RGSep under Release/Acquire Consistency

ELLEN ARLT✉, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

RGSep is a program logic for reasoning about the correctness of concurrent programs that combines rely-guarantee reasoning and separation logic. Although RGSep was initially developed for sequential consistency, we show that it is also sound under the much weaker release-acquire (RA) consistency model, which is a well-behaved subset of the C++11 concurrency model. Our result provides a simpler way to reason about RA programs than the state-of-the-art program logics that support weak memory consistency models.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Programming logic**; **Concurrency**.

Additional Key Words and Phrases: Rely-Guarantee, Weak Memory Consistency

ACM Reference Format:
Ellen Arlt and Viktor Vafeiadis. 2026. RGSep under Release/Acquire Consistency. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 375 (October 2026), 27 pages. https://doi.org/10.1145/3839507

1 Introduction

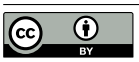
Since the introduction of *concurrent separation logic* (CSL) [26], there has been a lot of interest in verifying the correctness of concurrent programs with dynamic memory allocation and pointer data structures. It was quickly recognized that CSL is very good at reasoning about coarse-grained concurrent programs, where concurrent data structures are guarded by a global lock, but rather inconvenient for reasoning about fine-grained concurrency.

As a result, many more advanced program logics were proposed. Early work in this area assumed a *sequentially consistent* (SC) [24] context, where threads are executed in an interleaving fashion. An important first milestone in this line of work was the RGSep logic [37], which was used to establish the correctness of fine-grained concurrent list algorithms [33]. RGSep addressed an inherent limitation of CSL by combining it with rely-guarantee reasoning [14], which enables reasoning about interference between threads. Subsequent logics, such as deny-guarantee [9], CAP [8], LRG [11], TaDA [5], and Iris [15], introduced more modular forms of reasoning with advanced concepts, such as logical atomicity and view shifts.

Gradually, researchers started taking into account *weak memory consistency* (WMC) models, such as the TSO memory model employed by the Sparc and x86 architectures [27, 30], and the weaker *release-acquire* (RA) memory model [18, 20], which is a well-behaved fragment of the C++11 concurrency model [1, 23]. Reasoning about program correctness under WMC is much more challenging because threads can disagree about the order in which shared operations took place. Consider the following 4-thread program with two shared variables x and y, initially holding 0.

$$[x] := 1 \;\Big\|\; \begin{array}{l} a := [x]; \\ b := [y] \end{array} \;\Big\|\; \begin{array}{l} c := [y]; \\ d := [x] \end{array} \;\Big\|\; [y] := 1 \qquad \text{(IRIW)}$$

Authors' Contact Information: Ellen Arlt, MPI-SWS, Kaiserslautern, Germany, earlt@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Kaiserslautern and Saarland Infomatics Campus (SIC), Germany, viktor@mpi-sws.org.



ACM 2475-1421/2026/10-ART375
https://doi.org/10.1145/3839507

Under RA, the program can return $a = c = 1 \land b = d = 0$: i.e., the middle threads may observe the independent writes to x and y as happening in different orders.

To reason about program correctness in spite of such behaviors, several program logics for weak memory models were developed. Initial logics, such as RSL [36] and FSL [10], simply adopted CSL-style reasoning principles and thus were not powerful enough to reason about fine-grained concurrency. Subsequent logics, such as GPS [16, 32] and iRC11 [7], introduced more advanced reasoning principles, but diverged significantly from the logics for SC programs. They required invariants about the shared memory to either mention only one shared memory location or to be expressed in terms of low-level concepts, such as timestamps and message views (e.g., [6]).

Given a fine-grained concurrent program that has already been proven correct under SC, if we want to verify it under WMC, with the existing logics, we have to come up with an entirely new proof that conforms to the rules of these program logics. Even though there has been quite some success in verifying concurrent programs with the aforementioned logics under WMC, the fact that verification with these logics requires a very different proof from the one with an SC logic means that the existing correctness proof under SC is useless to us in that regard. But what if, on the contrary, an SC proof could be all we need to prove correctness of a program under WMC? This would drastically decrease the proof effort, since the reasoning principles in the SC logics are much simpler to use, making it much easier to write proofs for SC than WMC.

This is what we set out to do in this paper. Our aim is to simplify reasoning about WMC programs by using *the same* reasoning principles as we have been using for SC programs. Specifically, we consider the RGSep logic [37], which offers a sweet-spot between simplicity and expressivity. It is substantially simpler than its successors, and yet strong enough to verify fine-grained concurrent programs, as demonstrated in prior work [33].

In this paper, we prove that RGSep is, in fact, sound under RA [20] and that it is unsound under two slightly weaker memory models:

- *weak release acquire* (WRA) [19], also known as *causal consistency* [3], and
- *strong coherence* (SCOH), i.e., the "relaxed" fragment of RC11 [23].

While establishing unsoundness under WRA and SCOH is fairly easy (see §3), showing soundness under RA is much more challenging because we have to interpret the semantics of RGSep judgments in a setting where threads may disagree about the order in which independent shared operations took place. To do so, we design a novel operational semantics for RA that keeps track of the set of operations visible to each thread (§4) and interpret RGSep judgments from the perspective of the current thread (§5). For the latter, we introduce the concept of a logical configuration, which records the logical splitting of the global state into shared and local components, and define how each operation affects this splitting. These tools enabled us to prove the soundness of RGSep under RA (§6).

Our soundness proof has two main limitations. It does not support ownership transfer from shared to local, nor atomic blocks beyond RMW instructions. We discuss these limitations in §8.

Rocq Mechanization. All the definitions and proofs mentioned in this paper have been mechanized in the Rocq proof assistant and are available at https://zenodo.org/records/21710134.

2 RGSep: Combining Rely-Guarantee and Separation Logic

In this section, we introduce the RGSep program logic [37]. For concreteness, we first define a simple programming language (§2.1). We then discuss RGSep assertions (§2.2), rely-guarantee conditions (§2.3), and the proof rules for verifying programs (§2.4).

2.1 A Simple Concurrent Programming Language

We assume domains of memory locations Addr $\triangleq \mathbb{Z}^+$ and of values Val $\triangleq \mathbb{Z}$, and let l range over locations and v over values. The syntax of our programming language is as follows:

$$e ::= v \mid x \mid e_1 + e_2 \mid e_1 \geq e_2 \mid \ldots$$
$$C ::= e \mid \textbf{assert}(e) \mid \textbf{alloc} \mid [e] \mid [e_1] := e_2 \mid \mathsf{cas}(e_1, e_2, e_3) \mid \textbf{let}\ x = C_1\ \textbf{in}\ C_2 \mid C_1 \parallel C_2 \mid$$
$$\textbf{if}\ e\ \textbf{then}\ C_1\ \textbf{else}\ C_2 \mid \textbf{repeat}\ C\,.$$

Expressions, e, do not have any side-effects: they comprise values, variables, and arithmetic and logical operations.

In contrast, program computations, C, can access memory: they comprise simple expressions, assertions, memory allocation, memory reads, memory writes, compare-and-swap operations, sequential composition (the **let**-binding construct), parallel composition (that returns the value of the left thread), conditionals, and a looping construct repeating a computation while it returns 0. The SC semantics of these constructs is standard: we can define an interleaving-based small-step operational semantics, which we omit for brevity.

2.2 RGSep Assertions

RGSep logically divides the memory seen by a thread into two regions. One region is *local* to the thread in the sense that other threads cannot simultaneously access it. The other region is *shared* among threads, and may be simultaneously updated by other threads.

To describe these two regions, RGSep uses two levels of assertions.

On the lower level, we have standard *separation logic* (SL) assertions, A, which describe a single region: a partial function from locations to values, often called the *heap*.

$$A ::= e \mid \mathsf{emp} \mid e_1 \mapsto e_2 \mid A_1 * A_2 \mid A_1 \circledast\!\!- A_2 \mid$$
$$A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \Rightarrow A_2 \mid \exists x.\, A \mid \forall x.\, A \qquad \text{(SL assertions)}$$

Separation logic (SL) assertions, A, contain boolean expressions, the empty heap assertion, points-to predicates, and the standard connectives, which are interpreted classically. The points-to predicate, $l \mapsto v$, asserts that the heap contains exactly one location l with the value v. The *separating conjunction* $A_1 * A_2$ holds of a heap h if it can be split into two (disjoint) parts, one satisfying A_1 and the other satisfying A_2. *Septraction* $A_1 \circledast\!\!- A_2$ holds of a heap h if it can be extended with a disjoint part h' satisfying A_2 such that the total heap $h \uplus h'$ satisfies A_1.

Formally,

$$h \models A_1 * A_2 \text{ iff } \exists h_1, h_2 \text{ such that } h_1 \uplus h_2 = h \text{ and } h_1 \models A_1 \text{ and } h_2 \models A_2$$
$$h \models A_1 \circledast\!\!- A_2 \text{ iff } \exists h' \text{ such that } h \uplus h' \models A_1 \text{ and } h' \models A_2$$

On the upper level, we have RGSep assertions, P, Q, which describe both regions, and are modeled by pairs of mutually disjoint heaps $\langle h_L, h_S \rangle$, one representing the local heap and the other the shared heap. The syntax of these assertions is given below:

$$P, Q ::= A \mid \boxed{A} \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \exists x.\, P \mid \forall x.\, P \qquad \text{(RGSep assertions)}$$

RGSep assertions contain SL assertions about the local state, A, and about the shared state, $\boxed{A}$, and standard connectives between them. The local state assertion merely says that the local state satisfies the assertion (and the shared state can be arbitrary). In contrast, the shared state assertion, $\boxed{A}$, requires the local state to be empty, and requires the shared state to contain a portion that satisfies A (i.e., we interpret shared assertions intuitionistically). RGSep's separating conjunction acts as a separating conjunction only on the local state while acting as a regular conjunction on

the shared state: $P * Q$ holds of the heap pair $\langle h_L, h_S \rangle$ if the local part, h_L, can be split into two disjoint parts, h_1 and h_2, such that P holds in one part (h_1 along with the shared part, h_S) and Q in the other part (h_2, again, along with the same shared part h_S). The reason for this definition will become evident when we present RGSep's parallel composition rule in §2.4.

$$h_L, h_S \models A \text{ iff } h_L \models A$$
$$h_L, h_S \models \boxed{A} \text{ iff } h_L = \emptyset \text{ and } h_S \models A * \text{true}$$
$$h_L, h_S \models P * Q \text{ iff } \exists h_1, h_2 \text{ such that } h_1 \uplus h_2 = h_L \text{ and } h_1, h_S \models P \text{ and } h_2, h_S \models Q$$

The interpretations of the remaining connectives are standard.

2.3 RGSep Actions and Stability

RGSep uses two logical objects to capture the interference from the perspective of a given thread: the *rely* and the *guarantee*. The rely describes how the *environment* (the collection of all threads executing concurrently with a given thread) may interfere with the given thread by modifying the shared state. The guarantee, on the other hand, describes how the given thread may interfere with its environment.

In RGSep, we specify how the shared state may evolve via *actions*. An action, $A_1 \rightsquigarrow A_2 \mid A_F$, is a tuple of three SL assertions, describing that the portion of the shared state satisfying assertion A_1 can change into a portion satisfying A_2 when the remaining shared state satisfies $A_F * \text{true}$. As a shorthand notation, we write $A_1 \rightsquigarrow A_2$ for actions where $A_F = \text{emp}$. For example, the action $\exists v.\, l \mapsto v \rightsquigarrow l \mapsto 5$ captures the effect of a statement writing 5 to a shared location l.

Actions may also signify ownership transfer, conversion of local memory regions into shared regions, when the assertions A_1 and A_2 describe different numbers of memory locations. For example, $a \mapsto b \rightsquigarrow \exists l.\, a \mapsto l * l \mapsto b$ represents the atomic insertion of a node l into a linked list, which can be the result of an atomic write instruction changing a's next node to l. Here, ownership of node l is transferred.

Ultimately, we define the rely and guarantee as sets of actions that comprise the atomic changes of the shared state. The guarantee consists of actions due to the computation being verified and the rely consists of actions due to its environment.

An important concept in rely-guarantee reasoning in general, and in RGSep in particular, is that of *stability*. We say that a separation logic assertion A about the shared state is *stable* under an action if A continues to hold after the shared state is updated according to the action. Formally,

$$\text{stable}(A, A_1 \rightsquigarrow A_2 \mid A_F) \quad \text{holds iff} \quad ((A \circledast\!\!- A_1) \wedge (A_F * \text{true})) * A_2 \implies A\,.$$

An SL assertion A is stable under a set of actions (a rely) R, written $\text{stable}(A, R)$, if it is stable under every action in R.

We extend the notion of stability to RGSep assertions. An RGSep assertion P is stable under a rely R (P *s.u.* R) if all SL assertions appearing syntactically in boxes inside P are stable under R.

$$A \text{ s.u. } R \iff \text{always} \qquad P\ [*|\wedge|\vee]\ Q \text{ s.u. } R \iff P \text{ s.u. } R \text{ and } Q \text{ s.u. } R$$
$$\boxed{A} \text{ s.u. } R \iff \text{stable}(A * \text{true}, R) \qquad [\exists|\forall] x.\, P \text{ s.u. } R \iff P \text{ s.u. } R \text{ for all } x$$

The "$* \text{true}$" in the stability definition above is because we treat shared assertions intuitionistically: $\boxed{A}$ asserts that some part of the shared state satisfies A.

$$\frac{R, G \vdash \{P\}\, C\, \{x.\ Q\} \qquad R' \subseteq R \quad G \subseteq G' \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{R', G' \vdash \{P'\}\, C\, \{x.\ Q'\}}\ \text{Conseq}$$

$$\frac{R, G \vdash \{P\}\, C\, \{x.\ Q\} \qquad F \text{ s.u. } R \cup G \qquad x \notin \mathrm{fv}(F)}{R, G \vdash \{P * F\}\, C\, \{x.\ Q * F\}}\ \text{Frame}$$

$$\frac{P \text{ s.u. } R \qquad x \notin \mathrm{fv}(P)}{R, G \vdash \{P\}\, v\, \{x.\ x = v \wedge P\}}\ \text{Val} \qquad \frac{P \text{ s.u. } R \qquad P \Rightarrow e}{R, G \vdash \{P\}\, \mathbf{assert}(e)\, \{P\}}\ \text{Assn} \qquad \frac{\vdash_{\mathsf{SL}} \{A\}\, C\, \{x.\ A'\}}{R, G \vdash \{A\}\, C\, \{x.\ A'\}}\ \text{Local}$$

$$\frac{R, G \vdash \{P\}\, C_1\, \{x.\ Q\} \qquad R, G \vdash \{Q\}\, C_2\, \{y.\ Q'\} \qquad x \notin \mathrm{fv}(Q')}{R, G \vdash \{P\}\, \mathbf{let}\ x = C_1\ \mathbf{in}\ C_2\, \{y.\ Q'\}}\ \text{Let}$$

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\, C_1\, \{x.\ Q_1\} \qquad R \cup G_1, G_2 \vdash \{P_2\}\, C_2\, \{\_.\ Q_2\} \qquad x \notin \mathrm{fv}(Q_2)}{R, G_1 \cup G_2 \vdash \{P_1 * P_2\}\, C_1 \parallel C_2\, \{x.\ Q_1 * Q_2\}}\ \text{Par}$$

$$\frac{\begin{array}{l} P, Q \text{ s.u. } R \\ P \Rightarrow \boxed{\exists v.\, l \mapsto v * A} \\ x = v \wedge P \wedge \boxed{l \mapsto v * A} \Rightarrow Q \end{array}}{R, G \vdash \{P\}\, [l]\, \{x.\ Q\}}\ \text{AtRead} \qquad \frac{\begin{array}{l} P, Q \text{ s.u. } R \\ P \Rightarrow A_{\mathrm{L2S}} * \boxed{\exists v.\, l \mapsto v * A} \\ \boxed{l \mapsto v' * A * A_{\mathrm{L2S}}} \Rightarrow Q \\ \forall v.\, (l \mapsto v \rightsquigarrow l \mapsto v' * A_{\mathrm{L2S}} \mid A) \subseteq G \end{array}}{R, G \vdash \{P\}\, [l] := v'\, \{Q\}}\ \text{AtWrite}$$

$$\frac{\begin{array}{l} P, Q \text{ s.u. } R \\ P \Rightarrow A_{\mathrm{L2S}} * \boxed{\exists w.\, l \mapsto w * A} \\ x = 0 \wedge P * \boxed{\exists w.\, w \neq v \wedge l \mapsto w * A} \Rightarrow Q \\ x = 1 \wedge \boxed{l \mapsto v' * A * A_{\mathrm{L2S}}} \Rightarrow Q \\ (l \mapsto v \rightsquigarrow l \mapsto v' * A_{\mathrm{L2S}} \mid A) \subseteq G \end{array}}{R, G \vdash \{P\}\, \mathsf{cas}(l, v, v')\, \{x.\ Q\}}\ \text{CAS}$$

Fig. 1. A selection of RGSep proof rules for verifying concurrent programs.

2.4 RGSep Judgments and Proof Rules

RGSep program specifications consist of four components: two RGSep assertions, the precondition P and the postcondition Q; and two sets of actions, the rely R and the guarantee G. The main judgment of the logic is written as $R, G \vdash \{P\}\, C\, \{x.\ Q\}$, where the variable x binds the result of the computation C in the postcondition Q. When the postcondition does not mention the result of the computation, we often drop the binder and write $R, G \vdash \{P\}\, C\, \{Q\}$.

Intuitively, the meaning of the judgment is that if the computation C is executed in an environment where the initial state satisfies the precondition P, and all transitions of the environment satisfy the rely R, then all atomic transitions of the computation will satisfy the guarantee G, and the final state—if the computation terminates—will satisfy the postcondition Q. The actual definition is a bit more involved and can be found in [35, §7].

Figure 1 presents the most important proof rules for deriving such judgments. In the proof rules we use implications on RGSep assertions and interpret them such that any heap satisfying the antecedent must also satisfy the consequent. First off, something all these rules have in common is that they check, explicitly or implicitly, that the precondition and postcondition of the derived judgment are stable under the rely. The precondition should be stable because by the time the program gets to execute, the environment may have executed a few operations and thus changed the initial state. Stability ensures that the precondition continues to hold until the program gets to execute. The postcondition should similarly be stable because the environment could also change

the state after the program returns, invalidating the postcondition. In that case, the postcondition would be useless. Rules containing a judgment in their premises check for stability implicitly, since all rules introducing fresh pre- and postconditions explicitly check for stability. (The pre- and postcondition in the local rule and allocation rule are local assertions and thus automatically stable.)

We discuss the rules in turn.

(Conseq) The consequence rule allows one to strengthen the assumptions—namely, the precondition and the rely—and to weaken the postcondition and the guarantee.

(Frame) RGSep's frame rule is an adaptation of the corresponding rule of separation logic. Since, however, the RGSep frame F can also assert a property about the shared state, it is important that this property be stable under any possible changes to the shared state—either by the computation C itself or by its environment. For this reason, we require F to be stable under $R \cup G$.

(Val) The value rule represents the rule for termination and is quite straightforward.

(Assert) The assert rule is similar; it additionally requires the assertion to hold in the precondition.

(Local) The local rule allows one to use standard separation logic and verify a computation that only accesses local state. This is reflected both in the precondition and postcondition, which have to refer only to local assertions, and in the rely and guarantee, which can be arbitrary. With the local rule, we can thus reason about memory allocation and accesses to the local region.

(Let) The sequencing rule is the straightforward adaptation of the corresponding Hoare logic rule. Since the two computations execute in sequence, they share the same environment, and so they can have the same rely and guarantee components. (RGSep's proof rules for conditionals and loops are similar adaptations of the corresponding Hoare logic rules.)

(Par) The parallel composition rule combines the parallel composition rules of separation logic and rely-guarantee logic. Two threads may be executed in parallel if the local parts of the preconditions of the two threads are disjoint and each thread's specification for the shared part is aware of the presence of the concurrent thread. That is, computation C_1 is executed in an environment consisting of the outside environment of the parallel composition along with computation C_2; thus, its rely must be $R \cup G_2$ (and symmetrically $R \cup G_1$ for C_2). Each step of the parallel composition belongs to either C_1 or C_2; therefore the composition's guarantee is the union of the guarantees of its components, $G_1 \cup G_2$. We take the composition's precondition to be the separating conjunction of the preconditions of the two threads, meaning initially each thread has its own local state. As the separation of their local states should be maintained by the threads, the composition's postcondition is the separating conjunction of the postconditions of the two threads. Finally, since parallel compositions return the value returned by the first thread, Q_2 cannot mention a return value.

(AtRead) The atomic read rule first checks that the precondition P and the postcondition Q are stable. Second, it checks that the precondition implies that the read location is shared and contains some value v, which could be further specified in the frame assertion A. Third, the knowledge that the atomic read returns that value v along with the remaining shared-state assertions in A should imply the postcondition.

(AtWrite) The atomic write rule has similar premises as the atomic read rule. The key differences are: (1) that the value of the location l is not read but rather updated to value v'; (2) that the modification of the location should be allowed by the guarantee G; and (3) that the modification of the location may moreover transfer the heap described by A_{L2S} from the local state to the shared state.

(CAS) The atomic compare-and-swap rule is essentially a combination of the AtRead rule and the AtWrite rule. Similarly to the AtRead rule, it checks that the knowledge derived from the return value along with the remaining shared-state assertions in A implies the postcondition. Here, the CAS returns 1 if the value read matches the queried value v, and 0 if it does not. Thus, in the latter

case we know that the values do not match and as the CAS has not modified the state and by stability P still holds. Similarly to the AtWrite rule, the CAS rule allows transfer of ownership of the heap described by A_{L2S}. On the other hand, an important difference between the AtWrite rule and the CAS rule is that the modification of the location should be allowed by the guarantee G not for all previous values, but only when the location points to the value queried by the CAS operation.

We remark that our CAS rule allows transferring ownership only from the local region to the shared region, not the other way around. That is for the sake of simplicity and we plan to lift this limitation in the near future.

3 Using RGSep to Verify Release-Acquire Programs

Before we move on to the formalisms needed for the soundness proof, in this section we would like to give the reader an idea of how RGSep can be applied to reason about RA programs.

Read and Ignore Value. We start with a simple derived rule for atomic reads that does not assert anything about the value being read.

$$\frac{\begin{array}{l}P \text{ s.u. } R \\ P \Rightarrow \boxed{\exists v.\, l \mapsto v} * \mathsf{true}\end{array}}{R, G \vdash \{P\}\ [l]\ \{P\}}\ \text{AtRead-Ignore}$$

To derive this rule, we first apply the AtRead rule with $A \triangleq \mathsf{true}$ to get

$$R, \emptyset \vdash \left\{\boxed{\exists v.\, l \mapsto v}\right\}\ [l]\ \{\mathsf{emp}\}\,.$$

To do so, we technically need to show that $\boxed{\exists v.\, l \mapsto v}$ s.u. R. This always holds in our RGSep fragment, since we do not support ownership transfer from the shared region to the local region. We then apply the Frame rule with $F \triangleq P$ to obtain

$$R, \emptyset \vdash \left\{\boxed{\exists v.\, l \mapsto v} * P\right\}\ [l]\ \{\mathsf{emp} * P\}$$

and finally the Conseq rule, observing that $P \Rightarrow \boxed{\exists v.\, l \mapsto v} * P$ and $\mathsf{emp} * P \Rightarrow P$.

Atomic Increment. Consider the following implementation of an atomic increment operation, which repeatedly reads the value of a location and attempt to change its value to its increment with a CAS operation until the CAS succeeds.

$$\mathsf{inc}(l) \ \triangleq\ \textbf{repeat}\ (\textbf{let}\ a = [l]\ \textbf{in}\ \mathsf{cas}(l, a, a+1))$$

Using the existing rules, we can obtain a specification similar to RGSep's rule for CAS

$$\frac{\begin{array}{l}P, Q \text{ s.u. } R \\ P \Rightarrow A_{\text{L2S}} * \boxed{\exists v.\, l \mapsto v * A} \\ \boxed{l \mapsto v + 1 * A * A_{\text{L2S}}} \Rightarrow Q \\ (l \mapsto v \rightsquigarrow l \mapsto v + 1 * A_{\text{L2S}} \mid A) \subseteq G\end{array}}{R, G \vdash \{P\}\ \mathsf{inc}(l)\ \{Q\}}\ \text{Inc}$$

with the following proof outline:

$$R, G \vdash \{P\}\ \textbf{repeat}\ (\{P\}\ \textbf{let}\ a = [l]\ \textbf{in}\ \{P\}\ \mathsf{cas}(l, a, a+1)\ \{x.\ (x = 0 \wedge P) \vee (x = 1 \wedge Q)\})\ \{Q\}$$

by applying the AtRead-Ignore rule, the CAS rule, the Let rule, and the Loop rule.

Monotonic Increment. Our first example is a coherence test, where x is only incremented.

$$\begin{array}{l}\textbf{let } a = [x] \textbf{ in}\\ \textbf{let } b = [x] \textbf{ in}\\ \textbf{assert}(b \geq a)\end{array} \;\Big\|\; \begin{array}{l}[x] := 1;\\ [x] := 2\end{array} \tag{CohRR}$$

Assuming that location x initially contains 0, the assertion of Thread 1 holds under RA. Using RGSep, we can establish the correctness of the program with the following proof outline:

$$\emptyset, G \vdash \boxed{x \mapsto 0} \left(\begin{array}{l} G, \emptyset \vdash \\ \boxed{\exists v.\, x \mapsto v \wedge v \geq 0} \\ \textbf{let } a = [x] \textbf{ in} \\ \boxed{\exists v.\, x \mapsto v \wedge v \geq a} \\ \textbf{let } b = [x] \textbf{ in} \\ \{b \geq a \wedge \mathsf{emp}\} \\ \textbf{assert}(b \geq a) \\ \{\mathsf{emp}\} \end{array} \;\Big\|\; \begin{array}{l} \emptyset, G \vdash \\ \boxed{x \mapsto 0} \\ [x] := 1 \\ \boxed{x \mapsto 1} \\ [x] := 2 \\ \boxed{x \mapsto 2} \end{array} \right) \boxed{x \mapsto 2}$$

where $G = \{(x \mapsto v \rightsquigarrow x \mapsto v{+}1) \mid v \in \mathrm{Val}\}$, capturing the fact that Thread 2 can only increment x. It is easy to see that the assertions of Thread 1 are stable under G and that Thread 2 satisfies the guarantee G.

We note that the assertion in CohRR does not hold under memory models that allow the reordering of two reads of the same memory location, such as the Java memory model [25].

Message Passing. Our second example is the message passing idiom:

$$\begin{array}{l}[x] := 7;\\ [y] := 1\end{array} \;\Big\|\; \begin{array}{l}\textbf{repeat } [y];\\ \textbf{let } b = [x] \textbf{ in}\\ \textbf{assert}(b = 7)\end{array} \tag{MP}$$

Assuming that location y initially contains 0, the assertion in Thread 2 holds under RA because the only way in which the loop can exit is if the read of y reads from the $[y] := 1$ write of Thread 1. Under RA, this implies that the $[x] := 7$ write must already have been seen by Thread 2, and so the read of x must return 7.

This property can be nicely shown in RGSep with the following proof outline:

$$G, G \vdash \boxed{\begin{array}{l}A_{X0} \vee \\ A_{71}\end{array}} \left(\begin{array}{l} G, G \vdash \\ \boxed{A_{X0} \vee A_{71}} \\ [x] := 7; \\ \boxed{A_{70} \vee A_{71}} \\ [y] := 1 \\ \boxed{A_{71}} \end{array} \;\Big\|\; \begin{array}{l} G, \emptyset \vdash \\ \boxed{A_{X0} \vee A_{71}} \\ \textbf{repeat} \\ \quad \boxed{A_{X0} \vee A_{71}} \\ \quad [y] \\ \quad \{v.\ \left(v = 0 \wedge \boxed{A_{X0} \vee A_{71}}\right) \vee \left(v = 1 \wedge \boxed{A_{71}}\right)\} \\ \boxed{A_{71}} \\ \textbf{let } b = [x] \textbf{ in} \\ \{b = 7\} \\ \textbf{assert}(b = 7) \end{array} \right) \boxed{A_{71}}$$

where $G = \{(x \mapsto v \rightsquigarrow\!\!\!\!\rightsquigarrow x \mapsto 7), (y \mapsto v \rightsquigarrow\!\!\!\!\rightsquigarrow y \mapsto 1 \mid x \mapsto 7) \mid v \in \mathbb{N}\}$

$A_{X0} = x \mapsto \_ * y \mapsto 0 \qquad A_{70} = x \mapsto 7 * y \mapsto 0 \qquad A_{71} = x \mapsto 7 * y \mapsto 1\,.$

Again, it is straightforward to see that the assertions in the proof outline are stable under the relevant rely.

By applying the Par rule, one can further establish that the same specification holds for multiple parallel copies of MP, namely

$$G, G \vdash \left\{\boxed{A_{X0} \vee A_{71}}\right\} \mathsf{MP} \;\|\; \mathsf{MP} \left\{\boxed{A_{71}}\right\}$$

We note that the assertion in the MP program does not hold under the relaxed fragment of the C++ memory model nor under the slightly stronger *strong coherence model* (SCOH), indicating that RGSep is unsound under those models.

Coherence. Consider the following example where two threads write to and read from the same shared location.

$$\begin{array}{l} [x] := 1; \\ \textbf{let } a = [x] \textbf{ in} \\ [y] := a \end{array} \;\Big\|\; \begin{array}{l} [x] := 2; \\ \textbf{let } b = [x] \textbf{ in} \\ [z] := b \end{array} \tag{Coh}$$

Under RA, it can never happen that Thread 1 reads 2 while Thread 2 reads 1. In contrast, this weak outcome is possible under the relaxed fragment of the C++ memory model and under the slightly stronger *strong coherence model* (SCOH). With RGSep, we can prove that the said outcome is impossible with the following proof outline, which means that RGSep is unsound under SCOH.

$$\left\{y \mapsto \_ * z \mapsto \_ * \boxed{x \mapsto \_}\right\}$$

$$\begin{array}{l|l} G_2, G_1 \vdash & G_1, G_2 \vdash \\ \left\{y \mapsto \_ * \boxed{x \mapsto \_}\right\} & \left\{z \mapsto \_ * \boxed{x \mapsto \_}\right\} \\ [x] := 1; & [x] := 2; \\ \left\{y \mapsto \_ * \boxed{x \mapsto 1 \vee x \mapsto 2}\right\} & \left\{z \mapsto \_ * \boxed{x \mapsto 1 \vee x \mapsto 2}\right\} \\ \textbf{let } a = [x] \textbf{ in} & \textbf{let } b = [x] \textbf{ in} \\ \left\{y \mapsto \_ * \left(a = 1 \vee a = 2 \wedge \boxed{x \mapsto 2}\right)\right\} & \left\{z \mapsto \_ * \left(b = 1 \wedge \boxed{x \mapsto 1} \vee b = 2\right)\right\} \\ [y] := a & [z] := b \\ \left\{y \mapsto 1 \vee y \mapsto 2 * \boxed{x \mapsto 2}\right\} & \left\{z \mapsto 1 * \boxed{x \mapsto 1} \vee z \mapsto 2\right\} \end{array}$$

$$\{y \mapsto 1 * z \mapsto 1 \vee y \mapsto 2 * z \mapsto 2 \vee y \mapsto 1 * z \mapsto 2\}$$

where $G_v = \{(x \mapsto v' \rightsquigarrow x \mapsto v) \mid v' \in \mathbb{N}\}$.

Further RGSep Proofs. Given that the proof rules supported by our framework are restricted subset of the original RGSep rules, there are numerous RGSep proofs in the literature that we could re-use without modification to show correctness under RA. More concretely, we can re-use all existing RGSep proofs as long as they (1) do not use atomic blocks beyond CAS and (2) do not use ownership transfer from shared to local. The latter is essentially only needed in programs that deallocate shared memory cells, such as the lock coupling list of [37]. However, if we remove the deallocation instruction from the lock coupling list, we can trivially adapt the proof.

Thus, in principle, we can support the automatically-generated proofs of all but one of algorithms of [34, Fig. 1]: we do not support the ninth algorithm because it uses a DCAS operation, which does not exist in the RA model. Similarly, we do not support any of the RGSep linearizability proofs because they require general atomic blocks, which also do not exist in RA.

4 The Release-Acquire Memory Consistency Model

In this section, we define the *release-acquire* (RA) memory consistency model, which is a fragment of the C11 memory model [1, 23]. While RA is typically defined axiomatically [20], we will present an operational definition in the style of Kang et al. [17] because it is more suitable for defining the meaning of RGSep judgments and thereby showing the soundness of the RGSep proof rules.

Memory and Views. Unlike SC, where the memory is represented as a map from locations to values, under RA memory is represented as a set of messages M, each message representing an atomic write to memory. Each message is a tuple, $(l, v @ t, v_r, \mathscr{R})$, that records the location updated l, the value written v, a timestamp $t \in \mathbb{N}$, which is globally unique across messages writing to the same location, the value read v_r in the case of an RMW (or $\bot$ in the case of a normal write), and the message view $\mathscr{R}$: a set of messages that happened before the current one. A message m_1 *happens before* or is *hb-ordered* before another message m_2, if either

- m_1 and m_2 result from writes or RMWs executed by the same thread and the operation associated with m_1 appears earlier in the computation, or
- m_1 and m_2 result from writes or RMWs executed by different threads, but an earlier operation in the second thread reads from m_1 or a later message in the first thread .

To avoid a recursive definition due to the fact that each message contains a view, which is itself a set of messages, $\mathscr{R}$ is represented as a set of location-timestamp pairs. Such pairs uniquely determine a message in a valid memory configuration and do not contain views themselves.[1]

In addition to the set of messages M, each thread records its view: the set of messages it is aware of. We define a pair of a view and a message set $(\mathscr{V}, M)$ as *well-formed*, if

- $\mathscr{V}$ only refers to messages in M,
- whenever $\mathscr{V}$ contains a message m, it also contains all messages in m's view,
- the hb-order indicated by message views in M is transitive: if the view of a message in M contains another message, the message view also contains that message's view,
- whenever M contains a message m, m's view only consists of messages in M (that have been added to M before m) and does not contain m itself,
- no two messages in M share the same location and timestamp,
- all messages in M are either *initialization messages* in M (messages with timestamp 0) or happen after another messages writing to the same location, and
- for any RMW message m, there exists a message m_r that the RMW reads from; m_r happens before m, has the value recorded in m and the timestamp directly preceding m's timestamp.

Operational Semantics. Figure 2 presents the RA operational semantics. Physical program configurations (or physical configurations) are triples $\langle C, \mathscr{V}, M\rangle$, comprising a computation C, its view $\mathscr{V}$, and the memory M.

(IncView) First, at any point, a computation may increase its view $\mathscr{V}$ by incorporating a message $m \in M$ as long as $\mathscr{V}$ already contains all messages in m's view. This constraint enforces the basic causality property of RA: if a thread is aware of a certain update, it must also be aware of all previous updates.

(Read) Reading from a memory location l returns the value recorded in the message with the largest timestamp in the view, which is defined as follows:

$$|\mathscr{V}|(l) \triangleq \max\{t \mid \exists v, v_r, \mathscr{R}.\ (l, v @ t, v_r, \mathscr{R}) \in \mathscr{V}\}$$

[1] Kang et al. [17] represents views as functions from locations to timestamps. We prefer the more explicit presentation of views as set of messages because it simplifies the definition of the RGSep safety predicate in §5.

$$\frac{m = (l, v @ t, v_r, \mathscr{R}) \in M \quad \mathscr{R} \subseteq \mathscr{V}}{\langle C, \mathscr{V}, M\rangle \to \langle C, \mathscr{V} \cup \{m\}, M\rangle}\ \text{IncView}$$

$$\frac{(l, v @ t, v_r, \mathscr{R}) \in M \quad |\mathscr{V}|(l) = t}{\langle [l], \mathscr{V}, M\rangle \to \langle v, \mathscr{V}, M\rangle}\ \text{Read} \qquad \frac{\text{fresh}(M, l, t) \quad t > |\mathscr{V}|(l) \quad m = (l, v @ t, \bot, \mathscr{V})}{\langle [l] := v, \mathscr{V}, M\rangle \to \langle 0, \mathscr{V} \cup \{m\}, M \cup \{m\}\rangle}\ \text{Write}$$

$$\frac{(l, v @ t, v_r, \mathscr{R}) \in M \quad |\mathscr{V}|(l) = t \quad \text{fresh}(M, l, t+1) \quad m = (l, v' @ t+1, v, \mathscr{V})}{\langle \text{cas}(l, v, v'), \mathscr{V}, M\rangle \to \langle 1, \mathscr{V} \cup \{m\}, M \cup \{m\}\rangle}\ \text{CAS-Success}$$

$$\frac{(l, w @ t, v_r, \mathscr{R}) \in M \quad |\mathscr{V}|(l) = t \quad w \neq v}{\langle \text{cas}(l, v, v'), \mathscr{V}, M\rangle \to \langle 0, \mathscr{V}, M\rangle}\ \text{CAS-Fail}$$

$$\frac{l \notin \text{Locs}(M) \quad m = (l, 0 @ 0, \bot, \mathscr{V})}{\langle \textbf{alloc}, \mathscr{V}, M\rangle \to \langle C, \mathscr{V} \cup \{m\}, M \cup \{m\}\rangle}\ \text{Alloc} \qquad \frac{v \neq 0}{\langle \textbf{assert}(v), \mathscr{V}, M\rangle \to \langle 0, \mathscr{V}, M\rangle}\ \text{Assn}$$

$$\frac{\langle C_1, \mathscr{V}, M\rangle \to \langle C_1', \mathscr{V}', M'\rangle}{\langle \textbf{let } x = C_1 \textbf{ in } C_2, \mathscr{V}, M\rangle \to \langle \textbf{let } x = C_1' \textbf{ in } C_2, \mathscr{V}', M'\rangle}\ \text{Let}$$

$$\frac{}{\langle \textbf{let } x = v \textbf{ in } C_2, \mathscr{V}, M\rangle \to \langle C_2[v/x], \mathscr{V}, M\rangle}\ \text{Let-Val}$$

$$\frac{}{\langle C_1 \parallel C_2, \mathscr{V}, M\rangle \to \langle {C_1}^{\mathscr{V}} \|^{\mathscr{V}}_{\text{true}} C_2, \mathscr{V}, M\rangle}\ \text{Par-Begin} \qquad \frac{}{\langle {v_1}^{\mathscr{V}} \|^{\mathscr{V}}_{\text{true}} v_2, \mathscr{V}, M\rangle \to \langle v_1, \mathscr{V}, M\rangle}\ \text{Par-End}$$

$$\frac{}{\langle {C_1}^{\mathscr{V}_1} \|^{\mathscr{V}_2}_b C_2, \mathscr{V}, M\rangle \to \langle {C_2}^{\mathscr{V}_2} \|^{\mathscr{V}_1}_{\neg b} C_1, \mathscr{V}, M\rangle}\ \text{Par-Swap}$$

$$\frac{\langle C_1, \mathscr{V}_1, M\rangle \to \langle C_1', \mathscr{V}_1', M'\rangle}{\langle {C_1}^{\mathscr{V}_1} \|^{\mathscr{V}_2}_b C_2, \mathscr{V}, M\rangle \to \langle {C_1'}^{\mathscr{V}_1'} \|^{\mathscr{V}_2}_b C_2, \mathscr{V} \cup \mathscr{V}_1', M'\rangle}\ \text{Par-Thread}$$

Fig. 2. Operational semantics for RA programs (extract)

Remark Most operational RA definitions in the literature allow reads to read from any timestamp-wise larger message, seen or unseen, not just the largest one in the view, and to suitably update the thread view. We can, however, simulate this effect by first increasing the view and then reading the message. For example, say we have $M = \{m_0, m_1, m_2, m_3\}$ with $m_i = (l, i @ i, \bot, \{m_0, \ldots, m_{i-1}\})$ for $i = 0, 1, 2, 3$ and $\mathscr{V} = \{m_0, m_1\}$. To simulate a read from m_2 and update $\mathscr{V}$ accordingly, we can simply first perform a view increase by m_2 such that $\mathscr{V}$ becomes $\{m_0, m_1, m_2\}$. Then $|\mathscr{V}|(l)$ becomes 2 and so we can read from m_2.

(Write) Writing to a memory location l creates a new message m, which is added to both the memory and the thread's view. The timestamp of the message has to be fresh (i.e., there shouldn't exist another message in M with the same location and timestamp), as otherwise the new message would violate well-formedness of $(\mathscr{V}, M)$. Moreover, the timestamp should be greater than all the timestamps of location l in the view $\mathscr{V}$.

(CAS-Success) Successful RMWs are similar with the extra condition that the timestamp should precisely follow the timestamp of the message it read from.

(Alloc) Allocation assigns the value 0 to a fresh location. We use $\mathrm{Locs}(M) := \{m.\mathrm{loc} \mid m \in M_{\mathrm{diff}}\}$ to denote the set of locations of messages in M.

The semantics for assertions, let-bindings, conditionals, and loops are standard (the rules for the latter two constructs are omitted for brevity).

(Par) The treatment of parallel composition is more interesting. It requires us to extend the syntax of computations C with a new construct representing the parallel execution of two threads:

$$C ::= \ldots \mid C_1 \,{}^{\mathscr{V}_1}\|^{\mathscr{V}_2}_b\, C_2$$

We will refer to this construct as the *ParExec* (parallel execution) construct. It records the two threads executing in parallel with their respective views and a boolean b, the meaning of which will become clear shortly. ParExec constructs are not meant to be used in programs and only appear as part of the operational semantics so as to track the views of the corresponding threads.

The boolean b is needed as we made the following design choice: at all times only the left thread may take a step. This design choice will be advantageous when we define safety later in §5. In the face of this design choice, in order for the right thread to take a step we allow *swapping* the order of the two threads at any time. Despite that, upon termination we wish to return the value of the main thread which was initially the left one. We keep track of the current positions of the two threads using the boolean b. Whenever b is true the threads are in their initial position. Otherwise, if b is false, we know that the order of the threads has been swapped compared to the initial order.

Initially, a parallel composition is expanded to a ParExec construct by recording the thread view $\mathscr{V}$ as the view of both threads and initially setting b to be true. We swap the order of the threads by simply swapping them and their views while negating b to signal that the order has been reversed. The parallel composition returns when both threads finish, are back in their initial positions, and reach the same view $\mathscr{V}$. The latter constraint may strike the reader as odd, but is harmless: the views of the two threads can always be increased (by applying rule IncView repeatedly) so that they match.

(Par-Thread) The final rule in Fig. 2 covers the scenario where the left thread in the parallel composition takes a step. In that case, the ParExec construct also takes the corresponding step recording the updated components.

The Par-Thread rule features another design choice that we made. We augment the view of the parallel composition so that it always includes the views of its components. This update is not strictly necessary because the view of the parallel composition is never really used, and we could have just as well kept $\mathscr{V}$ as is. Nevertheless, the fact that the view of a parallel composition always includes the views of its components had some benefits regarding the proof of soundness discussed in §6. Thus, we included this fact as a well-formedness condition on physical configurations. We say that a physical configuration $\langle C, \mathscr{V}, M\rangle$ is well-formed if

- the pair $(\mathscr{V}, M)$ is well-formed, and
- for any parallel composition $C_1 \,{}^{\mathscr{V}_1}\|^{\mathscr{V}_2}_b\, C_2$ in C, $\mathscr{V}_1 \subseteq \mathscr{V}$ and $\mathscr{V}_2 \subseteq \mathscr{V}$, and whenever C_1 resp. C_2 contains another parallel composition $C_x \,{}^{\mathscr{V}_x}\|^{\mathscr{V}_y}_b\, C_y$, then $\mathscr{V}_x \subseteq \mathscr{V}_1$ and $\mathscr{V}_y \subseteq \mathscr{V}_1$ resp. $\mathscr{V}_x \subseteq \mathscr{V}_2$ and $\mathscr{V}_y \subseteq \mathscr{V}_2$.

Abort Semantics. Figure 3 presents the abort semantics of our programming language that declares certain program configurations as erroneous. A memory access (a read, a write or a CAS) aborts if the location accessed is not allocated according to the knowledge of the thread performing it; i.e., if the view does not contain any write to that location. Sequential and parallel compositions abort if their first component (which is currently executing) aborts.

$$\frac{l \notin \mathrm{Locs}(\mathscr{V})}{\langle [l], \mathscr{V}, M\rangle \to \textbf{abort}}\ \text{Read-Abort} \qquad \frac{l \notin \mathrm{Locs}(\mathscr{V})}{\langle [l] := v, \mathscr{V}, M\rangle \to \textbf{abort}}\ \text{Write-Abort}$$

$$\frac{l \notin \mathrm{Locs}(\mathscr{V})}{\langle \mathrm{cas}(l, v, v'), \mathscr{V}, M\rangle \to \textbf{abort}}\ \text{CAS-Abort} \qquad \frac{}{\langle \textbf{assert}(0), \mathscr{V}, M\rangle \to \textbf{abort}}\ \text{Assert-Abort}$$

$$\frac{\langle C_1, \mathscr{V}, M\rangle \to \textbf{abort}}{\langle \textbf{let}\ x = C_1\ \textbf{in}\ C_2, \mathscr{V}, M\rangle \to \textbf{abort}}\ \text{Seq-Abort} \qquad \frac{\langle C_1, \mathscr{V}_1, M\rangle \to \textbf{abort}}{\langle C_1 \,{}^{\mathscr{V}_1}\|_b^{\mathscr{V}_2} C_2, \mathscr{V}, M\rangle \to \textbf{abort}}\ \text{Par-Abort}$$

Fig. 3. Abort semantics capturing when a configuration is erroneous.

5 Interpreting RGSep Judgments over Release-Acquire

In this section, we interpret RGSep judgments over the RA semantics, thus defining semantic correctness, and explain how our interpretation differs from the SC one. We first describe on a high level which properties a correct program should satisfy (§5.1). We then proceed by listing and discussing the ghost state needed to define semantic correctness for RGSep under RA, defining our notion of *logical configurations* (§5.2). We use these configurations to formally define our notion of correctness (§5.3). To conclude, we then show that our notion of correctness is adequate (§5.4).

5.1 High-level Definition of Safety

To show soundness of the syntactic proof rules given in §2.4, we have to define what it means for the execution of a computation to satisfy a given RGSep specification. Recall that RGSep specifications comprise of four components: the precondition P, the postcondition Q, the rely R and the guarantee G. The idea is that a computation C satisfies such a specification if whenever it is run in an initial state satisfying the precondition P and in a concurrent environment that only modifies the (shared) state according to R, then the following three conditions hold. Firstly, if the computation terminates, the final state satisfies Q. Secondly, the computation C does not abort by accessing uninitialized memory at any time. Third, all modifications to the shared state conducted by the computation C are in the guarantee G.

Following Vafeiadis [35], we will define semantic program correctness using an inductively defined safety predicate safe_n over (logical) program configurations. The predicate safe_n asserts that a logical configuration (a physical configuration with some extra information attached) with a postcondition Q is safe for n steps. That is, the corresponding physical configuration does not reach an aborting configuration within the next n steps and if the program terminates within n steps, then the logical configuration at the point of termination satisfies the postcondition. A computation semantically satisfies an RGSep specification, $R, G \vDash \{P\}\, C\, \{Q\}$, if every initial logical configuration of C satisfying the precondition P satisfies safe_n under postcondition Q for all $n \in \mathbb{N}$.

A logical program configuration consists of multiple entries and contains information relevant for verification at a certain point during the execution of a program. In the original RGSep under SC a logical configuration consists of the the program C to be executed, the shared and local heaps, the rely and the guarantee. That is, it consists of the program, the program state and the additional ghost state needed for verification. Later we will define logical configurations analogously for RA.

Overarchingly, we would like to define the safety predicate for RGSep under RA such that a logical configuration is safe for $n + 1$ steps if

(1) whenever the program has reduced to a value (i.e. the program has terminated), the current state satisfies the postcondition,
(2) the configuration does not abort,
(3) whenever the program takes a step, its change to the shared state satisfies the guarantee and the resulting configuration is safe for another n steps,
(4) whenever the environment modifies the shared state in a way given by the rely, the new configuration is safe for another n steps.

Before we formalize this, however, there is one addition we need to make to the safety predicate. This addition is the case of a *view increase*. We exclude view increases from the program steps in case (3) of our safety definition, because they need to be treated differently. The modification of the program state due to the view increase cannot, in general, satisfy the guarantee because the thread increasing its view is not the one that created the message and hence has no control over which constraints are satisfied by the corresponding state modification. We will thus add the following case to our safety predicate to cover view increases.

(5) whenever a thread of the program increases its view by some existing message, the resulting configuration is safe for another n steps.

5.2 Logical Configurations

In RGSep's soundness proof against SC, computations directly manipulated the global heap, a simple map from memory locations to values. Thus, it was sufficient to add instrumentation (i.e., ghost state we add for the sake of the soundness proof) about how the total heap is partitioned into local and shared regions.

Since RA has a more complex state model with message sets and views, proving soundness of the same logic under RA requires more fine-grained instrumentation. Besides partitioning the global state into local and shared parts, we also have to track certain properties at the level of messages. There are four bits of information that we record for each message m: (1) the precondition of the message, (2) its rely and (3) its guarantee from the perspective of the thread that created the message, and (4) the ownership transfer resulting from it.

This information will be saved in what we will refer to as an *annotation map*, which maps each message to a tuple containing all of its *annotations*, i.e., all pieces of information we track per message for the sake of the soundness proof.

The message precondition is an assertion about the shared state that held at the point when that message was first created (i.e., the write or RMW was first executed), and ensures that the message's location is allocated and shared, meaning we can safely update the location.

We require message preconditions to be stable under the rely of the thread adding the message. Through this requirement we can prove the following. The message's precondition (once established) continues to hold at the point when it is incorporated into the view V of another thread through a view increase, no matter how many other messages were added to V in between. By construction, at the point when a message m is incorporated into the view V of some thread, V contains all the messages in m's view. Any additional messages that V contains cannot have been added by the thread that created m itself. (Recall that messages are observed in hb-order and that messages created by the same thread are hb-ordered with respect to each other. Therefore, if the thread that created message m also created other message $m' \in V$, then, as $m \notin V$, m' would have to be ordered before m, contradicting the fact that m.view consists of all messages ordered before m.) Thus, any message added to V between m.view and m must originate from some thread running in parallel to the thread T that created m. As all those threads behave according to T's rely, none of those messages can change the validity of the stable message precondition.

As mentioned above, we also annotate messages with the rely and guarantee from the perspective of the thread that created the message. More precisely, the thread rely consists of the set of actions that could be carried out by any thread in the environment other than the thread that spawned the message, while the thread guarantee consists of the set of actions that could be carried out by that thread itself. Given that we require message preconditions to be stable under the rely of a specific thread, it should make sense why we need the thread rely. The reason why we additionally track the message guarantee is that we need to ensure the following for safety. For any two distinct threads, the thread guarantee of one is contained in the thread rely of the other. That means that the actions in each thread (as long as they obey the thread guarantees) are actually tracked by the relies of other threads. We can state this criterion via messages by letting their associated thread guarantee be contained in another message's thread rely whenever the two distinct messages are not hb-ordered. By assuming that the messages are not hb-ordered we indirectly constrain them to originate from different threads, since within a thread all writes are hb-ordered according to program order. Thus, we obtain a slightly weaker criterion (messages from distinct threads can be hb-ordered), which is still strong enough to prove safety.

Finally, we also keep track of ownership transfer of a certain message by annotating that message with the set of locations mL2S that are shared when that message comes into effect. Annotating messages in such a way is necessary because we cannot simply add the transferred locations to the set of shared locations immediately when the message is created. Rather, because threads can have differing views and disagree on the set of shared locations, we have to keep track of the transferred locations to perform the ownership transfer at a later time.

We illustrate this point with the following OT program along with its RGSep correctness proof in Fig. 4.

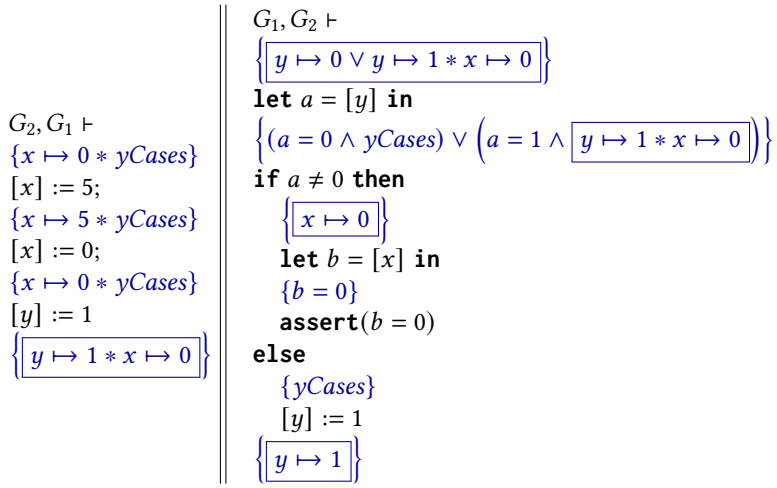
$$
\begin{array}{l}
[x] := 5; \\
[x] := 0; \\
[y] := 1
\end{array}
\;\Big\|\;
\begin{array}{l}
\textbf{let } b = [y] \textbf{ in} \\
\textbf{if } b \neq 0 \textbf{ then} \\
\quad \textbf{let } c = [x] \textbf{ in} \\
\quad \textbf{assert}(c = 0) \\
\textbf{else} \\
\quad [y] := 1
\end{array}
\tag{OT}
$$

The program comprises two threads. Thread 1 performs some assignments to x and then writes 1 to location y. Thread 2 reads y and if y contains a non-zero value, it checks that location x contains value 0; otherwise, it writes value 1 to y.

As shown in Fig. 4, if x and y initially contain value 0, then this program is provably correct.

We will assume the assertion $\boxed{y \mapsto 0}$ to be part of the precondition, making y shared from the beginning. As x appears on both sides of the parallel composition, clearly the location of x must also be considered shared from some point in the program. If we were to consider x to be a shared from the very beginning as well, however, we would run into problems during verification. The issue is that the actions $x \mapsto 0 \rightsquigarrow x \mapsto 5 \mid y \mapsto 0$ and $x \mapsto 0 \rightsquigarrow x \mapsto 5 \mid y \mapsto 1$ would have to be in the guarantee of Thread 1. Consequently, we cannot prove that x contains value 0 before the assert, as any assertion implying $\boxed{x \mapsto 0}$ will not be stable under the guarantee of thread 1. Therefore, x has to be initially local to Thread 1.

But when should we transfer ownership of x? On the level of the RGSep logic, the answer is that x should be transferred from local to shared with the assignment to y in Thread 1. That is because that assignment is an assignment to an already shared location and as such has an effect visible to the other thread. This visible effect is needed to signal to the other thread that the location x is now shared. That is also true on the level of the semantics. Due to possibly inconsistent views only when Thread 2 has observed the write to y of Thread 1 can we be sure that Thread 2 has observed



where $G_1 = \{(y \mapsto v \rightsquigarrow y \mapsto 1 * x \mapsto 0) \mid v \in \mathbb{N}\}$

$G_2 = \{(y \mapsto v \rightsquigarrow y \mapsto 1) \mid v \in \mathbb{N}\}$

$yCases = \boxed{y \mapsto 0 \vee y \mapsto 1}$

Fig. 4. Verification of OT.

the final assignment to x before it should be shared. Hence, in general, ownership can only be transferred with an assignment to a shared location.

Now, let us get to the point of this example. Why can we not simply transfer ownership of x for all threads as soon as the write to y is added to the message set? Say, we want to prove semantic safety of Thread 2 using the safety predicate outlined in §5.1. Further say the environment (Thread 1) starts by adding all of its messages (thus performing environment steps). If we add x to the set of shared locations directly upon adding the message transferring ownership to the message set, we run into a problem. We have not performed any view increases yet, and so Thread 2 has not observed any of the assignments to x so far. Thread 2 will first add the message setting x's value to 5 to its view. Here it gets problematic. As Thread 2 wrongfully considers x shared already (because we updated its logical state too early), from its point of view it observes a write to a shared location and hence expects the associated action to be in its rely. But the action associated with this assignment is not, and has no reason to be, in Thread 2's rely! After all the assignment was not meant to be an update of the shared state.

The above problem can be easily avoided by only adding shared locations when the message transferring ownership is being observed. Hence we need the annotation mL2S to keep track of ownership transfer.

Now that we have outlined the additional instrumentation needed to prove safety of RGSep under RA semantics, we can properly define our notion of a logical configuration and what it means for such a configuration to be safe. That is, we adapt the definitions used in the SC case from using a state modeled via simple heaps to a state modeled via pairs of message sets and views and add our additional instrumentation.

We define our *logical configurations* (or configurations where it is clear from context) to be of the form $(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G)$ for a program C, a message set M, a view set $\mathscr{V}$, an annotation map

$\mathscr{A}$, sets of locations L and S_{init}, rely R, and guarantee G. The annotation map maps messages to tuples (mPre, mRely, mGuar, mL2S) where mPre is the message precondition, mRely and mGuar are thread rely and guarantee respectively and mL2S is the set of locations shared by the message. In a configuration $(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G)$, the entries C, M, and $\mathscr{V}$ give the current state. Rely and Guarantee are self-explanatory and $\mathscr{A}$ has been explained above. L defines the set of local locations, whereas S_{init} describes the set of *initially* shared locations. Here, by initially we mean before considering any ownership transfer resulting from messages. The set of shared locations at a particular view $\mathscr{V}$ can be computed using $\mathscr{A}$, by adding the locations transferred by any seen messages to S_{init}.

$$\text{Shared}(\mathscr{V}, \mathscr{A}, S_{init}) \triangleq S_{init} \cup \bigcup_{m \in \mathscr{V}} \mathscr{A}(m).\text{mL2S}$$

The reader may wonder why we chose to include sets of shared and local locations in the configuration rather than heaps as was done in the SC case. The reason is that sets are more concise, making it easier to reason about properties such as disjointness and that heaps, compared to location sets, contain redundant information. The redundancy is due to the values being easily obtainable from the view. The shared and local heaps as in the instrumentation of RGSep under SC can be constructed from our configuration by mapping each shared resp. local location to the value of its timestamp-wise maximal message in the view. We will denote the heap constructed from a message view $\mathscr{V}$ and a set of locations Loc as $\mathscr{V}|_{Loc}$.

Thus, we can also trivially lift the definition of validity of RGSep assertions from SC to RA as follows. A message view $\mathscr{V}$ *satisfies* an RGSep assertion P with respect to sets of local locations L and initially shared locations S_{init}, written as $\mathscr{V}, L, S_{init} \models P$, if $\mathscr{V}|_L, \mathscr{V}|_{\text{Shared}(\mathscr{V},\mathscr{A},S_{init})} \models P$.

Well-formedness. Even though we spared ourselves some redundancy by using location sets instead of heaps, there are still many ways in which the different components of a logical configuration could contain contradicting information. For example, the message set and the view would contradict each other if the view contained messages outside of the memory. It may also be the case that a single component is invalid, as for example the happens-before-ordering indicated by message views of messages in the message set could be an invalid order. The above are potential cases of an invalid configuration in which it would be nonsensical to try to reason about safety. Thus, we define here formally what it means for a logical configuration to be *well-formed* for our purposes:

We consider a logical configuration $(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G)$ to be well-formed if it meets the following criteria.

(1) The physical configuration $\langle C, \mathscr{V}, M\rangle$ is well-formed.
(2) All locations in L or S_{init} are initialized: $L \cup S_{init} \subseteq \{m.\text{loc} \mid \text{InitMsg}(m) \wedge m \in \mathscr{V}\}$
(3) No location is both shared and local: $L \perp \text{Shared}(\mathscr{V}, \mathscr{A}, S_{init})$
(4) As the local messages originate from program steps, the thread being verified has seen all messages modifying local locations: $\{m \in \mathscr{V} \mid m.\text{loc} \in L\} = \{m \in M \mid m.\text{loc} \in L\}$
(5) Only messages modifying shared locations (and not initializing them) are annotated:
$dom(\mathscr{A}) = \{m \in M \mid m.\text{loc} \in \text{Shared}(m.\text{view}, \mathscr{A}, S_{init}) \wedge \neg\text{InitMsg}(m)\}$
Due to ownership transfer we consider the locations shared at the time the message was created.
(6) Every annotated message is well-annotated (defined below):
$\forall m \in dom(\mathscr{A}).\ \text{WellAnnotated}(m, \mathscr{A}, S_{init})$
(7) As we do not support ownership transfer from shared to local, a message cannot share a location that was shared before: $\forall m \in dom(\mathscr{A}).\ \mathscr{A}(m).\text{mL2S} \perp S_{init}$

(8) For the same reason, distinct messages cannot share the same location:
$\forall m, m' \in dom(\mathscr{A}).\ m \neq m' \implies \mathscr{A}(m).\mathsf{mL2S} \perp \mathscr{A}(m').\mathsf{mL2S}$
(9) If a message's location is shared by another message, the two messages have to be ordered by happens-before (and if the message whose location is shared happens after, then it cannot be the message initializing that location):
$\forall m \in dom(\mathscr{A}).\ \forall m' \in M.\ m'.\mathsf{loc} \in \mathscr{A}(m).\mathsf{mL2S} \implies$
$(m' \in m.\mathsf{view} \vee (\neg \mathrm{InitMsg}(m') \wedge m \in m'.\mathsf{view}))$
(10) Unordered messages should have compatible rely-guarantee annotations, i.e., the guarantee of each message must be included in the thread rely of the other:
$\forall\ m, m' \in dom(\mathscr{A}).\ m \neq m' \wedge m \notin m'.\mathsf{view} \wedge m' \notin m.\mathsf{view} \implies \mathrm{compat}(\mathscr{A}(m), \mathscr{A}(m'))$
(11) Since messages outside the view have been added by the environment, their guarantees should be part of the rely of the thread being verified: $\forall m \in M \setminus \mathscr{V}.\ \mathscr{A}(m).\mathsf{mGuar} \subseteq R$
(12) Conversely, the guarantee of the program we are verifying should also be part of the rely of any environment messages; $\forall\ m \in M \setminus \mathscr{V}.\ G \subseteq \mathscr{A}(m).\mathsf{mRely}$

We say that a message m is *well annotated*, $\mathrm{WellAnnotated}(m, \mathscr{A}, S_{init})$, if the following hold.

(1) Its precondition holds at the message view: $m.\mathsf{view}|_{\mathrm{Shared}(m.\mathsf{view}, \mathscr{A}, S_{init})} \models \mathscr{A}(m).\mathsf{mPre}$
(2) The precondition is intuitionistic (i.e., if it holds of a heap, it holds of all larger heaps).
(3) The precondition implies that the message's location is allocated:
$\models \mathscr{A}(m).\mathsf{mPre} \Rightarrow \exists v.\ m.\mathsf{loc} \mapsto v * \mathsf{true}$
(4) The precondition is stable under the message's rely: $\mathrm{stable}(\mathscr{A}(m).\mathsf{mPre}, \mathscr{A}(m).\mathsf{mRely})$
(5) The message effect satisfies the message's guarantee:
$\forall v.\ m.\mathsf{rval} \in \{\perp, v\} \implies (m.\mathsf{loc} \mapsto v \rightsquigarrow m.\mathsf{loc} \mapsto m.\mathsf{val} * A_{L2S} \mid A_{\mathrm{F}}) \in \mathscr{A}(m).\mathsf{mGuar}$
where A_{L2S} is the assertion precisely describing the heap $m.\mathsf{view}|_{\mathscr{A}(m).\mathsf{mL2S}}$ and
$A_{\mathrm{F}} \triangleq (\mathscr{A}(m).\mathsf{mPre} \circledast\!\!- m.\mathsf{loc} \mapsto v)$ is the frame (the remaining shared heap).
(6) the message only shares locations that it knows about (namely, that are initialized in its view):
$\mathscr{A}(m).\mathsf{mL2S} \subseteq \{m'.\mathsf{loc} \mid m' \in m.\mathsf{view}\}$

Here, we require the message precondition to be intuitionistic so that we can reason about preservation of $\mathscr{A}(m).\mathsf{mPre}$ when threads observe messages out of order, that is, when a thread observes a timestamp-wise larger message before a timestamp-wise smaller message $m_<$ to the same location. In that case, $m_<$ has no effect on the shared heap other than potentially transferring ownership. Thereby, as the value in $m_<.\mathsf{loc}$ is not set to $m_<.\mathsf{val}$, the effect of $m_<$ is not known to be in $\mathscr{A}(m_<).\mathsf{mGuar}$. Hence, we cannot prove via stability that the message precondition of a message m added by another thread continues to hold after adding $m_<$. On the other hand, $\mathscr{A}(m).\mathsf{mPre}$ being intuitionistic lets us prove that for any view $\mathscr{V}$ and set of shared locations S, if $\mathscr{V}|_S \models \mathscr{A}(m).\mathsf{mPre}$ then for any $S' \supseteq S$, $\mathscr{V}|_{S'} \models \mathscr{A}(m).\mathsf{mPre}$. In other words, validity is monotonic in the set of shared locations. Thus, $\mathscr{A}(m).\mathsf{mPre}$ does continue hold after adding $m_<$, because in doing so the set of shared locations can only increase and the shared heap remains unchanged on the previous set of shared locations.

5.3 Defining Safety Formally

We say that computation C is semantically correct or *safe* with respect to rely R, guarantee G, precondition P and postcondition Q, denoted as

$$R, G \vDash \{P\}\, C\, \{x.\ Q\}$$

if $\mathrm{safe}_n(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G, \lambda x.\ Q)$ holds for all $n \in \mathbb{N}$ and all well-formed logical configurations $(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G)$ such that $\mathscr{V}, L, S_{init} \models P$. We formalize the safety predicate safe_n for a logical program configuration and a postcondition Q below. The formalization follows the high-level

definition given in §5.1 with the only major addition being that we impose additional conditions on environment steps and program steps. These conditions are needed to preserve the well-formedness of the logical configuration under taking steps in the safety predicate.

Definition 5.1 (The RGSep Safety Predicate under RA).
$\text{safe}_0(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G, Q)$ holds always.
$\text{safe}_{n+1}(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G, Q)$ holds if and only if

(1) $\forall v \in \text{Val}.\ C = v \implies \mathscr{V}, L, S_{init} \models Q(v)$
(2) $\langle C, \mathscr{V}, M\rangle \nrightarrow \textbf{abort}$
(3) $\forall\, C', M_{\text{diff}}.$
$\langle C, \mathscr{V}, M\rangle \to \langle C', \mathscr{V} \uplus M_{\text{diff}}, M \uplus M_{\text{diff}}\rangle \implies$
$\exists\, L', \mathscr{A}'.$
  (a) $\forall m \in M_{\text{diff}}.\ m.\text{loc} \in L \cup \text{Shared}(m.\text{view}, \mathscr{A}, S_{init}) \vee m.\text{loc} \notin \text{Locs}(M)$
  (b) $dom(\mathscr{A}') = \{m \in M_{\text{diff}} \mid m.\text{loc} \in \text{Shared}(m.\text{view}, \mathscr{A}, S_{init})\}$
  (c) $L \uplus (\text{Locs}(M_{\text{diff}}) \setminus \text{Locs}(M)) = L' \uplus \bigcup_{m \in dom(\mathscr{A}')} \mathscr{A}'(m).\text{mL2S}$
  (d) $\forall m \in dom(\mathscr{A}').\ \text{WellAnnotated}(m, \mathscr{A} \cup \mathscr{A}', M, S_{init})$
  (e) $\forall m \in dom(\mathscr{A}').\ \{m' \in M \mid m'.\text{loc} \in \mathscr{A}'(m).\text{mL2S}\} \subseteq m.\text{view}$
  (f) $\forall m \in dom(\mathscr{A}').\ \forall m' \in dom(\mathscr{A}) \setminus m.\text{view}.\ \text{compat}(\mathscr{A}(m), \mathscr{A}'(m'))$
  (g) $\forall m \in dom(\mathscr{A}').\ R \subseteq \mathscr{A}'(m).\text{mRely}$
  (h) $\forall m \in dom(\mathscr{A}').\ \mathscr{A}'(m).\text{mGuar} \subseteq G$
  (i) $\text{safe}_n(C', M \uplus M_{\text{diff}}, \mathscr{V} \uplus M_{\text{diff}}, \mathscr{A} \cup \mathscr{A}', L', S_{init}, R, G, Q)$
(4) $\forall m, ann_m.$
  (a) $\nexists\, m' \in M.\ (m.\text{loc} = m'.\text{loc} \wedge m.\text{time} = m'.\text{time})$
  (b) $\bigcup_{m' \in m.\text{view}} m'.\text{view} \subseteq m.\text{view} \subseteq M$
  (c) $m.\text{loc} \notin L$
  (d) $ann_m \neq \bot \iff m.\text{loc} \in \text{Shared}(m.\text{view}, \mathscr{A}, S_{init})$
  (e) $ann_m \neq \bot \implies ann_m.\text{mL2S} \perp (L \cup S_{init} \cup \bigcup_{m' \in dom(\mathscr{A})} \mathscr{A}(m').\text{mL2S})$
  (f) $ann_m \neq \bot \implies \text{WellAnnotated}(m, \mathscr{A}[m \mapsto ann_m], M, S_{init})$
  (g) $ann_m \neq \bot \implies \{m' \in M \mid m'.\text{loc} \in ann_m.\text{mL2S}\} \subseteq m.\text{view}$
  (h) $ann_m \neq \bot \implies \forall m' \in M \setminus m.\text{view}.\ \text{compat}(\mathscr{A}(m'), ann_m)$
  (i) $ann_m \neq \bot \implies G \subseteq ann_m.\text{mRely}$
  (j) $ann_m \neq \bot \implies ann_m.\text{mGuar} \subseteq R$
  $\implies \text{safe}_n(C, M \cup \{m\}, \mathscr{V}, \mathscr{A}[m \mapsto ann_m], L, S_{init}, R, G, Q)$
(5) $\forall\, m \in M, C'.$
$m.\text{view} \subseteq \text{lookup}(C, \mathscr{V})$
$\wedge\ C' = \text{update}(C, m)$
$\implies \text{safe}_n(C', M, \mathscr{V} \cup \{m\}, \mathscr{A}, L, S_{init}, R, G, Q)$

The first two cases are straightforward: final configurations should satisfy the postcondition, and no configurations should ever abort.

Program steps. Case (3) shows the case in which the program takes a step. In this case, we assume we are given a configuration $(C, M, \mathscr{V}, L, S_{init}, R, G, \mathscr{A})$—the argument of the safety predicate—and a valid reduction step $\langle C, \mathscr{V}, M\rangle \to \langle C', \mathscr{V}', M'\rangle$ taken by the program. That reduction step can be any step allowed by the operational semantics other than a view increase (view increases are handled separately in Case (5)). These reduction steps only add messages to the message set which are immediately added to the view as well. Because of that we further specify the (possibly empty) set of messages M_{diff} that the program step adds. We need to show that after taking the step we can update our ghost state (i.e., choose instrumentation $L', \mathscr{A}'$, the new set of local locations

and message annotations for the newly added messages), such that the resulting configuration is well-formed (3a) - (3g), the action carried out by the program step is in the guarantee (3h) (implicitly uses (3f)) and the resulting configuration is safe for another n steps (3i).

Note that (3a) - (3g) do not exactly correspond to the well-formedness conditions given previously. Rather, they give a sufficient set of conditions which enable us to prove preservation of well-formedness when starting from a well-formed configuration. These conditions mean the following:

(a) A program step can only modify locations that are either local or have been shared prior, or allocate a new location.
(b) $\mathscr{A}'$ annotates newly added messages on shared locations.
(c) The new set of local locations should be the correct update of the old one with respect to ownership transfer and allocation. That is, freshly allocated locations should be added and all locations shared as a result of the executed program step should be removed from the set of local locations. As ownership transfer only goes from local to shared all elements of the set of freshly shared locations should have been formerly local.
(d) Any newly annotated messages should be well-annotated.
(e) If a newly added message shares another message's location, then the other message should be happens-before-ordered before the newly added message. This condition is needed to preserve well-formedness condition (9).
(f) The annotations of new messages should be compatible with those of existing messages not in their views. This condition is needed to preserve well-formedness condition (10).
(g) The rely of the newly added messages should contain the overall rely R, since the overall rely applies to all the threads of the program C.

Environment steps. Case (4) covers the case in which the environment modifies the state. The environment can do so by adding an arbitrary message to the message set. That message will not immediately be seen by the program threads and will thus not directly influence the observed state. However, the observed state will be affected later once a program thread increases its view. In any case, after the environment performs a modification of the state that is constrained by the rely, the resulting configuration should be safe for another n steps.

While the message added by the environment should be arbitrary, as should be its annotations ann_m, these objects should also be subject to some conditions to enforce well-formedness:

(a) With physical well-formedness we assume all messages on the same locations have different timestamps. To preserve this assumption the environment message must not have the same location *and* timestamp as another message in M.
(b) To preserve well-formedness of the physical configuration, the environment message should be transitively hb-ordered after all messages that are hb-ordered before any messages in its view, and should contain in its view only messages in the message set M.
(c) The location modified by the environment should not be local. The whole essence of a location being local is that it is only modified by the thread or program it is local to.
(d) The message added by the environment has annotations precisely if it modified a shared location. (Otherwise, it modified a location local to an environment thread, which by construction cannot interfere with the safety of C.)
(e) The environment message cannot share locations that the program owns (i.e., are part of L) or that are already shared by other messages (currently or later).
(f) The environment message is well-annotated.
(g) The environment message should preserve well-orderedness: If it shares a location, its view must include all messages to that location.

(h) The environment message annotation should be compatible with the annotations of existing messages not in its view.
(i) The environment message's rely should contain the guarantee of the program C. That means the environment threads, which are distinct from the program threads, should be aware of any possible actions carried out by the program threads.
(j) The environment message's guarantee should be included in the overall rely R of the program being verified. This condition, next to preserving well-formedness condition (11), implicitly constrains the message's corresponding action to be in the rely.

View Increases. Finally, case (5) concerns view increases.

To explain the conditions of this case, it is important to first clarify how we carry out view increases. Akin to our semantics only allowing steps to be taken by the left thread, we assume that only the left-most thread at the deepest level of nesting w.r.t. ParExec constructs can increase its view. That is, a thread C_t may only increase its view if it can be reached from the top-level computation by only following left threads and C_t does not contain any further ParExec constructs. (Normal parallel compositions do not matter here as they have not started executing and hence have not spawned any sub-threads.)

While the operational semantics given in §4 allow *any* left-most thread – regardless of its level of nesting – to increase its view, the restriction to the deepest level of nesting in the safety definition does not hurt us. As a property of our operational semantics, it is always a most deeply nested thread which will take the next step (which one is subject to swapping) or could be the cause of aborting. Hence, if the view increase only affects the program at a higher level of nesting, we do not have to perform the view increase right away. It is sufficient to only perform that view increase once that thread *becomes* the most deeply nested one.

Note also that it is only due to our swapping semantics that we can uniquely define where to perform the view increase among the most deeply nested threads. This simplifies proving properties about view increases and does not come with any losses as swapping allows us to change which thread we see as the left-most one.

Now that we have clarified which threads can increase their view, we need to discuss how to stop view increases from affecting well-formedness. Recall that for a physical configuration to be well-formed we require the union of the thread views to be a subset of the overall view of the parent thread. This property could be broken if we naively increase some thread view without considering the parent threads. Therefore, to maintain well-formedness, we do the following. Whenever we increase the view of a thread, we always recursively increase the outer views of all ParExec constructs the computation of the thread is nested inside. We visualize the result of a view increase and re-establishing well-formedness in the following example, where the thread executing C_5 increases its view by a message m:

$$\begin{aligned}
&\langle\left(\left(C_5{}^{\mathscr{V}_5}\|_b^{\mathscr{V}_6}C_6\right)^{\mathscr{V}_4}\|_b^{\mathscr{V}_3}C_3\right)^{\mathscr{V}_1}\|_b^{\mathscr{V}_2}C_2, M, \mathscr{V}\rangle \\
\xrightarrow{\text{increase thread view}}\ &\langle\left(\left(C_5{}^{\mathscr{V}_5\cup\{m\}}\|_b^{\mathscr{V}_6}C_6\right)^{\mathscr{V}_4}\|_b^{\mathscr{V}_3}C_3\right)^{\mathscr{V}_1}\|_b^{\mathscr{V}_2}C_2, M, \mathscr{V}\rangle \\
\xrightarrow[\text{outer views}]{\text{recursively increase}}\ &\langle\left(\left(C_5{}^{\mathscr{V}_5\cup\{m\}}\|_b^{\mathscr{V}_6}C_6\right)^{\mathscr{V}_4\cup\{m\}}\|_b^{\mathscr{V}_3}C_3\right)^{\mathscr{V}_1\cup\{m\}}\|_b^{\mathscr{V}_2}C_2, M, \mathscr{V}\cup\{m\}\rangle
\end{aligned}$$

The view increase and re-establishment of well-formedness of the program we express in the safety definition via the "update" function. The "update" function takes care of updating all thread views within the program, such that finally only the overall view $\mathscr{V}$ has to be increased.

Our safety definition further places a restriction on which messages can be incorporated into the view. That is because the happens-before order needs to be obeyed during view increases. Two

messages that are happens-before ordered should be incorporated into the view in that same order. For that reason we increase the view by one message $m_{inc} \in M$ at a time. We enforce that all messages in the view of m_{inc} (which are the messages happens-before ordered before m_{inc}) must have already been observed before m_{inc} is added to the view. As we perform the view increase at the left-most thread at the deepest level of nesting, we have to check whether exactly that thread's view (returned by the "lookup" function) contains m_{inc}'s view. Because the physical configuration is well-formed, this will then imply that all views m_{inc} is propagated to also contain m_{inc}'s view. Thus, we successfully preserve happens-before ordering on all levels of nesting.

The reader may note that we do not restrict the message added to the view to be unseen by the overall view. The reason for that lies in the operational semantics for termination of ParExec constructs. We need to be able to increase both thread views until they share the same view. Hence we need to be able to increase thread views with messages already seen by the overall view, so one thread can catch up with the other.

5.4 Adequacy

Since our semantic definition of RGSep judgments under RA is long-winded, we want to assure the reader that it actually implies something useful about program executions.

To do so, we need an auxiliary definition. A set of messages M *satisfies* an RGSep assertion P iff there exist disjoint heaps h_L, h_S such that $h_L \uplus h_S = M|_{\text{Addr}}$ (the projection of M to a heap) and $h_L, h_S \models P$,

We then prove the following adequacy theorem, which states that if a semantically correct program is executed from an initial state satisfying its precondition, then (1) it does not abort and (2) its final state upon termination satisfies the postcondition.

Theorem 5.2 (Adequacy). *Assume that $R, G \vDash \{P\}\, C\, \{x.\, Q\}$ holds. Let M be a set of messages satisfying P, in which all initialization messages happen before all other messages, and $\langle C', \mathscr{V}', M'\rangle$ be a configuration such that $\langle C, M, M\rangle \to^* \langle C', \mathscr{V}', M'\rangle$. Then:*

*(1) $\langle C', \mathscr{V}', M'\rangle \not\to$ **abort**, and*
(2) if $C' = v$ for some value $v \in Val$, then $\mathscr{V}'$ satisfies $Q[v/x]$.

Proof. We split the proof of this result into two parts. First, we prove that given $\mathscr{V}, L, S_{init}, \mathscr{A}$ such that $\langle C, \mathscr{V}, M\rangle \to^n \langle C', \mathscr{V}', M'\rangle$ and $\text{safe}_{n+1}(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G, Q)$, the conclusion of Theorem 5.2 holds. Second, we will prove the existence of such $\mathscr{V}, L, S_{init}$ and $\mathscr{A}$.

We prove the first part by induction on the number n.

- *Case $n = 0$.* We know $C' = C$ and $M' = M$ and $\mathscr{V}' = \mathscr{V}$. Hence by case 2 of the safety predicate, $\langle C, \mathscr{V}', M'\rangle$ does not abort and by case 1 of the safety predicate, $\mathscr{V}, L, S_{init} \models Q$, which implies $\mathscr{V}'$ satisfies Q as required.
- *Inductive step.* Fix some $d \in \mathbb{N}$ and let the statement hold for $n = d$. Let $n = d + 1$. Then, there exist $C_1, \mathscr{V}_1, M_1$ such that $\langle C, \mathscr{V}, M\rangle \to \langle C_1, \mathscr{V}_1, M_1\rangle \to^d \langle C', \mathscr{V}', M'\rangle$. Hence, by case 3 of the safety predicate, we can choose new instrumentation $L', \mathscr{A}'$ such that $\text{safe}_{n_s}(C_1, M_1, \mathscr{V}_1, \mathscr{A} \cup \mathscr{A}', L', S_{init}, R, G, Q)$ and so we obtain the result by the inductive hypothesis.

This concludes the proof of the first part.

For the second part, since M satisfies P, we know that there exist heaps h_L, h_S such that $h_L \uplus h_S = M|_{\text{Addr}}$ and $h_L, h_S \models P$. Letting $L = dom(h_L)$ and $S_{init} = dom(h_S)$, we clearly have $M, L, S_{init} \models P$. For a message $m \in M$, define:

$$G_m \triangleq \begin{cases} \{(m.\text{loc} \mapsto v \rightsquigarrow m.\text{loc} \mapsto m.\text{val}) \mid v \in \mathbb{N}\} & \text{if } m \text{ is a write} \\ \{(m.\text{loc} \mapsto m.\text{rval} \rightsquigarrow m.\text{loc} \mapsto m.\text{val})\} & \text{if } m \text{ is an RMW} \end{cases}$$

Further, define $\mathscr{A}$ with $dom(\mathscr{A}) = \{m \in M \mid m.\mathsf{loc} \in S_{init} \wedge \neg \mathsf{InitMsg}(m)\}$ such that $\mathscr{A}(m)$ is equal to:

$$\left(\mathsf{mPre} := \exists v.\, m.\mathsf{loc} \mapsto v * \mathsf{true},\ \mathsf{mRely} := \bigcup_{m' \in dom(\mathscr{A})} G_{m'},\ \mathsf{mGuar} := G_m,\ \mathsf{mL2S} := \emptyset\right).$$

By noting that $\mathsf{Shared}(\mathscr{V}', \mathscr{A}, S_{init}) = S_{init}$ regardless of $\mathscr{V}'$ and also $L \cup S_{init} = dom(h_L \uplus h_S) = \{m.\mathsf{loc} \mid m \in M\}$, it is easy to check that the logical configuration $(C, M, M, \mathscr{A}, L, S_{init}, R, G)$ is well-formed. Applying the definition of $R, G \vDash \{P\}\, C\, \{Q\}$ completes the proof. □

6 Proving RGSep Sound for Release-Acquire Semantics

As is standard, the soundness theorem states that whenever we can derive the judgment $R, G \vdash \{P\}\, C\, \{x.\ Q\}$ for rely R, guarantee G, computation C and pre- and postconditions P and Q using the rules of §2.4, then $R, G \vDash \{P\}\, C\, \{x.\ Q\}$ holds according to the semantic interpretation of §5.3.

Theorem 6.1 (Soundness). *If $R, G \vdash \{P\}\, C\, \{x.\ Q\}$, then $R, G \vDash \{P\}\, C\, \{x.\ Q\}$.*

We prove soundness by individually proving soundness of each proof rule given in §2.4. For most of the proof rules, we can prove $R, G \vDash \{P\}\, C\, \{x.\ Q\}$ with a fairly standard induction on the number of steps n for which the safety predicate holds.

There are, however, two rules we cannot simply prove by direct induction on the number of steps. These rules are the frame rule and the parallel composition rule. Of these two rules, the soundness of the frame rule can actually be shown to be a consequence of the soundness of the parallel composition rule and the value rule, by observing that a computation C is observationally equivalent to $C \parallel 0$.

To establish the soundness of the parallel composition rule, we reason about the intermediate ParExec constructs. We do so in two steps. First, we show parallel compositions are safe if they are still safe after taking a single program step. More precisely, we prove that safety of parallel compositions $C_1 \parallel C_2$ under view $\mathscr{V}$ follows from safety of the ParExec construct ${C_1}^{\mathscr{V}} \|_b^{\mathscr{V}} C_2$, which the parallel composition steps into via our operational semantics. That part of our proof is relatively simple, as this program step does not alter the state in any way.

The second and more involved part of the proof is then to show the safety of the resulting ParExec construct. For that purpose, we set up an inductive proof showing that if both threads are safe with respect to their individual views, then the corresponding ParExec construct is safe as well. The inductive proof leverages our ghost state and heavily relies on the assumption of physical and logical well-formedness. An important realization that dramatically simplified the induction is that well-formedness is preserved under taking steps in the safety predicate. In fact, the latter realization simplified the high-level proof to the point that it is not worth discussing here.

What turned out to be the most interesting part of the soundness proof was reasoning about environment steps in the form of unseen messages. The challenge was that while we restrict the action carried out by the environment to be in the environment thread's guarantee (and hence in the rely of the program being verified), it is non-trivial to show the corresponding environment message results in the same action when it is observed later. If it would not result in the action tracked by the rely, then the state would be modified in an unexpected way and we cannot reason about safety being preserved under such unexpected interference. Thus, it is crucial to show that the effect of observing an environment message will always satisfy the corresponding action in the rely, no matter when it is observed (as long as messages are observed in happens before order). The intuition why this property should hold is that the frame of the action we track in the thread guarantee is implied by the message precondition. We require the message precondition to be stable under the thread rely, whereby the message precondition should be preserved under adding

messages from other threads to the view. Hence the action frame, which held when the environment message was created) should still be satisfied when the environment message is finally observed.

The idea described above begs us to prove the following lemma, which requires a non-standard inductive proof.

Lemma 6.2. *Let $(C, M, \mathscr{V}, \mathscr{A}, L, S_{init}, R, G)$ be a well-formed logical configuration, let $m \in M \setminus \mathscr{V}$ be some message such that $m.view \subseteq \mathscr{V}$, and let P be a RGSep assertion that is stable under R. If $\mathscr{V}, L, S_{init} \models P$, then $\mathscr{V} \cup \{m\}, L, S_{init} \models P$.*

7 Related Work

As we have discussed RGSep [37] extensively throughout the paper, we will discuss some of its extensions for SC. Deny-guarantee [9] modifies rely-guarantee to support dynamical scoping of threads via fork and join operations. For that purpose, deny-guarantee inverts the 'rely' from rely-guarantee into a 'deny', expressing what the environment must *not* do, and introduces a permission structure for describing interference akin to separation logic permissions [4]. Thus, the interference through both deny and guarantee can be dynamically split and joined with the separating conjunction. Local rely-guarantee (LRG) [11] and CoLoSL [29] also introduce a notion of separating conjunction for rely/guarantee conditions, but rather for the purpose of locality. Their objective is to perform rely-guarantee reasoning more locally in cases where some state is only shared between a subset of the threads and hence need not be seen by the full environment. Since these RGSep extensions are orthogonal to our work, it would be interesting to incorporate them into our adaptation for RA in the future.

A further extension of deny-guarantee is CAP [8], which introduces so-called concurrent abstract predicates enabling reasoning about distinct parts of a shared data structure as if they were disjoint through resource permissions and stability checks. CAP is more modular than RGSep, but in exchange often requires more complex proofs. As in RGSep, CAP requires shared regions to only be accessed through a set of primitive atomic operations. This weakness is addressed by logically atomic triples in logics like TaDA [5]. A logically atomic triple expresses that a program behaves *as if* it were executing atomically with respect to some level of abstraction, a property also known as linearizability [13].

A program logic that has become widely popular in the last decade is Iris [15], which is strong enough to verify higher order imperative programs and is fully mechanized in Rocq. Iris is based on partial commutative monoids and invariants and has grown to contain numerous advanced concepts from view shifts to all sorts of modalities. Moreover, the Iris framework lets one instantiate any logic operating under SC and obtain a set of reasoning principles and rules for free.

There has also been quite some work on designing program logics for WMC directly. For example, Lahav and Vafeiadis [22] and Dalvandi et al. [6] develop two Owicki-Gries proof systems for RA. In terms of separation logics, RSL [36] was the first program logic to operate under a non-trivial fragment of the C11 relaxed memory model. The logic proposed reasoning about release writes and acquire reads through assertions that express ownership and permissions to execute these operations. FSL [10] then extended RSL with two modal operators for reasoning about C11 memory fences. RSL and FSL's reasoning power is akin to that of CSL: they use invariants and ownership transfer to reason about shared memory, and cannot directly describe how threads interfere. As such, they are only suitable for programs with coarse-grained synchronization. Nevertheless, their soundness proofs were highly non-trivial, in part because they were performed over the axiomatic C11 semantics.

GPS [32] subsequently enriched reasoning about WMC with more advanced verification techniques, mainly ghost state and per-location protocols, which enable a restricted form of rely/guarantee

reasoning. Nevertheless, even though one can use GPS directly to verify fine-grained concurrent programs under WMC, GPS proofs require considerable ingenuity to express shared state invariants and differ widely from the corresponding proofs in SC logics. Consequently, GPS—and its subsequent integrations into the Iris framework, namely iGPS [16] and iRC11 [7]—necessitate having to *reprove* correctness of programs operating under SC. Moreover, these program logics for WMC have more complicated proof rules than similar logics for SC, which results in considerable additional proof burden. The same holds for subsequent logics, such as AxSL [12] and SLR [31], which enable reasoning under even weaker memory consistency models, the Arm [28] and Promising semantics [17], respectively.

Robustness [2, 21] is a different approach to verifying programs for weak memory models by reusing proofs from SC. A program is *robust* under RA if all its behaviors under RA are also observable under SC. The idea of robustness is that if we can show that a program is robust under RA, then it suffices to verify the program using a logic designed for SC. While this proof method also lets us reason under RA by reusing the proofs from logics for SC, our approach does so with less effort as well as under less restrictions. Firstly, our approach eliminates the need to show robustness and lets us apply the proofs from SC directly and thus takes less effort. Secondly, our approach is less restrictive as it can be applied to all programs under RA, even if they do not only emit SC behaviors. For example, we can establish a trivial postcondition for the non-robust IRIW program from the introduction.

8 Discussion and Conclusions

We conclude this paper by discussing the two main limitations of our proof.

The first limitation is that our CAS rule supports ownership transfer from the local region to the shared region but not in the other direction: from the shared memory region to the local region. This lack of support for 'localizing' shared state simplifies our soundness proof but prevents us from verifying programs that deallocate shared data structures, like the lock-coupling list example of the original RGSep paper [37], and deriving the standard CSL specification for acquiring a lock, which gains ownership of the resource invariant.

The second limitation is that we do not support any rules for introducing auxiliary variables and/or reasoning about general atomic blocks that contain memory accesses at multiple memory locations. In part, this limitation is quite natural, since existing RA semantics do not support such general atomic blocks. It does, however, prevent us from establishing linearizability of concurrent algorithms.

In the future, we plan to address these two limitations. In addition, we would like to extend RGSep with additional proof rules for the weaker features of the RC11 memory model, such as relaxed accesses and memory fences, by incorporating the triangle modalities of FSL [10].

Data Availability Statement

The artifact provided as supplementary material to this paper contains the Rocq formalization of the definitions and proofs mentioned in this paper.

Acknowledgments

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

References

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL 2011* (Austin, Texas, USA). ACM, New York, NY, USA, 55–66. doi:10.1145/1926385.1926394

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

[3] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *POPL 2017*. ACM, New York, NY, USA, 626–638. doi:10.1145/3009837.3009888

[4] John Boyland. 2003. Checking interference with fractional permissions. In *SAS 2003*. Springer, Berlin, Heidelberg, 55–72. doi:10.1007/3-540-44898-5_4

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP 2014 (LNCS, Vol. 8586)*. Springer, Heidelberg, 207–231. doi:10.1007/978-3-662-44202-9_9

[6] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-Gries reasoning for C11 RAR. In *ECOOP 2020 (LIPIcs, Vol. 166)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:26. doi:10.4230/LIPIcs.ECOOP.2020.11

[7] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4 (12 2019), 1–29. doi:10.1145/3371102

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

[9] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP 2009* (York, UK). Springer, Berlin, Heidelberg, 363–377. doi:10.1007/978-3-642-00590-9_26

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

[11] Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL 2009*. ACM, New York, NY, USA, 315–327. doi:10.1145/1480881.1480922

[12] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An axiomatic basis for computer programming on the relaxed Arm-A architecture: The AxSL logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 604–637. doi:10.1145/3632863

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

[14] Cliff Jones. 1983. Specification and design of (parallel) programs. *Proc. IFIP Congress* 83, 321–332.

[15] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Functional Programming* 28 (2018), e20. doi:10.1017/S0956796818000151

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

[17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017* (Paris, France). ACM, New York, NY, USA, 175–189. doi:10.1145/3009837.3009850

[18] Ori Lahav. 2019. Verification under causally consistent shared memory. *ACM SIGLOG News* 6, 2 (April 2019), 43–56. doi:10.1145/3326938.3326942

[19] Ori Lahav and Udi Boker. 2022. What's decidable about causally consistent shared memory? *ACM Trans. Program. Lang. Syst.* 44, 2 (2022), 8:1–8:55. doi:10.1145/3505273

[20] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL 2016* (St. Petersburg, FL, USA). ACM, New York, NY, USA, 649–662. doi:10.1145/2837614.2837643

[21] Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *PLDI 2019* (Phoenix, AZ, USA). ACM, New York, NY, USA, 126–141. doi:10.1145/3314221.3314604

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

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

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

[25] Jeremy Manson, William W. Pugh, and Sarita V. Adve. 2005. The Java memory model. (2005), 378–391. doi:10.1145/1040305.1040336

[26] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. doi:10.1016/J.TCS.2006.12.035

[27] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *TPHOLs 2009* (Munich, Germany). Springer, Berlin, Heidelberg, 391–407. doi:10.1007/978-3-642-03359-9_27
[28] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. doi:10.1145/3158107
[29] Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent local subjective logic. In *ESOP 2015 (LNCS, Vol. 9032)*, Jan Vitek (Ed.). Springer, Heidelberg, 710–735. doi:10.1007/978-3-662-46669-8_29
[30] SPARC International Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall.
[31] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP 2018 (LNCS, Vol. 10801)*. Springer, Heidelberg, 357–384. doi:10.1007/978-3-319-89884-1_13
[32] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA 2014*. ACM, New York, NY, USA, 691–707. doi:10.1145/2660193.2660243
[33] Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Technical Report UCAM-CL-TR-726. University of Cambridge, Computer Laboratory. doi:10.48456/tr-726
[34] Viktor Vafeiadis. 2010. RGSep action inference. In *VMCAI 2010 (LNCS, Vol. 5944)*. Springer, Heidelberg, 345–361. doi:10.1007/978-3-642-11319-2_25
[35] Viktor Vafeiadis. 2011. Concurrent separation logic and operational semantics. *Elect. Not. Theor. Comput. Sci.* 276 (2011), 335–351. doi:10.1016/J.ENTCS.2011.09.029
[36] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA 2013*. ACM, New York, NY, USA, 867–884. doi:10.1145/2509136.2509532
[37] Viktor Vafeiadis and Matthew Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 (LNCS, Vol. 4703)*. Springer, Heidelberg, 256–271.

Received 2026-03-22; accepted 2026-08-06