Reasoning about Fences and Relaxed Atomics

Mengda He*† Viktor Vafeiadis‡ Shengchao Qin*† João F. Ferreira*§
*School of Computing, Teesside University, UK
†Shenzhen University, China
‡Max Planck Institute for Software Systems (MPI-SWS), Germany
§HASLab / INESC TEC, Universidade do Minho, Portugal
Email: {m.he,s.qin,jff}@tees.ac.uk, viktor@mpi-sws.org

***Abstract*—For efficiency reasons, *weak* (or *relaxed*) memory is now the norm on modern architectures. To cater for this trend, modern programming languages are adapting their memory models. The new C11 memory model [1] allows several levels of memory weakening, including non-atomics, relaxed atomics, release-acquire atomics, and sequentially consistent atomics. Under such weak memory models, multithreaded programs exhibit more behaviours, some of which would have been inconsistent under the traditional strong (i.e. sequentially consistent) memory model. This makes the task of reasoning about concurrent programs even more challenging. The GPS framework, recently developed by Turon et al. [22], has made a step forward towards tackling this challenge. By integrating ghost states, per-location protocols and separation logic, GPS can successfully verify programs with release-acquire atomics. In this paper, we present a program logic, an enhancement of the GPS framework, that can support the verification of a bigger class of C11 programs, that is, programs with release-acquire atomics, relaxed atomics and release-acquire fences. Key elements of our proposed logic include two new types of assertions, a more expressive resource model and a set of newly-designed verification rules.**

I. Introduction

Memory models are important for concurrent programs, as they define how different threads can interact with each other based on the shared resources in memory. Most work on concurrent program verification assumes the *sequentially consistency* (SC) memory model [12], which assumes a single global memory. Threads take turns to access it, while within each thread the program order is preserved, and each update to memory becomes visible to all threads at the same time and as soon as they occur. However, this assumption is no longer true for many modern architectures (like the ARM and PowerPC processors), in which memory consistency models are weakened for efficiency reasons.

The SC model is intuitive and simplifies reasoning about concurrent programs. However, such strong models are expensive for modern architectures to adopt as costly synchronisation instructions (e.g., hardware fences) would be required to keep memory operations properly synchronised. Modern architectures therefore employ relaxed memory models in which different threads may observe different orders of memory operations. For instance, the x86 architecture uses *total-store-order (TSO)*, where some ordering may be broken as long as a total order for all store operations is preserved; ARM and PowerPC architectures use even weaker memory models. To allow programmers to write more efficient concurrent code, programming languages like C/C++ and Java follow a weak memory model [1], [15]. However, there is a demand in search for programming logics that can reason about concurrent programs assuming weak memory models. Two notable examples are the recent frameworks *Relaxed Separation Logic* (RSL) [23] and *GPS* [22]. These frameworks offer well-designed reasoning support for release/acquire and SC atomics and have been successfully applied to verify real code in the Linux Kernel [20]. However, neither of them support *fences*, an important synchronisation mechanism. Moreover, the focus of GPS has been solely on release/acquire atomics, meaning that *relaxed atomics* are not yet supported.

In this current work we propose a program logic that enhances the GPS reasoning mechanism to support the verification of a much bigger class of C11 programs (than what GPS can support). More specifically, we propose two new types of assertions, namely *shareable* assertions and *waiting-to-be-acquired* assertions, to facilitate the reasoning about fences and relaxed atomics. We design a set of new verification rules that can verify programs with release/acquire atomics, relaxed atomics and release/acquire fences.

Our work is based on the C11 memory model [1], which will be depicted in §II. We briefly introduce GPS in §III and present our new program logic in §IV. The new rules are put into action in §V with an illustrative example. We present our new resource model in §VI before we conclude in §VII.

II. The Language and the Memory Model

We first present the syntax and semantics for a language capturing the essential C11 features, an extension of the core language used in GPS [22]; we then introduce the (simplified) C11 memory model on which our work is based.

A. The Language

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

Fig. 1: A language for C11 concurrency with relaxed atomics and fences

Our core language (Fig 1) is an expression-oriented language with pointer arithmetic, `let`-binding (which is the only evaluation context *K*), `repeat` e command (which repeatedly executes its body e until a non-zero value is returned), thread forking, conditional statement, memory allocation, load, store and fence operations annotated with a specific *memory order* (*MO*), and the atomic operations compare-and-swap and fetch-and-increment.

Note the memory order annotation can be `rel` (for release store atomic), `acq` (for acquire read atomic), `rlx` (for relaxed atomic), and `na` (for non-atomic).[1] Note also that we focus on fence commands annotated with `rel` or `acq` in this work. For the compare-and-swap command `CAS`, we assume it to have both `rel` and `acq` effects in case the operation succeeds, and `rlx` in case the update does not take place.

B. The Graph Semantics

Assuming a weak memory model, C11 allows different threads to have different observations of the memory. Therefore it is hard to express its semantics in terms of changing a single shared memory. Instead, we need to track the history of an execution, annotate the relations among its events, and then judge if that execution fulfils the memory model (e.g. whether an access to a certain location leads to a data-race, or if it is possible for a read action to return a certain value). This approach is followed by Batty et al. [2] to formally define the C11 memory model. The same approach is followed by RSL and GPS though with simplifications to make their focus clear. We follow the same approach and present a graph based semantics. Fig 2 gives the definition of an event graph, which is formed by an action map and three relations *sequenced before* sb, *modification order* mo and *read from* rf.

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

Fig. 2: Syntax of event graph

We follow the two-layer semantics given in GPS but extend it to support relaxed atomics and fences. Some of the semantic rules are shown in Fig 3 and Fig 4, where $\mathbf{C}$ is the word size. In the event layer, actions are generated from program expressions $e \xrightarrow{\alpha} e'$. Note that a load operation generates a read action $\mathbb{R}$ with an arbitrary value. The actual value read is constrained by the C11 memory model in the second layer of semantics. Note also that $\mathbb{S}$ stands for a skip action, $\mathbb{A}$ for a memory allocation, $\mathbb{W}$ for a write, $\mathbb{U}$ for an atomic update, and $\mathbb{F}$ for a fence action.

In the second layer of semantics, instead of transforming expressions, a machine step changes *machine configurations* $\langle T; G\rangle$. Here T is the pool of threads maintaining the identity of the last event produced by each thread and their corresponding continuation expressions, and G is the event graph built up so far. In the graph G, all the events that have taken place are recorded in the action map A and are connected with three kinds of directed edges, namely sb, mo and rf.

[1]GPS focuses only on `rel` and `acq` and denotes them as `at`.

$$
\begin{array}{llll}
\texttt{let } x = V \texttt{ in } e & \xrightarrow{\mathbb{S}} & e[V/x] & \\
\texttt{repeat } e \texttt{ end} & \xrightarrow{\mathbb{S}} & & \\
\quad \texttt{let } x = e \texttt{ in if } x \texttt{ then } x \texttt{ else repeat } e \texttt{ end} & & & \\
\texttt{alloc}(n) & \xrightarrow{\mathbb{A}(l..l+n-1)} & l & \\
[l]_O & \xrightarrow{\mathbb{R}(l,V,O)} & V & \\
[l]_O := V & \xrightarrow{\mathbb{W}(l,V,O)} & 0 & \\
\texttt{CAS}(l, V_o, V_n) & \xrightarrow{\mathbb{U}(l,V_o,V_n)} & 1 & \\
\texttt{CAS}(l, V_o, V_n) & \xrightarrow{\mathbb{R}(l,V',\texttt{rlx})} & 0 & V' \neq V_o \\
\texttt{FAI}(l) & \xrightarrow{\mathbb{U}(l,V,V')} & V & \\
 & & V' = (V+1) \bmod \mathbf{C} & \\
\texttt{fence}_O & \xrightarrow{\mathbb{F}(O)} & 0 & \\
K[e] & \xrightarrow{\alpha} & K[e'] & e \xrightarrow{\alpha} e'
\end{array}
$$

Fig. 3: Some event-step semantic rules: $e \xrightarrow{\alpha} e'$

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

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

Fig. 4: Machine step semantics: $\langle T; G\rangle \longrightarrow \langle T'; G'\rangle$

The *sequenced-before* relation ($\mathsf{sb} \subseteq \mathsf{dom}(A) \times \mathsf{dom}(A)$) records the order of events as specified in the program. As in GPS and RSL, we make this relation *not* transitive. Thus it relates each node only to its immediate successor in program order. Note that the *modification-order* ($\mathsf{mo} \subseteq \mathsf{dom}(A) \times \mathsf{dom}(A)$) is a strict, total order on all writing actions to the same location. The *reads-from* map ($\mathsf{rf} \in \mathsf{dom}(A) \rightharpoonup \mathsf{dom}(A)$) maps each reading action to a writing action which it reads from.

From a machine configuration $\langle T; G\rangle$, a move from an arbitrary thread can transfer into a new machine configuration $\langle T'; G'\rangle$ if the newly constructed graph G' is legal under C11 memory model: $\mathsf{consistentC11}(G')$.

C. The Memory Model

1) Happens-Before Relation: We have so far introduced sb, mo, and rf. Now we describe the essential part of the memory model: synchronisations. Different from GPS and RSL, now fences can also form synchronisations. Our memory model is still simplified when compared with the standard [1] (for example, the subtle *release-sequence* is omitted).

We first introduce a derived relation *synchronised-with* ($\mathsf{sw} \subseteq \mathsf{dom}(A) \times \mathsf{dom}(A)$). As illustrated in Fig 5, a pair of release write and acquire read can synchronise. Relaxed atomics can also synchronise with the help of corresponding fences.

The idea of synchronisation in C11 is that when an event c is synchronised with another event b, i.e. $(b, c) \in \mathsf{sw}$, then b's observation about its preceding memory updates becomes visible to c (and its succeeding events) as well. Based on this, the heart of the C11 memory model, *happens-before* relation, can be defined as: $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$.

For instance, Fig 6 illustrates a 2-thread message passing program, where `msg` is the message we intend to pass from the first thread to the second and `flg` is used for synchronisation.

$$\begin{tikzcd}
a\colon \mathbb{W}(-,-,\mathtt{rel}) \arrow[r, "\mathsf{rf}", "\mathsf{sw}"'] & b\colon \mathbb{R}(-,-,\mathtt{acq})
\end{tikzcd}$$

$$\begin{tikzcd}
a\colon \mathbb{W}(-,-,\mathtt{rel}) \arrow[r, "\mathsf{rf}"'] \arrow[rd, "\mathsf{sw}"'] & c\colon \mathbb{R}(-,-,\mathtt{rlx}) \arrow[d, "\mathsf{sb}^{+}"] \\
 & b\colon \mathbb{F}(\mathtt{acq})
\end{tikzcd}$$

$$\begin{tikzcd}
a\colon \mathbb{F}(\mathtt{rel}) \arrow[d, "\mathsf{sb}^{+}"] \arrow[rd, "\mathsf{sw}"] & \\
c\colon \mathbb{W}(-,-,\mathtt{rlx}) \arrow[r, "\mathsf{rf}"] & b\colon \mathbb{R}(-,-,\mathtt{acq})
\end{tikzcd}$$

$$\begin{tikzcd}
a\colon \mathbb{F}(\mathtt{rel}) \arrow[d, "\mathsf{sb}^{+}"] \arrow[rd, "\mathsf{sw}"'] & b\colon \mathbb{R}(-,-,\mathtt{rlx}) \arrow[d, "\mathsf{sb}^{+}"] \\
c\colon \mathbb{W}(-,-,\mathtt{rlx}) \arrow[ru, "\mathsf{rf}"] & b\colon \mathbb{F}(\mathtt{acq})
\end{tikzcd}$$

Fig. 5: Four ways to form synchronization

$$\begin{tikzcd}
\mathtt{Thread\ 1:} & & \mathtt{Thread\ 2:} \\
a : \mathbb{W}(\mathtt{msg}, 42, \mathtt{rlx}) \arrow[d, "\mathsf{sb}"] \arrow[rrd, "\mathsf{hb}"] & & c : \mathbb{R}(\mathtt{flg}, 1, \mathtt{acq}) \arrow[d, "\mathsf{sb}"] \\
b : \mathbb{W}(\mathtt{flg}, 1, \mathtt{rel}) \arrow[rru, "\mathsf{rf}"] \arrow[rru, "\mathsf{sw}"'] & & d : \mathbb{R}(\mathtt{msg}, 42, \mathtt{rlx})
\end{tikzcd}$$

Fig. 6: Message passing using release write and acquire read

Both msg and flg are initialised as 0. When the acquire load c reads from (rf) the release store b, a synchronised-with (sw) relation is established between them. Consequently, information observed by the source store b is eligible to be shared with the reader c. In particular, this ensures d is aware that a has happened, thus it will not read the stale value 0.

On the other hand if either one or both of actions on flg are relaxed as shown in Fig 7, such sw relation fails to be established, which means the out-of-order executions allowed by the C11 standard may cause d to read the value 0 as well.

$$\begin{tikzcd}
\mathtt{Thread\ 1:} & & \mathtt{Thread\ 2:} \\
a : \mathbb{W}(\mathtt{msg}, 42, \mathtt{rlx}) \arrow[d, "\mathsf{sb}"] & & c : \mathbb{R}(\mathtt{flg}, 1, \mathtt{rlx}) \arrow[d, "\mathsf{sb}"] \\
b : \mathbb{W}(\mathtt{flg}, 1, \mathtt{rlx}) \arrow[rru, "\mathsf{rf}"] & & d : \mathbb{R}(\mathtt{msg}, ?, \mathtt{rlx})
\end{tikzcd}$$

Fig. 7: Failed message passing

2) Data-Race and Memory Error: C11 provides various levels of memory consistency orders, from the most strict *sequentially-consistent* sc to the most relaxed *non-atomic* na (which does not even ensure atomicity), as a handy feature for users to flexibly balance the efficiency and safety of their programs. However, one must remember that when two events concurrently access a non-atomic location and at least one of them is a write event, it will lead to a *data race*. The C11 standard declares that if an execution is data-race free, the non-atomic actions will perform as they are sequentially-consistent; otherwise, the result of execution is undefined. Another situation that will lead to an undefined result is a *memory error*, which happens when an event accesses a location that has not been allocated.

3) Axioms: Following Batty et al. [2], the C11 memory model is formulated as a set of axioms (over an event graph G), denoted as $\mathsf{consistentC11}(G)$, the definition of which is left for the report [9]. In the machine step layer of semantics, the execution of our program is restricted by the C11 memory model via the checking with consistentC11.

4) Thin-Air Read and the Strengthening of the Memory Model: Our core language includes relaxed atomic operations. However in the C11 memory model, relaxed atomics are known to have the *thin-air-read* issue [2], which refers to the problem that a cleverly designed program will allow a relaxed atomic read to return any value out of the thin air, without breaking the very few restrictions applied to relaxed atomics. This problem makes it impossible to rigorously reason about a program with relaxed atomics. To rule out thin-air reads, we follow the same approach as RSL [23], i.e. we add an extra axiom to the consistency check:

$$\begin{aligned}
&\mathsf{consistentC11}((A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf})) \triangleq \\
&\quad \ldots \\
&\wedge\ \mathsf{acyclic}(\mathsf{hb} \cup \{(\mathsf{rf}(a), a) \mid a \in A\}) \\
&\qquad \text{where } \mathsf{acyclic}(R) \triangleq \nexists x \in A.\ R^{+}(x, x).
\end{aligned}$$

III. The GPS Framework

Our proposed reasoning mechanism is built on top of the GPS framework [22], [20], which combines three concepts advocated by state-of-the-art concurrent program logics (e.g. [24], [8], [7], [3], [13], [6], [21], [17], [19], [4], [11]), namely ghost states, protocols and separation logic, and adapts them in a novel way to support modular weak memory reasoning. We shall first give a brief introduction about GPS, focusing on atomic writes/reads and escrows, which are essential for synchronisations.

A. Protocols for Atomic Locations

Following the C11 standard, atomic locations in GPS are meant to be read and written concurrently. Therefore it is difficult to make any stable assertions about the precise contents of an atomic location. GPS advocates per-location protocols to describe how the contents of each atomic location can evolve over time. A *state assertion* $\boxed{l : s}\boxed{\tau}$ indicates that an atomic location l is governed by the protocol τ, and is at least at state s. All possible state transition relations have to be defined in τ as a partial order $\sqsubseteq_{\tau}$; and in τ, *state interpretation* $\tau(s, z)$ for each state s also has to be specified.

The state assertions about atomic locations belong to *knowledge* in GPS, which refers to assertions that do not depend on ownership. State assertions are ownership independent because according to the C11 standard, atomic locations are meant to be accessed concurrently (without hb ordering) in different threads. Correspondingly, GPS assertions about their states can be present in different threads at the same time. Conversely, the assertions about non-atomic locations (i.e. $x \hookrightarrow v$) are not knowledge and must be owned by one thread at a time as concurrent access to them may raise data races. Knowledge is indicated by a modality $\square$, and GPS has useful rules to reason about knowledge:

$$\boxed{l : s}\boxed{\tau} \Rightarrow \square\, \boxed{l : s}\boxed{\tau}, \quad \square P \Rightarrow P \quad \text{and} \quad \square P \Leftrightarrow \square P * \square P$$

The first rule says that a state assertion can be transformed into its knowledge form. The second says knowledge can always be turned back into its normal assertion. And the third shows that knowledge can be duplicated and thus be shared.

A state interpretation $\tau(s, z)$ for a protocol τ governing a location l is an assertion specifying what must be true for a thread to be permitted to write z to l and thus change it to state s. A read action which reads from this write may retrieve this assertion. This approach elegantly captures the idea of synchronisations in the C11 standard. Intuitively, the write action happens before that read (as a synchronised-with relation is formed between them), so it signifies that the effect of any preceding actions (those happened-before the write) can be transmitted to the reading thread.

The rule for atomic (i.e. *acquire*) read in GPS is given as:

$$\frac{\boxed{\text{GPS-ATOMIC-LOAD}} \\ \forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z) * P \Rightarrow \Box Q}{\{\boxed{l : s \mid \tau} * P\}\ [l]_{\texttt{acq}}\ \{z.\ \exists s'.\ \boxed{l : s' \mid \tau} * P * \Box Q\}}$$

The possible writes that an atomic read can observe are quantified in the premise. Note that only assertions in knowledge form ($\Box Q$) can be gained, as it is possible for multiple threads to all read the location at the same state and thus gain the same assertion. Therefore if the assertion is not an ownership independent knowledge, data races may occur. The inclusion of the assertion P enables *rely-guarantee* reasoning through protocols [22].

The atomic (i.e. *release*) write rule in GPS is defined as:

$$\frac{\boxed{\text{GPS-ATOMIC-STORE}} \\ P \Rrightarrow \tau(s'', v) * Q \qquad \forall s' \sqsupseteq_\tau s.\ \tau(s', -) * P \Rightarrow s'' \sqsupseteq_\tau s'}{\{\boxed{l : s \mid \tau} * P\}\ [l]_{\texttt{rel}} := v\ \{\boxed{l : s'' \mid \tau} * Q\}}$$

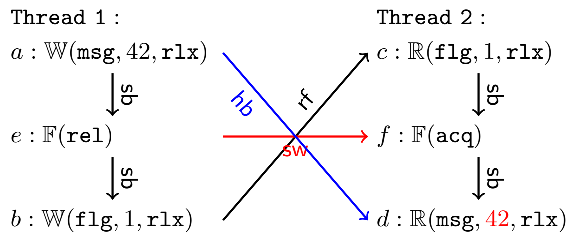
Note that from the precondition we only know the lower bound state for l is state s (i.e. the location l is at least at state s before the write takes place). Without knowing which exact state l might have possibly been moved to by environment actions prior to this write, here the write moves it to state s'' that is reachable from any state s' such that $s' \sqsupseteq_\tau s$. In the first premise, P is transformed to the state interpretation $\tau(s'', v)$ with some frame Q via a *ghost move* $\Rrightarrow$. Ghost moves are another important concept in GPS: they represent moves that only change logical states without affecting the actual machine states. Ghost moves can take place any time that suits the logic user's needs. They can do useful things like creating ghost assertions, packing and unpacking escrows, which we are going to discuss next.

B. Escrows for Non-Atomic Locations

According to the rule [GPS−ATOMIC−LOAD], only knowledge can be transmitted in synchronisations. However, very often we need to transfer the ownership of non-atomic locations. To do this, GPS allows them to be wrapped up into knowledge form and be retrieved at the right time, via the use of *escrows*. An escrow of the form $\sigma : P \rightsquigarrow Q$ can be considered as a safe-box protecting Q, and the key to open it is P (which is not duplicable). Ghost moves are used to pack and unpack escrows:

$$\frac{\boxed{\text{GPS-ESCROW-PACK}} \\ \sigma : P \rightsquigarrow Q}{Q \Rrightarrow [\sigma]} \qquad \frac{\boxed{\text{GPS-ESCROW-UNPACK}} \\ \sigma : P \rightsquigarrow Q}{P \wedge [\sigma] \Rrightarrow Q}$$

A packed escrow $[\sigma]$ is an ownership-independent assertion and can also be used in its knowledge form: $[\sigma] \Leftrightarrow \Box[\sigma]$.



Thread 1 :
Thread 2 :
$a : \mathbb{W}(\texttt{msg}, 42, \texttt{rlx})$
$c : \mathbb{R}(\texttt{flg}, 1, \texttt{rlx})$
sb
$e : \mathbb{F}(\texttt{rel})$
hb
rf
sw
$f : \mathbb{F}(\texttt{acq})$
sb
$b : \mathbb{W}(\texttt{flg}, 1, \texttt{rlx})$
$d : \mathbb{R}(\texttt{msg}, 42, \texttt{rlx})$

Fig. 8: Message passing using relaxed atomics with fences

The "key" P is consumed once it has been used to unpack an escrow. Therefore instead of using physical resources, ghost assertions are introduced to describe the permissions to unpack an escrow. A ghost assertion $\boxed{\gamma : t : \mu}$ says there is a ghost variable γ, whose value is ghost permission t drawn from some *partial commutative monoid* (PCM) μ. New ghost t can appear out of thin air, with a fresh identity: $\texttt{true} \Rrightarrow \exists\gamma.\ \boxed{\gamma : t : \mu}$.

A special kind of permission is *token* Tok. Tok has only two kinds of permissions: ξ is the unit and represents empty permission; and $\diamond$ represents for full permission. They are usually written as $\boxed{\gamma : \xi}$ and $\boxed{\gamma : \diamond}$ for short.

IV. Reasoning about Relaxed Atomics and Fences

We now present our key proposal: a program logic that supports the reasoning of a bigger class of C11 programs (than GPS), including relaxed atomics and release-acquire fences.

A. Two New Types of Assertions

We would like to handle relaxed atomic operations in a similar way as release and acquire atomics are treated in GPS, since they are also applied on atomic locations. Moreover, we would like to ensure that the idea of per-location protocols works for all of them. However, as defined in §II-C and illustrated in Fig 6 and Fig 7, one challenge is that relaxed atomics form synchronised-with relations *differently* from release-acquire atomics: a sw relation is automatically set up when an acquire load operation reads from a release store operation; but for relaxed atomics the C11 standard states that the sw relation can only be established with the help of fences[2]. Fig 8 shows that fences are needed to restore the sw and thus the hb relations for the example in Fig 7.

We interpret these restrictions as (i) a relaxed atomic store operation can only transmit the information that has been marked as shareable by a preceding release fence; and (ii) a relaxed load should not put the knowledge gained from its loading source to the current state, instead it should mark the knowledge as not yet available and await a succeeding acquire fence to transform them to normal knowledge form. To cater for these new scenarios, we introduce two new types of assertions: *shareable assertions* $\langle P \rangle$, and *waiting-to-be-acquired assertions* $\boxtimes P$.

Intuitively $\langle P \rangle$ indicates that P is shareable. That is, it can be transmitted to others (even by a relaxed store operation). $\boxtimes P$ signifies that knowledge received by a relaxed load is not yet available according to the C11 standard. Reading, updating or re-transmitting $\boxtimes P$ is not permitted until an acquire fence transforms it into normal knowledge $\Box P$.

[2]Or via release sequences, which we do not consider in this paper.

The formal semantics for these new assertions and their properties will be presented later in Sec VI. It is worth noting here that unlike $\Box P \Rightarrow P$, the property $\boxtimes P \Rightarrow P$ does not hold, as according to the C11 standard, $\boxtimes$ can only be stripped off by using an acquire fence. Moreover, unlike the knowledge symbol $\Box$ that can be nested, the nesting of shareable or waiting-to-be-acquired assertions is not allowed. As otherwise, if an assertion like $\boxtimes\langle P\rangle$ is permitted, after an acquire fence it immediately becomes a shareable assertion, which clearly violates the C11 standard.

It is also worth noting that, in order to prevent improper assertions (like $\boxtimes P$ or $\langle P\rangle$) from being included in state interpretations for atomic variables, we require that all state interpretations must be "normal" assertions, i.e. $\forall \tau, s, V.\ \mathsf{normal}(\tau(s, V))$, where $\mathsf{normal}(P) \triangleq P \Rrightarrow \mathtt{false} \vee \langle P\rangle \not\Rrightarrow \mathtt{false}$. A similar restriction is applied to the assertions used in escrows: for each escrow $\sigma : P \rightsquigarrow P'$, we require $\mathsf{normal}(P)$ and $\mathsf{normal}(P')$.

B. New Verification Rules

With the new forms of knowledge and assertions, we can now ensure that knowledge will be distributed in a controlled manner both from the starting point (a store operation) and at the finishing point (a load operation). We present a number of newly-designed verification rules in Fig 9. The rules that are inherited from GPS without change and the rule for `FAI` are left for the technical report [9].

$$\left[\underline{\textsc{release–store}}\right]$$

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

$$\left[\underline{\textsc{relaxed–store}}\right]$$

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

$$\left[\underline{\textsc{release–fence}}\right]$$

$$\frac{\langle P\rangle \not\Rrightarrow \mathtt{false}}{\{P\}\ \mathtt{fence}_{\mathtt{rel}}\ \{\langle P\rangle\}}$$

$$\left[\underline{\textsc{acquire–load}}\right]$$

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

$$\left[\underline{\textsc{relaxed–load}}\right]$$

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

$$\left[\underline{\textsc{acquire–fence}}\right]$$

$$\{\boxtimes P\}\ \mathtt{fence}_{\mathtt{acq}}\ \{\Box P\}$$

$$\left[\underline{\textsc{cas}}\right]$$

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

Fig. 9: New verification rules

Being atomic store operations, both release and relaxed stores can transmit some extra information to their readers. But according to the standard and as pointed out in their instrumented semantics we discussed before, the scopes of information that are available for them to release are different. This difference is captured by our rules. Being a store using a weaker memory order, a relaxed store can only use the assertion P_2 that is marked as shareable in its precondition to imply the interpretation of the state it is going to write, i.e. it can only transmit the things that are already marked as shareable. Meanwhile, a release store uses a general assertion P, which is not necessarily to be a shareable assertion, to ghostly imply the state interpretation it needs. Note that P can also contain shareable assertions, in which case the following $\left[\underline{\textsc{unshare}}\right]$ ghost move becomes handy if the normal form of these assertions is needed to imply the state interpretation:

$$\left[\underline{\textsc{unshare}}\right] : \langle P\rangle \Rrightarrow P$$

This ghost move allows us to convert a shareable assertion back to its previous form (where resources were held in the local part instead of the shareable part). The assertion P_1 in the $\left[\underline{\textsc{relaxed–store}}\right]$ rule is used to reduce the possible intermediate environment moves we need to consider.

A release fence marks resources that are ready to be shared. Our $\left[\underline{\textsc{release–fence}}\right]$ rule shows that an assertion P in its precondition is transformed into a shareable assertion after the fence (assuming it is possible to do so). The sanity check in the premise prevents `false` from being gained in the postcondition. Note that if the precondition P is already a shareable assertion or a waiting-to-be-acquired assertion (i.e. $\langle P\rangle \Rightarrow \mathtt{false}$), the release-fence would act like the skip action, and the postcondition would remain as P (according to the frame rule in Separation Logic).

For atomic loads, the $\left[\underline{\textsc{acquire–load}}\right]$ rule in GPS is compatible with our new setting. Note that the knowledge it retrieves from its load source is directly put in the postcondition. However as we have discussed, the knowledge gained by a relaxed load should not be considered as immediately available to the current thread (for reading, updating or re-transmitting). Therefore, in our new $\left[\underline{\textsc{relaxed–load}}\right]$ rule, the knowledge $\Box Q$ the load gains is marked as waiting-to-be-acquired knowledge $\boxtimes Q$ in its post condition. One can then use the $\left[\underline{\textsc{acquire–fence}}\right]$ rule to turn an acquirable knowledge into a normal one.

$\mathtt{CAS}(l, v_o, v)$ (compare and swap) is an important synchronisation operation, which is widely used in various lock algorithms. It performs the following things in one atomic step: firstly it loads from l, and compares the value it gets with the expected value v_o; if they are equal, it updates l with a new value v and returns 1 indicating its success, otherwise returns 0. The CAS in our $\left[\underline{\textsc{cas}}\right]$ rule is a release-acquire CAS, i.e. in the case of success (corresponding to the first premise) it behaves like a release store, and in the case of fail (corresponding to the second premise) it behaves like an acquire load that read some value other than v_o. Moreover, in the case of success, it can retrieve non-knowledge assertions from the interpretation of the state s'. As we require that all state interpretations must be normal assertions (or `false`), we do not need to be concerned that improper assertions, like shareable assertions that can be immediately re-transmitted by any following relaxed stores without a release fence, will be retrieved from $\tau(s', v_o)$ and left over in Q.

V. Illustrative Example

We illustrate our reasoning logic using the racy program shown in Fig 10. We first show how our logic can detect the data race and how it is unable to prove the program to be correct. We then show that after resolving the race by properly adding fences, our logic can prove it successfully.

```
                let x = alloc(1) in
                let y = alloc(1) in
                let z = alloc(1) in
                [x]na := 0;  [y]rel := 0;  [z]rel := 0;
[x]na := 1;   || repeat [y]rlx end; || repeat [z]acq end;
[y]rel := 1;  || [z]rlx := 1;       || [x]na := 2
```

Fig. 10: A program with a data race

Note that a message $\mathtt{x} \hookrightarrow 1$ is created in thread 1, and is passed to thread 2 by the release store to `y`. Thread 2 performs a relaxed store to `z`, intending to retransmit this message to thread 3, where the ownership of `x` is demanded to perform the non-atomic write.

According to the C11 standard, this program contains a data race as it is not properly synchronised. Despite the fact that in thread 1 the store operation to `y` is release atomic, the load operation in thread 2 that reads from it is relaxed. Without a subsequent acquire fence, no synchronisation can be established between thread 1 and 2. Similarly, though the acquire load operation of `z` in thread 3 reads from the store operation in thread 2, the two threads are not synchronised as the store operation is relaxed and lacking a release fence before it. Therefore, the chain of happens before (hb) relation breaks between thread 1 and 3. Without having a happens before relation, the non-atomic writes to `x` in thread 1 and 3 produce a data race.

We show in Fig 11a that, with the help of the two new types of assertions, our logic can detect the failure of synchronisation, and will not prove the racy program to be correct. First, we define the escrow for `x` and protocols for `y` and `z`, where each of `y` and `z` has only two protocol states 0 and 1, and $0 \sqsubseteq_{\mathbf{Prot}(l)} 1$ for $l \in \{\mathtt{y}, \mathtt{z}\}$:

$$\mathbf{XE} : \boxed{\gamma : \diamond} \rightsquigarrow x \hookrightarrow 1$$
$$\mathbf{Prot}(l)(0, v) \triangleq v{=}0 \quad \mathbf{Prot}(l)(1, v) \triangleq v{=}1 \wedge \Box[\mathbf{XE}] \qquad l \in \{\mathtt{y}, \mathtt{z}\}$$

As shown in Fig 11a, the verification could not be finished in thread 2. Even though in thread 1 the message about `x` is packed via ghost move from (1.2) to (1.3), and put into `y`'s state interpretation as a knowledge, the relaxed load operation of `y` in thread 2 can only extract the knowledge in a waiting-to-be-acquired form $\boxtimes[\mathbf{XE}]$ according to $\lceil$RELAXED–LOAD$\rceil$. Without subsequent acquire and release fences, this waiting-to-be-acquired knowledge is kept in this form and cannot be used to entail the required precondition for the next command $[\mathtt{z}]_{\mathtt{rlx}} := \mathtt{1}$, in which the packed escrow is expected to be in the shareable form $\langle \Box[\mathbf{XE}] \rangle$ according to the rule $\lceil$RELAXED–STORE$\rceil$.

To resolve the data race in this program, as shown in Fig 11b, an acquire fence and a release fence are needed to be inserted between the relaxed load operation of `y` and the release operation to `z` in thread 2, which will change the waiting-to-be-acquired knowledge into a normal knowledge and then a shareable knowledge before the relaxed store operation to `z` transfers it to thread 3.

It is worth noting that our logic supports modular reasoning. The verification of thread 1 and 3 can be conducted separately despite the error in the original thread 2.

We have also applied our reasoning logic to a number of more challenging programs as documented in the report [9].

VI. Resource Model

In this section we shall first briefly introduce the GPS resource model and then present our new resource model which is built on the GPS one.

A. GPS Resources

In GPS, resources are used to logically represent computation states. A resource $r \in \mathit{Resource}$ is a triple (Π, g, Σ) where the *physical location map* Π maps each location to either a value (for non-atomics) or a protocol and state (for atomics), the *ghost identity map* g keeps the ghost values, and the *known escrow set* Σ contains all escrows available. Resources form a PCM with composition $\oplus$. Some useful definitions are:

$$\begin{aligned} \mathtt{emp} &\triangleq ((\lambda n.\ \bot), (\lambda \mu.\ \lambda n.\ \epsilon_\mu), \emptyset) \\ r \leq r' &\triangleq \exists r''.\ r \oplus r'' = r' \\ r \# r' &\triangleq r \oplus r' \text{defined} \end{aligned}$$

Each proposition P in GPS is interpreted as a set of resources, i.e. $[\![P]\!] \subseteq \mathit{Resource}$. Moreover, the interpretation satisfies the following property:

$$\forall r \in [\![P]\!].\ \forall r' \# r.\ r \oplus r' \in [\![P]\!]$$

GPS also introduces a rely-guarantee-styled instrumented semantics for all actions. Let us take the release store operation as an example. Given a resource r_{pre} that meets the pre-condition of the write, and assuming resource r is the actual resource used by the write (note r can be different from r_{pre} as the environment may also make changes prior to the write), the effect of this atomic write can be illustrated by its guarantee definition as shown below, where r_{sb} is the resource that will be passed down to its sb successor in the execution graph and r_{rf} is the resource to be transmitted to its reader:

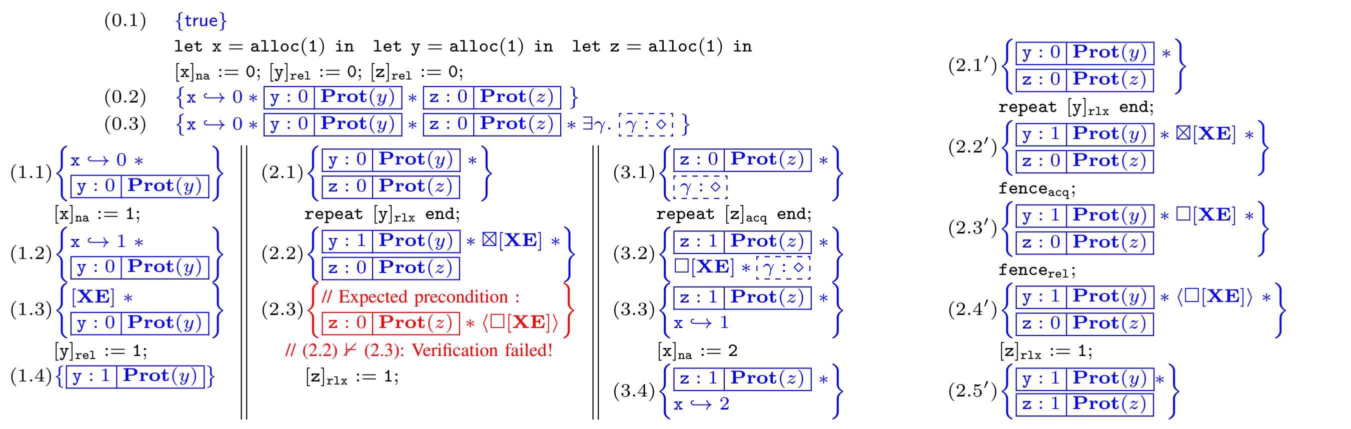
$$\begin{aligned} &(r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(r_{\mathsf{pre}}, r, \mathbb{W}(l, V, \mathtt{rel})) \text{ if} \\ &\exists \tau, s, S. r_{\mathsf{rf}} \in \mathsf{interp}(\tau)(s, V) \\ &\wedge r_{\mathsf{rf}} \oplus r_{\mathsf{sb}} = r[l := \mathtt{at}(\tau, S \cup \{s\})] \wedge r_{\mathsf{sb}}[l] = r_{\mathsf{rf}}[l] \\ &\wedge (r[l] = \mathsf{uninit} \wedge S = \emptyset \vee r[l] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s) \\ &\wedge \forall r_E. \begin{pmatrix} \exists \tau, s', V'.\ r_E \in \mathsf{interp}(\tau)(s', V') \\ \wedge\ r_{\mathsf{pre}}[l] \sqsubseteq_{\mathsf{at}} \mathcal{R}_E[l] \equiv_{\mathsf{at}} \mathtt{at}(\tau, S \cup \{s'\}) \\ \wedge\ r_{\mathsf{pre}} \# r_E \end{pmatrix} \\ &\Rightarrow r_E[l] \sqsubseteq_{\mathsf{at}} r_{\mathsf{rf}}[l] \end{aligned}$$

Note $\mathsf{interp}(\tau)(s, V)$ denotes the semantics of the state interpretation under the new state s, namely $[\![\tau(s, V)]\!]$, which carries the information we intend to transmit through this atomic write. The notation $r[l]$ is short for $r.\Pi(l)$, which is the value of the physical location l. For an atomic location, this is an atomic protocol value in the form of $\mathtt{at}(\tau, S)$, where τ is the protocol type governing that location and S is a trace of states the location has gone through. Some relations between these protocol values are defined as:

$$\begin{aligned} \mathtt{at}(\tau, S) \sqsubseteq_\tau \mathtt{at}(\tau, S') &\triangleq \forall s \in S.\ \exists s' \in S'.\ s \sqsubseteq_\tau s' \\ \pi \equiv_{\mathsf{at}} \pi' &\triangleq \pi \sqsubseteq_\tau \pi' \wedge \pi' \sqsubseteq_\tau \pi \end{aligned}$$

The assertion-level ghost move is defined in terms of resource-level ghost moves:

$$P \Rrightarrow Q \triangleq \forall r \in [\![P]\!].\ r \Rrightarrow [\![Q]\!]$$



```
(0.1)  {true}
       let x = alloc(1) in  let y = alloc(1) in  let z = alloc(1) in
       [x]_na := 0; [y]_rel := 0; [z]_rel := 0;
(0.2)  {x ↪ 0 ∗ [y : 0 | Prot(y)] ∗ [z : 0 | Prot(z)] }
(0.3)  {x ↪ 0 ∗ [y : 0 | Prot(y)] ∗ [z : 0 | Prot(z)] ∗ ∃γ. [γ : ◇] }

(1.1) {x ↪ 0 ∗ [y : 0 | Prot(y)]}        (2.1) {[y : 0 | Prot(y)] ∗ [z : 0 | Prot(z)]}        (3.1) {[z : 0 | Prot(z)] ∗ [γ : ◇]}
[x]_na := 1;                             repeat [y]_rlx end;                                  repeat [z]_acq end;
(1.2) {x ↪ 1 ∗ [y : 0 | Prot(y)]}        (2.2) {[y : 1 | Prot(y)] ∗ ⊠[XE] ∗ [z : 0 | Prot(z)]} (3.2) {[z : 1 | Prot(z)] ∗ □[XE] ∗ [γ : ◇]}
(1.3) {[XE] ∗ [y : 0 | Prot(y)]}         (2.3) {// Expected precondition :                    (3.3) {[z : 1 | Prot(z)] ∗ x ↪ 1}
[y]_rel := 1;                                   [z : 0 | Prot(z)] ∗ ⟨□[XE]⟩}                  [x]_na := 2
(1.4) {[y : 1 | Prot(y)]}                // (2.2) ⊬ (2.3): Verification failed!               (3.4) {[z : 1 | Prot(z)] ∗ x ↪ 2}
                                         [z]_rlx := 1;

(2.1′) {[y : 0 | Prot(y)] ∗ [z : 0 | Prot(z)]}
repeat [y]_rlx end;
(2.2′) {[y : 1 | Prot(y)] ∗ ⊠[XE] ∗ [z : 0 | Prot(z)]}
fence_acq;
(2.3′) {[y : 1 | Prot(y)] ∗ □[XE] ∗ [z : 0 | Prot(z)]}
fence_rel;
(2.4′) {[y : 1 | Prot(y)] ∗ ⟨□[XE]⟩ ∗ [z : 0 | Prot(z)]}
[z]_rlx := 1;
(2.5′) {[y : 1 | Prot(y)] ∗ [z : 1 | Prot(z)]}
```

(a) Failed verification of the racy program (b) Verification of the fixed thread 2

Fig. 11: Verification of Relayed Message Passing

For instance, the escrow packing rule is validated by the following resource-level ghost move:

$$\frac{\mathsf{interp}(\sigma) = (\llbracket P \rrbracket, \llbracket P' \rrbracket) \qquad r' \in \llbracket P' \rrbracket}{(\Pi, g, \Sigma) \oplus r' \Rrightarrow \lfloor (\Pi, g, \Sigma \cup \{\sigma\}) \rfloor}$$

Note that the escrow's interpretation $\mathsf{interp}(\sigma) = (\llbracket P \rrbracket, \llbracket P' \rrbracket)$ requires that $\llbracket P \rrbracket * \llbracket P' \rrbracket = \emptyset$. Note also that $\lfloor r \rfloor$ is defined as $\{r \oplus r' \mid r' \in \mathit{Resource}\}$.

B. The New Resource Model

To deal with the two new types of assertions, we extend the GPS resource model to a more expressive one by lifting resources to resource triples:

$$\mathit{ResTriple} \triangleq \{(r_1, r_2, r_3) \mid r_1, r_2, r_3 \in \mathit{Resource} \wedge r_1 \oplus r_2 \oplus r_3 \text{ defined}\}$$

For each resource triple $\mathcal{R} = (r_1, r_2, r_3)$ we use $\mathcal{R}[\mathsf{L}]$ to denote r_1,$\mathcal{R}[\mathsf{S}]$ for r_2, and $\mathcal{R}[\mathsf{A}]$ for r_3, representing resp. its *local*, *shareable*, and *waiting-to-be-acquired* component.

Like resources, *ResTriple* also forms a PCM. The composition operation $\oplus$ is defined point-wise; the compatibility can be defined as:

$$\mathcal{R} \# \mathcal{R}' \triangleq \mathcal{R} \oplus \mathcal{R}' \text{ defined}$$

EMP is defined as a resource triple comprising only empty resources: $\mathtt{EMP} \triangleq (\mathtt{emp}, \mathtt{emp}, \mathtt{emp})$. The semantics for propositions is lifted to the *ResTriple* model as well. The interpretation $\llbracket P \rrbracket$ of an assertion P is a set of resource triples satisfying the property:

$$\forall \mathcal{R} \in \llbracket P \rrbracket .\ \forall \mathcal{R}' \# \mathcal{R}.\ \mathcal{R} \oplus \mathcal{R}' \in \llbracket P \rrbracket$$

For any basic assertion P and resource triple $\mathcal{R}$, only the local part of $\mathcal{R}$ is needed when checking $\mathcal{R} \in \llbracket P \rrbracket$. For example,

$$\mathcal{R} \in \llbracket \boxed{l : s \mid \tau} \rrbracket \Leftrightarrow \exists S.\ \mathcal{R}[\mathsf{L}].\Pi(l) = \mathtt{at}(\tau, S) \wedge s \in S$$

Composed assertions like separating conjunction are directly lifted up to use resource triples:

$$\mathcal{R} \in \llbracket P_1 * P_2 \rrbracket \Leftrightarrow \exists \mathcal{R}_1, \mathcal{R}_2.\ \mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2 \wedge \mathcal{R}_1 \in \llbracket P_1 \rrbracket \wedge \mathcal{R}_2 \in \llbracket P_2 \rrbracket$$

The semantics for synchronisation related assertions, namely knowledge, shareable assertion and waiting-to-be-acquired assertions are defined as:

$$\mathcal{R} \in \llbracket \square P \rrbracket \Leftrightarrow |(\mathcal{R}[\mathsf{L}], \mathtt{emp}, \mathtt{emp})| \in \llbracket P \rrbracket$$
$$\mathcal{R} \in \llbracket \langle P \rangle \rrbracket \Leftrightarrow (\mathcal{R}[\mathsf{S}], \mathtt{emp}, \mathtt{emp}) \in \llbracket P \rrbracket$$
$$\mathcal{R} \in \llbracket \boxtimes P \rrbracket \Leftrightarrow (\mathcal{R}[\mathsf{A}], \mathtt{emp}, \mathtt{emp}) \in \llbracket \square P \rrbracket$$

Note the stripping $|\mathcal{R}|$ is a lifted version of the GPS stripping, i.e, $|(r_1, r_2, r_3)| \triangleq (|r_1|, |r_2|, |r_3|)$. [3]

Under the new resource model, the following properties hold. Note properties for knowledge that hold in GPS are all preserved in the new model but are omitted here.

$\langle P \rangle * \langle Q \rangle$	$\Leftrightarrow \langle P * Q \rangle$	$\boxtimes P$	$\Leftrightarrow \boxtimes P * \boxtimes P$
$\square \langle P \rangle$	$\Rightarrow \mathtt{false}$ if $\mathtt{EMP} \notin \llbracket P \rrbracket$	$\boxtimes \langle P \rangle$	$\Rightarrow \mathtt{false}$ if $\mathtt{EMP} \notin \llbracket P \rrbracket$
$\langle \boxtimes P \rangle$	$\Rightarrow \mathtt{false}$ if $\mathtt{EMP} \notin \llbracket P \rrbracket$	$\square \boxtimes P$	$\Rightarrow \mathtt{false}$ if $\mathtt{EMP} \notin \llbracket P \rrbracket$
$\boxtimes \boxtimes P$	$\Rightarrow \mathtt{false}$ if $\mathtt{EMP} \notin \llbracket P \rrbracket$	$\langle \langle P \rangle \rangle$	$\Rightarrow \mathtt{false}$ if $\mathtt{EMP} \notin \llbracket P \rrbracket$

1) Ghost Moves: As in GPS, assertion-level ghost moves are defined in terms of resource-level ghost moves: $P \Rrightarrow Q \triangleq \forall \mathcal{R} \in \llbracket P \rrbracket .\ \mathcal{R} \Rrightarrow \llbracket Q \rrbracket$. The only difference is that resource triples are now used in the resource level. For instance, the resource level escrow packing ghost move is changed to:

$$\frac{\mathsf{interp}(\sigma) = (\llbracket P \rrbracket, \llbracket P' \rrbracket) \quad \mathcal{R}' \in \llbracket P' \rrbracket \quad \mathcal{R}[\mathsf{L}] = (\Pi, g, \Sigma)}{\mathcal{R} \oplus \mathcal{R}' \Rrightarrow \lfloor ((\Pi, g, \Sigma \cup \{\sigma\}), \mathcal{R}[\mathsf{S}], \mathcal{R}[\mathsf{A}]) \rfloor}$$

Based on this definition, we can obtain the same escrow packing rule as that in GPS.

In addition to all ghost moves inherited from GPS, we also propose a new one:

$$\frac{\mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}] \oplus r \quad \mathcal{R}'[\mathsf{S}] \oplus r = \mathcal{R}[\mathsf{S}] \quad \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}]}{\mathcal{R} \Rrightarrow \lfloor \mathcal{R}' \rfloor}$$

This resource-level ghost move gives us the assertion-level ghost move rule [UNSHARE] (shown in Sec IV).

2) Rely/Guarantee Definitions: Following the GPS approach, we define the instrumented semantics for all actions in the rely/guarantee style (more details are left for the report [9]). But instead of manipulating resources, our actions work on resource triples, which is more expressive and allows us to describe the subtle difference among various kinds of actions. As an example, the guarantee definitions for release and relaxed writes are illustrated in Fig 12.

[3]In GPS, $|r|$ represents the duplicable part of r: $r = r \oplus |r|$. For duplicable items in r, like atomic values and the known escrow set, stripping keeps them unchanged. That is, we have $|r|.\Sigma = r.\Sigma$, and if $l_{\mathtt{at}}$ is an atomic location in r we have $|r|.\Pi(l_{\mathtt{at}}) = r.\Pi(l_{\mathtt{at}})$. For non-duplicable items, like non-atomic values, stripping removes them. For example, if $l_{\mathtt{na}}$ is a non-atomic location in r we have $|r|.\Pi(l_{\mathtt{na}}) = \bot$. The value $\diamond$ of ghost type Tok is also not duplicable, and all ghost locations of type Tok will be set as empty after stripping: $|r|.g(\mathsf{Tok})(-) = \xi$.

$$
\begin{array}{l}
(\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}, \mathbb{W}(l, V, \mathtt{rel})) \text{ if} \\
\quad \exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}. \\
\quad \left(\begin{array}{l}
\exists r_1, r_2.\ \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \\
\wedge\ \mathcal{R}[\mathsf{L}] = r_1 \oplus r_2 \wedge r_2 \le r_{\mathsf{rf}} \wedge \mathcal{R}'[\mathsf{L}] = r_1[l := \mathtt{at}(\tau, S \cup \{s\})] \\
\wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}] \oplus r_2[l := \mathtt{at}(\tau, S \cup \{s\})]
\end{array} \right) \\
\quad \wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V) \wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp}) \\
\quad \wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}' \\
\quad \ldots
\end{array}
$$

(a) New guarantee condition for release write

$$
\begin{array}{l}
(\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}, \mathbb{W}(l, V, \mathtt{rlx})) \text{ if} \\
\quad \exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}. \\
\quad \left(\begin{array}{l}
\mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \\
\wedge\ \mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}][l := \mathtt{at}(\tau, S \cup \{s\})] \\
\wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}][l := \mathtt{at}(\tau, S \cup \{s\})]
\end{array} \right) \\
\quad \wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V) \wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp}) \\
\quad \wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}' \\
\quad \ldots
\end{array}
$$

(b) Guarantee condition for relaxed write

Fig. 12: Guarantee conditions for release write vs relaxed write

Note that a release write can move a resource (r_2) from $\mathcal{R}[\mathsf{L}]$ to the shareable part $\mathcal{R}[\mathsf{S}]$ and transmit it, while the relaxed write can only use the resource already in the shareable component.

VII. Conclusion

We present a verification logic for weak memory programs, by enhancing the GPS mechanism with two new forms of assertions: shareable assertions $\langle P \rangle$ and waiting-to-be-acquired assertions $\boxtimes P$. This change enables us to control more precisely the synchronisations that happen between threads, making the reasoning about relaxed atomics and fences possible.

Our work is closely related to GPS [22] and RSL [23], both of which focus on program verification under the C11 weak memory model. RSL was intended to provide support for reasoning about release-acquire accesses in the style of Concurrent Separation Logic (CSL) [18]. Our logic inherits several ideas from GPS, including per-location protocols and escrows, which are also relevant with a previous work [21]. Another important concept we borrow from GPS are ghost resources as PCMs. This idea is related with [5], [10], [14], and a recent work [11].

We are currently working on the mechanised soundness proof in Coq [16] for our reasoning logic, in the style of the GPS encoding [22]. Future work includes the incorporation of *release sequence* and the consideration of more memory orders like *consume read*. The most recent work [20] demonstrates the power of GPS in reasoning about real code and inspires us to apply our logic to more real code.

Acknowledgement The work was supported in part by NSFC No. 61373033 and SZSTI No. JCYJ20141819354611 7.

References

[1] ISO/IEC 9899:2011. Programming Language C. 2011.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ Concurrency. pages 55–66, 2011.

[3] 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 *International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, pages 23–42, 2009.

[4] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, pages 207–231. 2014.

[5] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *ACM POPL*, pages 287–300, 2013.

[6] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, pages 504–528, 2010.

[7] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *ESOP*, pages 363–377, 2009.

[8] X. Feng. Local Rely-guarantee Reasoning. In *ACM POPL*, pages 315–327, 2009.

[9] M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira. Reasoning about Fences and Relaxed Atomics (Technical Report), 2015. School of Computing, Teesside University.

[10] J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, pages 377–396, 2012.

[11] 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 *ACM POPL*, pages 637–650, 2015.

[12] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[13] K. R. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222, 2009.

[14] R. Ley-Wild and A. Nanevski. Subjective Auxiliary State for Coarse-grained Concurrency. In *ACM POPL*, pages 561–574, 2013.

[15] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *ACM POPL*, pages 378–391, 2005.

[16] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project., 2014. Version 8.4pl6. URL: http://coq.inria.fr.

[17] A. Nanevski, R. Ley-Wild, I. Sergey, and G. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*, pages 290–310. 2014.

[18] P. W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[19] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, pages 149–168. 2014.

[20] J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying Read-Copy-Update in a Logic for Weak Memory. In *ACM PLDI*, Portland, OR, USA, 2015.

[21] A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency. In *ICFP*, pages 377–390, 2013.

[22] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *ACM OOPSLA*, pages 691–707, 2014.

[23] V. Vafeiadis and C. Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *ACM OOPSLA*, pages 867–884, 2013.

[24] V. Vafeiadis and M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *18th International Conference on Concurrency Theory (CONCUR'07)*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271, 2007.