
Article:
Dalvandi, S., Dongol, B., Doherty, S. et al. (1 more author) (2022) Integrating Owicki–Gries for C11-style memory models into Isabelle/HOL. Journal of Automated Reasoning, 66 (1). pp. 141-171. ISSN 0168-7433

https://doi.org/10.1007/s10817-021-09610-2

Journal of Automated Reasoning (2022) 66:141–171
https://doi.org/10.1007/s10817-021-09610-2

Integrating Owicki–Gries for C11-Style Memory Models into Isabelle/HOL

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

Received: 11 September 2020 / Accepted: 27 September 2021 / Published online: 16 November 2021

Abstract
Weak memory presents a new challenge for program verification and has resulted in the development of a variety of specialised logics. For C11-style memory models, our previous work has shown that it is possible to extend Hoare logic and Owicki–Gries reasoning to verify correctness of weak memory programs. The technique introduces a set of high-level assertions over C11 states together with a set of basic Hoare-style axioms over atomic weak memory statements (e.g. reads/writes), but retains all other standard proof obligations for compound statements. This paper takes this line of work further by introducing the first deductive verification environment in Isabelle/HOL for C11-like weak memory programs. This verification environment is built on the Nipkow and Nieto's encoding of Owicki–Gries in the Isabelle theorem prover. We exemplify our techniques over several litmus tests from the literature and two non-trivial examples: Peterson's algorithm and a read–copy–update algorithm adapted for C11. For the examples we consider, the proof outlines can be automatically discharged using the existing Isabelle tactics developed by Nipkow and Nieto. The benefit here is that programs can be written using a familiar pseudocode syntax with assertions embedded directly into the program.

Dalvandi and Dongol are supported by EPSRC grant EP/R032556/1. Dongol is further supported by EPSRC grants EP/V038915/1 and EP/R025134/2, VeTSS, and ARC Discovery Grant DP190102142. Doherty is supported by EPSRC Grant EP/R032351/1. Wehrheim is supported by DFG project WE2290/14-1.

✉ Brijesh Dongol
b.dongol@surrey.ac.uk

Sadegh Dalvandi
m.dalvandi@surrey.ac.uk

Simon Doherty
s.doherty@sheffield.ac.uk

Heike Wehrheim
heike.wehrheim@uni-oldenburg.de

[1] University of Surrey, Guildford, UK

[2] University of Sheffield, Sheffield, UK

[3] University of Oldenburg, Oldenburg, Germany

Keywords C11 · Hoare logic · Owicki–Gries · Isabelle/HOL · Weak memory · Deductive verification

1 Introduction

Hoare logic [19] is fundamental to understanding the intended design and semantics of sequential programs. Owicki and Gries' [31] framework extends Hoare logic to a concurrent setting by adding an interference-free check that guarantees stability of assertions in one thread against the execution of another. Although several other techniques for reasoning about concurrent programs have since been developed [12], Owicki–Gries reasoning remains fundamental to understanding concurrent systems and one of the main methods for performing deductive verification. Mechanised support for Owicki–Gries' framework has been developed for the Isabelle theorem prover [32] for programs under sequentially consistent memory model by Nipkow and Nieto [30] and is currently included in the standard distribution. This mechanisation provides a simple WHILE-language for writing multi-threaded programs and allows program commands to be annotated with assertions. It is also equipped with an automatic verification condition generator that creates all the standard Owicki–Gries local and interference freedom proof obligations.

Our work is in the context of C11 (the 2011 C standard), which has a weak memory model that is designed to enable programmers to take advantage of weak memory hardware [6,22,25,27]. Unlike sequentially consistent memory [28], states are represented by graphs with several relations (e.g. reads-from, modification-order, sequenced-before) that are used to track dependencies between memory events (e.g. reads, writes, updates). Declarative (or axiomatic) semantics [2,6,25,27] consider states corresponding to complete executions and define axioms that describe whether the state is valid for the given memory model. Operational semantics build state (graphs) in a stepwise manner [14], where each action introduces a new event as well as any necessary relations into the pre-state. The complexity of weak memory states means that it has not been possible to use the traditional Owicki–Gries framework to reason about concurrent programs under C11. Researchers have instead developed a set of specialised logics, e.g. [2], including those that extend Owicki–Gries framework [26] and separation logic [15,16,37,37,39] designed to cope with specific fragments of C11.

Our point of departure is the operational semantics of Doherty et al. [14] for the RC11-RAR fragment of C11 [27]. As indicated by the *RAR*, the memory model allows both relaxed and release-acquire accesses. Moreover, the model restricts the C11 memory model to disallow the "load-buffering" litmus test[1] [25,27]. A key advancement in the semantics developed by Doherty et al. is a transition relation over states modelled as C11 graphs, allowing program execution to be viewed as an interleaving of program statements as in classical approaches to concurrency. They provide a primitive assertion language for expressing properties of such states, which is manually applied to the message passing litmus test and Peterson's algorithm adapted to C11. However, the assertion language itself expresses state properties at a low level of abstraction (high level of detail), and hence is difficult to mechanise. We have recently recast Doherty et al.'s semantics in an equivalent timestamp-based semantics [9,21,22]. More importantly, we have developed a high-level set of assertions for stating properties of the C11 state [9]. These assertions have been shown to integrate well with a Hoare-style proof calculus, and, by extension, the Owicki–Gries proof method. Interestingly, the technique enables reuse of all standard Owicki–Gries proof rules for compound statements.

[1] Litmus tests are small code snippets with particularly interesting behaviour.

In this paper, we push this technique further by introducing the first deductive verification environment for C11-like weak memory programs in Isabelle/HOL. This environment is built on the Owicki–Gries encoding by Nipkow and Nieto [30]. Unlike [9], where program counters are used to model control flow and relations over C11 states are used to model program transitions, the approach in this paper is more direct. We show that once a correct proof outline has been encoded, the proof outlines can be validated with minimal user interaction. Our extension is parametric in the memory model, and can be adapted to reason about other C11-style operational models [25].

Contributions This work extends the contributions of our previous work [9] considerably. Our main contributions are thus:

1. A generic extension to the standard Isabelle/HOL encoding of Owicki–Gries to cope with C11-style weak memory,
2. An instantiation of the RC11-RAR operational semantics within Isabelle/HOL as an example memory model,
3. An integration with a high-level assertion language for reasoning about weak memory states, and
4. Verification of several examples in the extended theory in Isabelle/HOL, including two new large case studies: the read–copy–update (RCU) algorithm and a two-way message passing algorithm for C11.

Overview In Sect. 3, we briefly present the Owicki–Gries encoding by Nipkow and Nieto [30], as well as the message passing litmus test which serves as a running example. We describe how this encoding can be generically extended to cope with weak memory in Sect. 4, and present RC11-RAR as an example instantiation. In Sect. 5, we present a technique for reasoning about C11-style programs as encoded in Isabelle[2], which we apply to a number of examples. Further case studies are presented in Sect. 6, and we evaluate our proof strategy in Sect. 7. We present related work in Sect. 8.

2 A C11-Style Memory Model: RC11-RAR

In this section, we first describe a particular instance of a C11-style memory model that we work with in this paper, namely the RC11-RAR fragment, through an example (Sect. 2.1). This fragment disallows the load buffering litmus test [6,25,27], and all accesses are either relaxed, releasing or acquiring. It is straightforward to extend the model to incorporate more sophisticated notions such as release sequences and non-atomic accesses, but these are not considered as the complications they induce detract from the main contribution of our work. It is worth noting that RC11-RAR is still a non-trivial fragment [14]. We then briefly discuss our approach to deductive reasoning for weak memory in Sect. 2.2.

2.1 Message Passing

To motivate the memory model, we look at a simple message passing (MP) algorithm. First, we consider a version of the algorithm under sequential consistency (Fig. 1). It comprises two shared variables: d (that stores some data) and f (that stores a flag), both of which are initially 0. Under sequential consistency, the postcondition of the program is $r2 = 5$. This is because the loop in thread 2 only terminates after f has been updated to 1 in thread 1, which

[2] Our Isabelle/HOL formalisation is available here [10]. Note that the development requires Isabelle 2020.

Fig. 1 MP under sequential consistency

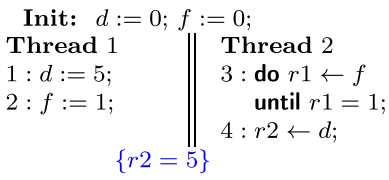


Fig. 2 Unsynchronised MP under RC11-RAR

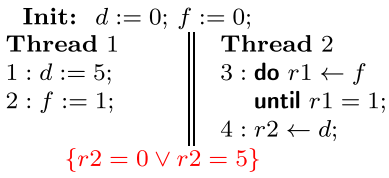
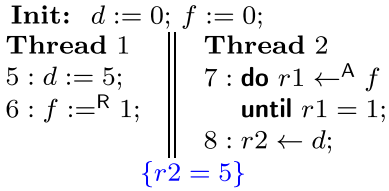


Fig. 3 MP with release-acquire synchronisation



in turn happens after d has been set to 5 by thread 1. Therefore, the only possible value of d for thread 2 to read is 5. The proof of this property is straightforward, and can be easily handled by Nipkow and Nieto's encoding [30].

Now, we consider again the MP example but for RC11-RAR (Figs. 2, 3). In Fig. 2, all accesses are relaxed, and hence the program can only establish the weaker postcondition $r2 = 0 \vee r2 = 5$ since it is possible for thread 2 to read 0 for d at line 4. In Fig. 3, the release annotation (line 2) and the acquire annotation (line 3) induces a *happens-before* relation if the read of f reads from the write at line 2 [6]. This in turn ensures that thread 2 sees the most recent write to d at line 5.

We use the operational semantics described in [9], which models the weak memory state using *timestamped writes* and *thread viewfronts* [17,21,22,34]. A timestamp is a rational number that *totally orders* the writes to each variable. A thread *viewfront*[3] records the *timestamp* that a thread has encountered for each variable — the idea is that a thread may read from *any* write whose timestamp is no smaller than the thread's current viewfront. Similarly, a write may be introduced at any timestamp greater than the current viewfront. The only caveat when introducing a write is that it may not be introduced directly after a write (in the modification order) that was previously read by a read–modify–write (RMW) operation. We refer to a write that was previously read from by a RMW operation as *covered write* (see [9,14]). This caveat is to ensure atomicity of RMW operations. In particular, a write to a variable x is covered whenever there is a RMW on x that reads from the write. In this instance, it would be unsound for another write to x to be introduced between the write that is read and the RMW (see [14] for further details).

Example 1 (Unsynchronised MP) Consider Fig. 4, depicting a possible execution of the unsynchronised MP example (Fig. 2). The execution comprises four weak memory states σ_0, σ_1, σ_2, σ_3. In each state, the timestamps themselves are omitted, but are assumed to be increasing in the direction of the arrows. The numbers depict the value of each variable at

[3] We borrow the term viewfront from Popkadaev et al. [34].

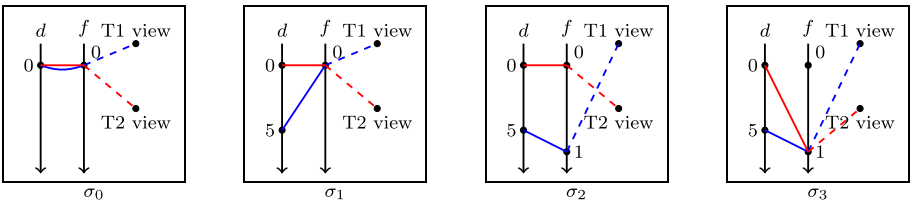


Fig. 4 An execution of the unsynchronised message passing

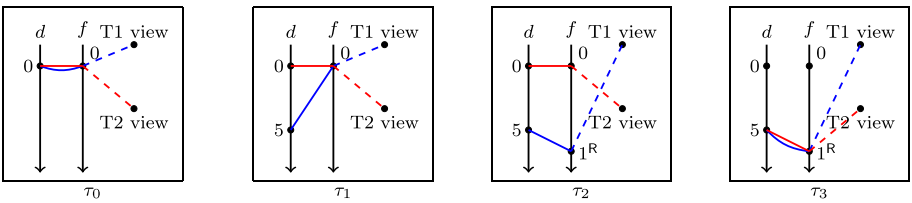


Fig. 5 An execution of the synchronised message passing

each timestamp. State σ_0 is the initial state. Each thread's viewfront in σ_0 is consistent with the initial writes.

After executing line 1, the program transitions to σ_1, which introduces a new write (with value 5) to d and updates the viewfront of thread 1 to the timestamp of this write. At this stage, thread 2's viewfront for d is still at the write with value 0. Thus, if thread 2 were to read from d, it would be permitted to return either 0 or 5. Moreover, if thread 2 were to write to d, it would be permitted to insert the write after 0 or 5.

After executing line 2, the program transitions to σ_2, which installs a (relaxed) write of f with value 1. Now, consider the execution of line 3. There are two possible poststates since there are two possible values of f that thread 2 could read. State σ_3 depicts the case where thread 2 reads from the new write $f = 1$. In this case, the view front of thread 2 is updated, but crucially, since there is no release-acquire synchronisation, the viewfront of thread 2 for d remains unchanged. This means that when thread 2 later reads from d in line 4, it may return either 0 or 5. We contrast this with the execution of the synchronised MP described in Example 2.

Example 2 (Synchronised MP) Consider Fig. 5, which depicts an execution of the program in Fig. 3. State τ_1 is a result of executing line 5 and is identical to σ_1. However, after execution of line 6, we obtain state τ_2, which installs a releasing write to f (denoted by 1^R). As in Example 1, the acquiring read in line 7 could read from either of the writes to f. State τ_3 depicts the case in which thread 2 reads from the releasing write 1^R. Now, unlike Example 2, this read establishes a release-acquire synchronisation, which means that the viewfront of thread 2 for *both* f and d are updated. Thus, if the execution continues so that thread 2 reads from d (line 8), the only possible value it may return is 5.

2.2 Deductive Reasoning for Weak Memory

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,

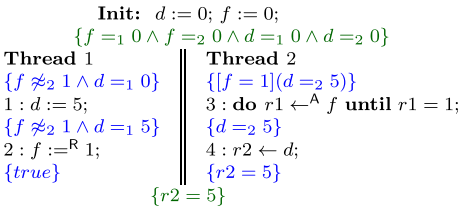


Fig. 6 Proof outline for message passing

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.

This intuition is captured formally using a semantics based on *timestamps* [17,21,22,34], which enables one to encode each thread's view and define how these views are updated. In [9], we characterise the release-acquire-relaxed subset of C11 [14] (C11 RAR) using timestamps, which has a restriction prohibiting the so-called *load-buffering* litmus test [27].

In [9], we also provide 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. 6. As already noted, the key advantage of these assertions is the fact that standard rules of Hoare and Owicki–Gries logic remain unchanged. To verify message passing, we require three main types of assertions:

- *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* 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. 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 Fig. 6, 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 $f :=^R 1$ 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.

```
datatype α ann_com =
AnnBasic (α assn) (α ⇒ α)
| AnnSeq (α ann_com) (α ann_com)
      ("_ ;; _")
| AnnCond (α assn) (α bexp) (α ann_com) (α ann_com)
      ("_ IF _ THEN _ ELSE _ FI")
| AnnWhile (α assn) (α bexp) (α assn) (α ann_com)
      ("_ WHILE _ INV _ DO _ OD")
| AnnAwait (α assn) (α bexp) (α com)
      ("_ AWAIT _ THEN _ END")
record mp_state = d::nat  f::nat  r1::nat  r2::nat
lemma message_passing:
||-
{| f = 0 ∧ d = 0 |}
  COBEGIN
    {| True |}                                  {| f=1 → d=5 |}
    d := 5 ;;                                   WHILE f≠1
    {| d = 5 |}                                 INV {| f=1 → d=5 |}
    f := 1                          ||          DO SKIP OD ;;
    {| True |}                                  {| d = 5 |}
                                                r2 := d
                                                {| r2 = 5 |}
  COEND
{| r2 = 5 |}
```

Fig. 7 Proving MP under sequential consistency using Nipkow and Nieto's encoding [30] of Owicki–Gries

3 Owicki–Gries in Isabelle/HOL

Nipkow and Nieto [30] present a formalisation of Owicki–Gries method in Isabelle/HOL. Their formalisation defines syntax, its semantics and Owicki–Gries proof rules in higher-order logic. Correctness of the proof rules with respect to the semantics is proved and their formalisation is part of the standard Isabelle/HOL libraries. To provide some context for our extensions, we provide an overview of this encoding here; an interested reader may wish to consult the original paper [30] for further details.

The defined programming language is a C-like `WHILE` language augmented with shared-variable parallelism (||) and synchronisation (`AWAIT`). Parallelism must not be nested, i.e. within c_1 || c_2 || ... || c_n, each c_i must be a sequential program. The programming language allows program constructs to be annotated with assertions in order to record proof outlines that can later be checked. The language also allows unannotated commands that may be placed within the body of `AWAIT` statements. As in the original treatment [31], `AWAIT` is an atomic command that executes under the precondition of the `AWAIT` block.

```
datatype α com =
Parallel (α ann_com option × α assn) list
| Basic (α ⇒ α)
| Seq (α com) (α com) ("_ ;; _")
| Cond (α bexp) (α com) (α com) ("IF _ THEN _ ELSE _ FI")
| While (α bexp) (α assn) (α com) ("WHILE _ INV _ DO _ OD")
```

In the datatype above, the concrete syntax is defined within `(" ... ")`. α `assn` and α `bexp` represent assertions and Boolean expressions, respectively. `AnnBasic` represents a basic (atomic) state transformation (e.g. an assignment). `AnnSeq` is sequential composition, `AnnCond` is conditional, `AnnWhile` is a loop annotated with an invariant, and `AnnWait` is a synchronisation

construct. The command `Parallel` is a list of pairs (c, q) where c is an annotated (sequential) command and q is a post-condition. The concrete syntax for parallel composition (not shown above) is: `COBEGIN` c_1 $\{\!|q_1|\!\}$ || ... || c_n $\{\!|q_n|\!\}$ `COEND`.

The semantics of programs are defined by transition rules between configurations, which are pairs comprising a program fragment and a state. The proof rules of the Owicki–Gries formalisation are syntax directed. A proof obligation generator has been implemented in the form of an Isabelle tactic called `oghoare`. Application of this tactic results in generation of all standard Owicki–Gries proof obligations, each of which can be discharged either automatically or via an interactive proof. We omit the full details of standard semantics and verification condition generation [30].

We provide an encoding of the MP litmus test in Fig. 7 to provide an example instantiation of the abstract syntax above, and to better highlight the extensions necessary to handle C11-style weak memory. The state of the program is defined using an Isabelle record where all shared variables and local registers are modelled as its fields. For the proof outline in Fig. 7, the `oghoare` tactic generates 29 proof obligations, each of which is automatically discharged.

4 Extending Owicki–Gries to C11-Style Memory Models

We defer a precise description of a C11-style operational semantics to Sect. 4.2 in order to highlight the fact that our Isabelle framework is essentially parametric in the memory model used. The fundamental requirement is that the memory model be an operational model featuring C-style, annotated memory operations. All that is needed to understand the rest of this section is some basic familiarity with weak memory models [9,14,21,34]. The functions encoding the weak memory operational semantics `WrX`, `WrR`, `RdX`, …will be instantiated in Sect. 4.2, and for the time being can be considered to be transition functions that construct a new weak memory state for a given weak memory prestate. However, a reader may wish to first read Sect. 4.2 for an example C11 memory model prior to continuing with the rest of this section.

To motivate our language extension, we reconsider MP (Figs. 2, 3) in a C11-style weak memory model. In particular, if all reads and writes are *relaxed*, C11 admits an execution in which thread 2 reads a "stale" value of d [21,26]. Thus, it is only possible to establish the weaker postcondition $r2 = 0 \vee r2 = 5$ (see Sect. 4.2 for details). To regain the expected behaviour, one must introduce additional synchronisation in the program. In particular, the write to f in thread 1 must be a *releasing write* (denoted $f :=^R 1$) and the read of f in thread 2 must be an *acquiring read* (denoted $r_1 \leftarrow^A f$).

A weak memory state can be encoded as a special variable in the standard semantics. Moreover, for the semantics that we employ [9,14], within each weak memory state, for each low-level weak memory event (e.g. read or write), we must keep track of the thread identifier (of type `T`), the shared variable (or location) that is accessed (of type `L`) and the value in that variable (of type `V`).

4.1 Syntactic Extension

To capture the so-called RAR fragment, we require five new programming constructs: *relaxed reads and writes*, *releasing writes*, and *acquiring reads*. Moreover, we wish to support a `SWAP[x, v]` command [14,41] that acquiringly reads `x` and releasingly writes `v` to `x` in

Fig. 8 Isabelle encoding of the load-buffering litmus test

```
consts
  x :: L  y :: L

record lb_state =
  r1 :: V
  r2 :: V
  σ :: Cstate

||-
{| r1 = 0 ∧ r2 = 0 ∧
  [x =1 0] σ ∧ [x =2 0] σ ∧
  [y =1 0] σ ∧ [y =2 0] σ |}
COBEGIN
  {| [y =2 0] σ ∧ r2 = 0 |}
  < r1 ←1 x σ > ;;
  {| [y =2 0] σ ∧ r2 = 0 |}
  < y :=1 1 σ >
  {| r1 = 0 ∨ r2 = 0 |}
||
  {| [x =1 0] σ ∧ r1 = 0 |}
  < r2 ←2 y σ > ;;
  {| [x =1 0] σ ∧ r1 = 0 |}
  < x :=2 1 σ >
  {| r1 = 0 ∨ r2 = 0 |}
COEND
{| r1 = 0 ∨ r2 = 0 |}
```

a single atomic step. This command is used in Peterson's algorithm (see Fig. 12) and is implemented in our model using a *read–modify–write update*.

All of the new extensions are defined using a shallow embedding and their concrete syntax is enclosed in brackets < ... > to avoid ambiguities in the Isabelle/HOL encoding. The annotated versions of these statements are given below. For completeness, we also require syntax for unannotated versions of each command, but their details are elided.

```
syntax
"_AnnWrX" :: "α assn ⇒ L ⇒ T ⇒ V ⇒ Cstate ⇒ α ann_com"
        ("(_ <_ :=_ _ _>)")
"_AnnWrR" :: "α assn ⇒ L ⇒ T ⇒ V ⇒ Cstate ⇒ α ann_com"
        ("(_ <_ :=R _ _ _>)")
"_AnnRdX"::"α assn ⇒ idt ⇒ T ⇒ L ⇒ Cstate ⇒ α ann_com"
        ("(_ < _ ←_ _ _ >)")
"_AnnRdA"::"α assn ⇒ idt ⇒ T ⇒ L ⇒ Cstate ⇒ α ann_com"
        ("(_ < _ ←A _ _ _ >)")
"_AnnSwap"::"α assn ⇒ L ⇒ V ⇒ T ⇒ Cstate ⇒ α ann_com"
        ("(_ <SWAP[_, _]_ _>)")
```

To cope with weak memory, we embed the weak memory state as a special variable in the standard encoding (see Figs. 8, 9). Each operation induces an update to this embedded weak memory state variable that can be observed by subsequent operations on the weak memory state.

`_AnnWrX` defines a relaxed write. Its first argument is an assertion (the precondition) of the command, the second is the variable being modified, the third is the thread performing the operation, the fourth is the value being written, and the fifth is the weak memory prestate. Similarly, `_AnnWrA` is a releasing write. `_AnnRdX` defines a relaxed read, which loads a value of the given location (of type `L`) from the given weak memory prestate into the second argument

(of type idt). An acquiring read, defined by _AnnRdA, is similar. Finally, _AnnSwap defines a swap operation that writes the given value (third argument) to the given location (second argument) using an update operation.

The semantics of this extended syntax is given by a translation, which updates the program variables, including the weak memory state. For the commands above, after omitting some low-level Isabelle details, we have:

```
translations
"r < x :=t v s>" ⇀
   "AnnBasic r (_update_name s (WrX s x v t))"
"r < x :=Rt v s>" ⇀
   "AnnBasic r (_update_name s (WrR s x v t))"
"r < x ←t y s>" ⇀
   "AnnAwait r ((_update_name s (fst (RdX s y t))),,
                (_update_name x (snd (RdX s y t))))"
"r < x ←At y s>" ⇀
   "AnnAwait r ((_update_name s (fst (RdA s y t))),,
                (_update_name x (snd (RdA s y t))))"
"r <SWAP[x, v]t s>" ⇀
    "AnnBasic r (_update_name s (fst (Upd s x v t)))"
```

These translations rely on an operational semantics defined by functions WrX (relaxed write), WrR (releasing write), RdX (relaxed read), RdA (acquiring read) and Upd (RMW update), which we define in Sect. 4.2.

Relaxed and acquiring writes update the embedded weak memory state to the state returned by WrX and WrA, respectively. A read event must return a post state (which is used to update the value of the embedded weak memory state) and the value read (which is used to update the value of the local variable storing this value). In order to preserve atomicity of the read, we wrap both updates within an annotated AWAIT statement. The translation of a SWAP is similar.

Note that a relaxed (acquiring) read comprises two calls to RdX (RdA), which one could mistakenly believe to cause two different effects on the weak memory state. However, as we shall see, these operations are implemented using Hilbert choice (SOME), hence, although there may be multiple values that a read could return, the two applications of RdX (RdA) are identical for the same value for the same parameters.

4.2 Operational Semantics of C11 RAR in Isabelle/HOL

We now present details of the memory model from Sect. 2.1 as encoded in Isabelle/HOL. Recall that the main purpose of this section is to instantiate the functions WrX, WrR, RdX, RdA and Upd from Sect. 4.1.

Recall that type L represents shared variables (or locations), T represents threads, and V represents values. We use type TS (which is synonymous with rational numbers) to represent timestamps. Each write can be uniquely identified by a variable-timestamp pair. The type Cstate is a nested record with fields

- writes, which is the set of all writes,
- covered, which is the set of covered writes (recalling that covered writes are used to preserve atomicity of read–modify–write updates),
- mods, which is a function mapping each write to a write record (see below),

Fig. 9 Isabelle encoding of the message-passing litmus test

```
consts
 d :: L   f :: L

record mp_state =
 r1 :: V  r2 :: V  σ::Cstate

||-
{| [f =1 0] σ ∧ [f =2 0] σ ∧
  [d =1 0] σ ∧ [d =2 0] σ |}
COBEGIN
  {| ¬[f ≈2 1] σ ∧ [d =1 0] σ |}
  <d :=1 5 σ> ;;
  {| ¬[f ≈2 1] σ ∧ [d =1 5] σ |}
  <f :=R1 1 σ>
  {| True |}
||
  DO {| [f = 1](d =2 5) σ |}
     < r1 ←A2 f σ>
  UNTIL ( r1 = 1)
  INV {| [f = 1](d =2 5) σ ∧
      ( r1 = 1 ⟶ [d =2 5] σ) |}
  OD;;
  {| [d =2 5] σ|}
  < r2 ←2 d σ>
  {| r2 = 5 |}
COEND
{| r2 = 5 |}
```

- `tview`, which is the viewfront (type `L` ⇒ `(L` × `TS)`) of each thread, and
- `mview`, which is the viewfront of each write.

A write record contains fields `val`, which is the value written and `rel`, which is a Boolean that is `True` if, and only if, the corresponding write is releasing.

```
record Cstate =
 writes::(L × TS) set
 covered::(L × TS) set
 mods::(L × TS) ⇒ write_record
 tview::T ⇒ L ⇒ (L × TS)
 mview::(L × TS) ⇒ L ⇒ (L × TS)
```

```
record write_record =
 val :: V
 rel :: bool
```

Next, we describe how the operations modify the weak memory state.

Read Transitions Both relaxed and acquiring reads leave all state components unchanged except for `tview`. To define their behaviours, we first define a function `visible_writes` σ `t x` [4] that returns the set of writes to `x` that thread `t` may read from in state σ. For a write `w` = `(x, q)`, we assume a pair of functions `var w` = `x` and `tst w` = `q` that return the variable and timestamp of `w`, respectively. Thus, we obtain:

```
definition "visible_writes σ t x ≡
   {w ∈ writes σ . var w = x ∧ tst(tview σ t x) ≤ tst w}"
```

We use a function `getVW` to select some visible write from which to read:

```
definition "getVW σ t x ≡
   (SOME w . w ∈ visible_writes σ t x)"
```

[4] `visible_writes` is the name of function and σ, `t` and `x` are its arguments.

Finally, we require functions RdX t w σ and RdA t w σ that update the tview component of σ for thread t reading write w. Function RdX t w σ updates tview σ t to (tview σ t)[var w := w], where $f[x := v]$ denotes functional override. That is, the viewfront of thread t for variable var w is updated to the write w that t reads. The viewfronts of the other threads as well as the viewfront of t on variables different from var w are unchanged. Thus, the function RdX required by the translation of a relaxed read command in Sect. 4 is thus defined by:

```
definition value σ w ≡ val (mods σ w)

fun RdX :: "L ⇒ T ⇒ Cstate ⇒ (Cstate × V)" where
    "RdX x t σ = (let w = getVW σ t x; v = value σ w in
                  (read_transX t w σ, v))"
```

We use value σ w to obtain the value of the write w in state σ. The update defined by function RdA t w σ for an acquiring read is conditional on whether w is a relaxed write. If w is relaxed, tview σ t is updated to (tview σ t)[var w := w] (i.e. behaves like a relaxed read). Otherwise, the viewfront of t must be updated to "catch up" with the viewfront of w. In particular, tview σ t is updated to (tview σ t) ⊗ (mview σ w), where

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

Overall, we have:

```
fun RdA :: "L ⇒ T ⇒ Cstate ⇒ (Cstate × V)" where
    "RdA x t σ = (let w = getVW σ t x; v = value σ w in
                  (read_transA t w σ, v))"
```

Write Transition Writes update all state components except covered. First, following Doherty et al. [14], we must identify an existing write w in the current state; the new write is to be inserted immediately after w. Moreover, w must be visible to the thread performing the write and covered by an RMW update. We define the following function:

```
definition "getVWNC σ t x ≡
   SOME w . w ∈ visible_writes σ t x ∧ w ∉ σ covered"
```

where NC stands for "not covered". The write operation must also determine a new timestamp, ts for the new write. Given that the new write is to be inserted immediately after the write operation w, the timestamp ts must be greater than tst w but smaller than the timestamp of other writes on var w after w. Thus, we obtain a new timestamp using:

```
definition "getTS σ w ≡
  SOME ts . tst w < ts ∧
    (∀ w' ∈ writes σ. var w' = var w ∧ tst w < tst w' ⟶
                              ts < tst w')
```

Finding such a timestamp is always possible since timestamps are rational numbers (i.e. are dense).

As with reads, we require a function write_trans t b w v σ ts that updates the state σ so that a new write w' = ((var w), ts) for thread t is introduced with write value v. The Boolean b is used to distinguish relaxed and releasing writes. The write w is the write after which the new write w' is to be introduced. The effect of write_trans is to update writes σ

to writes′, mods σ to mods′ and both tview σ t and mview σ w′ to tview′, where:

$$\begin{aligned} \texttt{writes}' &= (\texttt{writes}\ \sigma) \cup \{\texttt{w'}\} \\ \texttt{mods}' &= (\texttt{mods}\ \sigma\ \texttt{w'})[\texttt{val} := \texttt{v}, \texttt{rel} := \texttt{b}] \\ \texttt{tview}' &= (\texttt{tview}\ \sigma\ \texttt{t})[(\texttt{var}\ \texttt{w}) := \texttt{w'}] \end{aligned}$$

Thus, writes′ adds the new write w' to the set of writes of σ. The new mods′ sets the value for w' to v and the rel field to b (which is True iff the new write w' is releasing). Finally, tview′ updates tview of t for variable var w (the variable that both w and w' update) to w'.

Finally, the functions WrX and WrR required by the translations in Sect. 4 are given as follows:

```
fun WrX :: "L ⇒ V ⇒ T ⇒ Cstate ⇒ Cstate" where
  "WrX x v t σ =
      (let w = getVWNC σ t x ; ts' = getTS σ w in
                write_trans t False w v σ ts')"
fun WrR :: "L ⇒ V ⇒ T ⇒ Cstate ⇒ Cstate" where
  "WrR x v t σ =
      (let w = getVWNC σ t x ; ts' = getTS σ w in
                write_trans t True w v σ ts')"
```

Update Transition Following Doherty et al. [14], we assume that an update performs both an acquiring read and a releasing write in a single step (atomically). It is possible to define variations that do not synchronise the read or a write, but we omit such details for simplicity.

We first define a function update_trans t w v σ ts that modifies σ so that a releasing write w' = ((var w), ts) by thread t is introduced with write value v immediately after an existing write w. The effect of update_trans is to update writes σ to writes′, covered σ to covered′, and mods σ to mods′, and both tview σ t and mview σ w' to tview′, where

$$\begin{aligned} \texttt{writes}' &= (\texttt{writes}\ \sigma) \cup \{\texttt{w'}\} \\ \texttt{covered}' &= (\texttt{covered}\ \sigma) \cup \{\texttt{w}\} \\ \texttt{mods}' &= (\texttt{mods}\ \sigma\ \texttt{w'})[\texttt{val} := \texttt{v},\ \texttt{rel} := \texttt{True}] \\ \texttt{tview}' &= \begin{cases} (\texttt{tview}\ \sigma\ t)[(\texttt{var}\ \texttt{w}) := \texttt{w'}] \otimes (\texttt{mview}\ \sigma\ \texttt{w}) & \text{if } \texttt{rel}\ (\texttt{mods}\ \sigma\ \texttt{w}) \\ (\texttt{tview}\ \sigma\ t)[(\texttt{var}\ \texttt{w}) := \texttt{w'}] & \text{otherwise} \end{cases} \end{aligned}$$

Thus, writes′ adds the new write w' corresponding to the update to the set of writes of σ and covered′ adds the write w that w' reads from to the covered writes set of σ. The new mods′ sets the value for w' to v and sets the rel field to True. Finally, tview′ updates tview of t in the same way as a read operation, except that the first case is taken provided the write w being read is releasing.

The function Upd required by the translation in Sect. 4 is given as follows:

```
fun Upd :: "L ⇒ V ⇒ T ⇒ Cstate ⇒ (Cstate × V)" where
"Upd x v t σ = (let w = getVWNC σ t x ; ts = getTS σ w ;
                    v = value σ w in
                       (update_trans t w v σ ts, v))"
```

Well-Formedness Section 5 presents an assertion language for verifying C11 programs. The lemmas introduced there require states to be *well-formed*, which we characterise by predicate wfs defined below.

```
definition "writes_on σ x ≡ {w. var w = x ∧ w ∈ writes σ}"

definition "lastWr σ x ≡ (x, Max (tst`(writes_on σ x)))"

definition "wfs σ ≡
        (∀ t x. tview σ t x ∈ writes_on σ x) ∧
        (∀ w x. mview σ w x ∈ writes_on σ x) ∧
        (∀ x. finite(writes_on σ x)) ∧
        (∀ w. w ∈ writes σ ⟶ mview σ w (var w) = w) ∧
        (∀ w x. w = lastWr σ x ⟶ w ∉ covered σ)"
```

Function `writes_on` σ `x` returns the set of writes in σ to variable `x`. Function `lastWr` σ `x` returns the write on `x` whose timestamp is greater than the timestamp of all other writes on `x` in state σ.

In the definition of `wfs` σ, the first two conjuncts ensure that all writes recorded in `tview` and `mview` are consistent with `writes` σ. The third ensures the set of writes in σ is finite and the fourth ensures that for each write in σ, the write's modification view of the variable, it writes is the write itself. The final conjunct ensures that the last write to each variable (i.e. the write with the largest timestamp) is not covered. We have shown that `wfs` is stable under each of the transitions `WrX`, `WrR`, …. Thus, the well-formedness assumption made by the lemmas in Sect. 5 is trivially guaranteed.

5 An Assertion Language for Verifying C11 Programs

In the previous sections, we discussed how the existing Owicki–Gries theories in Isabelle can be extended with a weak memory C11 operational semantics in order to reason about C11-style programs using standard proof rules. We mentioned that how a novel set of assertions introduced in [9] can be used in our extension to annotate programs w.r.t. C11 state and reason about them. In this section, we introduce the assertion language and present their encodings in Isabelle through a number of examples and litmus tests. We also provide some of the rules (lemmas) that Isabelle uses to discharge proof obligations and validate the proof outlines. We show how C11 state is incorporated into the programs and shared variables are defined. We also present a fully verified encoding of the Peterson's mutual exclusion algorithm and read–copy–update (RCU) algorithm to further validate our approach.

5.1 Load Buffering

Our first example is the load-buffering litmus test given in Fig. 8. It comprises shared variables x and y both initialised to 0. Thread 1 loads x into local register $r1$, then updates y to 1. Thread 2 is symmetric; it loads y into local register $r2$, then updates x to 1. In some memory models [33], it is possible for both threads to read the later writes and terminate in the state $r1 = r2 = 1$. As discussed earlier, we use restricted C11 memory model described by Lahav et al. [27], and hence we prevent the program from terminating by reading 1 for both `x` in thread 1 and `y` in thread 2. Thus, the program guarantees the postcondition that $r1 = 0 \vee r2 = 0$, i.e. the program does not terminate in the state $r1 = 1 \vee r2 = 1$.

As mentioned earlier, the C11 state is represented as a field of the record corresponding to the state of the program (i.e. as a field of `lb_state` record for the load-buffering example). Updates to σ are via the underlying definition of the operations in accordance with the RC11-

RAR operational semantics as described in Sect. 4.2. In our encoding, shared variables are represented as constants representing locations in the C11 state (σ).

Now, consider the proof outline. The first assertion (lines 10-12) specifies the initial state of the program. The first two conjuncts are assertions on the value of local registers. The other four conjuncts are *definite observation* assertions. A definite observation assertion denoted $[x =_t n]\ \sigma$ states that thread t's viewfront is consistent with the last write to x in σ and that this write has value n. Thus, if t reads from x, it is guaranteed to return n. σ. Formally,

```
[x =ₜ n] σ ≡
    tview σ t = lastWr σ x ∧ value σ (lastWr σ x) = n
```

All weak-memory assertions in the proof outline of Fig. 8 are definite value assertions, and this is sufficient to prove the postcondition. However, to discharge the generated proof obligations, we require the following two proof rules over C11 assertions, which are defined as Isabelle lemmas:

```
lemma d_obs_RdX_pres:
assumes "wfs σ"
and "[x =ₜ u] σ"
shows "[x =ₜ u]
       (fst(RdX y t' σ))"
```

```
lemma d_obs_WrX_diff_var_pres:
 assumes "wfs σ"
   and "[x =ₜ u] σ"
   and "y ≠ x"
   shows"[x =ₜ u] (WrX y v t' σ)"
```

Lemmas d_obs_RdX_pres and d_obs_WrX_diff_var_pres give conditions for preserving definite value assertions for relaxed read and write transitions, respectively, for an arbitrary variable y and thread t'. Note d_obs_WrX_diff_var_pres requires that the variable y that is written to is different from the variable x that appears in the definite value assertion. Both lemmas are proved sound with respect to the operational semantics in Sect. 4.2. Of course, d_obs_RdX_pres also holds for an acquiring read transition and d_obs_WrX_diff_var_pres for a releasing write transition[5].

The assertions on lines 14 and 20 are locally correct because of the initial state. The assertions on lines 16 and 22 are locally correct using d_obs_RdX_pres. Local correctness of the assertions on line 18 and 24 is trivial follows by the definite value assertion. Interference freedom of the assertions in lines 14, 16, 20 and 22 is also established using the two lemmas.

5.2 Message-Passing

We now consider the assertions used in the message-passing algorithm (Fig. 9). The first new assertion used in the proof outline is the *possible observation* assertion. This assertion (denoted $[x \approx_t n]\ \sigma$) states that thread t may read value n if it reads from variable x. The formal definition in Isabelle is as follows:

```
[x ≈ₜ n] σ ≡ ∃ w.w ∈ visible_writes σ t x ∧ n = value σ w
```

The next assertion we introduce is the *conditional observation* assertion (denoted $[x = n]\langle y =_t m\rangle\ \sigma$) which states that if thread t reads a value n using an acquiring read for x, it synchronises with the corresponding write and obtains the definite value m for y. Note that this requires that any write to x with value n that t can read is a releasing write. The formal definition in Isabelle is as follows:

[5] In fact, our Isabelle theory provides a generic lemma that applies to both cases simultaneously.

```
[x = n]⟪y =t m⟫ σ ≡
  ∀ w ∈ visible_writes σ t x. value σ w = n ⟶
     mview σ w y = lastWr σ y ∧ value σ (lastWr σ y) = m
        ∧ rel (mods σ w)"
```

Here, we only introduce two of the interesting rules used in the proof, and refer the interested reader to the Isabelle theories for the remaining lemmas:

```
lemma c_obs_intro:
assumes "wfs σ"
and "[y =t m] σ"
and "¬[x ≈t' n] σ"
and "x ≠ y"
and "t ≠ t'"
shows "[x = u]⟪y =t' v⟫
         (WrR x u t σ)"
```

```
lemma c_obs_transfer:
assumes "wfs σ"
and "[x = u]⟪y =t v⟫ σ"
and "snd(RdA σ x t) = u"
shows "[y =t v] (fst(RdA x t σ))"
```

Consider the conditional observation assertion in line 17. Local correctness holds trivially by initialisation. Interference freedom under line 12 is straightforward. For interference freedom under line 14, we use `c_obs_intro`. In particular, the assertion at line 13 (i.e. the precondition of line 14) satisfies the critical premises of `c_obs_intro`. We use the conditional observation assertion (line 17 of thread 2) in combination with rule `c_obs_transfer` to establish a definite observation on a new variable in thread 2. We note that the variable read by the transition in rule `c_obs_transfer` is `x`, whereas the definite value assertion in the consequent is on variable `y`. Full details of this proof may be found in [9]; in this paper, we focus on automation, which we discuss in Sect. 7.

5.3 Read–Read Coherence

We have verified three versions of the read–read coherence (RRC) litmus test using our extended theories. The RRC litmus test guarantees whether or not two successive reads from the same variable are ordered. We have provided the more interesting of the three in Fig. 10, which comprises three threads. The other two versions are provided in "Appendix A".

- The first thread (i.e. thread 1) updates x to 1, then signals that this has been done using a releasing write to y.
- The second thread (i.e. thread 2) reads y using an acquiring read, then updates x to 2.
- The third thread (i.e. thread 3) performs two successive reads of x.

If thread 2 reads the value 1 for y, then it must have also encountered the write of $x = 1$ in thread 1. Thus, thread 2's update of x to value 2 must be ordered after the write $x = 1$. This means that if thread 3 reads the value 2 for x it must no longer be possible for it to read the value 1 since this would be against the coherence order $x = 1$ followed by $x = 2$.

In order to prove this example, a richer set of assertions is required. In particular, in addition to the assertions regarding observability of writes, we need assertions about the order of writes and the limits on the occurrence of values.

The first assertion used for this example that we discuss here is the *possible value order* assertion (denoted $[\mathtt{m} \prec_{\mathtt{x}} \mathtt{n}]\,\sigma$), which states that there exists a write to variable `x` with value `n` ordered after (i.e. with a greater timestamp) a write to `x` with value `m`. Similarly, a *definite value order* assertion (denoted $[\mathtt{m} \prec\!\!\prec_{\mathtt{x}} \mathtt{n}]\,\sigma$) states that all writes to `x` with value `n` are ordered after all writes to `x` with value `m`. These are formally defined in Isabelle as follows:

```
consts
  x :: L   y :: L
record rrc3_state =
  a :: V  b :: V  r1 :: V
  σ :: Cstate

||-
{[x =2 0] σ ∧ [y =3 0] σ}
COBEGIN
{[𝕆x 1]0 σ ∧ ¬[y ≈2 1] σ ∧ ([𝕆x 2]0 σ ⟶ [x =1 0] σ)}
  <x :=1 1 σ> ;;
{¬ [y ≈2 1] σ ∧ ([𝕆x 2]0 σ ⟶ [x =1 1] σ)}
  <y :=R1 1 σ>
{ True }
||
{[𝕆x 2]0 σ ∧ [y = 1](x =2 1) σ}
  < r1 ←A2 y σ> ;;
{( r1 = 1 ⟶ [ x =2 1] σ ∧ [x =enc2 1] σ) ∧ [𝕆x 2]0 σ}
  <x :=2 2 σ>
{ r1 = 1 ⟶ [1 ≺x 2] σ}
||
{ True }
  < a ←3 x σ> ;;
{ [x =enc3 a] σ }
  < b ←3 x σ>
{( a ≠ b ⟶ [ a ≺x b] σ)}
COEND
{ r1 = 1 ∧ a = 2 ⟶ b ≠ 1}
```

Fig. 10 Isabelle encoding of the read–read coherence litmus test with three threads. The proof additionally relies on a global invariant [init x 0] σ ∧ [init y 0] σ ∧ [𝟙x 1] σ ∧ [𝟙x 2] σ

```
[m ≺x n] σ ≡
 ∃ w w'. w ∈ writes_on σ x ∧ w' ∈ writes_on σ x ∧
      value σ w = m ∧ value σ w' = n ∧ (tst w) < (tst w')
```

```
  [m ≺≪x n] σ ≡
     (∀ w w'. w ∈ writes_on σ x ∧ w' ∈ writes_on σ x ∧
       value σ w = m ∧ value σ w' = n ⟶ (tst w) < (tst w'))
    ∧ [m ≺x n] σ
```

The other two assertions that appear in this proof outline fall into the *value occurrence* category: $[\mathbb{0}_x\, n]_i$ means that there has not been a write with value n to variable x (where i is the initial value of x) and $[\mathbb{1}_x\, n]$ means that there has been at most one write with value n to x. These two assertions are defined in terms of ordering assertions introduced earlier as follows. The predicate init σ x n holds iff the initial value of x in σ is n.

```
definition init σ x n ≡
 ∃ w . w ∈ writes_on σ x ∧ value σ w = n ∧
   (∀ w' ∈ writes_on σ x . w ≠ w' ⟶  (tst w) < (tst w'))

[𝕆x n]i σ ≡ init σ x n ∧ ¬[i ≺x n]

[𝟙x n] σ ≡ ¬[n ≺x n]
```

The last new assertion used in this proof outline is *encountered value* (denoted as $[x =^{enc}_t n]$)

means that thread t has had the opportunity to observe a write with value n of x. This assertion is formally defined in Isabelle as follows[6]:

```
[x ≝ᵉⁿᶜ_t n] σ ≡ ∃ w . w ∈ writes_on σ x ∧
                 tst(w) ≤ tst(tview σ t x) ∧ value σ w = n
```

The five assertions above, together with other assertions introduced earlier, are sufficient to specify the behaviour of the three-threaded version of RRC. The conditional observation assertion on line 18 is used to capture the possible synchronisation between threads 1 and 2. The ordering assertions in thread 2 and 3 specify that if the writes have happened in a specific order, the read order must remain coherent with respect to the order of writes. Namely, if thread 2 synchronises with thread 1 (i.e. r1 is set to 1), then it must have observed the write of x at line 12. Thus, the write to x with value 2 at line 23 must have happened after. Therefore, it must be impossible for the third thread to read value 2 for x at line 27, then subsequently read 1 for x at line 29. This reasoning is captured by the postcondition of the program.

5.4 Two-Way Message-Passing

Our next litmus test (taken from [26]) involves two-way message passing (Fig. 11). The program has two shared variables r and w. Thread 2 reads the value of w and writes it to r twice. Thread 1 writes 1 to w and then waits until it sees 1 for r to terminate. The goal here is to prove that once the program is terminated, the only visible value for r is 1. We stated this property as follows:

$$[r =_2 1]\ \sigma \wedge (\forall j.j \neq 1 \rightarrow \neg[r \approx_1 j]\ \sigma)$$

The above assertion states that thread 2 definitely observes 1 for r and it is impossible for thread 1 to see any value for r which is not equal to 1.

6 Case Studies

In addition to litmus tests given previously, to further investigate the effectiveness of our approach, we verified two larger case studies, namely Peterson's mutual exclusion algorithm and a version of read–copy–update (RCU) algorithm. This section provides more details on these two case studies (Figs. 12 and 13).

6.1 Peterson's Algorithm for C11

We now turn to our first 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[7], shown in Fig. 12, is a mutual exclusion algorithm for two threads implemented for C11 using release-acquire annotations [41]. The purpose

[6] Note that some of the notation in our Isabelle encoding is different. We use the notation from [9] here for a cleaner presentation.

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

```
1  consts
2    r :: L    w :: L
3  record twowaymp_state =
4    wr:: V  rr :: V    σ :: Cstate
5
6  ||-
7  {|[r =1 0] σ ∧ [r =2 0] σ ∧ [w =1 0] σ ∧ [w =2 0] σ ∧ [𝕆r 1]0 σ|}
8  COBEGIN
9  {|[w =1 0] σ ∧ ¬[w ≈2 1] σ ∧ (wr = 0) ∧ (∀j.j ≠ 0 ∧ j ≠ 1 → ¬[r ≈1 j] σ)|}
10 < w :=1R 1 σ >;;
11 DO
12 {|(∀j.j ≠ 0 ∧ j ≠ 1 → ¬[r ≈1 j] σ) ∧ (wr = 1 → [r =enc2 1] σ)|}
13   < wr ←1A r σ >
14 UNTIL wr = 1
15 INV {|(∀j.j ≠ 0 ∧ j ≠ 1 → ¬[r ≈1 j] σ) ∧ (wr = 1 → [r =enc2 1] σ)|}
16 OD
17 {|wr = 1 ∧ (∀j.j ≠ 0 ∧ j ≠ 1 → ¬[r ≈1 j] σ) ∧ [r =enc2 1] σ|}
18
19 ||
20
21 {|[r =2 0] σ ∧ rr = 0 ∧ wr = 0 ∧ (∀j.j ≠ 0 ∧ j ≠ 1 → ¬[w ≈2 j] σ)
22  ∧[w = 1](|w =2 1|) σ ∧ ¬[r ≈1 1] σ ∧ [𝕆r 1]0 σ|}
23 < rr ←2A w σ >;;
24 {|[r =2 0] σ ∧ (rr = 0 ∨ rr = 1) ∧ (∀j.j ≠ 0 ∧ j ≠ 1 → ¬[w ≈2 j] σ)
25 ∧[w = 1](|w =2 1|) σ ∧ (rr = 1 → [w =2 1] σ) ∧ wr ≠ 1 ∧ ¬[r ≈1 1] σ ∧ [0r1]0 σ|}
26 < r :=2R rr σ >;;
27 {|(rr = 0 ∨ rr = 1) ∧ (∀j.j ≠ 0 ∧ j ≠ 1 → ¬[w ≈2 j] σ) ∧ [w = 1](|w =2 1|) σ ∧ ([r =2 rr] σ)
28 ∧(rr = 1 → [r =2 1] σ ∧ [w =2 1] σ) ∧ (wr = 1 → [r =2 1] σ) ∧ (¬[r ≈1 1] σ ∨ ([r =2 1] σ ∧ [w =2 1] σ))
29 ∧[r = 1](|r =1 1|) σ ∧ (([𝕆r 1]0 σ ∧ [r =2 0] σ) ∨ ([0 ≮r 1] σ ∧ [r =2 1] σ))|}
30 < rr ←2A w σ >;;
31 {|([r =2 1] σ ∨ [r =2 0] σ) ∧ (rr = 0 ∨ rr = 1) ∧ ([r =2 1] σ → rr = 1)
32 ∧(rr = 1 → [w =2 1] σ) ∧ (wr = 1 → [r =2 1] σ) ∧ (¬[r ≈1 1] σ ∨ [r =2 1] σ)
33 ∧(([𝕆r 1]0 σ ∧ [r =2 0] σ) ∨ ([0 ≮r 1] σ ∧ [r =2 1] σ))|}
34 < r :=2R rr σ >
35 {|(wr = 1 → [r =2 1] σ) ∧ (¬[r ≈1 1] σ ∨ [r =2 1] σ) ∧ ([0 ≮r 1] σ ∨ [r =2 0] σ)|}
36 COEND
37 {|[r =2 1] σ ∧ (∀j.j ≠ 1 → ¬[r ≈1 j] σ)|}
```

Fig. 11 Isabelle encoding of a two-way message-passing

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 at the same time. As with the original algorithm, variable $flag_{\mathtt{i}}$, 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 $flag_{\mathtt{i}}$ 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 [5]), 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 8) using a relaxed write (which cannot induce any synchronisation), but is set to 0 (line 34) 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 18 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 18, and then to enter the critical section.

The busy wait loop forces thread 1 to wait until either $flag_2$ is 0 (indicating that thread 2 is not trying to enter the critical section) or $turn = 1$ (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 25) is relaxed.

```
1  ||-
2  {| ¬after1 ∧ ¬after2 ∧ [flag1 =1 0] σ ∧ [flag2 =2 0] σ ∧
3    [turn =1 0] σ ∧ [turn =2 0] σ |}
4  COBEGIN
5   {| ¬after1 ∧ [flag1 =1 0] σ ∧ (cvd[turn, 0] σ ∨ cvd[turn, 1] σ) ∧
6     (after2 ⟶ cvd[turn, 1] σ ∧ [turn = 1]⟨flag2 =1 1⟩ σ) ∧ ¬[turn ≈2 2] σ |}
7
8   <flag1 :=1 1 σ>;;
9
10  {| ¬after1 ∧ [flag1 =1 1] σ ∧ ¬[turn ≈2 2] σ ∧
11    (after2 -⟶ cvd[turn, 1] σ ∧ [turn = 1]⟨flag2 =1 1⟩ σ) |}
12
13  ⟨<SWAP[turn, 2]1 σ>,, after1 := True⟩ ;;
14
15  DO
16   {| after1 ∧ (after2 ∧ ([flag2 ≈1 0] σ ∨ [turn ≈1 1] σ) ⟶ [turn =2 1] σ)|}
17
18   < r1 ←A1 flag2 σ>;;
19
20   {| after1 ∧
21     (after2 ∧ ( r1 = 0 ∨ [turn ≈1 1] σ ∨ [flag2 ≈1 0] σ) ⟶ [turn =2 1] σ)|}
22
23   < r2 ←1 turn σ>
24
25  UNTIL ( r1 = 0 ∨ r2 = 1)
26  INV
27  {| after1 ∧
28    (after2 ∧ ( r1 = 0 ∨ r2 = 1 ∨ [turn ≈1 1] σ ∨ [flag2 ≈1 0] σ) ⟶ [turn =2 1] σ) |}
29  OD ;;
30  {| after1 ∧ (after2 ⟶ [turn =2 1] σ)|}
31
32   //Critical Section
33
34  ⟨<flag1 :=R1 0 σ>,, after1 := False⟩
35
36  {| True |}
37
38 || Thread 2 (symmetric)
39 COEND
40 {| True |}
```

Fig. 12 Proof outline for Peterson's algorithm under C11. The second thread (not shown here) 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 32) 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*:

$$\texttt{after1} \wedge (\texttt{after2} \longrightarrow [\texttt{turn} =_2 1]\,\sigma) \wedge \texttt{after2} \wedge (\texttt{after1} \longrightarrow [\texttt{turn} =_1 2]\,\sigma)$$

It is easy to see that this implies $[\texttt{turn} =_1 2]\,\sigma \wedge [\texttt{turn} =_2 1]\,\sigma$. However, this situation is impossible since two threads cannot have different definite observations.

The first step is more elaborate and we only describe certain aspects. The precondition of line 18 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.

The use of auxiliary variables is a standard technique used in Owicki–Gries proofs of Peterson's algorithm in the conventional, sequentially consistent, setting [5,30]. Note that introduction of auxiliary variables must follow the same rules as the classical setting [31] and must not be a shared constant that appears within the weak memory state

σ. This avoids the notions of unsoundness of auxiliary variables described in earlier work [26].

Proving that the precondition of line 18 is satisfied in the post-state of line 13 requires using a feature of the assertion language, closely related to the semantics of RMW operations, that we now introduce. The proof outline for this algorithm has the new assertion *covered* (denoted `cvd[x, n] σ`). The assertion `cvd[x, n] σ` means that every write to `x` in state σ except the last is covered and the value written by that last write is `n`. This assertion is formally defined in Isabelle as:

```
cvd[x, n] σ ≡
  ∀ w. w ∈ writes σ ∧ var σ w = x ∧ w ∉ covered σ ⟶
                                 w = lastWr σ x ∧ value σ w = n
```

Similar to the previous examples, in order to prove the Peterson's algorithm we will need to introduce new proof rules to deal with assertions involving `cvd`. Here, we present couple of the most interesting proof rules related to the covered assertion:

```
lemma cvd_wr_diff_var_pres:
assumes "wfs σ"
and "cvd[x, u] σ"
and "x ≠ y"
shows "cvd[x, u] (WrR y v t σ)"
```

```
lemma cvd_upd_intro:
assumes "wfs σ"
and "cvd[x, u] σ"
shows "cvd[x, v] (fst(Upd x v t σ))"
```

```
lemma cvd_c_obs_transfer:
assumes "wfs σ"
and "[x = u]⟨y =t v⟩ σ"
and "cvd[x, u] σ"
and "x ≠ y"
shows "[y =t v]
         (fst(Upd x m t σ))"
```

```
lemma cvd_rd_pres:
assumes "wfs σ"
and "cvd[x,u] σ"
shows "cvd[x,u] (fst(Rd y t b σ))"
```

The first lemma (`cvd_wr_diff_var_pres`) states that a write to a different variable preserves the covered assertion. Lemma `cvd_upd_intro` states that if all writes to a shared variable `x` are covered with value `u` in the pre-state, and we perform an update operation on the same variable which writes value `v`, then all the writes to `x` in the post-state are covered with value `v`. Lemma `cvd_c_obs_transfer` states that if in the pre-state, we have a conditional observation and also we know that all writes to `x` are covered, then any update operation on `x` by thread `t` transfers the definite observation on `y` to thread `t`. The final lemma states that read operations preserve covered assertions. All the above lemmas have been proved in Isabelle.

6.2 Read–Copy–Update (RCU)

Our final non-trivial case study is a simplified RCU example [13], which comprises a writer that synchronises with a reader before deallocating a memory address. This example has been considered (using a pen-and-paper verification) by Lahav and Vafeiadis [26]. Our treatment (See Fig. 13) differs from that of Lahav and Vafeiadis [26] in several ways:

1. Our memory model allows relaxed reads and writes and thus contains less weak-memory synchronisation. The memory model of Lahav and Vafeiadis [26] only considers release-acquire accesses. Thus, we are able to validate that for RC11 RAR, it is sufficient for the writer and reader to synchronise via a single release-acquire synchronisation between the writer and reader.
2. We omit the use of an explicit stopper thread to terminate the reader by terminating the reader once it has signalled to the writer. This avoids the potential livelock present in [26], where a reader may be stopped before signalling to the writer, causing the writer to wait forever.[8]
3. Our proof is mechanised in Isabelle/HOL.
4. We only consider a single writer and reader pair. It is straightforward to see that the proof extends to triple readers since a writer synchronises with each additional reader using the same mechanism. In particular, to handle multiple readers, we would
   - use a set of shared reader variables (one for each reader),
   - each reader thread is a copy of the reader in Fig. 13, and publishes confirmation that it has seen the update to w using its own reader variable, and
   - the writer repeats the do-until loop for each additional reader, waiting for the corresponding reader variable to be set to 1.

In our case study, we consider two objects that are pointed to by n_1 and n_2. The writer wishes to deallocate one of n_1 and n_2, but only after ensuring that the reader is not going to access them via a synchronisation protocol that we describe below. The address to be deallocated is determined by a variable mb, initially $mb \in \{1, 2\}$. Namely, after writer–reader synchronisation, the element n_{mb} will be deallocated; this is represented in our algorithm (Fig. 13) by the assignment $n_{mb} := 0$. We require that the reader does not see the deallocated value, thus its postcondition is $a \neq 0$.

The writer-reader synchronisation mechanism works as follows. The writer determines the address that is not to be deallocated in m (initially mb) as the opposite of mb, then starts the synchronisation protocol by setting the flag w (initially 0) to 1 using a releasing write. It then waits for a reader signal by waiting for r (initially 0) to be set to 1.

The reader first reads m into a local register, then reads from either n_1 or n_2 (depending on the value of m that was read). It then signals the writer by reading from w (using an acquiring read) then sending the read value back to the writer by updating r (using a relaxed read). Since a reader reads the m without any synchronisation with the writer, it may terminate after reading from either n_{mb} or $n_{mb'}$, where $mb' = mb \mod 2 + 1$. Importantly, if it reads from n_{mb}, it does so before the writer has deallocated n_{mb}.

Our proof is supported by the following reader invariant, where the input wr_v corresponds to variable wr in the code, etc.

[8] We also have a version of the algorithm with a stopper thread [26], showing that its presence does not alter the safety property.

```
definition r_inv wrv mbv rr1v rr2v av s ≡
(∀ j. j ≠ 0 ∧ j ≠ 1 ⟶ [w ≉₂ j] s)
∧ (∀ j. j ≠ 1 ∧ j ≠ 2 ⟶ [m ≉₂ j] s)
∧ [w = 1]⟨ w =₂ 1⟩ s
∧ [w = 1]⟨ m =₂ (mbv mod 2 + 1)⟩ s
∧ (rr2v = 1 ⟶ [m =₂ (mbv mod 2 + 1)] s ∧ [w =₂ 1] s)
∧ ([w ≈₂ 1] s ⟶ [w =₁ 1] s)
∧ ([r ≈₁ 1] s ⟶ rr2v = 1)
∧ av ≠ 0
∧ (mbv = 1 ∨ mbv = 2)
∧ (([n1 ≉₂ 0] s ∧ [n2 ≉₂ 0] s ∧ wrv = 0) ∨
(rr2v = 1 ∧ (mbv = 1 ⟶ [n2 ≉₂ 0] s) ∧ (mbv = 2 ⟶ [n1 ≉₂ 0] s)))
```

The first two conjuncts limit the values of w and m that are possible for the reader to see. The third ensures that if the reader sees 1 for w, then it sees the last value of w, and the fourth conjuncts ensure that synchronisation over w ensures transfer of the new value for m. The fifth conjunct makes use of these facts and ensures that if the reader has seen 1 for w (i.e. $w = 1$), then it definitely sees values of m and w as written by the writer.

The sixth and seventh conjuncts relate reader views with the state of the writer. Namely, if the reader can see $w = 1$, then the writer's view of w is definitely 1. Moreover, if the writer can see that the reader signal $r = 1$, then the reader must have read 1 for w, i.e. $rr_2 = 1$.

The eight conjunct supports the reader's postcondition, while the ninth retains the fact that the only possible values of mb are 1 or 2 (as established by initialisation). The final conjunct is used to ensure that a reader does not read a deallocated value. It ensures that either the writer has not seen the reader's signal ($wr = 0$), and no deallocations have taken place or the reader thas seen the writer's signal ($rr2 = 1$) and regardless of the value of the mb, the opposite (i.e. mb') is guaranteed not to have been deallocated.

In the context of this invariant, a number of smaller local assertions must be introduced in the proof outline itself (see Fig. 13). For example, the branching in the reader establishes a stronger guard $rr_1 = 1$ or $rr_1 = 2$ depending on the branch taken.

7 Verification and Automation

In this section, we briefly discuss our experience with the verification strategy used in different verification tasks and also our empirical observations on proof effort.

For each of the algorithms described in the previous sections, we employ a generic verification strategy and outline the proof effort. After encoding the algorithm and the assertions, the main steps in validating the proof outlines are as follows:

1. First, we use the built-in `oghoare` tactic to reduce an Owicki–Gries proof outline into a set of basic Hoare logic proof obligations over weak memory pre-post state assertions. This tactic is exactly as developed by Nipkow and Nieto [30], and is used without change.
2. We pipe this output (which is a set of proof obligations on atomic commands) to the Isabelle simplifier, transforming the set-based representation of assertions by Nipkow and Nieto into a predicate-based representation.
3. We finally apply the Isabelle simplifier to all the generated sub-goals. This allows the lemmas for weak memory that we have adapted from [9] to be automatically matched with the proof obligations.

The above three steps can be performed using the following Isabelle `apply` command:

```
apply (oghoare;(simp add: Collect_conj_eq[symmetric]
```

```
||-
{| wr = 0 ∧ rr1 = 0 ∧ rr2 = 0∧ mb ∈ {1, 2} ∧ [m =1 mb] σ ∧ [m =2 mb] σ
∧ a ≠ 0 ∧ [n1 ≉2 0] σ ∧ [n2 ≉2 0] σ ∧ [w =1 0] σ ∧ [w =2 0] σ ∧ [r =1 0] σ |}
COBEGIN
  {| [w =1 0] σ ∧ [w ≉2 1] σ ∧ [m =1 mb] σ ∧ (∀j. j ∉ {0,1} ⟶ [r ≉1 j] σ) ∧ mb ∈ {1,2}|}
  < m :=1 (mb mod 2) + 1 σ> ;;
  {| [w =1 0] σ ∧ [w ≉2 1] σ ∧ [m =1 (mb mod 2 + 1)] σ
  ∧ (∀j. j ∉ {0,1} ⟶ [r ≉1 j] σ) ∧ mb ∈ {1,2} |}
  < w :=R1 1 σ > ;;
  DO {| (∀j. j ∉ {0,1} ⟶ [r ≉1 j] σ) ∧ mb ∈ {1,2} |}
   < wr ←1 r σ >
  UNTIL wr = 1
  INV {| (∀j. j ∉ {0,1} ⟶ [r ≉1 j] σ) ∧ mb ∈ {1,2} |}
  OD ;; {| wr = 1 ∧ mb ∈ {1,2} |}
  IF mb = 1
  THEN {| wr = 1 ∧ mb = 1 |} < n1 :=1 0 σ>
  ELSE {| wr = 1 ∧ mb = 2 |} < n2 :=1 0 σ>
  FI {| True |}
||
  {| (∀j. j ∉ {0,1} ⟶ [w ≉2 j] σ) ∧ (∀j. j ∉ {1,2} ⟶ [m ≉2 j] σ)
  ∧ [r ≉1 1] σ ∧ ([w ≈2 1] σ ⟶ [w =1 1] σ) ∧ [w = 1]⟨w =2 1⟩ σ
  ∧ [w = 1]⟨m =2 (mb mod 2 + 1)⟩ σ ∧ [n1 ≉2 0] σ ∧ [n2 ≉2 0] σ
  ∧ rr1 = 0 ∧ rr2 = 0 ∧ wr = 0 ∧ mb ∈ {1,2} ∧ a ≠ 0 |}
  WHILE rr2 = 0
  INV {| r_inv wr mb rr1 rr2 a σ |}
  DO {| r_inv wr mb rr1 rr2 a σ |}
   < rr1 ←2 m σ > ;;
   {| r_inv wr mb rr1 rr2 a σ ∧ ( rr2 = 1 ⟶ rr1 = ( mb mod 2) + 1 ) ∧ rr1 ∈ {1,2}|}
   IF rr1 = 1
   THEN {| r_inv wr mb rr1 rr2 a σ ∧ ( rr2 = 1 ⟶ rr1 = ( mb mod 2) + 1) ∧ rr1 = 1|}
     < a ←2 n1 σ >
   ELSE {| r_inv wr mb rr1 rr2 a σ ∧ ( rr2 = 1 ⟶ rr1 = ( mb mod 2) + 1 )∧ rr1 = 2|}
     < a ←2 n2 σ >
   FI ;;
   {| r_inv wr mb rr1 rr2 a σ |}
   < rr2 ←A2 w σ >;;
   {| r_inv wr mb rr1 rr2 a σ ∧ rr2 ∈ {0,1} |}
   < r :=2 rr2 σ >
  OD
  {| a ≠ 0 |}
COEND {| a ≠ 0 |}
```

Fig. 13 Proof outline for RCU under C11 for a single writer and reader

```
Collect_imp_eq[symmetric] Collect_disj_eq[symmetric]
Collect_mono_iff)?; clarify?, simp_all?)
```

The above command first invokes `oghoare` tactic to generate OG proof obligations (in the form of Isabelle subgoals) and then applies a series of simplification to each of the generated proof obligations.

Table 1 summarises the proof effort for all the examples given in this paper. For the simple litmus tests, the above command either discharge all the proof obligations, or leave a few (maximum 6) proof obligations unproved. These proof obligations require slightly more sophisticated application of the lemmas over weak memory state than can be discharged by the simplifier alone. However, they can be automatically discharged using Isabelle's built-in `sledgehammer` tool [7].

This verification strategy works equally well for Peterson's and RCU algorithms. Although these are larger examples that generate a significantly higher number of proof obligations. The `oghoare` tactic generates 258 subgoals for Peterson's and 182 for RCU, but over half of these are discharged by the above apply command. Although automatic, repeated applications of `sledgehammer` to discharge so many proof obligations is rather tedious. However, one can

Table 1 Size of algorithms, number of generated proof obligations, and the extent of automation

Algorithm	Algorithm and proof outline size (LOC)	Number of threads	POs generated by oghoare	Proof automation (%)	Approx. time (s)
Load buffering	15	2	24	100	2
Message passing	19	2	41	100	2
RRC 2	15	2	24	96	2
RRC 3	21	3	59	90	5
RRC 4	23	4	64	98	3
Two-way MP	57	2	72	20	11
Peterson's	80	2	258	65	89
RCU	64	2	182	65	70

quickly discover common patterns in the proof steps allowing these proof obligations to be discharged via a few simple applications of `apply`-style proofs.

Our set of available proof rules currently contains 80 proof rules. These proof rules are defined as Isabelle simplification rules. This means that these rules are available to the simplifier, and if it manages to match a proof obligation against it, it can automatically discharge that proof obligation. Our experience shows that if the Isabelle simplifier or `sledgehammer` tool fail to find a proof automatically, we either lack an essential rule in the proof rule set or the proof outline is not provable. If we manage to identify and prove a proof rule that can assist the automatic provers to discharge the proof obligation, then we will extend the proof rule set. However, if a proof rule is not found, we should consider the possibility of non-provable proof outline where either the precondition needs weakening or the postcondition needs strengthening.

Apart from the basic proof rules, most of the rules in our proof rule set are identified as a result of failure of the simplifier or `sledgehammer` in finding a proof for a proof obligation automatically. These failures have also led the development of proof outlines, where in most cases the generated proof obligations hinted that what is missing in the pre-condition.

Once the proofs were completed, we re-ran each proof and timed how long it takes Isabelle/HOL to replay each proof. These are given in the final column of Table 1. The reported times are for Isabelle 2020 on a 2018 Macbook Pro with a 2.3 GHz Quad-Core Intel Core i5 processor and 16GB memory. The times were recorded using a stopwatch, and hence are approximate.

8 Related Work

As has been mentioned, the current paper builds on ideas found in [14]. That paper did not develop a program logic based on Hoare triples, and was limited to invariance style proofs. Both [14] and the current paper use the same definite value assertion, but the current paper employs a much richer and more powerful assertion language. In particular, the conditional value assertion is critical for enabling an Owicki–Gries based program logic. Finally, [14] does not consider mechanisation or automation.

Of course, a great deal of work has been put into the development of separation logics for C11-style weak memory models [15,16,21,39,40]. One of the most recent and perhaps most

fundamental of these is the Iris framework [21]. This framework has been formalised in the Coq proof assistant, and instantiations of it support a large fragment of C11. This fragment contains C11's *nonatomic accesses* but not relaxed accesses and is therefore incomparable to our own. In particular, nonatomic access cannot legally race, whereas relaxed accesses are designed to enable racy code. More generally, separation logics can become complicated when applied to weak-memory, and we are partly motivated by the desire to build verification frameworks atop simple and natural relational models (other authors [26] have made similar observations).

There have been a number of recent attempts to develop mechanised deductive verification support for weak memory. Summers and Müller [36] present an approach to automating deductive verification for weak memory programs by encoding Relaxed Separation Logic [40] and Fenced Separation Logic [15,16] into Viper [29]. Their work builds on separation logic, whereas ours builds on a relational framework. Apart from this fundamental difference, Summers and Müller encode the concurrent logics into the Viper sequential specification framework, which provides a high level of automation. On the other hand, and as the authors themselves note, encoding the logic in a foundational verification tool such as Isabelle provides a higher level of assurance about correctness. In particular, the entire development of our framework is verified in Isabelle, down to the operational semantics.

Another technique based on Owicki–Gries is that of [26], which defines a proof system for the *release-acquire* fragment of C11, a smaller fragment than the *release-acquire-relaxed* fragment that we treat in this paper. It is unclear how difficult it would be to extend [26] to deal with relaxed accesses. In any case, [26] does not deal with mechanisation or automation.

Alglave and Cousot have developed another extension to the Owicki–Gries method for weak memory models [2]. Because their method, like ours, is an extension of the Owicki–Gries method, their verification method first requires a proof outline. One novelty of their approach is that their method requires a *communication specification* (or CS), which involves specifying for each read operation in the program, which writes the variable can read from (which may be in another thread). Verifying that the proof outline and CS are together valid, and therefore that assertions of the proof outline do in fact hold for the program, involves two proof obligations. The first is to show that the proof outline is valid in our standard sense (so that it is locally correct and noninterfering), under the assumption that the CS is satisfied. The second obligation is that a given memory model satisfies the CS. This second obligation constitutes an additional proof effort, not required in our method, since we have a fixed memory model. The advantage they gain is that once a proof outline is known to be locally correct and noninterfering under a given CS, then the algorithm is known to be correct under any memory model that satisfies the CS.

The operational semantics in the current paper is inspired by the semantics described in [21,34]. The current paper is based on semantics and assertions found in [9], which also presents case-study verifications mechanised in Isabelle. The mechanisation in that paper is rudimentary. Programs are not represented in a while-style language as in the current paper. Instead, they use a program-counter based representation, where control flow must be explicitly modelled. As a consequence, proof obligations are not decomposed in the conventional Owicki–Gries style. Rather, the verifier must prove stability of a large global invariant mapping program counter locations to the assertions that hold at that location. Furthermore, there is little real automation, either in generating or discharging proof obligations. The current paper presents a highly structured and mechanised Owicki–Gries framework supporting a high level of automation.

Dan et al. [11] introduce an abstraction for the store buffers of the weak memory model which reduces the workload on program analysers. They provide a source-to-source transfor-

mation that realises the abstraction producing a program that can be analysed with verifiers for sequential consistency. The approach is integrated with CONCURINTERPROC [20] and uses the Z3 theorem prover. *Model checking* has also been targeted for weak memory, e.g. by explicitly encoding architectural structures leading to weak behaviour, like store buffers [3,38]. Ponce de León et al. [18,35] 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 [4] (for TSO), NIDHUGG [1] (for TSO and PSO), RCMC [23] (for C11) and GENMC [24]. Others [8] present an approach for modelling and verifying C11 programs using Event-B and ProB model checker.

9 Conclusion

In this paper, we introduced the first deductive verification environment for C11 weak memory programs in Isabelle. Our contribution extends a twenty-year old formalisation of Owicki–Gries proof calculus in Isabelle [30] in order to tackle the verification problem of C11 programs under weak memory. We start by developing the necessary language support for defining C11 programs and have shown that existing operational semantics for the RC11-RAR fragment [14] can be encoded in a straightforward manner, which provides an example instantiation. We have developed a set of proof rules to facilitate verification of C11 programs. Proof rules are defined as Isabelle simplification rules so that the built-in simplifier can match the proof obligations against the rules and discharge them automatically. We provided a number of litmus test and illustrated different properties that can be proved using our tool. To showcase the effectiveness of our approach, we have also verified two larger case studies, Peterson's mutual exclusion and read–copy–update algorithms. We detailed the proof effort and showed that for most algorithms (even for the larger case studies) a good degree of automation (over 65%) is achieved.

Our entire development has been carried out in the Isabelle theorem prover and is modular with respect to the underlying memory model. For the RC11-RAR fragment we consider, we have shown that the proofs are highly automated. As described in Sect. 7, a simple pattern of applying an Owicki–Gries specific proof method, and then invoking SMT solvers via Isabelle's sledgehammer tool was sufficient for verifying every proof outline. Moreover, the use of Isabelle means that we have flexibility in the specific operational semantics that we use.

```
1  consts
2    x :: L
3
4  record mp_state =
5    a :: V
6    b :: V
7    σ :: Cstate
8
9  ||-
10 {| a = 0 ∧ b = 0 |}
11 COBEGIN
12 {| [x =1 0] σ ∧ [0x 1] σ ∧ [1x 1] σ ∧ [0x 2] σ ∧ [1x 1] σ |}
13   <x :=1 1 σ>;;
14 {| [x =1 1] σ ∧ [x =enc1 1] σ ∧ [0x 2]0 σ ∧ [1x 1] σ ∧ [1x 2] σ |}
15   <x :=1 2 σ>
16 {| [1x 1] σ ∧ [1x 2] σ ∧ [1 ≺x 2] σ|}
17 ||
18 {| [1x 1] σ ∧ [1x 2] σ |}
19   < a ←2 x σ> ;;
20 {| [x =enc2 a] σ |}
21   < b ←2 x σ>
22 {| (a ≠ b ⟶ [a ≺x b] σ) |}
23 COEND
24 {| a = 2 ⟶ b ≠ 1 |}
```

Fig. 14 Isabelle encoding of the read–read coherence litmus test with two threads

A Appendix

We present proofs of two additional variations of the RRC litmus test. Figure 14 presents a simple two-threaded version with a writer thread that enforces a order of writes in program order and a reader thread that reads from these writes. The proof shows that the order of reads in the reader is consistent with the order of writes in the writer.

Figure 15 presents the standard four-threaded version in which the two writes to x occur in two different threads. There are two reader threads both of which read from x twice. One must prove that the order of writes read by both threads are consistent. In particular, if a is set to 1 and b to 2, then thread 3 must have seen the writes to x in that order. It should therefore not be possible for thread 3 to read 1 for x if it has already seen the value 2 in the first read at line 29.

```
consts
  x :: L

record rrc4_state =
  a :: V
  b :: V
  c :: V
  d :: V
  σ :: Cstate

||-
{| [x =1 0] σ ∧ [x =2 0] σ ∧ [x =3 0] σ ∧ [x =4 0] σ |}
COBEGIN
{| [𝕆x 1]0 σ ∧ [𝟙x 1] σ |}
  <x :=1 1 σ>
{| True |}
||
{| [𝕆x 2]0 σ ∧ [𝟙x 2] σ |}
  <x :=2 2 σ>
{| True |}
||
{| True |}
  < a ←3 x σ> ;;
{| [x =^enc_3 a] σ ∧ [𝟙x 2] σ ∧ (a = 1 ⟶ [𝟙x a] σ) |}
  < b ←3 x σ>
{|(a ≠ b ⟶ [a ≺x b] σ) ∧ (a = 1 ⟶ [𝟙x a] σ) ∧ ( b = 2 ⟶ [𝟙x b] σ)|}
||
{| True |}
  < c ←4 x σ> ;;
{|[x =^enc_4 c] σ ∧ (c = 2 ⟶ [𝟙x c] σ)|}
  < d ←4 x σ>
{| (c ≠ d ⟶ [c ≺x d] σ) ∧ (c = 2 ⟶ [𝟙x c] σ) ∧ (d = 1 ⟶ [𝟙x d] σ)|}
COEND
{| a = 1 ∧ b = 2 ∧ c = 2 ⟶ d ≠ 1 |}
```

Fig. 15 Isabelle encoding of the read–read coherence litmus test with four threads

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. Alglave, J., Cousot, P.: Ogre and Pythia: an invariance proof method for weak consistency models. In: Castagna, G., Gordon, A.D. (eds.) POPL, pp. 3–18. ACM (2017)
3. 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)
4. 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)
5. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science. Springer, Berlin (2009)
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 55–66. ACM (2011)
7. Böhme, S., Nipkow, T.: Sledgehammer: judgement day. In: IJCAR, Lecture Notes in Computer Science, vol. 6173, pp. 107–121. Springer (2010)

8. Dalvandi, M., Dongol, B.: Towards deductive verification of C11 programs with Event-B and ProB. In: Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs, p. 4. ACM (2019)
9. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki–Gries reasoning for C11 RAR. In: Hirschfeld, R., Pape, T. (eds.) ECOOP, LIPIcs, vol. 166, pp. 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.11
10. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Isabelle files for "Integrating Owicki–Grieslinebreak for C11-style memory models into Isabelle/HOL" (2021). https://doi.org/10.6084/m9.figshare.14387201.v1. https://figshare.com/articles/software/Isabelle_Files_for_Integrating_Owicki-Gries_for_C11-Style_Memory_Models_into_Isabelle_HOL_/14387201
11. Dan, A., Meshman, Y., Vechev, M., Yahav, E.: Effective abstractions for verification under relaxed memory models. Comput. Lang. Syst. Struct. **47**, 62–76 (2017)
12. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods, Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
13. Desnoyers, M., McKenney, P.E., Stern, A.S., Dagenais, M.R., Walpole, J.: User-level implementations of read-copy update. IEEE Trans. Parallel Distrib. Syst. **23**(2), 375–382 (2012). https://doi.org/10.1109/TPDS.2011.159
14. 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)
15. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: VMCAI, LNCS, vol. 9583, pp. 413–430. Springer (2016)
16. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: ESOP, pp. 448–475 (2017)
17. Dolan, S., Sivaramakrishnan, K., Madhavapeddy, A.: Bounding data races in space and time. In: PLDI, PLDI 2018, pp. 242–255. ACM, New York, NY, USA (2018)
18. 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
19. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
20. Jeannet, B.: Relational interprocedural verification of concurrent programs. Softw. Syst. Model. **12**(2), 285–306 (2013)
21. 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)
22. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL, pp. 175–189. ACM (2017)
23. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–17:32 (2018)
24. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: McKinley, K.S., Fisher, K. (eds.) PLDI, pp. 96–110. ACM (2019)
25. Lahav, O.: Verification under causally consistent shared memory. SIGLOG News **6**(2), 43–56 (2019)
26. 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)
27. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, pp. 618–632. ACM (2017)
28. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
29. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: International Conference on Verification, Model Checking, and Abstract Interpretation, pp. 41–62. Springer (2016)
30. Nipkow, T., Nieto, L.P.: Owicki/Gries in Isabelle/HOL. In: FASE, Lecture Notes in Computer Science, vol. 1577, pp. 188–203. Springer (1999)
31. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
32. Paulson, L.C.: Isabelle—A Generic Theorem Prover (with a contribution by T. Nipkow), LNCS, vol. 828. Springer (1994)
33. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: Müller, P. (ed.) ESOP, LNCS, vol. 12075, pp. 599–625. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_22
34. Podkopaev, A., Sergey, I., Nanevski, A.: Operational aspects of C/C++ concurrency. CoRR abs/1606.01400 (2016) . https://arxiv.org/abs/1606.01400

35. 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)
36. 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)
37. 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)
38. 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)
39. 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)
40. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 867–884 (2013)
41. Williams, A.: https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html (2018). Accessed 20 June 2018