Owicki-Gries Reasoning for RC11*

Sadegh Dalvandi[1], Simon Doherty[2], Brijesh Dongol[1], and Heike Wehrheim[3]

[1] University of Surrey, Guildford, UK
[2] University of Sheffield, Sheffield, UK
[3] Paderborn University, Paderborn, Germany

Abstract. Owicki-Gries reasoning for concurrent programs uses Hoare logic together with an *interference freedom* rule for concurrency. Prior works on adapting Owicki-Gries reasoning to weak memory have proposed changes to both the sequential and interference-freedom proof obligations. In this paper, we develop a new proof calculus for the RC11 memory model (a fragment of C11 with both relaxed and release-acquire accesses) that allows all Owicki-Gries proof rules for compound statements, including non-interference, to remain unchanged. Our proof method features novel assertions specifying *thread-specific views* on the state of programs. This is combined with a set of Hoare logic rules that describe how these assertions are affected by atomic program steps. We demonstrate the utility of our proof calculus by verifying a number of standard C11 litmus tests and Peterson's algorithm adapted for C11. Our proof calculus and its application to program verification have been fully mechanised in the theorem prover Isabelle.

1 Introduction

In 1976, Susan Owicki and David Gries proposed an extension of Hoare's axiomatic reasoning technique [14] to concurrent programs [26]. Their proof calculus allows one to reason about concurrent programs with shared variables via a number of proof rules, including the rules for sequential programs as introduced by Hoare plus an additional proof rule for concurrent composition. This composition rule basically allows for the conjunction of pre- and post-conditions of the process' individual proofs, given that their proof outlines are *interference free*. Interference freedom requires that an assertion in the proof of one process cannot be invalidated by a statement in another process, when executed under the statement's precondition.

Today, concurrent programs are run on multi-core processors. Multi-core processors come with *weak memory models* specifying the execution behaviour of concurrent programs. Reasoning consequently needs to be adapted to the memory model under consideration. Owicki-Gries reasoning is, however, fixed to the memory model of *sequential consistency* (SC) [22], and is unsound for weak

* Dalvandi, Doherty and Dongol are supported by EPSRC grants EP/R032351/1, EP/R032556/2, EP/R019045/2; Wehrheim by DFG grant WE 2290/12-1.

memory models. Recent research has thus worked towards new sound proof calculi for concurrent programs. Most often, such approaches involve concurrent separation logics (e.g., GPS and RSL [33,15]). These techniques constitute a radical departure from the (relatively) small and easy proof calculus of Owicki and Gries, further extending already complex logics. A proposal for a (rely-guarantee variant of) the Owicki-Gries proof system has been made by Lahav and Vafeiadis [21], however, requiring a strenghtened non-interference check.

In this paper, we develop a proof method based on the Owicki-Gries proof calculus, keeping all of the original proof rules including the non-interference check unchanged. Our technique introduces a set of basic axioms to cope with memory accesses (reads, writes, read-modify-writes) and simple assertions that describe the current configuration of the weak memory state. Our proof calculus targets the weak memory model of the C11 programming language [8]. Here, we deal with the release-acquire-relaxed (RC11) fragment of C11 (thereby going further than prior work on Owicki-Gries reasoning for C11 [21]).

The key idea of our approach is the usage of novel assertions, which allow to specify *thread-specific* views on shared variables. We also include a specific assertion containing a modality for release-acquire (RA) synchronisation, capturing particularities of C11 RA message passing. The use of non-standard assertions as a consequence necessitates the introduction of new rules of assignment, formalising the effect of assignments on assertions.

We build our proof calculus on top of an *operational* semantics for RC11. The semantics is a mixture of our operational semantics proposed in [11] (for RAR) and Kaiser et al.'s semantics [15] for RA plus non-atomics. Correctness of this novel proposal is shown by proving it to coincide with the semantics in [11] which in turn has been proven to coincide with the standard axiomatic semantics of Batty et al. [8]. We have formalised our semantics within the theorem prover Isabelle [27] and mechanically proved soundness of all of our new rules for C11 assertions. Moreover, we provide mechanical proofs[4] of several litmus tests from the literature (message passing, load buffering, read-read coherence) as well as a version of Peterson's algorithm adapted for C11 memory [11,34].

The paper is organised as follows. In the next section we start with an example explaining the behaviour of concurrent programs on C11, motivating our novel assertions. Section 3 defines the syntax of RC11 programs and Section 4 its semantics. We present the proof calculus and its novel assertions in Section 5 via proofs of correctness for some standard litmus tests and Peterson's algorithm. Section 6 describes our Isabelle mechanisation, Section 7 discusses related work and the last section concludes.

2 Deductive Reasoning for Weak Memory

In this section, we illustrate the basic principles of C11 synchronisation and our verification method by considering the message-passing example (Figs. 1 and 2).

[4] The Isabelle files may be downloaded from: `brijeshdongol.github.io/mechanisation/ESOP-2020-Isabelle.zip`.

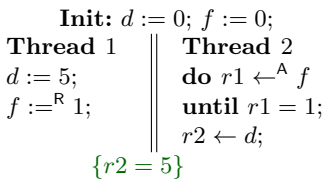

Fig. 1. Message-passing litmus test

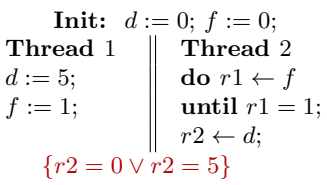

Fig. 2. Unsynchronised message passing

The two programs are almost identical and consist of two threads executing in parallel, accessing shared variables. The assertions in curly brackets at the end specify the programs' postconditions.

The programs comprise two shared variables: d (that stores some data) and f (that stores a flag). In both programs, both d and f are initially 0. Thread 1 updates d to 5, then updates f to 1. Thread 2 waits for f to be set to 1, then reads from d. Under sequential consistency, one would expect that the final value of $r2$ is 5, since the loop in Thread 2 only terminates after f has been updated to 1 in Thread 1, which in turn happens after d has been set to 5. However, the C11 semantics allows the behaviour in Fig. 2, where Thread 2 may read a stale value of d, and hence only the weaker postcondition $r2 = 0 \vee r2 = 5$ holds. To regain the expected behaviour, one must introduce additional synchronisation in the program. In particular, the write to f by Thread 1 must be a *releasing write* (i.e., $f :=^{\mathsf{R}} 1$) and the read of f in Thread 2 must be an *acquiring read* (i.e., $r_1 \leftarrow^{\mathsf{A}} f$) as in Fig. 1.

In sequential consistency all threads have a single common view of the shared state, namely all threads see the latest write that occurs for each variable. When a new write is executed, the views of all threads are updated so that they see this write. In contrast, each thread in C11 programs has its own view of each variable, which is affected by synchronisation annotations. Thus, for the program in Fig. 2, after initialisation, all threads see the initial writes (i.e., $d = 0, f = 0$). The assignments in Thread 1 only change Thread 1's view, and leave Thread 2's view unchanged. Thus, after execution of $f := 1$, Thread 2 has access to two values for d (i.e., $d \in \{0, 5\}$) and f (i.e., $f \in \{0, 1\}$). Even if Thread 2 reads $f = 1$, its view of d remains unchanged and it continues to have access to both values of d.

The program in Fig. 1 has a similar semantics for initialisation and execution of Thread 1, i.e., its execution does not affect the view of Thread 2. However, due to the release-acquire synchronisation on f (notation R and A), after Thread 2 reads $f = 1$, its view for d will be updated so that the stale value $d = 0$ is no longer available for it to read. One way to explain this behaviour is by thinking of Thread 1 as passing its knowledge of the write to d to Thread 2 via the variable f, which is synchronised using the release-acquire annotations.

This intuition is captured formally using a semantics based on *timestamps* [15,12,16,28], which enables one to encode each thread's view and define how these views are updated. In this paper, we characterise the release-acquire-

relaxed subset of C11 [11] (RC11) using timestamps, which has a restriction prohibiting the so-called *load-buffering* litmus test [19].

The main contribution of our paper is an assertion language that enables one to reason about thread views in a Hoare-style proof calculus, resulting in the proof outline given in Fig. 3. As already noted, the standard rules of Hoare and Owicki-Gries logic remain unchanged. For message passing, we require three main types of assertions (see Section 5):

Possible value. A possible value assertion (denoted $x \approx_t n$) states that thread t can read value n of global variable x, i.e., there is a write to x with value n beyond or including the *viewfront*[5] of thread t. Note that there may be more than one such write, and hence there may be several possible values for a given variable. Moreover, the last write to each variable is always viewable as a possible value.

Definite value. A definite value assertion (denoted $x =_t n$) states that thread t's viewfront is up-to-date with the writes to x (i.e., there is a single write to x beyond or including the viewfront of thread t), and this write updates x's value to n. Thus, t definitely knows the variable x to have value n.

Conditional value. A conditional value assertion (denoted $[x = n](y =_t m)$) captures the message passing idiom for variable y via variable x. It guarantees that when thread t reads x to be n via an acquiring read, a release-acquire synchronisation is induced and thereby t learns the definite value of y to be m. In particular, after reading $x = n$ via an acquiring read, the viewfront for t is updated so that the only write to y beyond or including this viewfront is a write with value m.

For the example in Fig. 3, after initialisation, both threads 1 and 2 have definite value 0 for both d and f. The precondition of $d := 5$ states that Thread 2 cannot possibly observe 1 for f (i.e., $f \approx_2 1$) and Thread 1 definitely observes 0 for d (i.e., $d =_1 0$). These assertions can be proven locally correct and interference free since Thread 2 neither modifies d nor f. The precondition of $f :=^{\mathsf{R}} 1$ is similar but with $d =_1 5$ in place of $d =_1 0$. The precondition of the **until** loop in Thread 2 contains a conditional value assertion, which ensures that if Thread 2 reads $f = 1$ then it will definitely read $d = 5$. This conditional value assertion enables one to establish local correctness of the precondition (i.e., $d =_2 5$) of the statement $r2 \leftarrow d$, which leads to the postcondition of the program. Each of the assertions in Thread 2 can be proven to be interference free against Thread 1.

3 Program Syntax

We start by defining the syntax of concurrent programs, starting with the structure of sequential programs (single threads). A thread may use *global* shared variables (from Var_G) and local registers (from Var_L). We let $Var = Var_G \cup Var_L$ and assume $Var_G \cap Var_L = \emptyset$. Global variables can be accessed in three different

[5] We borrow the term viewfront from Popkadaev et al [28].

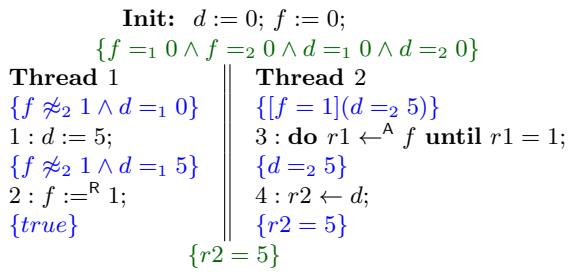

Fig. 3. Proof outline for message passing

synchronisation modes: acquire (A, for reads), release (R, for writes) and relaxed (no annotation). The annotation RA is employed for *update* operations, which read and write to a shared variable in single atomic step. We use x, y, z to range over global variables and $r1, r2, \ldots$ to range over local variables. We assume that $\ominus$ is a unary operator (e.g., $\neg$), $\oplus$ is a binary operator (e.g., $\wedge$, $+$, $=$) and n is a value (of type *Val*). Expressions may only involve local variables. For a treatment of expressions with global variables in the semantics see [11]. The syntax of sequential programs, *Com*, is given by the following grammar (with $r \in Var_L, x \in Var_G$):

$$
\begin{aligned}
Exp_L &::= Val \mid r \mid \ominus Exp_L \mid Exp_L \oplus Exp_L \\
ACom &::= \mathbf{skip} \mid x.\mathbf{swap}(n)^{\mathsf{RA}} \mid r := Exp_L \mid x :=^{[\mathsf{R}]} Exp_L \mid r \leftarrow^{[\mathsf{A}]} x \\
Com &::= ACom \mid Com; Com \mid \mathbf{if}\ B\ \mathbf{then}\ Com\ \mathbf{else}\ Com \mid \mathbf{while}\ B\ \mathbf{do}\ Com
\end{aligned}
$$

where we assume B to be an expression of type Exp_L that evaluates to a boolean. The statement $x.\mathbf{swap}(n)^{\mathsf{RA}}$ atomically reads the variable x (using an acquiring read) and updates x to value n (using a releasing write) in a single atomic step. Its execution therefore gives rise to an atomic read-modify-write update event. The notation $[\mathsf{X}]$ denotes that the annotation X is optional, where $\mathsf{X} \in \{\mathsf{A}, \mathsf{R}\}$, enabling one to distinguish relaxed, acquiring and releasing accesses. Loops will be used in other forms, like **do-until** or **do-while**, which are straightforward to define in terms of the command syntax above.

As is standard in Owicki-Gries proofs, we make use of *auxiliary variables*, which are variables that do not affect the meaning of a program, but appear in proof assertions. We require that each auxiliary variable is local to the thread in which it occurs. Moreover, the only assignments we allow to an auxiliary variable a are of the form $a := E$, where $E \in Exp_L$. Finally, we require that writes to auxiliary variables occur atomically in conjunction with another atomic program step. Such atomic operations are written as $\langle A, a := E \rangle$, where $A \in ACom$.

For simplicity, we assume concurrency at the top level only. We let *Tid* be the set of all thread identifiers and use a function $Prog : Tid \to Com$ to model a program comprising multiple threads. In examples, we typically write concurrent programs as $C_1 || \ldots || C_n$, where $C_i \in Com$. We further assume some initialisation of global variables. The structure of our programs thus is $\mathbf{Init}; \big(C_1 || \ldots || C_n\big)$.

4 Semantics

The operational semantics for this language is defined in two parts. The *program semantics* fixes the steps that the concurrent program can take. This gives rise to transitions $(P, lst) \xrightarrow{a}_t (P', lst')$ of a thread t where P and P' are programs, lst and lst' is the state of local variables and a is an action (possibly the silent action τ, see below). The program semantics is combined with a *memory semantics* which reflects the C11 state (denoted by σ), and in particular the write actions from which a read action can read.

We start by fixing the actions, where $x \in Var_G$ and $m, n \in Val$:

$$\mathsf{Act} = \{rd(x,n), rd^{\mathsf{A}}(x,n), wr(x,n), wr^{\mathsf{R}}(x,n), upd^{\mathsf{RA}}(x,n,m)\}$$

containing actions for (releasing) reads, (acquiring) writes and updates (reading value n and writing m). We furthermore employ a silent τ action and let $\mathsf{Act}_\tau = \mathsf{Act} \cup \{\tau\}$. For an action $a \in \mathsf{Act}$, we let $var(a) \in Var_G$ be the variable read (or written to), $rdval(a) \in Val$ be the value read and $wrval(a) \in Val$ be the value written. We let U denote the update actions, and distinguish the sets $\mathsf{W_R} \supseteq \mathsf{U}$ (write release), $\mathsf{R_A} \supseteq \mathsf{U}$ (read acquire), $\mathsf{W_X}$ (write relaxed) and $\mathsf{R_X}$ (read relaxed). Finally, we define $\mathsf{R} = \mathsf{R_A} \cup \mathsf{R_X}$ (all reads) and $\mathsf{W} = \mathsf{W_R} \cup \mathsf{W_X}$ (all writes). Typically, we refer to the elements of W as *writes*, but note that this set also includes update actions.

4.1 Program Semantics

In the program semantics, we assume a function $lst \in Tid \to (Var_L \nrightarrow Val)$, which returns the local state for the given thread. We assume that the local variables of threads are disjoint, i.e., if $t \neq t'$, then $\mathbf{dom}(lst(t)) \cap \mathbf{dom}(lst(t')) = \emptyset$. For an expression E over local variables, we write $[\![E]\!]_{ls}$ for the value of E in local state ls; we write $ls[r := n]$ to state that ls remains unchanged except for the value of local variable r which becomes n.

Figure 4 gives the transition rules of the program semantics. The last rule, Prog, lifts the transitions of threads to a transition for a concurrent program. The other rules concern the sequential part of the language. The rules in a sense ignore the fact that the language allows for global variables; the program semantics just details the values of local variables in component ls. When global variables are read, the program semantics allows for *all* possible values to be read. This is combined with the memory semantics (formalised by $\overset{a}{\rightsquigarrow}_t$) as follows:

$$\frac{(P, lst) \xrightarrow{\tau}_t (P', lst')}{(P, lst, \sigma) \Longrightarrow_t (P', lst', \sigma)} \qquad \frac{(P, lst) \xrightarrow{a}_t (P', lst') \quad \sigma \overset{a}{\rightsquigarrow}_t \sigma'}{(P, lst, \sigma) \Longrightarrow_t (P', lst', \sigma')}$$

The transitions defined by $\sigma \overset{a}{\rightsquigarrow}_t \sigma'$ ensure that read actions only return a value allowed by the C11 semantics and are defined in Section 4.2. The rules for all imperative program constructs (sequential composition, **if** and **while**) are standard.

$$\frac{r \in Var_L \quad n = [\![E]\!]_{ls}}{(r := E, ls) \xrightarrow{\tau} (\mathbf{skip}, ls[r := n])} \qquad \frac{x \in Var_G \quad a = wr^{[\mathsf{R}]}(x, [\![E]\!]_{ls})}{(x :=^{[\mathsf{R}]} E, ls) \xrightarrow{a} (\mathbf{skip}, ls)}$$

$$\frac{a = rd^{[\mathsf{A}]}(x, n) \quad n \in Val}{(r \leftarrow^{[\mathsf{A}]} x, ls) \xrightarrow{a} (\mathbf{skip}, ls[r := n])} \qquad \frac{a = upd^{\mathsf{RA}}(x, m, n) \quad m \in Val}{(x.\mathbf{swap}(n)^{\mathsf{RA}}, ls) \xrightarrow{a} (\mathbf{skip}, ls)}$$

$$\frac{(C_1, ls) \xrightarrow{a} (C_1', ls')}{(C_1; C_2, ls) \xrightarrow{a} (C_1'; C_2, ls')} \qquad \frac{}{(\mathbf{skip}; C_2, ls) \xrightarrow{\tau} (C_2, ls)}$$

$$\frac{[\![B]\!]_{ls}}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, ls) \xrightarrow{\tau} (C_1, ls)} \qquad \frac{\neg[\![B]\!]_{ls}}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, ls) \xrightarrow{\tau} (C_2, ls)}$$

$$\frac{[\![B]\!]_{ls}}{(\mathbf{while}\ B\ \mathbf{do}\ C, ls) \xrightarrow{\tau} (C; \mathbf{while}\ B\ \mathbf{do}\ C, ls)} \qquad \frac{\neg[\![B]\!]_{ls}}{(\mathbf{while}\ B\ \mathbf{do}\ C, ls) \xrightarrow{\tau} (\mathbf{skip}, ls)}$$

$$\textsc{Aux}\ \frac{(A, ls) \xrightarrow{a} (\mathbf{skip}, ls') \quad (a := E, ls') \xrightarrow{\tau} (\mathbf{skip}, ls'')}{(\langle A; a := E\rangle, ls) \xrightarrow{a} (\mathbf{skip}, ls'')} \qquad \textsc{Prog}\ \frac{(P(t), lst(t)) \xrightarrow{a} (C, ls) \quad a \in \mathsf{Act}_\tau}{(P, lst) \xrightarrow{a}_t (P[t := C], lst[t := ls])}$$

Fig. 4. Program semantics

4.2 Memory Semantics

Next, we detail the memory semantics, which is equivalent to an earlier operational reformulation [11] of the RC11 semantics [19].

C11 State. Table 1 summarises the components of a C11 state. Each global write is represented by a pair $(a, q) \in \mathsf{W} \times \mathbb{Q}$, where a is a write action, and q is a rational number that we use as a *timestamp* (c.f., [15,12,28]. The timestamps totally order the writes to each variable, and are also referred to as the *modification order* [19,11] or *coherence order* [3]. For each write $w = (a, q)$, we denote w's timestamp by $tst(w) = q$. We also lift the functions *var* and *wrval* to timestamped writes, e.g., $var((a, q)) = var(a)$. The set of all writes that have occurred in the execution thus far is recorded in the state component $writes \subseteq \mathsf{W} \times \mathbb{Q}$.

Table 1. Components of a C11 state

Component	Informal meaning	Initial value
$writes \subseteq \mathsf{W} \times \mathbb{Q}$	The writes which have happened so far	$writes_{\mathbf{Init}}$
$tview_t \in Var_G \to writes$	The view of a thread t	$tview_{\mathbf{Init}}$
$mview_w \in Var_G \to writes$	The view of a thread when writing w	$mview_{\mathbf{Init}}$
$covered \subseteq writes$	The covered writes	$\emptyset$

As described in Section 2, each state must record the writes that are observable to each read. To achieve this, we use two families of functions from global variables to writes, both of which record the *viewfronts* (c.f., [28,16]).

- A function $tview_t$ that returns the *viewfront* of thread t. The thread t can read from any write to variable x whose timestamp is not earlier than $tview_t(x)$. Accordingly, we define, for each state σ, thread t and global variable x, the set of *observable writes*:

$$\sigma.OW(t,x) = \{(a,q) \in \sigma.writes \mid var(a) = x \wedge tst(\sigma.tview_t(x)) \leq q\} \quad (1)$$

- A function $mview_w$ that records the *viewfront* of write w, which is set to be the viewfront of the thread that executed w at the time of w's execution. We use $mview_w$ to compute a new value for $tview_t$ if a thread t *synchronizes* with w, i.e., if $w \in \mathsf{W_R}$ and another thread executes an $e \in \mathsf{R_A}$ that reads from w.

Finally, our semantics maintains a variable $covered \subseteq writes$. In RC11, each update action occurs in modification order immediately after the write that it reads from [11]. This property constitutes the atomicity of updates. In order to preserve this property, we must prevent any newer write from intervening between any update and the write that it reads from. As we explain below, *covered* writes are those that are immediately prior to an update in modification order, and new write actions never interact with a covered write.

Initialisation. Table 1 also states how these components are initialised by **Init**. If $Var_G = \{x_1, \ldots, x_n\}$, $Var_L = \{r_1, \ldots, r_m\}$ and $k_1, \ldots, k_n, l_1, \ldots, l_m \in Val$, we assume $\mathbf{Init} = x_1 := k_1; \ldots, x_n := k_n; [r_1 := l_1;] \ldots [r_m := l_m;]$, where we use the notation $[r_i := l_i;]$ to mean that the assignment $r_i := l_i$ may optionally appear in **Init**. Thus each shared variable is initialised exactly once and each local variable is initialised at most once. The initial values of the state components are then as follows, where we assume that 0 is the initial timestamp.

$$\begin{aligned} writes_{\mathbf{Init}} &= \{(wr(x_1,k_1),0), \ldots, (wr(x_n,k_n),0)\} \\ tview_{\mathbf{Init}}(x_i) &= (wr(x_i,k_i),0) \quad \text{for each thread } x_i \in Var_G \\ mview_{\mathbf{Init}} &= tview_{\mathbf{Init}} \end{aligned}$$

Moreover, the local state of each thread must be compatible with **Init**.

Transition semantics. The transition relation of our semantics for global reads and writes is given in Fig. 5. Each transition $\sigma \overset{a}{\rightsquigarrow}_t \sigma'$ is labelled by an action a and thread t. The premise of each rule must identify the write w that the action interacts with. This is made more precise below.

READ **transition by thread** t**.** Here we assume that

- a is either a relaxed or acquiring read to variable x,
- w is a write to x that t can observe (i.e., $(w,q) \in \sigma.OW(t,x)$), and
- the value read by a is the value written by w.

$$\textsc{Read}\ \frac{a \in \{rd(x,n), rd^{\mathsf{A}}(x,n)\} \quad (w,q) \in \sigma.OW(t,x) \quad wrval(w) = n \qquad tview'_t = \begin{cases} \sigma.tview_t \otimes \sigma.mview_{(w,q)} & \text{if } (w,a) \in \mathsf{W_R} \times \mathsf{R_A} \\ \sigma.tview_t[x := (w,q)] & \text{otherwise} \end{cases}}{\sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t]}$$

$$\textsc{Write}\ \frac{a \in \{wr(x,n), wr^{\mathsf{R}}(x,n)\} \quad (w,q) \in \sigma.OW(t,x) \setminus \sigma.covered \quad fresh(q,q') \qquad writes' = \sigma.writes \cup \{(a,q')\} \qquad tview'_t = \sigma.tview_t[x := (a,q')]}{\sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, writes := writes']}$$

$$\textsc{Update}\ \frac{\begin{array}{c} a = upd^{\mathsf{RA}}(x,m,n) \quad (w,q) \in \sigma.OW(t,x) \setminus \sigma.covered \\ wrval(w) = m \quad fresh(q,q') \\ writes' = \sigma.writes \cup \{(a,q')\} \quad covered' = \sigma.covered \cup \{(w,q)\} \\ tview'_t = \begin{cases} \sigma.tview_t[x := (a,q')] \otimes \sigma.mview_{(w,q)} & \text{if } w \in \mathsf{W_R} \\ \sigma.tview_t[x := (a,q')] & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} \sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, \\ \qquad writes := writes', covered := covered'] \end{array}}$$

Fig. 5. Transition relation of the memory semantics, where
$fresh(q,q') = q < q' \wedge \forall w' \in \sigma.writes.\ q < tst(w') \Rightarrow q' < tst(w')$

Each read causes the viewfront of t to be updated. This is computed as follows. If the read synchronises with the write, then the thread's new view will be a combination of its existing view, and the view of that write. In particular, for each variable x the new view of x will be the later of either $tview_t(x)$ or $mview_w(x)$, in timestamp order. To express this, we use an operation that combines two views v_1 and v_2, by constructing a new view that takes the later of the writes at each variable:

$$(v_1 \otimes v_2)(x) = \begin{cases} v_1(x) & \text{if } tst(v_2(x)) \leq tst(v_1(x)) \\ v_2(x) & \text{otherwise} \end{cases}$$

If w and a do not synchronise, then $tview_t$ is simply updated to include the new write.

For illustration, consider the picture in Fig. 6. The x-axis depicts the timestamps of the writes, the y-axis the variables x, y and z, which we assume are initialised by writes x_0, y_0 and z_0, respectively. The orange line shows the view of a thread, say t_1, and the blue line depicts the view of another thread that executes $w = (wr^{\mathsf{R}}(y,42),3)$. If thread t_1 performs an acquiring read of y and reads from w (i.e., it performs a sychronising read), thread t_1's view changes to the diagram on the right, whereby its current viewfront is combined with the viewfront of w.

Write **transition by thread** t**.** A write transition must identify the write (w,q) after which a occurs. This w must be observable and must *not* be covered — the second condition is required to preserve the read-modify-write atomicity

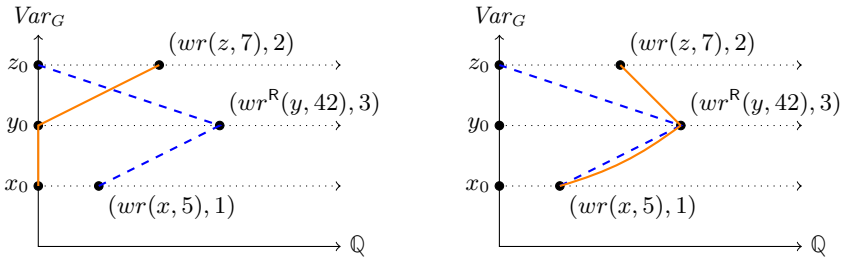

Fig. 6. Illustration of views and view updates: pre-state (left) and post-state (right)

of updates. We must choose a fresh timestamp $q' \in \mathbb{Q}$ for a, which is formalised by $\mathit{fresh}(q, q')$. The predicate $\mathit{fresh}(q, q')$ ensures that q' is a new timestamp for the variable x, such that (a, q') occurs immediately after (w, q). The new write is added to the set *writes*. We update tview_t to include the new write, which means t can no longer observe any writes prior to (a, q'). Finally, we set the viewfront of (a, q') to be the new viewfront of t, i.e., $\mathit{mview}_{(a,q')} := \mathit{tview}'_t$. Now, if some other thread synchronises with this new write in some later transition, that thread's view will become at least as recent as t's view at this transition.

UPDATE **transition by thread** t**.** These transitions are best understood as a combination of the read and write transitions. As with a write transition, we must choose a valid fresh q', and the state components *writes* and *mview* are updated in the same way. As discussed earlier, in UPDATE transitions it is necessary to record that the write that the update interacts with is now covered, which is achieved by adding that write to *covered*. Finally, we must compute a new thread view, which is similar to a READ transition, except that the thread's new view always includes the new write introduced by the update.

4.3 Relationship to the Axiomatic Semantics

We prove that the timestamp-based semantics presented here is equivalent to an earlier operational semantics [11] that is already known to be equivalent to the RC11 fragment. Here, we just roughly sketch how this proof proceeds.

The semantics in [11] describes C11 states in the form $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$, where X is a set of read and write events (roughly equivalent to actions) and sb, rf and mo describe the sequenced-before and reads-from relation as well as the modification order of the C11 axiomatic semantics. A number of further relations are derived from these, in particular the extended coherence order eco and the happens-before order hb. The proof of equivalence of the semantics shows the two semantics to *simulate* each other. For this, we need to define a correspondence between C11 states of form E and of form σ such that: (1) For $\sigma.\mathit{writes}$, we take $X \cap \mathsf{W}$; (2) For $\sigma.\mathit{covered}$, we take the writes w in $X \cap \mathsf{W}$ such that there is an update u with $(w, u) \in \mathsf{rf}$; and (3) For *mview* and *tview*, we use a downward closure operator, **cclose**, which for a given set of events S determines the set of events prior to S in the relation $\mathsf{eco}^? \circ \mathsf{hb}^?$. Then $\sigma.\mathit{tview}_t =$

$\mathbf{max}_{\mathsf{mo}}(X.\mathbf{cclose}(X_t))$ and $\sigma.mview_w = \mathbf{max}_{\mathsf{mo}}(X.\mathbf{cclose}(\{w\}))$, where $\mathbf{max}_{\mathsf{mo}}$ selects writes being maximal wrt. mo and X_t are all actions of t in X. In all these cases, timestamps for writes have to be selected consistent with mo.

Given such a correspondence, the proof proceeds by showing this correspondence is preserved by the read, write and update transitions.

4.4 Well Formedness

Our proofs in subsequent sections require that the state under consideration is *well-formed*. This is formalised by predicate *wfs* over a C11 state σ, where

$$\begin{aligned} wfs(\sigma) \iff\ & ran((\textstyle\bigcup_t \sigma.tview_t) \cup (\bigcup_w \sigma.mview_w)) \subseteq \sigma.writes \wedge \\ & finite(\sigma.writes) \wedge \sigma.covered \subseteq \sigma.writes \wedge \\ & (\forall w.\ w \in \sigma.writes \Rightarrow \sigma.mview_w(var(w)) = w) \end{aligned}$$

The first conjunct ensures that each viewable write is in $\sigma.writes$. The second conjunct ensures there are only a finite number of writes, and the third ensures that every covered write is an actual write. The final conjunct ensures that for each write in $\sigma.writes$, the viewfront of w for $var(w)$ is w itself.

Well-formedness is invariant for any program, i.e., every initialisation establishes well-formedness and every program transition preserves well-formedness.

Lemma 1. *For any program C constructed using the syntax described in Section 3, $wfs(\sigma)$ is invariant.*

Proof. In Isabelle. We show that every initialisation establishes $wfs(\sigma)$. Furthermore, if $wfs(\sigma)$ and $\sigma \overset{a}{\rightsquigarrow}_t \sigma'$, then $wfs(\sigma')$ for any action a and thread t. □

5 Hoare Logic and Owicki-Gries Reasoning for C11

In this section, we present a Hoare logic [14] for RC11 that enables Owicki-Gries reasoning [26]. For compound statements (including concurrent composition) we use the standard rules of Hoare logic as well as the standard interference freedom proof obligations described by Owicki and Gries. Our contribution is a novel set of high-level predicates that describe the *observations* of each thread for a C11 state, together with a set of *basic axioms* that describe how these predicates interact with read, write and update transitions. Soundness of these axioms has been checked using Isabelle.

We present the basic axioms in stages, using specific examples to motivate each group of assertions. In Section 5.1, we link our operational semantics to the proof outlines of Hoare logic and Owicki-Gries' notion of interference freedom. In Section 5.2, we verify the load buffering litmus test and in Section 5.3, we verify message passing. Section 5.4 presents proofs of three different versions of the read-read coherence litmus test, which demonstrates the use of assertions describing value orders. Our final example Section 5.5 is Peterson's algorithm adapted for C11. The proof outlines of all of these examples have been verified using Isabelle.

$$\textsc{Skip}\ \frac{}{\{p\}\mathbf{skip}\{p\}} \qquad \textsc{Seq}\ \frac{\{p\}C_1\{r\} \quad \{r\}C_2\{q\}}{\{p\}C_1; C_2\{q\}}$$

$$\textsc{If}\ \frac{\{p \wedge B\}C_1\{q\} \quad \{p \wedge \neg B\}C_2\{q\}}{\{p\}\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\{q\}} \qquad \textsc{While}\ \frac{\{p \wedge B\}C\{p\}}{\{p\}\mathbf{while}\ B\ \mathbf{do}\ C\{p \wedge \neg B\}}$$

$$\textsc{Until}\ \frac{\{p\}C\{r\} \quad \{r\}\mathbf{while}\ \neg B\ \mathbf{do}\ C\{r \wedge B\}}{\{p\}\mathbf{do}\ C\ \mathbf{until}\ B\{r \wedge B\}} \qquad \textsc{Cons}\ \frac{p \Rightarrow p' \quad \{p'\}C\{q'\} \quad q' \Rightarrow q}{\{p\}C\{q\}}$$

Fig. 7. Classical proof rules for sequential programs

5.1 Soundness and Classical Verification Rules

We first define the meaning of a Hoare triple under partial correctness and present the classical proofs rules for compound statements. Unlike Hoare logic, where a state is modelled by a mapping from variables to values, as we have seen in Section 4.1, states of a C11 program contain two components: a local state *lst* and a global state σ. We let Σ_G be the set of all possible global state configurations (as described in Table 1) and let $\Sigma_{C11} = (Var_L \rightarrow Val) \times \Sigma_G$ be the set of all possible C11 states. Predicates over Σ_{C11} are therefore of type $\Sigma_{C11} \rightarrow \mathbb{B}$. This leads to the following definition of a Hoare triple, which we note is the same as the standard definition — the only difference is that the state component is of type Σ_{C11}.

Definition 2. *Suppose $p, q \in \Sigma_{C11} \rightarrow \mathbb{B}$, $C \in Com$ and t is a thread. The semantics of a Hoare triple under partial correctness is given by:*

$$\{p\}C\{q\} = \forall lst, \sigma, lst', \sigma'.\ p(lst, \sigma) \wedge (C, lst, \sigma) \Longrightarrow_t^* (\mathbf{skip}, lst', \sigma') \Rightarrow q(lst', \sigma')$$

The classical rules of sequential Hoare logic for compound (i.e., non-atomic) statements are given in Fig. 7. Soundness of these proof rules (with respect to Definition 2) holds for exactly the same reason as soundness of Hoare logic [14].

The sequential part is combined with the Owicki-Gries rule for concurrent composition in the standard way [26,7]. First, we construct *proof outlines* for every component of the concurrent program in isolation. A proof outline inserts assertions (in {, } brackets) into a program. In a so-called *standard* proof outline every statement R of the program has exactly one assertion before it. This assertion is its *precondition*, $pre(R)$. Next, all assertions in one component have to be checked for non-interference with all statements in other components.

Definition 3. *A statement $R \in ACom$ with precondition $pre(R)$ (in the standard proof outline) does* not interfere *with an assertion p if*

$$\{p \wedge pre(R)\}\ R\ \{p\}\ .$$

Interference freedom guarantees that proof outlines in each thread is stable under the execution of other threads. This is formalised in the Owicki-Gries proof rule

Init: $x := 0$; $y := 0$; $r1 := 0$; $r2 := 0$;
$\{x =_1 0 \wedge y =_2 0 \wedge r1 = 0 \wedge r2 = 0\}$

Thread 1	**Thread** 2
$\{y =_2 0 \wedge r2 = 0\}$	$\{x =_1 0 \wedge r1 = 0\}$
$1 : r1 \leftarrow x$;	$3 : r2 \leftarrow y$;
$\{y =_2 0 \wedge r2 = 0\}$	$\{x =_1 0 \wedge r1 = 0\}$
$2 : y := 1$;	$4 : x := 1$;
$\{r1 = 0 \vee r2 = 0\}$	$\{r1 = 0 \vee r2 = 0\}$

$\{r1 = 0 \vee r2 = 0\}$

Fig. 8. Proof outline for load buffering

for concurrent composition:

$$\frac{\text{Proof outlines } \{p_i\}C_i\{q_i\} \text{ are interference free}}{\{\bigwedge_{i=1}^{n} p_i\}\ C_1||\ldots||C_n\ \{\bigwedge_{i=1}^{n} q_i\}}$$

We say a proof outline is *valid* if it is both sequentially valid (or locally correct) and interference free.

5.2 Load Buffering

Our first example is the load buffering litmus test (see Fig. 8), which we can show satisfies the postcondition $r1 = 0 \vee r2 = 0$ since our semantics assumes absence of "thin air" reads [19,11]. The assertions about the C11 state capture properties about *definite observations* (see Section 2), which we formalise below.

For a set of writes W and variable $x \in Var_G$, let $W_x = \{w \in W \mid var(w) = x\}$ be the set of writes in W that write to x. We define the *last write* to x in W as:

$$last(W, x) = w \iff w \in W_x \wedge (\forall w' \in W_x.\ tst(w') \leq tst(w))$$

Moreover, we define the definite observation of a view function, *view* with respect to a set of writes as follows:

$$dview(view, W, x) = n \iff view(x) = last(W, x) \wedge wrval(last(W, x)) = n$$

The first conjunct ensures that the viewfront of *view* for x is the last write to x in W, and the second conjunct ensures that the value written by the last write to x in W is n.

Definite observation. A definite observation (denoted $x =_t n$) states that thread t *must* read the value n if it reads from the variable x. Formally:

$$x =_t n \quad = \quad \lambda\sigma.\ dview(\sigma.tview_t, \sigma.writes, x) = n$$

Expanding this out, we obtain:

$$\sigma.tview_t(x) = last(\sigma.writes, x) \wedge wrval(last(\sigma.writes, x)) = n$$

The first conjunct ensures that the viewfront of t for x is the last write to x in σ (thus t can only read this last write to x). The second conjunct ensures that the value written by the last write is n. The function *dview* is also used in the definition of conditional observation in Section 5.3.

The proof of load buffering relies on the basic axioms in the following lemma. We assume $atoms(\mathbf{Init})$ returns the set of assignments contained within **Init**.

Lemma 4. *Each of the basic axioms below is sound (as per Definition 2), where the statements are decorated with the thread identifier of the executing thread.*

$$\textsc{Init}\ \frac{x := n \in atoms(\mathbf{Init})}{\{true\}\ \mathbf{Init}\ \{x =_t n\}} \qquad \textsc{DOPres-Rd}\ \frac{}{\{x =_{t'} m\}\ r \leftarrow_t^{[\mathsf{A}]} y\ \{x =_{t'} m\}}$$

$$\textsc{DOPres-Wr}\ \frac{x \neq y}{\{x =_{t'} n\}\ y :=_t m\ \{x =_{t'} n\}}$$

Proof. In Isabelle. □

Thus by rule Init an assignment $x := n$ in Init ensures that $x =_t n$ for all threads t holds at program start. Note that such an initial assertion for the entire program is not subject to non-interference checks. The rule DOPres-Rd states that a definite observation $x =_{t'} m$ is invariant over a read step executed by thread t. Note that pre/post conditions for DOPres-Rd refer to thread t', while the read statement refers to thread t. Also note that there is no additional restriction on t and t', thus the rule applies regardless of whether $t = t'$, or not. Similarly, there are two global variables x and y mentioned in the rule, but there are no further restrictions on their values. Rule DOPres-Wr gives a condition for invariance of a definite observation assertion over a write. It requires that the variable being observed is different from the variable that is updated.

Theorem 5. *The proof outline for load buffering in Fig. 8 is valid.*

Proof. The proof has been established in Isabelle. We outline the main steps below as it is instructive to understand the high-level proof strategy. First we establish local correctness:

- The initial condition is established by rule Init, which is in turn used to establish the initial assertions in both threads.
- In thread 1, local correctness of the postcondition of line 1 (precondition of line 2) follows from rule DOPres-Rd, and the postcondition of line 2 follows by weakening. The proof of local correctness in thread 2 is symmetric.

We now establish interference freedom. The precondition of line 1 is interference free wrt line 3 by DOPres-Rd, and wrt line 4 by DOPres-Wr. This argument also applies to the precondition of line 2. Interference freedom of the postcondition of line 2 is trivial. The proof of interference freedom of the assertions in thread 2 is symmetric. □

5.3 Message Passing

Next we return to the message passing example from Section 2. Its verification requires the introduction of two new types of assertions.

Possible observation. A possible observation (denoted $x \approx_t n$) states that thread t *may* read the value n if it reads from the variable x. Formally:

$$x \approx_t n \quad = \quad \lambda\sigma.\ \exists w \in \sigma.OW(t,x).\ wrval(w) = n$$

Conditional observation. A conditional observation (denoted $[x = n](y =_t m)$) states that if thread t reads a value n for x, it synchronises with the corresponding write and obtains the definite value m for y. Formally:

$$[x = n](y =_t m) \quad = \quad \lambda\sigma.\ \forall w \in \sigma.OW(t,x).\ wrval(w) = n \Rightarrow act(w) \in \mathsf{W_R} \wedge dview(\sigma.mview_w, \sigma.writes, y) = m$$

The antecedent assumes that the value read for x is n, and the consequent ensures that w is a releasing write such that the definite view of this write for variable y returns m. As we shall see, one useful way of establishing this condition is by falsifying the antecedent by ensuring that thread t cannot observe n for x (see (4) below).

Some useful relationships between the assertions above are given by the lemma below.

Lemma 6. *For variables $x, y \in Var_G$, thread t and values $m, n \in Val$, each of the following holds:*

$$wfs \wedge x =_t n \Rightarrow x \approx_t n \tag{2}$$

$$wfs \wedge x =_t n \wedge x \approx_t m \Rightarrow n = m \tag{3}$$

$$x \not\approx_t n \Rightarrow [x = n](y =_t m) \tag{4}$$

$$x =_t n \wedge x =_{t'} m \Rightarrow n = m \tag{5}$$

Proof. In Isabelle. □

By (2), given a well-formed state any definite observation implies a possible observation, and by (3) a definite observation must agree with a possible observation. By (4) if it is not possible to observe the antecedent of a conditional observation, then the conditional observation must hold. By (5) any two definite value observations must agree (since they both observe the last write to x).

The next lemma lists the basic axioms that are used to prove correctness of the message passing example.

Lemma 7. *Each of the rules next is sound (as per Definition 2), where the statements are decorated with the thread identifier of the executing thread.*

$$\textsc{ModLast}\ \frac{}{\{x =_t n\}\ x :=_t m\ \{x =_t m\}} \qquad \textsc{ModSome}\ \frac{}{\{true\}\ x :=_t m\ \{x \approx_t m\}}$$

$$\textsc{NPOPres}\ \frac{}{\{x \not\approx_t m\}\ r \leftarrow^{[\mathsf{A}]}_{t'} y\ \{x \not\approx_t m\}} \qquad \textsc{NoOW}\ \frac{x \neq y}{\{x \not\approx_t n\}\ y :=_{t'} m\ \{x \not\approx_t n\}}$$

$$\textsc{ReadLast}\ \frac{}{\{x =_t m\}\ r \leftarrow_t x\ \{r = m\}}$$

$$\textsc{CO-Intro}\ \frac{x \neq y}{\{y =_t m \wedge x \not\approx_{t'} n\}\ x :=^{\mathsf{R}}_t n\ \{[x = n](y =_{t'} m)\}}$$

$$\textsc{Transfer}\ \frac{}{\{[x = n](y =_t m)\}\ r \leftarrow^{\mathsf{A}}_t x\ \{r = n \Rightarrow y =_t m\}}$$

Proof. In Isabelle. □

Theorem 8. *The proof outline of message passing in Fig. 3 is valid.*

Proof. The proof has been established in Isabelle. We outline the main steps below. First we show local correctness.

- Using Init we establish the precondition $f =_1 0 \wedge f =_2 0 \wedge d =_1 0 \wedge d =_2 0$.
- The precondition of the program implies the initial assertions of both threads. In thread 1, we use (3) to establish $f \not\approx_2 1$ since (3) is logically equivalent to

$$wfs \wedge x =_t n \wedge n \neq m \Rightarrow x \not\approx_t m$$

  In thread 2, we use (3) in combination with (4).
- In thread 1, the post condition of line 1 (precondition of line 2) follows by application of NoOW and ModLast. The post condition of line 2 is trivial.
- In thread 2, the postcondition of line 3 follows by application of Transfer, while the postcondition of line 4 follows by application of ReadLast.

Next we show interference freedom.

- The preconditions of lines 1 and 2 can be shown to be interference free by applying NPOPres to the first conjunct and DOPres-Rd to the second.
- The precondition of line 3 is interference free against line 1 due to NoOW using the existing precondition $f \not\approx_2 1$ of line 1. The proof then follows by application of (4). Interference freedom against line 2, is proved using CO-Intro and the precondition at line 2.
- The precondition of line 4 is interference free against line 1 by (5) (i.e., since the preconditions are of lines 1 and 4 are contradictory). Interference freedom holds against line 2 by rule DOPres-Wr.
- The postconditions of lines 2 and 4 are trivially interference free. □

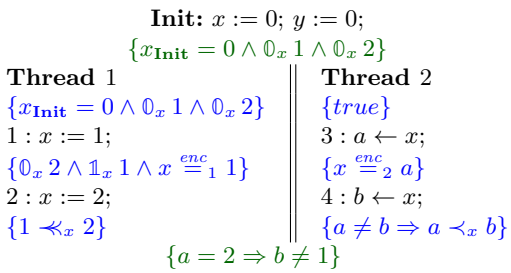

Fig. 9. Proof outline for RRC2, where $x \in Var_G$ and $a, b \in Var_L$

5.4 Read-Read Coherence

Next, we verify three versions of the read-read coherence (RRC) litmus test as given in Figs. 9, 10 and 11. The original RRC litmus test (Fig. 10) guarantees that if one thread sees the writes to x (by threads 1 and 2) in a certain order, then the other thread see the writes in the same order. Here, the postcondition assumes that thread 3 has observed the write $x := 1$, then the write $x := 2$, while thread 4 has already seen the write $x := 2$ when reading x at line 5. It requires that thread 4 does not subsequently see value 1 when it reads x at line 6. Fig. 9 presents a simpler variation where the ordering of writes to x is enforced by the thread ordering. Fig. 11 combines RRC with message passing.

Unlike message passing (which is a litmus test over two different variables), the RRC examples demonstrate the need for assertions that capture the *order* in which writes occur to a single variable. Therefore, we introduce the following new types of assertions:

Possible value order. A possible value order assertion (denoted $m \prec_x n$), as the name suggests, holds in a state whenever there are two writes to variable x with values m and n, such that the timestamp of the write with value m is less than the timestamp of the write with value n.

$$m \prec_x n \quad = \quad \lambda\sigma.\ \exists w, w' \in \sigma.writes_x.\ wrval(w) = m \wedge wrval(w') = n \wedge tst(w) < tst(w')$$

Note that this $m \prec_x n$ does not preclude $n \prec_x m$. E.g., if a thread writes m to x, then n, then m again, both $m \prec_x n$ and $n \prec_x m$ will hold. In this scenario, $m \prec_x m$ also holds since there are two separate writes to x with value m.

Definite value order. A definite value order assertion (denoted $m \prec\!\!\ll_x n$), holds in a state whenever $m \prec_x n$ holds, and any write to x with value m precedes any other write to value n.

$$m \prec\!\!\ll_x n \quad = \quad \lambda\sigma.\ (m \prec_x n)(\sigma) \wedge (\forall w, w' \in \sigma.writes_x.\ wrval(w) = m \wedge wrval(w') = n \Rightarrow tst(w) < tst(w'))$$

Unlike possible value orders if $m \prec\!\!\ll_x n$ holds then $n \not\prec\!\!\ll_x m$. Note that our definition allows several writes to x with values m and n provided all writes with value m occur (in timestamp order) before all writes with value n.

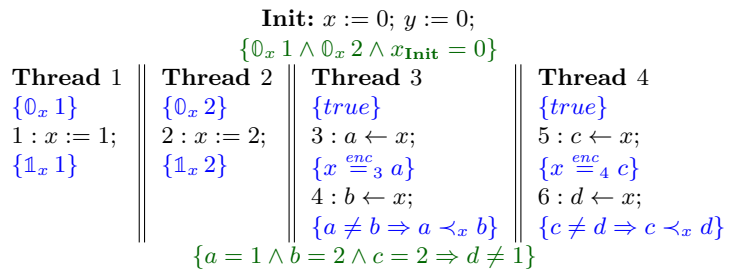

Fig. 10. Proof outline for RRC, where $x \in Var_G$ and $a, b, c, d \in Var_L$

Initial value. An initial value assertion (denoted $x_{\mathbf{Init}} = n$) holds if n is the first value recorded for x. It can be uniquely determined in each state as the value of the write with the least timestamp.

$$x_{\mathbf{Init}} = n \quad = \quad \lambda\sigma.\ \exists w \in \sigma.writes_x.\ wrval(w) = n \wedge (\forall w' \in \sigma.writes_x.\ w \neq w' \Rightarrow tst(w) < tst(w'))$$

Note that for the construction in this paper, it suffices to return the write to x with timestamp 0 since we assume that writes are initialised with timestamp 0. The definition above however, is more robust since it also applies to situations where variables are not initialised, or initialised to an arbitrarily chosen timestamp (as is the case in our Isabelle encoding).

Encountered value. An encountered value assertion (denoted $x \overset{enc}{=}_t n$) determines whether thread t has seen (or had the opportunity to see) the value n for the variable x. That is $x \overset{enc}{=}_t n$ holds iff there is a write to x with value n whose timestamp is at most the timestamp of the viewfront of t for x. Formally:

$$x \overset{enc}{=}_t n \quad = \quad \lambda\sigma.\ \exists w \in \sigma.writes_x.\ tst(w) \leq tst(\sigma.tview_t(x)) \wedge wrval(x) = n$$

Note that $x \overset{enc}{=}_t n$ does not guarantee that t has read the value n for x. For instance, $x \overset{enc}{=}_t n$ could hold if there is a write, say w, of x with value n and t writes to x with a write whose timestamp is greater than $tst(w)$.

Value limits. Value limit assertions (e.g., $\mathbb{0}_x\, n$, $\mathbb{1}_x\, n$) are used to determine how many writes to x with value n are present in a given C11 state. Assertion $\mathbb{0}_x\, n$ states that there are no writes to x with value n, while $\mathbb{1}_x\, n$ states that there is at most one write to x with value n. These are straightforward to define in terms of our value order assertions above. Formally:

$$\begin{aligned} \mathbb{0}_x\, n &= \exists m.\ x_{\mathbf{Init}} = m \wedge m \neq n \wedge m \not\prec_x n \\ \mathbb{1}_x\, n &= n \not\prec_x n \end{aligned}$$

Thus, if $\mathbb{0}_x\, n$ holds then there is no write with value n. If $\mathbb{1}_x\, n$ holds, then either there is no write to x with value n, or if there is a write with value n, this is the only such write.

To understand the interaction between value ordering and write limit assertions, consider the following lemma. It states that if there is a possible value order on x with m preceeding n and there is at most one write with these values, then there is a definite value order on x with m preceeding n.

Lemma 9. *For $x \in Var_G$ and $m, n \in Val$, we have:*

$$m \prec_x n \wedge \mathbb{1}_x\, m \wedge \mathbb{1}_x\, n \Rightarrow m \prec\!\!\!\prec_x n \tag{6}$$

$$m \prec\!\!\!\prec_x n \Rightarrow n \not\prec_x m \tag{7}$$

Proof. In Isabelle. □

We discuss the proof of RRC2 in detail. Its proof relies on the following lemma which captures some basic properties about value assertions.

Lemma 10. *Each of the rules below is sound (as per Definition 2), where the statements are decorated with the thread identifier of the executing thread.*

$$\textsc{ZWr}\ \frac{m \neq n}{\{\mathbb{0}_x\, m\}\ y :=_t^{[\mathsf{R}]} n\ \{\mathbb{0}_x\, m\}} \qquad \textsc{DVPres}\ \frac{}{\{m \prec\!\!\!\prec_x n\}\ r \leftarrow_t^{[\mathsf{A}]} y\ \{m \prec\!\!\!\prec_x n\}}$$

$$\textsc{1Intro}\ \frac{i \neq m}{\left\{\begin{array}{c} x_{\mathbf{Init}} = i \\ \wedge\ \mathbb{0}_x\, m \end{array}\right\}\ y :=_t^{[\mathsf{R}]} m\ \{\mathbb{1}_x\, m\}} \qquad \textsc{EncWr}\ \frac{}{\{true\}\ x :=_t^{[\mathsf{R}]} m\ \{x \overset{enc}{=}_t m\}}$$

$$\textsc{EncRd}\ \frac{}{\{true\}\ r \leftarrow_t^{[\mathsf{A}]} x\ \{x \overset{enc}{=}_t r\}} \qquad \textsc{EPO}\ \frac{}{\{x \overset{enc}{=}_t m\}\ r \leftarrow_t^{[\mathsf{A}]} x\ \left\{\begin{array}{c} r \neq m \Rightarrow \\ m \prec_x r \end{array}\right\}}$$

$$\textsc{DVIntro}\ \frac{i \neq n}{\{x_{\mathbf{Init}} = i \wedge \mathbb{0}_x\, n \wedge \mathbb{1}_x\, m \wedge x \overset{enc}{=}_t m\}\ x :=_t^{[\mathsf{R}]} n\ \{m \prec\!\!\!\prec_x n\}}$$

$$\textsc{1PresR}\ \frac{}{\{\mathbb{1}_x\, m\}\ r \leftarrow_t^{[\mathsf{A}]} y\ \{\mathbb{1}_x\, m\}} \qquad \textsc{POrd}\ \frac{}{\{m \prec_x n\}\ C\ \{m \prec_x n\}}$$

Proof. In Isabelle. □

Theorem 11. *The proof outline for RRC2 in Fig. 9 is valid.*

Proof. This proof has been mechanised in Isabelle. Once again, we describe the proof outline to give an overview of how our proofs are used. For local correctness we have the following.

- The initialisation clearly satisfies the precondition of the program, and this implies the precondition of thread 1. The precondition of thread 2 is trivial.
- Next we consider the postcondition of line 1. The first conjunct holds by ZWr, the second conjunct holds by 1Intro and the third by rule EncWr.
- The postcondition of line 2 holds by rule DVIntro.
- In thread 2, the postcondition of line 3 holds by rule EncRd, and the postcondition of line 4 holds by rule EPO.

Next we check interference freedom.

- The precondition of line 1 is stable with respect to lines 3 and 4 by ZWR.
- Next consider the precondition of line 2. The first and second conjuncts are stable with respect to lines 3 and 4 by ZWR and 1PRESR, respectively. The third conjunct is trivially preserved (see Isabelle).
- The postcondition of line 2 holds by DVPRES.
- The precondition of line 3 is trivial and the postcondition of line 3 holds by PORD. □

Correctness of RRC and RRC3 is established by the following theorem.

Theorem 12. *The proof outlines for RRC and RRC3 in Fig. 10 and Fig. 11, respectively are valid.*

Proof. In Isabelle. □

For RRC (Fig. 10), the precondition of line 4 records the fact that thread 3 has encountered a (whatever the value of a may be). Moreover, it guarantees that there is at most one write of x with values 1 and 2. The first conjunct (i.e., $x \overset{enc}{=}_3 a$) allows us to conclude that after x is read at line 4, if a and b are different, then the value for a is possibly ordered before the value for b. The second and third conditions are used to establish the postconditions $\mathbb{1}_x\, 1$ and $\mathbb{1}_x\, 2$. This argument also applies to the assertions in Thread 4. Finally, we show that the postcondition of the program holds as follows, where we assume *post* is the conjunction of the postcondition of each thread.

$$
\begin{aligned}
& \mathit{post} \Rightarrow (a = 1 \wedge b = 2 \wedge c = 2 \Rightarrow d \neq 1) \\
\iff & \mathit{post} \wedge a = 1 \wedge b = 2 \wedge c = 2 \wedge d = 1 \Rightarrow \mathit{false} && \text{(logic)} \\
\impliedby & \mathbb{1}_x\, 1 \wedge \mathbb{1}_x\, 2 \wedge 1 \prec_x 2 \wedge 2 \prec_x 1 \Rightarrow \mathit{false} && \text{(logic)} \\
\impliedby & 1 \prec\!\!\!\prec_x 2 \wedge 2 \prec\!\!\!\prec_x 1 \Rightarrow \mathit{false} && (6) \\
\impliedby & \mathit{true} && (7)
\end{aligned}
$$

The calculation above has been verified with Isabelle, but we recall the proof here as it provides insight into the interactions between different value assertions.

RRC3 (Fig. 11) combines message passing on y with RRC on x. Namely, knowledge of $x := 1$ in thread 1 is transferred to thread 2 using a release-acquire synchronisation on y. Thus, if thread 2 reads 1 for y it must also have encountered 1 for x. Thus, if $r = 1$, then the write on line 4 must have happened *after* the write on line 1. This means that it should be impossible for thread 3 to read 2 for x (at line 5) then read 1 for x (at line 6). Unlike message passing, in RRC3, the "data" variable x is updated both before and after synchronisation. Thus, the assertions on definite values (e.g., $x =_1 1$) become conditional on whether line 4 has already been executed. In particular, the antecedent $\mathbb{0}_x\, 2$ allows us to assume that line 4 has not yet been executed. As with RRC, we must separately prove that the conjunction of the postconditions of the threads implies the postcondition of the program. This proof is mechanised in Isabelle, and is elided here.

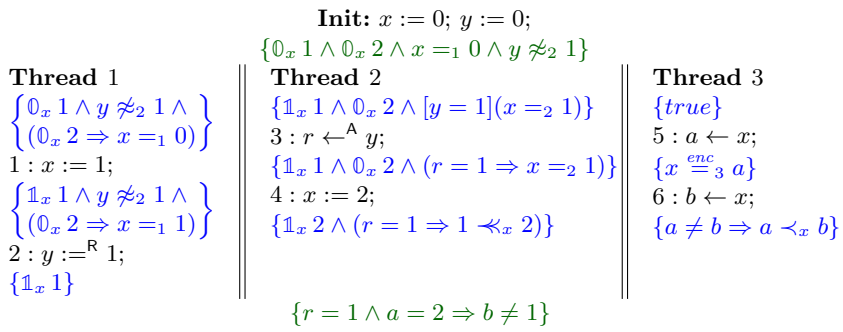

Fig. 11. Proof outline for RRC3, where $x, y \in \mathit{Var}_G$ and $a, b \in \mathit{Var}_L$

5.5 Peterson's algorithm

We turn to our final case study, the verification of the mutual exclusion property of a version of Peterson's algorithm. The complexity of this case study is much greater than our earlier examples. This program contains a loop, features a careful mixture of relaxed and release/acquire operations to the same variable, and an RMW operation whose precise semantics is critical to the correctness of the algorithm.

Our version of Peterson's algorithm[6], presented in Fig. 12 is a mutual exclusion algorithm for two threads implemented for RC11 using release-acquire annotations [34]. As with the original algorithm, variable flag_i, for $i \in \{1, 2\}$ is used to indicate whether thread i intends to enter its critical section. In this version of the algorithm, we let flag_i range over $\{0, 1\}$, where 0 is used for the boolean value "false", and 1 is used for the boolean value "true". The shared variable *turn* is used to cause a thread to "give way" when both threads intend to enter their critical sections at the same time. Our verification uses auxiliary variables after_i for each thread i (as does the proof for a sequentially consistent setting in [7]), the purpose of which we describe below.

We describe the algorithm for thread 1; the other thread is symmetric. For now, we ignore the assertions. The flag variable is set to 1 (line 1) using a relaxed write (which cannot induce any synchronisation), but is set to 0 (line 7) using a release annotation. The intention of the latter is to synchronise this write (of 0 to flag_1) with the read of flag_1 at line 3 in thread 2. The value of *turn* is set using a **swap** command. The **swap** is implemented using an RC11 RMW operation that has both the release and acquire annotations. When the **swap** is executed, as part of the same transition, the auxiliary variable after_1 is also set, indicating that thread 1 is ready to enter the busy wait loop beginning at line 3, and then to enter the critical section.

The busy wait loop forces thread 0 to wait until either flag_2 is 0 (indicating that thread 2 is not trying to enter the critical section) or $\mathit{turn} = 1$ (indicating that it is thread 1's turn to enter the critical section). Note that the read of *turn* within the guard of the busy wait loop (line 5) is relaxed.

[6] For simplicity our version of the algorithm does not have an outermost loop.

Init: $\mathit{flag}_1 = 0 \land \mathit{flag}_2 = 0 \land \mathit{turn} = 0$

Thread 1

$\{\neg \mathit{after}_1 \land \mathit{flag}_1 =_1 0 \land \mathit{turn} \not\approx_2 2 \land (\mathbf{C}^0_{\mathit{turn}} \lor [\mathit{turn} = 1](\mathit{flag}_2 =_1 1))$
$\{\land (\mathit{after}_2 \Rightarrow \mathbf{C}^1_{\mathit{turn}} \land [\mathit{turn} = 1](\mathit{flag}_2 =_1 1))$

1: $\mathit{flag}_1 := 1$;
$\{\neg \mathit{after}_1 \land \mathit{flag}_1 =_1 1 \land \mathit{turn} \not\approx_2 2 \land (\mathit{after}_2 \Rightarrow \mathbf{C}^1_{\mathit{turn}} \land [\mathit{turn} = 1](\mathit{flag}_2 =_1 1))\}$
2: $\langle \mathit{turn}.\mathsf{swap}(2)^{\mathsf{RA}} ; \mathit{after}_1 := \mathit{true}\rangle$
$\{\mathit{after}_1 \land (\mathit{after}_2 \land (\mathit{flag}_2 \approx_1 0 \lor \mathit{turn} \approx_1 1) \Rightarrow \mathit{turn} =_2 1)\}$
do
3: $r_1 \leftarrow^A \mathit{flag}_2$
$\{\mathit{after}_1 \land (\mathit{after}_2 \land (r_1 = 0 \lor \mathit{turn} \approx_1 1 \lor \mathit{flag}_2 \approx_1 0) \Rightarrow \mathit{turn} =_2 1)\}$
4: $r_2 \leftarrow \mathit{turn}$
$\{\mathit{after}_1 \land (\mathit{after}_2 \land (r_1 = 0 \lor r_2 = 1 \lor \mathit{turn} \approx_1 1 \lor \mathit{flag}_2 \approx_1 0) \Rightarrow \mathit{turn} =_2 1)\}$
5: **until** $(r_1 = 0 \lor r_2 = 1)$
$\{\mathit{after}_1 \land (\mathit{after}_2 \Rightarrow \mathit{turn} =_2 1)\}$
6: Critical section ;
7: $\langle \mathit{flag}_1 :=^{\mathsf{R}} 0 ; \mathit{after}_1 := \mathit{false}\rangle$

Fig. 12. Peterson's algorithm (adapted from [34]) and its proof outline. **Thread** 2 (not shown) is symmetric

We turn now to the proof that this version of Peterson's algorithm has the mutual exclusion property. We prove mutual exclusion in two steps. First, we show that the given proof outline is valid, and second, that the conjunction of the precondition of thread 1's critical section (line 6) and thread 2's must be false. Therefore, the two threads cannot simultaneously be in their critical sections.

We deal with the second step first by showing that the formula below is *false*:

$$\mathit{after}_1 \land (\mathit{after}_2 \Rightarrow \mathit{turn} =_2 1) \land \mathit{after}_2 \land (\mathit{after}_1 \Rightarrow \mathit{turn} =_1 2)$$

It is easy to see that this implies $\mathit{turn} =_1 2 \land \mathit{turn} =_2 1$. However, by (5) this situation is impossible.

The first step is more elaborate and we only describe certain aspects. The precondition of line 3 is also an invariant of the busy wait loop. This assertion ensures that if thread 1 is able to exit the busy wait loop, then the precondition of the critical section will be satisfied. Note that thread 1 exits the loop if it reads 0 from flag_2 (which is only possible when $\mathit{flag}_2 \approx_1 0$) or it reads 1 from *turn* (which is only possible when $\mathit{turn} \approx_1 1$). The invariant states that if one of these conditions holds in a state where thread 2 is waiting to enter the critical section (that is, after_2), we can conclude $\mathit{turn} =_1 2$ as required.

Proving that the precondition of line 3 is satisfied in the post-state of line 2 requires using a feature of our assertion language, closely related to the semantics of RMW operations, that we now introduce. Recall from the UPDATE rule in Fig. 5 that whenever a write w is read-from by an RMW operation, w becomes *covered*, so that no later write (or RMW) operation can be inserted between w and the RMW. This feature of RC11 is critical to the correctness of Peterson's algorithm. Observe that the *turn* variable is only modified by RMW operations, and therefore every write to *turn* is covered, except the last. The assertion $\mathbf{C}^n_x$,

defined as follows, can be used to describe situations such as these.

$$\mathbf{C}_x^n \quad = \quad \lambda\sigma.\ \forall w \in \sigma.writes_x.\ w \notin \sigma.covered \Rightarrow wrval(w) = n \wedge w = last(W, x)$$

So $\mathbf{C}_x^n$ means that every write to x except the last is covered and the value written by that last write is n.

The precondition of line 2 asserts that if thread 2 is ready to enter the critical section (that is, $after_2$) then the RMW to be executed at line 2 must read from the last write which has value 1 (that is, $\mathbf{C}_{turn}^1$) and when this RMW occurs then thread 1 will definitely see $flag_2$ set (that is, $[turn = 1](flag_2 =_1 1)$). This is enough to show that if $after_2$ then in the post-state of the RMW, $flag_2 \not\approx_1 0$ which is sufficent to prove the precondition of line 2.

Of course, the sequential reasoning above must be combined with an interference freedom check, which is supported by a set of basic lemmas describing how $\mathbf{C}_x^n$ is updated. This leads to the following theorem, which establishes validity of the proof outline.

Theorem 13. *The proof outline of Peterson's algorithm (Fig. 12) is valid.*

Proof. In Isabelle. □

We note that Peterson's algorithm represents a challenge in deductive verification. Unlike the litmus tests presented above, there is sufficient complexity in the algorithm and the resulting proof outline so that pen-and-paper proofs cannot be trusted. Using our mechanisation, we explored several variations of the proof outline in Fig. 12, and discovered simplifications to our original pen-and-paper proofs.

6 Mechanisation

As already mentioned, the operational semantics as well as all lemmas and theorems presented in this paper have been mechanised in Isabelle. In this section, we discuss our mechanisation effort.

To prove the lemmas about basic assertions, we typically prove a more general result relating to reads and writes, which are then specialised so that they can be used in the verification of the algorithms. For example, we first prove the lemma in Fig. 13, which describes changes to definite values and applies to any writing transition. This is then specialised to the corollaries on the right, which are easier for Isabelle to find when performing the verification of the proof outlines.

The generic lemmas require some amount of interactive work. However, once verified, it is straightforward to use the generic lemmas to prove the corollaries. For example, corollary `d_obs_WrX_set` in Fig. 13 is verified with the command "`by (metis WrX_def avar.simps(2) d_obs_Wr_set wr_val.simps(1))`", which is found automatically by Isabelle's built in `sledgehammer` tool [9].

Such lemmas and corollaries are in turn used in the proofs of programs. First the program state (i.e., Σ_{C11}) is encoded as a `record` type with a special variable that models the C11 state. The programs themselves are encoded as a

```
lemma d_obs_Wr_set:
  assumes "wfs σ"
    and "wr_val a = Some n"
    and "avar a = x"
    and "[x =t m] σ"
    and "step t a σ σ'"
  shows "[x =t n] σ'"
```

```
corollary d_obs_WrX_set:
  "wfs σ ⟹ [x =t m] σ ⟹
  σ [x := n]t σ' ⟹ [x =t n] σ'"

corollary d_obs_WrR_set :
  "wfs σ ⟹ [x =t m] σ ⟹
  σ [x :=R n]t σ' ⟹ [x =t n] σ'"

corollary d_obs_RMW_set :
  "wfs σ ⟹ [x =t m] σ ⟹
  σ RMW[x,w,n]t σ' ⟹ [x =t n] σ'"
```

Fig. 13. Isabelle encoding of basic axioms over C11 assertions

relation over these records with program counters modelling control flow. This allows the proof outlines to be encoded as predicates mapping program counters to the assertions at that control point. We then verify a set of lemmas that guarantee local correctness and interference freedom, where we decompose proofs and apply case analysis over the individual program steps (e.g., reads, writes for each thread). Once a proof has been decomposed, `sledgehammer` is able to find the relevant corollaries (e.g., those in Fig. 13) to discharge proofs automatically.

We are currently developing improved automation[7] for program verification by incorporating the operational semantics into the existing Owicki-Gries library that is included in the existing Isabelle distribution [25].

7 Related Work

The semantics and verification of programs running on weak memory models has recently received a lot of attention. Lahav [20] gives a brief survey for C11.

Our timestamp based operational semantics is motivated by ideas in [12] and is similar to the semantics of Kaiser et al. [15,16]. We note there are differences in coverage of the memory models in [12,15,16]. Dolan et al [12] cover a sequentially consistent (SC) and relaxed accesses for OCAML, where the SC operations behave in a Java-like manner. Kaiser et al [15] covers non-atomics and release-acquire, while Kang et al [16] support a much larger fragment of C11, including so-called load-buffering cycles.

Abdulla et al. have shown the reachability problem for release-acquire to be *undecidable* [2]. A number of works target *model checking* for weak memory, e.g., by explicitly encoding architectural structures leading to weak behaviour, like store buffers [32,5]. Ponce de León et al. [29,13] have developed a bounded model checker for weak memory models, taking the axiomatic description of a memory model as input. (Bounded) model checkers for specific weak memory models are furthermore the tools CBMC [6] (for TSO), Nidhugg [1] (for TSO and PSO), RCMC [17] (for C11) and GenMC [18] (again, parametric in memory model).

[7] A set of preliminary results is available `brijeshdongol.github.io/mechanisation/OG-Isabelle.zip`.

A (non-automatic) reasoning technique for proving invariants – parameterised by a weak memory model – has been proposed by Alglave and Cousot [4]. They propose a new semantics, different from an operational one without any coherence order (or modification order) constraining the order of writes to memory. Their assertions contain so-called pythia variables to uniquely identify values of read events, and require a separate communication proof (differentiating their method from standard Owicki-Gries reasoning). They say "In addition to the initialisation, sequential, and non-interference proof, the main difference with Owicki and Gries [26] (and Lamport 1977) is the use of pythia variables and the read-from relation in assertions and the communication proof showing that reads-from is well-formed." [4]. Our method in contrast only requires the initialisation, sequential, and non-interference proofs as with the original technique.

Recent works have furthermore introduced specific *program logics* for reasoning about weak memory models. Only a few, however, target the C11 memory models. Closest to us is the work of Lahav and Vafeiadis [21] who also develop an Owicki-Gries style proof calculus, however, in a rely-guarantee form. Contrary to us, their approach requires adapting a number of proof rules for compound statements, including the non-interference check. This means that standard proof techniques, e.g., introduction of auxiliary variables, can result in an unsound proof. We furthermore handle relaxed accesses, whereas Lahav and Vafeiadis assume all accesses are release-acquire.

A frequently employed starting point for program logic is separation logic, for which a number of extensions to weak memory exist (GPS [33], RSL [15]). Svendsen et al. [31] propose a separation logic based on the promising semantics of Kang et al. [16]. The principle of ownership transfer used therein naturally fits to message passing using release acquire. Prover support for such separation logic based proofs – like ours with Isabelle – has been proposed by Summers et al. [30] and the Iris proof system [15].

8 Conclusion

In this paper, we have introduced an assertion language for RC11 which allows to re-use the entire Owicki-Gries proof calculus except for the axiom of assignment. The assertion language is based on an operational semantics for RC11 which we have shown to be sound wrt. standard axiomatic semantics. We have exemplified reasoning on a number of standard RC11 litmus tests as well as a RC11 annotated version of Peterson's algorithm. All proofs ranging are mechanised within Isabelle — this includes soundness of the basic axioms for weak memory reads, writes and updates, and validity of proof outlines for the examples presented.

As future work, we aim at establishing completeness of our proof calculus, via a weakest precondition version of the axiom of assignment. Further extensions may include fragments of C11 larger than RC11. Of particular interest are fragments that allow the load buffering example to terminate with postcondition $r1 = 1 \wedge r2 = 1$ [8,16].

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. Acta Inf. **54**(8), 789–818 (2017)
2. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.N.: Verification of programs under the release-acquire semantics. In: McKinley and Fisher [24], pp. 1117–1132
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
4. Alglave, J., Cousot, P.: Ogre and Pythia: an invariance proof method for weak consistency models. In: Castagna and Gordon [10], pp. 3–18
5. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP. LNCS, vol. 7792, pp. 512–532. Springer (2013)
6. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 141–157. Springer (2013)
7. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009)
8. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL. pp. 634–648. ACM (2016)
9. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: IJCAR. Lecture Notes in Computer Science, vol. 6173, pp. 107–121. Springer (2010)
10. Castagna, G., Gordon, A.D. (eds.): POPL. ACM (2017)
11. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) PPoPP. pp. 355–365. ACM (2019)
12. Dolan, S., Sivaramakrishnan, K., Madhavapeddy, A.: Bounding data races in space and time. In: PLDI. pp. 242–255. PLDI 2018, ACM, New York, NY, USA (2018)
13. Gavrilenko, N., Ponce de Le'on, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV. LNCS, vol. 11561, pp. 355–365. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19, `https://doi.org/10.1007/978-3-030-25540-4_19`
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
15. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: Müller, P. (ed.) ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
16. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna and Gordon [10], pp. 175–189
17. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–17:32 (2018)
18. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: McKinley and Fisher [24], pp. 96–110
19. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI. pp. 618–632. ACM (2017)
20. Lahav, O.: Verification under causally consistent shared memory. SIGLOG News **6**(2), 43–56 (2019)

21. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP. LNCS, vol. 9135, pp. 311–323. Springer (2015)
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979)
23. Lynch, N.A.: Distributed algorithms. Elsevier (1996)
24. McKinley, K.S., Fisher, K. (eds.): PLDI. ACM (2019)
25. Nipkow, T., Nieto, L.P.: Owicki/Gries in Isabelle/HOL. In: FASE. Lecture Notes in Computer Science, vol. 1577, pp. 188–203. Springer (1999)
26. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
27. Paulson, L.C.: Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow), LNCS, vol. 828. Springer (1994)
28. Podkopaev, A., Sergey, I., Nanevski, A.: Operational aspects of C/C++ concurrency. CoRR **abs/1606.01400** (2016)
29. Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC with memory models as modules. In: Bjørner, N., Gurfinkel, A. (eds.) FMCAD. pp. 1–9. IEEE (2018)
30. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs. In: Beyer, D., Huisman, M. (eds.) TACAS. LNCS, vol. 10805, pp. 190–209. Springer (2018)
31. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP. LNCS, vol. 10801, pp. 357–384. Springer (2018)
32. Travkin, O., Mütze, A., Wehrheim, H.: SPIN as a linearizability checker under weak memory models. In: Bertacco, V., Legay, A. (eds.) HVC. LNCS, vol. 8244, pp. 311–326. Springer (2013)
33. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: Black, A.P., Millstein, T.D. (eds.) OOPSLA. pp. 691–707. ACM (2014)
34. Williams, A.: `https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html` (2018), accessed: 2018-06-20

A Correctness of the Operational Semantics

In this section, we prove that the operational semantics presented in this paper is sound w.r.t to a standard version of the RC11 axiomatic semantics.

In earlier work [11], we presented an operational semantics for RC11, which we proved to be equivalent to a suitable fragment of the RC11 axiomatic semantics of [8]. In what follows, we refer to the semantics of [11] as the *Explicit semantics* as the states of its memory semantics are simply consistent RC11 executions. We refer to the semantics presented in this paper as the *View semantics*.

In Section A.1, we describe the operational semantics of [11]. In Section A.2, we present a simulation relation from the View memory semantics to the Explicit memory semantics. This is sufficent to show that all sequences of operations accepted by the View semantics are also accepted by the Explicit semantics.

In what follows, we refer to states of the Explicit semantics using the variables E, E' and states of the View semantics as V, V'.

A.1 The Explicit Memory Semantics

We refer the reader to [11] for a full discussion of the Explicit semantics. In this paper, we present only the semantics that relates to memory operations, and we do so only briefly.

States of the Explicit semantics are simply RC11 executions, of the form $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$. Where $\mathsf{sb}, \mathsf{rf}, \mathsf{mo}$ are relations on the set of operations X with their usual meanings. That is $\mathsf{sb} \subseteq X \times X$ is the *sequenced-before* relation; $\mathsf{rf} \subseteq \mathsf{W} \times \mathsf{R} \cap X \times X$ is the *reads-from* relation; and $\mathsf{mo} \subseteq \mathsf{W} \times \mathsf{W} \cap X \times X$ is the *modification order*.

In an Explicit state $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ we let $X \subseteq \mathsf{Act} \times \mathbb{Q} \times T$, where Act is the set of operations and$T$ is the set of threads. In the original presentation, [11] we specify $X \subseteq \mathsf{Act} \times G \times T$ where G is a set of tags that are not further specified, which serves to distinguish repeated occurrences of the same operation. Here, we let $G = \mathbb{Q}$, for uniformity with]the view semantics.

The explicit semantics uses additional *synchronized-with* (denoted sw) and *happens-before* (denoted hb) relations, defined as follows:

$$\mathsf{sw} = \mathsf{rf} \cap (\mathsf{W_R} \times \mathsf{R_A}) \tag{8}$$

$$\mathsf{hb} = (\mathsf{sb} \cup \mathsf{sw})^+ \tag{9}$$

In the explicit semantics, all variables are initialised by a special *initialising thread* $0 \in T$. Define the set of *initialising writes* to be $\mathsf{IWr} = \{w \in \mathsf{W} \mid \mathit{tid}(w) = 0\}$. The initial states of our operational model are those of the form $E_0 = ((I, \emptyset), \emptyset, \emptyset)$ where $I \subseteq \mathsf{IWr}$, and for each variable x, there is exactly one write $w \in I$ such that $\mathit{var}(w) = x$. For a state $E = ((X, \_), \_, \_)$, let $I_E = X \cap \mathsf{IWr}$.

The relation $\mathsf{fr} = (\mathsf{rf}^{-1}; \mathsf{mo}) \backslash Id$ (where ; is relational composition) is the "from-read" relation, that relates each read to all writes that are mo-after the write the read has read from. We must subtract Id (identity) edges from $\mathsf{rf}^{-1}; \mathsf{mo}$ to cope with update events, which have the potential to induce reflexivity in fr [8,19].

In addition, our semantics uses the *extended coherence order* [19], denoted eco, which fixes the order of reads and writes to each variable. Formally we define:

$$\mathsf{eco} = (\mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf})^+ \tag{10}$$

The set of *encountered writes* are the writes that thread t is aware of (either directly or indirectly) in state $E = ((X, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$, and are given by:

$$E.EW = \{w \in \mathsf{W} \cap D \mid \exists e \in D.\ \mathit{tid}(e) = t \wedge (w, e) \in \mathsf{eco}^?; \mathsf{hb}^?\} \cup I_E$$

where $R^?$ is the reflexive closure of relation R. Thus, for each $w \in E.EW$, there must exist an event e of thread t such that w is either eco- or hb- or $\mathsf{eco}; \mathsf{hb}$-prior to e.

$$\textsc{Read}\ \frac{\begin{array}{c}a \in \{rd(x,n), rd^{\mathsf{A}}(x,n)\} \quad wrval(w) = n \\ w \in E.OW(t,x) \quad \mathsf{rf}' = \mathsf{rf} \cup \{(w,e)\} \quad \mathsf{mo}' = \mathsf{mo}\end{array}}{((X,\mathsf{sb}),\mathsf{rf},\mathsf{mo}) \overset{e}{\Rightarrow} ((X,\mathsf{sb}) + e, \mathsf{rf}', \mathsf{mo}')}$$

$$\textsc{Write}\ \frac{\begin{array}{c}a \in \{wr(x,n), wr^{\mathsf{R}}(x,n)\} \quad w \in E.OW(t,x) \backslash E.covered \\ \mathsf{rf}' = \mathsf{rf} \quad \mathsf{mo}' = \mathsf{mo}[w,e]\end{array}}{((X,\mathsf{sb}),\mathsf{rf},\mathsf{mo}) \overset{e}{\Rightarrow} ((X,\mathsf{sb}) + e, \mathsf{rf}', \mathsf{mo}')}$$

$$\text{RMW}\ \frac{\begin{array}{c}a = upd^{\mathsf{RA}}(x,m,n) \quad w \in E.OW(t,x) \backslash E.covered \\ wrval(w) = m \quad \mathsf{rf}' = \mathsf{rf} \cup \{(w,e)\} \quad \mathsf{mo}' = \mathsf{mo}[w,e]\end{array}}{((D,\mathsf{sb}),\mathsf{rf},\mathsf{mo}) \overset{e}{\Rightarrow} ((X,\mathsf{sb}) + e, \mathsf{rf}', \mathsf{mo}')}$$

Fig. 14. Event semantics assuming $E = ((X,\mathsf{sb}),\mathsf{rf},\mathsf{mo})$, $e = (g,a,t)$ and $g \notin tags(X)$

From these, we determine the *observable writes*, which are the writes that thread t can observe in its next read. These are defined as:

$$E.OW(t,x) = \{w \in \mathsf{W} \cap E.X \mid loc(w) = x \wedge \forall w' \in E.EW(t) \wedge (w,w') \notin E.\mathsf{mo}\}$$

Thus, observable writes are not succeeded by any encountered write in modification order, i.e., the thread has not seen another write overwriting the value being read.

Finally, to guarantee *atomicity* of the update events, there cannot be any write operations (in modification order) between the write that an update reads from and the write of the update itself. We therefore define the set of *covered writes* as follows:

$$E.covered = \{w \in \mathsf{W} \cap E.X \mid \exists u \in \mathsf{U}.\ (w,u) \in \mathsf{rf}\}$$

The transition relation of the Explicit semantics is given in Figure 14.

Lemma 14 (Invariants of the Explicit Semantics). *If $E = (X,\mathsf{sb},\mathsf{rf},\mathsf{mo})$ and E is reachable from an initial state via a sequence of transitions of the Explicit semantics, then*

- mo *totally orders the writes to each variable x. That is, for all $w,w' \in X \cap \mathsf{W}$, such that $loc(w) = loc(w')$.*

$$(w,w') \in \mathsf{mo} \vee (w',w) \in \mathsf{mo} \tag{11}$$

- *For each variable x, there is at least one write. That is, for each x*

$$\exists w \in X \cap \mathsf{W}.loc(w) = x \tag{12}$$

Proof. These are simple inductive invariants of the semantics.

A.2 Soundness of the View Semantics

We turn now to the soundness of the View semantics. Note that in the semantics as presented in the body of this paper, writes are recorded in the state as a pair $(w, q) \in \mathsf{W} \times \mathbb{Q}$. For uniformity with the Explicit state semantics, in this proof we record writes in the state as a triple $(w, q, t) \in \mathsf{W} \times \mathbb{Q} \times T$. The transition rule for writes for the View semantics now becomes

$$\textsc{Write}\ \frac{\begin{array}{c} a \in \{wr(x,n), wr^{\mathsf{R}}(x,n)\} \quad (w,q) \in \sigma.OW(t,x) \setminus \sigma.covered \quad fresh(q,q') \\ writes' = \sigma.writes \cup \{(a,q',t)\} \qquad tview'_t = \sigma.tview_t[x := (a,q',t)] \end{array}}{\sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, writes := writes']}$$

The other rules are changed similarly. This transformation has no effect on the observable behaviour of the semantics. But now, the set of writes $V.writes$ in some View state now has the same type (or structure) as the set of events in some Explicit state $E.X$, whic will prove to be convenient later.

In this section, we prove the following theorem, which states that every behaviour of the View semantics is also a behaviour of the Explicit semantics.

Theorem 15. *For every sequence of steps of the View semantics*

$$V_0 \overset{a_1}{\rightsquigarrow} V_1 \overset{a_2}{\rightsquigarrow} \ldots \overset{a_n}{\rightsquigarrow} V_n$$

such that V_0 is an initial state of the View semantics, there is a sequence of steps of the Explicit semantics

$$V_0 \overset{a_1}{\Longrightarrow} V_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} V_n$$

such that E_0 is an initial state of the Explicit semantics, having the same sequence of actions.

We prove this using the method of *simulation relations* (see e.g., [23]).

First some preliminary definitions.

Definition 16. *Given an Explicit state $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$*

1. *(Events of a thread.) Let $E.X_t = \{e \in X \mid tid(e) = t\}$*
2. *(mo-maximal.) For $S \subseteq X$, let $E.\mathbf{max}_{\mathsf{mo}}(S) = \{w \in S \cap \mathsf{W} \mid \forall w' \in X.(w, w') \notin \mathsf{mo}\}$*
3. *(Causal closure.) For $S \subseteq X$, let $E.\mathbf{cclose}(S) = \{e \mid \exists e' \in S.(e, e') \in \mathsf{eco}^? \circ \mathsf{hb}^?\} \cup I_E$*
4. *(View modification order.) Let $V.\mathsf{mo} = \{(w, w') \mid w, w' \in V.writes \wedge tst(w) < tst(w')\}$*

where $P.\mathsf{eco}$ and $P.\mathsf{hb}$ are defined for an RC11 execution as usual.

We sometimes omit the Explicit state from these auxilliary variables, when the state in question is clear from context. For example, $E.\mathbf{max}_{\mathsf{mo}}$ sometimes becomes $\mathbf{max}_{\mathsf{mo}}$.

Lemma 17. *For any Explicit state* $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$

$$EW(t) = \mathbf{cclose}(X_t) \cap \mathsf{W} \tag{13}$$

and for any set $S \subseteq X$*,*

1.

$$\mathbf{max}_{\mathsf{mo}}(S) \subseteq X \tag{14}$$

2.

$$\mathbf{max}_{\mathsf{mo}}(S) \subseteq \mathsf{W} \tag{15}$$

3. for each variable x *there is precisely one write* $w \in \mathbf{max}_{\mathsf{mo}}(S)$ *such that* $loc(w) = x$*.*

Proof. We prove each property in turn.

- (13) This follows immediately from the definitions of EW and **cclose**:

$$\begin{aligned} w \in EW(t) &\iff w \in \mathsf{W} \wedge \exists e' \in X_t.(e, e') \in \mathsf{eco}^?; \mathsf{hb} \\ &\iff w \in \mathsf{W} \wedge w \in \mathbf{cclose}(X_t) \\ &\iff w \in \mathbf{cclose}(X_t) \cap \mathsf{W} \end{aligned}$$

- (14) By the definition of $\mathbf{max}_{\mathsf{mo}}$, if $w \in \mathbf{max}_{\mathsf{mo}}(S)$ then $w \in S \subseteq X$
- (15) By the definition of $\mathbf{max}_{\mathsf{mo}}$, if $w \in \mathbf{max}_{\mathsf{mo}}(S)$ then $w \in \mathsf{W}$.
- (3) By Property 11 of the explicit semantics, $E.\mathsf{mo}$ totally orders the writes to each x. Thus, for any distinct writes v, w such that $loc(v) = loc(w) = x$, either $(v, w) \in E.\mathsf{mo}$ or $(w, v) \in E.\mathsf{mo}$. In each case, the mo-earlier of the two writes cannot be mo-maximal, and therefore v and w cannot both be in $E.\mathbf{max}_{\mathsf{mo}}(S)$. This ensures uniqueness. The fact that a write to x exists is immediate from Property 12 of the Explicit semantics.

Property 3 shows that $\mathbf{max}_{\mathsf{mo}}(X)$ defines a function from variables to writes. We denote by $\mathbf{max}_{\mathsf{mo}}(X, x)$ the unique write in $\mathbf{max}_{\mathsf{mo}}(X)$ such that $loc(w) = x$.

For any Explicit state E, and any $S \subseteq E.X \cap \mathsf{W}$, we say that S is *complete* if for every location x there is some $w \in S$ with $x = loc(w)$. Note that if S is complete then $\mathbf{max}_{\mathsf{mo}}(S, x)$ is defined.

Lemma 18. *For any Explicit state* E*, and any* $S \subseteq E.X \cap \mathsf{W}$*,* $E.\mathbf{cclose}(S)$ *is complete.*

Proof. $I_E \subseteq E.\mathbf{cclose}(S)$, and I_E is clearly complete.

Lemma 19. *For any Explicit state* E*, and any complete* $S, S' \subseteq E.X$ *where* S *is complete, if there exists a* $w \in S'$ *such that* $loc(w) = x$*,*

$$\begin{aligned} &E.\mathbf{max}_{\mathsf{mo}}(S \cup S', x) \\ &= \begin{cases} E.\mathbf{max}_{\mathsf{mo}}(S, x) & \text{if } (E.\mathbf{max}_{\mathsf{mo}}(S', x), E.\mathbf{max}_{\mathsf{mo}}(S, x)) \in E.\mathsf{mo} \\ E.\mathbf{max}_{\mathsf{mo}}(S', x) & \text{otherwise} \end{cases} \end{aligned} \tag{16}$$

otherwise

$$E.\mathbf{max}_{\mathsf{mo}}(S \cup S', x) = E.\mathbf{max}_{\mathsf{mo}}(S, x) \tag{17}$$

Proof. If there exists a $w \in S'$ such that $loc(w) = x$ then both $E.\mathbf{max}_{\mathsf{mo}}(S', x)$ and $E.\mathbf{max}_{\mathsf{mo}}(S, x)$ are defined and $E.\mathbf{max}_{\mathsf{mo}}(S \cup S', x)$ is the maximum of these.

If there is no such write then

$$(S \cup S') \cap \{w \mid loc(w) = x\} = S \cap \{w \mid loc(w) = x\}$$

and thus $E.\mathbf{max}_{\mathsf{mo}}(S \cup S', x) = E.\mathbf{max}_{\mathsf{mo}}(S, x)$ as required.

Note that for every step of the Explicit semantics $E \overset{e}{\Rightarrow} E'$ there is some unique write that the event e interacts with, for example the write that e reads from if e is a read or RMW. The write w mentioned in each of the Explicit semantics transition rules of Figure 14. In what follows we exhibit this write as a label on the transition relation. Thus we write $E \overset{w,e}{\Longrightarrow} E'$ iff $E \overset{e}{\Rightarrow} E'$ and w is the write that e interacts with.

Definition 20. *Given an Explicit state E, we say that a thread t is* before *a write w at location x, denoted $t\mathbf{before}_x w$ if*

$$(E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x), E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x)) \in E.\mathsf{mo}$$

Likewise, we say that a write w is before *a write t at location x, denoted $w\mathbf{before}_x t$ if*

$$(E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x), E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)) \in E.\mathsf{mo}$$

Lemma 21 (Max Encountered Writes). *For any Explicit transition $E \overset{w,e}{\Longrightarrow} E'$ where w and e synchronise, and for all x, if e is an (acquiring) read then*

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = \begin{cases} E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) & \text{if } w \ \mathbf{before}_x \ t \\ E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x) & \text{otherwise} \end{cases} \tag{18}$$

and if e is an update then

$$\begin{aligned} &E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\ &= \begin{cases} e & \text{if } loc(w) = x \\ E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) & \text{if } loc(w) \neq x \wedge w \ \mathbf{before}_x \ t \\ E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}) & \text{otherwise} \end{cases} \end{aligned} \tag{19}$$

Further, for any Explicit transition $E \overset{w,e}{\Longrightarrow} E'$ where w and e do not synchronise, if e is a read then

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = \begin{cases} w & \text{if } loc(w) = x \\ E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) & \text{otherwise} \end{cases} \tag{20}$$

and if e is a write or update then

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = \begin{cases} e & \text{if } loc(w) = x \\ E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) & \text{otherwise} \end{cases} \tag{21}$$

Proof. We prove each in turn.

(Equation 18) Consider

$$
\begin{array}{ll}
& E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\
= & \quad \text{(trans-rel)} \\
& E.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\
= & \quad \text{(trans-rel and defn of } \mathbf{cclose}) \\
& E.\mathbf{max}_{\mathsf{mo}}(E.EW(t) \cup E.\mathbf{cclose}(\{w\}), x) \\
= & \quad \text{(see below)} \\
& \begin{cases} E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) & \text{if } w \ \mathbf{before}_x \ t \\ E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x) & \text{otherwise} \end{cases}
\end{array}
$$

This last step is a consequence of Lemma 19 and the fact that both $E.EW(t)$ and $E.\mathbf{cclose}(\{w\})$ are complete.

(Equation 19) We first consider the case when $x = loc(e)$. In this case

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = e$$

as required. If $x \neq loc(e)$, then

$$
\begin{array}{ll}
& E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\
= & \quad \text{(trans-rel)} \\
& E.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\
= & \quad \text{(trans-rel and defn of } \mathbf{cclose}) \\
& E.\mathbf{max}_{\mathsf{mo}}(E.EW(t) \cup E.\mathbf{cclose}(\{w\}) \cup \{e\}, x) \\
= & \quad (x \neq loc(e) \text{ and Lemma 19}) \\
& E.\mathbf{max}_{\mathsf{mo}}(E.EW(t) \cup E.\mathbf{cclose}(\{w\}), x) \\
= & \quad \text{(see below)} \\
& \begin{cases} E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) & \text{if } w \ \mathbf{before}_x \ t \\ E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x) & \text{otherwise} \end{cases}
\end{array}
$$

Agin, this last step is a consequence of Lemma 19 and the fact that both $E.EW(t)$ and $E.\mathbf{cclose}(\{w\})$ are complete.

(Equation 20) We first consider the case when $x = loc(e)$. In this case

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = e$$

as required. If $x \neq loc(e)$, then

$$\begin{array}{ll} & E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\ = & \quad \text{(trans-rel)} \\ & E.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) \\ = & \quad \text{(trans-rel and defn of } \mathbf{cclose}) \\ & E.\mathbf{max}_{\mathsf{mo}}(E.EW(t) \cup \{w\} \cup \{e\}, x) \\ = & \quad \text{(because } x \neq loc(e) \text{ and Lemma 19)} \\ & E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) \end{array}$$

(Equation 21) The proof here is identical to that for Equation 20.

We define a simulation relation R between a View state $V = (writes, tview, mview, covered)$ and an Explicit state $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ to be the conjunction of the following properties.

- Both executions contain the same set of writes:

$$V.writes = E.X \cap \mathsf{W} \tag{22}$$

- For all threads t and variables x,

$$V.tview(t, x) = E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) \tag{23}$$

- For all $w \in writes$ and variables x,

$$V.mview(w, x) = E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x) \tag{24}$$

- Both executions agree on the order of writes.

$$V.\mathsf{mo} = E.\mathsf{mo} \tag{25}$$

  recall that $V.\mathsf{mo} = \{(w, w') \mid w, w' \in V.writes \wedge tst(w) < tst(w')\}$.
- The executions agree on the set of covered writes:

$$V.covered = E.covered \tag{26}$$

Our first lemma relates the viewfront and the observable writes of R-related states.

Lemma 22. *For any View state V and Explicit state E such that $(V, E) \in R$, and for all threads t and locations x, $V.OW(t, x) = E.OW(t, x)$.*

Proof. First, observe that for all w such that $loc(w) = x$

$$tst(V.tview_t(x)) \leq tst(w) \iff (E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x), w) \in E.\mathsf{mo} \tag{27}$$

But this is an immediate consequence of the fact that $V.tview(t, x) = E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)$ and that $E.\mathsf{mo} = V.\mathsf{mo}$ (both of these are consequences of R).

But now, because $E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)$ is mo-maximal among the set of encountered writes, we have

$$(E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x), w) \in E.\mathsf{mo} \iff \forall w' \in E.EW(t), x).(w, w') \notin E.\mathsf{mo} \tag{28}$$

Now consider

$$\begin{array}{ll} & w \in V.OW(t, x) \\ \iff & \quad \text{(definition)} \\ & w \in V.writes \wedge loc(a) = x \wedge tst(V.tview_t(x)) \leq tst(w) \\ \iff & \quad \text{(by } R\text{)} \\ & w \in E.X \cap \mathsf{W} \wedge loc(a) = x \wedge tst(V.tview_t(x)) \leq tst(w) \\ \iff & \quad (28) \\ & w \in E.X \cap \mathsf{W} \wedge loc(a) = x \wedge \forall w' \in E.EW(t), x).(w, w') \notin E.\mathsf{mo} \\ \iff & w \in E.OW(t, x) \end{array}$$

as required.

In the preservation proof, we use the following stability properties:

Lemma 23. *For all $E \overset{w,e}{\Longrightarrow} E'$, every location x, and every thread $t' \neq tid(e)$*

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t'), x) = E.\mathbf{max}_{\mathsf{mo}}(E.EW(t'), x) \tag{29}$$

and for every write $w' \neq e$

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.\mathbf{cclose}(\{w'\}), x) = E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x) \tag{30}$$

Further, for all $V \overset{w,e}{\rightsquigarrow} V'$, every location x, and every thread $t' \neq tid(e)$

$$V'.tview(t', x) = V.tview(t', x) \tag{31}$$

and for every write $w' \neq e$

$$V'.mview(w', x) = V.mview(w', x) \tag{32}$$

Lemma 24 (Mod-order agreement). *For any View state*

$$V = (writes, tview, mview, covered)$$

and Explicit state $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ such that $(V, E) \in R$, and every thread t, write w and variable x

$$tst(V.tview(t)(x)) < tst(V.mview(w)(x)) \iff t \ \mathbf{before}_x \ w$$

Proof. We reason thusly

$$\begin{array}{rll} tst(V.tview(t)(x)) < tst(V.mview(w)(x)) & \iff & \\ tst(E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)) < tst(E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)) & \iff & \text{by 23} \\ t \ \mathbf{before}_x \ w & & \text{by 25} \end{array}$$

Lemma 25 (View Mod Order). *For any $V \overset{w,e}{\rightsquigarrow}_t V'$ where e is a write or update, $V'.\mathsf{mo} = V.\mathsf{mo}[w,(e,q,t)]$ where q is the fresh rational used to tag the operation in the View transition relation.*

Proof. The new write is added into mo immediately after the write w and before all subsequent writes to the same variable.

Lemma 26 (R-preservation). *For any View state*

$$V = (\mathit{writes}, \mathit{tview}, \mathit{mview}, \mathit{covered})$$

and Explicit state $E = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ such that $(V,E) \in R$, and every View state $V' = (\mathit{writes}', \mathit{tview}', \mathit{mview}', \mathit{covered}')$ such that $V \overset{w,e}{\rightsquigarrow} V'$, we have $E \overset{w,e}{\rightsquigarrow} E'$ for some E' and $(V',E') \in R$.

Proof. We proceed by cases on the type of the operation e and whether or not the operation is synchronising. In what follows, let $t = \mathit{tid}(e)$.

Case 1. e is of the form $rd(x,n)$. Let $E = (X', \mathsf{sb}', \mathsf{rf}', \mathsf{mo}')$ where

$$\begin{aligned}(X', \mathsf{sb}') &= (X, \mathsf{sb}) + e\\ \mathsf{rf}' &= \mathsf{rf} \cup \{(w,e)\}\\ \mathsf{mo}' &= \mathsf{mo}\end{aligned}$$

The precondition of the View transition ensures that $w \in V.\mathit{OW}(t,x)$, and thus by Lemma 22, we have $w \in E.\mathit{OW}(t)$. Thus, we have $E \overset{w,e}{\rightsquigarrow} E'$. It remains to show that $(V',E') \in R$, which we do by considering each of the equations in the definition of R in turn.

–

$$\begin{aligned}V'.\mathit{writes} &= V.\mathit{writes} && \text{by View trans-rel}\\ &= E.X \cap \mathsf{W} && \text{by } R\\ &= E'.X \cap \mathsf{W} && \text{by Explicit trans-rel}\end{aligned}$$

– We need to show that $V'.\mathit{tview}(t,x) = E'.\mathbf{max}_{\mathsf{mo}}(E'.\mathit{EW}(t),x)$. For all $t' \neq t$, it is easy to see that Equations 29 and 31 ensure that this property is preserved.
Several cases remain. We discuss the first two. The remaning cases follow in a similar way. In the first case, we assume the following

$$w, e \text{ synchronise} \tag{33}$$
$$\mathit{loc}(e) = x \tag{34}$$
$$\mathit{tst}(V.\mathit{tview}(t)(x)) < \mathit{tst}(V.\mathit{mview}(w)(x)) \tag{35}$$
$$(E.\mathbf{max}_{\mathsf{mo}}(E.\mathit{EW}(t),x), E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}),x)) \in E.\mathsf{mo} \tag{36}$$

Note that by 24 the last two assumptions are equivalent. Then,

$$\begin{aligned} V'.tview(t)(x) &= (V.tview(t) \otimes V.mview(w))(x) && \text{by View trans-rel} \\ &= V.mview(w)(x) && \text{by hyp. and def of } \otimes \\ &= E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x) && \text{by } R \\ &= E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) && \text{see below} \end{aligned}$$

The last step is a consequence of the first, second, and fourth assumptions together with Lemma 18.
In the second case, we assume

$$w, e \text{ synchronise} \tag{37}$$
$$loc(e) = x \tag{38}$$
$$tst(V.mview(w)(x)) < tst(V.tview(t)(x)) \tag{39}$$
$$(E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(\{w\}), x), E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)) \in E.\mathsf{mo} \tag{40}$$

(The new hyptothesis differs from the previous in that we have changed the view that supplies the latest write.) Note that by 24 the last two assumptions are equivalent. Then,

$$\begin{aligned} V'.tview(t)(x) &= (V.tview(t) \otimes V.mview(w))(x) && \text{by View trans-rel} \\ &= V.tview(t)(x) && \text{by hyp. and def of } \otimes \\ &= E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x) && \text{by } R \\ &= E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) && \text{see below} \end{aligned}$$

The last step is a consequence of the first, second, and fourth assumptions together with Lemma 18.

– We need to show that $V'.mview(w, x) = E'.\mathbf{max}_{\mathsf{mo}}(E'.\mathbf{cclose}(\{w\}), x)$ for all w. But because e is a read we have

$$\begin{aligned} V'.mview &= V.mview \\ E'.\mathbf{cclose} &= E.\mathbf{cclose} \\ E'.\mathbf{max}_{\mathsf{mo}} &= E.\mathbf{max}_{\mathsf{mo}} \end{aligned}$$

so the preservation result follows immediately.

–

$$\begin{aligned} V'.\mathsf{mo} &= V.\mathsf{mo} && \text{by View trans-rel} \\ &= E.\mathsf{mo} && \text{by } R \\ &= E'.\mathsf{mo} && \text{by Explicit trans-rel} \end{aligned}$$

–

$$\begin{aligned} V'.covered &= V.covered && \text{by View trans-rel} \\ &= E.covered && \text{by } R \\ &= E'.covered && \text{by Explicit trans-rel} \end{aligned}$$

Case 2. e is of the form $wr(x, n)$. Let $E = (X', \mathsf{sb}', \mathsf{rf}', \mathsf{mo}')$ where

$$\begin{aligned}(X', \mathsf{sb}') &= (X, \mathsf{sb}) + e\\ \mathsf{rf}' &= \mathsf{rf}\\ \mathsf{mo}' &= \mathsf{mo}[w, e]\end{aligned}$$

The precondition of the View transition ensures that $w \in V.OW(t, x)$, and thus by Lemma 22, we have $w \in E.OW(t)$. Thus, we have $E \overset{w,e}{\rightsquigarrow} E'$. It remains to show that $(V', E') \in R$, which we do by considering each of the equations in the definition of R in turn.

–

$$\begin{aligned}V'.writes &= V.writes \cup \{e\} && \text{by View trans-rel}\\ &= (E.X \cap \mathsf{W}) \cup \{e\} && \text{by } R\\ &= E'.X \cap \mathsf{W} && \text{by Explicit trans-rel}\end{aligned}$$

– If $loc(e) \neq x$ then the thread view and mo restricted to x are unchanged. If $loc(e) = x$, then $E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = e$ and $V'.tview(t, x) = e$ so

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.EW(t), x) = V'.tview(t, x) = e$$

as required.

– If $loc(e) \neq x$ then the thread view and mo restricted to x are unchanged. Assume $loc(e) = x$. For all $w' \in V'.writes$ where $w' \neq e$, then

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.\mathbf{cclose}(w), x) = E.\mathbf{max}_{\mathsf{mo}}(E.\mathbf{cclose}(w), x)$$

so the property is preserved. In the final case,

$$E'.\mathbf{max}_{\mathsf{mo}}(E'.\mathbf{cclose}(e), x) = E.\mathbf{max}_{\mathsf{mo}}(E.EW(t) \cup \{e\}) = e$$

but $V'.mview(e, x) = e$ as required.

–

$$\begin{aligned}V'.\mathsf{mo} &= V.\mathsf{mo}[w, e] && \text{by Lemma 25}\\ &= E.\mathsf{mo}[w, e] && \text{by } R\\ &= E'.\mathsf{mo} && \text{by Explicit trans-rel}\end{aligned}$$

–

$$\begin{aligned}V'.covered &= V.covered && \text{by View trans-rel}\\ &= E.covered && \text{by } R\\ &= E'.covered && \text{by Explicit trans-rel}\end{aligned}$$

Case 3. e is of the form $upd^{RA}(x, m, n)$. Let $E = (X', \mathsf{sb}', \mathsf{rf}', \mathsf{mo}')$ where

$$\begin{aligned}(X', \mathsf{sb}') &= (X, \mathsf{sb}) + e\\ \mathsf{rf}' &= \mathsf{rf} \cup \{(w, e)\}\\ \mathsf{mo}' &= \mathsf{mo}[w, e]\end{aligned}$$

The precondition of the View transition ensures that $w \in V.OW(t, x)$, and thus by Lemma 22, we have $w \in E.OW(t)$. Thus, we have $E \overset{w,e}{\rightsquigarrow} E'$. It remains to show that $(V', E') \in R$, which we do by considering each of the equations in the definition of R in turn.

The proof that $(V', E') \in R$ is essentially a combination of the proof for reads and writes.

–

$$\begin{aligned}V'.writes &= V.writes \cup \{e\} && \text{by View trans-rel}\\ &= (E.X \cap \mathsf{W}) \cup \{e\} && \text{by } R\\ &= E'.X \cap \mathsf{W} && \text{by Explicit trans-rel}\end{aligned}$$

– The proof here is the same as that for reads.
– The proof here is the same as that for writes.
–

$$\begin{aligned}V'.\mathsf{mo} &= V.\mathsf{mo}[w, e] && \text{by Lemma 25}\\ &= E.\mathsf{mo}[w, e] && \text{by } R\\ &= E'.\mathsf{mo} && \text{by Explicit trans-rel}\end{aligned}$$

–

$$\begin{aligned}V'.covered &= V.covered \cup \{w\} && \text{by View trans-rel}\\ &= E.covered \cup \{w\} && \text{by } R\\ &= E'.covered && \text{by Explicit trans-rel}\end{aligned}$$

Finally, we must prove that for every initial View state, there is an R-related initial Explicit state.

Lemma 27. *For every initial state of the View semantics V_0, there is an initial state of the Explicit semantics E_0 such that $(V_0, E_0) \in R$*

Proof. Let $V_0 = (writes, tview, mview, covered)$. We let $E_0 = (X, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$, where [8]

$$X = writes \tag{41}$$
$$\mathsf{sb} = \emptyset \tag{42}$$
$$\mathsf{rf} = \emptyset \tag{43}$$
$$\mathsf{mo} = \emptyset \tag{44}$$

We prove each property of the simulation relation in turn.

[8] Recall that in our setting *writes* and X have the same type.

- (22) This is immediate.
- (23) For all threads t and variables x, $E.\mathbf{max}_{\mathsf{mo}}(E.EW(t), x)$ is the initialising write to x, which is also the value of $V.tview(t, x)$.
- (24) Similarly to 23, for all writes w and variables x, $E.\mathbf{cclose}(w)$ and $V.mview(w)$ is the initialising write to x.
- (25) This is immediate.
- (26) This is immediate.