Owicki-Gries Reasoning for C11 RAR

Sadegh Dalvandi
University of Surrey, United Kingdom
m.dalvandi@surrey.ac.uk

Simon Doherty
University of Sheffield, United Kingdom
s.doherty@sheffield.ac.uk

Brijesh Dongol
University of Surrey, United Kingdom
b.dongol@surrey.ac.uk

Heike Wehrheim
Paderborn University, Germany
wehrheim@upb.de

Abstract

Owicki-Gries reasoning for concurrent programs uses Hoare logic together with an *interference freedom* rule for concurrency. In this paper, we develop a new proof calculus for the C11 RAR 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.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Hoare logic; Theory of computation → Concurrency; Theory of computation → Operational semantics; Theory of computation → Program reasoning

Keywords and phrases C11, Verification, Hoare logic, Owicki-Gries, Isabelle

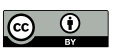
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.11

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.15`.

Funding *Sadegh Dalvandi*: Supported by EPSRC Grant EP/R032556/1.
Simon Doherty: Supported by EPSRC Grant EP/R032351/1.
Brijesh Dongol: Supported by EPSRC Grant EP/R032556/1.
Heike Wehrheim: Supported by DFG grant WE 2290/12-1.

1 Introduction

In 1976, Susan Owicki and David Gries proposed an extension of Hoare's axiomatic reasoning technique [15] to concurrent programs [25]. 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) [23], and is unsound for weak 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 [32, 16]). 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 strengthened 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 (RAR) 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 enables the specification of *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 C11 RAR. The semantics is a mixture of the operational semantics proposed by Doherty et al. [12] (for RAR) and Kaiser et al.'s semantics [16] for RA plus non-atomics. Correctness of this novel proposal is shown by proving it to coincide with the semantics defined by Doherty et al. [12] 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 [26] and mechanically proved soundness of all of our new rules for C11 assertions. Moreover, we provide mechanical proofs 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 [12, 34].

Overview. 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 C11 RAR 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 a case study of Peterson's algorithm in Section 6. Section 7 describes our Isabelle mechanisation, Section 8 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 (Figures 1 and 2). 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.

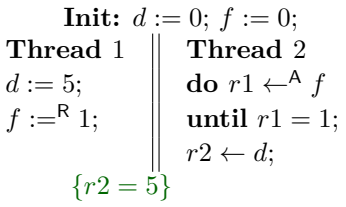


Figure 1 Message-passing litmus test.

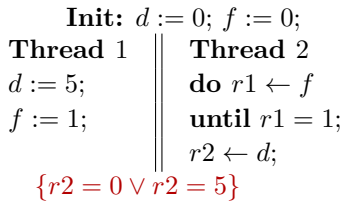


Figure 2 Unsynchronised message passing.

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 Figure 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 Figure 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 Figure 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 Figure 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* [16, 13, 17, 27], 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 [12] (C11 RAR) using timestamps, which has a restriction prohibiting the so-called *load-buffering* litmus test[1] [22].

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 Figure 3. As already noted, the key advantage of these assertions is the fact that 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*[2] of thread t. Note that there may be more than one such write, and hence

[1] Litmus tests are small code snippets with particularly interesting behaviour.
[2] We borrow the term viewfront from Popkadaev et al. [27].

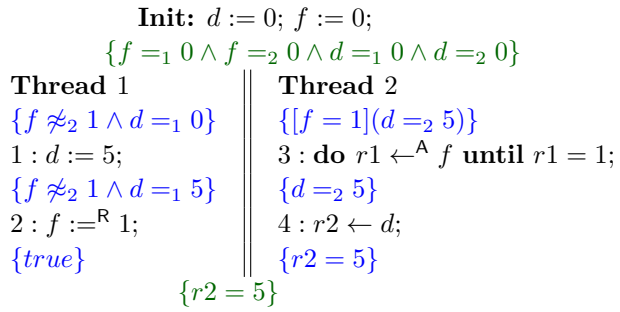


Figure 3 Proof outline for message passing.

there may be several possible values for a given variable. For instance, there might be one write to x with value v_1 in thread t's viewfront and two more writes to x with values v_2 and v_3 beyond the viewfront. Then assertions $x \approx_t v_1$, $x \approx_t v_2$ and $x \approx_t v_3$ all hold.

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 Figure 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., $f \not\approx_2 1$, needed for interference freedom of proof outlines) 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 *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 a 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., $\land$, $+$, $=$) 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 [12]. 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. We have not included a **CAS** operation here; it could similarly be implemented by an update event (see e.g. [33]).

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. Auxiliary variables may only occur in assignments, not in conditional statements, and only in the form $\alpha := E$, where $E \in Exp_L$ and α is an auxiliary variable[3]. Finally, we require that writes to auxiliary variables occur atomically in conjunction with another (non-auxiliary) atomic program step. Such atomic operations are written as $\langle A, \alpha := E \rangle$, where $A \in ACom$. This is more of a technical requirement which could also easily be relaxed. It guarantees that the programs without and with auxiliary variables have the same number of transitions (no stuttering steps).

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 variables. The structure of our programs thus is $\mathbf{Init}; (C_1 || \ldots || C_n)$.

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

[3] The locality requirement is the only difference to "normal" Owicki-Gries auxiliary variables.

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

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

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

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

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

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

Figure 4 Program semantics.

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 $lst \in Tid \to (Var_L \nrightarrow Val)$ ($\nrightarrow$ being a partial function), 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 $ls \in (Var_L \nrightarrow Val)$; 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 *any* possible value to be read. This is combined with the memory semantics (formalised by $\xrightarrow{a}_t$) as follows:

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

The transitions defined by $\sigma \xrightarrow{a}_t \sigma'$ 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.

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 viewfront of a thread t	$tview_{\mathbf{Init}}$
$mview_w \in Var_G \to writes$	The viewfront of a thread when writing w	$mview_{\mathbf{Init}}$
$covered \subseteq writes$	The covered writes	$\emptyset$

4.2 Memory Semantics

Next, we detail the memory semantics, which is equivalent to an earlier operational reformulation [12] of the RAR fragment from [22].

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., [16, 13, 27]). The timestamps totally order the writes to each variable; the ordering induced by timestamps is also referred to as the *modification order* [22, 12] or *coherence order* [6]. 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 $writes \subseteq \mathsf{W} \times \mathbb{Q}$.

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., [27, 17]).

- A function $tview_t$ that returns the *viewfront* of thread t (one for each global variable). 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*:

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

- 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 C11 RAR, each update action occurs in modification order immediately after the write that it reads from [12]. 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.

$$\begin{aligned} \mathit{writes}_{\mathbf{Init}} &= \{(wr(x_1,k_1),0),\ldots,(wr(x_n,k_n),0)\} \\ \mathit{tview}_{\mathbf{Init}}(x_i) &= (wr(x_i,k_i),0) \quad \text{for each thread } x_i \in \mathit{Var}_G \\ \mathit{mview}_{\mathbf{Init}} &= \mathit{tview}_{\mathbf{Init}} \end{aligned}$$

The initial local state component of each thread must also be compatible with **Init**, i.e., for each t if $r_i \in \mathbf{dom}(\mathit{lst}(t))$ we have that $(\mathit{lst}(t))(r_i) = l_i$ provided $r_i := l_i$ appears in **Init**.

We let $\mathit{lst}_{\mathbf{Init}}$ be the local state compatible with **Init**, let $\sigma_{\mathbf{Init}}$ denote the initial state defined by **Init**, and define $\Gamma_{\mathbf{Init}} = (\mathit{lst}_{\mathbf{Init}}, \sigma_{\mathbf{Init}})$.

Transition semantics. The transition relation of our semantics for global reads and writes is given in Figure 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,q) is a write to x that t can observe (i.e., $(w,q) \in \sigma.\mathit{OW}(t,x)$), and
- the value read by a is the value written by 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 $\mathit{tview}_t(x)$ or $\mathit{mview}_w(x)$, 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:

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

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

For illustration, consider the picture in Figure 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 of updates. We must choose a fresh timestamp $q' \in \mathbb{Q}$ for a, which is formalised by $\mathit{fresh}(q,q')$:

$$\sigma.\mathit{fresh}(q,q') = q < q' \wedge \forall w' \in \sigma.\mathit{writes}.\ q < \mathit{tst}(w') \Rightarrow q' < \mathit{tst}(w')$$

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)[4]. 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').

[4] This does not exclude that later some other write is placed in between q and q'.

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

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

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

Figure 5 Transition relation of the memory semantics.

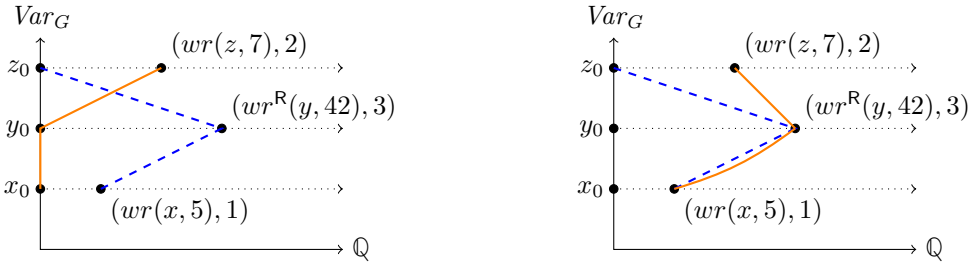


Figure 6 Illustration of views and view updates: pre-state (left) and post-state (right) after executing $rd^A(y,42)$ by thread t_1 (orange).

Finally, we set the viewfront of (a,q') to be the new viewfront of t, i.e., $mview_{(a,q')} := tview'_t$. 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 [12] that is already known to be equivalent to the C11 RAR fragment. Here, we just roughly sketch how this proof proceeds.

The semantics in [12] 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 $\sigma.writes$, we take $X \cap \mathsf{W}$; (2) For $\sigma.covered$, 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 $\mathsf{eco}^? \circ \mathsf{hb}^?$ (where $R^?$ is the reflexive closure of a relation R). Then $\sigma.tview_t = \mathbf{max}_{\mathsf{mo}}(X.\mathbf{cclose}(X_t))$ 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 [15] for C11 RAR that enables Owicki-Gries reasoning [25]. 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.

In Section 5.1, we link our operational semantics to the proof outlines of Hoare logic and Owicki-Gries' notion of interference freedom. Section 5.2 provides an overview of our assertion language and briefly discusses the main categories of assertions, i.e., assertions

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

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

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

Figure 7 Classical proof rules for sequential programs.

describing observability, ordering and occurrences of writes. We present the basic axioms in stages, using specific litmus tests (in Sections 5.3, 5.4, 5.5) to motivate each group of assertions. The proof outlines of all litmus tests have been verified using Isabelle.

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 $\Sigma_{C11} = (Var_L \to Val) \times \Sigma_G$ be the set of all possible C11 states. Predicates over Σ_{C11} are therefore of type $\Sigma_{C11} \to \mathbb{B}$. 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 $p, q \in \Sigma_{C11} \to \mathbb{B}$, $P \in Prog$ and $\boldsymbol{E} = \lambda t : Tid.\ \boldsymbol{skip}$. The semantics of a Hoare triple under partial correctness is given by:*

$$\begin{aligned}
\{p\}\mathbf{Init}\{q\} &= q(\Gamma_{\mathbf{Init}}) \\
\{p\}P\{q\} &= \forall lst, \sigma, lst', \sigma'.\ p(lst, \sigma) \wedge (P, lst, \sigma) \Longrightarrow^* (\boldsymbol{E}, lst', \sigma') \Rightarrow q(lst', \sigma') \\
\{p\}\mathbf{Init}; P\{q\} &= \exists r.\ \{p\}\mathbf{Init}\{r\} \wedge \{r\}P\{q\}
\end{aligned}$$

The classical rules of sequential Hoare logic for compound (i.e., non-atomic) statements are given in Figure 7. Soundness of these proof rules (with respect to Definition 2) holds for exactly the same reason as soundness of Hoare logic [15].

The sequential part is combined with the Owicki-Gries rule for concurrent composition in the standard way [25, 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*

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

Proof outlines of concurrent programs are interference free *if no statement in one thread interferes with an assertion in another thread.*

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

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

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

Finally, there is a standard proof rule for auxiliary variables in parallel programs [7]. Let V be a set of auxiliary variables of a parallel program P and q be a predicate that does not mention auxiliary variables. Then we can prove that a Hoare triple holds for a program extended with auxiliary variables and transfer this proof to the original program:

$$\textsc{AuxVar}\frac{\{\mathit{true}\}\ \mathbf{Init}; P\ \{q\}}{\{\mathit{true}\}\ \mathbf{Init}_0; P_0\ \{q\}}\text{provided } \mathit{vars}(q) \cap V = \emptyset$$

where $\mathbf{Init}_0$ is obtained from **Init** by removing all auxiliary assignments and P_0 is obtained by replacing all statements $\langle A, a := E\rangle$ in P (for $a \in V$) by A.

5.2 An Assertion Language

We studied a number of well-known litmus tests and examples and discovered three main categories of assertions required for specification and verification of a wide range of problems. These three main categories are dealing with (values of) writes to variables and the order in which they occur.

- **Observability.** Observability assertions describe if or when a thread may observe or has encountered a write to a variable. As described in Section 2, these assertions are thread-specific and deal with the thread's view. We repeat the main ideas here to simplify comparison with the other types of assertions. The main observability assertions are as follows:
  1. **Possible observation** which is denoted by $x \approx_t u$ means that thread t *may* observe value u for x. The formal definition and an example motivating this assertion is given in Section 5.4.
  2. **Definite observation** which is denoted by $x =_t u$ means that thread t *must* observe the value u for x. The formal definition and an example motivating this assertion is given in Section 5.3.
  3. **Conditional observation** which is denoted by $[x = u](y =_t v)$ means that if thread t synchronises with a write to variable x with value u, it *must* observe value v for y. The formal definition and an example motivating this assertion is given in Section 5.4.
  4. **Encountered value** which is denoted by $x \overset{enc}{=}_t v$ means that thread t has encountered (had the opportunity to observe) a write to variable x with value v. The formal definition and three examples motivating this assertion are given in Section 5.5.
- **Ordering.** Ordering assertions specify the order of values written to a variable by different writes. These assertions are thread-independent and specify an order over the timestamp of various writes with specific values:

1. **Possible value order** which is denoted by $m \prec_x n$ means that there exists two writes w and w' to variable x where the timestamp of w' is larger than the timestamp of w and the value of w and w' is m and n, respectively.
2. **Definite value order** which is denoted by $m \prec\!\!\prec_x n$ means that for all writes w and w' to x where the value of w is m and the value of w' is n, the timestamp of w' is larger than the timestamp of w and $m \prec_x n$.

Both the above assertions are formally defined in Section 5.5 and examples showing their usage are provided.

- **Occurrence.** Occurrence assertions specify the occurrence of a write with a specific value to a variable (regardless of observability). Similar to the previous category, these assertions are thread-independent:
  1. **Value occurrence** assertions specify the limit of occurrence of writes to a variable with a specific value. For instance, $\mathbb{0}_x\, n$ means that no write with value n to variable x has occurred or $\mathbb{1}_x\, n$ means that there is *at most* one write with value n to x in the current state. The formal definition and examples of these assertions are given in Section 5.5.
  2. **Initial value** which is denoted by $x_{\mathbf{Init}} = n$ means that the initial value written to x is n. The formal definition and examples of this assertion are also given in Section 5.5.
  3. **Covered write** assertions, denoted by $\mathbf{C}_x^n$, state that all writes to variable x except the last write are covered by an update (see Section 4.2), and that the last write to x has value n. This assertion is formally defined in Section 6 and is used in verification of Peterson's mutual exclusion algorithm.

5.3 Load Buffering

Our first example is the load buffering litmus test (see Figure 8), which we can show satisfies the postcondition $r1 = 0 \vee r2 = 0$ since our semantics assumes absence of cycles in the sequence-before relation combined with reads-from [22, 12]. The assertions about the C11 state capture properties about *definite observations* (i.e., observability assertions), 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. For a variable x, thread t and value n, we define:

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

Expanding this out, we obtain:

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

$$
\begin{array}{l}
\textbf{Init: } x := 0;\ y := 0;\ r1 := 0;\ r2 := 0; \\
\{x =_1 0 \wedge y =_2 0 \wedge r1 = 0 \wedge r2 = 0\}
\end{array}
$$

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

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

Figure 8 Proof outline for load buffering.

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 5.4.

The proof of load buffering relies on the basic axioms in the following lemma. We assume $atoms(\textbf{Init})$ 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.*

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

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

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 Figure 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. ◀

5.4 Message Passing

Next we return to the message passing example from Section 2. Its verification requires the usage of the other two observability assertions.

Possible observation. For a variable x, thread t and value n, we define:

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

Thus, there is a write to x that is observable to thread t with a value n.

Conditional observation. For variables x, y, thread t and values m, n, we define:

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

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

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

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

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

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

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

Proof. In Isabelle. ◀

▶ **Theorem 8.** *The proof outline of message passing in Figure 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. ◀

5.5 Read-Read Coherence

Next, we verify three versions of the read-read coherence (RRC) litmus test as given in Figures 9, 10 and 11. The original RRC litmus test (Figure 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. Figure 9 presents a simpler variation where the ordering of writes to x is enforced by the thread ordering. Figure 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 *ordering* and *occurrence* assertions which we introduce next.

Possible value order. For values m, n and variable x, we define:

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

Thus, there are two writes two x with values m and n, where the timestamp of the write with value m precedes the timestamp of the write with value n. 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.

$$
\begin{array}{l}
\textbf{Init: } x := 0;\ y := 0; \\
\{x_{\textbf{Init}} = 0 \land \mathbb{0}_x\, 1 \land \mathbb{0}_x\, 2\} \\
\textbf{Thread } 1 \quad \| \quad \textbf{Thread } 2 \\
\{x_{\textbf{Init}} = 0 \land \mathbb{0}_x\, 1 \land \mathbb{0}_x\, 2\} \quad \| \quad \{true\} \\
1 : x := 1; \quad \| \quad 3 : a \leftarrow x; \\
\{\mathbb{0}_x\, 2 \land \mathbb{1}_x\, 1 \land x \overset{enc}{=}_1 1\} \quad \| \quad \{x \overset{enc}{=}_2 a\} \\
2 : x := 2; \quad \| \quad 4 : b \leftarrow x; \\
\{1 \prec\!\!\!\prec_x 2\} \quad \| \quad \{a \neq b \Rightarrow a \prec_x b\} \\
\{a = 2 \Rightarrow b \neq 1\}
\end{array}
$$

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

Definite value order. For values m, n and variable x, we define:

$$
\begin{array}{rcl}
m \prec\!\!\!\prec_x n & = & \lambda\sigma.\ (m \prec_x n)(\sigma) \land (\forall w, w' \in \sigma.writes_x. \\
& & \quad wrval(w) = m \land wrval(w') = n \Rightarrow \\
& & \quad tst(w) < tst(w'))
\end{array}
$$

Note that this implies $m \neq n$. Unlike possible value orders if $m \prec\!\!\!\prec_x n$ holds then $n \not\prec\!\!\!\prec_x m$. Note also 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.

Initial value. For values n and variable x, we define:

$$
\begin{array}{rcl}
x_{\textbf{Init}} = n & = & \lambda\sigma.\ \exists w \in \sigma.writes_x.\ wrval(w) = n \land \\
& & \quad (\forall w' \in \sigma.writes_x.\ w \neq w' \Rightarrow tst(w) < tst(w'))
\end{array}
$$

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. For a variable x, thread t and value n, we define:

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

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. 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 occurrence. These are straightforward to define in terms of our value order assertions above. For a variable x, thread t and value n, we define:

$$
\begin{array}{rcl}
\mathbb{0}_x\, n & = & \exists m.\ x_{\textbf{Init}} = m \land m \neq n \land m \not\prec\!\!\!\prec_x n \\
\mathbb{1}_x\, n & = & n \not\prec\!\!\!\prec_x n
\end{array}
$$

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.

$$\textbf{Init: } x := 0;\ y := 0;$$
$$\{\mathbb{0}_x\, 1 \wedge \mathbb{0}_x\, 2 \wedge x_{\textbf{Init}} = 0\}$$

Thread 1	**Thread** 2	**Thread** 3	**Thread** 4
$\{\mathbb{0}_x\, 1\}$	$\{\mathbb{0}_x\, 2\}$	$\{true\}$	$\{true\}$
$1 : x := 1;$	$2 : x := 2;$	$3 : a \leftarrow x;$	$5 : c \leftarrow x;$
$\{\mathbb{1}_x\, 1\}$	$\{\mathbb{1}_x\, 2\}$	$\{x \overset{enc}{=}_3 a\}$	$\{x \overset{enc}{=}_4 c\}$
		$4 : b \leftarrow x;$	$6 : d \leftarrow x;$
		$\{a \neq b \Rightarrow a \prec_x b\}$	$\{c \neq d \Rightarrow c \prec_x d\}$

$$\{a = 1 \wedge b = 2 \wedge c = 2 \Rightarrow d \neq 1\}$$

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

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

$$m \prec_x n \wedge \mathbb{1}_x\, m \wedge \mathbb{1}_x\, n \Rightarrow m \preccurlyeq_x n \tag{6}$$
$$m \preccurlyeq_x n \Rightarrow n \not\preccurlyeq_x m \tag{7}$$

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.*

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

$$\textsc{1Intro}\ \frac{i \neq m}{\{x_{\textbf{Init}} = i \wedge \mathbb{0}_x\, m\}\ x :=_t^{[R]} m\ \{\mathbb{1}_x\, m\}} \qquad \textsc{EncWr}\ \frac{}{\{true\}\ x :=_t^{[R]} m\ \{x \overset{enc}{=}_t m\}}$$

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

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

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

Proof. In Isabelle. ◀

▶ **Theorem 11.** *The proof outline for RRC2 in Figure 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.

Init: $x := 0$; $y := 0$;
$\{\mathbb{0}_x\, 1 \wedge \mathbb{0}_x\, 2 \wedge x =_1 0 \wedge y \not\approx_2 1\}$

Thread 1	**Thread** 2	**Thread** 3
$\{\mathbb{0}_x\, 1 \wedge y \not\approx_2 1 \wedge (\mathbb{0}_x\, 2 \Rightarrow x =_1 0)\}$	$\{\mathbb{1}_x\, 1 \wedge \mathbb{0}_x\, 2 \wedge [y = 1](x =_2 1)\}$	$\{true\}$
$1 : x := 1;$	$3 : r \leftarrow^{\mathsf{A}} y;$	$5 : a \leftarrow x;$
$\{\mathbb{1}_x\, 1 \wedge y \not\approx_2 1 \wedge (\mathbb{0}_x\, 2 \Rightarrow x =_1 1)\}$	$\{\mathbb{1}_x\, 1 \wedge \mathbb{0}_x\, 2 \wedge (r = 1 \Rightarrow x =_2 1)\}$	$\{x \overset{enc}{=}_3 a\}$
$2 : y :=^{\mathsf{R}} 1;$	$4 : x := 2;$	$6 : b \leftarrow x;$
$\{\mathbb{1}_x\, 1\}$	$\{\mathbb{1}_x\, 2 \wedge (r = 1 \Rightarrow 1 \prec\!\!\!\prec_x 2)\}$	$\{a \neq b \Rightarrow a \prec_x b\}$

$\{r = 1 \wedge a = 2 \Rightarrow b \neq 1\}$

Figure 11 Proof outline for RRC3, where $x, y \in Var_G$ and $a, b \in Var_L$.

- 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 Figure 10 and Figure 11, respectively are valid.*

Proof. In Isabelle. ◀

For RRC (Figure 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.

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

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 (Figure 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.

6 Case study: 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[5], presented in Figure 12 is a mutual exclusion algorithm for two threads implemented for C11 using release-acquire annotations [34]. The purpose of verification is to show that this algorithm actually guarantees mutual exclusion, i.e., that the two threads can never be in their critical sections (line 6) at the same time. 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 C11 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.

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.

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

Init: $flag_1 := 0; flag_2 := 0; turn := 0 \wedge after_1 := false; after_2 := false$

Thread 1

$\left\{\begin{array}{l}\neg after_1 \wedge flag_1 =_1 0 \wedge turn \not\approx_2 2 \wedge (\mathbf{C}^0_{turn} \vee [turn = 1](flag_2 =_1 1))\\ \wedge (after_2 \Rightarrow \mathbf{C}^1_{turn} \wedge [turn = 1](flag_2 =_1 1))\end{array}\right\}$

1: $flag_1 := 1$;

$\left\{\neg after_1 \wedge flag_1 =_1 1 \wedge turn \not\approx_2 2 \wedge (after_2 \Rightarrow \mathbf{C}^1_{turn} \wedge [turn = 1](flag_2 =_1 1))\right\}$

2: $\langle turn.\mathbf{swap}(2)^{\mathsf{RA}}$; $after_1 := true\rangle$

$\left\{after_1 \wedge (after_2 \wedge (flag_2 \approx_1 0 \vee turn \approx_1 1) \Rightarrow turn =_2 1)\right\}$

do

3: $r_1 \leftarrow^{\mathsf{A}} flag_2$

$\left\{after_1 \wedge (after_2 \wedge (r_1 = 0 \vee turn \approx_1 1 \vee flag_2 \approx_1 0) \Rightarrow turn =_2 1)\right\}$

4: $r_2 \leftarrow turn$

$\left\{after_1 \wedge (after_2 \wedge (r_1 = 0 \vee r_2 = 1 \vee turn \approx_1 1 \vee flag_2 \approx_1 0) \Rightarrow turn =_2 1)\right\}$

5: **until** $(r_1 = 0 \vee r_2 = 1)$

$\{after_1 \wedge (after_2 \Rightarrow turn =_2 1)\}$

6: Critical section ;

7: $\langle flag_1 :=^{\mathsf{R}} 0$; $after_1 :=false\rangle$

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

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

$$after_1 \wedge (after_2 \Rightarrow turn =_2 1) \wedge after_2 \wedge (after_1 \Rightarrow turn =_1 2)$$

It is easy to see that this implies $turn =_1 2 \wedge turn =_2 1$. 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 $flag_2 \approx_1 0$) or it reads 1 from *turn* (which is only possible when $turn \approx_1 1$). 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 $turn =_2 1$ 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 Figure 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 C11 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. To formally state this, we need the third *occurrence* assertion $\mathbf{C}^n_x$, defined as follows.

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

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

We use the following lemma on covered.

▶ **Lemma 13.**

$$\textit{CVD-Upd}\ \frac{}{\{\boldsymbol{C}_x^n\}\ x.\mathsf{swap}(l)^{RA}\ \{\boldsymbol{C}_x^l\}} \qquad \textit{CVD-Wr}\ \frac{x \neq y}{\{\boldsymbol{C}_x^n\}\ y :=^{[R]} m\ \{\boldsymbol{C}_x^n\}}$$

$$\textit{CVD-Rd}\ \frac{}{\{\boldsymbol{C}_x^n\}\ r \leftarrow^{[A]} y\ \{\boldsymbol{C}_x^n\}} \qquad \textit{CVD-DObs}\ \frac{}{\{\boldsymbol{C}_x^n\}\ x.\mathsf{swap}(l)^{RA}\ \{x =_t l\}}$$

Rule CVD-Upd states that if $\mathbf{C}_x^n$ holds in the pre-state, then after executing $x.\mathsf{swap}(l)^{\mathsf{RA}}$, we obtain a new covered predicate $\mathbf{C}_x^l$. Thus, it is possible to maintain a covered predicate in a program (with possibly different return values) by ensuring each modifiation to the covered variable is via a swap. This is a property that is true of Peterson's algorithm as given in Figure 12. Rules CVD-Wr and CVD-Rd give preservation properties for the covered assertion for a read and a write, respectively. Finally, CVD-DObs is used to establish a definite observation of a covered assertion after a swap command.

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, $\mathbf{C}_{\mathit{turn}}^1$) and when this RMW occurs then thread 1 will definitely see flag_2 set (that is, $[\mathit{turn} = 1](\mathit{flag}_2 =_1 1)$). This is enough to show that if after_2 then in the post-state of the RMW, $\mathit{flag}_2 \not=_1 0$ which is sufficent to prove the postcondition 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 14.** *The proof outline of Peterson's algorithm (Figure 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 Figure 12, and discovered simplifications to our original pen-and-paper proofs.

7 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 Figure 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 them to prove the corollaries. For example, `d_obs_WrX_set` in Figure 13 is verified using "`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 [10].

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 relation over these records with program

```
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'"
```

Figure 13 Isabelle encoding of basic axioms over C11 assertions.

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 Figure 13) to discharge proofs automatically.

8 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 [13] and is similar to the semantics of Kaiser et al. [16, 17]. We note there are differences in coverage of the memory models in [13, 16, 17]. Dolan et al. [13] cover a sequentially consistent (SC) and relaxed accesses for OCAML, where the SC operations behave like Java volatiles. Kaiser et al [16] covers non-atomics and release-acquire, while Kang et al. [17] 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* [1]. A number of works target *model checking* for weak memory, e.g., by explicitly encoding architectural structures leading to weak behaviour, like store buffers [31, 4]. Ponce de León et al. [28, 14] 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 [5] (for TSO), Nidhugg [2] (for TSO and PSO), RCMC [18] (for C11) and GenMC [19] (again, parametric in memory model).

A (non-automatic) reasoning technique for proving invariants – parameterised by a weak memory model – has been proposed by Alglave and Cousot [3]. 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 [25] (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." [3]. Our method in contrast only requires the initialisation, sequential, and non-interference proofs as with the original technique.

Another manual method for the RC11 memory model has been developed by Doherty et al. [12], who cover the message passing example and Peterson's algorithm. Our work is inspired by this existing work, however, there are several differences. They use a classical model of the C11 state (expressed in terms of a set of relations, e.g., reads-from, sequenced-before etc), develop assertions over these relations and a small proof calculus for these assertions. Moreover, their methods are at a lower level of abstraction than the techniques presented in this paper since the assertions are stated in terms of individual relations that make up each state. Thus, it is not possible to directly develop a Hoare logic for their assertions and mechanisation itself is more difficult.

Also close to our work is that of Lahav and Vafeiadis [21] who also develop an Owicki-Gries style proof calculus. We consider all their examples except RCU – our logic can handle the RCU example, but this proof has thus far not been mechanised. Moreover, we include several other case studies such as litmus tests that combine read-read coherence with message passing and the non-trivial Peterson's algorithm. There are several additional differences to note. (1) Lahav and Vafeiadis' proof calculus is developed in the absence of an operational semantics, and hence, their definition of a valid Hoare triple is non standard (see [21, Definition 9]). A consequence of this is that they must be careful about the introduction of auxiliary variables, resorting to the more restricted notion of a *ghost* variable. In contrast, we use traditional auxiliary variables – an auxiliary variable must not affect the control flow of a program nor be assigned to any program variable. Note however, that to simplify the presentation, we use auxiliary variables in a more restricted manner (see Section 3). (2) They do not handle relaxed accesses – as stated in their conclusion: "While OGRA's non-interference condition appears to be restrictive, it is unsound for weaker memory models, such as C11's relaxed accesses ...". (3) They do not provide a mechanisation.

A frequently employed starting point for program logic is separation logic, for which a number of extensions to weak memory exist (GPS [32], RSL [16]). Svendsen et al. [30] propose a separation logic based on the promising semantics of Kang et al. [17]. 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 developed for the Iris proof system [16]. Tool support has also been developed by Summers and Müller [29], where the RSL logic has been encoded in the Viper tool, offering a level of proof automation. Their encoding is proved sound and complete with respect to RSL. However, such efforts do not provide a clear link between C11 semantics and traditional reasoning using Hoare logics.

9 Conclusion

In this paper, we have introduced an assertion language for C11 RAR which enables re-use of the entire Owicki-Gries proof calculus except for the axiom of assignment. The assertion language is based on an operational semantics for C11 RAR which we have shown to be sound wrt. standard axiomatic semantics. We have exemplified reasoning on a number of standard C11 RAR litmus tests as well as a C11 RAR 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.

We are currently integrating this work [11] into the standard Owicki-Gries library that is included in the Isabelle distribution [24]. As future work, we aim to tackle fragments of C11 larger than C11 RAR, e.g., fragments that allow the load buffering example to terminate with postcondition $r1 = 1 \wedge r2 = 1$ [8, 17], SC annotations [22], as well as release sequences and fences [9]. Extending our operational semantics to handle the final two features is straightforward, but is not considered in this paper as it complicates the semantics and detracts from our main contribution, i.e., a simple extension to Hoare logic to enable reasoning about C11 programs. Hoare-style reasoning that incorporates the other two features is currently being investigated.

References

1 P. A. Abdulla, J. Arora, M. F. Atig, and S. N. Krishna. Verification of programs under the release-acquire semantics. In K. S. McKinley and K. Fisher, editors, *PLDI*, pages 1117–1132. ACM, 2019.

2 P. Aziz Abdulla, S. Aronis, M. Faouzi Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. *Acta Inf.*, 54(8):789–818, 2017. doi:10.1007/s00236-016-0275-0.

3 J. Alglave and P. Cousot. Ogre and Pythia: an invariance proof method for weak consistency models. In G. Castagna and A. D. Gordon, editors, *POPL*, pages 3–18. ACM, 2017.

4 J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 512–532. Springer, 2013. doi:10.1007/978-3-642-37036-6_28.

5 J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013. doi:10.1007/978-3-642-39799-8_9.

6 J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

7 K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009. doi:10.1007/978-1-84882-745-5.

8 M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

9 M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *POPL*, pages 55–66. ACM, 2011. doi:10.1145/1926385.1926394.

10 S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

11 S. Dalvandi, B. Dongol, and S. Doherty. Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. *CoRR*, abs/2004.02983, 2020. arXiv:2004.02983.

12 S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. Verifying C11 programs operationally. In J. K. Hollingsworth and I. Keidar, editors, *PPoPP*, pages 355–365. ACM, 2019. doi:10.1145/3293883.3295702.

13 S. Dolan, K. C. Sivaramakrishnan, and A. Madhavapeddy. Bounding data races in space and time. In *PLDI*, PLDI 2018, pages 242–255, New York, NY, USA, 2018. ACM.

14 N. Gavrilenko, H. Ponce de Le'on, F. Furbach, K. Heljanko, and R. Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In I. Dillig and S. Tasiran, editors, *CAV*, volume 11561 of *LNCS*, pages 355–365. Springer, 2019. doi:10.1007/978-3-030-25540-4_19.

15 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

16 J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In P. Müller, editor, *ECOOP*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.17.

17 J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In G. Castagna and A. D. Gordon, editors, *POPL*, pages 175–189. ACM, 2017.

18 M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018. doi:10.1145/3158105.

19 M. Kokologiannakis, A. Raad, and V. Vafeiadis. Model checking for weakly consistent libraries. In K. S. McKinley and K. Fisher, editors, *PLDI*, pages 96–110. ACM, 2019.

20 O. Lahav. Verification under causally consistent shared memory. *SIGLOG News*, 6(2):43–56, 2019. doi:10.1145/3326938.3326942.

21 O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *ICALP*, volume 9135 of *LNCS*, pages 311–323. Springer, 2015. doi:10.1007/978-3-662-47666-6_25.

22 O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, pages 618–632. ACM, 2017.

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

24 T. Nipkow and L. P. Nieto. Owicki/Gries in Isabelle/HOL. In *FASE*, volume 1577 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 1999.

25 S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976. doi:10.1007/BF00268134.

26 L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *LNCS*. Springer, 1994. doi:10.1007/BFb0030541.

27 A. Podkopaev, I. Sergey, and A. Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016. arXiv:1606.01400.

28 H. Ponce de León, F. Furbach, K. Heljanko, and R. Meyer. BMC with memory models as modules. In N. Bjørner and A. Gurfinkel, editors, *FMCAD*, pages 1–9. IEEE, 2018. doi:10.23919/FMCAD.2018.8603021.

29 A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In D. Beyer and M. Huisman, editors, *TACAS*, volume 10805 of *LNCS*, pages 190–209. Springer, 2018. doi:10.1007/978-3-319-89960-2_11.

30 K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. A separation logic for a promising semantics. In A. Ahmed, editor, *ESOP*, volume 10801 of *LNCS*, pages 357–384. Springer, 2018. doi:10.1007/978-3-319-89884-1_13.

31 O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013. doi:10.1007/978-3-319-03077-7_21.

32 A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In A. P. Black and T. D. Millstein, editors, *OOPSLA*, pages 691–707. ACM, 2014. doi:10.1145/2660193.2660243.

33 J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In G. Castagna and A. D. Gordon, editors, *POPL*, pages 190–204. ACM, 2017.

34 A. Williams. https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html, 2018. Accessed: 2018-06-20.