefore does not induce synchronisation, $\phi(v)$ has to be a pure predicate. The write rule updates the record of writes with the value written, timestamped with a timestamp newer than any timestamp for that location that the thread has observed so far; this is expressed by relating it to a previous timestamp that the thread has to provide through an $\mathsf{O}(x, \_, t)$ assertion in the precondition.

$$\frac{\phi(v) \text{ is pure}}{\vdash \left\{\begin{matrix} e_1 = x * e_2 = v * \mathsf{W}^\pi(x, X) \\ * \mathsf{Rel}(x, \phi) * \phi(v) * \mathsf{O}(x, \_, t) \end{matrix}\right\} [e_1]_{\mathtt{rlx}} := e_2 \left\{\begin{matrix} \exists t' > t. \\ \mathsf{W}^\pi(x, \{(v, t')\} \cup X) \end{matrix}\right\}} \qquad \text{(W-RLX)}$$

The $\mathsf{Rel}(x, \phi)$ assertion is duplicable, so there is no need for the rule to keep it.

[7] This ∇ modality is similar in spirit, but weaker than that of FSL [8].

In practice, $\mathsf{O}(x, \_, t)$ is taken to be that of the last read from x if it was the last operation on x, and $\mathsf{O}(x, \mathit{fst}(\max(X)), \mathit{snd}(\max(X)))$ if the last operation on x was a write, including the initial write. The latter can be obtained by

$$\mathsf{W}^{\pi}(x, X) * (v, t) \in X \Rrightarrow \mathsf{W}^{\pi}(x, X) * \mathsf{O}(x, v, t) \qquad \text{(Write-Observed)}$$

Relaxed read rule. To read from location x (to which the location expression e evaluates), the thread needs to own an $\mathsf{Acq}(x, \phi)$ assertion, which gives it the right to (almost) obtain assertion $\phi(v)$ upon reading value v from location x. The thread then keeps its $\mathsf{Acq}(x, \phi)$, and obtains an assertion $\mathsf{O}(x, r, t')$ stating that it has read the value now in register r from location x, timestamped with t'. This timestamp is no older than any timestamp for that location that the thread has observed so far, expressed again by relating it to an $\mathsf{O}(x, \_, t)$ assertion in the precondition. Moreover, it obtains the pure portion $\nabla\phi(r)$ of the assertion $\phi(r)$ corresponding to the value read in register r

$$\begin{aligned} \vdash\ & \big\{e = x * \mathsf{Acq}(x, \phi) * \mathsf{O}(x, \_, t)\big\} \\ & \quad r := [e]_{\texttt{rlx}} \\ & \big\{\exists t' \geq t.\ \mathsf{Acq}(x, \phi) * \mathsf{O}(x, r, t') * \nabla\phi(r)\big\} \end{aligned} \qquad \text{(R-RLX)}$$

Again, we can obtain $\mathsf{O}(x, v_0^x, 0)$, where v_0^x is the initial value of x, from the initial write permission for x, and distribute it to all the threads that will read from x, expressing the fact that the initial value is available to all threads, and use it as the required $\mathsf{O}(x, \_, t)$ in the precondition of the read rule.

Moreover, if a thread owns the exclusive write permission for a location x, then it can take advantage of the fact that it is the only writer at that location to obtain more precise information about its reads from that location: they will read the last value it has written to that location.

$$\begin{aligned} \vdash\ & \big\{e = x * \mathsf{Acq}(x, \phi) * \mathsf{W}^1(x, X)\big\} \\ & \quad r := [e]_{\texttt{rlx}} \\ & \big\{\exists t.\ (r, t) = \max(X) * \mathsf{Acq}(x, \phi) * \mathsf{W}^1(x, X) * \mathsf{O}(x, r, t) * \nabla\phi(r)\big\} \end{aligned} \qquad \text{(R-RLX*)}$$

Separation. With these assertions, we can straightforwardly specify and verify the Disjoint-Lists example. Ownership of an element of a list is simply expressed using a full write permission, $\mathsf{W}^1(x, X)$. This allows including the Disjoint-Lists as a snippet in a larger program where the lists can be shared before or after, and still enforce the separation property we want to establish. While this reasoning sounds underwhelming (and we elide the details), we remark that it is unsound in models that allow OOTA behaviours.

2.3 Reasoning about coherence

An important feature of many memory models is coherence, that is, the existence of a per-location total order on writes that reads respect. Coherence becomes interesting where there are multiple simultaneous writers to the same location (write/write races). In our logic, write assertions can be split and combined as follows: if $\pi_1 + \pi_2 \leq 1$, $0 < \pi_1$ and $0 < \pi_2$ then

$$\mathsf{W}^{\pi_1+\pi_2}(x, X_1 \cup X_2) \Leftrightarrow \mathsf{W}^{\pi_1}(x, X_1) * \mathsf{W}^{\pi_2}(x, X_2) \qquad \text{(Combine-Writes)}$$

To reason about coherence, the following rules capture the fact that the timestamps of the writes at a given location are all distinct, and totally ordered:

$$\mathsf{W}^{\pi}(x, X) * (v, t) \in X * (v', t') \in X * v \neq v' \Rrightarrow \mathsf{W}^{\pi}(x, X) * t \neq t' \qquad \text{(Different-Writes)}$$

$$\mathsf{W}^{\pi}(x, X) * (\_, t) \in X * (\_, t') \in X \Rrightarrow \mathsf{W}^{\pi}(x, X) * (t < t' \vee t = t' \vee t' < t) \qquad \text{(Writes-Ordered)}$$

CoRR2. One of the basic tests of coherence is the CoRR2 litmus test, which tests whether two threads can disagree on the order of two writes to the same location. The following program, starting with location x holding 0, should not be allowed to finish with $r_1 = 1 * r_2 = 2 * r_3 = 2 * r_4 = 1$, as that would mean that the third thread sees the write of 1 to x before the write of 2 to x, but that the fourth thread sees the write of 2 before the write of 1:

$$[x]_{\mathtt{rlx}} := 1 \;\Big\|\; [x]_{\mathtt{rlx}} := 2 \;\Big\|\; \begin{array}{l} r_1 := [x]_{\mathtt{rlx}}; \\ r_2 := [x]_{\mathtt{rlx}} \end{array} \;\Big\|\; \begin{array}{l} r_3 := [x]_{\mathtt{rlx}}; \\ r_4 := [x]_{\mathtt{rlx}} \end{array} \qquad \text{(CoRR2)}$$

Coherence enforces a total order on the writes to x that is respected by the reads, so if the third thread reads 1 then 2, then the fourth cannot read 2 then 1.

We use the timestamps in the $\mathsf{O}(x, a, t)$ assertions to record the order in which reads read values, and then link the timestamps of the reads with those of the writes. Because we do not transfer anything, the predicate for x is $\lambda v. \top$ again, and we elide the associated clutter below.

The proof outline for the writers just records what values have been written:

$$\begin{array}{l} \{\mathsf{W}^{1/2}(x, \{(0,0)\}) * \ldots\} \\ [x]_{\mathtt{rlx}} := 1 \\ \{\exists t_1. \mathsf{W}^{1/2}(x, \{(1, t_1), (0,0)\}) * \ldots\} \end{array} \;\Big\|\; \begin{array}{l} \{\mathsf{W}^{1/2}(x, \{(0,0)\}) * \ldots\} \\ [x]_{\mathtt{rlx}} := 2 \\ \{\exists t_2. \mathsf{W}^{1/2}(x, \{(2, t_2), (0,0)\}) * \ldots\} \end{array}$$

The proof outline for the readers just records what values have been read, and — crucially — in which order.

$$\Big\|\; \begin{array}{l} \{\mathsf{Acq}(x, \lambda v. \top) * \mathsf{O}(x, 0, 0)\} \\ r_1 := [x]_{\mathtt{rlx}}; \\ \{\exists t_a. \mathsf{Acq}(x, \lambda v. \top) * \mathsf{O}(x, r_1, t_a) * 0 \leq t_a * \ldots\} \\ r_2 := [x]_{\mathtt{rlx}} \\ \{\exists t_a, t_b. \mathsf{O}(x, r_1, t_a) * \mathsf{O}(x, r_2, t_b) * 0 \leq t_a * t_a \leq t_b\} \end{array} \;\Big\|\; \begin{array}{l} r_3 := [x]_{\mathtt{rlx}}; \\ r_4 := [x]_{\mathtt{rlx}} \end{array}$$

At the end of the program, by combining the two write permissions using rule Combine-Writes, we obtain $\mathsf{W}^1(x, \{(1, t_1), (2, t_2), (0, 0)\})$. From this, we have $t_1 < t_2$ or $t_2 < t_1$ by rules Different-Writes and Writes-Ordered. Now, assuming $r_1 = 1$ and $r_2 = 2$, we have $t_a < t_b$, and so $t_1 < t_2$ by rule Reads-from-Write. Similarly, assuming $r_3 = 2$ and $r_4 = 1$, we have $t_2 < t_1$. Therefore, we cannot have $r_1 = 1 * r_2 = 2 * r_3 = 2 * r_4 = 1$, so coherence is respected, as desired.

2.4 Handling release and acquire accesses

Next, consider release and acquire accesses, which, in addition to coherence, provide synchronisation and enable the message passing idiom.

$$\begin{array}{l} [x]_{\texttt{rlx}} := 1; \\ [y]_{\texttt{rel}} := 1 \end{array} \; \Bigg\| \; \begin{array}{l} r_1 := [y]_{\texttt{acq}}; \\ \textsf{\textbf{if}}\ r_1 = 1\ \textsf{\textbf{then}}\ r_2 := [x]_{\texttt{rlx}} \end{array} \tag{MP}$$

The first thread writes data (here, 1) to a location x, and signals that the data is ready by writing 1 to a "flag" location y with a release write. The second thread reads the flag location y with an acquire read, and, if it sees that the first thread has signalled that the data has been written, reads the data. The release/acquire pair is sufficient to ensure that the data is then visible to the second thread.

Release/acquire can be understood abstractly in terms of views [15]: a release write contains the view of the writing thread at the time of the writing, and an acquire read updates the view of the reading thread with that of the release write it is reading from. This allows one-way synchronisation of views between threads.

To handle release/acquire accesses in SLR, we can adapt the rules for relaxed accesses by enabling ownership transfer according to predicate associated with the Rel and Acq permissions. The resulting rules are strictly more powerful than the corresponding RSL [27] rules, as they also allow us to reason about coherence.

Release write rule. The release write rule is the same as for relaxed writes, but does not require the predicate to be a pure predicate, thereby allowing sending of actual resources, rather than just information:

$$\begin{array}{l} \vdash \left\{ e_1 = x * e_2 = v * \mathsf{W}^{\pi}(x, X) * \mathsf{Rel}(x, \phi) * \phi(v) * \mathsf{O}(x, \_, t) \right\} \\ \quad [e_1]_{\texttt{rel}} := e_2 \\ \quad \left\{ \exists t' \geq t.\ \mathsf{W}^{\pi}(x, \{(v, t')\} \cup X) \right\} \end{array} \tag{W-REL}$$

Acquire read rule. Symmetrically, the acquire read rule is the same as for relaxed reads, but allows the actual resource to be obtained, not just its pure portion:

$$\begin{array}{l} \vdash \left\{ e = x * \mathsf{Acq}(x, \phi) * \mathsf{O}(x, \_, t) \right\} \\ \quad r := [e]_{\texttt{acq}} \\ \quad \left\{ \exists t' \geq t.\ \mathsf{Acq}(x, \phi[r \mapsto \top]) * \mathsf{O}(x, r, t') * \phi(r) \right\} \end{array} \tag{R-ACQ}$$

We have to update ϕ to record the fact that we have obtained the resource associated with reading that value, so that we do not erroneously obtain that resource twice; $\phi[v' \mapsto P]$ stands for $\lambda v.\ \mathit{if}\ \ v = v'\ \mathit{then}\ P\ \mathit{else}\ \phi(v)$.

As for relaxed accesses, we can strengthen the read rule when the reader is also the exclusive writer to that location:

$$\begin{array}{l} \vdash \left\{ \mathsf{Acq}(x, \phi) * \mathsf{W}^1(x, X) \right\} \\ \quad r := [x]_{\texttt{acq}} \\ \quad \left\{ \begin{array}{l} \exists t.\ (r, t) = \max(X) * \mathsf{Acq}(x, \phi[r \mapsto \top]) \\ \qquad * \mathsf{W}^1(x, X) * \mathsf{O}(x, r, t) * \phi(r) \end{array} \right\} \end{array} \tag{R-ACQ*}$$

Additionally, we allow duplicating of release assertions and splitting of acquire assertions, as expressed by the following two rules.

$$\mathsf{Rel}(x,\phi) \Leftrightarrow \mathsf{Rel}(x,\phi) * \mathsf{Rel}(x,\phi) \qquad \text{(Release-Duplicate)}$$
$$\mathsf{Acq}(x, \lambda v.\, \phi_1(v) * \phi_2(v)) \Rrightarrow \mathsf{Acq}(x,\phi_1) * \mathsf{Acq}(x,\phi_2) \qquad \text{(Acquire-Split)}$$

Message passing. With these rules, we can easify verify the message passing example. Here, we want to transfer a resource from the writer to the reader, namely the state of the data, x. By transferring the write permission for the data to the reader over the "flag" location, y, we allow the reader to use it to read the data precisely. We do that by picking the predicate

$$\phi_y = \lambda v.\ v = 1 \wedge \mathsf{W}^1(x,[1;0]) \vee v \neq 1$$

for y. Since we do not transfer any resource using x, the predicate for x is $\lambda v.\top$.

The writer transfers the write permissions for x away on y using ϕ_y:

```
$\{\mathsf{W}^1(x,[0]) * \mathsf{Rel}(x,\lambda v.\top) * \mathsf{W}^1(y,[0]) * \mathsf{Rel}(y,\phi_y)\}$
$[x]_{\mathtt{rlx}} := 1;$
$\{\mathsf{W}^1(x,[1;0]) * \mathsf{W}^1(y,[0]) * \mathsf{Rel}(y,\phi_y)\}$
  $\{\mathsf{W}^1(y,\{(0,0)\}) * \mathsf{Rel}(y,\phi_y) * \phi_y(1) * \mathsf{O}(x,0,0)\}$
$[y]_{\mathtt{rel}} := 1$
  $\{\exists t_1.\, \mathsf{W}^1(y,\{(1,t_1)\} \cup \{(0,0)\}) * 0 < t_1\}$
$\{\mathsf{W}^1(y,[1;0]) * \mathsf{Rel}(y,\phi_y)\}$
```

The proof outline for the reader uses the acquire permission ϕ_y for y to obtain $\mathsf{W}^1(x,[1;0])$, which it then uses to know that it reads 1 from x.

```
|| $\{\mathsf{Acq}(y,\phi_y)) * \mathsf{O}(y,0,0) * \mathsf{Acq}(x,\lambda v.\top)\}$
|| $r_1 := [y]_{\mathtt{acq}};$
|| $\{\exists t_1^y \geq 0.\, \mathsf{Acq}(y,\phi_y[r_1 \mapsto \top]) * \mathsf{O}(y,r_1,t_1^y) * \phi_y(r_1) * \mathsf{Acq}(x,\lambda v.\top)\}$
|| $\{\phi_y(r_1) * \mathsf{Acq}(x,\lambda v.\top)\}$
|| if $r_1 = 1$ then
||   $\{\mathsf{W}^1(x,[1;0]) * \mathsf{Acq}(x,\lambda v.\top)\}$
||   $r_2 := [x]_{\mathtt{rlx}}$
||   $\{\mathsf{Acq}(x,\lambda v.\top) * \mathsf{W}^1(x,[1;0]) * (r_2 = 1)\}$
|| $\{r_1 = 1 \Longrightarrow r_2 = 1\}$
```

2.5 Plain accesses

Our formal development (in the technical appendix) also features the usual "partial ownership" $x \overset{\pi}{\mapsto} v$ assertion for "plain" (non-atomic) locations, and the usual corresponding rules.

3 The promising semantics

In this section, we provide an overview of the promising semantics [13], the model for which we prove SLR sound. Formal details can be found in [1, 13].

The promising semantics is an operational semantics that interleaves execution of the threads of a program. Relaxed behaviour is introduced in two ways:

- As in the "strong release/acquire" model [15], the memory is a pool of timestamped messages, and each thread maintains a "view" thereof. A thread may read any value that is not older than the latest value observed by the thread for the given location; in particular, this may well not be the latest value written to that particular location. Timestamps and views model non-multi-copy-atomicity: writes performed by one thread do not become simultaneously visible by all other threads.
- The operational semantics contains a non-standard step: at any point a thread can nondeterministically *promise* a write, provided that, at every point before the write is actually performed, the thread can *certify* the promise, that is, execute the write by running on its own from the current state. Promises are used to enable load-store reordering.

The behaviour of promising steps can be illustrated on the LB+data+fakedep litmus test from the Introduction. The second thread can, at the very start of the execution, promise a write of 1 to x, because it can, by running on its own from the current state, read from y (it will read 0), then write 1 to x (because $0 + 1 - 0 = 1$), thereby fulfilling its promise. On the other hand, the first thread cannot promise a write of 1 to y at the beginning of the execution, because, by running on its own, it can only read 0 from x, and therefore only write 0 to y.

3.1 Storage subsystem

Formally, the semantics keeps track of writes and promises in a *global configuration*, $gconf = \langle M, P\rangle$, where M is a memory and $P \subseteq M$ is the *promise memory.* We denote by $gconf.\mathtt{M}$ and $gconf.\mathtt{P}$ the components of $gconf$. Both *memories* are finite sets of messages, where a *message* is a tuple $\langle x :_i^o v, R@t]\rangle$, where $x \in Loc$ is the location of the message, $v \in Val$ its value, $i \in Tid$ its originating thread, $t \in Time$ its *timestamp*, R its message *view*, and $o \in \{\mathtt{rlx}, \mathtt{rel}\}$ its message mode, where *Time* is an infinite set of timestamps, densely totally ordered by $\leq$, with a minimum element, 0. (We return to views later.) We denote $m.\mathtt{loc}$, $m.\mathtt{val}$, $m.\mathtt{time}$, $m.\mathtt{view}$ and $m.\mathtt{mod}$ the components of a message m. We use the following notation to restrict memories:

$$M(i) \overset{def}{=} \{m \in M \mid m.\mathtt{tid} = i\} \qquad M(\mathtt{rel}) \overset{def}{=} \{m \in M \mid m.\mathtt{mod} = \mathtt{rel}\}$$
$$M(x) \overset{def}{=} \{m \in M \mid m.\mathtt{loc} = x\} \qquad M(\mathtt{rlx}) \overset{def}{=} \{m \in M \mid m.\mathtt{mod} = \mathtt{rlx}\}$$
$$M(i, x) \overset{def}{=} M(i) \cap M(x)$$

A global configuration *gconf* evolves in two ways. First, a message can be "promised" and be added both to $gconf.\mathtt{M}$ and $gconf.\mathtt{P}$. Second, a message can be written, in which case it is either added to $\mathit{gconf}.\mathtt{M}$, or removed from $\mathit{gconf}.\mathtt{P}$ (if it was promised before).

3.2 Thread subsystem

A *thread state* is a pair $TS = \langle \sigma, V \rangle$, where σ is the internal state of the thread and V is a *view*. We denote by $TS.\sigma$ and $TS.V$ the components of TS.

Thread internal state. The internal state σ consists of a thread store (denoted $\sigma.\mu$) that assigns values to local registers and a statement to execute (denoted $\sigma.s$). The transitions of the thread internal state are labeled with *memory actions* and are given by an ordinary sequential semantics. As these are routine, we leave their description to the technical appendix.

Views. Thread views are used to enforce coherence, that is, the existence of a per-location total order on writes that reads respect. A view is a function $V : \mathit{Loc} \to \mathit{Time}$, which records how far the thread has seen in the history of each location. To ensure that a thread does not read stale messages, its view restricts the messages the thread may read, and is increased whenever a thread observes a new message. Messages themselves also carry a view (the thread's view when the message comes from a release write, and the bottom view otherwise) which is incorporated in the thread view when the message is read by an acquire read.

Additional notations. The order on timestamps, $\leq$, is extended pointwise to views. $\bot$ and $\sqcup$ denote the natural bottom elements and join operations for views. $\{x@t\}$ denotes the view assigning t to x and $\mathtt{0}$ to other locations.

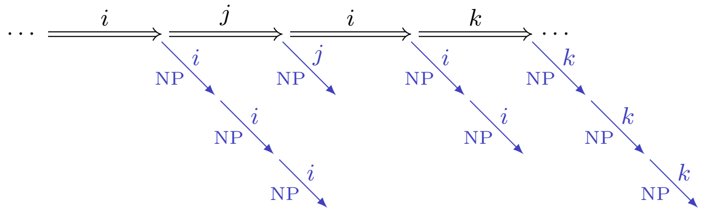
3.3 Interaction between a thread and the storage subsystem

The interaction between a thread and the storage subsystem is given in terms of transitions of *thread configurations*. Thread configurations are tuples $\langle TS, \langle M, P \rangle \rangle$, where TS is a thread state, and $\langle M, P \rangle$ is a global configuration. These transitions are labelled with $\beta \in \{\mathrm{NP}, \mathrm{prom}\}$ in order to distinguish whether they involve promises or not. A thread can:

- Make an internal transition with no effect on the storage subsystem.
- Read the value v from location x, when there is a matching message in memory that is not outdated according to the thread's view. It then updates its view accordingly: it updates the timestamp for location x and, in addition, incorporates the message view if the read is an acquire read.
- Write the value v to location x. Here, the thread picks a timestamp greater than the one of its current view for the message it adds to memory (or removes from the promise set). If the write is a release write, the message carries the view of the writing thread. Moreover, a release write to x can only be performed when the thread has already fulfilled all its promises to x.
- Non-deterministically promise a relaxed write by adding a message to both M and P.

3.4 Constraining promises

Now that we have described how threads and promises interact with memory, we can present the certification condition for promises, which is essential to avoid out-of-thin-air behaviours. Accordingly, we define another transition system, $\Longrightarrow$, on top of the previous one, which enforces that the memory remains "consistent", that is, all the promises that have been made can be certified. A thread configuration $\langle TS, \langle M, P\rangle\rangle$ is called *consistent* w.r.t. $i \in \mathit{Tid}$ if thread i can fulfil its promises by executing on its own, or more formally if $\langle TS, \langle M, P\rangle\rangle \xrightarrow{\text{NP}}{}^{*}_{i} \langle TS', \langle M', P'\rangle\rangle$ for some TS', M', P' such that $P'(i) = \emptyset$. Certification is *local*, that is, only thread i is executing during its certification; this is crucial to avoid out-of-thin-air. Further, the certification itself cannot make additional promises, as it is restricted to NP-steps. Here is a visual representation of a promise machine run, together with certifications.



The thread configuration $\Longrightarrow$-transitions allow a thread to (1) take any number of non-promising steps, provided its thread configuration at the end of the sequence of step (intuitively speaking, when it gives control back to the scheduler) is consistent, or (2) take a promising step, again provided that its thread configuration after the step is consistent.

3.5 Full machine

Finally, the full machine transitions simply lift the thread configuration $\Longrightarrow$-transitions to the machine level. A *machine state* is a tuple $\mathbf{MS} = \langle \mathcal{TS}, \langle M, P\rangle\rangle$, where $\mathcal{TS}$ is a function assigning a thread state TS to every thread, and $\langle M, P\rangle$ is a global configuration. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state $\langle \sigma_i^0, \bot\rangle$, where σ_i^0 is the thread's initial local state and $\bot$ is the zero view (all timestamps in views are 0); the initial memory M^0 consisting of one message $\langle x :_0^{\texttt{rlx}} 0, \bot@0]\rangle$ for each location x; and the empty set of promises.

4 Semantics and soundness

In this section, we present the semantics of SLR, and give a short overview of the soundness proof. Our focus is not on the technical details of the proof, but on the two main challenges in defining the semantics and proving soundness:

1. *Reasoning about promises.* This difficulty arises because promise steps can be nondeterministically performed by the promise machine at any time.

2. *Reasoning about release-acquire ownership transfer in the presence of promises.* The problem is that writes may be promised before the thread has acquired enough resources to allow it to actually perform the write.

4.1 The intuition

SLR assertions are interpreted by (sets of) *resources*, which represent permissions to write to a certain location and/or to obtain further resources by reading a certain message from memory. As is common in semantics of separation logics, the resources form a partial commutative monoid, and SLR's separating conjunction is interpreted as the composition operation of the monoid.

When defining the meaning of a Hoare triple $\{P\}\ s\ \{Q\}$, we think of the promise machine as if it were manipulating resources: each thread owns some resources and operates using them. The intuitive description of the Hoare triple semantics is that every run of the program s starting from a state containing the resources described by the precondition, P, will be "correct" and, if it terminates, will finish in a state containing the resources described by the postcondition, Q. The notion of a program running correctly can be described in terms of threads "respecting" the resources they own; for example, if a thread is executing a write or fulfilling a promise, it should own a resource representing the write permission.

4.2 A closer look at the resources and the assertion semantics

We now take a closer look at the structure of resources and the semantics of assertions, whose formal definitions can be found in Figs. 2 and 3.

The idea is to interpret assertions as predicates over triples consisting of memory, a view, and a resource. We use the resource component to model assertions involving ownership (i.e., write assertions and acquire assertions), and model other assertions using the memory and view components. Once a resource is no longer needed, SLR allows us to drop these from assertions: $P * Q \Rightarrow P$. To model this we interpret assertions as upwards-closed predicates, that may own more than explicitly asserted. The ordering on memories and views is given by the promising semantics, and the ordering on resources is induced by the composition operation in the resource monoid. For now, we leave the resource composition unspecified, and return to it later.

In addition, however, we have to deal with assertions that are parametrised by predicates (in our case, $\mathsf{Rel}(x, \phi)$ and $\mathsf{Acq}(x, \phi)$). Doing so is not straightforward because naïve attempts of giving semantics to such assertions result in circular definitions. A common technique for avoiding this circularity is to treat predicates stored in assertions syntactically, and to interpret assertions relative to a *world*, which is used to interpret those syntactic predicates. In our case, worlds consist of two components: the *WrPerm* component associates a syntactic SLR predicate with every location (this component is used to interpret release permissions), while the *AcqPerm* component associates a syntactic predicate with a finite number of currently allocated predicate identifiers (this component is used to interpret acquire permissions). The reason for the more complex structure for acquire

$$\begin{aligned}
\iota \in \mathit{PredId} &\overset{def}{=} \mathbb{N} \quad \text{(predicate identifiers)}\\
\mathit{Perm} &\overset{def}{=} \{\pi \in \mathbb{Q} \mid 0 \leq \pi \leq 1\} \quad \text{(fractional permissions)}\\
\mathit{Write} &\overset{def}{=} \mathcal{P}(\mathit{Val} \times \mathit{Time})\\
\mathit{WrPerm} &\overset{def}{=} \mathit{Loc} \to \{(\pi, X) \in \mathit{Perm} \times \mathit{Write} \mid \pi = 0 \Rightarrow X = \emptyset\}\\
\mathit{AcqPerm} &\overset{def}{=} \mathit{Loc} \to \mathcal{P}(\mathit{PredId})\\
r = (r.\mathtt{wr}, r.\mathtt{acq}) \in \mathit{Res} &\overset{def}{=} \mathit{WrPerm} \times \mathit{AcqPerm} \quad \text{(resources)}\\
\mathcal{W} = (\mathcal{W}.\mathtt{rel}, \mathcal{W}.\mathtt{acq}) \in \mathit{World} &\overset{def}{=} (\mathit{Loc} \to \mathit{Pred}) \times (\mathit{PredId} \rightharpoonup_{\mathit{fin}} \mathit{Pred}) \quad \text{(worlds)}\\
\mathit{Prop} &\overset{def}{=} \mathit{World} \to_{\mathit{mon}} \mathcal{P}^{\uparrow}(\mathit{Mem} \times \mathit{View} \times \mathit{Res})
\end{aligned}$$

Fig. 2. Semantic domains used in this section.

permissions is that they can be split (see (Acquire-Split)). Therefore, we allow multiple predicate identifiers associated with a single location. When acquire permissions are divided and split between threads, new predicate identifiers are allocated and associated with predicates in the world. The world ordering, $\mathcal{W}_1 \leq \mathcal{W}_2$, expresses that world $\mathcal{W}_2$ is an extension of $\mathcal{W}_1$ in which new predicate identifiers may have been allocated, but all existing predicate identifiers are associated with the same predicates.

Let us now focus our attention on the assertion semantics. The semantics of assertions, $[\![P]\!]^{\eta}_{\mu}$, is relative to a thread store μ that assigns values to registers, and an environment η that assigns values to logical variables.

The standard logical connectives and quantifiers are interpreted following their usual intuitionistic semantics. The semantics of our novel assertions is given in Fig. 3 and can be explained as follows:

- The observed assertion $\mathsf{O}(x, v, t)$ says that the memory contains a message at location x with value v and timestamp t, and the current thread knows about it (i.e., the thread view contains it).
- The write assertion $\mathsf{W}^{\pi}(x, X)$ asserts ownership of a (partial, with fraction π) write resource at location x, and requires that the largest timestamp recorded in X does not exceed the view of the current thread.
- The acquire assertion, $\mathsf{Acq}(x, \phi)$, asserts that location x has some predicate identifier ι associated with the ϕ predicate in the current world $\mathcal{W}$.
- The release assertion, $\mathsf{Rel}(x, \phi)$, asserts that location x is associated with some predicate ϕ' in the current world such that there exists a syntactic proof of the entailment, $\vdash \forall v.\, \phi(v) \Rightarrow \phi'(v)$. The implication allows us to strengthen the predicate in release assertions.
- Finally, ∇P states that P is satisfiable in the current world.

Note that $\mathsf{W}^{\pi}(x, X)$, $\mathsf{Acq}(x, \phi)$, and $\mathsf{Rel}(x, \phi)$ only talk about owning certain resources, and do not constrain the memory itself at all. In the next subsection, we explain how we relate the abstract resources with the concrete machine state.

$$\begin{aligned}
[\![\mathsf{O}(x, v, t)]\!]^{\eta}_{\mu}(\mathcal{W}) &\stackrel{\mathit{def}}{=} \{(M, V, r) \mid \\
&\qquad \exists j, R, o.\, \langle [\![x]\!]^{\eta}_{\mu} :^{o}_{j} [\![v]\!]^{\eta}_{\mu}, R@[\![t]\!]^{\eta}_{\mu}] \rangle \in M \wedge [\![t]\!]^{\eta}_{\mu} \leq V(x)\} \\
[\![\mathsf{W}^{\pi}(x, X)]\!]^{\eta}_{\mu}(\mathcal{W}) &\stackrel{\mathit{def}}{=} \{(M, V, r) \mid \exists \pi' \geq [\![\pi]\!]^{\eta}_{\mu}.\ r.\mathtt{wr}([\![x]\!]^{\eta}_{\mu}) = (\pi', [\![X]\!]^{\eta}_{\mu}) \\
&\qquad\qquad \wedge\ \mathit{snd}(\max([\![X]\!]^{\eta}_{\mu})) \leq V([\![x]\!]^{\eta}_{\mu})\} \\
[\![\mathsf{Acq}(x, \phi)]\!]^{\eta}_{\mu}(\mathcal{W}) &\stackrel{\mathit{def}}{=} \{(M, V, r) \mid \exists \iota \in r.\mathtt{acq}([\![x]\!]^{\eta}_{\mu}).\, \mathcal{W}.\mathtt{acq}(\iota) = \phi\} \\
[\![\mathsf{Rel}(x, \phi)]\!]^{\eta}_{\mu}(\mathcal{W}) &\stackrel{\mathit{def}}{=} \{(M, V, r) \mid\ \vdash \forall v.\, \phi(v) \Rightarrow \mathcal{W}.\mathtt{rel}([\![x]\!]^{\eta}_{\mu})(v)\} \\
[\![\nabla P]\!]^{\eta}_{\mu}(\mathcal{W}) &\stackrel{\mathit{def}}{=} \{(M, V, r) \mid [\![P]\!]^{\eta}_{\mu}(\mathcal{W}) \neq \emptyset\}
\end{aligned}$$

Fig. 3. Interpretation of SLR assertions, $[\![\_]\!]^{\eta}_{\mu} \colon \mathit{Assn} \to \mathit{Prop}$

4.3 Relating concrete state and resources

Before giving a formal description of the relationship between abstract resources and concrete machine states, we return to the intuition of threads manipulating resources presented in Section 4.1.

Consider what happens when a thread executes a release write to a location x. At that point, the thread has to own a release resource represented by $\mathsf{Rel}(x, \phi)$, and to store the value v, it has to own the resources represented by $\phi(v)$. As the write is executed, the thread gives up the ownership of the resources corresponding to $\phi(v)$. Conversely, when a thread that owns the resource represented by $\mathsf{Acq}(x, \phi)$ performs an acquire read of a value v from location x, it will gain ownership of resources satisfying $\phi(v)$. However, this picture does not account for *what happens to the resources that are "in flight"*, i.e., the resources that have been released, but not yet acquired.

Our approach is to associate in-flight resources to messages in the memory. When a thread does a release write, it attaches the resources it released to the message it just added to the memory. That way, a thread performing an acquire read from that message can easily take ownership of the resources that are associated to the message. Formally, as the execution progresses, we update the assignment of resources to messages,

$$u \colon M(\mathtt{rel}) \to (\mathit{PredId} \to \mathit{Res})\,.$$

For every release message in memory M, the message resource assignment u gives us a mapping from predicate identifiers to resources. Here, we again use predicate identifiers to be able to track which acquire predicate is being satisfied by which resource. The intended reading of $u(m)(\iota) = r$ is that the resource r attached to the message m satisfies the predicate with the identifier ι.

We also require that the resources attached to a message (i.e., the resources released by the thread that wrote the message) suffice to satisfy all the acquire predicates associated with that particular location. Together, these two properties of our message resource assignment, as formalised in Fig. 4, allow us to describe the release/acquire ownership transfer.

The last condition in the message resource satisfaction relation has to do with relaxed accesses. Since relaxed accesses do not provide synchronisation, we disallow ownership transfer through them. Therefore, we require that the release

$$
\begin{aligned}
&M \models r, u, \mathcal{W} \overset{def}{=} \\
&\left.\begin{aligned}
&\quad \forall m \in M(\texttt{rel}).\ r.\texttt{acq}(m.\texttt{loc}) = \mathit{dom}(u(m)) \\
&\ \wedge \forall \iota \in \mathit{dom}(u(m)). \\
&\quad (M, m.\texttt{view}, u(m)(\iota)) \in [\![\mathcal{W}.\texttt{acq}(\iota)(m.\texttt{val})]\!]^{[]}_{[]}(\mathcal{W})
\end{aligned}\right\} \text{attached resources satisfy predicates they are supposed to} \\
&\left.\begin{aligned}
&\ \wedge \forall x, v.\ \vdash \mathcal{W}.\texttt{rel}(x)(v) \Rightarrow \circledast_{\iota \in r.\texttt{acq}(x)} \mathcal{W}.\texttt{acq}(\iota)(v) \\
&\ \wedge \forall m \in \mathit{dom}(u).\ \mathit{dom}(u(m)) \subseteq \mathit{dom}(\mathcal{W}.\texttt{acq})
\end{aligned}\right\} \text{released resources are enough to satisfy acquires} \\
&\left.\begin{aligned}
&\ \wedge \forall m \in M(\texttt{rlx}). \\
&\quad (\langle \emptyset, \emptyset \rangle, \lambda x.\, 0, \varepsilon) \in [\![\mathcal{W}.\texttt{rel}(m.\texttt{loc})(m.\texttt{val})]\!]^{[]}_{[]}(\mathcal{W}.\texttt{rel}, [])
\end{aligned}\right\} \text{no ownership transfer via relaxed accesses}
\end{aligned}
$$

Fig. 4. Message resource satisfaction.

predicates connected with the relaxed messages are satisfiable with the empty resource. This condition, together with the requirement that the released resources satisfy acquire predicates, forbids ownership transfer via relaxed accesses.

The resource missing from the discussion so far is the write resource (modelling the $\mathsf{W}^\pi(x, X)$ assertion). Intuitively, we would like to have the following property: whenever a thread adds a message to the memory, it has to own the corresponding write resource. Recall there are two ways a thread can produce a new message:

1. *A thread performs a write.* This is the straightforward case: we simply require the thread to own the write resource and to update the set of value-timestamp pairs recorded in the resource accordingly.
2. *A thread promises a write.* Here the situation is more subtle, because the thread might not own the write resource at the time it is issuing the promise, but will acquire the appropriate resource by the time it fulfils the promise. So, in order to assert that the promise step respects the resources owned by the thread, we also need to be able to talk about the resources that the thread *can acquire* in the future.

When dealing with the promises, the saving grace comes from the fact that all promises have to be certifiable, i.e., when issuing a promise a thread has to be able to fulfil it without help from other threads.

Intuitively, the existence of a certification run tells us that even though at the moment a thread issues a promise, it might not have the resources necessary to actually perform the corresponding write, the thread should, by running uninterrupted, still be able to obtain the needed resources before it fulfils the promise. This, in turn, tells us that the needed resources have to be already released by the other threads by the time the promise is made: only resources attached to messages in the memory are available to be acquired, and only the thread that made the promise is allowed to run during the certification; therefore all the available resources have already been released.

The above reasoning shows what it means for the promise steps to "respect resources": when promises are issued, the resources currently owned by a thread, together with all the resources it is able to acquire according to the resources it owns and the current assignment of resources to messages, have to contain the appropriate write resource for the write being promised. The notion of "resources a thread is able to acquire" is expressed through the $\mathrm{canAcq}(r, u)$ predicate.

$$r_1 \bullet r_2 \overset{def}{=} (r_1.\mathtt{wr} \bullet_{\mathtt{wr}} r_2.\mathtt{wr}, r_1.\mathtt{acq} \bullet_{\mathtt{acq}} r_2.\mathtt{acq}) \qquad \varepsilon \overset{def}{=} ([], \lambda\_.\,\emptyset)$$

$$f_1 \bullet_{\mathtt{wr}} f_2 \overset{def}{=} \begin{cases} \lambda x.\,(f_1(x).\mathtt{perm} + f_2(x).\mathtt{perm}, f_1(x).\mathtt{msgs} \cup f_2(x).\mathtt{msgs}) \\ \quad \text{if } f_1(x).\mathtt{perm} + f_2(x).\mathtt{perm} \leq 1 \text{ for all locations } x \\ \text{undefined otherwise} \end{cases}$$

$$g_1 \bullet_{\mathtt{acq}} g_2 \overset{def}{=} \text{if } \forall x.\, g_1(x) \cap g_2(x) = \emptyset \text{ then } \lambda x.\, g_1(x) \cup g_2(x) \text{ else undefined}$$

Fig. 5. Resource composition.

$$\begin{aligned} \lfloor r_F, u, \mathcal{W} \rfloor_T \overset{def}{=} \{\langle M, P\rangle \mid \mathbf{let}\ r = \textstyle\prod_{i \in TId} r_F(i) \bullet \prod_{m \in M} \prod_{\iota \in dom(u(m))} u(m)(\iota)\ \mathbf{in} \\ (1) \quad & M \models r, u, \mathcal{W} \wedge \\ (2) \quad & \forall x.\, \{(m.\mathtt{val}, m.\mathtt{time}) \mid m \in M(x) \setminus P\} = r.\mathtt{wr}(x).\mathtt{msgs} \wedge \\ (3) \quad & \forall m \in P.\ m.\mathtt{tid} \notin T \Rightarrow \\ & \quad (r_F(m.\mathtt{tid}) \bullet \mathrm{canAcq}(r_F(m.\mathtt{tid}), u)).\mathtt{wr}(m.\mathtt{loc}).\mathtt{perm} > 0\} \end{aligned}$$

r_F : *ThreadId* $\rightarrow$ *Res* maps threads to the resources they own.
r is the sum of all the resources distributed among the threads and messages.

Fig. 6. Erasure.

canAcq(r, u) performs a fixpoint calculation: the resources we have (r) allow us to acquire some more resources from the messages in memory (assignment of resources to messages is given by u), which allows us to acquire some more, and so on. Its formal definition can be found in the technical appendix, and hinges on the fact that u precisely tracks which resources satisfy which predicates.

An important element that was omitted from the discussion so far is the definition of the composition in the resource monoid *Res*. The resource composition, defined in Fig. 5, follows the expected notion of per-component composition. The most important feature is in the composition of write resources: a full permission write resource is only composable with the empty write resource.

At this point, we are equipped with all the necessary ingredients to relate abstract states represented by resources to concrete states $\langle M, P\rangle$ (where M is memory, and P is the set of promised messages). We define a function, called *erasure*, that given an assignment of resources to threads, r_F : *ThreadId* $\rightarrow$ *Res*, an assignment of resources to messages, u, and a world, $\mathcal{W}$, gives us a set of concrete states satisfying the following conditions:

1. Memory M is consistent with respect to the total resource r and the message resource assignment u at world $\mathcal{W}$.
2. The set of *fulfilled* writes to each location x in $\langle M, P\rangle$ must match the set of writes of all write permissions owned by any thread or associated with any messages, when combined.
3. For all unfulfilled promises to a location x by thread i, thread i must currently own or be able to acquire from u at least a shared write permission for x.

Our formal notion of erasure, defined in Fig. 6, has an additional parameter, a set of thread identifiers T. This set allows us to exclude promises of threads T from the requirement of respecting the resources. As we will see in the following subsection, this additional parameter plays a subtle, but key, role in the soundness proof. (The notion of erasure described above corresponds to the case when $T = \emptyset$.)

Note also that the arguments of erasure very precisely account for who owns which part of the total resource. This diverges from the usual approach in separation logic, where we just give the total resource as the argument to the erasure. Our approach is motivated by Lemma 1, which states that a reader that owns the full write resource for location x knows which value it is going to read from x. This is the key lemma in the soundness proof of the (R-RLX*) and (R-ACQ*) rules.

Lemma 1. If $(M, V, r_F(i)) \in [\![\mathsf{W}^1(x, X)]\!]^{\eta}_{\mu}(\mathcal{W})$, and $\langle M, P\rangle \in \lfloor r_F, u, \mathcal{W}\rfloor_{\{i\}}$ then for all messages $m \in M(x) \setminus P(i)$ such that $V(x) \leq m.\mathtt{time}$, we have $m.\mathtt{val} = \mathit{fst}(\max(X))$.

Lemma 1 is looking from the perspective of thread i that owns the full write resource for the location x. This is expressed by $(M, V, r_F(i)) \in [\![\mathsf{W}^1(x, X)]\!]^{\eta}_{\mu}(\mathcal{W})$ (recall that $r_F(i)$ are the resources owned by the thread i). Furthermore, the lemma assumes that the concrete state respects the abstract resources, expressed by $\langle M, P\rangle \in \lfloor r_F, u, \mathcal{W}\rfloor_{\{i\}}$. Under these assumptions, the lemma intuitively tells us that the current thread knows which value it will read from x. Formally, the lemma says that all the messages thread i is allowed to read (i.e., messages in the memory that are not outstanding promises of thread i and whose timestamp is greater or equal to the view of thread i) have the value that appears as the maximal element in the set X.

To see why this lemma holds, consider a message $m \in M(x) \setminus P(i)$. If m is an unfulfilled promise by a different thread j, then, by erasure, it follows that j currently owns or can acquire at least a shared write permission for x. However, this is a contradiction, since thread i currently owns the exclusive write permission, and, by erasure, $r_F(i)$ is disjoint from the resources of all other threads and all resources currently associated with messages by u. Hence, m must be a fulfilled write. By erasure, it follows that the set of fulfilled writes to x is given by the combination of all write permissions. Since $r_F(i)$ owns the exclusive write permission, this is just $r_F(i).\mathtt{wr}$. Hence, the set of fulfilled writes is X, and the value of the last fulfilled write is $\mathit{fst}(\max(X))$.

Note that in the reasoning above, it is crucial to know which thread and which message owns which resource. Without precisely tracking this information, we would be unable to prove Lemma 1.

4.4 Soundness

Now that we have our notion of erasure, we can proceed to formalise the meaning of triples, and present the key points of the soundness proof.

Recall our intuitive view of Hoare triples saying that the program only makes steps which respect the resources it owns. This notion is formalised using the *safety* predicate: safety (somewhat simplified; we give its formal definition in Fig. 7) states that it is always safe to perform zero steps, and performing $n + 1$ steps is safe if the following two conditions hold:

$$
\begin{aligned}
\mathrm{safe}_0(\sigma, B)(\mathcal{W}_1) &\overset{def}{=} \mathit{Mem} \times \mathit{View} \times \mathit{Res} \\
\mathrm{safe}_{n+1}(\sigma, B)(\mathcal{W}_1) &\overset{def}{=} \{(M_1, V_1, r_1) \mid \forall (M, V, r) \geq (M_1, V_1, r_1).\, \forall \mathcal{W} \geq \mathcal{W}_1. \\
&\qquad (\sigma.s = \textbf{skip} \Rightarrow (M, V, r) \in vs(B(\sigma.\mu))(\mathcal{W})) \\
&\qquad \wedge\ (\forall P, r_F, \sigma', M', P', V', u, i.\ \langle M, P\rangle \in \lfloor r_F[i \mapsto r], u, \mathcal{W}\rfloor_\emptyset \wedge \\
&\qquad\quad \langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \Longrightarrow_i \langle\langle \sigma', V'\rangle, \langle M', P'\rangle\rangle \\
&\qquad\quad \Rightarrow \exists r', u', \mathcal{W}' \geq \mathcal{W}.\, \langle M', P'\rangle \in \lfloor r_F[i \mapsto r'], u', \mathcal{W}'\rfloor_\emptyset \wedge \\
&\qquad\qquad (\langle M', P'\rangle, V', r') \in \mathrm{safe}_n(\sigma', B)(\mathcal{W}'))\}
\end{aligned}
$$

Fig. 7. Safety.

1. If no more steps can be taken, the current state and resources have to satisfy the postcondition B.
2. If we can take a step which takes us from the state $\langle M, P\rangle$ (which respects our current resources r, the assignment of resources to messages u, and world $\mathcal{W}$) to the state $\langle M', P'\rangle$, then
   (a) there exist resources r', an assignment of resources to messages u', and a future world $\mathcal{W}'$, such that $\langle M', P'\rangle$ respects r', u', and $\mathcal{W}'$, and
   (b) we are safe for n more steps starting in the state $\langle M', P'\rangle$ with resources given by r', u' and $\mathcal{W}'$.

Note the following:

– Upon termination, we are not required to satisfy exactly the postcondition B, but its *view shift*. A view shift is a standard notion in concurrent separation logics, which allows updates of the abstract resources which do not affect the concrete state. In our case, this means that resource r can be view-shifted into r' satisfying B as long as the erasure is unchanged. The formal definition of view shifts is given in the appendix.
– Again as is standard in separation logics, safety requires framed resources to be preserved. This is the role of r_F in the safety definition. Frame preservation allows us to compose safety of threads that own compatible resources. However, departing from the standard notion of frame preservation, we precisely track who owns which resource in the frame, because this is important for erasure.

The semantics of Hoare triples is simply defined in terms of the safety predicate. The triple $\{P\}\ s\ \{Q\}$ holds if every logical state satisfying the precondition is safe for any number of steps:

$$
[\![\vdash \{P\}\ s\ \{Q\}]\!] \overset{def}{=} \forall n, \mu, \eta, \mathcal{W}.\, [\![P]\!]^{\eta}_{\mu}(\mathcal{W}) \subseteq \mathrm{safe}_n((\mu, s), \lambda\mu'.\, [\![Q]\!]^{\eta}_{\mu'})(\mathcal{W})
$$

To establish soundness of the SLR proof rules, we have to prove that the safety predicate holds for arbitrary number of steps, including promise steps. The trouble with reasoning about promise steps is that they can nondeterministically appear at any point of the execution. Therefore, we have to account for them in the soundness proof of every rule of our logic. To make this task manageable, we encapsulate reasoning about the promise steps in a theorem, thus enabling the proofs of soundness for proof rules to consider only the non-promise steps.

To do so, once again certification runs for promises play a pivotal role. Recall that whenever a thread makes a step, it has to be able to fulfil its promises without help from other threads (Section 3.4). Since there will be no interference by other threads, performing promise steps during certification is of no use (because promises can only be used by other threads). Therefore, we can assume that the certification runs are always promise-free.

Now that we have noted that certifications are promise-free, the key idea behind encapsulating the reasoning about promises is as follows. If we know that all executions of our program are safe for arbitrarily many non-promising steps, we can use this to conclude that they are safe for promising steps too. Here, we use the fact that certification runs are possible runs of the program, and the fact that certifications are promise-free.

Let us now formalise our key idea. First, we need a way to state that executions are safe for non-promising steps. This is expressed by the *non-promising safety* predicate defined in Fig. 8. What we want to conclude is that non-promising safety is enough to establish safety, as expressed by Theorem 1:

Theorem 1 (Non-promising safety implies safety).

$$\forall n, \sigma, B, \mathcal{W}.\ \mathrm{npsafe}_{(n+1,0)}(\sigma, B)(\mathcal{W}) \subseteq \mathrm{safe}_n(\sigma, B)(\mathcal{W})$$

We now discuss several important points in the definition of non-promising safety which enable us to prove this theorem.

Non-promising safety is indexed by pairs of natural numbers. When proving Theorem 1, we use promise-free certification runs to establish the safety of the promise steps. A problem we face here is that the length of certification runs is unbounded. Somehow, we have to know that whenever the thread makes a step, it is npsafe for arbitrarily many steps. Our solution is to have npsafe transfinitely indexed over pairs of natural numbers ordered lexicographically. That way, if we are npsafe at index $(n+1, 0)$ and we take a step, we know that we are npsafe at index (n, m) for every m. We are then free to choose a sufficiently large m depending on the length of the certification run we are considering.

Non-promising safety considers configurations that may contain promises. It is important to note that the definition of non-promising safety does not require that there are no promises in the starting configuration. The only thing that is required is that no more promises are going to be issued. This is very important for Theorem 1, since safety considers all possible starting configurations (including the ones with existing promises), and if we want the lemma to hold, non-promising safety has to consider all possible starting configurations too.

Erasure used in the non-promising safety does not constrain promises of the current thread. Non-promising safety does not require promises by the thread being reduced (i.e., thread i) to respect resources. Thus, when reasoning about non-promising safety of thread i, we cannot assume that existing promises by thread i respect resources, but crucially we also do not have to worry about

$$
\begin{array}{l}
\mathrm{npsafe}_{(0,m)}(\sigma, B)(\mathcal{W}) \overset{def}{=} \mathit{Mem} \times \mathit{View} \times \mathit{Res} \\
\mathrm{npsafe}_{(n+1,0)}(\sigma, B)(\mathcal{W}) \overset{def}{=} \bigcap_{m \in \mathbb{N}} \mathrm{npsafe}_{(n,m)}(\sigma, B)(\mathcal{W}) \\
\mathrm{npsafe}_{(n+1,m+1)}(\sigma, B)(\mathcal{W}_1) \overset{def}{=} \{(M_1, V_1, r_1) \mid \forall (M, V, r) \geq (M_1, V_1, r_1).\, \forall \mathcal{W} \geq \mathcal{W}_1. \\
\quad (\sigma.s = \textsf{skip} \Rightarrow (M, V, r) \in vs(B(\sigma.\mu))(\mathcal{W})) \\
\quad \wedge (\forall P, r_F, f, \sigma', M', P', V', u, i. \\
\qquad \langle M, P\rangle \in \lfloor r_F[i \mapsto r \bullet f], u, \mathcal{W} \rfloor_{\{i\}} \quad \text{(weak erasure)} \\
\qquad \wedge \quad \langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \xrightarrow{\mathrm{NP}}_i \langle\langle \sigma', V'\rangle, \langle M', P'\rangle\rangle \quad \text{(only non-promising steps allowed)} \\
\qquad \wedge\, \mathrm{wf_{prom}}(P(i), V) \wedge \mathrm{wf_{prom}}(P'(i), V') \quad \text{(promises well formed)} \\
\qquad \Rightarrow \exists r', u', \mathcal{W}' \geq \mathcal{W}.\, M' \in \lfloor r_F[i \mapsto r' \bullet f], u', \mathcal{W}' \rfloor_{\{i\}} \quad \text{(weak erasure)} \\
\qquad\quad \wedge (M', V', r') \in \mathrm{npsafe}_{(n+1,m)}(\sigma', B)(\mathcal{W}') \\
\qquad\quad \wedge\, r' \bullet \mathrm{canAcq}(r', u') \leq_{\mathrm{o}} r \bullet \mathrm{canAcq}(r, u) \quad \text{(no new res. acquirable after taking a step)} \\
\qquad\quad \wedge\, \forall m \in (M' \setminus P') \setminus (M \setminus P).\, r.\mathtt{wr}(m.\mathtt{loc}).\mathtt{perm} > 0\} \quad \begin{array}{l}\text{(when performing a write} \\ \text{or fulfiling a promise} \\ \text{the thread has to own} \\ \text{the appropriate write res.)}\end{array}
\end{array}
$$

$$r_1 \leq_{\mathrm{o}} r_2 \overset{def}{=} \forall x.\ r_1.\mathtt{wr}(x).\mathtt{perm} \leq r_2.\mathtt{wr}(x).\mathtt{perm}$$

$$\mathrm{wf_{prom}}(P, V) \overset{def}{=} \forall m \in P.\ V(m.\mathtt{loc}) < m.\mathtt{time}$$

Fig. 8. Non-promising safety.

recertifying thread i's promises. However, since the $\xrightarrow{\mathrm{NP}}$ reduction does not recertify promises, we explicitly require that the promises are well formed (via $\mathrm{wf_{prom}}$ predicate) in order to ensure that we still only consider executions where threads do not read from their own promises.

Additional constraints by the non-promising safety. Non-promising safety also imposes additional constraints on the reducing thread i. In particular, any write permissions owned or acquirable by i after the reduction were already owned or acquirable by i before the reduction step. Intuitively, this holds because thread i can only transfer away resources and take ownership of resources it was already allowed to acquire before reducing. Lastly, non-promising safety requires that if the reduction of i performs any new writes or fulfils any old promises, it must own the write permission for the location of the given message. Together, these two conditions ensure that if a promise is fulfilled during a thread-local certification and the thread satisfies non-promising safety, then the thread already owned or could acquire the write permission for the location of the promise. This is expressed formally in Lemma 2.

Lemma 2. Assuming that $(\langle M, P\rangle, V, r) \in \mathrm{npsafe}_{(n+1,k)}(\sigma, B)(W)$ and $\langle M, P\rangle \in \lfloor r_F[i \mapsto r \bullet f], u, W \rfloor_{\{i\}}$ and $\langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \xrightarrow{\mathrm{NP}}{}_i^{k} \langle\langle \sigma', V'\rangle, \langle M', P'\rangle\rangle$ and $m \in (M' \setminus P') \setminus (M \setminus P)$, we have $(r \bullet \mathrm{canAcq}(r, u)).\mathtt{wr}(m.\mathtt{loc}).\mathtt{perm} > 0$.

The intuition for why Lemma 2 holds is that since only thread i executes, we know by the definition of non-promising safety that any write permission owned or acquirable by i when the promise is fulfilled, it already owns or can acquire in the initial state. Furthermore, whenever a promise is fulfilled, the non-promising safety definition explicitly requires ownership of the corresponding write permission. It follows that the thread already owns or can acquire the write permission for the location of the given promise in the initial state.

Lemma 2 gives us exactly the property that we need to reestablish erasure after the operational semantics introduces a new promise. This makes Lemma 2 the key step in the proof of Theorem 1, which allows us to disentangle reasoning about promising steps and normal reduction steps. Theorem 1 tells us that, in order to prove a proof rule sound, it is enough to prove that the non-promising safety holds for arbitrary indices. This liberates us of the cumbersome reasoning about promise steps and allows us to focus on non-promising reduction steps when proving the proof rules sound.

We can now state our top-level correctness theorem, Theorem 2. Since our language only has top-level parallel composition, we need a way to distribute initial resources to the various threads, and to collect all the resources once all the threads have finished. The correctness theorem gives us precisely that:

Theorem 2 (Correctness). If A is a finite set of locations and
1. $\vdash \forall x \in A.\, \phi_x(0)$
2. $\vdash \circledast_{x \in A} \mathsf{Rel}(x, \phi_x) * \mathsf{Acq}(x, \phi_x) * \mathsf{W}^1(x, \{(0,0)\}) \Rrightarrow \circledast_{i \in \mathit{Tid}} P_i$
3. $\vdash \{P_i\}\ s_i\ \{Q_i\}$ for all i
4. $\langle \lambda i.\, \langle(\mu_i, s_i), \bot\rangle, \langle M^0, \emptyset\rangle\rangle \Longrightarrow^* \langle \mathcal{TS}, \mathit{gconf}\rangle$ and $\mathcal{TS}(i).\sigma = \textsf{skip}$ for all i
5. $\vdash \circledast_{i \in \mathit{Tid}} Q_i \Rrightarrow Q$
6. $\mathit{FRV}(Q_i) \cap \mathit{FRV}(Q_j) = \emptyset$ for all distinct $i, j \in \mathit{Tid}$

then there exist μ, r, and $\mathcal{W}$ such that $(\mathit{gconf}.M, \sqcup_i \mathcal{TS}(i).V, r) \in [\![Q]\!]^{[]}_{\mu}(\mathcal{W})$ and $\forall i \in \mathit{Tid}.\, \forall a \in \mathit{FRV}(Q_i).\, \mu(a) = \mathcal{TS}(i).\mu(a)$, where $\mathit{FRV}(P)$ denotes the set of free register variables in P.

5 Related work

There are a number of techniques for reasoning under relaxed memory models, but besides the DRF theorems and some simple invariant logics [10, 13], no other techniques have been proved sound for a model allowing the weak behaviour of LB+data+fakedep from the introduction. The "invariant-based program logics" are by design unable to reason about programs like the random number generator, where having a bound on the set of values written to a location is not enough, let alone reasoning about functional correctness of a program.

Relaxed separation logic (RSL). Among program logics for relaxed memory, the most closely related is RSL [27]. There are two versions of RSL: a weak one that is sound with respect to the C/C++11 memory model, which features out-of-thin-air reads, and a stronger one that is sound with respect to a variant of the C/C++11 memory that forbids load buffering.

The weak version of RSL forbids relaxed writes completely, and does not constrain the value returned by a relaxed read. The stronger version provides single-location invariants for relaxed accesses, but its soundness proof relies strongly on a strengthened version of C/C++11 without $po \cup rf$ cycles (where po is program order, and rf is the reads-from relation), which forbids load buffering.

When it comes to reasoning about coherence properties, even the strong version of RSL is surprisingly weak: it cannot be used to verify any of the coherence examples in this paper. In fact, RSL can be shown sound with respect to much weaker coherence axioms than what C/C++11 relaxed accesses provide.

One notable feature of RSL which we do not support is read-modify-write (RMW) instructions (such as compare-and-swap and fetch-and-add). However, the soundness proof of SLR makes no simplifying assumptions about the promising semantics which would affect the semantics of RMW instructions. Therefore, we are confident that enhancing SLR with rules for RMW instructions would not substantially affect the structure of the soundness proof, presented in Section 4.

Other program logics. FSL [8] extends (the strong version of) RSL with stronger rules for relaxed accesses in the presence of release/acquire fences. In FSL, a release fence can be used to package an assertion with a modality, which a relaxed write can then transfer. Conversely, the ownership obtained by a relaxed read is guarded by a symmetric modality than needs an acquire fence to be unpacked. The soundness proof of FSL also relies on $po \cup rf$ acyclicity. Moreover, it is known to be unsound in models where load buffering is allowed [9, §5.2].

A number of other logics—GPS [26], iGPS [12], OGRA [16], iCAP-TSO [24], the rely-guarantee proof system for TSO of Ridge [23], and the program logic for TSO of Wehrman and Berdine [28]—have been developed for even stronger memory models (release/acquire or TSO), and also rely quite strongly on—and try to expose—the stronger consistency guarantees provided by those models.

The framework of Alglave and Cousot [2] for reasoning about relaxed concurrent programs is parametric with respect to an axiomatic "per-execution" memory model. By construction, as argued by Batty et al. [3], such models cannot be used to define a language-level model allowing the weak behaviour of LB+data+fakedep and similar litmus tests while forbidding out-of-thin-air behaviours. Moreover, their framework does not provide the usual abstraction facilities of program logics.

The lace logic of Bornat et al. [6] targets hardware memory models, in particular Power. It relies on annotating the program with "per-execution" constraints, and on syntactic features of the program. For example, it distinguishes LB+data+fakedep from LB+data+po, its variant where the write of second thread is $[x]_{\texttt{rlx}} := 1$, and is thus unsuitable to address out-of-thin-air behaviours.

Other approaches. Besides program logics, another way to reason about programs under weak memory models is to reduce the act of reasoning under a memory model M to reasoning under a stronger model M'—typically, but not necessarily, sequential consistency [7, 18]. One can often establish DRF theorems stating that a program without any races when executed under M' has the same behaviours when executed under M as when executed under M'. For the promising semantics, Kang et al. [13, §5.4] have established such theorems for M' being release-acquire consistency, sequential consistency, and the promise-free promising semantics, for suitable notions of races. The last one, the "Promise-Free DRF" theorem, is applicable to the Disjoint-Lists program from the introduction, but none of these theorems can be applied to any of the other examples of this paper, as they are racy. Moreover, these theorems are not compositional, as they do not state anything about the Disjoint-Lists program when put inside a larger, racy program—for example, just an extra read of a from another thread.

6 Conclusion

In this paper, we have presented the first expressive logic that is sound under the promising semantics, and have demonstrated its expressiveness with a number of examples. Our logic can be seen both as a general proof technique for reasoning about concurrent programs, and also as tool for proving the absence of out-of-thin-air behaviour for challenging examples, and reasoning about coherence. In the future, we would like to extend the logic to cover more of relaxed memory, more advanced reasoning principles, such as those available in GPS [26], and mechanise its soundness proof.

Interesting aspects of relaxed memory we would like to also cover are read-modify-writes and fences. These would allow us to consider concurrent algorithms like circular buffers and the atomic reference counter verified in FSL++ [9]. This could be done by adapting the corresponding rules of RSL and GPS; moreover, we could adapt them with our new approach to reason about coherence.

To mechanise the soundness proof, we intend to use the Iris framework [11], which has already been used to prove the soundness of iGPS [12], a variant of the GPS program logic. To do this, however, we have to overcome one technical limitation of Iris. Namely, the current version of Iris is step-indexed over $\mathbb{N}$, while our semantics uses transfinite step-indexing over $\mathbb{N} \times \mathbb{N}$ to define non-promising safety and allow us to reason about certifications of arbitrary length for each reduction step. Progress has been made towards transfinitely step-indexed logical relations that may be applicable to a transfinitely step-indexed version of Iris [25].

Acknowledgments. We would like to thank the reviewers for their feedback. The research was supported in part by the Danish Council for Independent Research (project DFF – 4181-00273), by a European Research Council Consolidator Grant for the project "RustBelt" (grant agreement no. 683289), and by Len Blavatnik and the Blavatnik Family foundation.

References

1. Supplementary material for this paper, available at `http://plv.mpi-sws.org/slr/appendix.pdf`

2. Alglave, J., Cousot, P.: Ogre and Pythia: an invariance proof method for weak consistency models. In: POPL 2017. pp. 3–18. ACM, New York (2017), `http://doi.acm.org/10.1145/2994593`
3. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer, Heidelberg (2015)
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011. pp. 55–66. ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1926385.1926394`
5. Boehm, H.J., Demsky, B.: Outlawing ghosts: Avoiding out-of-thin-air results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness. pp. 7:1–7:6. MSPC '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2618128.2618134`
6. Bornat, R., Alglave, J., Parkinson, M.: New lace and arsenic (2016), `https://arxiv.org/abs/1512.01416`
7. Bouajjani, A., Derevenetc, E., Meyer, R.: Robustness against relaxed memory models. In: Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Deutschland. pp. 85–86 (2014)
8. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 413–430. Springer, Heidelberg (2016)
9. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: ESOP 2017. LNCS, vol. 10201, pp. 448–475. Springer, Heidelberg (2017)
10. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: LICS 2016. pp. 759–767. ACM, New York (2016)
11. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up (2017)
12. Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: ECOOP 2017 (2017)
13. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017. ACM, New York (2017), `http://doi.acm.org/10.1145/3009837.3009850`
14. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI (2017)
15. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL 2016. pp. 649–662. ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2837614.2837643`
16. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Automata, Languages, and Programming, ICALP'15, LNCS, vol. 9135, pp. 311–323. Springer, Berlin, Heidelberg (2015)
17. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL. pp. 378–391. ACM, New York (2005)
18. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: ECOOP 2010: 24th European Conference on Object-Oriented Programming. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
19. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics. pp. 391–407. TPHOLs '09, Springer-Verlag, Berlin, Heidelberg (2009)

20. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: POPL 2016. pp. 622–633. ACM, New York (2016)
21. Podkopaev, A., Lahav, O., Vafeiadis, V.: Promising compilation to armv8 POP. In: ECOOP 2017. LIPIcs, vol. 74, pp. 22:1–22:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74 (2002), https://doi.org/10.1109/LICS.2002.1029817
23. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer (2010)
24. Sieczkowski, F., Svendsen, K., Birkedal, L., Pichon-Pharabod, J.: A separation logic for fictional sequential consistency. In: ESOP 2015. LNCS, vol. 9032, pp. 736–761. Springer (2015)
25. Svendsen, K., Sieczkowski, F., Birkedal, L.: Transfinite step-indexing: Decoupling concrete and logical steps. In: Proceedings of ESOP 2016. pp. 727–751 (2016)
26. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: Navigating weak memory with ghosts, protocols, and separation. In: 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 691–707. OOPSLA '14, ACM, New York (2014)
27. Vafeiadis, V., Narayan, C.: Relaxed separation logic: A program logic for C11 concurrency. In: OOPSLA 2013. pp. 867–884. ACM, New York (2013)
28. Wehrman, I., Berdine, J.: A proposal for weak-memory local reasoning. In: LOLA (2011)