IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 10, 2020, accepted August 31, 2020, date of publication September 18, 2020, date of current version October 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3024681

A Program Logic for Reasoning About C11 Programs With Release-Sequences

MENGDA HE[1], SHENGCHAO QIN[1,2], (Senior Member, IEEE), AND ZHIWU XU[2]
[1]School of Computing, Engineering and Digital Technologies, Teesside University, Middlesbrough TS1 3BX, U.K.
[2]College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China

Corresponding author: Shengchao Qin (s.qin@tees.ac.uk)

This work was supported by the National Natural Science Foundation of China under Grant 61772347, Grant 61972260, and Grant 61836005.

ABSTRACT With the popularity of weak/relaxed memory models widely used in modern hardware architectures, the C11 standard introduced a language level weak memory model, A.K.A the C11 memory model, that allows C/C++ programs to exploit the optimisation provided by the hardware platform in memory ordering and gain benefits in efficiency. On the other hand, with the weakened memory ordering allowed, more program behaviours are introduced, among which some are counterintuitive and make it even more challenging for programmers to understand or to formally reason about C11 multithread programs. To support the formal verification of the C11 weak memory programs, several program logics, e.g. RSL, GPS, FSL, and GPS+, have been developed during the last few years. However, due to the complexity of the weakened memory model, some intricate C11 features still cannot be handled in these logics. A notable example is the lack of supporting to the reasoning about a highly flexible C11 synchronisation mechanism, the release-sequence. Recently, the FSL++ logic proposed by Doko and Vafeiadis moves one step forward to address this problem, but FSL++ only considers the scenarios with atomic update operations in a release-sequence. In this article, we propose a new program logic, GPS++, that supports the reasoning about C11 programs with fully featured release-sequences. We also introduce fractional read permissions to GPS++, which are essential to the reasoning about a large number of real-world concurrent programs. GPS++ is a successor of our previous program logic GPS+, but it comes with much finer control over the resource transmission with the newly introduced restricted-shareable assertions and an enhanced protocol system. A more sophisticated resource model is devised to support the soundness proof of our new program logic. We also demonstrate GPS++ in action by verifying C11 programs with release-sequences that could not be handled by existing program logics.

INDEX TERMS Program logic, formal semantics, program verification, C11 weak memory model, release-sequence.

I. INTRODUCTION

To discuss the behaviours of shared-memory concurrent programs, a memory model must be assumed, as it fundamentally defines how the threads communicate with each other. The traditional strong memory model, i.e., the sequential-consistency (SC) [1] model, assumes a single global memory that is accessed by all threads in an interleaving manner while the instructions in each thread are executed strictly following their program orders. However, this model is abandoned by most of the modern hardware architectures, as it leaves very little room for optimisation. Modern hardware architectures embrace more relaxed memory models, which allow memory accessing operations to be reordered and threads to have their own observations about the memory states. For instance, the memory model used for the x86 architecture is the total-store-order (TSO) model instead of the SC model, as with write buffers facilitated, the x86 architecture allows some store operations to be reordered after the following load operations as long as a total order for all store operations is preserved. Other platforms like ARM and PowerPC adopt even weaker memory models.

With the various levels of memory weakening allowed by different hardware platforms, a unified interface is essential

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

TABLE 1. Program Logics' Support to C11 Synchronisations.

Synchronisation Paradigms	RSL	GPS	FSL	GPS+	FSL++	GPS++
With Release Store & Acquire Load Pairs	✓	✓	✓	✓	✓	✓
With Fences			✓	✓	✓	✓
With Release-sequences with only Atomic Updates					✓	✓
With Fully Featured Release-sequences						✓

to help programmers to compose programs that have best performance regardless the underlying platforms. Therefore, in the C11 standard [2], [3], a language-level weak memory model, i.e., the C11 memory model, is introduced and has later been formalised by Batty *et al.* [4]. However, it remains challenging to understand or to reason about the counter-intuitive behaviours introduced by this weakened memory model. Several program logics (RSL [5], GPS [6], [7] FSL [8], GPS+ [9], [10], and FSL++ [11]) focusing on C11 programs have been proposed during the last few years, but the reasoning about some highly intricate features of the C11 memory model is still not supported. A notable example is that the support to a highly flexible C11 synchronisation mechanism, i.e., the *release-sequences*, is usually left out. As shown in Table 1, there are four commonly used paradigms to establish C11 synchronisations. From the most straightforward way that only uses release write and acquire read pairs to the ones with fully featured release-sequences involved, the complexity increases along with its flexibility, as more program commands' interactions need to be taken under consideration. The formal reasoning about the most flexible paradigm facilitating with fully featured release-sequences is not supported until this work.

Beside the memory optimisations, another approach to maximising the benefits of multiprocessing is to reduce the fractions of tasks that could not be parallelised. For instance, instead of enforcing all the accesses to a shared resource to be mutually exclusive, a readers-writer-lock allows multiple readers (or a single writer) to exist at a time. The readers-writer-lock is widely adopted by real-world programs (e.g. the Linux kernel, pthread library, etc.). However, to formally verify a sophisticated algorithm like the readers-writer-lock, fractional permissions [12], [13] will be a necessary ingredient, which is not yet supported in the GPS family.

In this article, we propose GPS++, a program logic that supports the reasoning about C11 programs with fully featured release-sequences. To achieve this, our logic is facilitated with an enhanced per-location protocol system, a new type of assertions, i.e., the restricted shareable assertion, and a set of new reasoning rules to deal with C11 release-sequences related operations. We also introduce the support to fractional permissions, which enables us to use the proposed logic against sophisticated real-world concurrent algorithms such as the readers-writer-lock.

This article extends the earlier conference version [14] significantly with the following additional contributions: 1. The support for fractional read permissions in the new logic, that allows it to reason about real-world concurrent algorithms that otherwise would not be possible; 2. The proof in the new logic of the correctness of a non-trivial concurrent program, i.e. readers-write-lock (which cannot be handled by previous logics in the GPS family) that illustrates the applicability of GPS++; 3. A more sophisticated resource-map-based instrumented semantic model that help characterise the subtle new features in GPS++; 4. The formulation of the soundness of the GPS++ logic, together with proof sketches for lemmas and theorems.

$$
\begin{array}{llll}
\textit{Val} & v & ::= & x \mid V \text{ where } V \in \mathbb{N} \\
\textit{Exp} & e & ::= & v \mid v+v \mid v-v \mid v==v \mid v \,\texttt{mod}\, v \\
 & & & \texttt{let}\, x = e \,\texttt{in}\, e \mid \texttt{repeat}\, e \,\texttt{end} \mid \\
 & & & \texttt{if}\, v \,\texttt{then}\, e \,\texttt{else}\, e \mid \texttt{fork}\, e \mid \texttt{alloc}(V) \\
 & & & [v]_O \mid [v]_O := v \mid \texttt{CAS}_{O,O}(v,v,v) \mid \texttt{fence}_O \\
\textit{MO} & O & ::= & \texttt{rel} \mid \texttt{acq} \mid \texttt{rlx} \mid \texttt{na} \\
\textit{EvalCtx} & K & ::= & [\,] \mid \texttt{let}\, x = K \,\texttt{in}\, e
\end{array}
$$

FIGURE 1. A language for C11 concurrency.

In the rest of the article, we first introduce our core language that captures most essential C11 features and the C11 memory model in §II. Then we discuss this work's foundation, the GPS+ logic, in §III, before presenting our new logic in §IV. We demonstrate the power of our new program logic by using it to verify example programs including a variant of the readers-writer-lock in §V. After that, we present the formal foundation of GPS++ in Section VI and illustrate its soundness in §VII. Finally, related work is discussed in §VIII and the article is concluded in §IX.

II. THE LANGUAGE AND THE MEMORY MODEL

A. THE LANGUAGE

We use the expression-oriented language presented in Fig. 1 as our core language. It is used to capture the essential features of the C11 memory model. With the support to atomic read/write (load/store), fences, and compare-and-swap (CAS) our language can express C11 programs using various kinds of inter-thread synchronisation mechanisms, including the powerful release-sequence.

Our language has variable names (represented by metavariable x) and integer values (represented by metavariable V) as values (*Val*). The pointer arithmetic, `let`-binding, loop command `repeat` e, conditional statement `if...then...else`, thread forking `fork` e, memory allocation `alloc`(V), memory load (read) $[v]_O$ and store (write) $[v]_O := v$, atomic update operation $\texttt{CAS}_{O,O}(v, v, v)$, and fence operations $\texttt{fence}_O$ are supported as our expressions (e). Specifically, in the loop command `repeat` e the loop body e will be repeatedly executed until a non-zero value is returned.

Note that a memory order O needs to be specified for some expressions indicating which degree of memory relaxation can be applied to the annotated operation. Following the C11 language, we require a memory location is either atomic or non-atomic and this cannot be changed once defined. For an atomic location v_1, the memory order O used in a load operation $[v_1]_O$ can be acquire (`acq`) or relaxed (`rlx`); the memory order O in a store operation $[v_1]_O := v$ can be either release (`rel`) or relaxed (`rlx`). Meanwhile, memory accesses to non-atomic locations can only be annotated as non-atomic (`na`). The compare-and-swap expression $\texttt{CAS}_{O_1,O_2}(v_1, v_2, v_3)$ requires v_1 to be the address of an atomic location and performs the following steps in a single atomic move: firstly, the value of v_1 is loaded with the memory order O_1 (which can be either `acq` or `rlx`), then the value is used to "compare" with the expected value v_2; if they are the same, the value v_3 is stored to location v_1 with the memory order O_2 (which can be `rel` or `rlx`) (the "swapping") and a numerical value 1 is returned indicating its success; otherwise 0 is returned indicating the failure of the compare-and-swap process.

B. THE GRAPH SEMANTICS

The C11 memory model resides at language-level aiming at abstracting away the differences of underlying hardware memory models. Therefore, it is not straightforward to express it in an operational manner. The axiomatic approach is often used instead to formalise the C11 memory model, where execution graphs are used to represent candidate executions (with the program actions/events as vertices and their relations represented by the edges) and a set of axioms decide if an execution is legal or not. This is the approach adopted by Batty *et al.* [4] to give the first formalisation of the C11 memory model. The memory models used in the C11 program logics [5], [6], [8]–[11] are defined in a similar manner but with certain simplifications, e.g., only accepting synchronisations created using a simple way without release-sequences involved. This work follows the graph based axiomatic semantics, with extra information added for threads (highlighted in Fig. 2), and supports synchronisations with fully featured release-sequences.

As shown in Fig. 2, the event graph $\mathcal{G}$ $(A, T, \mathsf{sb}, \mathsf{mo}, \mathsf{rf})$ concerns a set of events, and records their action type information in A (the action types will be further discussed shortly), their thread identities in T (i.e. to which threads they belong), as well as their relations in sb, mo, and rf. The *sequenced-before* relation (sb) represents the non-transitive program order. All store operations accessing a same location form a strict-total order, which we record in the *modification-order* (mo). When a load operation reads from a store operation, this relation is tracked in the *reads-from* map (rf). We formalise the thread pool as $\mathcal{T}$ which tracks each numerically indexed thread's last event and the expressions to be executed.

As shown in Fig. 3 and Fig. 4, we adopt a two-layer semantics, namely, event-step and machine-step, following GPS and GPS+. An event-step ($e \xrightarrow{\alpha} e'$) is the execution of e, resulting

$$
\begin{array}{llll}
\textit{Action} & \alpha & ::= & \mathbb{S} \mid \mathbb{A}(\ell..\ell') \mid \mathbb{W}(\ell, V, O) \mid \mathbb{R}(\ell, V, O) \mid \\
& & & \mathbb{U}(\ell, V, V, O, O) \mid \mathbb{F}(O) \\
\textit{EventName} & a & & \text{(from an infinite set)} \\
\textit{ActMap} & A & \in & \textit{EventName} \xrightarrow{fin} \textit{Action} \\
\textit{ThreadID} & T & \in & \textit{EventName} \xrightarrow{fin} \mathbb{N} \\
\textit{Graph} & \mathcal{G} & ::= & (A, T, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \text{ where} \\
\multicolumn{4}{r}{\mathsf{sb}, \mathsf{mo} \subseteq \mathsf{dom}(A) \times \mathsf{dom}(A), \mathsf{rf} \in \mathsf{dom}(A) \rightharpoonup \mathsf{dom}(A)} \\
\textit{ThreadMap} & \mathcal{T} & \in & \mathbb{N} \xrightarrow{fin} (\textit{EventName} \times \textit{Exp})
\end{array}
$$

FIGURE 2. Syntax of event graph.

in a return value or a remainder expression (e') and an action (α) to be added to the execution graph in the machine-steps. The executions of arithmetic, let-banding, repeat, and conditional expressions returns different values or remainder expressions but they only generates skip actions ($\mathbb{S}$) as they do not involve memory accesses. The allocation expression $\texttt{alloc}(n)$ generates an allocation action $\mathbb{A}(\ell..\ell + n - 1)$, which indicates n fresh memory location ranging from ℓ to $\ell + n - 1$ are allocated and ℓ is returned. Store and fence expresses generate corresponding write ($\mathbb{W}$) and fence ($\mathbb{F}$) actions with specified memory orders; and we let them always return 0 as they should not be used to change the program control flow. Note that the event-step rule for read action $\mathbb{R}$ only specifies that it should return some numerical value V. The actual value can be read is constrained by the memory model axioms (`consistentC11`) in the machine-steps (which will be discussed shortly in this section) with the global execution taken under consideration. There are two rules for CAS expressions to correspondingly capture the successful and failure cases. In the case of success, an update action $\mathbb{U}(\ell, V_o, V_n, O_r, O_w)$ is generated, where the current value V_o stored at location ℓ is required to be the same as specified in the expression; otherwise, the CAS should be considered as failed and is treated as an atomic read action which reads some value other than V_o.

We use the machine configuration $\langle \mathcal{T}; \mathcal{G} \rangle$ to represent the execution states, where the thread pool $\mathcal{T}$ contains the expressions to be executed in each thread and the execution graph $\mathcal{G}$ is a record of the execution history. Machine-step rules are used to update the machine configurations. The first rule states that an arbitrary thread from $\mathcal{T}$ can take a move ($e \xrightarrow{\alpha} e'$) and generate a new machine configuration ($\langle \mathcal{T}'; \mathcal{G}' \rangle$) based on the current one ($\langle \mathcal{T}; \mathcal{G} \rangle$) with the new event ($a'$) and the corresponding relations added to the event graph given that the C11 memory model axioms are preserved in the extended graph ($\mathsf{consistentC11}(\mathcal{G}')$). More specifically, assume the thread i in the thread pool $\mathcal{T}$ is chosen to execute, its last event is a and the expression to be executed is e. Then e is reduced to e' following the corresponding event-step semantics rule, yielding a new action α with a new event name a'. We update i in the thread pool with this information (a', e'), then add the newly generated event to the event graph as the following, yielding a new graph $\mathcal{G}'$. Firstly, the mapping $a' \mapsto \alpha$ and $a' \mapsto i$ are added to the action map ($\mathcal{G}'.A$) and the thread map ($\mathcal{G}'.T$), respectively. This information will be crucial for reasoning about programs with C11

$$
\begin{array}{rcll}
n+m & \xrightarrow{\mathbb{S}} & k & k=n+m \\
n-m & \xrightarrow{\mathbb{S}} & k & k=n-m \text{ if } n \geq m \\
n-m & \xrightarrow{\mathbb{S}} & 0 & \text{if } n<m \\
n==m & \xrightarrow{\mathbb{S}} & 1 & n=m \\
n==m & \xrightarrow{\mathbb{S}} & 0 & n \neq m \\
\texttt{let}\, x=V\, \texttt{in}\, e & \xrightarrow{\mathbb{S}} & e[V/x] & \\
\texttt{repeat}\, e\, \texttt{end} & \xrightarrow{\mathbb{S}} & \texttt{let}\, x=e\, \texttt{in}\, \texttt{if}\, x\, \texttt{then}\, x\, \texttt{else}\, \texttt{repeat}\, e\, \texttt{end} & \\
\texttt{if}\, V\, \texttt{then}\, e_1\, \texttt{else}\, e_2 & \xrightarrow{\mathbb{S}} & e_1 & V \neq 0 \\
\texttt{if}\, V\, \texttt{then}\, e_1\, \texttt{else}\, e_2 & \xrightarrow{\mathbb{S}} & e_2 & V=0 \\
\texttt{alloc}(n) & \xrightarrow{\mathbb{A}(\ell..\ell+n-1)} & \ell & \\
[\ell]_O & \xrightarrow{\mathbb{R}(\ell,V,O)} & V & \\
[\ell]_O := V & \xrightarrow{\mathbb{W}(\ell,V,O)} & 0 & \\
\texttt{CAS}_{O_r,O_w}(\ell,V_o,V_n) & \xrightarrow{\mathbb{U}(\ell,V_o,V_n,O_r,O_w)} & 1 & \\
\texttt{CAS}_{O_r,O_w}(\ell,V_o,V_n) & \xrightarrow{\mathbb{R}(\ell,V',O_r)} & 0 & V' \neq V_o \\
\texttt{fence}_O & \xrightarrow{\mathbb{F}(O)} & 0 & \\
K[e] & \xrightarrow{\alpha} & K[e'] & e \xrightarrow{\alpha} e'
\end{array}
$$

FIGURE 3. Event-steps semantic rules: $e \xrightarrow{\alpha} e'$.

$$
\frac{e \xrightarrow{\alpha} e' \quad \texttt{consistentC11}(\mathcal{G}') \quad \mathcal{G}'.A = \mathcal{G}.A \uplus [a' \mapsto \alpha] \quad \mathcal{G}'.T = \mathcal{G}.T \uplus [a' \mapsto i] \\ \mathcal{G}'.\mathsf{sb} = \mathcal{G}.\mathsf{sb} \uplus (a,a') \quad \mathcal{G}'.\mathsf{mo} \supseteq \mathcal{G}.\mathsf{mo} \quad \mathcal{G}'.\mathsf{rf} \in \{\mathcal{G}.\mathsf{rf}, \mathcal{G}.\mathsf{rf} \uplus [a' \mapsto b]\}}{\langle \mathcal{T} \uplus [i \mapsto (a,e)]; \mathcal{G} \rangle \longrightarrow \langle \mathcal{T} \uplus [i \mapsto (a',e')]; \mathcal{G}' \rangle}
$$

$$
\langle \mathcal{T} \uplus [i \mapsto (a, K[\texttt{fork}\,(e)])]; \mathcal{G} \rangle \longrightarrow \langle \mathcal{T} \uplus [i \mapsto (a, K[0]) \uplus [j \mapsto (a,e)]; \mathcal{G} \rangle
$$

FIGURE 4. Machine-step semantics: $\langle \mathcal{T}; \mathcal{G} \rangle \longrightarrow \langle \mathcal{T}'; \mathcal{G}' \rangle$.

release-sequences, as we will need to know if different writes are from a same thread or not. Secondly, we know that a' comes after a, so we record it in the sequenced-before relation ($\mathcal{G}'.\mathsf{sb}$). Finally, the modification-order only gets updated if a' is a write ($\mathbb{W}$) or a success update action ($\mathbb{U}$), so we have $\mathcal{G}'.\mathsf{mo} \supseteq \mathcal{G}.\mathsf{mo}$. Similarly, the read-from relation only gets updated if a' is a read ($\mathbb{R}$) or an update action ($\mathbb{U}$) which reads from a write/update action b. The exact way these two relations to be updated is restricted by the C11 memory model axioms `consistentC11` that to be presented in the next subsection.

The second machine-step rule indicates that the `fork` e command creates a new thread (i.e. j) that will be added to the thread pool. The expression e is waiting to be executed in the new thread while the parent thread (i) has whatever left in the evaluation context $K[0]$.

A thread terminates if its expression is reduced to be a pure value and the program terminates when all its threads terminate.

C. THE MEMORY MODEL

A memory model defines how different CPU cores can access a shared memory, and thus controls how multithread programs should behave. As a language-level memory model that must be sufficiently generalised, the C11 memory model regulates the program behaviours by using a group of axioms based on the event graph. We have discussed the event graph; now we first introduce several derived relations before we can formally introduce the C11 memory model axioms.

1) HAPPENS-BEFORE RELATION

The *happens-before* relation is the cornerstone of the causality in C11 programs. That is, for two events a and b, unless we can establish that a *happens before* b ($a \xrightarrow{\mathsf{hb}} b$), there is no guarantee that a's effect will be observed by b. The happens-before relation is derived from the *sequenced-before* and the *synchronised-with* relations (to be discussed shortly in §II-C2), i.e., $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$. Intuitively, the happens-before relation preserves the program order for events within the same thread; however, for the events from different threads can by ordered in hb only if their threads are synchronised at appropriate locations.

This idea is demonstrated in Fig. 5 with an unsuccessful message passing program. In this example, we assume both x and y are initialised as 0. In the first thread, the flag y is changed to 1 (event b) *after* the message x is set to be 42 (event a); the second thread first reads y to be 1 (event c) then reads x (event d). Though a chain of relations can be established as $a \xrightarrow{\mathsf{sb}} b \xrightarrow{\mathsf{rf}} c \xrightarrow{\mathsf{sb}} d$, the stale value 0 can still be read by d as the read-from relation between b and c is not strong enough to form a synchronisation, thus the

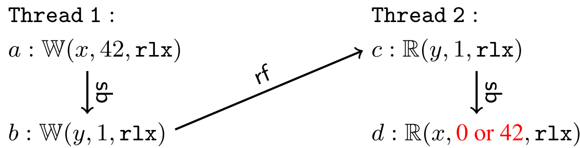


FIGURE 5. A failed message passing example.

happens-before relation $a \overset{\mathsf{hb}}{\to} d$ could not be derived and a's effect is not guaranteed to be seen by d.

2) RELEASE-SEQUENCE AND SYNCHRONISATION

As a weak memory model, the C11 memory model allows threads to have different observations about the memory.[1] But when necessary, one thread's observation can be passed to another if they are synchronised. To form a synchronisation and share its observation, the sharer's thread must first perform a *release* action. Intuitively, the release action would label its up-to-date memory observation as ready to be shared. Then a sequence of store operations, i.e. a *release-sequence*, work like messengers notifying their readers that some information can be acquired. However, the acquisition is only successful after an acquire action is performed in the reader's thread. In this way, the acquire action is synchronised with the release action and is obliged to acknowledge the memory modifications happened before the release action.

The release-sequence plays the crucial role in this process. It is led by an action with the *release* memory order, i.e., the *release head*, which can either be a release fence or a release store, and is followed by the longest sub-sequence of store operations from the modification order (mo) where these store operations are either in the same thread as the release head or atomic update operations (i.e., $\mathbb{U}$).

As the example shown in Fig. 6a the release write event a is the release head; and the release-sequence also contains b and c as they follow a in the mo order and is either in a's thread (b) or is an atomic update (c). Another thread can acquire a's observation by reading from any event from this sequence. Fig. 6b shows that a release-sequence can be interrupted if a non-update store operation (c in this case) from a different thread is positioned in the chain of mo order. In the case shown in Fig. 6b, the release-sequence only contains the head a. As shown in Fig. 6c, a relaxed write can lead a *hypothetical* release-sequence and in this case itself is called a *hypothetical* release head. If an operation acquires from this sequence, it is synchronised with a release fence prior to the hypothetical release head if there is any.

To formally define the synchronised-with relation, we first define a predicate (along with some shorthand definitions) that indicates if an action b is qualified to be a member of the release-sequence led by action a:

$$\mathsf{rs_element}(a, b) \triangleq \mathsf{sameThread}^{\mathcal{G}}(a, b) \vee \mathsf{isCAS}^{\mathcal{G}}(b)$$
$$\mathsf{sameThread}^{\mathcal{G}}(a, b) \triangleq \mathcal{G}.T(a) = \mathcal{G}.T(b)$$

[1] For instance, in the example shown in Fig. 5, thread 2 can observe a state where $x = 0 \wedge y = 1$, which is infeasible from the perspective of thread 1.

$$\begin{aligned}\mathsf{isCAS}^{\mathcal{G}}(b) &\triangleq \mathcal{G}.A(b) = \mathbb{U}(-,-,-,-,-)\\ \mathsf{W}_O(a) &\triangleq \mathcal{G}.A(a)\\ &= \mathbb{W}(-,-,O) \vee \mathbb{U}(-,-,-,-,O)\\ \mathsf{R}_O(a) &\triangleq \mathcal{G}.A(a)\\ &= \mathbb{R}(-,-,O) \vee \mathbb{U}(-,-,-,O,-)\end{aligned}$$

For a store action a, we say that event b is in a's release-sequence, $a \overset{\mathsf{rs}}{\to} b$ or $\mathsf{rs}(a, b)$, if and only if:

$$\mathsf{W}_O(a) \wedge \begin{pmatrix} a = b \vee \mathsf{rs_element}(a, b) \wedge a \overset{\mathsf{mo}}{\to} b \wedge \\ \forall c.\, a \overset{\mathsf{mo}}{\to} c \overset{\mathsf{mo}}{\to} b \Rightarrow \mathsf{rs_element}(a, c) \end{pmatrix}$$

Then we can formally define the synchronised-with relation as that shown in Fig. 7:

D. DEMONSTRATING C11 SYNCHRONISATIONS

We have introduced the highly flexible release-sequence based C11 synchronisation mechanism. In Fig. 8, we demonstrate how C11 synchronisations can be formed in different manners by restoring the message passing protocol discussed in Fig. 5 in various ways. Recall that, as summarised in Table 1, existing C11 program logics usually support simplified versions of the C11 synchronisation mechanism, due to its complicity, with limited scenarios allowed to from synchronisations. These demonstrations are also used to illustrate which types of C11 synchronisations can be supported by existing C11 program logics.

The first C11 program logics, RSL and GPS, can only be used to reason about C11 programs with synchronisations formed between release write and acquire read pairs as that is shown in Fig. 8a. With C11 fences supported, FSL and GPS+ can also reason about programs like that is shown in Fig. 8b. Still, they do not accept release-sequences with more than one element. FSL++ overcomes this limitation, but expect the release head, it only accepts atomic update operations to be in a release-sequence (Fig. 8c). To the best of our knowledge, only this work supports the reasoning about C11 programs with synchronisations based on fully-featured release-sequences, including the scenario shown in Fig. 8d.

E. THE AXIOMATIC MODEL

With the preparations that have been made about synchronisations, happens-before relations and etc., in this subsection, we present the axiomatic definitions for the C11 memory model in Fig. 9 following a similar approach used by [4]. Intuitively, the axioms are regulations that rule out illegal executions, e.g., "no one can read from an event that happens *after* itself" or "an update action cannot be interrupted", and etc. These axiomatic rules also leave us enough room to ensure the aforementioned principle: no guarantee of observation without happens-before relations.

Specifically, ConsistentMO1 states that mo is a binary relation over writing actions. ConsistentMO2 requires all writing actions in mo to follow a strict total order. ConsistentRF1 indicates there always is at least one writing action

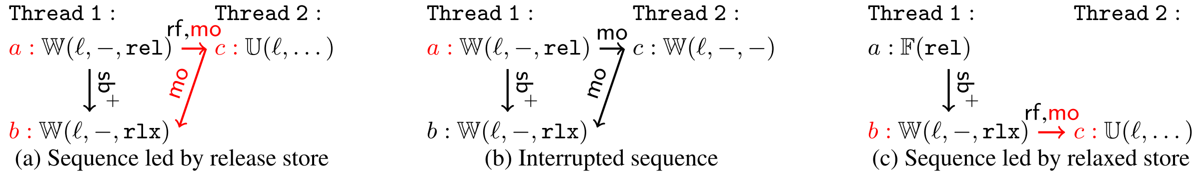


FIGURE 6. Examples of release-sequence (highlighted in red).

$$
\mathsf{sw} \triangleq \left\{ (a,b) \;\middle|\; \begin{array}{l}
\mathsf{W}_{\mathtt{rel}}(a) \wedge \mathsf{R}_{\mathtt{acq}}(b) \wedge \exists c.\ \mathsf{rs}(a,c) \wedge \mathsf{rf}(b) = c\ \vee \\
\mathsf{W}_{\mathtt{rel}}(a) \wedge \mathcal{G}.A(b) = \mathbb{F}(\mathtt{acq}) \wedge \exists c,d.\ \mathsf{R}_{\mathtt{rlx}}(d) \wedge \mathsf{rs}(a,c) \wedge \mathsf{rf}(d) = c \wedge (d,b) \in \mathsf{sb}^+\ \vee \\
\mathcal{G}.A(a) = \mathbb{F}(\mathtt{rel}) \wedge \mathsf{R}_{\mathtt{acq}}(b) \wedge \exists c,d.\ \mathsf{W}_{\mathtt{rlx}}(c) \wedge (a,c) \in \mathsf{sb}^+ \wedge \mathsf{rs}(c,d) \wedge \mathsf{rf}(b) = d\ \vee \\
\mathcal{G}.A(a) = \mathbb{F}(\mathtt{rel}) \wedge \mathcal{G}.A(b) = \mathbb{F}(\mathtt{acq})\ \wedge \\
\quad \exists c,d,e.\ \mathsf{W}_{\mathtt{rlx}}(c) \wedge \mathsf{R}_{\mathtt{rlx}}(e) \wedge (a,c) \in \mathsf{sb}^+ \wedge (e,d) \in \mathsf{sb}^+ \wedge \mathsf{rs}(c,d) \wedge \mathsf{rf}(e) = d
\end{array} \right\}
$$

FIGURE 7. Synchronised-with relation.

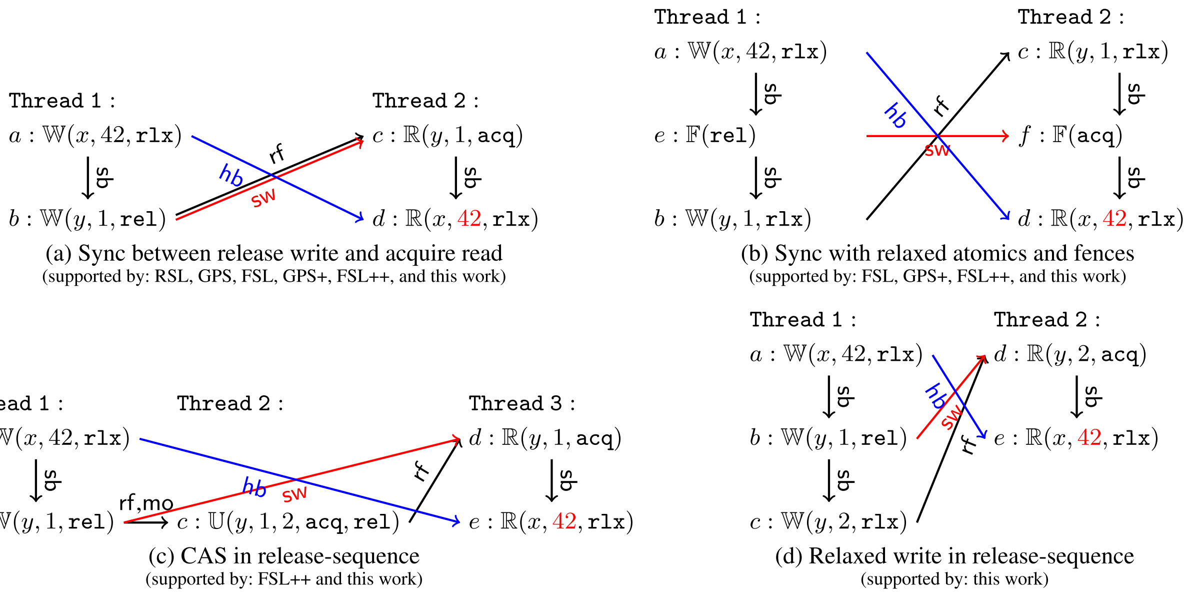


FIGURE 8. Examples of C11 synchronisations.

$$
\begin{array}{lr}
\mathsf{consistentC11}((A, T, \mathsf{sb}, \mathsf{mo}, \mathsf{rf})) \triangleq & \\
\quad \forall a, b.\ \mathsf{mo}(a,b) \Rightarrow \exists \ell.\ \mathsf{writes}(a,\ell,-), \mathsf{writes}(b,\ell,-) & \text{(ConsistentMO1)} \\
\wedge\ \forall \ell.\ \mathsf{strictTotalOrder}(\{a \mid \mathsf{writes}(a,\ell,-)\}, \mathsf{mo}) & \text{(ConsistentMO2)} \\
\wedge\ \forall b.\ \mathsf{rf}(b) \neq \bot \Leftrightarrow \exists \ell, a.\ \mathsf{writes}(a,\ell,-) \wedge \mathsf{reads}(b,\ell,-) \wedge \mathsf{hb}(a,b) & \text{(ConsistentRF1)} \\
\wedge\ \forall a, b.\ \mathsf{rf}(b) = a \Rightarrow \exists \ell, V.\ \mathsf{writes}(a,\ell,V) \wedge \mathsf{reads}(b,\ell,V) \wedge \neg \mathsf{hb}(b,a) & \text{(ConsistentRF2)} \\
\wedge\ \forall a, b.\ \mathsf{rf}(b) = a \wedge (\mathsf{isNonatomic}(a) \vee \mathsf{isNonatomic}(b)) \Rightarrow \mathsf{hb}(a,b) & \text{(ConsistentRFNA)} \\
\wedge\ \forall a, b.\ \mathsf{hb}(a,b) \Rightarrow a \neq b \wedge \neg \mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a))\ \wedge \neg \mathsf{mo}(\mathsf{rf}(b), a) \wedge \neg \mathsf{mo}(b, \mathsf{rf}(a)) \wedge \neg \mathsf{mo}(b,a) & \text{(Coherence)} \\
\wedge\ \forall a, c.\ \mathsf{isUpd}(c) \wedge \mathsf{rf}(c) = a \Rightarrow \mathsf{mo}(a,c) \wedge \nexists b.\ \mathsf{mo}(a,b) \wedge \mathsf{mo}(b,c) & \text{(AtomicCAS)} \\
\wedge\ \forall a \neq b, \vec{\ell}, \vec{\ell}'.\ A(a) = \mathbb{A}(\vec{\ell}) \wedge A(b) = \mathbb{A}(\vec{\ell}') \Rightarrow \vec{\ell} \cap \vec{\ell}' = \varnothing & \text{(ConsistentAlloc)} \\
\wedge\ \mathsf{acyclic}(\mathsf{hb} \cup \mathsf{rf}) & \text{(Acyclic)}
\end{array}
$$

$$
\begin{array}{l}
\text{where } \mathsf{strictTotalOrder}(S, R) \triangleq (\nexists a.\ R(a,a)) \wedge (\forall a,b,c.\ R(a,b) \wedge R(b,c) \Rightarrow R(a,c)) \\
\qquad\qquad \wedge\ (\forall a, b \in S.\ a \neq b \Rightarrow R(a,b) \vee R(b,a)) \\
\quad \mathsf{acyclic}(R) \triangleq \nexists x.\ R^+(x,x) \\
\quad \mathsf{reads}(a,\ell,V) \triangleq A(a) \in \mathbb{R}(\ell,V,-), \mathbb{U}(\ell,V,-) \\
\quad \mathsf{writes}(a,\ell,V) \triangleq A(a) \in \mathbb{W}(\ell,V,-), \mathbb{U}(\ell,-,V)
\end{array}
$$

FIGURE 9. The C11 axioms.

before a reading on the same location, that is, all locations are initialised before reading. ConsistentRF2 says that a reading action cannot read from a writing that happens after itself. For a non-atomic reading action, ConsistentRFNA requires

it must read from a writing action that happens before it. Coherence puts restrictions on happens-before relations and the modification orders, e.g. a reading actions should not read older value than its happens-before ancestors. AtomicCAS enforces no interruption could happen to an atomic update action. ConsistentAlloc states the sets of locations allocated by two allocation actions will not intersect, that is, no location will be allocated more than once. Acyclic is introduced to rule out the thin-air-read problem following [6], [9].

Note that while atomic locations are meant to be accessed concurrently, concurrent accesses (accesses that are not ordered in hb) to non-atomic locations with at least one write action lead to a hazardous situation called *data-race*, in which the program behaviour is undefined. The *memory error* is another hazardous situation, which involves accessing a location before it is allocated. Definitions for these two hazardous situations are needed to complete our semantical model as we need them to rule out executions with undefined results.

$$\mathsf{dataRace}(\mathcal{G}) \triangleq \exists \ell.\exists a, b \in \mathsf{dom}(\mathcal{G}.A).\\ \begin{pmatrix} \mathcal{G}.A(a) = \mathbb{W}(\ell, -, \mathtt{na}) \wedge \mathcal{G}.A(b) = \mathbb{W}(\ell, -, \mathtt{na}) \vee \\ \mathcal{G}.A(a) = \mathbb{W}(\ell, -, \mathtt{na}) \wedge \mathcal{G}.A(b) = \mathbb{R}(\ell, -, \mathtt{na}) \vee \\ \mathcal{G}.A(a) = \mathbb{R}(\ell, -, \mathtt{na}) \wedge \mathcal{G}.A(b) = \mathbb{W}(\ell, -, \mathtt{na}) \end{pmatrix}\\ \wedge \neg((a, b) \in \mathsf{hb} \vee (b, a) \in \mathsf{hb})$$

$$\mathsf{memErr}(\mathcal{G}) \triangleq \exists l.\exists b \in \mathsf{dom}(\mathcal{G}.A).\ (\mathcal{G}.A(b) = \mathbb{W}(\ell, -, -)\\ \vee\ \mathcal{G}.A(b) = \mathbb{R}(\ell, -, -) \vee \mathcal{G}.A(b) = \mathbb{U}(\ell, -, -, -, -))\\ \wedge\ \nexists a \in \mathsf{dom}(\mathcal{G}.A).A(a) = \mathbb{A}(\vec{\ell}) \wedge \ell \in \vec{\ell} \wedge (a, b) \in \mathsf{hb}$$

III. RECAP OF GPS+

Verifying concurrent programs is difficult. To reason about C11 concurrent programs with weak memory behaviours is even harder. To do this, GPS-like logics amalgamate three techniques from state-of-the-art concurrent program logics (e.g. [15]–[28]), namely ghost states, protocols and separation logic. Our proposed program logic, GPS++, follows the line of GPS works. In this section, we lay the foundation for the discussion of our program logic by briefly introducing its closest predecessor GPS+.

A. PROTOCOLS FOR ATOMIC LOCATIONS

As illustrated in previous sections, in C11 concurrent programs threads communicate with each other by writing and reading atomic locations. As updates may happen in other threads, it is difficult to determine the precise value of an atomic location. Therefore, GPS/GPS+ employs the concept of *per-location protocols* and each atomic location has its own protocol depicting how its states can evolve along with the program execution. The possible states for an atomic location are partially ordered, and the *state assertion* is used to describe the lower-bound state an atomic location can be in. That is, assuming an atomic location ℓ's protocol is τ, the state assertion $\boxed{\ell : s}\boxed{\tau}$ says that ℓ is at least at state s.

Atomic locations can be concurrently accessed by different threads, therefore their state assertions should be able to be duplicated and shared between different threads. These assertions that do not require exclusive ownership are called *knowledge* in GPS logics. A knowledge assertion is decorated with the notation $\square$ and the following rules applied:

$$\boxed{\ell : s}\boxed{\tau} \Rightarrow \square\,\boxed{\ell : s}\boxed{\tau} \qquad \square P \Leftrightarrow \square P * \square P \qquad \square P \Rightarrow P$$

The first rule states that a state assertion is knowledge. The second rule says that copies can be made out of a knowledge assertion. When necessary a knowledge assertion can also be transformed back to its normal form using the third rule.

To complete the protocol definition τ for its atomic location ℓ, a partial order $\sqsubseteq_\tau$ must be provided to depict all ℓ's possible state transitions. That is, assuming ℓ is currently in state s and a write command moves it to state s', this is valid if and only if $s \sqsubseteq_\tau s'$ is defined in the protocol. In addition, state interpretations need to be specified for all states. A state interpretation $\tau(s, z)$ describes the conditions that must be satisfied for a write command to be permitted to write value z to ℓ and transfer ℓ to state s. On the other hand, when a reader reads ℓ at state s, it knows some conditions have been established in the writer's thread. Therefore, they can reach agreement about these facts, i.e., they are synchronised. The following rules formally capture this process.

[GPS+-RELEASE-STORE]

$$\frac{P \Rrightarrow \tau(s'', v) * Q \qquad \forall s' \sqsupseteq_\tau s.\ \tau(s', -) * P \Rightarrow s'' \sqsupseteq_\tau s'}{\left\{\boxed{\ell : s}\boxed{\tau} * P\right\}\ [\ell]_{\mathtt{rel}} := v \left\{\boxed{\ell : s''}\boxed{\tau} * Q\right\}}$$

The [GPS+-RELEASE-STORE] rule states that to move ℓ to the target state s'', the state interpretation $\tau(s'', v)$ must be derivable from the resource P currently possessed in its precondition (the first premise). Note that the *ghost move* $\Rrightarrow$ used to imply the state interpretation refers to a transition that only affects auxiliary states leaving physical states unchanged. Note also that due to the possible moves from the environment, the actual state of ℓ before the writing may be different from the lower-bond state s in the triple's precondition. Therefore, the second premise is introduced requiring that the target state s'' is reachable from any possible state s' that ℓ might be currently in.

[GPS+-ACQUIRE-LOAD]

$$\frac{\forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z) * P \Rightarrow \square Q}{\left\{\boxed{\ell : s}\boxed{\tau} * P\right\}\ [\ell]_{\mathtt{acq}}\ \left\{z.\ \exists s'.\ \boxed{\ell : s'}\boxed{\tau} * P * \square Q\right\}}$$

The [GPS+-ACQUIRE-LOAD] states that some knowledge $\square Q$ can be added to an acquire read's postcondition if the knowledge can be derived from the candidate states' interpretations. Note that only knowledge assertions can be retrieved, as the read operations from different threads may observe a same state and try to retrieve the same information. If the information to be retrieved is not duplicable, conflicts will raise. The assertion P included in this rule enables *rely-guarantee* style of reasoning and reduces the possible states that can be read [6].

B. ESCROWS FOR NON-ATOMIC LOCATIONS

The [GPS+-ACQUIRE-LOAD] rule only retrieve knowledge assertions from state interpretations. However, the ownership of some exclusive resource is also often needed to be transmitted across threads. This can be done by using *escrows* in GPS/GPS+. Intuitively, an escrow $\sigma : P \rightsquigarrow Q$ is a safe protecting Q with a non-duplicable resource P as the key. Following ghost move rules are used to put some resource (Q) under escrow and retrieve it:

$$\frac{\sigma : P \rightsquigarrow Q}{Q \Rrightarrow [\sigma]} \qquad \frac{\sigma : P \rightsquigarrow Q}{P \wedge [\sigma] \Rrightarrow Q}$$

An escrow $[\sigma]$ is no longer ownership-dependent. It can be transformed into knowledge following the rule: $[\sigma] \Leftrightarrow \square[\sigma]$. As shown in the second rule, the "key" P will be consumed once it is used to open the safe and retrieve the resource Q. Therefore P is preferred to be a *ghost assertion* instead some assertions relating to physical resources. A ghost assertion in the form $\boxed{\gamma : t}_{\mu}$ indicates that γ is a ghost variable, and its value is a ghost permission t drawn from some *partial commutative monoid* (PCM) type μ. With a fresh identity, a new ghost value t can be introduced out of thin air: true $\Rrightarrow \exists \gamma. \boxed{\gamma : t}_{\mu}$.

C. WORKING WITH FENCES

As shown in Fig. 8b, synchronisations can also be established by using relaxed write and read operations but only with the help from C11 fences. In GPS+, this process is interpreted as that a relaxed write can only share the resource made *shareable* (denoted as $\langle P \rangle$) by a prior release fence. On the other hand, when some resource is retrieved by a relaxed read, it cannot be used instantly. Instead, it is marked as *waiting-to-be-acquired* $\boxtimes P$ and an acquire fence is needed to transform it back to its normal form.

The fence related rules are adopted by our new program logic. Detailed discussions will be provided in the next section to illustrate how we make them compatible with other more complicated features.

IV. REASONING ABOUT C11 RELEASE-SEQUENCES AND FRACTIONAL PERMISSIONS

The use of release-sequences provides C11 programs great flexibility to choose the best way to synchronise their threads. However, as discussed in previous sections no existing program logic supports the formal verification about the use of fully featured release-sequences due to its complexity. Also, there is no work in GPS family that supports fractional permissions. In this section, we introduce our new reasoning framework, GPS++, that support the aforementioned features with the aid from several novel techniques.

A. A NEW TYPE OF ASSERTION AND THE ENHANCED PROTOCOL SYSTEM

The key to reason about the C11 synchronisation process is to deal with the relaxed write operations involved. As that is discussed in §II-C2, unlike a release write, a relaxed write cannot form a synchronisation by itself. That is, no resource can be shared by a relaxed write to its readers unless (1) there is a release fence prior to it; or (2) it belongs to a release write's release-sequence. To reason about the behaviours of a relaxed write in the C11 synchronisation, its context must be taken under consideration. The first scenario is relatively easier, as we can adopt the *shareable* assertion introduced by GPS+ to indicate if there is a prior release fence that makes some resource available for the relaxed write operation to share. The second scenario is more complicated, as we need to know if there is a prior release write to the same memory location and if so, whether or not the release-sequence led by that release write is still valid at the point where the relaxed write takes place.

To tackle this problem, a naive solution is to introduce location-based restricted shareable assertions. That is, a release write operation on location ℓ may create an assertion $\langle P \rangle_\ell$, which indicates P is shareable by following relaxed writes operation on ℓ who are assumed to be the members of its release-sequence. However, as discussed in §II-C2 a release-sequence could be interrupted by non-update writes from other threads and this definition is not sufficient to be used to detect these potential interruptions. Therefore, we introduce state-based restricted shareable assertions (which we call *restricted-shareable assertions* for short) instead. Specifically, a release write that changes location ℓ to state s may make some resource P shareable, $\langle P \rangle_s$, for the members of its release-sequence. To check if a following relaxed write belongs to the release-sequence and can use the restricted-shareable resource $\langle P \rangle_s$, we first check (1) whether they are operations on the same location; (2) whether they are in the same thread; and (3) whether the sequence is free from interruptions. The check for condition (1) can be done by simply examining whether the two writes follow the same protocol. To enable the checks for condition (2) and (3), we extend the state interpretation $\tau(s, z)$ used in GPS+ to the form like $\tau(s, z, tid, upd)$, where tid indicates in which threads the target location can be transformed to the state s, and upd is 1 if the state s can only be reached by atomic update operations or 0 otherwise. With these preparations, we derive the following predicates:

$$\begin{aligned} &\mathsf{sameThread}(s, s') \\ &\quad \triangleq \forall t, t', v, v', c, c'. (\tau(s, v, t, c) \not\Rrightarrow \texttt{false} \wedge \\ &\qquad \tau(s', v', t', c') \not\Rrightarrow \texttt{false}) \Rightarrow t = t' \\ &\mathsf{isCAS}(s) \\ &\quad \triangleq \forall v, t, u. (\tau(s, v, t, u) \not\Rrightarrow \texttt{false}) \Rightarrow u = 1 \end{aligned}$$

With these definitions, the thread and interruption checks can be formalised as:

$$\forall s''. s' \sqsupseteq_\tau s'' \sqsupseteq_\tau s \Rightarrow \mathsf{sameThread}(s'', s) \vee \mathsf{isCAS}(s'').$$

where s is the state established by the release head and s' is the target state of the relaxed write being checked. The following properties can be derived for our new restricted-shareable

assertions based on our semantical model:

$$\frac{\text{[SEPARATION-R]}}{\langle P_1 * P_2\rangle_s \Leftrightarrow \langle P_1\rangle_s * \langle P_2\rangle_s} \qquad \frac{\text{[UNSHARE-R]}}{\langle P\rangle_s \Rrightarrow P}$$

The [SEPARATION-R] rule indicates that a restricted-shareable assertion can be split as while as several restricted-shareable assertions can be merged if they are restricted to the same state. The [UNSHARE-R] rule states that a restricted-shareable assertion can be transformed back into its normal form via a ghost move.

The new restricted-shareable assertion, the enhanced protocol system, and their properties are semantically supported by our upgraded resource model, which will be presented in later sections.

B. REASONING ABOUT C11 RELEASE-SEQUENCES

With our new restricted-shareable assertions and the enhanced protocol system introduced, new reasoning rule can be devised to handle the C11 programs with fully featured release-sequences. In this section, we first introduce the essential rules most related to C11 synchronisations in Fig. 10. Rules to deal with fractional permissions are presented in §IV-C, while other rules are discussed in §IV-D.

Unlike a release write in GPS+ which only needs to concern about what resource it can share to its readers, a release write in this work can also initiate a release-sequence, that is, it can make some resource shareable by the qualified relaxed writes followed. This idea is formalised in our [RELEASE-STORE] rule as that part of the resource P currently held in the release write's precondition can be transformed into a restricted shareable assertion $\langle Q_1\rangle_{s''}$.

We require the P used in the rule must be *normal*, i.e., it can not contain any special forms of assertions (e.g., shareable assertions or waiting-to-be-acquired assertions). This ensures that Q_1 is also free from special assertions and we do not create the problematic nesting of special assertions by putting Q_1 into $\langle \ldots\rangle_{s''}$. The formal definition for the normality check is $\mathsf{normal}(P) \triangleq P \Rightarrow \texttt{false} \vee \langle P\rangle \not\Rightarrow \texttt{false}$. This is not the only occasion where the normality check is used. Allowing special assertions to be transmitted across threads also raises problems, therefore we require state assertions to be *normal* as well to prevent special assertions from being included.

The [RELAXED-STORE-2] rule illustrate how a release write works in a release write's release-sequence. With the restricted-shareable assertion $\langle P_2\rangle_{s_o}$ created by a release write and passed down to the relaxed write, the relaxed write knows it may be in a release-sequence created at state s_o. The validity of the release-sequence needs to be checked using the second premise (recall §IV-A) then P_2 can be used to imply the target state interpretation (the first premise).

Our reasoning framework is compatible with the rules developed in GPS+ for reasoning about the synchronisation initiated by a relaxed write with the help from a prior release fence. Therefore we inherit these rules as our [RELAXED-STORE-1] and [RELEASE-FENCE] rules. They state that a release fence can turn some resource into a (unrestricted) shareable resource and then being used by *any* relaxed write that follows. Intuitively, these tow rules are sound for C11 release-based synchronisation mechanism because every relaxed write after a release fence is a (hypothetical) release head and is allowed to share the observations established at point where the release fence took place (recall §II-C2).

On the other side, if the reader is an acquire read, the [ACQUIRE-LOAD] rule applies. It states that when observing the location ℓ at a certain state s', some knowledge $\square Q$ can be learnt from the state interpretation. Similarly, as shown in the [RELAXED-LOAD] rule, a relaxed read can also retrieve some knowledge from the state interpretation, but this knowledge $\boxtimes Q$ is not instantly useable and is *waiting-to-be-acquired* by a following acquire fence that may transfer it to a normal knowledge according to the [ACQUIRE-FENCE] rule. These three rules are adopted from GPS+ with minor changes, as our extension with release-sequence is still compatible with the principles working on the reader's side.

Compare-and-swap (CAS) plays an important role in C11 concurrent programming. It is the foundation to the implementation of many locks and non-blocking algorithms. It can join in a release-sequence without being in the same thread as the release head. Therefore, sophisticated concurrent algorithms like the atomic reference counter [11] can use CAS operations to create synchronisations between many different threads. GPS+ provides some basic support to CASes without user specified memory orders. In this work, we devise a set of rules to cover CAS operations with all possible memory order specifications. We first take a close look at the [ACQ-REL-CAS] rule. The first premise corresponds to the case of success where ℓ's value is same v_o as expected. In this case, the acquire-release CAS performs as a release store However, unlike normal release write which can only use the resource P in its precondition to imply its target state's interpretation, a successful CAS can also use the resource from the state interpretation of ℓ's current state ($\tau(s', v_o, -, -)$). In this way, the CAS can retransmit the information passed down in its release-sequence. Moreover, a successful CAS can retrieve non-knowledge resources from the state interpretation of ℓ's current state. The second premise corresponds to the case of failure where ℓ is found to have some value other than v_o. In this case, the acquire-release CAS performs as an acquire read and some knowledge $\square R$ can be retrieved from the actual state observed.

The ideas for the other CAS rules are similar. However, when a CAS has relaxed memory order for its reading component ([RLX-REL-CAS], [RLX-RLX-CAS-1], and [RLX-RLX-CAS-2]), it still can retrieve some information (Q) from the state it reads but this information needs to be marked as "waiting-to-be-acquired" in its post condition ($\boxtimes Q$). When a CAS has relaxed memory order for its writing component ([ACQ-RLX-CAS], [RLX-RLX-CAS-1], and [RLX-RLX-CAS-2]), it can only use the resources that are already sharable to derive its target state interpretation.

[RELEASE-STORE]

$$\frac{P \Rrightarrow \tau(s'', v, -, 0) * Q_1 * Q_2 \qquad \mathsf{normal}(P) \qquad \forall s' \sqsupseteq_\tau s.\ \tau(s', -, -, -) * P \Rightarrow s'' \sqsupseteq_\tau s'}{\{\boxed{\ell : s \mid \tau} * P\}\ [\ell]_{\mathtt{rel}} := v\ \{\boxed{\ell : s'' \mid \tau} * \langle Q_1 \rangle_{s''} * Q_2\}}$$

[RELAXED-STORE-1]

$$\frac{P_2 \Rrightarrow \tau(s'', v, -, 0) * Q \qquad \forall s' \sqsupseteq_\tau s.\ \tau(s', -, -, -) * P_1 * P_2 \Rightarrow s'' \sqsupseteq_\tau s'}{\{\boxed{\ell : s \mid \tau} * P_1 * \langle P_2 \rangle\}\ [\ell]_{\mathtt{rlx}} := v\ \{\boxed{\ell : s'' \mid \tau} * P_1 * Q\}}$$

[RELAXED-STORE-2]

$$\frac{P_2 \Rrightarrow \tau(s'', v, -, 0) * Q \qquad \forall s'.\ s'' \sqsupseteq_\tau s' \sqsupseteq_\tau s_o \Rightarrow \mathsf{sameThread}(s', s_o) \vee \mathsf{isCAS}(s') \qquad \forall s' \sqsupseteq_\tau s.\ \tau(s', -, -, -) * P_1 * P_2 \Rightarrow s'' \sqsupseteq_\tau s'}{\{\boxed{\ell : s \mid \tau} * P_1 * \langle P_2 \rangle_{s_o}\}\ [\ell]_{\mathtt{rlx}} := v\ \{\boxed{\ell : s'' \mid \tau} * P_1 * Q\}}$$

[RELEASE-FENCE]

$$\frac{\mathsf{normal}(P)}{\{P\}\ \mathtt{fence_{rel}}\ \{\langle P \rangle\}}$$

[ACQUIRE-FENCE]

$$\frac{}{\{\boxtimes P\}\ \mathtt{fence_{acq}}\ \{\Box P\}}$$

[ACQUIRE-LOAD]

$$\frac{\forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z, -, -) * P \Rightarrow \Box Q}{\{\boxed{\ell : s \mid \tau} * P\}\ [\ell]_{\mathtt{acq}}\ \{z.\ \exists s'.\ \boxed{\ell : s' \mid \tau} * P * \Box Q\}}$$

[RELAXED-LOAD]

$$\frac{\forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z, -, -) * P \Rightarrow \Box Q}{\{\boxed{\ell : s \mid \tau} * P\}\ [\ell]_{\mathtt{rlx}}\ \{z.\ \exists s'.\ \boxed{\ell : s' \mid \tau} * P * \boxtimes Q\}}$$

[ACQ-REL-CAS]

$$\frac{\forall s' \sqsupseteq_\tau s.\ \tau(s', v_o, -, -) * P \Rrightarrow \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', v_n, -, 1) * Q \qquad \forall s'' \sqsupseteq_\tau s.\ \forall y \neq v_o.\ \tau(s'', y, -, -) * P \Rightarrow \Box R}{\{\boxed{\ell : s \mid \tau} * P\}\ \mathtt{CAS_{acq,rel}}(\ell, v_o, v_n)\ \{\ z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} * ((z = 1 * Q) \vee (z = 0 * P * \Box R))\ \}}$$

[RLX-REL-CAS]

$$\frac{\forall s' \sqsupseteq_\tau s.\ \tau(s', v_o, -, -) * P \Rrightarrow \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', v_n, -, 1) * Q \qquad \forall s'' \sqsupseteq_\tau s.\ \forall y \neq v_o.\ \tau(s'', y, -, -) * P \Rightarrow \Box R}{\{\boxed{\ell : s \mid \tau} * P\}\ \mathtt{CAS_{rlx,rel}}(\ell, v_o, v_n)\ \left\{\begin{array}{l} z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} \\ * ((z = 1 * \boxtimes Q) \vee \\ (z = 0 * P * \Box R)) \end{array}\right\}}$$

[ACQ-RLX-CAS-1]

$$\frac{\begin{array}{c} \forall s' \sqsupseteq_\tau s.\ \tau(s', v_o, -, -) * P_2 \Rrightarrow \\ \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', v_n, -, 1) * Q \\ \forall s_i.\ s' \sqsupseteq_\tau s_i \sqsupseteq_\tau s_o \Rightarrow \mathsf{sameThread}(s_i, s_o) \vee \mathsf{isCAS}(s_i) \\ \forall s'' \sqsupseteq_\tau s.\ \forall y \neq v_o.\ \tau(s'', y, -, -) * P_1 * P_2 \Rightarrow \Box R \end{array}}{\left\{\begin{array}{l} \boxed{\ell : s \mid \tau} * \\ P_1 * \langle P_2 \rangle_{s_o} \end{array}\right\}\ \mathtt{CAS_{acq,rlx}}(\ell, v_o, v_n)\ \left\{\begin{array}{l} z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} \\ * ((z = 1 * Q) \vee \\ (z = 0 * P * \Box R)) \end{array}\right\}}$$

[ACQ-RLX-CAS-2]

$$\frac{\begin{array}{c} \forall s' \sqsupseteq_\tau s.\ \tau(s', v_o, -, -) * P_2 \Rrightarrow \\ \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', v_n, -, 1) * Q \\ \forall s'' \sqsupseteq_\tau s.\ \forall y \neq v_o.\ \tau(s'', y, -, -) * P_1 * P_2 \Rightarrow \Box R \end{array}}{\left\{\begin{array}{l} \boxed{\ell : s \mid \tau} * \\ P_1 * \langle P_2 \rangle \end{array}\right\}\ \mathtt{CAS_{acq,rlx}}(\ell, v_o, v_n)\ \left\{\begin{array}{l} z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} \\ * ((z = 1 * Q) \vee \\ (z = 0 * P * \Box R)) \end{array}\right\}}$$

[RLX-RLX-CAS-1]

$$\frac{\begin{array}{c} \forall s' \sqsupseteq_\tau s.\ \tau(s', v_o, -, -) * P_2 \Rrightarrow \\ \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', v_n, -, 1) * Q \\ \forall s_i.\ s' \sqsupseteq_\tau s_i \sqsupseteq_\tau s_o \Rightarrow \mathsf{sameThread}(s_i, s_o) \vee \mathsf{isCAS}(s_i) \\ \forall s'' \sqsupseteq_\tau s.\ \forall y \neq v_o.\ \tau(s'', y, -, -) * P_1 * P_2 \Rightarrow \Box R \end{array}}{\left\{\begin{array}{l} \boxed{\ell : s \mid \tau} * \\ P_1 * \langle P_2 \rangle_{s_o} \end{array}\right\}\ \mathtt{CAS_{rlx,rlx}}(\ell, v_o, v_n)\ \left\{\begin{array}{l} z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} \\ * ((z = 1 * \boxtimes Q) \vee \\ (z = 0 * P * \Box R)) \end{array}\right\}}$$

[RLX-RLX-CAS-2]

$$\frac{\begin{array}{c} \forall s' \sqsupseteq_\tau s.\ \tau(s', v_o, -, -) * P_2 \Rrightarrow \\ \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', v_n, -, 1) * Q \\ \forall s'' \sqsupseteq_\tau s.\ \forall y \neq v_o.\ \tau(s'', y, -, -) * P_1 * P_2 \Rightarrow \Box R \end{array}}{\left\{\begin{array}{l} \boxed{\ell : s \mid \tau} * \\ P_1 * \langle P_2 \rangle \end{array}\right\}\ \mathtt{CAS_{rlx,rlx}}(\ell, v_o, v_n)\ \left\{\begin{array}{l} z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} \\ * ((z = 1 * \boxtimes Q) \vee \\ (z = 0 * P * \Box R)) \end{array}\right\}}$$

FIGURE 10. Synchronisation related verification rules.

C. DEALING WITH FRACTIONAL PERMISSIONS

As discussed in §II-E, while atomic locations are designed for concurrent accesses, concurrent accesses (i.e., the accesses not ordered in hb) to a non-atomic location with at least one of them being a store operation lead to *data-races*. To ensure the verified programs are data-race-free, previous work in GPS family models each non-atomic location as a resource that could only be exclusively held by one thread at a time, which means these logics can not support the reasoning about programs with concurrent non-atomic reads (though they would not result in any race-condition). To verify real-world concurrent programs with concurrent non-atomic reads (e.g. the readers-writer-lock algorithm), we introduce fractional permissions for non-atomic locations.

Fractional permissions technique is an example of partial permissions [12], [13]. In our setting, a fraction in the interval [0, 1] is used to represent the portion of the ownership to a non-atomic location. The full permission $\ell \overset{1}{\mapsto} -$ is needed for a thread to write to ℓ; while for a non-atomic read, only a fraction of the permission will be sufficient:

[NON-ATOMIC-STORE]

$$\frac{}{\{\mathsf{uninit}(\ell) \vee \ell \overset{1}{\mapsto} -\}\ [\ell]_{\mathtt{na}} := v\ \{\ell \overset{1}{\mapsto} v\}}$$

[NON-ATOMIC-LOAD]

$$\frac{p \in (0, 1]}{\{\ell \overset{p}{\mapsto} v\}\ [\ell]_{\mathtt{na}}\ \{x.\ x = v * \ell \overset{p}{\mapsto} v\}}$$

The empty permission $\ell \overset{0}{\mapsto} -$ is semantically equivalent to `emp`. Permissions can also be combined or separated as defined below:

[SEPARATION-F]

$$\ell \overset{p}{\mapsto} v * \ell \overset{q}{\mapsto} v \iff \begin{cases} \ell \overset{p\oplus q}{\mapsto} v & \text{if } p \oplus q \text{ is defined} \\ \mathtt{false} & \text{otherwise} \end{cases}$$

$$\text{where} \quad p \oplus q = \begin{cases} p+q & \text{if } p, q, p+q \in [0, 1] \\ \textit{undefined} & \text{otherwise} \end{cases}$$

According to the composition rules, a full permission (writing permission) is not compatible with another full permission or any other non-zero permissions. As a result, a program verified by our logic would not have any race condition where a write goes in parallel with other accesses to the same non-atomic location.

D. OTHER RULES

Besides the rules highlighted in previous subsections, we also have the following rules that make our reasoning system complete. We gather them into groups for the convenience of discussion.

The following inference rules depict properties of knowledge assertions. That is, knowledge can be transformed back to its normal form; knowledge symbol can be safely nested; a piece of knowledge acts like pure information and thus the separation assertion is equivalent to the logical conjunction; a picked escrow, an assertion about atomic location and a pure term are all knowledge; and a duplicable ghost term is also a form of knowledge.

[KNOWLEDGE-MANIPULATION-1...7]

$$\Box P \Rightarrow P \qquad \Box P \Rightarrow \Box\Box P \qquad \Box P * Q \Leftrightarrow \Box P \wedge Q$$

$$[\sigma] \Rightarrow \Box[\sigma] \qquad \boxed{t : t' \mid \tau} \Rightarrow \Box \boxed{t : t' \mid \tau} \qquad t = t' \Rightarrow \Box t = t'$$

$$\frac{t \cdot_{\mu} t = t}{\boxed{\gamma : t \mid \mu} \Rightarrow \Box \boxed{\gamma : t \mid \mu}}$$

The first inference rule below states that ghost terms can be composed or separated according to their PCM definitions. The second inference rule states that two atomic assertions about the same location only coherence if the protocols are same and the states are reachable from one to another or the other way around.

[SEPARATION-1...2]

$$\boxed{\gamma : t \mid \mu} * \boxed{\gamma : t' \mid \mu} \Leftrightarrow \boxed{\gamma : t \cdot_{\mu} t' \mid \mu}$$

$$\boxed{\ell : s \mid \tau} * \boxed{\ell : s' \mid \tau'} \Rightarrow \tau = \tau' \wedge (s \sqsubseteq_{\tau} s' \vee s' \sqsubseteq_{\tau} s)$$

Following rules are about possible ghost moves. Particularly, similar to the [UNSHARE-R] rule we have discussed before, the fifth rule allows us to change an unrestricted shareable assertion to its normal form. The seventh rule states that a new ghost term can popup from thin air with a fresh identifier. The eighth rule states that a ghost variable can be updated to a new value as long as the new value is compatible with the environment. The last two rules are inherited from GPS/GPS+ to cope escrows.

[GHOST-MOVE-1...8]

$$\frac{P \Rrightarrow Q}{P \Rrightarrow Q} \qquad \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R} \qquad \frac{P \Rrightarrow Q \quad Q \Rrightarrow R}{P \Rrightarrow R} \qquad \langle P \rangle \Rrightarrow P$$

$$\mathsf{true} \Rrightarrow \exists \gamma. \boxed{\gamma : t \mid \mu} \qquad \frac{\forall t_F : [\![\mu]\!].\ t_1 \#_{\mu} t_F \Rightarrow t_2 \#_{\mu} t_F}{\boxed{\gamma : t_1 \mid \mu} \Rrightarrow \boxed{\gamma : t_2 \mid \mu}}$$

$$\frac{\sigma : P \rightsquigarrow Q}{Q \Rrightarrow [\sigma]} \qquad \frac{\sigma : P \rightsquigarrow Q}{P \wedge [\sigma] \Rrightarrow Q}$$

Note that, following the GPS/GPS+ logic for PCM terms with type μ we use the shorthand notation $t_1 \#_{\mu} t_2$ to indicate that $t_1 \oplus t_2$ is defined. The type declaration can be omitted when it is obvious.

The following rule is for the memory allocation. Starting with any valid precondition, $\mathtt{alloc}(n)$ allocates n fresh and continuous locations, which are marked as uninitialised, and uses the leading location as its return value.

[ALLOCATION]

$$\{\mathsf{true}\}\ \mathtt{alloc}(n)\ \{x.\ x \neq 0 * \mathsf{uninit}(x) * \cdots * \mathsf{uninit}(x + n - 1)\}$$

The following tow rules are for atomic initialisation. In the precondition P must hold as changing an atomic location to a particular state requires the state interpretation to be satisfied.

[INITIALISATION-1...2]

$$\frac{P \Rightarrow \tau(s, v)}{\{\mathsf{uninit}(\ell) * P\}\ [\ell]_{\mathtt{rel}} := v\ \left\{\boxed{\ell : s \mid \tau}\right\}}$$

$$\frac{P \Rightarrow \tau(s, v)}{\{\mathsf{uninit}(\ell) * \langle P \rangle\}\ [\ell]_{\mathtt{rlx}} := v\ \left\{\boxed{\ell : s \mid \tau}\right\}}$$

[CONSEQUENCE-RULE]

$$\frac{P' \Rrightarrow P \qquad \{P\}\ e\ \{x.\ Q\} \qquad \forall x.\ Q \Rrightarrow Q'}{\{P'\}\ e\ \{x.\ Q'\}}$$

[FRAME-RULE]

$$\frac{\{P\}\ e\ \{x.\ Q\}}{\{P * R\}\ e\ \{x.\ Q * R\}}$$

Essentially, the following rules states that the special assertions are not to be nested. Nesting special assertions is

problematic as it may introduce things violate the exquisite design of the whole system. For instance, assuming we allow an assertion in the form of $\boxtimes\langle P\rangle$, it immediately becomes shareable, $\langle P\rangle$, after an acquire fence which does not have the releasing semantics. Therefore, we prevent such nesting from the resource model level and these inference rules are corollaries of our resource model design. Note that the annotation (e.g. a) used in these rules can be any valid label (for restricted shareable assertions) or nothing (for unrestricted shareable assertions).

[ASSERTION-PROPERTY-1…7]

$$\Box\langle P\rangle_a \Rightarrow \texttt{false} \text{ if } \texttt{EMP} \notin [\![P]\!]^\rho$$
$$\boxtimes\langle P\rangle_a \Rightarrow \texttt{false} \text{ if } \texttt{EMP} \notin [\![P]\!]^\rho$$
$$\boxtimes\boxtimes P \Rightarrow \texttt{false} \text{ if } \texttt{EMP} \notin [\![P]\!]^\rho$$
$$\langle\boxtimes P\rangle_a \Rightarrow \texttt{false} \text{ if } \texttt{EMP} \notin [\![P]\!]^\rho$$
$$\Box\boxtimes P \Rightarrow \texttt{false} \text{ if } \texttt{EMP} \notin [\![P]\!]^\rho$$
$$\langle P\rangle_a * \langle Q\rangle_a \Leftrightarrow \langle P * Q\rangle_a$$
$$\langle\langle P\rangle_{a_1}\rangle_{a_2} \Rightarrow \texttt{false} \text{ if } \texttt{EMP} \notin [\![P]\!]^\rho$$

[PURE-REDUCTION-AXIOM-1…2]

$$\{\text{true}\}\ v\ \{x.\ x=v\} \qquad \{\text{true}\}\ v==v'\ \{x.\ x=1 \Leftrightarrow v=v'\}$$

We also have the following rules for conditional statements, let-binding, fork expression, and the repeat loop. The fork rule states that given e can be safely executed from the precondition Q, we can fork a new thread with the precondition $\{P * Q\}$ to execute e and leaving only P to the parent thread.

[CONDITIONAL]

$$\frac{\{P * v \neq 0\}\ e_1\ \{x.Q\} \quad \{P * v = 0\}\ e_2\ \{x.Q\}}{\{P\}\ \texttt{if}\, v\, \texttt{then}\, e_1\, \texttt{else}\, e_2\ \{x.Q\}}$$

[LET-BINDING]

$$\frac{\{P\}\ e\ \{x.\ Q\} \quad \forall x.\ \{Q\}\ e'\ \{y.\ R\}}{\{P\}\ \texttt{let}\, x = e\, \texttt{in}\, e'\ \{y.\ R\}}$$

[FORK]

$$\frac{\{Q\}\ e\ \{\text{true}\}}{\{P * Q\}\ \texttt{fork}\, e\ \{P\}}$$

[REPEAT]

$$\frac{\{P\}\ e\ \{x.\ (x = 0 \wedge P) \vee (x \neq 0 \wedge Q)\}}{\{P\}\ \texttt{repeat}\, e\, \texttt{end}\ \{x.\ Q\}}$$

V. CASE STUDIES

In this section, we first demonstrate our logic with an illustrative example using a release-sequence to pass messages between three threads. Then, we further illustrate the power of our logic by using it to verify a readers-writer-lock implementation where both the release-sequence and concurrent reads are involved.

A. AN ILLUSTRATIVE EXAMPLE

In Fig. 11 we show a message passing program. In this example, the initial values for x and y are both 0. In the first thread, the message x is set to be 42 then y is set to be 1. As the write operation to y is a release write, it initiates a release-sequence that contains the following relaxed write and may contain the CAS in the second thread. In the third thread,

```
[x]rlx := 42;  ||  CASacq,rel(y, 1, 2)  ||  repeat [y]acq end;
[y]rel := 1;   ||                       ||  r := [x]rlx
[y]rlx := 3;   ||                       ||  {r = 42}
```

FIGURE 11. Message passing using release-sequence.

y is repeatedly checked until a non-zero value is observed. Then the message x is examined. Note that, for readability we use $x = e_1; e_2$ as an equivalent expression for the command $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ (or simply $e_1; e_2$ if the evaluation result of e_1 is not used in e_2). For the same reason, we use $||$ to separate the threads forked.

We assert that at the end of the execution, the reading of x must return the new value 42. Intuitively, this is because for the third thread to exit the loop, a non-zero value y must be observed, which can only be the result from one of the writes in the release-sequence led by the release write to y in the first thread. Therefore a synchronisation is formed between the release write to y and the acquire read of y, ensuring the information about $x = 42$ is available when the their thread reads the value of x. To formally reason about this procedure, the protocols for x and y must be defined first. We call x's states $\mathbf{x}_o$ (the initial state) and $\mathbf{x}_n$ (the new state). Its protocol $\mathbf{P}_x$ allows one possible state transition: $\mathbf{x}_o \sqsubseteq_{\mathbf{P}_x} \mathbf{x}_n$. The state interpretations can be defined as:

$$\mathbf{P}_x(s, v, t, c) \triangleq s = \mathbf{x}_n \wedge v = 42 \wedge t = 1 \wedge c = 0,$$

which states that thread 1 is allowed to change x to state $\mathbf{x}_n$ by writing 42 to it and it is not necessary to be a CAS.

There are four states for y: $\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2$ and $\mathbf{y}_3$ and the following transitions are permitted:

$$\mathbf{y}_0 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_1, \mathbf{y}_1 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_2, \mathbf{y}_1 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_3, \text{ and } \mathbf{y}_2 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_3.$$

The state interpretations are defined as:

$$\begin{aligned}\mathbf{P}_y(s, v, t, c) \triangleq\ & s = \mathbf{y}_1 \wedge v = 1 \wedge t = 1 \wedge c = 0 \wedge \Box\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x}\\ \vee\ & s = \mathbf{y}_2 \wedge v = 2 \wedge t = 2 \wedge c = 1 \wedge \Box\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x}\\ \vee\ & s = \mathbf{y}_3 \wedge v = 3 \wedge t = 1 \wedge c = 0 \wedge \Box\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x},\end{aligned}$$

which indicates what values, threads, and the CAS indicators are needed to move y to a corresponding state. Most importantly, the interpretations also specify that the stores must have the knowledge $\Box\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x}$ at hand before the actions can be taken. Therefore, when the acquire load in the third thread reads from any one of them, the knowledge about x can be retrieved.

The proof of the program is illustrated in Fig. 12. As the threads start with observing x and y in their initial states. In the first thread, the relaxed store to x moves x to its new state thus we have $\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x}$ in (1.2). This resource is essential for the next command to be performed as it is required to know $\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x}$ in the state interpretation of $\mathbf{y}_1$. With $\boxed{x : \mathbf{x}_n}\boxed{\mathbf{P}_x}$ at hand, the release write to y can be performed and moves y to the state $\mathbf{y}_1$. Moreover, according

$(1.1)\{\boxed{x:\mathbf{x}_o \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_0 \mid \mathbf{P}_y}\}$
$[x]_{\texttt{rlx}} := 42;$
$(1.2)\{\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_0 \mid \mathbf{P}_y}\}$
$[y]_{\texttt{rel}} := 1;$
$(1.3)\left\{\begin{array}{l}\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_1 \mid \mathbf{P}_y} \\ * \langle \boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} \rangle_{\mathbf{y}_1}\end{array}\right\}$
$[y]_{\texttt{rlx}} := 3;$
$(1.4)\{\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_3 \mid \mathbf{P}_y}\}$

$(2.1)\{\boxed{x:\mathbf{x}_o \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_0 \mid \mathbf{P}_y}\}$
$\texttt{CAS}_{\texttt{acq,rel}}(y, 1, 2);$
$(2.2)\left\{\begin{array}{l} z = 0 \wedge \boxed{x:\mathbf{x}_o \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_0 \mid \mathbf{P}_y} \vee \\ z.\ z = 1 \wedge \boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_2 \mid \mathbf{P}_y} \vee \\ z = 0 \wedge \boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_3 \mid \mathbf{P}_y}\end{array}\right\}$

$(3.1)\{\boxed{x:\mathbf{x}_o \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_0 \mid \mathbf{P}_y}\}$
$\texttt{repeat}\ [y]_{\texttt{acq}}\ \texttt{end};$
$(3.2)\{\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_1 \mid \mathbf{P}_y}\}$
$r := [x]_{\texttt{rlx}};$
$(3.3)\left\{\begin{array}{l}\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x} * \boxed{y:\mathbf{y}_1 \mid \mathbf{P}_y} \\ \wedge\ r = 42\end{array}\right\}$

FIGURE 12. Verification of the message passing program.

```
[x]rlx := 42;  ||  [y]rlx := 2  ||  repeat [y]acq end;
[y]rel := 1;   ||               ||  r := [x]rlx
[y]rlx := 3;   ||               ||  {r = 42?}
```

FIGURE 13. A broken release-sequence.

to the [RELEASE-STORE] rule and rules about knowledge it can make a restricted-shareable cope of $\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x}$, which can be used by the relaxed store that follows. When processing the relaxed store to y, the [RELAXED-STORE-2] rule is applied. The release-sequence validity check will success as the only state that can be in the middle of the release head $\mathbf{y}_1$ and the target state $\mathbf{y}_3$ is $\mathbf{y}_2$, a state can only be reached with a CAS operation, and CAS does not interrupt a release-sequence.

In different scheduling, the CAS from the second thread may find y to be in state $\mathbf{y}_0$, $\mathbf{y}_1$, or $\mathbf{y}_3$ before its execution. The CAS only success if it observes $\mathbf{y}_1$. But even when it fails, the protocol still holds. According to the [ACQ-REL-CAS] rule, the CAS operation's postcondition can be derived as shown in (2.2).

In the third thread, first y is repeatedly read. According to the [REPEAT] rule and the definitions of $\mathbf{P}_y$, exiting the loop needs y is at least at state $\mathbf{y}_1$ as that is denoted in (3.2). According to the [ACQUIRE-LOAD] rule, some common knowledge can be retrieved from the state interpretation, which is $\boxed{x:\mathbf{x}_n \mid \mathbf{P}_x}$ in this case. Therefore, when x is read in the last step, it is guaranteed to return the latest value 42.

Our verification system can also detect possible interruptions in a release-sequences and will not allow the verification to go through. This is illustrated in the following example shown in Fig. 13, where the CAS operation in the second thread is changed to a relaxed store.

For the new program, state transitions and interpretations for y have to be changed to:

$$\mathbf{y}_0 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_1 \wedge \mathbf{y}_0 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_2 \wedge \mathbf{y}_1 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_2\ \wedge$$
$$\mathbf{y}_1 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_3 \wedge \mathbf{y}_2 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_1 \wedge \mathbf{y}_2 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_3 \wedge \mathbf{y}_3 \sqsubseteq_{\mathbf{P}_y} \mathbf{y}_2$$
$$\mathbf{P}_y(s, v, t, c) \triangleq$$
$$s = \mathbf{y}_1 \wedge v = 1 \wedge t = 1 \wedge c = 0 \wedge \Box \boxed{x:\mathbf{x}_n \mid \mathbf{P}_x}$$
$$\vee\ s = \mathbf{y}_2 \wedge v = 2 \wedge t = 2 \wedge c = 0$$
$$\vee\ s = \mathbf{y}_3 \wedge v = 3 \wedge t = 1 \wedge c = 0 \wedge \Box \boxed{x:\mathbf{x}_n \mid \mathbf{P}_x}$$

If we attempt to apply [RELAXED-STORE-2] to the command $[y]_{\texttt{rlx}} := 3$ and change y to state $\mathbf{y}_3$ using the restricted-shareable resource obtained from release head $\mathbf{y}_1$, the validity check for the release-sequence (the second premise) will fail as according y's protocol definition, a non-CAS state $\mathbf{y}_2$ from a different thread may be interrupting. The verification will fail as we would have expected.

```
new() ≜
  x = alloc(2);
  [x.data]na := 0;
  [x.count]rel := 0;
  x

read(x) ≜
  repeat
    repeat r1 = [x.count]rlx; r1 ≠ N end;
    CASrlx,rlx(x.count, r1, r1 + 1)
  end;
  fenceacq;
  v = [x.data]na;
  repeat
    r2 = [x.count]rlx;
    CASrlx,rel(x.count, r2, r2 − 1)
  end;
  v

write(x, v) ≜
  repeat
    repeat r = [x.count]rlx; r == 0 end;
    CASrlx,rlx(x.count, 0, N)
  end;
  fenceacq;
  [x.data]na := v;
  [x.count]rel := 0
```

FIGURE 14. A readers-writer-lock.

B. VERIFYING THE READERS-WRITER-LOCK

In this section, we use our reasoning system to verify the readers-writer-lock implementation shown in Fig. 14. Note that, this lock has a bounded capacity **N**, i.e. it allows at most **N** readers (or one writer) to access the protected non-atomic data field at a time. Note also that, for readability we use the following field offsets: $x.\texttt{data} \triangleq x + 0$, which is the location of the data field; and $x.\texttt{count} \triangleq x + 1$, which refers to a counter that keeps track of the number of active players (a reader is counted as 1, while a writer is solely counted as **N**). Once the lock is created and initialised by the `new()` function, a reader can atomically increase the counter by 1

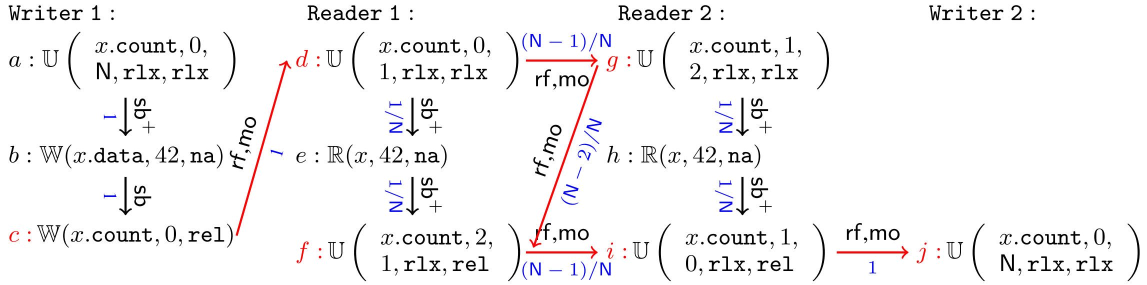


FIGURE 15. An execution of readers-writer-lock.

(when there are vacancies available, i.e. $r_1 \neq \mathsf{N}$ in Fig. 14[2]) to inform other players that there is an active reader, which will prevent a writer from obtaining the lock. After using the shared resource, the reader relinquishes the reader's lock by atomically decrease the counter by 1. A writer waits until the counter is 0 (which indicates that there is no other reader nor writer); then it atomically set the counter to **N** indicating that the shared resource is fully occupied; then it can safely modify the non-atomic data. When finishing, the writer releases the lock by setting the counter back to 0.

Our readers-writer-lock allows concurrent reads. To verify such an algorithm, we use fractional permissions. The idea is to divide the permission to access $x.\mathtt{data}$ into **N** pieces ($1/\mathsf{N}$ each). Correspondingly, the value of the counter represents how many pieces of the fractional permissions have been distributed. Atomically increasing the counter by 1, a reader gains one piece of $1/\mathsf{N}$ permission, which is sufficient for it to perform the read action. However, a writer will have to update the counter from 0 to **N** to retrieve the full permission that consists of all of the **N** pieces. When releasing the lock, the permissions go back to the invariant (or protocol, in our terminology). This design is demonstrated in the execution graph shown in Fig. 15, which is annotated with the permissions transferred along the execution. In this particular execution, `Writer 1` first sets $x.\mathtt{data}$ to 42 after obtaining the full permission. Then this information is released together with the full ownership of the protected data by the release write c. In fact, c initiates a release-sequence, from which the two readers both retrieve one piece of the $1/\mathsf{N}$ permission to access $x.\mathtt{data}$ and read the value 42. The rest of the permission goes to `Writer 2` when j reads from the release-sequence (which is underlined): $\underline{c \xrightarrow{\mathsf{mo}} d \xrightarrow{\mathsf{mo}} g \xrightarrow{\mathsf{mo}} f \xrightarrow{\mathsf{mo}} i} \xrightarrow{\mathsf{rf}} j$. The two $1/\mathsf{N}$ permissions assigned to the readers are also transferred to `Writer 2` via release-sequences: $\underline{f \xrightarrow{\mathsf{mo}} i} \xrightarrow{\mathsf{rf}} j$ and $\underline{i} \xrightarrow{\mathsf{rf}} j$. Thereupon, `Writer 2` can write to $x.\mathtt{data}$ freely.

Now we embed this idea into the definitions of the counter's protocol $\mathbf{P}_c$ and prove that our algorithm works as intended while the protocol is preserved. Firstly, we choose $\mathbf{c}_{i,j}$ as the counter's states, where i tracks how many pieces of the fractional permissions have been issued so far and j represents the number of the fractional permissions that have been returned. As the capacity of our lock is **N**, we require that for any valid state $i \in [j, j+\mathsf{N}]$ holds. For valid states, the state transitions are defined as $\mathbf{c}_{i,j} \sqsubseteq_{\mathbf{P}_c} \mathbf{c}_{i+1,j}$ (when the counter is increased and new permission is issued) and $\mathbf{c}_{i,j} \sqsubseteq_{\mathbf{P}_c} \mathbf{c}_{i,j+1}$ (when the counter is decreased and some permission is returned). The state interpretation is defined as below:

$$\mathbf{P}_c(s, v, t, u) \triangleq \; s = \mathbf{c}_{i,j} \wedge v = i - j \wedge (i = j \vee i > j \wedge u = 1) \\ * \exists v'.\, x.\mathtt{data} \overset{(\mathsf{N}-v)/\mathsf{N}}{\longmapsto} v'.$$

This definition specifies that at state $\mathbf{c}_{i,j}$, we have $x.\mathtt{count} = i - j$. Both CASes and atomic writes can change $x.\mathtt{count}$ to 0 ($i = j$); however, we must use a CAS ($u = 1$) to change $x.\mathtt{count}$ to other states (where $i < j$). Most importantly, at state $\mathbf{c}_{i,j}$ we must ensure that there is $(\mathsf{N} - (i - j))/\mathsf{N}$ permission under the guard of the protocol. This enables a player to retrieve some permission when it increases the counter (move i forward) and enforces a player to return permission when it decrease the counter (move j forward). With these preparations, we can verify our readers-writer-lock algorithm. Firstly, we demonstrate in Fig. 16 that the `new()` function prepares the lock invariant.

$$
\begin{array}{l}
\mathtt{new}() \triangleq \\
(1)\{\mathtt{emp}\} \\
\quad x = \mathtt{alloc}(2); \\
(2)\{x.\mathtt{data} = \mathtt{uninit} * x.\mathtt{count} = \mathtt{uninit}\} \\
\quad [x.\mathtt{data}]_{\mathtt{na}} := 0; \\
(3)\left\{x.\mathtt{data} \overset{1}{\mapsto} 0 * x.\mathtt{count} = \mathtt{uninit}\right\} \\
\quad [x.\mathtt{count}]_{\mathtt{rel}} := 0; \\
(4)\{\boxed{x.\mathtt{count} : \mathbf{c}_{0,0}}\boxed{\mathbf{P}_c}\} \\
\quad x \\
(5)\{z.\ z = x \wedge \boxed{x.\mathtt{count} : \mathbf{c}_{0,0}}\boxed{\mathbf{P}_c}\}
\end{array}
$$

FIGURE 16. Verifying the initialisation of a Readers-Writer-Lock.

Then, as shown in Fig. 17, a reader begins with some (maybe dated) knowledge of the counter. It repeatedly reads from the counter until it reads some value that is not equal to **N**, which indicates that the lock is not fully occupied. According to the protocol, we can deduce that at state (2) r_2 is actually smaller than **N**. This is critical for our reasoning, as when the reader increases the counter in the next step, we will have to know it would not bring the counter over the bound and break the protocol. Therefore, after a

[2] $v_1 \neq v_2 \triangleq (v_1 == v_2) == 0$

$$
\begin{array}{l}
\mathtt{read}(x) \triangleq \\
(1)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_0,j_0}}\boxed{\mathbf{P}_c}\} \\
\quad \mathtt{repeat} \\
\quad\quad \mathtt{repeat}\ r_1 = [x.\mathtt{count}]_{\mathtt{rlx}};\ r_1 \neq \mathsf{N}\ \mathtt{end}; \\
(2)\{\exists \mathbf{c}_{i_1,j_1}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_1,j_1}}\boxed{\mathbf{P}_c} \wedge r_1 = i_1 - j_1 < \mathsf{N}\} \\
\quad\quad \mathtt{CAS}_{\mathtt{rlx},\mathtt{rlx}}(x.\mathtt{count}, r_1, r_1+1) \\
(3)\left\{ z_1.\ \begin{array}{l} \exists \mathbf{c}_{i_2,j_2}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} \wedge \\ (z_1 = 0 \wedge (i_2 - j_2 \neq i_1 - j_1)) \vee \\ (z_1 = 1 \wedge i_2 = i_1 + 1 \wedge j_2 = j_1 * \\ \exists v'.\ \boxtimes\, x.\mathtt{data} \overset{1/\mathsf{N}}{\mapsto} v') \end{array} \right\} \\
\quad \mathtt{end}; \\
(4)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} * \boxtimes x.\mathtt{data} \overset{1/\mathsf{N}}{\mapsto} v')\} \\
\quad \mathtt{fence}_{\mathtt{acq}}; \\
(5)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} * x.\mathtt{data} \overset{1/\mathsf{N}}{\mapsto} v')\} \\
\quad v = [x.\mathtt{data}]_{\mathtt{na}}; \\
(6)\{v = v' \wedge \boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} * x.\mathtt{data} \overset{1/\mathsf{N}}{\mapsto} v')\} \\
\quad \mathtt{repeat} \\
\quad\quad r_2 = [x.\mathtt{count}]_{\mathtt{rlx}}; \\
(7)\left\{ \begin{array}{l} v = v' \wedge \exists \mathbf{c}_{i_3,j_3}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_3,j_3}}\boxed{\mathbf{P}_c} \wedge \\ r_2 = i_3 - j_3 > 0 * x.\mathtt{data} \overset{1/\mathsf{N}}{\mapsto} v') \end{array} \right\} \\
\quad\quad \mathtt{CAS}_{\mathtt{rlx},\mathtt{rel}}(x.\mathtt{count}, r_2, r_2-1) \\
(8)\left\{ z_2.\ \begin{array}{l} v = v' \wedge \exists \mathbf{c}_{i_4,j_4}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_4,j_4}}\boxed{\mathbf{P}_c} \wedge \\ (z_2 = 1 \wedge i_4 = i_3 \wedge j_4 = j_3 + 1) \vee \\ (z_2 = 0 \wedge (i_4 - j_4 \neq i_3 - j_3 * x.\mathtt{data} \overset{1/\mathsf{N}}{\mapsto} v') \end{array} \right\} \\
\quad \mathtt{end}; \\
(9)\{v = v' \wedge \boxed{x.\mathtt{count} : \mathbf{c}_{i_4,j_4}}\boxed{\mathbf{P}_c}\} \\
\quad v \\
(10)\{z_3.\ z_3 = v = v' \wedge \boxed{x.\mathtt{count} : \mathbf{c}_{i_4,j_4}}\boxed{\mathbf{P}_c}\}
\end{array}
$$

FIGURE 17. Verifying the Reader's Lock.

successful CAS that increases the counter by 1, the reader exits the loop knowing that the protocol is preserved and a fraction of the ownership of $x.\mathtt{data}$ is retrieved as shown in state (4). At this stage, the resource retrieved is *waiting-to-be-acquired*, as the CAS itself has only relaxed memory order for loading. An acquire fence turns the resource to its normal form at (5). Then $x.\mathtt{date}$ can be read according to the [NON-ATOMIC-LOAD] rule. When unlocking, the reader first gets the latest value of the counter (in state (7)). As it have a fractional ownership of $x.\mathtt{data}$ at hand, we can deduce that the environment cannot change the counter to 0, which requires that all the fractions of $x.\mathtt{data}$'s ownership to be returned. This idea is also formalised in the [RELAXED-LOAD] rule, according to which we can only read states whose interpretation is compatible with the resource we currently hold. Thus, we know that the counter's latest value must be greater than 0 and can be safely decreased by 1 using a *release* CAS when we return the fractional permission (state (9)). At last, we return the value read from $x.\mathtt{data}$.

The verification of the writer's program is shown in Fig. 18. A writer's lock can only be acquired when the value of the counter is 0. When it reads 0 from the counter, it starts attempting to update the counter to **N** using CAS. When the CAS successes, the full ownership of $x.\mathtt{data}$ can be retrieved according the protocol and the [RELAXED-LOAD] rule (in state (3)). Then, an acquire fence makes the

$$
\begin{array}{l}
\mathtt{write}(x, v) \triangleq \\
(1)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_0,j_0}}\boxed{\mathbf{P}_c}\} \\
\quad \mathtt{repeat} \\
\quad\quad \mathtt{repeat}\ r = [x.\mathtt{count}]_{\mathtt{rlx}};\ r == 0\ \mathtt{end}; \\
(2)\{\exists \mathbf{c}_{i_1,j_1}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_1,j_1}}\boxed{\mathbf{P}_c} \wedge r = i_1 - j_1 = 0\} \\
\quad\quad \mathtt{CAS}_{\mathtt{rlx},\mathtt{rlx}}(x.\mathtt{count}, 0, \mathsf{N}) \\
(3)\left\{ z_1.\ \begin{array}{l} \exists \mathbf{c}_{i_2,j_2}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} \wedge \\ (z_1 = 0 \wedge (i_2 - j_2 \neq 0) \vee \\ (z_1 = 1 \wedge i_2 = i_1 + \mathsf{N} \wedge j_2 = j_1 * \\ \exists v'.\ \boxtimes\, x.\mathtt{data} \overset{1}{\mapsto} v') \end{array} \right\} \\
\quad \mathtt{end}; \\
(4)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} * \boxtimes x.\mathtt{data} \overset{1}{\mapsto} v'\} \\
\quad \mathtt{fence}_{\mathtt{acq}}; \\
(5)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} * x.\mathtt{data} \overset{1}{\mapsto} v'\} \\
\quad [x.\mathtt{data}]_{\mathtt{na}} := v; \\
(6)\{\boxed{x.\mathtt{count} : \mathbf{c}_{i_2,j_2}}\boxed{\mathbf{P}_c} * x.\mathtt{data} \overset{1}{\mapsto} v\} \\
\quad [x.\mathtt{count}]_{\mathtt{rel}} := 0 \\
(7)\{\exists \mathbf{c}_{i_3,j_3}.\ \boxed{x.\mathtt{count} : \mathbf{c}_{i_3,j_3}}\boxed{\mathbf{P}_c} \wedge i_3 = i_2 = j_3 = j_2 + \mathsf{N}\}
\end{array}
$$

FIGURE 18. Verifying the Writer's Lock.

waiting-to-be-acquired resource locally available before it can be changed to the new value v that is given in the parameters. Releasing the writer's lock is easier than releasing a reader's lock. As the writer owns the full permissions to the protected data (state (6)), it knows that the environment cannot change the counter to another state (which requires to add or remove fractional permissions from the protocol) during the time it holes the lock. Therefore, the writer can simply use a release write to change the counter back to 0 and release the full ownership of the data.

VI. THE RESOURCE BASED SEMANTIC MODEL

With memory weakening behaviours, it is difficult to model the semantics of C11 programs based on a single piece of sequentially consistent heap shared by all threads. Instead, GPS and GPS+ use resources and resource triples respectively to depict the computational states of the threads and, based on the concept of resource, GPS and GPS+ develop an instrumented semantics to bridge the program logics and the event/machine level semantics similar to the one we have defined in §II. In this section we first give a brief introduction to our predecessors' semantic models, on top of which we can introduce our new resource-map model that is much more expressive and enables us to depict the behaviours of C11 release-sequences.

A. RESOURCES AND RESOURCE TRIPLES

In the GPS logic, resources are used to logically describe computational states. A resource $r \in Resource$ is a triple combined with a *physical location map*, a *ghost identity map*, and a set of *known escrows*: (Π, g, Σ). For a non-atomic location, the *physical location map* Π maps it to some value $\mathtt{na}(V)$; for an atomic location, Π maps it to a trace of states governed by the corresponding protocol $\mathtt{at}(\tau, S)$; and there are also infinite uninitialised locations are mapped to $\perp$. Similarly, the ghost identity map tracks the ghost values for our auxiliary variables. All established escrows are recorded

in the known escrow set. Resources form a PCM with composition $\oplus$. Some useful definitions are:

$$\texttt{emp} \triangleq ((\lambda n.\ \bot), (\lambda \mu.\ \lambda n.\ \epsilon_\mu), \varnothing)$$
$$r \leq r' \triangleq \exists r''.\ r \oplus r'' = r'$$
$$\lfloor r \rfloor \triangleq \{r \oplus r' \mid r' \in Resource\}$$

With resources defined, the semantics for a GPS proposition can be defined as a set of resources, i.e., $[\![P]\!]^\rho \subseteq Resource$, where ρ is a term interpretation assumed for protocol states and other PCM terms. Note that, GPS propositions depict the lower bound of the states, that is: $\forall r \in [\![P]\!]^\rho.\ \forall r' \# r.\ r \oplus r' \in [\![P]\!]^\rho$.

To support the reasoning about C11 fences, in GPS+ we introduced two new types of assertions, i.e. shareable assertions and waiting-to-be-acquired assertions. Correspondingly, its semantic representation is lifted from the resource based model to a model based on resource triples. A resource triple $(r_\mathsf{L}, r_\mathsf{S}, r_\mathsf{A})$ combines three resources to represent the locally available resource, shareable resource, and the resource retrieved from other threads that is waiting for an acquire action to merge it into the local resource.

B. NEW RESOURCE MAP BASED MODEL

In this work, to deal with the even subtler restricted sharable assertions, we extend the underlying resource model in GPS/GPS+ logic to a more expressive *label indexed resource map*:

$$\mathcal{R} \in ResMap \triangleq Label \rightarrow Resource$$

where $Label \triangleq \{\mathsf{L}, \mathsf{S}, \mathsf{A}\} \cup \mathbb{S}$ and $\mathbb{S}$ is the domain of the atomic locations' states. There are three special labels, $\mathcal{R}(\mathsf{L})$, $\mathcal{R}(\mathsf{S})$, and $\mathcal{R}(\mathsf{A})$, which respectively represent the *local, (unrestricted) shareable* and the *waiting-to-be-acquired* resources. For a resource indexed by a state, $\mathcal{R}(s)$ ($s \in \mathbb{S}$), it represents the resource that is made shareable at state s. The idea behind resource maps is to partition the resource available to a thread into fragments and label them for easier manipulation. When putting the resources under all the different labels together, we should get a meaningful resource back. Therefore, we define that a resource map $\mathcal{R}$ is well-formed if the sum of all its components, i.e. $\bigoplus_{l \in Label} \mathcal{R}(l)$ (or $\bigoplus \mathcal{R}$ for short), is defined.

Similar to the resources introduced in the GPS logic, the *ResMap* is also a PCM. Its composition operation $\oplus$ is point-wisely lifted from the resource composition operations. We say two *ResMap*s are compatible $\mathcal{R}\#\mathcal{R}'$ if we have $\mathcal{R} \oplus \mathcal{R}'$ defined. There is an identity element $\texttt{EMP}$, which represents empty for all labels: $\forall l.\ \texttt{EMP}(l) = \texttt{emp}$. Like that in the GPS logic, propositions in GPS++ are lower bounds of the described states:

$$\forall \mathcal{R} \in [\![P]\!]^\rho.\ \forall \mathcal{R}' \# \mathcal{R}.\ \mathcal{R} \oplus \mathcal{R}' \in [\![P]\!]^\rho$$

The interpretations for GPS++ propositions are also lifted to the *ResMap*-based model. To check if a resource map $\mathcal{R}$ satisfies a "basic" proposition P (an assertion that does not contain knowledge, shareable, or waiting-to-be-acquired parts) only its local component is concerned. For instance, $\mathcal{R} \in [\![\ \boxed{\ell : s \mid \tau}\ ]\!]^\rho \Leftrightarrow \exists S.\ \mathcal{R}(\mathsf{L}).\Pi(\ell) = \texttt{at}(\tau, S) \wedge s \in S$. For knowledge, (restricted/unrestricted) shareable, and waiting-to-be-acquired assertions, the resources under particular labels are picked out and checked:

$$\mathcal{R} \in [\![\square P]\!]^\rho \Leftrightarrow |\texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{L})]| \in [\![P]\!]^\rho$$
$$\mathcal{R} \in [\![\langle P \rangle]\!]^\rho \Leftrightarrow \texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{S})] \in [\![P]\!]^\rho$$
$$\mathcal{R} \in [\![\langle P \rangle_s]\!]^\rho \Leftrightarrow \texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(s)] \in [\![P]\!]^\rho$$
$$\mathcal{R} \in [\![\boxtimes P]\!]^\rho \Leftrightarrow \texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{A})] \in [\![P]\!]^\rho$$

Note that the stripping operation on a resource map $|\mathcal{R}|$ is a point-wise lifted GPS stripping, that is:

$$|\mathcal{R}| \triangleq \mathcal{R}' \wedge \forall l.\ \mathcal{R}'(l) = |\mathcal{R}(l)|.^3$$

Note also that, $\mathcal{R}[l \mapsto r]$ represents a new resource map that is generated from $\mathcal{R}$ by updating $\mathcal{R}(l)$ to r. The third definition gives the semantics for our newly introduced restricted-shareable-assertions. Intuitively, it states that $\mathcal{R}$ satisfies $\langle P \rangle_s$ if the resource indexed by its s label contains the information needed by P. The semantics for other special assertions is defined in a similar way. By defining the semantics like this, we can securely rule out the undesirable nesting of special assertions. For instance, one can not construct a $\mathcal{R}$ that satisfies $\boxtimes\langle P \rangle$ where P is not $\texttt{emp}$.

Composed propositions like the separating conjunction are straightly lifted to the resource map based model:

$$\mathcal{R} \in [\![P_1 * P_2]\!]^\rho \Leftrightarrow \exists \mathcal{R}_1, \mathcal{R}_2.\ \mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2 \ \wedge \ \mathcal{R}_1 \in [\![P_1]\!]^\rho \wedge \mathcal{R}_2 \in [\![P_2]\!]^\rho$$

The semantics for other types of assertions is left in appendix.

1) MODELLING PHYSICAL LOCATIONS

In GPS/GPS+ $\texttt{na}(V)$ is used to represent a non-atomic resource with value V assigned to a non-atomic location in the physical location map Π. A $\texttt{na}(V)$ resource is not duplicable as it is only compatible with the empty resource $\bot$. As we now support fractional permissions to non-atomic locations, we use $\texttt{na}(V, f)$ to represent a non-atomic resource where $f \in [0, 1]$. We have the composition of non-atomic resource defined as:

$$\texttt{na}(V, f) \oplus \bot \triangleq \texttt{na}(V, f)$$
$$\texttt{na}(V, f_1) \oplus \texttt{na}(V, f_1) \triangleq \texttt{na}(V, f_1 + f_2) \text{ if } f_1 + f_2 \in (0, 1]$$

Following GPS/GPS+, an atomic location's value is modelled as $\texttt{at}(\tau, S)$ instead of some concrete values. The τ is the protocol the location follows, and S is a trace (totally ordered set) of states it has gone through. One atomic value is compatible with another if they follow the same protocol and the union of their traces remains to be a well-formed trace. For atomic values, we have following shorthand definitions:

$$\texttt{at}(\tau, S) \sqsubseteq_\tau \texttt{at}(\tau, S') \triangleq \forall s \in S.\ \exists s' \in S'.\ s \sqsubseteq_\tau s'$$
$$\pi \equiv_\tau \pi' \triangleq \pi \sqsubseteq_\tau \pi' \wedge \pi' \sqsubseteq_\tau \pi$$

α	$\mathcal{R}_{\text{rely}}$	$\mathcal{R}_{\text{rely}} \in \text{rely}(\mathcal{R}, \alpha)$ if
$\mathbb{R}(\ell, V, \mathtt{na})$	$\mathcal{R}$	$\mathcal{R}(\mathsf{L})[\ell] = \mathtt{na}(V', f) \Rightarrow V = V' \wedge f \in (0, 1]$
$\mathbb{R}(\ell, V, \mathtt{rlx})$	$\mathcal{R}'$	$\exists r_{\mathsf{rf}}.\ \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow r_{\mathsf{rf}} \in \mathsf{envMv}(\mathcal{R}, \ell, V) \wedge \mathcal{R}'(\mathsf{A}) = \mathcal{R}(\mathsf{A}) \oplus \lvert r_{\mathsf{rf}} \rvert$ $\wedge\, \mathcal{R}'(\mathsf{L}) = \mathcal{R}(\mathsf{L})[\,\ell := r_{\mathsf{rf}}[\ell]] \wedge \forall l \neq \mathsf{L} \vee \mathsf{A}.\ \mathcal{R}'(l) = \mathcal{R}(l)$
$\mathbb{R}(\ell, V, \mathtt{acq})$	$\mathcal{R}'$	$\exists r_{\mathsf{rf}}.\ \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow r_{\mathsf{rf}} \in \mathsf{envMv}(\mathcal{R}, \ell, V) \wedge \forall l \neq \mathsf{L}.\ \mathcal{R}'(l) = \mathcal{R}(l) \wedge \mathcal{R}'(\mathsf{L}) = \mathcal{R}[\mathsf{L}] \oplus \lvert r_{\mathsf{rf}} \rvert$
$\mathbb{W}(\ell, V, \mathtt{at})$	$\mathcal{R}$	$\mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow \exists V'.\ \mathsf{envMv}(\mathcal{R}, \ell, V') \neq \varnothing$
$\mathbb{U}(\ell, V, V', \mathtt{acq}, \mathtt{at})$	$\mathcal{R}'$	$\exists r_{\mathsf{rf}}.\ \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow r_{\mathsf{rf}} \in \mathsf{envMv}(\mathcal{R}, \ell, V) \wedge \forall l \neq \mathsf{L}.\ \mathcal{R}'(l) = \mathcal{R}(l) \wedge \mathcal{R}'(\mathsf{L}) = \mathcal{R}(\mathsf{L}) \oplus r_{\mathsf{rf}}$
$\mathbb{U}(\ell, V, V', \mathtt{rlx}, \mathtt{at})$	$\mathcal{R}'$	$\exists r_{\mathsf{rf}}.\ \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow r_{\mathsf{rf}} \in \mathsf{envMv}(\mathcal{R}, \ell, V) \wedge \mathcal{R}'(\mathsf{A}) = \mathcal{R}(\mathsf{A}) \oplus r_{\mathsf{rf}}$ $\wedge\, \mathcal{R}'(\mathsf{L}) = \mathcal{R}(\mathsf{L})[\,\ell := r_{\mathsf{rf}}[\ell]] \forall l \neq \mathsf{L} \vee \mathsf{A}.\ \mathcal{R}'(l) = \mathcal{R}(l)$
otherwise	$\mathcal{R}$	always

FIGURE 19. Rely conditions for actions.

Intuitively, the first definition depicts the fact that the atomic value $\mathtt{at}(\tau, S')$ is newer than $\mathtt{at}(\tau, S)$. In practice, if a thread holding $\mathtt{at}(\tau, S)$ for an atomic location ℓ discovers that the environment has changed their copy of ℓ to $\mathtt{at}(\tau, S')$; then it may update ℓ to the new value as well. The second definition depicts the situation that two states under the same protocol are mutually transferable. This usually happens when two unsynchronised threads are going to change a same atomic location to different states. Therefore, the two states must be defined as mutually transferable to cope the unpredictable scheduling.

A physical location can also be uninitialised uninit or unallocated $\perp$.

2) GHOST MOVES

Similar to that in GPS/GPS+, the semantics for our ghost moves are given by resource-level ghost moves:

$$\rho \models P \Rrightarrow Q \triangleq \forall \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \Rrightarrow [\![Q]\!]^{\rho}.$$

The difference is that resource maps are now used at the resource-level for greater expressiveness. For example, our new ghost move [UNSHARE-R] $\langle P \rangle_s \Rrightarrow P$ is validated by the following resource-level rule:

$$\frac{\mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}] \oplus r \quad l \in \{\mathsf{S}\} \cup \mathbb{S} \quad \mathcal{R}'[l] \oplus r = \mathcal{R}[l] \quad \forall l' \neq \mathsf{L} \vee l.\ \mathcal{R}'[l'] = \mathcal{R}[l']}{\mathcal{R} \Rrightarrow \lfloor \mathcal{R}' \rfloor}$$

Intuitively, it states that a resource map $\mathcal{R}$ can move some resource r from one of its shareable components (unrestricted shareable component L or restricted shareable component indexed by one of the labels in $\mathbb{S}$) back to its local component.

The resource-level ghost move rules are designed to avoid modifying physical states; therefore, the ghost moves are guaranteed to only change auxiliary/logical computation states. The rest of the resource-level ghost move rules are left in appendix.

3) RELY/GUARANTEE DEFINITIONS

Following GPS/GPS+, we provide an instrumented semantics for all actions allowed in our model in a rely/guarantee style. But unlike GPS/GPS+ instrumented semantics which are built on resources or resource triples, our actions manipulate more expressive resource maps, which gives us the flexibility to depict the subtle differences between all kinds of actions including their interactions in release-sequences. A full definition of the rely and guarantee conditions are presented in Fig. 19 and Fig. 20.

First we introduce the definition of environment moves $\mathsf{envMv}(\mathcal{R}, \ell, V)$, which depicts a set of resources that could be acquired[4] by reading the location ℓ with value V.

$$\begin{aligned} & r_{\mathsf{rf}} \in \mathsf{envMv}(\mathcal{R}, \ell, V) \\ & \quad \triangleq\ \exists \tau, s.\ \mathrm{EMP}[\mathsf{L} \mapsto r_{\mathsf{rf}}] \in \mathsf{interp}(\tau)(s, V)\ \wedge \\ & \qquad \forall l.\ \mathcal{R}(l) \# r_{\mathsf{rf}} \wedge \mathcal{R}(\mathsf{L})[\ell] \sqsubseteq_{\tau} r_{\mathsf{rf}}[\ell] \equiv_{\tau} \mathtt{at}(\tau, \{s\}) \end{aligned}$$

The resource that could be acquired is depicted by $\mathsf{interp}(\tau)(s, V)$, which is described by the state interpretation of the state read s, i.e. $[\![\tau(s, V)]\!]^{\rho}$. The $\mathcal{R}$ in the definition is the resource map the current thread holds. It is used to limit the values can be read as in a well-behaving environment it is impossible to perform a write operation that requires resources incompatible with $\mathcal{R}$.

For an action α and its precondition represented by $\mathcal{R}$, the rely condition $\mathsf{rely}(\mathcal{R}, \alpha)$ describes which kind of resource maps $\mathcal{R}_{\mathsf{rely}}$ (with potential environment changes taken under consideration) can be accepted for α to be safely executed. To help the readers understand the rely conditions defined in Fig. 19, we interpret them in an intuitive way below. First, a non-atomic read relies on its current resource map's local component to have the expect value to read and the permission for the designated memory location is greater than zero. A relaxed read relies on the condition that there is some environment move to be read from and then the resource retrieved can be put into the local resource map's waiting-to-be-acquired component. Meanwhile, an acquire read can put the retrieved resource directly into the resource map's local component. Note that only knowledge can be retrieved for read operations. The rely conditions for the atomic update operations are similar to the read operations with corresponding memory order, except that they can retrieve non-knowledge resource.

[4] Recall that resources can be transmitted when a thread reads an atomic location written by another thread.

A guarantee condition $\mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha)$ signifies that the action α guarantees to generate some resources ($\mathcal{R}_{\mathsf{sb}}, r_{\mathsf{rf}}$), in which the resource map $\mathcal{R}_{\mathsf{sb}}$ is left for its sb successors and the resource r_{rf} could be passed to its potential readers. Note that the action's precondition is represented by $\mathcal{R}_{\mathsf{pre}}$; however the action is actually executed with $\mathcal{R}$ due to the possible environment moves. We use `at` to represent atomic memory orders (`rel` or `rlx` for writing orders; `acq` or `rlx` for reading orders), when it is unnecessary to specify which particular order is concerned. Intuitively, a skip action guarantees that it gives nothing if some action tries to read from it; and leaves everything to its sb successor. An allocation action leaves its sb successor a resource map with newly allocated locations. The read actions guarantee that the resource map left for their sb successor contains some non-trivial information about the location read. A non-atomic write requires full permission for the target location (or the location is uninitialised) in $\mathcal{R}$ and leaves the location with updated value in the resource map for its sb successor. The guarantee conditions for relaxed and release write will be compared below. Atomic update operations have the same guarantee conditions with the write operations having the same memory order. For release fence, it can move resource between the resource map's local and shareable components; and the acquire fence may move resource between the resource map's local and waiting-to-be-acquired components.

With the support from resource map based model, our new rely/guarantee definitions can depict the subtleness of the C11 release-sequence behaviours. For instance, both relaxed and release write can send some information to their potential readers; but their guarantee definitions are very different. A relaxed write who changes the target location to state s can only send out the information made shareable by previous release fences or valid release heads, which is represented using s' in the following definition:

$$\mathsf{validS}(s) \triangleq \left\{ \begin{array}{l} s' \mid \exists \tau.\ s' \sqsubseteq_\tau s \wedge \mathsf{sameThread}(s, s') \wedge \\ \forall s''.\ s' \sqsubseteq_\tau s'' \sqsubseteq_\tau s \Rightarrow \mathsf{sameThread}(s, s'') \\ \vee \mathsf{isCAS}(s'') \end{array} \right\}$$

Therefore, from $\mathcal{R}'$ (the resource map after the writing takes effect), a relaxed write can take a r_{rf} that satisfies its state interpretation from the labels valid for it to share ($r_{\mathsf{rf}} \leq \bigoplus_{l \in \mathsf{validS}(s)} \mathcal{R}'(l)$), and leave everything else ($\mathcal{R}_{\mathsf{sb}}$) to its sb successor. Note that we ensure that the internal participation of $\mathcal{R}_{\mathsf{sb}}$ is consistent with $\mathcal{R}'$ (no fragment of resources is misplaced under a different label) by requiring: $\forall l.\ \mathcal{R}_{\mathsf{sb}}(l) \leq \mathcal{R}'(l)$. On the other hand, a release write can directly use the resource under its local label,[5] i.e. $r_{\mathsf{rf}} \leq \mathcal{R}'(\mathsf{L})$. In addition, a release write can make some resource (r_2) restricted shareable to the following writes in its release-sequence. The relaxed and release atomic updates are similar to the corresponding writing actions only that they can use the resource from their reading sources (r_{in}) to fulfil their target states' interpretations, as they can be always considered as a part of release-sequences.

[5] If a resource under a shareable label is demanded, it can be moved to local via ghost move first.

VII. SOUNDNESS

In this section, we formulate the soundness of our proposed program logic. As in GPS/GPS+, our reasoning framework is compositional. That is, triples can be proved individually and then a bigger proof can be generated by connecting the proved triples with the `let` and `fork` rules provided. To bridge the gap between the localised reasoning and the threads' global interactions and non-sequential-consistent behaviours, we formulate the notions of *local safety* and *global safety* and provide the soundness proof in both layers.

A. LOCAL SAFETY

The rely-guarantee reasoning [15], [29] is deeply rooted in the soundness of GPS++. As GPS/GPS+, we formulate *local safety* to indicate that given a thread's rely-condition respected by other threads' guarantee-conditions, it confirms to its own guarantee. However, resource maps are used as the base model in our proofs to capture the subtle C11 synchronisation features such as release-sequences.

Based on the rely and guarantee definitions we have introduced in the previous section, we define $\mathsf{LSafe}_n(e, \Phi)$ as the set of resource maps on which the command e can safely execute for n steps and end up with Φ, which is the interpretation of triple's postcondition with the return value filled in place holders, being satisfied:

$$\begin{array}{l} \mathcal{R} \in \mathsf{LSafe}_0(e, \Phi) \triangleq \ \text{always} \\ \mathcal{R} \in \mathsf{LSafe}_{n+1}(e, \Phi) \triangleq \ \text{If } e \in \mathit{Val} \text{ then } \mathcal{R} \Rrightarrow [\![\Phi(e)]\!]^\rho \\ \quad \textit{If } e = K[\mathtt{fork}\, e'] \text{ then} \\ \qquad \mathcal{R} \in \mathsf{LSafe}_n(K[0], \Phi) * \mathsf{LSafe}_n(e', \mathsf{true}) \\ \quad \text{If } e \xrightarrow{\alpha} e' \text{ then } \forall \mathcal{R}_F \# \mathcal{R}.\ \forall \mathcal{R}_{pre} \Rrightarrow \mathsf{rely}(\mathcal{R} \oplus \mathcal{R}_F, \alpha). \exists P'. \\ \qquad \mathcal{R}_{pre} \in [\![P']\!]^\rho \wedge \forall \mathcal{R}' \in [\![P']\!]^\rho.\ (\mathcal{R}_{pre}, \mathcal{R}') \in \mathsf{wpe}(\alpha) \\ \qquad \Longrightarrow \exists \mathcal{R}_{post}.\ (\mathcal{R}_{post} \oplus \mathcal{R}_F, -) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}', \alpha) \\ \qquad\quad \wedge\ \mathcal{R}_{post} \in \mathsf{LSafe}_n(e', \Phi) \end{array}$$

It is worth noting that with the possible environment moves taken under consideration, the expression e actually works on some $\mathcal{R}'$ that follows the action's rely condition. Note also that the wpe provides a sanity check to rule out the obvious problematic environment changes.

α	$(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}') \in \mathsf{wpe}(\alpha)$ if
$\mathbb{A}(\ell_1..\ell_n)$	$\forall i.\ 1 \leq i \leq n \Rightarrow \mathcal{R}'(\ell_i) = \bot$
$\mathbb{W}(\ell, -, \mathtt{at})$	$\mathcal{R}_{\mathsf{pre}}(\mathsf{L})[\ell] = \mathtt{at}(-) \wedge \mathcal{R}'(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow \exists \mathcal{R}_E \in \mathsf{envMv}(\mathcal{R}_{\mathsf{pre}}, \ell, -).\ \mathcal{R}_E(\mathsf{L})[\ell] = \mathcal{R}'(\mathsf{L})[\ell]$
$\mathbb{U}(\ell, -, -, -)$	$\mathcal{R}_{\mathsf{pre}}(\mathsf{L})[\ell] = \mathtt{at}(-) \Rightarrow \mathcal{R}'(\mathsf{L})[\ell] \equiv \mathcal{R}_{pre}(\mathsf{L})[\ell]$

Intuitively, the definitions above state that a memory allocation action will only allocation fresh locations; an atomic write may observe its target location at a state other than the state in its precondition, while this is not allowed for an atomic update action.

α	$(\mathcal{R}_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha)$ if
$\mathbb{S}$	$r_{\mathsf{rf}} = \mathtt{emp} \wedge \mathcal{R}_{\mathsf{sb}} = \mathcal{R}$
$\mathbb{A}(\ell..\ell')$	$r_{\mathsf{rf}} = \mathtt{emp} \wedge \forall l \neq \mathsf{L}.\ \mathcal{R}_{\mathsf{sb}}(l) = \mathcal{R}(l) \wedge \mathcal{R}(\mathsf{L}) = \mathcal{R}(\mathsf{L})[\ell..\ell' := \mathsf{uninit}]$
$\mathbb{R}(\ell, V, \mathtt{na})$	$r_{\mathsf{rf}} = \mathtt{emp} \wedge \mathcal{R}_{\mathsf{sb}} = \mathcal{R} \wedge \mathcal{R}(\mathsf{L}) = \mathtt{na}(-)$
$\mathbb{R}(\ell, V, \mathtt{at})$	$r_{\mathsf{rf}} = \mathtt{emp} \wedge \mathcal{R}_{\mathsf{sb}} = \mathcal{R} \wedge \mathcal{R}(\mathsf{L}) = \mathtt{at}(-)$
$\mathbb{W}(\ell, V, \mathtt{na})$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{emp} \wedge \mathcal{R}(\mathsf{L})[\ell] \in \{\mathsf{uninit}, \mathtt{na}(-, 1)\} \wedge$ $\forall l \neq \mathsf{L}.\ \mathcal{R}_{\mathsf{sb}}(l) = \mathcal{R}(l) \wedge \mathcal{R}_{\mathsf{sb}}(\mathsf{L}) = \mathcal{R}(\mathsf{L})[\ \ell := \mathtt{na}(V, 1)\ ]$
$\mathbb{W}(\ell, V, \mathtt{rlx})$	$\exists \tau, s, S, \mathcal{R}'.$ $\mathcal{R}'(\mathsf{L}) = \mathcal{R}(\mathsf{L})[\ell := \mathtt{at}(\tau, S \cup \{s\})] \wedge \forall l \neq \mathsf{L}.\ \mathcal{R}'(l) = \mathcal{R}(l)$ $\wedge\ r_{\mathsf{rf}} \leq \bigoplus_{l \in \mathsf{validS}(s)} \mathcal{R}'(l) \wedge \bigoplus \mathcal{R}_{\mathsf{sb}} \oplus r_{\mathsf{rf}} = \bigoplus \mathcal{R}' \wedge \mathtt{EMP}[\mathsf{L} \mapsto r_{\mathsf{rf}}] \in \mathsf{interp}(\tau)(s, V)$ $\wedge\ \forall l.\ \mathcal{R}_{\mathsf{sb}}(l) \leq \mathcal{R}'(l) \wedge \forall r_E \in \mathsf{envMv}(\mathcal{R}_{\mathsf{pre}}, \ell, -).\ r_E[\ell] \sqsubseteq_{\mathtt{at}} r_{\mathsf{rf}}[\ell]$ $\wedge\ \mathcal{R}_{\mathsf{pre}}(\mathsf{L})[\ell] \neq \bot \wedge (\mathcal{R}(\mathsf{L})[\ell] = \mathsf{uninit} \wedge S = \varnothing \vee \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$
$\mathbb{W}(\ell, V, \mathtt{rel})$	$\exists \tau, s, S, \mathcal{R}'.$ $\exists r_1, r_2.\ \mathcal{R}(\mathsf{L}) = r_1 \oplus r_2 \wedge \mathcal{R}'(\mathsf{L}) = r_1[\ell := \mathtt{at}(\tau, S \cup \{s\})]$ $\wedge\ \mathcal{R}'(s) = \mathcal{R}(s) \oplus r_2[\ell := \mathtt{at}(\tau, S \cup \{s\})] \wedge \forall l \notin \{\mathsf{L}, s\}.\ \mathcal{R}'(l) = \mathcal{R}(l)$ $\wedge\ r_{\mathsf{rf}} \leq \mathcal{R}'(\mathsf{L}) \wedge \bigoplus \mathcal{R}_{\mathsf{sb}} \oplus r_{\mathsf{rf}} = \bigoplus \mathcal{R}' \wedge \mathtt{EMP}[\mathsf{L} \mapsto r_{\mathsf{rf}}] \in \mathsf{interp}(\tau)(s, V)$ $\wedge\ \forall l.\ \mathcal{R}_{\mathsf{sb}}(l) \leq \mathcal{R}'(l) \wedge \forall r_E \in \mathsf{envMv}(\mathcal{R}_{\mathsf{pre}}, \ell, -).\ r_E[\ell] \sqsubseteq_{\mathtt{at}} r_{\mathsf{rf}}[\ell]$ $\wedge\ \mathcal{R}_{\mathsf{pre}}(\mathsf{L})[\ell] \neq \bot \wedge (\mathcal{R}(\mathsf{L})[\ell] = \mathsf{uninit} \wedge S = \varnothing \vee \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$
$\mathbb{U}(\ell, V, V', \mathtt{at}, \mathtt{rlx})$	$\exists \tau, s, S, r_{\mathsf{in}}, \mathcal{R}'.$ $\mathcal{R}'(\mathsf{L}) = \mathcal{R}(\mathsf{L})[\ell := \mathtt{at}(\tau, S \cup \{s\})] \wedge \forall l \neq \mathsf{L}.\ \mathcal{R}'(l) = \mathcal{R}(l)$ $\wedge\ \forall l.\ \mathcal{R}_{\mathsf{sb}}(l) \leq \mathcal{R}'(l) \wedge \bigoplus \mathcal{R}_{\mathsf{pre}} \oplus r_{\mathsf{in}} \leq \bigoplus \mathcal{R}' \wedge \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-)$ $\wedge\ r_{\mathsf{rf}} \leq \bigoplus_{l \in \mathsf{validS}(s)} \mathcal{R}'(l) \oplus r_{\mathsf{in}} \wedge \bigoplus \mathcal{R}_{\mathsf{sb}} \oplus r_{\mathsf{rf}} = \bigoplus \mathcal{R}' \wedge \mathtt{EMP}[\mathsf{L} \mapsto r_{\mathsf{rf}}] \in \mathsf{interp}(\tau)(s, V)$ $\wedge\ \mathcal{R}_{\mathsf{pre}}(\mathsf{L})[\ell] \neq \bot \wedge (\mathcal{R}(\mathsf{L})[\ell] = \mathsf{uninit} \wedge S = \varnothing \vee \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$
$\mathbb{U}(\ell, V, V', \mathtt{at}, \mathtt{rel})$	$\exists \tau, s, S, r_{\mathsf{in}}, \mathcal{R}'.$ $\exists r_1, r_2.\ \wedge \mathcal{R}(\mathsf{L}) = r_1 \oplus r_2 \wedge \mathcal{R}'(\mathsf{L}) = r_1[\ell := \mathtt{at}(\tau, S \cup \{s\})]$ $\wedge\ \mathcal{R}'(s) = \mathcal{R}(s) \oplus r_2[\ell := \mathtt{at}(\tau, S \cup \{s\})] \wedge \forall l \notin \{\mathsf{L}, s\}.\ \mathcal{R}'(l) = \mathcal{R}(l)$ $\wedge\ \forall l.\ \mathcal{R}_{\mathsf{sb}}(l) \leq \mathcal{R}'(l) \wedge \bigoplus \mathcal{R}_{\mathsf{pre}} \oplus r_{\mathsf{in}} \leq \bigoplus \mathcal{R}' \wedge \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(-)$ $\wedge\ r_{\mathsf{rf}} \leq \mathcal{R}'(\mathsf{L}) \oplus r_{\mathsf{in}} \wedge \bigoplus \mathcal{R}_{\mathsf{sb}} \oplus r_{\mathsf{rf}} = \bigoplus \mathcal{R}' \wedge \mathtt{EMP}[\mathsf{L} \mapsto r_{\mathsf{rf}}] \in \mathsf{interp}(\tau)(s, V)$ $\wedge\ \mathcal{R}_{\mathsf{pre}}(\mathsf{L})[\ell] \neq \bot \wedge (\mathcal{R}(\mathsf{L})[\ell] = \mathsf{uninit} \wedge S = \varnothing \vee \mathcal{R}(\mathsf{L})[\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$
$\mathbb{F}(\mathtt{rel})$	$r_{\mathsf{rf}} = \mathtt{emp} \wedge \forall l \neq \mathsf{L} \vee \mathsf{S}.\ \mathcal{R}_{\mathsf{sb}}(l) = \mathcal{R}(l) \wedge \mathcal{R}_{\mathsf{sb}}(\mathsf{L}) \oplus \mathcal{R}_{\mathsf{sb}}(\mathsf{S}) = \mathcal{R}(\mathsf{L}) \oplus \mathcal{R}(\mathsf{S})$
$\mathbb{F}(\mathtt{acq})$	$r_{\mathsf{rf}} = \mathtt{emp} \wedge \forall l \neq \mathsf{L} \vee \mathsf{A}.\ \mathcal{R}_{\mathsf{sb}}(l) = \mathcal{R}(l) \wedge \mathcal{R}_{\mathsf{sb}}(\mathsf{L}) \oplus \mathcal{R}_{\mathsf{sb}}(\mathsf{A}) = \mathcal{R}(\mathsf{L}) \oplus \mathcal{R}(\mathsf{A})$

FIGURE 20. Guarantee conditions for actions.

As that in GPS and GPS+, we formulate the local soundness definition as:

$$\rho \vDash \{P\}\ e\ \{x.Q\} \triangleq \forall n, \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho).$$

The local soundness provides semantics for our Hoare triples. It states that starting from any computation state $\mathcal{R}$ in the triple's precondition P, it is safe for the expression e to execute as many steps as necessary; and when e terminates, we can expect that all its possible result states satisfies the triple's postcondition $\lambda x.\ [\![Q]\!]^\rho$, where x is e's return value.

Ghost moves and corollary inference rules also play an important role in our reasoning system. To validate the (local) soundness of our reasoning system, we first demonstrate the correctness of our ghost move and corollary inference rules introduced in §IV.

Our reasoning system is featured with new corollary inference rules (in the form of $P \Rightarrow Q$), namely [SEPARATION-R], [SEPARATION-F], [SEPARATION-1…2], and [KNOWLEDGE-MANIPULATION-1…7], to deal with the newly introduced types of assertions. These rules' correctness is ensured by the enhanced resource model and is formalised in Corollary 1 (whose proof is left in the appendix).

Corollary 1 (Soundness of Corollary Inference Rules): Our corollary inference rules are semantically sound. That is, given an inference rule allowing $P \Rightarrow Q$, we have $\exists \mathcal{R}.\ \lfloor \mathcal{R} \rfloor \cap [\![P]\!]^\rho \subseteq [\![Q]\!]^\rho$

In our reasoning system we support ghost moves depicted by rules [UNSHARE-R] and [GHOST-MOVE-1…8]. A ghost move is a transition that only modifies auxiliary/logical computation states. This is ensured by the resource-level ghost moves and formalised in Corollary 2 (whose proof is left in the appendix).

Corollary 2 (Soundness of Corollary Inference Rules): Our ghost move rules are semantically sound. That is, given a ghost move rule allowing $P \Rrightarrow Q$, we have $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q]\!]^\rho$.

Then we formalise the two of our structural rules below.

Theorem 1 (Consequence Rule): Given $\rho \vDash P' \Rrightarrow P$, $\rho \vDash \{P\}\ e\ \{x.\ Q\}$, and $\forall x.\ \rho \vDash Q \Rrightarrow Q'$, we can prove that $\rho \vDash \{P'\}\ e\ \{x.\ Q'\}$.

Proof: From the first premise of the theorem, we have the following property $\forall \mathcal{R} \in [\![P']\!]^\rho.\ \mathcal{R} \Rrightarrow [\![P]\!]^\rho$.

From the second premise of the theorem, we have the following property $\forall n, \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho)$.

According to [GHOST-MOVE-3] rule (ghost transitive rule), we have the proof obligation transformed to the form

(1) $\forall n, \mathcal{R} \in [\![P']\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho)$. So far, the precondition strenthning is proven.

In (1), we choose an arbitrarily large n, and unfold LSafe by its definition. For an e that is terminating, the proof obligation can be reduced to $\forall n, \mathcal{R} \in [\![P']\!]^\rho.\ \mathcal{R} \Rrightarrow \lambda x.\ [\![Q]\!]^\rho$. From the third premise, we have $\forall x.\ \forall \mathcal{R} \in [\![Q]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q']\!]^\rho$. By putting them together the consequence rule is proven. □

Theorem 2 (Frame Rule): Given $\rho \vDash \{P\}\ e\ \{x.\ Q\}$, we have $\rho \vDash \{P * R\}\ e\ \{x.\ Q * R\}$.

Proof: From the premise of the theorem, we have the following property: $\forall n, \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho)$.

With this, we are going to prove the term that $\forall n, \mathcal{R} \in [\![P * R]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho) * [\![R]\!]^\rho$.

According to the definition of separation assertions, the proof obligation can be transformed into:

$$\forall n, \mathcal{R}_1, \mathcal{R}_2.\ \mathcal{R}_1 \# \mathcal{R}_2 \wedge \mathcal{R}_1 \in [\![P]\!]^\rho \wedge \mathcal{R}_2 \in [\![R]\!]^\rho \Rightarrow \mathcal{R}_1 \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho) \wedge \mathcal{R}_2 \in [\![R]\!]^\rho$$

By simplification, we can translate the formula above into the form $\forall n, \mathcal{R}_1 \in [\![P]\!]^\rho.\ \mathcal{R}_1 \Rrightarrow \mathsf{LSafe}_n(e, \lambda x.\ [\![Q]\!]^\rho)$, which matches the premise.

The frame rules is proven. □

Finally, we formalise the local soundness of our reasoning system as shown below.

Theorem 3 (Local Soundness): Our verification logic is locally sound. That is, if $\{P\}\ e\ \{x.\ Q\}$ is provable, then for all closing ρ we have $\rho \vDash \{P\}\ e\ \{x.\ Q\}$.

Proof: The expression e may be a single expression or a series of expressions connected by let-binding. We first prove that given a single expression e, our reasoning rules are locally sound. Then we prove by structural induction that our reasoning system is locally sound for e with arbitrary layers of let-binding.

For the rule [PURE-REDUCTION-1], where e is a value v (or an arithmetic term that results in v), we are going to prove that $\forall n, \mathcal{R} \in [\![\mathsf{true}]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(v, \lambda x.\ [\![x = v]\!]^\rho)$. The case $n = 0$ holds trivially. In the case that $n > 0$, as $e \in Val$ we have: $\mathcal{R} \Rrightarrow \mathsf{LSafe}_n(v, \lambda x.\ [\![x = v]\!]^\rho) \triangleq \mathcal{R} \Rrightarrow \lambda x.\ [\![x = v]\!]^\rho\ v$, which can be derived from $\mathcal{R} \in [\![\mathsf{true}]\!]^\rho$ that is given by the precondition. The proof for [PURE-REDUCTION-1] is finished.

For the rule [PURE-REDUCTION-2], where e is a relational statement $v == v'$, we are going to prove that for all n and $\mathcal{R} \in [\![\mathsf{true}]\!]^\rho$:
$\mathcal{R} \Rrightarrow \mathsf{LSafe}_n((v == v'), \lambda x.\ [\![x = 1 \Leftrightarrow v = v']\!]^\rho)$. According to the event-step rules in our semantics, the expression $v == v'$ will be evaluated as 1 if $v = v'$ and 0 otherwise. Therefore, in the case $v = v'$ we have that the term $\mathcal{R} \Rrightarrow \mathsf{LSafe}_n((v == v'), \lambda x.\ [\![x = 1 \Leftrightarrow v = v']\!]^\rho)$ is semantically equivalent to $\mathcal{R} \in [\![1 = 1 \Leftrightarrow v = v']\!]^\rho)$, which can be further reduced to $\mathcal{R} \in [\![\mathsf{true}]\!]^\rho$ that is given by the precondition. Similarly, in the case $v \neq v'$, we have that the term $\mathcal{R} \Rrightarrow \mathsf{LSafe}_n((v == v'), \lambda x.\ [\![x = 1 \Leftrightarrow v = v']\!]^\rho)$ is semantically equivalent to $\mathcal{R} \in [\![0 = 1 \Leftrightarrow \mathtt{false}]\!]^\rho)$ which can be derived from $\mathcal{R} \in [\![\mathsf{true}]\!]^\rho$ that is given by the precondition. The proof for [PURE-REDUCTION-2] is finished.

For the [FORK] rule, where $e = \mathtt{fork}\, e'$, we are going to prove that $\forall n, \mathcal{R} \in [\![P * Q]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n((\mathtt{fork}\, e'), [\![P]\!]^\rho)$, with the premise $\forall n, \mathcal{R} \in [\![Q]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e', [\![\mathsf{true}]\!]^\rho)$. The case $n = 0$ is trivial. In the case $n > 0$, according to the definition of local safety, the proof obligation

$$\mathcal{R} \in \mathsf{LSafe}_n((\mathtt{fork}\, e'), [\![P]\!]^\rho)$$

is equivalent to the

$$\mathcal{R} \in \mathsf{LSafe}_{n-1}(0, [\![P]\!]^\rho) * \mathsf{LSafe}_{n-1}(e', [\![\mathsf{true}]\!]^\rho),$$

and this formula can be further reduced to the following form according to the definitions of the local safety: $\mathcal{R} \in [\![P]\!]^\rho * \mathsf{LSafe}_{n-1}(e', [\![\mathsf{true}]\!]^\rho)$. According to the definition of separation assertions we have:

$$\forall \mathcal{R} \in [\![P * Q]\!]^\rho.\ \exists \mathcal{R}', \mathcal{R}''.\ \mathcal{R} = \mathcal{R}' \oplus \mathcal{R}'' \wedge \mathcal{R}' \in [\![P]\!]^\rho \wedge \mathcal{R}'' \in [\![Q]\!]^\rho.$$

By putting together with the premise, we have

$$\forall \mathcal{R} \in [\![P * Q]\!]^\rho.\ \exists \mathcal{R}', \mathcal{R}''.\ \mathcal{R} = \mathcal{R}' \oplus \mathcal{R}'' \wedge \mathcal{R}' \in [\![P]\!]^\rho \wedge \mathcal{R}'' \in \mathsf{LSafe}_{n-1}(e', [\![\mathsf{true}]\!]^\rho),$$

which implies the proof obligation. The proof for [FORK] is finished.

For the rules: [ALLOCATION], [INITIALISATION-1 …2], [ACQUIRE-LOAD], [RELAXED-LOAD], [NON-ATOMIC-LOAD], [RELEASE-STORE], [RELAXED-STORE], [NON-ATOMIC-STORE], [ACQ-REL-CAS], [RLX-REL-CAS], [ACQ-RLX-CAS-1], [ACQ-RLX-CAS-2], [RLX-RLX-CAS-1], [RLX-RLX-CAS-2], [RELEASE-FENCE], and [ACQUIRE-FENCE], where e is allocation, initialisation, read, write, CASes, or fence. The event-step used would be $e \xrightarrow{\alpha} v$, where α could be $\mathbb{A}, \mathbb{R}, \mathbb{W}, \mathbb{U}$, or $\mathbb{F}$. We prove the soundness of the triple by unfolding the rely (Fig. 19), guarantee (Fig. 20), and wpe definitions corresponding to action α; then it is trivial to check that

$$\begin{aligned}
&\forall n > 0, \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, [\![x.Q]\!]^\rho)\\
\text{where}\quad &\mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e, [\![x.Q]\!]^\rho) \triangleq\\
&\forall \mathcal{R}_F \# \mathcal{R}.\ \forall \mathcal{R}_{pre} \Rrightarrow \mathsf{rely}(\mathcal{R} \oplus \mathcal{R}_F, \alpha).\\
&\exists P'.\ \mathcal{R}_{pre} \in [\![P']\!]^\rho \wedge \forall \mathcal{R}' \in [\![P']\!]^\rho.\\
&(\mathcal{R}_{pre}, \mathcal{R}') \in \mathsf{wpe}(\alpha) \Rightarrow\\
&\exists \mathcal{R}_{post}.\ (\mathcal{R}_{post} \oplus \mathcal{R}_F, -) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}', \alpha)\\
&\quad \wedge \mathcal{R}_{post} \in [\![\Phi(v)]\!]^\rho
\end{aligned}$$

We have also discussed that when $n = 0$, the local safety holds by definition. Therefore the aforementioned rules are locally sound.

For the case $e = \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ in the rule of [CONDITIONAL], we prove that:

$$\forall n, \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n((\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2), \lambda x.\ [\![Q]\!]^\rho).$$

In the case $v \neq 0$, we can add the pure assertion to triple's precondition according to the semantics for assertions, i.e. $\forall \mathcal{R}.\ \mathcal{R} \in [\![P]\!]^{\rho} \Rightarrow \mathcal{R} \in [\![P \wedge t \neq 0]\!]^{\rho} = [\![P * v \neq 0]\!]^{\rho}$; then the triple is validated by the first premise. Similarly, in the case of $v = 0$, the triple is validated by the second premise. The proof for [REPEAT] is finished.

For the [REPEAT] rule, where $e = \texttt{repeat}\ e' \texttt{end}$, we prove that $\forall n, \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n((\texttt{repeat}\ e'\ \texttt{end}), \lambda x.\ [\![Q]\!]^{\rho})$. In the case that e' is evaluated as some non-zero value v, The proof obligation is reduced to $\forall n, \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \Rrightarrow \lambda x.\ [\![Q]\!]^{\rho}$, which is validated by the premise. Otherwise according to the event-step semantics for repeat and conditional expressions, the definition of local safety, the rely/guarantee conditions for $\mathbb{S}$ action, this proof obligation can be reduced to that for all $n > 0$ and $\mathcal{R} \in [\![P]\!]^{\rho}$ $\mathcal{R} \Rrightarrow \mathsf{LSafe}_{n-1}((\texttt{repeat}\ e' \texttt{end}), \lambda x.\ [\![Q]\!]^{\rho})$ This transformation can be recursively performed until e' is evaluated as non-zero or it reaches LSafe_0, which holds trivially. The proof for [REPEAT] is finished.

For the [LET-BINDING] rule, where $e = (\texttt{let}\ x = e'\ \texttt{in}\ e'')$, we prove that:

$$\forall n, \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n((\texttt{let}\ x = e'\ \texttt{in}\ e''), \lambda y.\ [\![R]\!]^{\rho})$$

by using structural induction. We first prove triples with single layer let-binding, that is e' is one of the aforementioned expresses and does not contain let-binding,[6] as the base case.

For $e = (\texttt{let}\ x = \texttt{fork}\ e'\ \texttt{in}\ e'')$, we have following premises:

$$\rho \vDash \{P * Q\}\ \texttt{fork}\ e'\ \{x.\ P\} \text{ and } \rho \vDash \forall x.\ \{P\}\ e''\ \left\{y.\ P'\right\}.$$

We are going to prove that $\rho \vDash \{P * Q\}\ e''[0/x]\ \left\{y.\ P'\right\}$. From the first premise, we have:

$$\forall n, \mathcal{R} \in [\![P * Q]\!]^{\rho}.\ \mathcal{R} \in \mathsf{LSafe}_n(\texttt{fork}\ e', [\![x.\ P]\!]^{\rho}),$$

which can be unfolded to the following from according to the definitions of local safety:

$$\forall n, \mathcal{R} \in [\![P * Q]\!]^{\rho}.\ \mathcal{R} \in \mathsf{LSafe}_n([\![P[0/x]]\!]^{\rho}) * \mathsf{LSafe}_n(e', \mathsf{true}).$$

From the second premise, we have $\rho \vDash \forall x.\ \{P\}\ e''\ \left\{y.\ P'\right\}$ and thus:

$$\begin{aligned}\forall n, \mathcal{R}' \in [\![P[0/x]]\!]^{\rho}.\ \mathcal{R}' \in \mathsf{LSafe}_n(\texttt{let}\ x = 0\ \texttt{in}\ e'', [\![y.\ P']\!]^{\rho})\\ = \mathsf{LSafe}_n(K[0], [\![y.\ P']\!]^{\rho}).\end{aligned}$$

Therefore, we can derive that:

$$\begin{aligned}&\forall n, \mathcal{R} \in [\![P * Q]\!]^{\rho}.\\ &\mathcal{R} \in \mathsf{LSafe}_n(K[0], [\![y.\ P']\!]^{\rho}) * \mathsf{LSafe}_n(e', \mathsf{true}).\end{aligned}$$

For other single expressions let us assume that e' can be reduced to numerical value v. According to the event-step definition for let-binding and the corresponding rely/guarantee definitions, the proof obligation can be transformed into $\forall n, \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \Rrightarrow \mathsf{LSafe}_n(e''[v/x], \lambda y.\ [\![R]\!]^{\rho})$. At the same time the first premise can be simplified as $\forall \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \in [\![Q]\!]^{\rho}$; together with the second premise, the proof obligation is met.

Then we move on to the inductive case. Let us assume $\rho \vDash \{P\}\ e'\ \{x.\ Q\}$, where $e' = (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$, and prove $\rho \vDash \{P\}\ \texttt{let}\ x = e'\ \texttt{in}\ e''\ \{x.\ Q\}$. According to the event-step definition for evaluation context, we need to evaluate e' first. According to the assumption, e' can be savely executed as many steps as possible until it is reduced to a numerical value v. Then we have $\forall \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \in [\![Q]\!]^{\rho}$; together with the second premise $\forall n, v, \mathcal{R} \in [\![Q]\!]^{\rho}.\ \mathcal{R} \in \mathsf{LSafe}_n(e', \lambda y.\ [\![R]\!]^{\rho})$, the soundness of the triples:

$$\forall n, \mathcal{R} \in [\![P]\!]^{\rho}.\ \mathcal{R} \in \mathsf{LSafe}_n(e'[v/x], \lambda y.\ [\![R]\!]^{\rho})$$

is proven. □

B. GLOBAL SAFETY AND THE FINAL SOUNDNESS THEOREM

As our target programs assume a concurrent environment, in addition to their local safety, it is also necessary to demonstrate that given the triple $\{P\}e\{x.Q\}$ provable the executions of e are free from data races, memory errors, nor dangling reads under all possible threads interleaving. Therefore, we formulate the global soundness of the proposed program logic similar to its predecessors GPS/GPS+. However the new logic provides full support to C11 release-sequences, which makes it much trickier to prove some critical properties such as data-race-freedom.

Before we can formally define global soundness, we first provide definition for program execution (with an arbitrary number of steps) $\mathsf{execs}(e)$ and the semantics for C11 programs $[\![e]\!]$ in Fig. 21 on top of the machine-step semantics discussed earlier in §II.

The definitions indicate that the result of a program e is either some value that can be validated by a legal execution or an error state if e allows race conditions or memory errors in its execution.

With these preparations, we define the global soundness as:

$$\textit{if}\ \vdash \{\mathsf{true}\}\ e\ \{x.P\}\ \textit{then}\ [\![e]\!] \subseteq \{V \mid [\![P[V/x]]\!] \neq \varnothing\}.$$

Intuitively, this definition requires that a provable Hoare triple about a close program e must precisely predict the result of e regarding to its executions under the C11 memory model. To demonstrate the global soundness of our proposed Hoare triples, a property called *global safety* is defined as below:

$$\begin{aligned}&\mathsf{GSafe}_n(\mathcal{T}, \mathcal{G}, \mathcal{L}) \triangleq\\ &\mathsf{valid}(\mathcal{G}, \mathcal{L}, N) = N \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge\\ &\forall a \in N.\ \mathcal{L}(\mathsf{sb}, a, \bot) = \bigoplus\{\mathcal{R} \mid \exists i.\ \mathcal{T}_{ins}(i) = (a, -, \mathcal{R}, -)\} \wedge\\ &\forall i.\ \mathcal{T}_{ins}(i) = (a, e, \mathcal{R}, \Phi) \Longrightarrow \mathcal{R} \in \mathsf{LSafe}_n(e, \Phi)\\ &\text{where}\\ &N \triangleq \mathsf{dom}(\mathcal{G}.A) \wedge \mathcal{T}_{ins} \in \mathit{IThreadMap} \triangleq \{\mathbb{N} \rightarrow (a, e, \mathcal{R}, \Phi)\}\end{aligned}$$

[6] Technically speaking, composed expressions like conditional expression, fork expression and the repeat expression may contain let-binding. However, as long as they terminate with numerical values we can treat them as "single expressions".

$$\begin{array}{rcl} \mathsf{execs}(e) & \triangleq & \{(e', \mathcal{G}) \mid \langle [i \mapsto (\mathsf{start}, e)]; ([\mathsf{start} \mapsto \mathbb{S}], \varnothing, \varnothing, \varnothing, \varnothing)\rangle \longrightarrow^* \langle [i \mapsto (-, e') \uplus T]; \mathcal{G}\rangle\} \\ [\![e]\!] & \triangleq & \begin{cases} \mathbf{error} & \exists(-, \mathcal{G}) \in \mathsf{execs}(e).\ \mathsf{dataRace}(\mathcal{G}) \vee \mathsf{memErr}(\mathcal{G}) \\ \{V \mid (V, -) \in \mathsf{execs}(e)\} & \text{otherwise} \end{cases} \end{array}$$

FIGURE 21. Semantics for program execution.

We annotate/label the edges in our execution graphs with the computation resources they carry from one node to another and a labelling maps $\mathcal{L}$ is used to record that information. The $\mathcal{T}_{ins}$ is an instrumented thread pool, in which all threads are depicted in tuples like $(a, e, \mathcal{R}, \Phi)$. We have a as a thread's last generated event in the graph, e as the thread's continuation, $\mathcal{R}$ as the resource map the thread currently holds (representing the thread's computation state), and the postcondition expected after the execution of the thread is depicted by Φ. The instrumented thread pool $\mathcal{T}_{ins}$ can be down casted to a machine thread pool $\mathcal{T}$ (recall §II-B) using $\mathsf{erase}(\mathcal{T}_{ins})$, where $\forall i.\ \mathcal{T}_{ins}(i) = (a, e, \mathcal{R}, \Phi) \Rightarrow \mathsf{erase}(\mathcal{T})(i) = (a, e)$. The predicate valid signifies that a set of nodes and with their edges properly labelled. By requesting $N \triangleq \mathsf{dom}(\mathcal{G}.A)$, the global safety definition requires all nodes in the graph are properly labelled. The $\mathsf{compat}(\mathcal{G})$ indicates that for any group of hb-independent edges in the event graph $\mathcal{G}$, the sum of resources they carried is defined. We call a group of edges are hb-*independent* if for any pair of the edges (a, a') and (b, b') we have $\neg\mathsf{hb}^{=}(a', b)$. Note that, we only consider the compatibility of the resource maps' local components, which should be sufficient as the local components are the actual resources involved in computation. We also show in our proofs that when the resources under other labels merged to local, the compatibility keeps preserved. The conform states that the mo order for all atomic writes is consistent with the predefined state transitions.

Intuitively, the global safety definition $\mathsf{GSafe}_n(\mathcal{T}, \mathcal{G}, \mathcal{L})$ indicates that based on an event graph $\mathcal{G}$ and with the resource maps recorded in $\mathcal{L}$, it is safe for any thread from the thread pool $\mathcal{T}$ to execute n steps. We aim to demonstrate that this global safe property is to be preserved during the entire executions of legal programs allowed in our logic. To achieve this, we introduce the method we used to update the labelling $\mathcal{L}$ for the event graph when a new node (i.e., a new event) is added. The labelling process is then formalised into five lemmas which will demonstrate the restoration of the global safety for the event graph with the new node added. In this section, we focus to convey the high-level ideas about this process leaving the lemmas' proof and the detailed definitions to the appendix.

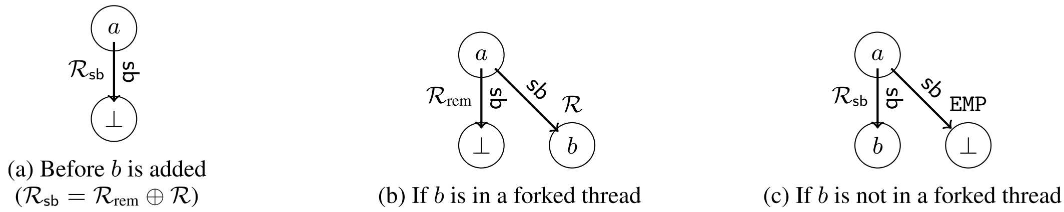
When adding a node the the event graph, its sb incoming edge is to be labelled first. Suppose that the node to be added is b and a is its sb predecessor in the event graph. Initially, node a's sb outgoing edge points to a sink node, i.e., $\mathsf{sb}(a, \bot)$, and is labelled with the resource map $\mathcal{R}_{\mathsf{sb}}$ which will be passed to a's sb successor. If a is followed by a fork command and b is the first event in the forked thread, part of the $\mathcal{R}_{\mathsf{sb}}$, namely $\mathcal{R}$, is taken and used to label the $\mathsf{sb}(a, b)$ edge, while the remaining resource $\mathcal{R}_{\mathrm{rem}}$ is left in a's sb sink edge for a's local thread. If there is no new thread involved and b is from the same thread as a, the entire $\mathcal{R}_{\mathsf{sb}}$ should be used to label $\mathsf{sb}(a, b)$. Note that we assume there is a $\mathbb{S}$ node with all its outgoing edges labelled as empty in the initial event graph; therefore if b is the first event generated in the program, for generality it will take that skip node as its sb predecessor. We illustrate this process in Fig. 22 and formalise it as the following lemma.

Lemma 1 (Step Preparation):

$$\begin{array}{rl} \textit{if} & \mathsf{consistentC11}(\mathcal{G}) \\ & \wedge \mathsf{consistentC11}(\mathcal{G}') \\ & \wedge \mathsf{dom}(\mathcal{G}'.A) = \mathsf{dom}(\mathcal{G}.A) \uplus b \\ & \wedge \mathcal{L}(\mathsf{sb}, a, \bot) = \mathcal{R} \oplus \mathcal{R}_{rem} \\ & \wedge \mathsf{dom}(\mathcal{G}.A) \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\ & \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \\ & \wedge \forall c \in \mathsf{dom}(\mathcal{G}.A).\ \mathcal{G}.A(c) = \mathcal{G}'.A(c) \\ & \wedge \mathcal{G}'.\mathsf{sb} = G.\mathsf{sb} \uplus [a, b) \\ & \wedge \forall c \in \mathsf{dom}(\mathcal{G}.A).\ \mathcal{G}.\mathsf{rf}(c) = \mathcal{G}'.\mathsf{rf}(c) \\ & \wedge \mathcal{G}'.\mathsf{mo} \supseteq \mathcal{G}.\mathsf{mo} \\ \textit{then} & \exists \mathcal{L}'.\ \mathsf{dom}(\mathcal{G}.A) = \mathsf{valid}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}.A)) \\ & \wedge \mathsf{compat}(\mathcal{G}', \mathcal{L}') \\ & \wedge \mathsf{conform}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}'.A)) \\ & \wedge \mathcal{L}'(\mathsf{sb}, a, \bot) = \mathcal{R}_{rem} \\ & \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{sb}) = \mathcal{R} \\ & \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{rf}) = \mathrm{EMP} \\ & \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{esc}) = \mathrm{EMP} \\ & \wedge \forall a' \neq a.\ \mathcal{L}'(\mathsf{sb}, a', b) = \mathrm{EMP} \\ & \wedge \mathsf{out}(\mathcal{L}', b, \mathsf{all}) = \mathrm{EMP} \\ & \wedge \forall a' \neq a.\ \mathcal{L}'(\mathsf{sb}, a', \bot) = \mathcal{L}(\mathsf{sb}, a', \bot) \end{array}$$

Note that the shorthand notations $\mathsf{in}(\mathcal{L}, a, \mathsf{t})$ and $\mathsf{out}(\mathcal{L}, a, \mathsf{t})$ correspondingly stand for the sum of resource maps labelled with a's incoming or outgoing edges of type t.

Next, the new node's rf incoming edge will be labelled. Note that this labelling process is for atomic reading actions and CASes. A non-atomic load simply returns the value recorded in its thread-local resource map. Meanwhile, an atomic load (or a CAS) is able to read from any writer with respect to the C11 memory model consistentC11. Initially, a writing event's rf outgoing resource, which can be referred to as r_{rf}, is associated to its rf sink edge. When the new node reads from that write, their rf edge is labelled following four different approaches according to the read event's type (atomic read or CAS) and the memory order used (relaxed



(a) Before b is added ($\mathcal{R}_{sb} = \mathcal{R}_{rem} \oplus \mathcal{R}$)
(b) If b is in a forked thread
(c) If b is not in a forked thread

FIGURE 22. Labelling event b's sb incoming edge.

or acquire). If the new event is a relaxed read, its rf incoming edge is labelled with a resource map $\mathtt{EMP}[\mathsf{A} \mapsto |r_{\mathsf{rf}}|]$, indicating that it can retrieve some knowledge from the write but the knowledge is "waiting-to-be-acquired". If the new event is an acquire read, the retrieved knowledge is directly put under the local label: $\mathtt{EMP}[\mathsf{L} \mapsto |r_{\mathsf{rf}}|]$. The labelling for the CASes is similar, but for CASes the information can be retrieved is not limited to knowledge. Note that, we always left $|r_{\mathsf{rf}}|$ in the writer's rf sink edge for other readers to read. This process is illustrated in Fig. 23. By labelling the new rf edge in this way, the following lemma can be proved.

Lemma 2 (Rely Step):

$$
\begin{aligned}
\textit{if}\quad & \mathcal{G}.A(a) = \alpha \\
& \wedge \mathsf{dom}(\mathcal{G}.A) = N \uplus a \\
& \wedge N \in \mathsf{prefix}(\mathcal{G}) \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) = \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathtt{EMP} \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) = \mathtt{EMP} \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{esc}) = \mathtt{EMP} \\
& \wedge \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathtt{EMP} \\
\textit{then}\quad & \exists \mathcal{L}'.\, N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}', N) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \mathsf{in}(\mathcal{L}', a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{rf}) \in \mathsf{rely}(\mathsf{in}(\mathcal{L}', a, \mathsf{sb}), \alpha) \\
& \wedge \mathsf{in}(\mathcal{L}', a, \mathsf{esc}) = \mathsf{out}(\mathcal{L}', a, \mathsf{all}) = \mathtt{EMP} \\
& \wedge \forall b, c.\, \mathcal{L}'(\mathsf{sb}, b, c) = \mathcal{L}(\mathsf{sb}, b, c) \\
& \wedge \forall b.\, \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot)
\end{aligned}
$$

When a piece of resource $\mathcal{R}$ is packed into an escrow by event a, $\mathcal{R}$ is removed from a's working resource map and put into a's esc sink edge for safe keeping. Another event b owning the resource $\mathcal{R}'$ that is required to open the escrow may retrieve $\mathcal{R}$ through a new escrow edge created associating a and itself. Then $\mathcal{R}'$ is dumped to b's escrow sink edge. This process (and other local ghost moves) is depicted in Fig. 24 and is formalised in the following lemma.

Lemma 3 (Ghost Step):

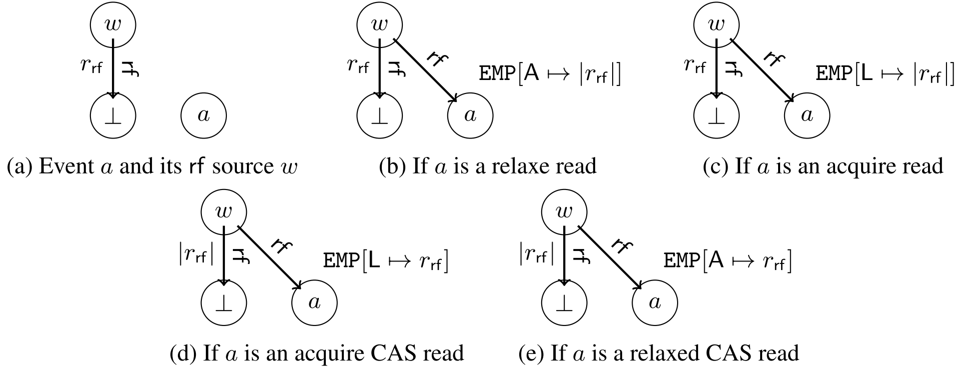
$$
\begin{aligned}
\textit{if}\quad & \mathsf{dom}(\mathcal{G}.A) = N \uplus a \wedge N \in \mathsf{prefix}(\mathcal{G}) \\
& \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}[(\mathsf{esc}, -, a, \bot) = \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathcal{R}_{before} \triangleq \mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{esc}) \\
& \wedge \mathcal{R}_{after} \triangleq \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}_{before} \Rrightarrow_{\mathcal{I}} \mathcal{R}_{after} \wedge |\mathcal{R}_{before}| \leq \mathcal{R} \wedge \mathcal{R} \Rrightarrow \mathcal{P} \\
& \wedge \forall c.\, \mathcal{L}(\mathsf{esc}, -, a, e) = \mathtt{EMP} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\, \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) = \\
& \bigoplus \left\{ \mathcal{R}_E \;\middle|\; \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathcal{R}_E \in \mathcal{Q}', \\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \\ (\nexists b.\, \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\} \\
\textit{then}\quad & \exists \mathcal{L}', \mathcal{I}', \mathcal{R}', \mathcal{R}'_{before}, \mathcal{R}'_{after} \in \mathcal{P}.\, N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}', \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}'[(\mathsf{esc}, -, a, \bot) := \mathcal{L}'(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}']) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}', N) \\
& \wedge \mathcal{R}'_{before} \triangleq \mathsf{in}(\mathcal{L}', a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{rf}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{esc}) \\
& \wedge \mathcal{R}'_{after} \triangleq \mathcal{R}' \oplus \mathsf{out}(\mathcal{L}', a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}'_{before} \Rrightarrow_{\mathcal{I}'} \mathcal{I}'\mathcal{R}'_{after} \\
& \wedge \forall b.\, \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot) \\
& \wedge \forall b.\, \mathcal{L}'(\mathsf{rf}, b, \bot) = \mathcal{L}(\mathsf{rf}, b, \bot) \\
& \wedge \forall b, c.\, \mathcal{L}'(\mathsf{sb}, b, c) = \mathcal{L}(\mathsf{sb}, b, c) \\
& \wedge \forall b, c.\, \mathcal{L}'(\mathsf{rf}, b, c) = \mathcal{L}(\mathsf{rf}, b, c) \\
& \wedge \forall c.\, \mathcal{L}'(\mathsf{esc}, -, a, e) = \mathtt{EMP} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}'.\, \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) = \\
& \bigoplus \left\{ \mathcal{R}_E \;\middle|\; \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}', \mathcal{R}_E \in \mathcal{Q}', \\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \\ (\nexists b.\, \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\}
\end{aligned}
$$

The following lemma demonstrates that by labelling the graph in the way we have described, the new node's all incoming resource map will satisfy its wpe requirements.

Lemma 4 (Protocol Equivalence for Writes):

$$
\begin{aligned}
\textit{if}\quad & \mathsf{dom}(\mathcal{G}.A) = N \uplus a \wedge N \in \mathsf{prefix}(\mathcal{G}) \\
& \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}[(\mathsf{esc}, -, a, \bot) \\
& := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge |\mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf})| \leq \mathcal{R} \\
\textit{then}\quad & (\mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}), \mathsf{in}(\mathcal{L}, a, \mathsf{all})) \in \mathsf{wpe}(\mathcal{G}.A(a))
\end{aligned}
$$

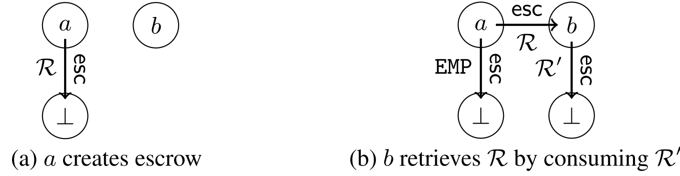
Finally, the new node's outgoing edges (sb and rf) will be labelled in guarantee step, using the corresponding resource



(a) Event a and its rf source w (b) If a is a relaxe read (c) If a is an acquire read

(d) If a is an acquire CAS read (e) If a is a relaxed CAS read

FIGURE 23. Labelling event a's rf incoming edge.



(a) a creates escrow (b) b retrieves $\mathcal{R}$ by consuming $\mathcal{R}'$

FIGURE 24. Labelling event b's escrow incoming edge.

map and resource $(\mathcal{R}_{\mathsf{sb}}, r_{\mathsf{rf}})$ derived from the action's guarantee definition. These resources are initially assigned to the corresponding sink edges of the new node, until the node's future sb and rf successors are added to the graph and take the resources for the labelling of the corresponding edges. Note that, as annotation for read-from sink edge, r_{rf}, is a resource instead of resource map, we require it to be compatible with other resource maps' local component at the compat checking.

Lemma 5 (Guarantee Step):

$$
\begin{aligned}
\textit{if}\quad & \mathcal{G}.A(a) = \alpha \wedge \mathsf{dom}(\mathcal{G}.A) = N \uplus a \wedge N \in \mathsf{prefix}(\mathcal{G})\\
& \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A))\\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}])\\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge \mathsf{consistentC11}(\mathcal{G})\\
& \wedge \mathcal{R}_{\mathsf{pre}} = \mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf})\\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond})\\
& \wedge \mathcal{R}_{\mathsf{pre}} \in \mathsf{rely}(-, \alpha) \wedge |\mathsf{in}(\mathcal{L}, a, \mathsf{all})| \leq \mathcal{R}\\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}'\\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) =\\
& \bigoplus \left\{ \mathcal{R}_E \;\middle|\; \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathcal{R}_E \in \mathcal{Q}',\\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'),\\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\}\\
& \wedge \mathsf{out}(\mathcal{L}, a, \mathsf{sb}) = \mathsf{out}(\mathcal{L}, a, \mathsf{rf}) = \mathtt{EMP}\\
& \wedge (\mathcal{R}_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha) \wedge \mathsf{wpe}(\alpha, \mathcal{R}_{\mathsf{pre}}, \mathsf{in}(\mathcal{L}, a, \mathsf{all}))\\
\textit{then}\quad & \exists \mathcal{L}'.\ \mathsf{dom}(\mathcal{G}.A) = \mathsf{valid}(\mathcal{G}, \mathcal{L}', \mathsf{dom}(\mathcal{G}.A))\\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}') \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}', \mathsf{dom}(\mathcal{G}.A))\\
& \wedge \forall b \neq a.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot) \wedge \mathcal{L}'(\mathsf{sb}, a, \bot) = \mathcal{R}_{\mathsf{sb}}
\end{aligned}
$$

In what follows we formulate the Theorem *Instrumented Execution*. Intuitively, this theorem states that is a program is globally safe to be executed for $n+1$ steps from the current machine configuration, an arbitrarily scheduled move will lead to a new machine configuration, based on which the rest of the program will be global-safely for another n steps.

Theorem 4 (Instrumented Execution): If we have $\mathsf{GSafe}_{n+1}(\mathcal{T}_{ins}, \mathcal{G}, \mathcal{L}) \wedge \langle \mathsf{erase}(\mathcal{T}_{ins}); \mathcal{G}\rangle \longrightarrow \langle \mathcal{T}'; \mathcal{G}'\rangle$ *then* $\exists \mathcal{T}'_{ins}, \mathcal{L}'.\ \mathsf{erase}(\mathcal{T}'_{ins}) = \mathcal{T}' \wedge \mathsf{GSafe}_n(\mathcal{T}'_{ins}, \mathcal{G}', \mathcal{L}')$.

Proof: Starting from $\mathsf{GSafe}_{n+1}(\mathcal{T}_{ins}, \mathcal{G}, \mathcal{L})$, a machine step transforms the graph into $\mathcal{G}'$ with a new event b and the thread pool into $\mathcal{T}'$, leaving n more locally safe steps for the active thread.

By applying Lemma 1, we have that there exists a labelling $\mathcal{L}_1$ derived from the original labelling $\mathcal{L}$ with $\mathsf{in}(\mathcal{L}_1, b, \mathsf{sb}) = \mathcal{R} \wedge \mathcal{L}_1(\mathsf{sb}, a, \bot) = \mathcal{R}_{\mathrm{rem}}$, which ensures:

$$
\begin{aligned}
& \mathsf{dom}(\mathcal{G}.A) = \mathsf{valid}(\mathcal{G}', \mathcal{L}_1, \mathsf{dom}(\mathcal{G}.A))\\
& \qquad \wedge \mathsf{compat}(\mathcal{G}', \mathcal{L}_1 \wedge \mathsf{conform}(\mathcal{G}', \mathcal{L}_1, \mathsf{dom}(\mathcal{G}'.A)).
\end{aligned}
$$

By applying Lemma 2, we have that there exists $\mathcal{L}_2$ updated from $\mathcal{L}_1$ with the new node's rfincoming label and maintains the valid set.

By applying Lemma 3, we have that there exists $\mathcal{L}_3$ updated from $\mathcal{L}_2$ with escrow incoming and outgoing edges taken under consideration. With the new labelling, the compat and conform properties are recovered.

By applying Lemma 4, we have that with $\mathcal{L}_3$, the sum of new event's incoming resource maps satisfies its wpe definitions.

By applying Lemma 5, we have that there is a labelling $\mathcal{L}'$ with the new event's outgoing edges updated. The valid, compat, and conform along with other properties are preserved for the new graph and labelling. Therefore, the new graph is $\mathsf{GSafe}_n(\mathcal{T}'_{ins}, \mathcal{G}', \mathcal{L}')$. □

Now we present one more lemma, whose proof is left in the appendix, to demonstrate that all possible executions are free from data-race, memory errors, and uninitialised reads.

Lemma 6 (Error Free): If we have $\mathsf{GSafe}_n(\mathcal{T}_{ins}, \mathcal{G}, \mathcal{L})$ *then* $\neg\mathsf{dataRace}(\mathcal{G})$*,*$\neg\mathsf{memErr}(\mathcal{G})$*, and there is no dangling reads.*

Ultimately we give the global soundness Theorem *Adequacy*.

Theorem 5 (Adequacy): If we have $\vdash \{\mathsf{true}\}\ e\ \{x.P\}$ *then* $[\![e]\!] \subseteq \{V \mid [\![P[V/x]]\!] \neq \varnothing\}$.

Proof: From $\vdash \{\mathsf{true}\}\ e\ \{x.P\}$, we can derive that from true the program e is locally safe for an arbitrarily large number of steps. We assume e terminates in n steps. That is, according to the step-level semantics e will be reduced to some pure value, which can be referred to as V, after n steps. We assert $\mathsf{LSafe}_{n+1}(e, [\![x.P]\!])$, based on which we construct:

$$\mathsf{GSafe}_{n+1}\begin{pmatrix} [0 \mapsto (\mathsf{start}, e, \mathtt{emp}, [\![x.P]\!])], \\ ([\mathsf{start} \mapsto \mathbb{S}], \varnothing, \varnothing, \varnothing), \\ [(\mathsf{sb}, \mathsf{start}, \bot) \mapsto \mathtt{EMP}] \uplus [(\mathsf{rf}, \mathsf{start}, \bot) \mapsto \mathtt{EMP}] \end{pmatrix}.$$

By repeatedly applying the Theorem *Instrumented Execution* for n times, we can derive:
$\exists \mathcal{T}'_{ins}, \mathcal{R}.\ \mathsf{GSafe}_1(\mathcal{T}'_{ins} \uplus [\,0 \mapsto (-,\ V, \mathcal{R},\ [\![\,x.P]\!])], -, -)$.
From this condition we can imply that $\mathcal{R} \Rrightarrow [\![P[V/x]]\!]$ and thus we can conclude that $[\![P[V/x]]\!] \neq \varnothing$. □

VIII. RELATED WORK

Our work is closely related to the GPS logic [6] and the RSL logic [5]. RSL can be used to reason about release-acquire synchronisations in the style of Concurrent Separation Logic (CSL) [16]. Following up work, FSL and FSL++ [8], [11], provide support to C11 fences and part of the release-sequence feature. GPS framework integrates ghost states, protocols and separation logic, which are the most handy tools for reasoning about concurrent program advocated by the state-of-the-art literature [15], [17]–[26]. Tassarotti *et al.* [7] demonstrated the power of GPS by apply it to real world Linux synchronisations algorithms. Following up work GPS+ [9], [10] provides support to C11 fences and based on which our work provides the full support to C11 release-sequences and fractional permissions.

There are some recent works aiming at providing automations in the verification of C11 weak memory programs. With the Iris [26] semantics supported, [30] provides the first mechanical verification tool for a subset of C11 programs with the GPS and RSL style reasoning. [31] encodes the line of RSL works [5], [8], [11] into the Viper verification infrastructure [32]. Similar automatic implementations would be one of our future interests.

Semantics-wise, we follow the axioms-based approach like that introduced by [4]. Similar axioms-based approach is also used for the Java memory model [33]. For various hardware memory models, operational semantics are also used [34]–[36]. Some recent works about the "promising" semantics [36], [37] propose a relatively more economical way to eliminate the "thin-air-read" problem without sacrificing too many possible optimisations. Its usefulness has been demonstrated in the formalisation of the ARMv8 memory model [38]. It is yet to be proved how handy it could be in supporting weak memory program logics.

IX. CONCLUSION

To the best of our knowledge, our proposed program logic GPS++ is the first one that provides the support to the reasoning about fully featured C11 release-sequences. Built on top of the GPS+ logic, our work is extended with the powerful restricted shareable assertions $\langle P \rangle_s$, an enhanced per-location protocol model, and a set of new verification rules. In addition, GPS++ also has the support to fractional permission, which makes it more practical than its predecessors, as now the reasoning about concurrent non-atomic reads is allowed.

In our future work, we aim to bring more memory orders, e.g., the *consume read*, into our reasoning framework. We also aim to apply GPS++ to more real-world C11 programs and develop reasoning aids with certain degrees of atomisation.

APPENDIX

MORE DEFINITIONS AND PROOF OF LEMMAS

In this section we present the less interesting yet indispensable definitions used in our reasoning system and semantic framework, and the proof the lemmas and corollaries we have discussed in the main text.

Our reasoning system is featured with new types of assertions and the corresponding inference rules to reason about them. We first present the semantics for our assertions; and then prove the soundness of our inference rules with the form $P \Rightarrow Q$.

	R	$\mathcal{R} \in [\![R]\!]^\rho$ iff
(1)	$t = t'$	$[\![t]\!]^\rho = [\![t']\!]^\rho$
(2)	$t \sqsubseteq_\tau t'$	$[\![t]\!]^\rho \sqsubseteq_\tau [\![t']\!]^\rho$
(3)	$\mathsf{uninit}(t)$	$\mathcal{R}(\mathsf{L}).\Pi([\![t]\!]^\rho) = \mathsf{uninit}$
(4)	$t \overset{f}{\mapsto} t'$	$\mathcal{R}(\mathsf{L}).\Pi([\![t]\!]^\rho) = \mathtt{na}([\![t']\!]^\rho, f) \wedge f \in (0, 1]$
(5)	$\boxed{t : t' \mid \tau}$	$\exists S.\ \mathcal{R}(\mathsf{L}).\Pi([\![t]\!]^\rho) = \mathtt{at}(\tau, S) \wedge [\![t']\!]^\rho \in S$
(6)	$t : t' \mid \mu$ (dashed box)	$\mathcal{R}(\mathsf{L}).g(\mu)([\![t]\!]^\rho) \geq [\![t']\!]^\rho$
(7)	$[\sigma]$	$\sigma \in \mathcal{R}(\mathsf{L}).\Sigma$
(8)	$P \wedge Q$	$\mathcal{R} \in [\![P]\!]^\rho \cap [\![Q]\!]^\rho$
(9)	$P \vee Q$	$\mathcal{R} \in [\![P]\!]^\rho \cup [\![Q]\!]^\rho$
(10)	$P \Rightarrow Q$	$\lfloor \mathcal{R} \rfloor \cap [\![P]\!]^\rho \subseteq [\![Q]\!]^\rho$
(11)	$\forall X.\ P$	$\mathcal{R} \in \bigcap_{d \in \mathtt{sort}(X)} [\![P]\!]^{\rho[X \mapsto d]}$
(12)	$\exists X.\ P$	$\mathcal{R} \in \bigcup_{d \in \mathtt{sort}(X)} [\![P]\!]^{\rho[X \mapsto d]}$
(13)	$P_1 * P_2$	$\mathcal{R} \in [\![P_1]\!]^\rho * [\![P_2]\!]^\rho$
(14)	$\square P$	$\lvert \mathtt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{L})] \rvert \in [\![P]\!]^\rho$
(15)	$\langle P \rangle$	$\mathtt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{A})] \in [\![P]\!]^\rho$
(16)	$\langle P \rangle_s$	$\mathtt{EMP}[\mathsf{L} \mapsto \mathcal{R}(s)] \in [\![P]\!]^\rho$
(17)	$\boxtimes P$	$\mathtt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{A})] \in [\![P]\!]^\rho$

Corollary 1 (Soundness of Corollary Inference Rules): Our corollary inference rules are semantically sound. That is, given an inference rule allowing $P \Rrightarrow Q$, we have $\exists \mathcal{R}.\ \lfloor \mathcal{R} \rfloor \cap [\![P]\!]^\rho \subseteq [\![Q]\!]^\rho$

Proof: For the [SEPARATION-R] rule: $\langle P * Q \rangle_s \Leftrightarrow \langle P \rangle_s * \langle Q \rangle_s$, from left to right by the definition (16), we have $\mathcal{R} \in [\![\langle P * Q \rangle_s]\!]^\rho \triangleq \texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(s)] \in [\![P * Q]\!]^\rho$. According to the definition (13), the term can be transformed into:

$$\exists r_1, r_2.\ \mathcal{R}(s) = r_1 \oplus r_2 \ \wedge$$
$$\texttt{EMP}[\mathsf{L} \mapsto r_1] \in [\![P]\!]^\rho \wedge \texttt{EMP}[\mathsf{L} \mapsto r_2] \in [\![Q]\!]^\rho,$$

which implies the right hand side term. Similarly, we can do it from right to left. The proof of this rule is finished.

For the [KNOWLEDGE-MANIPULATION-1] rule: $\Box P \Rrightarrow P$, according to definition (14) the semantic for its left hand side is $\mathcal{R} \in \Box P \triangleq |\texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{L})]| \in [\![P]\!]^\rho$. As $|\texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{L})]| \leq \mathcal{R}$, $\mathcal{R}$ is also in $[\![P]\!]^\rho$. The proof of this rule is finished.

Next we prove the rule [KNOWLEDGE-MANIPULATION-2]: $\Box P \Rrightarrow \Box\Box P$. Start from the rule's left hand side, we have $|\texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{L})]| \in [\![P]\!]^\rho$. Therefore, $||\texttt{EMP}[\mathsf{L} \mapsto \mathcal{R}(\mathsf{L})]|| \in [\![P]\!]^\rho$. The proof of this rule is finished.

For [KNOWLEDGE-MANIPULATION-3]: $\Box P * Q \Leftrightarrow \Box P \wedge Q$, as $\Box P$ is knowledge, the semantic definition (8) $[\![\Box P]\!]^\rho \cap [\![Q]\!]^\rho$ is equivalent to (13) $[\![\Box P]\!]^\rho * [\![Q]\!]^\rho$ in terms of evaluation results. Therefore, the rule is proven.

For [KNOWLEDGE-MANIPULATION-4…7], they are semantically sound as according to the definition of stripping operation and their corresponding semantic definitions ((7), (5), (1), and (6)) the resource representing these assertions does not change after stripping. Therefore they are able to be transformed into knowledge form.

For [SEPARATION-1]: $\boxed{\gamma : t \mid \mu} * \boxed{\gamma : t' \mid \mu} \Leftrightarrow \boxed{\gamma : t \cdot_\mu t' \mid \mu}$, starting from the rule's left hand side, we have:

$$\mathcal{R} \in [\![\boxed{\gamma : t \mid \mu} * \boxed{\gamma : t' \mid \mu}]\!]^\rho \triangleq$$
$$\exists \mathcal{R}_1 \in [\![\boxed{\gamma : t \mid \mu}]\!]^\rho, \mathcal{R}_2 \in [\![\boxed{\gamma : t' \mid \mu}]\!]^\rho.\ \mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2,$$

which is equivalent to the semantics of the rule's right hand side. The rule is proven.

For the [SEPARATION-2] rule: $\boxed{\ell : s \mid \tau} * \boxed{\ell : s' \mid \tau'} \Rrightarrow \tau = \tau' \wedge (s \sqsubseteq_\tau s' \vee s' \sqsubseteq_\tau s)$, starting from the rule's left hand side, we have:

$$\mathcal{R} \in [\![\boxed{\ell : s \mid \tau} * \boxed{\ell : s' \mid \tau'}]\!]^\rho \triangleq$$
$$\exists \mathcal{R}_1 \in [\![\boxed{\ell : s \mid \tau}]\!]^\rho, \mathcal{R}_2 \in [\![\boxed{\ell : s' \mid \tau'}]\!]^\rho.\ \mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2.$$

Given $\mathcal{R}_1 \# \mathcal{R}_2$, according to the definition of protocol compositions, the right hand side is implied. The rule is proven.

For [ASSERTION-PROPERTY-1…7], we prove that special assertions can not be nested. According to the definition (14-17), the nesting shown on the left hand side results in an empty resource map that is used to be checked with the original assertion P, which implies false unless P is $\texttt{emp}$. The proof is finished. $\square$

A ghost move is a transition that only changes auxiliary/logical computation states. This is ensured by our resource-level ghost moves. We first present our resource-level ghost moves below:

$$(1)\ \frac{\mathcal{R} \in [\![P]\!]^\rho}{\mathcal{R} \Rrightarrow [\![P]\!]^\rho} \qquad (2)\ \frac{\mathcal{R}_0 \in [\![P]\!]^\rho \quad \forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![P']\!]^\rho}{\mathcal{R}_0 \Rrightarrow [\![P']\!]^\rho}$$

$$(3)\ \frac{m \in [\![\mu]\!]}{\mathcal{R} \Rrightarrow \lfloor \mathcal{R} \rfloor * \{\texttt{EMP}[\mathsf{L} \mapsto (\bot, [\mu \mapsto [i \mapsto m]], \varnothing)]\}}$$

$$(4)\ \frac{\forall g_F \# g.\ g_F \# g'}{\texttt{EMP}[\mathsf{L} \mapsto (\Pi, g, \Sigma)] \Rrightarrow \lfloor \texttt{EMP}[\mathsf{L} \mapsto (\Pi, g', \Sigma)] \rfloor}$$

$$(5)\ \frac{\mathsf{interp}(\sigma) = ([\![P]\!]^\rho, [\![P']\!]^\rho) \quad \mathcal{R}' \in [\![P']\!]^\rho \quad \mathcal{R}[\mathsf{L}] = (\Pi, g, \Sigma)}{\mathcal{R} \oplus \mathcal{R}' \Rrightarrow \lfloor \mathcal{R}[\mathsf{L} \mapsto (\Pi, g, \Sigma \cup \{\sigma\})] \rfloor}$$

$$(6)\ \frac{\mathsf{interp}(\sigma) = ([\![P]\!]^\rho, [\![P']\!]^\rho) \quad \sigma \in \mathcal{R}[\mathsf{L}].\Sigma \quad \mathcal{R} \in [\![P]\!]^\rho}{\mathcal{R}_0 \oplus \mathcal{R} \Rrightarrow \lfloor \mathcal{R}_0 \rfloor * [\![P']\!]^\rho}$$

$$(7)\ \frac{\mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}] \oplus r \quad l \in \{\mathsf{S}\} \cup \mathbb{S} \quad \mathcal{R}'[l] \oplus r = \mathcal{R}[l] \qquad \forall l' \neq \mathsf{L} \vee l.\ \mathcal{R}'[l'] = \mathcal{R}[l']}{\mathcal{R} \Rrightarrow \lfloor \mathcal{R}' \rfloor}$$

Corollary 2 (Soundness of Ghost Moves): Our ghost move rules are semantically sound. That is, given a ghost move rule allowing $P \Rrightarrow Q$, we have $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q]\!]^\rho$.

Proof: We prove the ghost move rules one by one.

For the [GHOST-MOVE-1] rule, we are going to prove that$\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q]\!]^\rho$, with the premise that is given as $\forall \mathcal{R}.\ \lfloor \mathcal{R} \rfloor \cap [\![P]\!]^\rho \subseteq [\![Q]\!]^\rho$. The premise can be simplified as $[\![P]\!]^\rho \subseteq [\![Q]\!]^\rho$. Therefore, we have $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \in [\![Q]\!]^\rho$. By using the resource level ghost move (1), we have $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q]\!]^\rho$. [GHOST-MOVE-1] is proven.

For the [GHOST-MOVE-2] rule, we are going to prove that $\forall \mathcal{R} \in [\![P * R]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q * R]\!]^\rho$. According to the definition of separation assertions, it can be transformed to:

$$\forall \mathcal{R} \in [\![P]\!]^\rho, \mathcal{R}' \in [\![R]\!]^\rho.$$
$$\mathcal{R} \oplus \mathcal{R}' \Rrightarrow \{\mathcal{R}_1 \oplus \mathcal{R}_2 \mid \mathcal{R}_1 \in [\![Q]\!]^\rho \wedge \mathcal{R}_2 \in [\![R]\!]^\rho\}.$$

According to the premise, we have $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q]\!]^\rho$. We can check that the proof obligation is valid for all possible ghost moves allowed by the resource-level ghost move rules. [GHOST-MOVE-2] is proven.

For the [GHOST-MOVE-3] rule, initially we have the following property: $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![Q]\!]^\rho \wedge \forall \mathcal{R} \in [\![Q]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![R]\!]^\rho$, and we are going to prove that $\forall \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![R]\!]^\rho$. From the first premise $\forall \mathcal{R} \in [\![P]\!]^\rho$ can be transformed into some resource in $[\![Q]\!]^\rho$ for all possible ghost moves. Together with the second premise, the rule is proven.

The [GHOST-MOVE-3] and [UNSHARE-R] can be proved by moving the resource under the shareable labels to the resource maps' local component, which is allowed by the resource level ghost move (7).

For the [GHOST-MOVE-4] rule, we are going to prove that $\forall \mathcal{R} \in [\![\mathsf{true}]\!]^\rho.\ \mathcal{R} \Rrightarrow [\![\exists \gamma.\ \boxed{\gamma : t_1 \mid \mu}]\!]^\rho$, which can be done by applying the resource level ghost move (3).

The [GHOST-MOVE-5] is a corollary of the resource level ghost move (4), while the [GHOST-MOVE-6] is a corollary of the resource level ghost move (5), and [GHOST-MOVE-7] is a corollary of the resource level ghost move (6). □

Now we present the proofs of the labelling lemmas we have discussed for our reasoning system's global safety with the following definitions (which has been informally discussed in the main text):

$$\begin{aligned}
&\mathcal{R} \Rrightarrow_{\mathcal{I}} \mathcal{R}' \triangleq \exists g.\ \mathcal{R}' = (\mathcal{R}.\Pi \cup \{\sigma \,|\, (\sigma, -) \in \mathcal{I}\}) \\
&N \triangleq \mathsf{dom}(G.A) \\
&a \in \mathsf{valid}(G, \mathcal{L}, N) \triangleq \exists, \mathcal{R}, \mathcal{I}. \\
&\quad \mathcal{L} \in \mathit{labelling}(G) \\
&\quad \mathsf{in(sb)} \oplus \mathsf{in(rf)} \oplus \mathsf{in(esc)} \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out(esc)} \oplus \mathsf{out(cond)} \\
&\quad (\mathsf{out(sb)}, \mathsf{out(rf)}) \in \mathsf{guar}(\mathsf{in(sb)} \oplus \mathsf{in(rf)}, \mathcal{R}, G.A(a)) \\
&\quad (\forall b \in N.\ \mathsf{isUpd}(b) \wedge \mathsf{rf}(b) = a) \Longrightarrow (\mathcal{L}(\mathsf{rf}, a, b) = \mathsf{out(rf)}) \\
&\quad (\nexists b \in N.\ \mathsf{isUpd}(b) \wedge \mathsf{rf}b = a) \Longrightarrow (\mathcal{L}(\mathsf{rf}, a, \bot) = \mathsf{out(rf)}) \\
&\quad |\mathcal{L}(\mathsf{rf}, a, \bot)| = |\mathsf{out(rf)}| \\
&\quad \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{P}, \mathcal{P}') \Longrightarrow \mathcal{R}_E \in \mathcal{P}' \\
&\quad \mathcal{L}(\mathsf{esc}, a, \bot) \ = \\
&\qquad \bigoplus \left\{ \mathcal{R}_E \;\middle|\; \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathcal{R}_E \in \mathcal{P}', \\ \mathsf{interp}(\sigma) = (\mathcal{P}, \mathcal{P}'), \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{P}) \end{array} \right\} \\
&\quad \mathit{where},\ \mathsf{in}(x) \triangleq \bigoplus \{\mathcal{L}(x, b, a) \mid (x, b, a) \in \mathsf{dom}(\mathcal{L})\} \\
&\qquad\quad \mathsf{out}(x) \triangleq \bigoplus \{\mathcal{L}(x, a, c) \mid (x, a, c) \in \mathsf{dom}(\mathcal{L})\} \\
&\mathsf{compat}(G, \mathcal{L}) \triangleq \forall \epsilon \subseteq \mathsf{dom}(\mathcal{L}). \\
&\quad \nexists e_1, e_2 \in \epsilon.\ G.\mathsf{hb}^*(\mathsf{target}(e_1), \mathsf{source}(e_2)) \\
&\qquad \Longrightarrow \bigoplus_{\eta \in \epsilon} \mathcal{L}(\eta) \text{defined} \\
&\mathsf{conform}(G, \mathcal{L}, N) \triangleq \forall \ell.\ \forall a, b \in N. \\
&\quad G.\mathsf{mo}_{l,\mathtt{at}}(a, b) \Longrightarrow \mathsf{out}(\mathcal{L}, a, \mathsf{rf})[\ell] \sqsubseteq_{\mathtt{at}} \mathsf{out}(\mathcal{L}, b, \mathsf{rf})[\ell]
\end{aligned}$$

Lemma 1 (Step Preparation):

$$\begin{aligned}
\mathit{if}\ \ & \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathsf{consistentC11}(\mathcal{G}') \\
& \wedge \mathsf{dom}(\mathcal{G}'.A) = \mathsf{dom}(\mathcal{G}.A) \uplus b \\
& \wedge \mathcal{L}(\mathsf{sb}, a, \bot) = \mathcal{R} \oplus \mathcal{R}_{rem} \\
& \wedge \mathsf{dom}(\mathcal{G}.A) \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \forall c \in \mathsf{dom}(\mathcal{G}.A).\ \mathcal{G}.A(c) = \mathcal{G}'.A(c) \\
& \wedge \mathcal{G}'.\mathsf{sb} = G.\mathsf{sb} \uplus [a, b) \\
& \wedge \forall c \in \mathsf{dom}(\mathcal{G}.A).\ \mathcal{G}.\mathsf{rf}(c) = \mathcal{G}'.\mathsf{rf}(c) \\
& \wedge \mathcal{G}'.\mathsf{mo} \supseteq \mathcal{G}.\mathsf{mo} \\
\mathit{then}\ \ & \exists \mathcal{L}'.\ \mathsf{dom}(\mathcal{G}.A) = \mathsf{valid}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}', \mathcal{L}') \\
& \wedge \mathsf{conform}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}'.A)) \\
& \wedge \mathcal{L}'(\mathsf{sb}, a, \bot) = \mathcal{R}_{rem} \\
& \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{sb}) = \mathcal{R} \\
& \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{rf}) = \mathtt{EMP} \\
& \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{esc}) = \mathtt{EMP} \\
& \wedge \forall a' \neq a.\ \mathcal{L}'(\mathsf{sb}, a', b) = \mathtt{EMP} \\
& \wedge \mathsf{out}(\mathcal{L}', b, \mathsf{all}) = \mathtt{EMP} \\
& \wedge \forall a' \neq a.\ \mathcal{L}'(\mathsf{sb}, a', \bot) = \mathcal{L}(\mathsf{sb}, a', \bot)
\end{aligned}$$

Proof: Firstly, we prove the compat property. Notice that for all edges in that are hb-independent with the edge $\mathsf{sb}(a, b)$, they are hb-independent with the $\mathsf{sb}(a, \bot)$ edge as well. Suppose the sum of the resources carried under their local label r is incompatible with $\mathcal{L}(\mathsf{sb}, a, b)(\mathsf{L})$, i.e., $\neg r \# \mathcal{L}(\mathsf{sb}, a, b)(\mathsf{L})$. According our labeling rule $\mathcal{L}(\mathsf{sb}, a, b)(\mathsf{L}) \leq \mathcal{L}(\mathsf{sb}, a, \bot)(\mathsf{L})$. That is there exists some r' that makes $\mathcal{L}(\mathsf{sb}, a, \bot)(\mathsf{L}) = \mathcal{L}(\mathsf{sb}, a, b)(\mathsf{L}) \oplus r'$. Therefore we can deduce the following property $\neg r \# \mathcal{L}(\mathsf{sb}, a, \bot)(\mathsf{L})$ as $r \oplus \mathcal{L}(\mathsf{sb}, a, \bot)(\mathsf{L}) = r \oplus \mathcal{L}(\mathsf{sb}, a, b)(\mathsf{L}) \oplus r'$ which has undefined result. However $\neg r \# \mathcal{L}(\mathsf{sb}, a, \bot)(\mathsf{L})$ contradicts with the premise where the $\mathsf{compat}(\mathcal{G}, \mathcal{L})$ property holds. Therefore, the sum of the resources carried by all the edges hb-independent with $\mathsf{sb}(a, b)$ is compitable with its resource, and we can derive that $\mathsf{compat}(\mathcal{G}', \mathcal{L}')$

To prove the $\mathsf{conform}$ property, notice that the atomic locations are unchanged in this step's labelling process. Therefore $\mathsf{conform}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}'.A))$ is essentially equivalent to $\mathsf{conform}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}'.A))$. To prove the valid property and the validity of the updated labelling, we unfold the corresponding definitions and check the requirements with our labelling results. □

Lemma 2 (Rely Step):

$$\begin{aligned}
\mathit{if}\ \ & \mathcal{G}.A(a) = \alpha \\
& \wedge \mathsf{dom}(\mathcal{G}.A) = N \uplus a \\
& \wedge N \in \mathsf{prefix}(\mathcal{G}) \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) = \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathtt{EMP} \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) = \mathtt{EMP} \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{esc}) = \mathtt{EMP} \\
& \wedge \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathtt{EMP} \\
\mathit{then}\ \ & \exists \mathcal{L}'.\ N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}', N) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \\
& \wedge \mathsf{in}(\mathcal{L}', a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{rf}) \in \mathsf{rely}(\mathsf{in}(\mathcal{L}', a, \mathsf{sb}), \alpha) \\
& \wedge \mathsf{in}(\mathcal{L}', a, \mathsf{esc}) = \mathsf{out}(\mathcal{L}', a, \mathsf{all}) = \mathtt{EMP} \\
& \wedge \forall b, c.\ \mathcal{L}'(\mathsf{sb}, b, c) = \mathcal{L}(\mathsf{sb}, b, c) \\
& \wedge \forall b.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot)
\end{aligned}$$

Proof: We first focus on the compat property. For non-update reading actions, we argue that only knowledge is taken into the new node and knowledge is always compatible with the environment. For an relaxed atomic update, the resource taken in from its rf edge is left to be acquired, therefore the local resources are still compatible. For an acquire atomic update, which can take non-duplicable resource to its

local resource, we notice that there must be a nearest release action b that made the resource shareable. Therefore we have b happens before a, which does not break the compat property.

We assert that $\mathsf{conform}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}'.A))$ is essentially equivalent to $\mathsf{conform}(\mathcal{G}', \mathcal{L}', \mathsf{dom}(\mathcal{G}'.A))$ as there is no changes to atomic locations in this labelling step. To prove the valid property and the validity of the updated labelling, we unfold the corresponding definitions and check the requirements with our labelling results. □

Lemma 3 (Ghost Step):

$$\begin{aligned}
\textit{if}\quad & \mathsf{dom}(\mathcal{G}.A) = N \uplus a \wedge N \in \mathsf{prefix}(\mathcal{G}) \\
& \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathcal{R}_{before} \triangleq \mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{esc}) \\
& \wedge \mathcal{R}_{after} \triangleq \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}_{before} \Rrightarrow_{\mathcal{I}} \mathcal{R}_{after} \wedge |\mathcal{R}_{before}| \leq \mathcal{R} \wedge \mathcal{R} \Rrightarrow \mathcal{P} \\
& \wedge \forall c.\ \mathcal{L}(\mathsf{esc}, -, a, e) = \mathrm{EMP} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) \ = \\
& \bigoplus \left\{ \mathcal{R}_E \mid \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathcal{R}_E \in \mathcal{Q}', \\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\} \\
\textit{then}\quad & \exists \mathcal{L}', \mathcal{I}', \mathcal{R}', \mathcal{R}'_{before}, \mathcal{R}'_{after} \in \mathcal{P}. N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}', \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}'[(\mathsf{esc}, -, a, \bot) := \mathcal{L}'(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}']) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}', N) \\
& \wedge \mathcal{R}'_{before} \triangleq \mathsf{in}(\mathcal{L}', a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{rf}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{esc}) \\
& \wedge \mathcal{R}'_{after} \triangleq \mathcal{R}' \oplus \mathsf{out}(\mathcal{L}', a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}'_{before} \Rrightarrow_{\mathcal{I}'} \mathcal{I}' \mathcal{R}'_{after} \\
& \wedge \forall b.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot) \\
& \wedge \forall b.\ \mathcal{L}'(\mathsf{rf}, b, \bot) = \mathcal{L}(\mathsf{rf}, b, \bot) \\
& \wedge \forall b, c.\ \mathcal{L}'(\mathsf{sb}, b, c) = \mathcal{L}(\mathsf{sb}, b, c) \\
& \wedge \forall b, c.\ \mathcal{L}'(\mathsf{rf}, b, c) = \mathcal{L}(\mathsf{rf}, b, c) \\
& \wedge \forall c.\ \mathcal{L}'(\mathsf{esc}, -, a, e) = \mathrm{EMP} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}'.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) = \\
& \bigoplus \left\{ \mathcal{R}_E \mid \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}', \mathcal{R}_E \in \mathcal{Q}', \\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\}
\end{aligned}$$

Proof: To prove the compat property, firstly we assert that the escrowed resource a retrieved must have been put under escrow by an event that happens before a. This is because our labelling process ensures that the escrowed resource is initially attached to the creator's escrow sink edge, and can only appear in a node's local component if that node is in a chain of $(\mathsf{sb} \cup \mathsf{sw})^+$ following the creator. Then we assert that the compat property holds for the updated graph following the same argument as that used in the compat proof for the Lemma *Rely Step*. We also assert that the conform, valid properties and the new labelling are valid for the same reasons discussed in the proof for the previous lemma. □

Lemma 4 (Protocol Equivalence for Writes):

$$\begin{aligned}
\textit{if}\quad & \mathsf{dom}(\mathcal{G}.A) = N \uplus a \wedge N \in \mathsf{prefix}(\mathcal{G}) \\
& \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge |\mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf})| \leq \mathcal{R} \\
\textit{then}\quad & (\mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}), \mathsf{in}(\mathcal{L}, a, \mathsf{all})) \in \mathsf{wpe}(\mathcal{G}.A(a))
\end{aligned}$$

Proof: We prove this lemma by firstly unfold the definitions of valid, compat, *and* conform. Then we do case analysis based on the type of a and check the corresponding wpe definitions. □

Lemma 5 (Guarantee Step):

$$\begin{aligned}
\textit{if}\quad & \mathcal{G}.A(a) = \alpha \wedge \mathsf{dom}(\mathcal{G}.A) = N \uplus a \wedge N \in \mathsf{prefix}(\mathcal{G}) \\
& \wedge N \subseteq \mathsf{valid}(\mathcal{G}, \mathcal{L}, \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}, N) \wedge \mathsf{consistentC11}(\mathcal{G}) \\
& \wedge \mathcal{R}_{\mathsf{pre}} = \mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}_{\mathsf{pre}} \in \mathsf{rely}(-, \alpha) \wedge |\mathsf{in}(\mathcal{L}, a, \mathsf{all})| \leq \mathcal{R} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) \ = \\
& \bigoplus \left\{ \mathcal{R}_E \mid \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathcal{R}_E \in \mathcal{Q}', \\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\} \\
& \wedge \mathsf{out}(\mathcal{L}, a, \mathsf{sb}) = \mathsf{out}(\mathcal{L}, a, \mathsf{rf}) = \mathrm{EMP} \\
& \wedge (\mathcal{R}_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha) \wedge \mathsf{wpe}(\alpha, \mathcal{R}_{\mathsf{pre}}, \mathsf{in}(\mathcal{L}, a, \mathsf{all})) \\
\textit{then}\quad & \exists \mathcal{L}'.\ \mathsf{dom}(\mathcal{G}.A) = \mathsf{valid}(\mathcal{G}, \mathcal{L}', \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \mathsf{compat}(\mathcal{G}, \mathcal{L}') \wedge \mathsf{conform}(\mathcal{G}, \mathcal{L}', \mathsf{dom}(\mathcal{G}.A)) \\
& \wedge \forall b \neq a.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot) \wedge \mathcal{L}'(\mathsf{sb}, a, \bot) = \mathcal{R}_{\mathsf{sb}}
\end{aligned}$$

Proof: For the new node a's sb outgoing edge, we first proved the proof for its compat property. Suppose there is an edge ξ that is hb-independent with the new node a's sboutgoing edge $\mathsf{sb}(a, \bot)$ and the resource map it carries is incompatible with $\mathcal{L}(\mathsf{sb}, a, \bot)$. Supposing a is not an acquire action, i.e. the incompatibility is not caused by moving resources from the current resource map's A component to its local component, we use case analysis to demonstrate that the action's guarantee condition would not introduce new incompatibility; however if the incompatibility is not newly introduced, it would appear in one of the incoming edges, which violates the compat property in the premise. If a is an acquire action, which could move resources from the A component to L, we argue incompatibility can not be introduced in this process by contradiction. Assuming the incompatible resource originally in A is r, there must be another node,

say b that moves r from L to a shareable component and thereafter r could be loaded to A in a's thread, according to our labeling method. According to the C11 memory model, we derive that b happens before a. If there exists an edge ξ that is hb-independent with $\mathcal{L}'(\mathsf{sb}, a, \bot)$ and carries the incompatible resource r, it also hb-independent with one of a's incoming edges that carries r (the case that r is created by a is trivial), which violates the $\mathsf{compat}(\mathcal{G}, \mathcal{L})$. With the same argument used for the release fence case, we argue that the compat property holds for a's rf outgoing edge.

For the new graph's $\mathsf{conform}$ property, we prove by exhaustion on all possible type of actions, and check unfolded $\mathsf{conform}$ definitions against the guarantee conditions for the corresponding type of actions. Same to that for previous lemmas the proof for the valid property and the validity of the updated labelling, can be demonstrated by unfolding the corresponding definitions and check the requirements with our labelling results. □

Lemma 6 (Error Free): If $\mathsf{GSafe}_n(\mathcal{T}_{ins}, \mathcal{G}, \mathcal{L})$ *then we have* $\neg\mathsf{dataRace}(\mathcal{G})$, $\neg\mathsf{memErr}(\mathcal{G})$*, and all reads are initialised.*

Proof: We first prove that an event graph supporting safe executions is free from race conditions using proof by contradiction. Suppose event a and b in $\mathcal{G}$ cause a data race, by the definition of race condition we can deduce that $\neg\mathsf{hb}(a, b) \wedge \neg\mathsf{hb}(b, a)$. Also, we can derive that there is a location ℓ that appears in both a and b's incoming edges that holds some non-trivial (not $\bot$) non-atomic values. However, given that a's incoming edges are hb-independent with that of b's the appearance of the non-atomic resource about location ℓ in both groups violates the compatproperty in the global safety definitions. Therefore, there are no two events in $\mathcal{G}$ that could raise a data race.

To prove the graph is free from memory errors, i.e., there is no memory access to unallocated memory locations, first notice that ensured by our instrumented semantics to manipulate a location ℓ, an event b must have the information about ℓ in one of its incoming edge's local component. Then we define a recursive search procedure to find the action that allocates ℓ and demonstrate that the memory accesses to ℓ are error free.

Firstly, we search backwards following the sb^+ edges in the graph starting from b, until we reach an event a where the information about ℓ is not in one of its incoming edge's local component. In the case that a is an allocation action and it allocates ℓ, the search ends. In the case that a is an acquire fence and it moves ℓ from the waiting-to-be-acquired component in one of its incoming edge to the local component in its outgoing resource maps, we assert that there exists an read or update event a_0 with relaxed memory order prior to a in the sb^+ relation according to our labelling process, and the information about ℓ appears in a_0's read-from incoming edge under the waiting-to-be-acquired label. Then we recursively perform this search procedure starting with the write event that a_0 reads from until we find the right allocation action. If it is not the cases aforementioned, we check if a's immediate sbsuccessor a' is a read or update event with acquire memory order. If so, we recursively perform this search procedure starting with the write event that a' reads from. We assert these are all the cases needed to be considered as any other case violates the global safety definitions according to our labelling process. Therefore, a globally safe event graph is error free. □

REFERENCES

[1] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[2] *Information Technology–Programming Language–C*, Standard ISO/IEC 9899:2011, 2011.

[3] *Information Technology–Programming Language–C++*, Standard ISO/IEC 14882:2011, 2011.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 55–66, Jan. 2011.

[5] V. Vafeiadis and C. Narayan, "Relaxed separation logic: A program logic for C11 concurrency," in *Proc. OOPSLA*, 2013, pp. 867–884.

[6] A. Turon, V. Vafeiadis, and D. Dreyer, "GPS: Navigating weak memory with ghosts, protocols, and separation," in *Proc. OOPSLA*, 2014, pp. 691–707.

[7] J. Tassarotti, D. Dreyer, and V. Vafeiadis, "Verifying read-copy-update in a logic for weak memory," in *Proc. PLDI*, 2015, pp. 110–120.

[8] M. Doko and V. Vafeiadis, "A program logic for C11 memory fences," in *Proc. VMCAI*. New York, NY, USA: Springer-Verlag, 2016.

[9] M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira, "Reasoning about fences and relaxed atomics," in *Proc. PDP*, Feb. 2016, pp. 520–527.

[10] M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira, "GPS++: Reasoning about fences and relaxed atomics," *Int. J. Parallel Program.*, vol. 46, no. 6, pp. 1157–1183, Dec. 2018.

[11] M. Doko and V. Vafeiadis, "Tackling real-life relaxed concurrency with FSL++," in *Proc. ESOP*, H. Yang, Ed. Berlin, Germany: Springer, 2017, pp. 448–475.

[12] J. Boyland, "Checking interference with fractional permissions," in *Proc. 10th Int. Conf. Static Anal. (SAS)*. Berlin, Germany: Springer-Verlag, 2003, pp. 55–72.

[13] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA, 2005, pp. 259–270.

[14] M. He, S. Qin, and J. Ferreira, "Towards a program logic for C11 release-sequences," in *Proc. Int. Symp. Theor. Aspects Softw. Eng. (TASE)*, Aug. 2018, pp. 28–35.

[15] V. Vafeiadis and M. Parkinson, "A marriage of rely/guarantee and separation logic," in *Proc. CONCUR*. New York, NY, USA: Springer-Verlag, 2007.

[16] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, nos. 1–3, pp. 271–307, Apr. 2007.

[17] X. Feng, "Local rely-guarantee reasoning," in *Proc. POPL*, 2009, pp. 315–327.

[18] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis, "Deny-guarantee reasoning," in *Proc. ESOP*. New York, NY, USA: Springer-Verlag, 2009, pp. 363–377.

[19] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Proc. TPHOLs*. New York, NY, USA: Springer-Verlag, 2009, pp. 23–42.

[20] K. R. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with chalice," in *Foundations of Security Analysis and Design V*, A. Aldini, G. Barthe, and R. Gorrieri, Eds. Berlin, Germany: Springer-Verlag, 2009, pp. 195–222.

[21] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *Proc. ECOOP*, 2010, pp. 504–528.

[22] A. Turon, D. Dreyer, and L. Birkedal, "Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency," in *Proc. 18th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, 2013, pp. 377–390.

[23] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco, "Communicating state transition systems for fine-grained concurrent resources," in *Proc. ESOP*. New York, NY, USA: Springer-Verlag, 2014, pp. 290–310.

[24] K. Svendsen and L. Birkedal, "Impredicative concurrent abstract predicates," in *Proc. ESOP*. New York, NY, USA: Springer-Verlag, 2014, pp. 149–168.

[25] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, "TaDA: A logic for time and data abstraction," in *Proc. ECOOP*, 2014, pp. 207–231.

[26] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proc. POPL*, 2015, pp. 637–650.

[27] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, "Views: Compositional reasoning for concurrent programs," in *Proc. 40th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2013, pp. 287–300.

[28] R. Ley-Wild and A. Nanevski, "Subjective auxiliary state for coarse-grained concurrency," *ACM SIGPLAN Notices*, vol. 48, no. 1, pp. 561–574, Jan. 2013.

[29] R. G. Babb, "Issues in the specification and design of parallel programs," in *Proc. 6th Int. Workshop Softw. Specification Design (IFIP)*, vol. 83, 1983, pp. 321–332.

[30] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis, "Strong logic for weak memory: Reasoning about release-acquire consistency in iris," in *Proc. 31st Eur. Conf. Object-Oriented Program. (ECOOP)*, vol. 74, P. Müller, Ed. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 17:1–17:29.

[31] A. J. Summers and P. Müller, "Automating deductive verification for weak-memory programs (extended version)," *Int. J. Softw. Tools Technol. Transf.*, pp. 1–20, Mar. 2020.

[32] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Proc. Int. Conf. Verification, Model Checking, Abstract Interpretation*. Cham, Switzerland: Springer, 2016, pp. 41–62.

[33] W. Pugh, "Fixing the Java memory model," in *Proc. ACM Conf. Java Grande (JAVA)*, New York, NY, USA, 1999, pp. 378–391.

[34] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-CC multiprocessor machine code," in *Proc. 36th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA, 2008, pp. 379–391.

[35] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER multiprocessors," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, New York, NY, USA, 2011, pp. 175–186.

[36] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA, 2016, pp. 608–621.

[37] S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis, "Promising 2.0: Global optimizations in relaxed memory concurrency," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2020, pp. 362–376.

[38] C. Pulte, J. Pichon-Pharabod, J. Kang, S.-H. Lee, and C.-K. Hur, "Promising-ARM/RISC-V: A simpler and faster operational concurrency model," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2019, pp. 1–15.

MENGDA HE received the Ph.D. degree in computer science from Teesside University, in 2018. He is currently a Research Associate with the School of Computing, Engineering, and Digital Technologies, Teesside University. His research interests include formal methods, program logics and software verification, concurrent programming, and memory models. Particularly, he has been working on the C11 memory model related problems since his Ph.D., using separation logic, rely-guarantee logic, and various techniques for concurrent semantics. He contributed several papers to this field.

SHENGCHAO QIN (Senior Member, IEEE) received the Ph.D. degree in applied mathematics from Peking University. He also worked as a Postdoctoral Research Fellow at the National University of Singapore, under the Singapore-MIT Alliance program, before moving his job to U.K. While in U.K., he worked as a University Lecturer at Durham University, and as a Reader in Teesside University, before he was promoted to Professor (Chair) of Computer Science in 2011. His research interests include formal methods, software engineering, and programming languages, in particular, formal specification and modeling, program analysis and verification, theories of programming, and program logic such as separation logic. Until now, he has published over 120 papers in international journals and peer-refereed international conferences. He is a Senior Member of the ACM. He serves as a full member of EPSRC Peer Review College and a member of the UKRI Future Leaders Fellowship Peer Review College.

ZHIWU XU received the Ph.D. degree in computer science from University Paris Diderot - Paris 7 and University of Chinese Academy of Sciences, under the joint cultivation, in 2013. He is currently an Associate Professor with Shenzhen University. His research is in the area of program analysis and verification, type systems, software security, automata theory and logic, and machine learning.