

Unifying Operational Weak Memory Verification: An Axiomatic Approach

SIMON DOHERTY, University of Sheffield, UK

SADEGH DALVANDI and BRIJESH DONGOL, University of Surrey, UK

HEIKE WEHRHEIM, University of Oldenburg, Germany

In this article, we propose an approach to program verification using an abstract characterisation of weak memory models. Our approach is based on a hierarchical axiom scheme that captures the *observational properties* of a memory model. In particular, we show that it is possible to prove correctness of a program with respect to a particular axiom scheme, and we show this proof to suffice for *any* memory model that satisfies the axioms. Our axiom scheme is developed using a characterisation of *weakest liberal preconditions* for weak memory. This characterisation naturally extends to Hoare logic and Owicki-Gries reasoning by lifting weakest liberal preconditions (defined over read/write events) to the level of programs. We study three memory models (SC, TSO, and RC11-RAR) as example instantiations of the axioms, then we demonstrate the applicability of our reasoning technique on a number of litmus tests. The majority of the proofs in this article are supported by mechanisation within Isabelle/HOL.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; *Concurrency*; **Hoare logic**; **Logic and verification**;

Additional Key Words and Phrases: Weak memory models, axiom hierarchy, verification, Hoare logic, Owicki-Gries, Isabelle/HOL

ACM Reference format:

Simon Doherty, Sadegh Dalvandi, Brijesh Dongol, and Heike Wehrheim. 2022. Unifying Operational Weak Memory Verification: An Axiomatic Approach. *ACM Trans. Comput. Logic* 23, 4, Article 27 (October 2022), 39 pages.

<https://doi.org/10.1145/3545117>

1 INTRODUCTION

The introduction and mass adoption of *weak memory models* (i.e., memory models weaker than *sequential consistency* [40]) has resulted in numerous program verification challenges. Here, a concurrent program's semantics is no longer faithfully modelled by an interleaving of threads, since

Doherty is supported by EPSRC Grant No. EP/R032351/1. Dalvandi and Dongol are supported by EPSRC Grant No. EP/R032556/1. Dongol is additionally supported by EPSRC Grant No. EP/V038915/1, EPSRC Grant No. EP/R025134/2, ARC Grant No. DP190102142, and VeTSS. Wehrheim is supported by DFG Grant No. WE 2290/14-1.

Authors' addresses: S. Doherty, University of Sheffield, Department of Computer Science, Sheffield, UK; email: s.doherty@sheffield.ac.uk; S. Dalvandi and B. Dongol, University of Surrey, Department of Computer Science, Guildford, UK; emails: {m.dalvandi, b.dongol}@surrey.ac.uk; H. Wehrheim, University of Oldenburg, Department of Computer Science, Oldenburg, Germany; email: heike.wehrheim@uni-oldenburg.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

1529-3785/2022/10-ART27 \$15.00

<https://doi.org/10.1145/3545117>

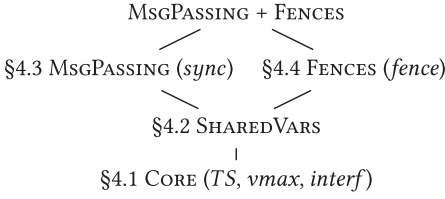


Fig. 1. Axiom hierarchy.

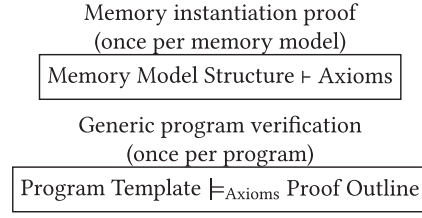


Fig. 2. Proof steps.

the effects of caching and compiler optimisations may cause the instructions within a thread to occur in an order different to the given sequential order. Therefore, many different semantics [6, 10, 14, 44, 46, 47, 54] and associated logics [19, 24, 33, 55, 56] have been developed to enable verification.

A popular style of semantics is the declarative style, which considers a complete (terminated) execution of a program, followed by a post-hoc check to determine whether the execution is allowable for the memory model [10, 14, 39]. However, it is more difficult to develop deductive verification techniques over such a semantics, since one cannot reason about a program in a step-wise manner. Another way to think about weak memory phenomena is by reasoning over each thread's view of the system [15, 17, 24, 33, 34], keeping in mind that a thread may not necessarily be up-to-date with writes of other threads. Different threads may have different views of a system, and hence, their reads may return different values. Thus, a thread may have access to (i.e., read from) writes that have already been overwritten by other threads in global real-time order.

Different memory models present different guarantees on the views that a thread may have. **Sequential consistency (SC)**, for instance, ensures that all threads see the most recent write to each variable. The **total store order (TSO)** model allows buffer-based reorderings [10, 47, 54], and hence ensures the most recent write is only available to the thread that performs the write (until its write buffer is flushed). In RC11-RAR,¹ knowledge of the last write is transferred from one thread to another using *release-acquire* synchronisation.

In light of these observations, this article presents a new approach to deductive verification in a weak memory setting with the aim of abstracting view-based reasoning. The key idea in our framework is an *axiomatic basis* that parametrically captures the key properties of several different memory models. We provide SC, TSO, and RC11-RAR as example memory models, showing that the methods (a) apply to more than one memory model and (b) can cover both hardware- and language-level weak memory. We organise our axioms in a hierarchical manner to capture the different guarantees provided by different memory models (see Figure 1). Each memory model can be seen as a particular instantiation of the axioms defined within this hierarchy.

Our axiomatic verification framework should not be confused with axiomatic semantics for weak memory (e.g., References [10, 13]). An analogy should instead be drawn with axiomatic descriptions of algebraic structures, e.g., semi-groups, monoids, groups, and so on, which also form a natural hierarchy. Such structures have been shown to be a mathematical basis for many different instances (models of computation), e.g., References [27, 31]. The benefit here is that any instantiation of an algebraic structure inherits all the properties (theorems) of that structure.


In this article, we show that it is possible to (a) develop such an axiomatic framework for weak memory, (b) prove that the axioms represent abstractions of several memory models, and (c) use

¹RC11-RAR is a restricted sub-language of the C11 memory model. RC11 refers to the repairing model by Lahav et al. [39], which preserves program order and disallows load-buffering (see Section 7.1). RAR refers to the fact that the memory model includes both *release-acquire* and *relaxed* atomics [14]. We discuss a potential technique for lifting the RC11 restriction in Section 8.

the abstract axioms in program verification. Moreover, the entire development can be encoded within the Isabelle/HOL proof assistant—in fact, Isabelle’s *locale* environment provides a convenient mechanism for encoding our axiomatic framework. Our methodology is outlined in Figure 2. For a given memory model, we perform a single proof to show that the corresponding memory model structure satisfies the axiom scheme. This means that any program verified correct with respect to the axiom scheme is guaranteed to be correct for the memory model.

A second key innovation in our approach is the use of the *weakest liberal precondition* predicate transformer as mechanism for reasoning about weak memory behaviour. This provides a previously unexplored basis for verification for relaxed memory models. In particular, the basic axioms in the existing Hoare logic for RC11-RAR [19] are derived directly from the weakest liberal precondition using the standard partial correctness encoding, i.e., $\{P\}S\{Q\} \Leftrightarrow P \subseteq \text{wlp}(S, Q)$. As we shall see, in our axioms (see **RW7** and **MP**), wlp provides a convenient mechanism for expressing preconditions where a thread is yet to perform an action. Moreover, since we establish generic Hoare logic proof rules over the axiom scheme, the rules hold for any instantiation of these axioms (including SC, TSO and RC11-RAR). Thus, by proving correctness w.r.t. an axiom scheme, a single proof is sufficient and directly carries over to all memory models instantiating the axioms.

Contributions. This article consists of the following main contributions.

- (i) A parameterised, hierarchical axiom scheme, defined in terms of weakest liberal precondition, that enables formalisation of multiple memory models.
- (ii) Instantiation of the axiom scheme to capture behaviours of multiple memory models—SC, TSO, and RC11-RAR.
- (iii) Verification of several example programs over the generic axiom scheme.
- (iv) Mechanisation of the framework, the memory model instantiations and generic verification within Isabelle/HOL. Our mechanisation may be downloaded from Reference [23]. We use the symbol  to indicate sections, lemmas and theorems that have been mechanised in Isabelle/HOL.

Note that we do not present all of the details of the memory models we consider (e.g., in RC11-RAR, we do not consider read-modify-writes or fences), since these details detract from the main contributions. These can be introduced using prior works. In particular, for RMWs, one can use the operational semantics of RC11-RAR with RMW updates [19, 24], and release-acquire fences can be encoded by a release-acquire RMW update to a special variable that is not used by the program [39]. We discuss the possibility of lifting program order restrictions (including in RC11-RAR) in Section 8. The purpose of this article is to introduce a generic abstract framework, based on wlp, and show how it enables (mechanised) program verification.

Article outline. We present an abstract language with a semantics parameterised by a memory model in Section 2. Here, we also provide SC and RC11-RAR as example instances for the parameters of the language. We motivate our approach and present the key concepts behind the axiomatisation in Section 3. Our axiomatic framework itself is presented in Section 4. Then in Section 5, we give the full instantiation for three memory models (SC, TSO, RC11-RAR) and detail the axioms they satisfy. In Section 6, we provide generic verification techniques over the axiom scheme, which we use to verify programs (Section 7).

2 AN ABSTRACT LANGUAGE FOR WEAK MEMORY

To make the upcoming discussion more concrete, we start by defining an abstract language and semantics. We assume that concurrent programs are written in a simple imperative programming

language that abstractly captures synchronisation primitives available to different memory models.

2.1 Syntax and Local Semantics

We assume each thread (from the set Tid) runs a sequential program. A thread may use *global* shared variables (from Var_G) and local variables (registers, from Var_L). We let $Var = Var_G \cup Var_L$ and assume $Var_G \cap Var_L = \emptyset$. We use x, y, z to range over global variables and r, r_1, r_2 to range over local variables. We assume that \ominus is a unary operator (e.g., \neg), \oplus is a binary operator (e.g., \wedge , $+$, $=$) and n is a value (of type Val). Expressions may only involve local variables.² The syntax of sequential programs, Com , is given by the following grammar (with $r \in Var_L, x \in Var_G$):

$$\begin{aligned} Exp_L &::= Val \mid Var_L \mid \ominus Exp_L \mid Exp_L \oplus Exp_L, \\ ACom &::= \mathbf{skip} \mid \mathbf{fnc} \mid r := Exp_L \mid x :=^{[WS]} Exp_L \mid r \leftarrow^{[RS]} x, \\ Com &::= ACom \mid Com; Com \mid \mathbf{if} B \mathbf{then} Com \mathbf{else} Com \mid \mathbf{while} B \mathbf{do} Com, \end{aligned}$$

where we assume B to be an expression of type Exp_L that evaluates to a Boolean. The notation $[WS]$ denotes that the annotation WS is optional (similarly, $[RS]$). These annotations are used to denote whether the corresponding action accesses the corresponding global variable using a *synchronisation mode*. We use $x :=^{[WS]} e$ and $r \leftarrow^{[RS]} x$ for writes to and reads from global variables, possibly annotated to indicate synchronisation.

The language foresees two synchronisation primitives: annotations of reads and writes (as discussed above) as well as fences (**fnc**). We use fences from our abstract language to support the store-buffering example (see Section 7.3) [10]. Other fence abstractions, e.g., as used in C11 can be defined, but we elide such details in this article.

For simplicity, we assume concurrency at the top level only.³ We use a function $\Pi : Tid \rightarrow Com$ to model a program comprising multiple threads. In examples, we typically write concurrent programs as $C_1 \parallel \dots \parallel C_n$, where $C_i \in Com$. We further assume some initialisation of variables as defined by **Init**. The structure of our programs thus is **Init**; $(C_1 \parallel \dots \parallel C_n)$.

Semantics. The semantics of this language is split into two parts: a *local* part, detailing interactions between language constructs, local variables (registers) and shared-memory actions, and a *global* part, detailing the interaction between shared-memory actions and shared memory. These are later reconciled to form a *combined semantics*. The local semantics is common across the example instantiations of the abstract language that we consider, but the global semantics differs between different memory models. Below, we treat the global semantics as a parameter, which allows us to obtain a generic combined operational semantics.

Local Semantics. The local semantics simply assumes that *any* value can be read from shared variables. These values are later constrained in the combined semantics.

It hence only tracks the values of local variables. We first fix the set of *abstract actions*

$$Act \triangleq \bigcup_{x \in Var_G, r \in Var_L, n \in Val} \{rd^{[RS]}(x, r, n), wr^{[WS]}(x, n), r := n\} \cup \{fence\},$$

containing actions for reads and writes (potentially annotated), assignment to local registers, and fences. We furthermore employ a silent τ action and let $Act_\tau \triangleq Act \cup \{\tau\}$. Note that the read actions keep track of the corresponding local register r . Later, in Section 6, we develop a program logic for our abstract language, where it becomes important to know how the local state is being modified.

²For a treatment of expressions with global variables in the semantics, see Reference [24].

³This restriction is common in the weak-memory literature, and can be generalised to support, e.g., fork/join parallelism.

$$\begin{array}{c}
\frac{r \in \text{Var}_L \quad n = \llbracket E \rrbracket_{ls}}{(r := E, ls) \xrightarrow{\tau} (\mathbf{skip}, ls[r := n])} \quad \frac{a = wr^{[WS]}(x, n) \quad x \in \text{Var}_G \quad n = \llbracket E \rrbracket_{ls}}{(x :=^{[WS]} E, ls) \xrightarrow{a} (\mathbf{skip}, ls)} \quad \frac{a = rd^{[RS]}(x, r, n) \quad n \in \text{Val}}{(r \leftarrow^{[RS]} x, ls) \xrightarrow{a} (\mathbf{skip}, ls[r := n])} \\
\\
\frac{a = \text{fence}}{(\mathbf{fnc}, ls) \xrightarrow{a} (\mathbf{skip}, ls)} \quad \frac{(C_1, ls) \xrightarrow{a} (C'_1, ls')}{(C_1; C_2, ls) \xrightarrow{a} (C'_1; C_2, ls')} \quad \frac{}{(\mathbf{skip}; C_2, ls) \xrightarrow{\tau} (C_2, ls)} \\
\\
\frac{\llbracket B \rrbracket_{ls}}{(IF, ls) \xrightarrow{\tau} (C_1, ls)} \quad \frac{\neg \llbracket B \rrbracket_{ls}}{(IF, ls) \xrightarrow{\tau} (C_2, ls)} \quad \frac{\llbracket B \rrbracket_{ls}}{(WH, ls) \xrightarrow{\tau} (C; WH, ls)} \\
\\
\frac{\neg \llbracket B \rrbracket_{ls}}{(WH, ls) \xrightarrow{\tau} (\mathbf{skip}, ls)} \quad \text{PAR} \frac{(\Pi(t), lst(t)) \xrightarrow{a} (C, ls) \quad a \in \text{Act}_\tau}{(\Pi, lst) \xrightarrow{a}_t (\Pi[t := C], lst[t := ls])}
\end{array}$$

Fig. 3. Local program semantics where $IF \triangleq \text{if } B \text{ then } C_1 \text{ else } C_2$ and $WH \triangleq \text{while } B \text{ do } C$.

The rules of Figure 3 define the operational semantics of the language. We assume a function $lst \in \text{Tid} \rightarrow (\text{Var}_L \rightarrow \text{Val})$ (\rightarrow 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 $\text{dom}(lst(t)) \cap \text{dom}(lst(t')) = \emptyset$. For an expression E over local variables, we write $\llbracket E \rrbracket_{ls}$ for the value of E in local state $ls \in (\text{Var}_L \rightarrow \text{Val})$; we write $ls[r := n]$ to state that ls remains unchanged except for the value of local variable r , which becomes n . The rule PAR lifts the steps of sequential programs to a concurrent program $\Pi : \text{Tid} \rightarrow \text{Com}$.

Combined Semantics. The combined semantics is defined with respect to a transition relation $\sigma \xrightarrow{\text{memop}(a), t} \sigma'$, where σ and σ' are states over global variables (in Var_G), a is an action generated by the local semantics in Figure 3, $\text{memop}(a)$ returns a concrete action (for a specific memory model) corresponding to the given abstract action a , and $t \in \text{Tid}$.

We provide example instantiations of these parameters (σ, \rightarrow and memop) for memory models SC and RC11-RAR (partially) in Section 2.2 and for TSO and the remaining part of RC11-RAR in Section 5.2. For now, we assume the existence of these parameters to define the combined semantics of the abstract language above as follows:

$$\begin{array}{c}
\text{SILENT} \frac{(\Pi, lst) \xrightarrow{\tau}_t (\Pi', lst')}{(\Pi, lst, \sigma) \Rightarrow_t (\Pi', lst', \sigma)}, \quad \text{LOCAL} \frac{(\Pi, lst) \xrightarrow{r:=n}_t (\Pi', lst')}{(\Pi, lst, \sigma) \xRightarrow{r:=n}_t (\Pi', lst', \sigma)}, \\
\\
\text{MEMORY} \frac{(\Pi, lst) \xrightarrow{a}_t (\Pi', lst') \quad a \neq (r := n) \quad \sigma \xrightarrow{\text{memop}(a), t} \sigma'}{(\Pi, lst, \sigma) \xRightarrow{a}_t (\Pi', lst', \sigma')}.
\end{array}$$

Auxiliary Variables. In proofs, we must sometimes introduce auxiliary variables to record the history of program executions. To this end, we assume a set of auxiliary variables Var_A such that $\text{Var}_A \cap \text{Var} = \emptyset$, an auxiliary state of type $\Sigma_A \triangleq \text{Var}_A \rightarrow \text{Val}$, and syntax $\langle C; x_a := E_a \rangle$, where $x_a \in \text{Var}_A$, $C \in \text{ACom}$, E_a is an expression over $\text{Var}_A \cup \text{Var}_L$ and $\langle \rangle$ are used to denote that C and $x_a := E_a$ are executed in a single step. This ensures that auxiliary variables do not interfere with the normal program execution. In the presence of auxiliary variables, the rules for atomic commands C in Figure 3 are replaced by corresponding rules of the form, where $as \in \Sigma_A$:

$$\text{AuxVar} \frac{(C, ls) \xrightarrow{a} (\mathbf{skip}, ls') \quad n = \llbracket E_a \rrbracket_{ls \cup as}}{\langle C; x_a := E_a \rangle, ls \cup as \xrightarrow{a} (\mathbf{skip}, ls' \cup as[x_a := n])}.$$

$$\begin{array}{c}
\text{SC-READ} \frac{a = rd(x, n) \quad n = \sigma(x)}{\sigma \xrightarrow{a.t}_{SC} \sigma} \quad \text{SC-WRITE} \frac{a = wr(x, n)}{\sigma \xrightarrow{a.t}_{SC} \sigma[x := n]} \quad \text{SC-SKIP} \frac{a = skip}{\sigma \xrightarrow{a.t}_{SC} \sigma}
\end{array}$$

Fig. 4. Memory model semantics for SC.

This basic rule is lifted to compound statements and programs in the standard manner to allow auxiliary variables within a program.

2.2 Global Semantics and Example Instantiations

We now present example languages and memory models that instantiate our generic scheme. These examples serve two purposes: First they provide examples of the parameters alluded to above, and second they introduce the concepts of *views* and *view synchronisation*, which we axiomatise abstractly in the remainder of the article.

Example (SC). Our first example is the **sequential consistency (SC)** memory model [40]. Here, states have type $\Sigma_{SC} \triangleq \text{Var} \rightarrow \text{Val}$, i.e., are functions from variables to values. The set of actions for SC are given by $\text{Act}_{SC} \triangleq \{rd(x, n), wr(x, n), skip \mid x \in \text{Var}_G \wedge n \in \text{Val}\}$ and $\text{memop} : \text{Act} \rightarrow \text{Act}_{SC}$ is a partial function such that $\text{memop}(rd^{[RS]}(x, r, n)) = rd(x, n)$, $\text{memop}(wr^{[WS]}(x, n)) = wr(x, n)$, and $\text{memop}(fence) = skip$. Note that in SC, the strong synchronisation requirements of fences are already guaranteed by *skip* statements. Finally, the (standard) transition rules are given in Figure 4.

Example (RC11-RAR). Our second example is the RC11-RAR memory model, which is a sub-language of C11 that contains *relaxed* as well as *releasing* writes (annotation R) and *acquiring* reads (annotation A). Release and acquire are specific synchronisation features for programmers that can be used to regain some form of shared knowledge about global variables. Atomic commands on global variables generate the set of actions⁴:

$$\text{Act}_{C11} \triangleq \bigcup_{x \in \text{Var}_G, n \in \text{Val}} \{rd^{[A]}(x, n), wr^{[R]}(x, n)\}.$$

In RC11-RAR, release and acquire implement write and read synchronisation, respectively, i.e., $\text{memop}(rd(x, r, n)) = rd(x, n)$, $\text{memop}(rd^{RS}(x, r, n)) = rd^A(x, n)$, $\text{memop}(wr(x, n)) = wr(x, n)$ and $\text{memop}(wr^{WS}(x, n)) = wr^R(x, n)$. We define the write and read actions as follows:

$$Wr \triangleq \{wr^{[R]}(x, n) \mid x \in \text{Var}_G \wedge n \in \text{Val}\},$$

$$Rd \triangleq \{rd^{[A]}(x, n) \mid x \in \text{Var}_G \wedge n \in \text{Val}\}.$$

States of RC11-RAR are typically represented by structures such as graphs of events [24] or functions over timestamped writes [19, 33, 34]. Below, we present the recent operational semantics of Dalvandi et al. [19], based on the timestamped writes, that has been shown to coincide with the declarative graph-based semantics of Lahav et al. [39]. Here, states are of type $\Sigma_{C11} \triangleq \text{TView} \times \text{MView} \times \text{Writes}$, where assuming \mathbb{Q} is the set of rational numbers (used for timestamps), we have

$$\text{TView} \triangleq \text{Tid} \rightarrow (\text{Var}_G \rightarrow (Wr \times \mathbb{Q})),$$

$$\text{Writes} \triangleq 2^{Wr \times \mathbb{Q}},$$

$$\text{MView} \triangleq (Wr \times \mathbb{Q}) \rightarrow (\text{Var}_G \rightarrow (Wr \times \mathbb{Q})).$$

We will give the instantiation of the transition relation in Section 5.3 as it requires several more definitions. Here, we informally explain the state components. First, the set of writes that have occurred thus far is recorded within the state component *Writes*. Every write is equipped with a

⁴Fences are not part of RC11-RAR.

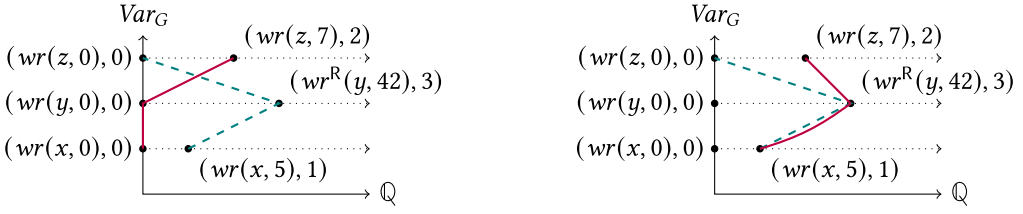


Fig. 5. Illustration of views and view updates: pre-state (left) and post-state (right) after executing $rd^A(y, 42)$ by thread t_1 (purple).

timestamp. For each variable, the timestamp of each write is unique, and hence they define a total order.

Contrary to SC, where all threads always see the most recent write, in RC11-RAR, each thread has its own *view* of the writes to each variable. This is encoded within state component *TView*, aka *thread view*. Figure 5 illustrates the views of two threads t_1 (purple, solid line) and t_2 (teal, dashed line) for three global variables x , y , and z in two different states. The x-axis is used for the timestamps; timestamp 0 contains the initialising writes $wr(x, 0)$, $wr(y, 0)$, and $wr(z, 0)$. We see that in the state depicted on the left, the views of threads t_1 and t_2 differ for all variables.

The third component *MView* (aka *modification view*) is required to formalise the semantics of synchronisation primitives A (acquire) and R (release). When an acquiring read action reads the value written by a releasing write, a form of *view transfer* from one thread to another happens. To define this view transfer, each write event w stores in *MView* its thread's view of the (other) writes when w occurs. An acquiring read reading from a releasing write induces a view update, in which the reading thread's view is updated to (at least) the view of the releasing write.

For illustration, consider again the picture in Figure 5. Starting with the thread's views on the left-hand side, we assume that thread t_1 (purple) performs an action $rd^A(y, 42)$, thereby reading the value written by $w = (wr^R(y, 42), 3)$ of thread t_2 (teal). This is a release-acquire synchronisation, thus it updates thread t_1 's view of y as well as its view of x to that of t_2 (to be precise: the current viewfront of t_1 is combined with the modification view of w). This then gives rise to the state on the right-hand side of Figure 5.

3 OVERVIEW AND MOTIVATION

As we have seen in Section 2.2, there are different instantiations of our abstract language in Section 2.1. More importantly, in the presence of weak memory, the complexity of the models can increase quickly even when focussing one's attention to limited fragments of a memory model. Our work is thus motivated by a simple question: *Is there a way to abstractly capture memory model phenomena so that programs can be proved correct directly over the abstract language?* The presence of such a technique would mean that a program verifier need not master the intricacies of the low-level memory models. For any language (and associated memory model) that has been shown to satisfy our high-level axioms, verification can focus on the abstract language and semantics, whose proofs *automatically apply* to the low-level language.

In this section, we present an overview of our framework, which is designed to answer the question above. The main idea behind the framework is the observation that it is possible to relax SC along three main dimensions (Section 3.1). Then in Section 3.2, we discuss the implications of these relaxations with respect to the message passing example. We also discuss how an abstract proof carries over to implementation memory models such as SC, TSO, and RC11-RAR.

3.1 Relaxing SC

We start by recalling three important properties guaranteed by SC. First, SC ensures that all threads have the same *view* of the shared state. Second, SC ensures that operations on distinct variables are *independent* (i.e., operations on different variables commute), and read operations do not change the state (i.e., are side-effect free). Third, SC ensures *global synchronisation* across threads so that all reads to a variable read from the last write to that variable. These properties are not guaranteed by weak memory models. To accommodate this, our framework relaxes each of the guarantees of SC as follows.

3.1.1 Weakening Views. Our first weakening is to abandon the idea of a *global* view, agreed upon by all threads, in favour of a set of per-thread local views. Intuitively, the view of a thread defines which writes on each variable are currently known to the thread, or in other words, how much of *modification order* [10, 14, 39] the thread has observed. As a thread might see one of several writes, it has a certain amount of nondeterminism as to the current value of a variable. A thread can *advance* its view by “encountering” more of the writes in the system. Thereby a thread reduces its nondeterminism. For operational models in which writes can be flushed from per-thread buffers (such as TSO), we can regard these flush actions as advancing threads’ views. A thread with the maximal knowledge about the writes to a variable (i.e., the least degree of nondeterminism) is called *view-maximal* w.r.t. that variable.

3.1.2 Read/Write Independence. Reasoning under SC is simplified by the fact that it has the two useful features, described below. One or both of these may be invalid under standard view-based operational semantics for weak memory [17, 19, 33, 34, 52].

(Semi-)commutation. In SC, operations such as a read or write acting on different variables are *independent*, and hence can be reordered. For example, if $\llbracket C \rrbracket$ returns the pre/post state relation corresponding to command C , and x, y are shared variables where $x \neq y$, then $\llbracket \text{read}(x); \text{write}(y, 42) \rrbracket = \llbracket \text{write}(y, 42); \text{read}(x) \rrbracket$. In standard (view-based) weak-memory operational semantics, the execution of a read action can change the outcome of a subsequent operation on a different variable (as the read may change a thread’s view), which causes (semi-)commutativity to fail.

Read Invisibility. In SC, reads do not change the state, i.e., $\llbracket \text{read}(x) \rrbracket \subseteq \text{id}$ for any x , where id is the identity relation over states. In standard (view-based) weak-memory operational semantics, a read can cause the state representing the view of the reading thread to change. Thus, reads can have a side-effect, and hence, do not produce subrelations of id .

In our framework, we recover weakenings of properties of this kind using a notion of *interference of an action*, which specifies the effect of that action that is visible to other threads. As described later (Section 4), interference can be defined so that read actions are mapped to an effect that does not change the state of any other thread, and write operations are mapped to an effect that only changes the shared state. When we do this, the interference of each action satisfies the desirable semi-commutation and read-invisibility properties possessed by actions under SC.

Aside. An alternative⁵ would be to assume that the concrete memory models contain a separate (internal) transition that advances the view and reads return the value at the current view (without advancing the view). This would allow the semi-commutation and read invisibility properties to once again be satisfied, since the reads no longer contain a side effect of shifting the view forward. However, this formulation would induce alternative axioms that abstractly describe legal “view

⁵We thank one of our TOCL reviewers for this suggestion.

Init: $d := 0; f := 0;$

Thread 1 \parallel **Thread 2**
 $1 : d := 5;$ $3 : \text{do } r_1 \leftarrow^{\text{RS}} f$
 $2 : f :=^{\text{WS}} 1;$ $\quad \text{until } r_1 = 1;$
 $\quad 4 : r_2 \leftarrow d;$
 $\{r_2 = 5\}$

Fig. 6. Message passing (MP).

Init: $d := 0; f := 0;$

Thread 1 \parallel **Thread 2**
 $1 : d := 5;$ $3 : \text{do } r_1 \leftarrow f$
 $2 : f := 1;$ $\quad \text{until } r_1 = 1;$
 $\quad 4 : r_2 \leftarrow d;$
 $\{r_2 = 5\}$

Fig. 7. MP for SC and TSO.

Init: $d := 0; f := 0;$

Thread 1 \parallel **Thread 2**
 $1 : d := 5;$ $3 : \text{do } r_1 \leftarrow^{\text{A}} f$
 $2 : f :=^{\text{R}} 1;$ $\quad \text{until } r_1 = 1;$
 $\quad 4 : r_2 \leftarrow d;$
 $\{r_2 = 5\}$

Fig. 8. MP for RC11-RAR.

shifts” for each memory model and their interactions with synchronising operations (e.g., fences and release-acquire synchronisation) and *view maximality* (described below). Moreover, one would require a separate proof that the concrete operational semantics with an independent view shift is equivalent to its corresponding standard semantics. Pursuing such an algebraic formulation would be an interesting direction for future work.

3.1.3 Weakening Synchronisation Operations. In SC, all threads have maximal determinism: all operations have precisely one possible outcome and the value of each read is determined by the most recent write to that variable. We refer to this property as *view maximality*. In weak memory, this is no longer the case and reads may read from stale writes. To address this, a programmer must use explicit synchronisation constructs such as fences or release/acquire annotations to ensure appropriate orderings and visibility guarantees.

In our framework, we characterise the behaviour of such constructs using a notion of *transfer* of view maximality from one thread to another. Different synchronisation constructs and memory models induce different view transfer properties. Our framework provides a program syntax with a simple *abstract* annotation scheme. Read and write operations can be annotated with RS and WS annotations, respectively, specifying that the operation should provide synchronisation guarantees that transfer view maximality.

3.2 Example: Message Passing Synchronisation

To make these concepts clearer, we present a simple example. The concurrent program in Figure 6 is the so called **message passing (MP)** litmus test, where reading f ’s value as 1 by thread 2 causes the “message” of $d = 5$ to be passed from thread 1 to 2, ensuring the postcondition $r_2 = 5$.

In our axiomatisation, we think of the message passing as a form of *view transfer* from thread 1 to thread 2. Due to weak memory, view transfer may not occur for every read and write; thus the write at line 2 and the read at line 3 include WS and RS annotations, specifying that these operations should *synchronise* to support message passing. We “compile” this annotation differently depending on the memory model. For the memory models SC and TSO [47, 54] bare reads and writes are sufficient, so we can simply strip the annotation (see Figure 7). For C11, the write must be *releasing* and the read *acquiring* (see Figure 8).

However, we do not wish to reprove the same example for each concrete weak memory model of the program template in Figure 6. Instead, we develop an abstract reasoning framework directly for Figure 6, using the relaxation principles described in Section 3.1 to capture weak memory effects. The abstract framework is defined by a set of axioms (see Section 4) based on weakest liberal preconditions, which gives rise to a proof outline (Figure 13) containing abstract assertions that can be validated using a standard Owicki-Gries [48] verification framework (Section 6). This proof is *inherited*, for free, by *any* memory model that instantiates the axiom scheme used by the abstract proof. We show that the axiom scheme can be arranged into a hierarchy (Figure 1) to distinguish properties available to different concrete memory models.

4 UNIFYING SC AND WEAK MEMORY REASONING

In this section, we define our axiomatic framework for unifying reasoning about concurrent programs under both SC and weak memory. In particular, we re-establish SC proof principles in the context of a *view-based* model. Note that all of the theory in this section has been mechanised [23].

We aim for an *operational* semantics, thus our starting point is a generic *transition system* $TS \triangleq (\text{Act}, \Sigma, I, T)$, where Act is a set of actions, Σ is a set of states, $I \subseteq \Sigma$ is a set of initial states and $T \in \text{Tid} \times \text{Act} \rightarrow 2^{\Sigma \times \Sigma}$ defines a set of transitions. Further structure on the actions will be defined in Section 4.2. For a set of actions $A \subseteq \text{Act}$, we define $T(t, A) \triangleq \bigcup_{a \in A} T(t, a)$.

We present a family of axiomatisations over transition systems (see Figure 1), arranged in a hierarchy that is analogous to an algebraic hierarchy (e.g., monoids, groups, rings). Each level specifies a set of axioms, which capture aspects of the reasoning capabilities of the memory models that satisfy those axioms. A *memory model structure*, defined below, describes the signature of each axiomatisation.

Definition 4.1. A *memory model structure* is a tuple (TS, m_1, \dots, m_n) , where TS is a transition system and m_1, \dots, m_n are functions over TS and Tid .

The functions m_1, \dots, m_n are analogous to algebraic operations in a conventional algebraic hierarchy. Axioms define constraints on these functions and the transition system. Thus, the transition system TS and each function m_i are *parameters* of the axiomatisation that the concrete memory models instantiate.

Our axiomatisations make use of the *weakest liberal precondition transformer*, as a basis for property specification and verification. For a relation R and set of states P (representing a predicate), let $\text{wlp} : 2^{\Sigma \times \Sigma} \times 2^\Sigma \rightarrow 2^\Sigma$, with

$$\text{wlp}(R, P) \triangleq \{\sigma \in \Sigma \mid \forall \sigma'. (\sigma, \sigma') \in R \Rightarrow \sigma' \in P\}.$$

All of the standard wlp properties, such as (anti-)monotonicity, composition, conjunctivity, etc hold. In this work, R is typically instantiated to the relation $T(t, a)$, where T is the transition relation, t is a thread and a is an action. We say R is *disabled* in a state σ iff $\sigma \in \text{dis}(R)$ holds, where $\text{dis}(R) \triangleq \text{wlp}(R, \emptyset)$.

With these basics in place, we now define our axiom hierarchy. Roughly speaking, the *core* axiomatisation (Section 4.1) addresses weakening in general; the *shared variables* axiomatisation (Section 4.2) addresses weakening read/write properties (Section 3.1.2); and the axiomatisations *message passing* (Section 4.3) and *fences* (Section 4.4) address weakening synchronisation (Section 3.1.3).

4.1 The CORE Axiomatisation

This level contains axioms that formalise the basics of our approach. Memory model structures that satisfy the core axiomatisation have the form $(TS, \text{vmax}, \text{interf})$, where TS is a transition system, $\text{vmax} \in \text{Tid} \times \text{Act} \rightarrow 2^\Sigma$ is the *view maximality* predicate (see Section 3.1.1) and $\text{interf} \in \text{Tid} \times \text{Act} \rightarrow 2^{\Sigma \times \Sigma}$ the *interference* relation (see Section 3.1.2). We first describe vmax and explain interf below.⁶

Intuitively, in states satisfying $\text{vmax}(t, a)$, thread t has “at least as much determinism” in its execution of the action a as any other thread. In a conventional memory model, where the actions are reads and writes on shared variables we might think of states satisfying $\text{vmax}(t, a)$, as being

⁶Although our framework is formalised in terms of view maximality, further generalisations can be obtained by relaxing vmax throughout this section with a general predicate over threads and actions, thus allowing synchronisation and transfer over more general properties.

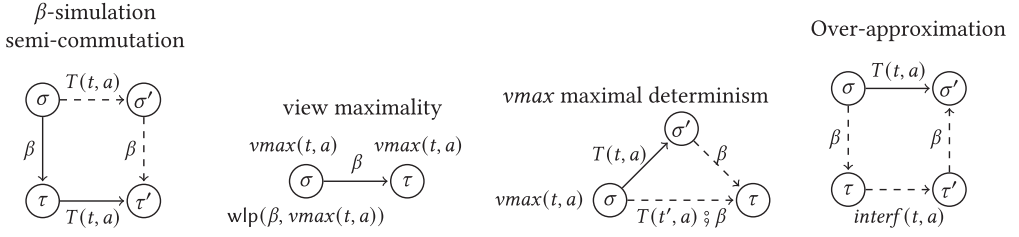


Fig. 9. Visualisations of β -simulation semi-commutation, β -simulation view-maximality, and axioms **C2** and **C3**, where dashed arrows denote existential quantification.

those where t 's view is “the most up-to-date” (and hence the least nondeterministic) w.r.t. the variable that a reads from. The axioms over $vmax$ and $interf$ rely on the concept of a *view preserving simulation*. Let \circledcirc denote relational composition.

Definition 4.2. For a transition system $(\text{Act}, \Sigma, I, T)$, *view-preserving simulation*, denoted β , is the weakest relation R satisfying the following, for all threads t and actions $a \in \text{Act}$:

$$\begin{aligned} R \circledcirc T(t, a) &\subseteq T(t, a) \circledcirc R && \text{(semi-commutation)} \\ vmax(t, a) &\subseteq wlp(R, vmax(t, a)) && \text{(view maximality)} \end{aligned}$$

Since the view-preserving simulation relation is fixed, we also refer to a view-preserving simulation as a β -simulation. It is straightforward to check that satisfaction of these properties is closed under arbitrary unions, so β is well-defined. Furthermore, β is a preorder (i.e., reflexive and transitive). When $(\sigma, \tau) \in \beta$ and $\sigma \neq \tau$, we think of τ as being “more deterministic” than σ .

Technically, a β -simulation is a standard simulation relation, that additionally preserves view maximality. However, unlike simulation relations that are used to prove refinement (which relate states of two different systems), β -simulations are over the same state space and their purpose is to describe how views “move forward.”

A visualisation of ([semi-commutation](#)) is given in Figure 9. It ensures that for any state σ , if it is possible to perform a β -simulation step (reducing non-determinism), and then for t to execute a to reach a state τ' , then it is also possible for t to execute a , then perform a β -simulation step and reach τ' . Note that the commutation does not necessarily hold in the other direction, i.e., $T(t, a) \circledcirc \beta \subseteq \beta \circledcirc T(t, a)$ may not hold, since β might reduce non-determinism in a way that makes the transitions in $T(t, a)$ impossible. We also visualise ([view maximality](#)) in Figure 9. If t is already view maximal on a , then since $wlp(\beta, vmax(t, a))$ holds, performing a β step will result in a state satisfying $vmax(t, a)$.

We can now state the first two axioms of the CORE axiomatisation, which contain the key constraints on the $vmax$ predicate.

C1 ($vmax$ initialisation). For all $t \in \text{Tid}$ and $a \in \text{Act}$,
 $I \subseteq vmax(t, a)$.

C2 ($vmax$ maximal determinism). For all $\sigma, \sigma' \in \Sigma$, $t, t' \in \text{Tid}$ and $a \in \text{Act}$, we have
 $\sigma \in vmax(t, a) \wedge (\sigma, \sigma') \in T(t, a) \Rightarrow \exists \tau \in \Sigma. (\sigma', \tau) \in \beta \wedge (\sigma, \tau) \in T(t', a) \circledcirc \beta$.

Axiom **C1** states that in initial states of the transition system TS , all threads must be view maximal w.r.t. all actions. In an operational model featuring flush actions on thread buffers (e.g., TSO), this corresponds to saying that initially all buffers are empty. For models such as RC11-RAR, this corresponds to saying that all threads are synchronised with the last write on each variable.

Axiom **C2** (visualised in Figure 9) formalises the idea that $vmax(t, a)$ is true in states where t can only read from the latest write to the variable of a . Note that any other thread t' can also read from the latest write, and thus we can think of t as having “at least as much determinism” in its execution of a as t' whenever $vmax(t, a)$ holds. Axiom **C2** states that we can achieve the effect of t 's execution of a (followed by view updates) by having t' execute a and then advancing the views of other threads (including t).

In Section 3.1, we indicated that our framework involves weakening the semi-commutation and read-invisibility properties of SC using a notion of *interference*. Our key tool here is the labelled interference relation *interf* (see also Section 3.1.2). In the memory models we consider (i.e., SC, TSO and RC11-RAR), for any thread t and action a , if $(\sigma, \sigma') \in \text{interf}(t, a)$ and $\sigma \neq \sigma'$ then a is a write action. More generally, at the SharedVars level (see Section 4.2), *interf* accommodates the introduction of a new value (e.g., via a write) to the state that other actions (e.g., a read) can see.

The final two axioms of the CORE axiomatisation describe the requirements on *interf*.

C3 (Over-approximation). For all $t \in \text{Tid}$ and $a \in \text{Act}$,

$$T(t, a) \subseteq \beta \circ \text{interf}(t, a) \circ \beta.$$

C4 (View stability). For all $t \in \text{Tid}$ and $a, b \in \text{Act}$,

$$vmax(t, a) \subseteq \text{wlp}(\text{interf}(t, b), vmax(t, a)).$$

Axiom **C3** states that *interf* together with β over-approximates the behaviour of the given action, i.e., if transition $T(t, a)$ is possible, then we must also be able to possibly perform some view updates, perform *interf*(t, a), possibly followed by more view updates.⁷ Note that we could remove the left-most occurrence of β for SC, TSO, and RC11-RAR, but this would only result in a stronger axiom with fewer models, and no *useful* additional power in the axiom.

Axiom **C4** is analogous to (view maximality) and states that the interference relation preserves every view-maximality property of the thread performing the interference. Together **C3** and **C4** ensure that a thread t can never “lose” view maximality by executing an action.

In Section 7, we show how to use our framework in Owicki-Gries [48] style proofs where *global correctness* is a critical proof step: We must show that each assertion of each thread is preserved by all actions of all other threads. The relation *interf* is critical in that context.

β -stablePredicates. The axioms of the CORE level allow us to define a natural class of predicates, which we refer to as *β -stable predicates*. These are predicates that are stable under any β transition.

Definition 4.3. Predicate $P \in 2^\Sigma$ is *β -stable* iff $P \subseteq \text{wlp}(\beta, P)$.

Examples of predicates that are not β -stable are those that fix a view of a thread to a particular write. If we regard the β relation as an abstraction of the flushing behaviour of an operational semantics (e.g., for TSO), then β -stable predicates are those that are stable under this flushing behaviour. In Section 6, we present a language of assertions that are guaranteed to be β -stable by construction, and we describe how β -stable predicates can be *transferred* from one thread to another, when the two threads share appropriate view-maximality properties.

For a β -stable predicate, the following lemma relates stability under interference of a transition with the stability under the transition itself.

LEMMA 4.4 (NONINTERFERENCE). For any $a \in \text{Act}$, thread t and β -stable predicate P , if $P \subseteq \text{wlp}(\text{interf}(t, a), P)$, then $P \subseteq \text{wlp}(T(t, a), P)$.

Thus, given that P is β -stable, if P is stable under interference of t and a , then it is also stable under the transition defined by t and a .

⁷Note that we say “possibly” here w.r.t. view updates, since β is reflexive.

Note that the properties described thus far are stated and proved entirely within our axiom system, i.e., are independent of any specific memory model. Thus, any memory model that satisfies the axioms inherits these properties. Our mechanisation contains further lemmas of the CORE axiomatisation that are omitted here for space reasons. Also note that the framework thus far is independent of the variables and values that the actions act upon. This is addressed in the next level (Section 4.2), which introduces axioms that describe interactions between the variables and values of an action and T , $vmax$, and $interf$.

4.2 The SHAREDVARS Axiomatisation

The next level in our hierarchy inherits the axioms at the CORE level and introduces the concepts of shared variables and actions that read values from and write values to shared variables. The axioms have been fine-tuned to enable verification of our main examples in Section 7.

Memory model structures that satisfy the SHAREDVARS axiomatisation have the signature $(TS, vmax, interf)$, where $(TS, vmax, interf)$ satisfies the axioms of the CORE memory model axiomatisation. In addition, we assume the existence of three functions var , $rdval$, and $wrval$, where $var \in Act \rightarrow Var_G$ is a partial function specifying the variable an action operates on. We lift var to sets of actions in the obvious way. To identify reads and writes, we assume $rdval, wrval : Act \rightarrow Val \cup \{\perp\}$ are functions from actions to values, where $\perp \notin Val$ is used to denote an undefined value, $rdval$ returns the value (if any) that the given action reads and $wrval$ returns the value (if any) that the given action writes. Thus, $Rd = \{a \in Act \mid wrval(a) = \perp \wedge rdval(a) \neq \perp\}$ and $Wr = \{a \in Act \mid rdval(a) = \perp \wedge wrval(a) \neq \perp\}$. Let $Rd|_x$ and $Wr|_x$ be the sets of reads and writes on variable x , and let $Rd[v] \triangleq \{a \in Rd \mid rdval(a) = v\}$ and $Wr[v] \triangleq \{a \in Wr \mid wrval(a) = v\}$ be the reads and writes on a value v , respectively.

We can use these notions to construct assertions that describe the values a given thread can read from a given variable. These assertions are analogous to equations in a conventional setting. In a weak memory setting, a thread might be able to read more than one value from a given variable and distinct threads might be able to observe distinct sets of values (cf. Reference [19]).

Definition 4.5. For a thread t , a variable $x \in Var_G$, and a value $v \in Val$, we define

$$\begin{aligned} [x \neq v]_t &\triangleq \text{dis}(T(t, Rd|_x[v])) \quad (\text{Impossible value}), & [x \equiv v]_t &\triangleq \bigcap_{u \neq v} [x \neq u]_t \quad (\text{Def value}), \\ x \uparrow_t &\triangleq \bigcap_{a \in Act|_x} vmax(t, a) \quad (\text{Maximal view}), & [x = v]_t &\triangleq [x \equiv v]_t \cap x \uparrow_t \quad (\text{Synced value}). \end{aligned}$$

For example, the set of states in which thread t cannot read the value of x to be 42 is captured by $\text{dis}(T(t, Rd|_x[42]))$, and this is captured by the (impossible value) shorthand $[x \neq 42]_t$. Recall that for a set of actions A , $T(t, A) = \bigcup_{a \in A} T(t, a)$. Thus, $\sigma \in [x \neq v]_t \Leftrightarrow \sigma \in \text{wlp}(T(t, Rd|_x[v]), \emptyset) \Leftrightarrow \sigma \in \bigcap_{r \in Rd|_x[v]} \text{wlp}(T(t, r), \emptyset)$.

The set of states in which t is guaranteed to read 42 for x (definite value) is the set of states in which t cannot read any other value. A synchronised value assertion strengthens the definite value assertion and additionally states that t is view maximal over the variable. As we will see in Section 6, each of these assertions is β -stable.

The axioms over SHAREDVARS are split into general axioms describing the interaction between actions and $interf$ and $vmax$, and a set of axioms over reads and writes.

Action interaction with $vmax$ and $interf$. The first two axioms of the SHAREDVARS axiomatisation are as follows. Both axioms deal with the independence of actions on distinct variables (Section 3.1.2).

SV1 (Action independence). For any $a, b \in Act$ and $t, t' \in \text{Tid}$ such that $var(a) \neq var(b)$,
 $interf(t', b) \circ T(t, a) \subseteq T(t, a) \circ interf(t', b)$.

SV2 (Action view stability). For any $a, b \in \text{Act}$ and $t, t' \in \text{Tid}$, such that $\text{var}(a) \neq \text{var}(b)$,
 $\text{vmax}(t, a) \subseteq \text{wlp}(\text{interf}(t', b), \text{vmax}(t, a))$.

SV1 is a weakening of the commutation property present in **SC**: all pairs of operations on distinct variables are independent. The axiom is analogous to (**semi-commutation**), but where the relation R in (**semi-commutation**) is $\text{interf}(t', b)$. **SV2** states that a view-maximality property of any thread is stable under actions on any other variable.

Interaction between reads and writes. The next set of axioms describe the interactions between reads and writes, the underlying transition system (with states in Σ), and vmax and interf .

RW1 (Write independence). For any $t, t' \in \text{Tid}$, $x \in \text{Var}_G$, $r \in \text{Rd}_{|x}$ and $w \in \text{Wr}_{|x}$, such that $\text{rdval}(r) \neq \text{wrval}(w)$, we have

$$\text{interf}(t', w) \circ T(t, r) \subseteq T(t, r) \circ \text{interf}(t', w).$$

RW2 (Read independence). For any $a \in \text{Act}$, $t, t' \in \text{Tid}$ and $r \in \text{Rd}_{|\text{var}(a)}$, we have

$$\text{interf}(t', r) \circ T(t, a) \subseteq T(t, a) \circ \text{interf}(t', r).$$

RW3 (Read view stability). For any $a \in \text{Act}$, $t, t' \in \text{Tid}$ and $r \in \text{Rd}_{|\text{var}(a)}$, we have

$$\text{vmax}(t, a) \subseteq \text{wlp}(\text{interf}(t', r), \text{vmax}(t, a)).$$

RW4 (Read enabledness). For any $x \in \text{Var}_G$ and $t \in \text{Tid}$, we have

$$\Sigma \subseteq \text{dom}(T(t, \text{Rd}_{|x})).$$

RW5 (Write visibility). For any $x \in \text{Var}_G$, $w \in \text{Wr}_{|x}$ and $v = \text{wrval}(w)$, we have

$$\Sigma \subseteq \text{wlp}(T(t, w), \text{dom}(T(t, \text{Rd}_{|x}[v]))).$$

RW6 (Weak coherence). For any $x \in \text{Var}_G$, $t \in \text{Tid}$, and $v \in \text{Val}$, we have

$$x \uparrow_t \subseteq \bigcup_{v \in \text{Val}} [x \equiv v]_t.$$

RW7 (Same variable synchronisation). For any $x \in \text{Var}_G$, $w, r, a \in \text{Act}_{|x}$, $t, t' \in \text{Tid}$, $r \in \text{Rd}$, such that $\text{wrval}(w) = \text{rdval}(r)$ and $t \neq t'$, we have

$$\text{vmax}(t, w) \cap \text{dis}(T(t', r)) \subseteq \text{wlp}(T(t, w), \text{wlp}(T(t', r), \text{vmax}(t', a))).$$

Axioms **RW1** and **RW2** capture semi-commutativity properties for writes and reads, respectively, and are analogous to **SV1**. While **SV1** states semi-commutativity of actions on different variables, axiom **RW1** specifically deals with reads and writes to the *same* variable, but with different values. Because interf (together with β) abstracts writes (see **C3**), under **SC**, **RW1** is trivial, since the left-hand side of the inclusion reduces to \emptyset . Under **TSO** and **RC11-RAR**, **RW1** ensures that if a written value is not seen by a read that follows immediately after the write, then it must also be possible to delay the write until after the read has occurred.

Axiom **RW2** considers the case when a read reads from a variable also accessed by an arbitrary other action. In particular, if it is possible to execute the interference of a read r followed by an action a (on the same variable as r), then it must be possible to execute the interference of r after a . In our concrete models (**SC**, **TSO**, and **RC11-RA**), this is achieved trivially by instantiating $\text{interf}(t, r)$ to id for thread t and read action r .

Axiom **RW3** captures view stability for reads and is analogous to **SV2**, again the difference being that action r reads from the variable of action a . Here, r must preserve view maximality on *any* action a on the same variable as r regardless of which threads execute r and a . As with **RW2**, this is trivially achieved in our concrete models by instantiating $\text{interf}(t, r)$ to id .

Axiom **RW4** states that for every variable, some read of that variable is always possible, and **RW5** states that thread t executing write action w ends in a state in which t can read the value written by w . **RW4** is a restriction on the read actions of a transition system, whereas **RW5** relates writes and reads. Both are natural properties of the transition relation of **SC**, **TSO**, and **RC11-RAR**.

RW6 states that whenever t is view maximal on actions over variable x , then t has a definite value assertion over *some* value for x . Intuitively, this means that in view-maximal states a thread can see just a single value of a variable x . In concrete memory models, **RW6** is analogous to stating that the “modification-order” relation [10, 14, 39] restricted to a single variable is a total order.

Finally, axiom **RW7** defines coherence for actions w (write), r (read), and a (arbitrary), all of which act on the same variable and different threads t and t' . Here, if t is view maximal on w and t' currently cannot read the value written by w , then t executing w will transition to a state such that t' reading the value written by w causes t' to also be view maximal on a .

β -stability over Reads and Writes. The axioms **SV1** and **RW2** together ensure that read interference commutes with all memory actions. Further, **SV2** and **RW3** ensure that read interference preserves view-maximality for all events. Thus, the interference relation for reads is a subset of β , as desired.

This observation gives rise to an important lemma, which we use in Section 6 to define β -stable assertions that are amenable to wlp-based verification. By **C3**, because the transition relation of read actions is guaranteed to be a subset of read interference composed with β , the transition relation of any read action is itself a subset of β . Thus, any read action preserves any β -stable property, so we recover the read-invisibility of SC.

LEMMA 4.6 (READS PRESERVE β -STABLE PREDICATES). *For any $P \subseteq \Sigma$, thread $t, r \in Rd$ and β -stable predicate P , we have $P \subseteq \text{wlp}(T(t, r), P)$.*

4.3 The MSGPASSING Axiomatisation

In SC, every new write is immediately “encountered” by all threads so that the value read for the corresponding variable is the value written by this write without introducing any additional synchronisation. This is not true in weak memory, e.g., in programming languages like C11 [14], specific annotations on read and write operations are required to achieve synchronisation. The MSGPASSING axiomatisation describes this situation by extending the SHAREDVARs and CORE levels with an additional axiom. Memory model structures that satisfy the MSGPASSING axiomatisation have the form $(TS, v\text{max}, \text{interf}, \text{sync})$, where $(TS, v\text{max}, \text{interf})$ satisfies the SHAREDVARs axiomatisation, and $\text{sync} \subseteq Wr \times Rd$ specifies pairs of write/read actions that are capable of synchronising. For example, in RC11-RAR, releasing writes and acquiring reads form synchronising pairs. The additional axiom at this level is the following. We use colour to highlight the transitions of the two threads.

MP (Message passing). For $w, r, b \in \text{Act}$ and $t, t' \in \text{Tid}$ such that $(w, r) \in \text{sync}$, $\text{var}(w) = \text{var}(r)$, $\text{wrval}(w) = \text{rdval}(r)$, $\text{var}(b) \neq \text{var}(w)$, and $t \neq t'$, we have

$$v\text{max}(t, b) \cap \text{wlp}(T(t', r), v\text{max}(t', b)) \subseteq \text{wlp}(T(t, w), \text{wlp}(T(t', r), v\text{max}(t', b))).$$

The assumptions of **MP** state that (w, r) is a synchronising pair of actions, where w is a write to a variable, say x , with value, say v , and r is a read of x with value v . Moreover, b is a third action that operates on a different variable, and the threads t and t' are different. The consequent states:

- if we are in a state such that t is view maximal w.r.t. b , and t' executing read r would cause t' to also become maximal w.r.t. b ,
- then if t executes a write w (enabling r) followed by t' actually executing r does indeed cause t' to become maximal w.r.t. b .

This is precisely the situation that we have seen in Figure 6 immediately after executing $d := 5$, where t is thread 1, t' is thread 2, b is the action corresponding to $d := 5$, w is the action corresponding to $f :=^{\text{WS}} 1$, and r is the action corresponding to $r1 \leftarrow^{\text{RS}} f$.

4.4 The FENCES Axiomatisation

Fences provide another way of synchronising threads. In TSO, fences flush store buffers of threads to main memory and thereby make writes visible to other threads. In terms of views, fences in a thread advance the views of other threads. Memory model structures that satisfy the FENCES axiomatisation have the form $(TS, vmax, interf, fence)$, where $(TS, vmax, interf)$ satisfies the SHARED VARS axiomatisation. Moreover, we have $fence \in Act$ and $fence \notin dom(var)$.

FNC (Fence view). For all $a \in Act$ and $t, t' \in Tid$,
 $vmax(t, a) \subseteq wlp(T(t, fence), vmax(t', a))$.

Note that axiom FNC corresponds to the behaviour of a TSO fence (we will see this relationship more formally in Section 5.2). Moreover, we show that, using FNC, it is possible to verify the store buffering litmus test (Section 7.3).

There are a variety of other types of fences available in the weak memory literature, which advance thread views in different ways. As already discussed, we do not consider axiomatisations for each of these in this article, since they are not part of the main contribution.

4.5 Example Proof

It is instructive to now discuss how these axioms can be used for program verification. Suppose we have a concurrent program with threads (1 and 2) and wish to show correctness of the following Hoare triple (to be made more precise in Section 6):

$$\{[x \equiv 42]_1\} y :=_2 10 \{[x \equiv 42]_1\}.$$

This can be read as “if thread 1 definitely knows x to be 42, then this is also the case after thread 2 has written 10 to y .” We require such a property to hold even in a weak memory setting, and in our proofs below, this reasoning is often needed to show that one thread’s operations do not interfere with the assertions of another thread. By Definition 4.5 (definite value), the proof of this Hoare triple requires showing for all $v \neq 42$:

$$\text{dis}(T(1, Rd_{|x}[v])) \subseteq wlp(T(2, Wr_{|y}[10]), \text{dis}(T(1, Rd_{|x}[v]))). \quad (1)$$

We have the following calculation, where we assume $T_R = T(1, Rd_{|x}[v])$ and $I_W = interf(2, Wr_{|y}[10])$:

$$\begin{aligned} & \text{dis}(T_R) \\ & \subseteq wlp(T_R, wlp(I_W, \emptyset)) && wlp \text{ monotonicity, since } \text{dis}(T_R) = wlp(T_R, \emptyset), \\ & \subseteq wlp(T_R \circ I_W, \emptyset) && wlp \text{ composition,} \\ & \subseteq wlp(I_W \circ T_R, \emptyset) && \mathbf{SV1} \text{ and } wlp \text{ anti-monotonicity,} \\ & \subseteq wlp(I_W, \text{dis}(T_R)) && wlp \text{ composition and definition of } \text{dis}. \end{aligned}$$

In Section 6, we show that impossible value assertions are β -stable. Thus, Lemma 4.4 applies, and the above calculation ensures Equation (1).

5 MEMORY MODELS

We now describe three instantiations of the axioms in Section 4 by showing how core properties of three popular memory models can, in fact, be captured by our axiom schemes. We instantiate sequential consistency SC (Section 5.1), TSO (Section 5.2), and RC11-RAR (Section 5.3) as example memory models. Once these instantiations have been shown to satisfy the axiom scheme, it is possible to use the axiom schemes directly in program verification by appealing to reasoning over weakest liberal preconditions.

5.1 Sequential Consistency (SC)

For SC, we can readily establish compatibility with MSGPASSING+FENCES, since it is a memory model in which all actions are synchronised. The parameters of our framework are as follows:

$$\begin{aligned} T_{SC}(t, a) &\triangleq \xrightarrow{a, t}_{SC}, & \text{sync}_{SC} &\triangleq Wr \times Rd, & \text{fence}_{SC} &\triangleq \text{skip}, \\ v\text{max}_{SC}(t, a) &\triangleq \Sigma, & \text{interf}_{SC}(t, a) &\triangleq \text{if } a \in Rd \cup \{\text{skip}\} \text{ then } id \text{ else } \xrightarrow{a, t}_{SC}. \end{aligned}$$

Note that $\xrightarrow{a, t}_{SC}$ has already been given in Section 2.2. Thus, $T_{SC}(t, a)$ is as defined in Figure 4 and $v\text{max}_{SC}(t, a)$ is the entire set of states (since all threads are always maximal). The interference relation interf_{SC} approximates reads and skip statements by id and writes by the writing transition itself. Finally, the synchronisation relation is $Wr \times Rd$, i.e., all write/read pairs induce synchronisation.

We now check that these parameters induce a valid memory model structure by checking that the axioms in Section 4 are satisfied.

THEOREM 5.1. *Suppose $TS_{SC} = (\text{Act}_{SC}, \Sigma_{SC}, I, T_{SC})$, where $I \subseteq \Sigma_{SC}$. Then the memory model structure $(TS_{SC}, v\text{max}_{SC}, \text{interf}_{SC}, \text{sync}_{SC}, \text{fence}_{SC})$ satisfies MSGPASSING+FENCES.*

5.2 Total Store Order (TSO)

In this section, we show that the TSO memory model satisfies the axiom scheme specified by MSGPASSING+FENCES. A difficulty in proving that TSO is an instance of our axiom scheme arises from the fact that the buffer to be flushed is non-deterministically selected. This makes it difficult to ascertain the order in which cached writes may be seen by other threads. To solve this, we develop an alternative characterisation of TSO, called **prophetic TSO (PTSO)**, that resolves the non-determinism of buffer flushes at an earlier point in a program's execution. In particular, the global order in which each write is to be flushed is determined when the write itself occurs. We then show that prophetic TSO approximates the standard definition of TSO by showing that any behaviour produced by TSO is a behaviour that can be produced by prophetic TSO.

Formally, we use a timestamping strategy, which is often used in the context of weak memory models [19, 33, 34, 50] and which we have already seen for RC11-RAR in Section 2.2. Each write, when executed, is associated with a timestamp, which is a rational number that “prophetically” sets the time at which its associated flush is going to occur. By ensuring each timestamp for each write is unique across all threads, these timestamps determine a unique ordering for flushing buffered writes across all threads. To ensure FIFO order for the writes issued by each thread, we require that the timestamp for a write executed by a thread t is greater than the timestamp of each write that is already in t 's buffer. Note, however, that a timestamp order need not be maintained across threads, and hence, a write by thread t may “overtake” an existing buffered write w issued by another thread by picking an earlier timestamp than w .

The actions of the TSO model are given by

$$\text{Act}_{PTSO} \triangleq \bigcup_{x \in \text{Var}_G, v \in \text{Val}} \{rd(x, v), wr(x, v)\} \cup \{\text{fence}\}.$$

Moreover, we have that $\text{memop}(rd^{[RS]}(x, r, n)) = rd(x, n)$, $\text{memop}(wr^{[WS]}(x, n)) = wr(x, n)$ and $\text{memop}(\text{fence}) = \text{fence}$.

Each state of the TSO memory model (see Σ_{PTSO} below) consists of the shared store s and a timestamped write buffer wb_t per thread t . Let val , var and ts be functions that return the variable x , value v and timestamp q of a given triple (x, v, q) , respectively, where (x, v, q) is a write in the

$$\begin{array}{c}
\text{PTSO-READ} \frac{a = rd(x, v) \quad v = val_\sigma(t, x)}{\sigma \xrightarrow{a, t}_{PTSO} \sigma} \qquad \text{PTSO-WRITE} \frac{a = wr(x, v) \quad fresh_\sigma(t, q) \quad wb' = \sigma.wb[t := \sigma.wb_t \cdot \langle (x, v, q) \rangle]}{\sigma \xrightarrow{a, t}_{PTSO} (s, wb')} \\
\text{PTSO-FENCE} \frac{a = fence \quad \sigma.wb_t = \langle \rangle}{\sigma \xrightarrow{a, t}_{PTSO} \sigma} \qquad \text{PTSO-FLUSH} \frac{wb_t = \langle (x, v, _) \rangle \cdot w \quad s' = \sigma.s[x := v] \quad nextFlush_\sigma(t) \quad wb' = \sigma.wb[t := w]}{\sigma \xrightarrow{flush, t}_{PTSO} (s', wb')}
\end{array}$$

Fig. 10. Operational semantics for prophetic TSO (PTSO).

write buffer. We let $(wb_t)_{|x}$ be the sequence wb_t restricted to variable x . For a state σ , thread t , variable x , value u , and sequence z , we define a number of predicates/operations:

$$\begin{aligned}
x \in z &\triangleq \exists z_1, z_2. z = z_1 \cdot \langle x \rangle \cdot z_2, \\
fresh_\sigma(t, q) &\triangleq (\forall t' \in \text{Tid}. (_, _, q) \notin \sigma.wb_{t'}) \wedge (\forall (_, _, q') \in \sigma.wb_t. q > q'), \\
nextFlush_\sigma(t) &\triangleq \exists q. (_, _, q) = wb_t(0) \wedge \forall t' \in \text{Tid} \setminus \{t\}. \forall (_, _, q') \in wb_{t'}. q' > q.
\end{aligned}$$

Let $wbVal_\sigma(t, x)$ and $wbTS_\sigma(t, x)$ be partial functions that are defined iff $(x, _, _) \in wb_t$. If defined, then we have

$$wbVal_\sigma(t, x) = val(last((\sigma.wb_t)_{|x})), \qquad wbTS_\sigma(t, x) = ts(last((\sigma.wb_t)_{|x})).$$

Finally, the value of a variable x for thread t in state σ is defined as follows:

$$val_\sigma(t, x) \triangleq \text{if } x \notin \sigma.wb_t \text{ then } \sigma.s(x) \text{ else } wbVal_\sigma(t, x).$$

The operational semantics with actions read, write and fence (specific to threads) and flush (system step) is given in Figure 10. Note that the rule PTSO-WRITE picks a fresh compatible timestamp q during the write by thread t , where compatibility only requires that q is greater than the timestamps of the writes that are already in t 's buffer. During PTSO-FLUSH, we require that t is allowed to perform the flush by checking $nextFlush_\sigma(t)$, which holds iff t 's buffer is non-empty and the timestamp of the first write in t 's buffer is less than the timestamp of all other buffered writes. Thus, flushes are executed in a deterministic manner.

The semantics of standard TSO is similar to the semantics given in Figure 10, except that there is no timestamping of buffered writes and non-determinism of flushes is not resolved until the flush occurs. Hence, $fresh_\sigma(t, q)$ is omitted from rule PTSO-WRITE and $nextFlush_\sigma(t)$ is omitted from rule PTSO-FLUSH. The following theorem establishes that prophetic TSO is a sound approximation of TSO and is verified by establishing a backward simulation between TSO (concrete model) and prophetic TSO (abstract model).

THEOREM 5.2 (SOUNDNESS 🌈). *Every trace generated by TSO can be generated by prophetic TSO.*

Thus, any program that can be proved correct with respect to prophetic TSO semantics remains correct with respect to TSO semantics.

We now show that prophetic TSO satisfies the MSGPASSING+FENCES axiom scheme. The parameters for PTSO are defined as follows, where $\xrightarrow{a, t}_{PTSO}$ is the relation defined in Figure 10. Note that $vmax$ and $interf$ need not be defined for $flush$, since a program does not explicitly execute a

flush, i.e., $flush \notin \text{Act}$. Moreover, we define $FL \triangleq (\bigcup_{t \in \text{Tid}} \xrightarrow{flush, t}_{PTSO})^*$:

$$\begin{aligned} \Sigma_{PTSO} &\triangleq (Var_G \rightarrow Val) \times (\text{Tid} \rightarrow (Var_G \times Val \times \mathbb{Q})^*), & T_{PTSO}(t, a) &\triangleq FL \circ \xrightarrow{a, t}_{PTSO}, \\ I_{PTSO} &\triangleq \{(s, wb) \mid s \in Var_G \rightarrow Val \wedge \forall t. wb_t = \langle \rangle\}, & sync_{PTSO} &\triangleq Wr \times Rd, \\ v_{max_{PTSO}}(t, a) &\triangleq \text{if } a \in Wr \cup Rd \text{ then } maxTS(t, var(a)) \text{ else } \Sigma_{PTSO}, & fence_{PTSO} &\triangleq fence, \\ interf_{PTSO}(t, a) &\triangleq \text{if } a \in Rd \cup \{fence\} \text{ then } id \text{ else } ins(t, a) \circ FL, \end{aligned}$$

where $maxTS(t, x)$ returns the set of states in which t 's view of x is maximal and $ins(t, a)$ inserts a into the write buffer of t as an entry $(var(a), wrval(a), q)$ such that q is a valid time stamp:

$$maxTS(t, x) \triangleq \{\sigma \mid (x, _, _) \in \sigma.wb_t \wedge (\forall t', q. (x, _, q) \in \sigma.wb_{t'} \Rightarrow wbTS_\sigma(t, x) \geq q) \cup \{\sigma \mid \forall t'. (x, _, _) \notin \sigma.wb_{t'}\},$$

$$\begin{aligned} ins(t, a)(\sigma, \sigma') &\triangleq \\ &\text{let } x = var(a), v = wrval(a) \text{ in} \\ &\exists z, z'. \sigma.wb_t = z \cdot z' \wedge \sigma.s = \sigma'.s \wedge \\ &\exists q. (\forall e \in z. ts(e) < q) \wedge (\forall e \in z'. q < ts(e)) \wedge \\ &(\forall t' \neq t. (_, _, q) \notin \sigma.wb_{t'}) \wedge \sigma'.wb[t := z \cdot \langle (x, v, q) \rangle \cdot z']. \end{aligned}$$

Thus, the state space Σ_{PTSO} is just the shared memory together with sequences of timestamped writes for each thread (modelling the timestamped buffer). The transition system is the transition system defined in Figure 10. Finally, the interference for any thread t and action a is id whenever a is a *fence* or read. If a is a write, then the interference relation is the transition corresponding to the write together with actions that insert the write into the buffer at arbitrary positions (including at the end) then perform some (possibly zero) flushes.

THEOREM 5.3. *Let $TS_{PTSO} = (\text{Act}_{PTSO}, T_{PTSO}, I_{PTSO}, \Sigma_{PTSO})$ be the transition system for PTSO. The memory model structure $(TS_{PTSO}, v_{max_{PTSO}}, interf_{PTSO}, sync_{PTSO}, fence_{PTSO})$ satisfies MSGPASSING+FENCES.*

5.3 RC11-RAR

Next, we explain the memory model RC11-RAR in more detail. Note that RC11-RAR assumes a restriction [39] that ensures program order is maintained. In Section 2.2, we have already explained the state Σ_{C11} ; Figure 11 now gives the operational semantics. For this, we define

$$sync_{C11} \triangleq Wr_R \times Rd_A,$$

where $Wr_R \subseteq Wr$ is the set of writes annotated by a release (similarly, $Rd_A \subseteq Rd$).

Central to the operational semantics is the notion of *observable writes* (of a global variable x), which define the values a thread t can read from (rule C11-Read) as well as determine the timestamps of new writes (rule C11-Write):

$$OW_\sigma(t, x) \triangleq \{(w, q) \in \sigma.writes_x \mid q \geq \sigma.tview_t(x)\}.$$

State components $tview \in TView$ and $mview \in MView$ together facilitate the transfer of views from one thread to another when a release-acquire synchronisation (in a read event) occurs. In particular, during release-acquire synchronisation (i.e., when $(w, r) \in sync_{C11}$ for some read action r), the view of the executing thread is updated using

$$v_1 \otimes v_2 \triangleq \lambda x. \text{if } tst(v_2(x)) \leq tst(v_1(x)) \text{ then } v_1(x) \text{ else } v_2(x),$$

where $v_1, v_2 \in Var \rightarrow (\text{Act}_{C11} \times \mathbb{Q})$ (the $tview$ of a thread and the $mview$ of a write), and $tst((w, q)) \triangleq q$ returns the timestamp of the write event (w, q) . Thus, for each variable x , $(v_1 \otimes v_2)(x)$ returns the action-timestamp pair corresponding to the action with the larger timestamp out of $v_1(x)$ and $v_2(x)$. In rule C11-READ, \otimes allows $\sigma.tview_t$ to inherit knowledge from

$$\begin{array}{c}
\text{C11-READ} \frac{r \in \{rd(x, n), rd^\wedge(x, n)\} \quad (w, q) \in \sigma.OW(t, x) \quad wrval(w) = n \quad \textcolor{red}{tview}' = \text{if } (w, r) \in \text{sync}_{C11} \text{ then } \sigma.tview_t \otimes \sigma.mview_{(w, q)} \text{ else } \sigma.tview_t[x := (w, q)]}{\sigma \xrightarrow{r, t}_{C11} (\sigma.tview[t := \textcolor{red}{tview}'], \sigma.mview, \sigma.writes)} \\
\\
\text{C11-WRITE} \frac{w \in \{wr(x, n), wr^R(x, n)\} \quad (w_0, q_0) \in \sigma.OW(t, x) \quad \text{fresh}_\sigma(q_0, q) \quad \text{writes}' = \sigma.writes \cup \{(w, q)\} \quad \textcolor{red}{tview}' = \sigma.tview_t[x := (w, q)]}{\sigma \xrightarrow{w, t}_{C11} (\sigma.tview[t := \textcolor{red}{tview}'], \sigma.mview[(w, q) := \textcolor{red}{tview}'], \text{writes}')}
\end{array}$$

Fig. 11. Memory model semantics for RC11-RAR.

$\sigma.mview_{(w, q)}$ if the read synchronises (via release-acquire) with an observable (w, q) . Note that read actions change the state, i.e., read invisibility is not given in RC11-RAR anymore.

In rule C11-WRITE, when the new write action is inserted immediately after a write with timestamp q_0 , the timestamp q of the new action must satisfy: $\text{fresh}_\sigma(q_0, q) \triangleq q_0 < q \wedge \forall w' \in \sigma.writes. q_0 < \text{tst}(w') \Rightarrow q < \text{tst}(w')$. In addition, for each write w , we instantiate $mview_w$ to be the thread view $\textcolor{red}{tview}'$ of the thread t that issued w at the time of writing (see C11-WRITE).

Next, we detail the parameters for the RC11-RAR memory model. The transition system for RC11-RAR is defined as follows:

$$\begin{aligned}
\Sigma_{C11} &\triangleq TView \times MView \times Writes, & T_{C11}(t, a) &\triangleq \xrightarrow{a, t}_{C11}, \\
I_{C11} &\triangleq \left\{ (tview, mview, writes) \mid \exists F \in Var_G \rightarrow Val. \text{let } I_x = (wr(x, F(x)), 0) \text{ in } \right. \\
&\quad \left. \begin{array}{l} \text{writes} = \{I_x \mid x \in Var_G\} \wedge \\ \text{tview} = \lambda t \in \text{Tid}. \lambda x \in Var_G. I_x \wedge \\ \text{mview} = \lambda w \in \text{writes}. \lambda x \in Var_G. I_x \end{array} \right\}.
\end{aligned}$$

In the initial state, I_{C11} , we fix the timestamp of each write to 0, and assume a function F mapping each variable to an initial value.

In RC11-RAR, a thread t is view maximal with respect to an action a iff t is maximal with respect to $\text{var}(a)$, i.e.,

$$vmax_{C11}(t, a) \triangleq \{\sigma \mid \forall (b, q) \in \sigma.writes. \text{var}(a) = \text{var}(b) \Rightarrow q \leq \text{tst}(\sigma.tview_t(\text{var}(a)))\}.$$

This guarantees that if thread t is capable of performing an action a , then the action interacts with the last write in $\sigma.writes$ for any $\sigma \in vmax_{C11}(t, a)$.

Next, we present the over-approximating interference relation interf_{C11} . For a read, we simply have that interf_{C11} allows *id* steps. For a write, the write must be introduced with an arbitrary fresh timestamp (but not before the initial write):

$$\text{interf}_{C11}(t, a) \triangleq \text{if } a \in Rd \text{ then } id \text{ else } ins_W(t, a),$$

where $ins_W(t, a)$ inserts a into the state with a valid fresh timestamp q and (a, q) 's view of the other variables except for $\text{var}(a)$ is the initial write. Assume $\text{first}_\sigma(x) = w \Leftrightarrow w \in \sigma.writes_{|x} \wedge \text{tst}(w) = 0$ is the initial write to x in σ :

$$\begin{aligned}
(\sigma, \sigma') \in ins_W(t, a) &\iff \\
&\exists (w_0, q_0) \in \sigma.writes. \text{var}(w_0) = \text{var}(a) \wedge \text{fresh}_\sigma(q_0, q) \wedge \\
&\quad \text{let } mv' = (\lambda x \in Var_G. \text{if } x = \text{var}(a) \text{ then } (a, q) \text{ else } \text{first}_\sigma(x)) \text{ in} \\
&\quad \sigma' = (\sigma.tview \otimes mv', \sigma.mview[(a, q) := mv'], \sigma.writes \cup \{(a, q)\}).
\end{aligned}$$

Finally, recall that the synchronisation relation for message passing, i.e., sync_{C11} , has already been defined above for the operational semantics. Note that sync_{C11} is weaker than the conventional definition of release-acquire synchronisation, which requires the synchronising read and write to

be on the same variable and have the same value. This is not required in our setting, since the behaviour of operations in *sync* are described by the axiom **MP**, which restricts the read and write to the same variable with the same value.

THEOREM 5.4. *Let $TS_{C11} \triangleq (\text{Act}_{C11}, \Sigma_{C11}, I_{C11}, T_{C11})$ be the transition system for C11. Then the memory model structure $(TS_{C11}, \text{vmax}_{C11}, \text{interf}_{C11}, \text{sync}_{C11})$ satisfies MSGPASSING.*

6 VERIFICATION WITH WEAKEST LIBERAL PRECONDITIONS

In this section, we develop a Hoare-logic [30] using the proposal of Owicki and Gries for concurrent programs [48]. Owicki-Gries reasoning builds on *assertions* put in between program statements (thereby obtaining *proof outlines*). The proof rules for imperative programming constructs like that in our language are standard (see, e.g., Apt et al. [11]). Proving soundness of a proof outline requires showing two sorts of proof obligations:

Local Correctness. For every program command *com* with pre-assertion *P* and post-assertion *Q* in (the proof outline of) thread *t*, prove the Hoare-triple $\{P\}com_t\{Q\}$.

Global Correctness (non-interference). For every assertion *R* in the proof outline of thread *t* and command *com* with pre-assertion *P* in the program of thread *t'*, prove $\{R \wedge P\}com_{t'}\{R\}$.

We say that a proof outline is *valid* if it is locally and globally correct.

The axiomatisation builds on the transition relation *T* and thus describes the global semantics only. To reason about programs, both the local and global semantics need to be taken into consideration. To this end, we define a transition relation \widehat{T} over *Act* as follows:

$$((lst, \sigma), (lst', \sigma')) \in \widehat{T}(t, a) \quad \text{iff} \quad \exists \Pi, P'. (\Pi, lst, \sigma) \xRightarrow{a}_t (P', lst', \sigma').$$

We also use \widehat{T} as a relation for the weakest liberal precondition transformer.

The correspondence $\{P\}S\{Q\} \Leftrightarrow P \subseteq \text{wlp}(S, Q)$ between Hoare-triples and *wlp* in the standard SC setting is well known [45]. The corresponding proof obligation for showing $\{P\}com_t\{Q\}$ in our approach is a proof of

$$P \subseteq \text{wlp}(\widehat{T}(t, a), Q)$$

for all actions *a* that arise in executions of *com* in *t*. More formally, we define a mapping *act* : *ACom* → 2^{Act} mapping commands to language actions, where

$$\begin{aligned} \text{act}(\mathbf{fnc}) &\triangleq \{\text{fence}\}, & \text{act}(r := E) &\triangleq \{r := n \mid \exists ls. \llbracket E \rrbracket_{ls} = n\}, \\ \text{act}(x :=^{[\text{WS}]} E) &\triangleq \{\text{wr}^{[\text{WS}]}(x, n) \mid \exists ls. n = \llbracket E \rrbracket_{ls}\}, & \text{act}(r \leftarrow^{[\text{RS}]} x) &\triangleq \{rd^{[\text{RS}]}(x, r, n) \mid n \in \text{Val}\}. \end{aligned}$$

We start with the introduction of *assertions* specifying program properties. We distinguish between *global* properties (on shared variables) and *local* properties (on registers). A property is given as the set of states satisfying it. The following syntax describes the global properties:

$$\text{Prop}_G ::= \emptyset \mid \text{wlp}(T(t, a), \text{Prop}_G) \mid \text{vmax}(t, a) \mid \text{Prop}_G \cap \text{Prop}_G \mid \text{Prop}_G \cup \text{Prop}_G,$$

where *a* ∈ *Act* is an action and *t* ∈ *Tid* a thread. This, in particular, allows expressing all of the assertions in Definition 4.5. Note that in contrast to normal Hoare-logic, these assertions are specific to threads. As local properties, we allow all Boolean expressions from *Exp_L*. Global and local properties can be combined as follows, where *b* ∈ *Act*:

$$\text{Prop} ::= \emptyset \mid \text{Prop}_G \mid \text{Exp}_L \mid \text{wlp}(\widehat{T}(t, b), \text{Prop}) \mid \text{Prop} \cap \text{Prop} \mid \text{Prop} \cup \text{Prop}.$$

PROPOSITION 6.1 (🌀). *Let *M* be a memory model structure satisfying the axioms of CORE and SHARED VARS. Then all predicates defined by the grammar *Prop* are β-stable.*

The axiomatisation provides us with a way of reasoning about the weak memory model semantics in terms of wlp. To connect reasoning about shared and local state, we employ a number of *lifting* rules. The first two lifting rules lift β -stable global properties P to stability under read and fence transitions of the full semantics, thereby expressing the intended meaning of β as abstraction of view advancements performed by reads or fences in weak memory. The third lifting rule lifts Hoare triples (weakest liberal precondition constraints on actions) about global properties P and Q :

$$\frac{P \subseteq \text{wlp}(\beta, P)}{P \subseteq \text{wlp}(\widehat{T}(t, \text{rd}), P)}, \quad \frac{P \subseteq \text{wlp}(\beta, P)}{P \subseteq \text{wlp}(\widehat{T}(t, \text{fence}), P)}, \quad \frac{P \subseteq \text{wlp}(T(t, a), Q)}{P \subseteq \text{wlp}(\widehat{T}(t, a), Q)}.$$

Further, more specialised lifting rules can be found in our Isabelle/HOL mechanisation [23]. The non-axiomatised language semantics part with local properties about registers can directly be defined on the full semantics. Here, we get the two (standard) rules of assignment from Hoare logic:

$$\text{expr}[r := n] \subseteq \text{wlp}(\widehat{T}(t, \text{rd}^{[\text{RS}]}(x, r, n)), \text{expr}), \quad (2)$$

$$\text{expr}[r := n] \subseteq \text{wlp}(\widehat{T}(t, r := n), \text{expr}), \quad (3)$$

where $\text{expr} \in \text{Exp}_L$ and $\text{expr}[r := n]$ is the expression expr with r syntactically replaced by n .

This completes the description of the proof technique. As we can see, there are no specific additional proof rules required; we build entirely on the classical concurrency verification technique, proving Hoare triples via the wlp predicate transformer. The key contribution of our article is thus a *memory model agnostic* verification methodology.

7 APPLICATIONS

In this section, we present example verifications of classic litmus tests using our axiomatic framework. We present the proof outlines using generic axioms for program templates, which means that they only have to be validated (using Owicki-Gries rules) once per program. Any memory model instantiation of the axioms also satisfies these proofs.

7.1 Load Buffering

The load buffering program and its proof outline are as given in Figure 12. Thread 1 loads x into r_1 then assigns 1 to y , whereas thread 2 loads y into r_2 then assigns 1 to x . We expect the programs to terminate so that either r_1 or r_2 is 0, i.e., it should not be the case that both threads 1 and 2 read the new value for y and x , respectively. We expect all three of the memory models we consider to satisfy the proof outline even in the absence of any explicit synchronisation.⁸

PROPOSITION 7.1 (LOAD BUFFERING 🍷). *The proof outline in Figure 12 is valid under SHARED VARS.*

We give an outline of the proof of Proposition 7.1 to provide some intuition for our verification framework. First, we note that the proof follows the Owicki-Gries style reasoning, which requires that we prove local correctness and global correctness.

⁸Note that we consider RC11 [39], which disallows cycles over sequenced before (aka program order) and the “reads from” relation. For the load buffering proof outline to be valid for such memory models, load buffering requires an explicit dependency between the load and the store [34, 49], which would need to be reflected in the program. We consider such examples to be future work.

Init: $x := 0; y := 0; r_1 := 0; r_2 := 0;$
 $\{[x = 0]_1 \cap [y = 0]_2 \cap r_1 = 0 \cap r_2 = 0\}$
Thread 1 \parallel **Thread 2**
 $\{[y = 0]_2 \cap r_2 = 0\} \parallel \{[x = 0]_1 \cap r_1 = 0\}$
 $1 : r_1 \leftarrow x; \quad 3 : r_2 \leftarrow y;$
 $\{[y = 0]_2 \cap r_2 = 0\} \parallel \{[x = 0]_1 \cap r_1 = 0\}$
 $2 : y := 1; \quad 4 : x := 1;$
 $\{r_1 = 0 \cup r_2 = 0\} \parallel \{r_1 = 0 \cup r_2 = 0\}$
 $\{r_1 = 0 \cup r_2 = 0\}$

Fig. 12. Proof outline for load buffering.

Init: $d := 0; f := 0;$
 $\{[f = 0]_1 \cap [f = 0]_2 \cap [d = 0]_1 \cap [d = 0]_2\}$
Thread 1 \parallel **Thread 2**
 $\{[f \neq 1]_2 \cap [d = 0]_1\} \parallel \{[f = 1][d = 5]_2\}$
 $1 : d := 5; \quad 3 : \text{do } r_1 \leftarrow^{\text{RS}} f$
 $\{[f \neq 1]_2 \cap [d = 5]_1\} \parallel \quad \text{until } r_1 = 1;$
 $2 : f :=^{\text{WS}} 1; \quad \{[d = 5]_2\}$
 $\{true\} \parallel \{r_2 = 5\}$
 $\{r_2 = 5\}$

Fig. 13. Proof outline for message passing.

Local correctness. For local correctness of thread 1, the initial condition is trivially satisfied, since it is implied by the initialisation of the program. Moreover, we must prove that for any v :

$$[y = 0]_2 \cap r_2 = 0 \subseteq \text{wlp}(\widehat{T}(1, rd(x, r_1, v)), [y = 0]_2 \cap r_2 = 0), \quad (4)$$

$$[y = 0]_2 \cap r_2 = 0 \subseteq \text{wlp}(\widehat{T}(1, wr(y, 1)), r_1 = 0 \cup r_2 = 0). \quad (5)$$

The proof of Equation (4) proceeds as follows:

$$\begin{aligned}
 & (4) \\
 &= ([y = 0]_2 \cap r_2 = 0 \subseteq \text{wlp}(\widehat{T}(1, rd(x, r_1, v)), [y = 0]_2)) \quad \text{Conjunctivity of wlp} \\
 &\quad \cap ([y = 0]_2 \cap r_2 = 0 \subseteq \text{wlp}(\widehat{T}(1, rd(x, r_1, v)), r_2 = 0)) \\
 &\Leftarrow ([y = 0]_2 \subseteq \text{wlp}(\widehat{T}(1, rd(x, r_1, v)), [y = 0]_2)) \quad \text{Logic} \\
 &\quad \cap (r_2 = 0 \subseteq \text{wlp}(\widehat{T}(1, rd(x, r_1, v)), r_2 = 0)) \\
 &\Leftarrow ([y = 0]_2 \subseteq \text{wlp}(T(1, rd(x, r_1, v)), [y = 0]_2)) \quad \text{Lifting and (2)} \\
 &\quad \cap (r_2 = 0 \subseteq r_2 = 0) \\
 &\Leftarrow \text{True} \quad \beta\text{-stability and Lemma 4.6.}
 \end{aligned}$$

The final step relies on a preservation property for reads, which states that a synced value assertion cannot be affected by a read. For Equation (5) by disjunctivity of wlp and logic, we prove

$$r_2 = 0 \subseteq \text{wlp}(\widehat{T}(1, wr(y, 1)), r_2 = 0).$$

This then follows directly by Equation (2). Local correctness of thread 2 is symmetric.

Global correctness. For the precondition of line 1, we have proof obligations:

$$[y = 0]_2 \cap r_2 = 0 \cap [x = 0]_1 \cap r_1 = 0 \subseteq \text{wlp}(\widehat{T}(2, rd(y, r_2, v)), [y = 0]_2 \cap r_2 = 0), \quad (6)$$

$$[y = 0]_2 \cap r_2 = 0 \cap [x = 0]_1 \cap r_1 = 0 \subseteq \text{wlp}(\widehat{T}(2, wr(x, 1)), [y = 0]_2 \cap r_2 = 0). \quad (7)$$

To prove Equation (6), by conjunctivity of wlp and logic (weakening antecedent), it suffices to prove:

$$[y = 0]_2 \subseteq \text{wlp}(\widehat{T}(2, rd(y, r_2, v)), [y = 0]_2), \quad (8)$$

$$[y = 0]_2 \subseteq \text{wlp}(\widehat{T}(2, rd(y, r_2, v)), r_2 = 0). \quad (9)$$

The proof of Equation (8) is once again by lifting and the *syncVal* preservation property described above. The proof of Equation (9) is via a property on reading synced values. In particular, by definition, every read $rd(y, r_2, v)$ for $v \neq 0$ is disabled, and hence it is impossible for v to return a value different from 0.

Init: $x := 0; y := 0; f_1 := 0; f_2 := 0$

Thread 1

$\{(f_2 = 0 \cup r_2 = 1 \cup [y = 1]_1) \cap f_1 = 0 \cap [x = 0]_1\}$
 1 : $x := 1;$
 $\{(f_2 = 0 \cup r_2 = 1 \cup [y = 1]_1) \cap f_1 = 0 \cap [x = 1]_1\}$
 2 : $\langle \text{fnc}; f_1 := 1 \rangle;$
 $\{(f_2 = 0 \cup r_2 = 1 \cup [y = 1]_1) \cap f_1 = 1 \cap [x = 1]_2\}$
 3 : $\langle r_1 \leftarrow y; f_1 := 2 \rangle$
 $\{(f_2 = 0 \cup r_2 = 1 \cup r_1 = 1 \cup f_2 = 1) \cap \}$
 $\{f_1 = 2 \cap [x = 1]_2\}$

$\{f_1 = 0 \cap f_2 = 0\}$

Thread 2

$\{(f_1 = 0 \cup r_1 = 1 \cup [x = 1]_2) \cap f_2 = 0 \cap [y = 0]_2\}$
 4 : $y := 1;$
 $\{(f_1 = 0 \cup r_1 = 1 \cup [x = 1]_2) \cap f_2 = 0 \cap [y = 1]_2\}$
 5 : $\langle \text{fnc}; f_2 := 1 \rangle;$
 $\{(f_1 = 0 \cup r_1 = 1 \cup [x = 1]_2) \cap f_2 = 1 \cap [y = 1]_1\}$
 6 : $\langle r_2 \leftarrow x; f_2 := 2 \rangle$
 $\{(f_1 = 0 \cup r_1 = 1 \cup r_2 = 1 \cup f_1 = 1) \cap \}$
 $\{f_2 = 2 \cap [y = 1]_1\}$

$\{r_1 = 1 \cup r_2 = 1\}$

Fig. 14. Store buffering (SB).

Details of the global correctness proofs of the other assertions are omitted for brevity, and we once again refer the interested reader to our Isabelle/HOL mechanisation [23].

7.2 Message Passing

Next, consider the proof outline of the program for the message passing example (Figure 13), which is the program discussed in Section 3.2. Note that the loop **do** C **until** B is shorthand for $C; \text{while } \neg B \text{ do } C$. The proof outline contains a further annotation:

$$\langle y = u \rangle [x = v]_t \hat{=} \text{wlp}(T(t, rd^{\text{RS}}(y, u)), [x = v]_t).$$

Here, $\langle y = u \rangle [x = v]_t$ is a *conditional observation* assertion [19], which ensures that thread t obtains a synchronised value assertion from Definition 4.5 on x if it performs a synchronising read to y with value u .

Moreover, we can generically lift **MP** to a property over variables.

LEMMA 7.2. *For any variables x and y such that $x \neq y$, read $r \in Rd_{|x}$ and write $w \in Wr_{|x}$ such that $(w, r) \in \text{sync}$ and $rdval(r) = wrval(w)$, and any threads $t, t' \in \text{Tid}$,*

$$y_{\uparrow t} \cap \text{wlp}(T(t', r), y_{\uparrow t'}) \subseteq \text{wlp}(T(t, w), \text{wlp}(T(t', r), y_{\uparrow t'})).$$

In the proof outline (Figure 13), the conditional observation assertion in the precondition of line 3 allows us to establish the synchronised value assertion on d .

PROPOSITION 7.3 (MESSAGE PASSING 🎲). *The proof outline in Figure 13 is valid under MSGPASSING.*

For brevity, we illustrate just one aspect here. For global correctness of thread 2, we need to prove that the assertion $\langle f = 1 \rangle [d = 5]_2$ is preserved by the assignment $f := 1$ in thread 1, i.e.,

$$\langle f = 1 \rangle [d = 5]_2 \cap [f \neq 1]_2 \cap [d = 5]_1 \subseteq \text{wlp}(\widehat{T}(1, wr^{\text{WS}}(f, 1)), \langle f = 1 \rangle [d = 5]_2). \quad (10)$$

Recall that synced value assertions imply view maximality. That means part of proving Equation (10) requires that we show the following, which is an application of Lemma 7.2:

$$d_{\uparrow 1} \cap \text{wlp}(T(2, rd^{\text{RS}}(f, r_1, 1)), d_{\uparrow 2}) \subseteq \text{wlp}(T(1, wr^{\text{WS}}(f, 1)), \text{wlp}(T(2, rd^{\text{RS}}(f, r_1, 1)), d_{\uparrow 2})).$$

7.3 Store Buffering

We now consider the program and proof outline for the store buffering example Figure 14, which comprises shared variables x and y (both initially 0). Thread 1 updates x to 1, then later reads y into r_1 . Thread 2 is similar.

In both threads, between the store and the load, we must include additional **fnc** synchronisation that ensures the write will be seen by the other thread. In SC, the **fnc** synchronisation simplifies into a *skip* (or *id* step). In PTSO (and hence TSO), **fnc** is implemented by a *fence* action. The key visibility property we require is that the view of thread 1 on x is synchronised (with value 1) and that after the **fnc** thread 2 is also synchronised on x with value 1.

The proof outline requires the introduction of two *auxiliary variables* that indicate whether the fence on each thread or the load of the shared variable has been executed. These variables are purely local and have no effect on the program's execution. In particular, we introduce variables f_1 and f_2 (both initially 0); f_i is set to 1 by thread i when it executes **fnc**, and to 2 when it executes the load of the shared variable.

PROPOSITION 7.4 (STORE BUFFERING 🧠). *The proof outline in Figure 14 is valid under FENCES.*

The local and global correctness proofs of Figure 14 are largely mechanical. The most interesting aspect of the proof is a “view transfer” property when a thread executes **fnc**. In particular, consider the pre- and post-conditions of line 4, which contain assertions $[y = 1]_2$ and $[y = 1]_1$, respectively. Note that the synchronised value assertion changes from thread 2 to thread 1. In the context of SC, such a transfer property is natural, since all threads interact with the maximal element. Therefore it is sufficient to use a *skip* action to get view transfer. In the context of (P)TSO, the **fnc** must be implemented by a *fence* to ensure the view transfer. This fact is used in the global correctness proof of assertion $f_2 = 0 \cup r_2 = 1 \cup [y = 1]_1$ in the precondition of line 3 against line 5.

As we discussed in Propositions 5.1 and 5.3, for SC, **fnc** is mapped to the action *skip* via *memop*. In (P)TSO, **fnc** is mapped to *fence*. As established by Propositions 5.1 and 5.3, both memory model structures satisfy FENCES, and hence the axiom FNC.

We contrast this with the unsynchronised store buffering example in Appendix A, where we are only able to prove $r_1, r_2 \in \{0, 1\}$ as a postcondition of the program.

7.4 Read-Copy-Update (RCU)

Our final example is a simplified **Read-Copy-Update (RCU)** example employed as a synchronisation mechanism in Linux [22], which comprises a writer that synchronises with a reader before deallocating a memory address.

The program uses local registers wr in the writer and a, rr_1 , and rr_2 in the reader. The index to be deallocated (either n_1 or n_2) is controlled by mb , which is non-deterministically set to either 1 or 2 during initialisation. The writer first stores the opposite value of mb in m , then signals to the reader that this update has occurred by setting shared variable w to 1 using a synchronising write. It then waits for the reader to acknowledge that this write has been seen in a loop (lines 3 and 4), where it tests that the reader flag r has been set. Once this acknowledgement is received the writer deallocates the *old* element indicated by mb . The reader spins until a flag *stop* has been set.⁹ The reader first reads from shared index (n_1 or n_2) into a depending on the value it loads for m and attempts to synchronise with the writer after reading. The required postcondition of the reader (and the program) is that it never loads a deallocated element into a .

Our proof is over the generic axiom scheme, and hence applies to SC, TSO, and RC11-RAR. Moreover, it makes precise the required (weak-memory) synchronisation between the writer and the reader. Namely, the write of w in line 2 must synchronise with the read of w in line 13. Our algorithm is simplified, in the sense that we only consider a single writer and reader, however it

⁹Following Reference [38], this can be modelled by a third concurrent thread.

Assume: $mb \in \{1, 2\} \cap n_1 \neq \perp \cap n_2 \neq \perp \cap a \neq \perp$

Init: $m := mb; w := 0; r := 0; wr := 0; rr_1 := 0; rr_2 := 0;$

$$\left\{ \begin{array}{l} wr = 0 \cap rr_1 = 0 \cap rr_2 = 0 \cap mb \in \{1, 2\} \cap [m = mb]_1 \cap [m = mb]_2 \cap a = \perp \\ \cap [n_1 \neq \perp]_1 \cap [n_2 \neq \perp]_2 \cap [w = 0]_1 \cap [w = 0]_2 \cap [r = 0]_1 \end{array} \right\}$$

Thread 1

```

{[w = 0]_1 \cap [w \neq 1]_2 \cap [m = mb]_1 \cap [m = mb]_2 \cap
{(\bigcap_{j \in \{0,1\}} [r \neq j]_1) \cap mb \in \{1, 2\}}
1 : m := (mb mod 2) + 1;
{[w = 0]_1 \cap [w \neq 1]_2 \cap [m = (mb mod 2) + 1]_1 \cap
{(\bigcap_{j \in \{0,1\}} [r \neq j]_1) \cap mb \in \{1, 2\}}
2 : w :=WS 1;
{(\bigcap_{j \in \{0,1\}} [r \neq j]_1) \cap mb \in \{1, 2\}}
3 : do wr \leftarrow r
{(\bigcap_{j \in \{0,1\}} [r \neq j]_1) \cap mb \in \{1, 2\}}
4 : until wr = 1;
{wr = 1 \cap mb \in \{1, 2\}}
5 : if mb = 1 {wr = 1 \cap mb = 1}
6 :   n_1 := \perp
   else {wr = 1 \cap mb \in 2}
7 :   n_2 := \perp fi
{true}

```

Thread 2

```

{r_pre}
8 : while \neg stop do
{r_inv}
9 :   rr_1 \leftarrow m;
{r_inv \cap rr_1 \in \{1, 2\} \cap
{rr_2 = 1 \Rightarrow rr_1 = mb mod 2 + 1}}
10 :   if rr_1 = 1
{r_inv \cap rr_1 = 1 \cap (rr_2 = 1 \Rightarrow rr_1 = mb mod 2 + 1)}
11 :     a \leftarrow n_1;
   else
{r_inv \cap rr_1 = 2 \cap (rr_2 = 1 \Rightarrow rr_1 = mb mod 2 + 1)}
12 :     a \leftarrow n_2; fi
{r_inv}
13 :   rr_2 \leftarrowRS w;
{r_inv \cap rr_2 \in \{0, 1\}}
14 :   r \leftarrow rr_2;
{a \neq \perp}

```

Fig. 15. Read-copy update (RCU).

is straightforward to see that writer-reader synchronisation mechanism extends to more than one reader, by repeating the loop in lines 3 and 4 for each reader-writer synchronisation.

The proof outline in Figure 15 uses two further assertions, where assertion r_pre is the precondition of the reader, and r_inv is the reader invariant.

$$r_pre = (\bigcap_{j \in \{0,1\}} [w \neq j]_2) \cap (\bigcap_{j \in \{1,2\}} [m \neq j]_2) \cap [r \neq 1]_1 \cap ([w \neq 1]_2 \cup [w = 1]_1) \cap \langle w = 1 \rangle [w = 1]_1 \cap \langle w = 1 \rangle [m = mb \bmod 2 + 1]_2 \cap [n_1 \neq \perp]_2 \cap [n_2 \neq \perp]_2 \cap rr_1 = 0 \cap rr_2 = 0 \cap wr = 0 \cap mb \in \{1, 2\} \cap a \neq \perp,$$

$$r_inv = (\bigcap_{j \in \{0,1\}} [w \neq j]_2) \cap (\bigcap_{j \in \{1,2\}} [m \neq j]_2) \cap \langle w = 1 \rangle [w = 1]_2 \cap \langle w = 1 \rangle [m = (mb \bmod 2 + 1)]_2 \cap (rr_2 \neq 1 \cup [m = (mb \bmod 2 + 1)]_2 \cap [w = 1]_2) \cap ([w \neq 1]_2 \cup [w = 1]_1) \cap ([r \neq 1]_1 \cup rr_2 = 1) \cap a \neq \perp \cap mb \in \{1, 2\} \cap \left(([n_1 \neq \perp]_2 \cap [n_2 \neq \perp]_2 \cap wr = 0) \cup (rr_2 = 1 \cap (mb \neq 1 \cup [n_2 \neq \perp]_2) \cap (mb \neq 2 \cup [n_1 \neq \perp]_2)) \right).$$

PROPOSITION 7.5 (RCU 🧠). *The proof outline in Figure 15 is valid under MSGPASSING.*

8 RELATED WORK

Our proofs about β -stability, and preservation of various assertions under interference, make heavy use of semi-commutation. The importance of semi-commutations in concurrency reasoning has long been recognised in the setting of SC memory [41, 43]. This includes works exploring the connections between weakest (liberal) preconditions, invariants and semi-commutations [41]. Our article revives such ideas in the context of weak memory models.

Concurrency verification for weak memory models has been intensively studied in the recent years. An overview on verification techniques, decidability questions and semantics is provided by Lahav for causally consistent memory [37]. Most existing verification techniques are specific

to memory models, e.g., to TSO [12, 16, 57], PSO [1, 8, 21],¹⁰ POWER [3, 4, 9], or various subsets of C11 [2, 5, 35, 38, 55]. Many of these models are based on declarative techniques defined at the level of the *semantics* of the languages, and hence cannot be used to reason directly over program code.

To enable reasoning about program code, a number of papers have explored the use of operational semantics for weak memory. Early operational models, e.g., those for TSO [47] encode low-level details such as store buffers of threads into the program state. Although precise, such models involve fine-grained interleaving of micro-architectural details making program verification very challenging. Others have developed methods for incrementally generating executions fulfilling predicates that are consistent with the declarative semantics, e.g., References [34, 42, 46], sometimes employing *promises* about future events. These are accompanied by specific logics, in particular, variants of separation logic [25, 26, 56, 58].

In the context of RC11-RAR, the idea of view-based assertions has been proposed by Doherty et al. [24] and has been used in an **Owicki-Gries (OG)** proof calculus [19, 20]. OG-based reasoning has also been proposed by Lahav and Vafeiadis [38]. They employ a non-standard interpretation of assertions on shared variables as well, but use a semantics based on execution graphs as states. View-based semantics for weak memory models have originally been proposed in References [33, 50], both for subsets of C11. Each of the works mentioned above is focussed on only one memory model, as opposed to our general approach, where the operational proofs apply to any instantiation of the abstract language.

Generic (parametric) verification techniques, independent of a weak memory model, have been proposed [7, 29, 36, 51]. They all build on the consistency relations employed in declarative semantics of weak memory. Alglave and Cousot [7] derive specific communication constraints necessary to get sound proofs of invariants, and later show particular memory models to fulfill these constraints. Their so called “pythia” variables reflect specific reads-from relations and thereby encode thread-specific values of shared variables. Although generic, their proofs are rather complex, as they embed the communication properties of each memory model directly into each assertion. This means that each proof requires them to check communication verification conditions in addition to local and global correctness (cf., Section 6). Moreover, their proofs are not supported by any mechanisation.

Ponce de León et al. [51] employ bounded model checking parametric in the underlying memory model (a technique they call “memory models as modules”). The memory model modules add specific constraints for SMT solver queries during BMC. Similarly, Kokologiannakis et al. [36] implement a tool called GenMC, incrementally generating execution graphs, and applying consistency checks during generation. These tools are focussed on model checking, and hence have not established any relationship to classical deductive reasoning techniques of Owicki-Gries as we have done.

Finally, there are tools (Herd [10], Memalloy [59], and the range of tools developed as part of the REMS project¹¹) that allow one to experiment with declarative consistency constraints, and thereby allow to test executions, compare memory models, and so on. These can also be classified as generic techniques, because they are not tied to a specific memory model. However, such tools have been developed to simulate, test and compare memory models as opposed to being tools for program verification.

Orthogonal to our article is a series of works that address more relaxed memory semantics that allow causality cycles [32, 34, 42, 49], e.g., the load-buffering litmus test, which our semantics (like Reference [39]) disallows.

¹⁰The works cited here sometimes apply to more than one memory model, but are still not generic.

¹¹<https://www.cl.cam.ac.uk/~pes20/rems/index-models.html>.

For hardware memory models, both declarative and operational semantics are well established (e.g., References [28, 52, 53]). However, the operational models either model low-level hardware features (e.g., buffers and caches), or depend on so-called “promising writes” that predict future executions. Both approaches make deductive reasoning difficult. Recent developments include the use of rely-guarantee reasoning to handle intra-thread reorderings permitted by weak memory [18]. Extensions of our work to abstractly capture such features remains a topic of future work.

For language models, including C11, weak memory semantics that appropriately handles load-buffering yet rules out thin-air reads has been an ongoing research topic [32, 34, 42, 49]. In recent work, Wright et al. [60] have shown that the “sequence-before” total order (which enforces program order in RC11) can be relaxed to the “semantic dependency” partial order of Paviotti et al. [49] in a manner that allows an Owicki-Gries logic based on thread views (cf. References [19, 20]) to be obtained. Since this change only affects enabledness of actions in the transition relation, we conjecture that abstractions of this work provide a pathway towards addressing memory models that are more relaxed than RC11.

9 CONCLUSIONS

This article presents a new foundational basis for deductive verification of weak memory models (based on weakest liberal preconditions). Our axiom scheme has been inspired by algebraic structures and is hierarchical, providing a fresh perspective on deductive verification under weak memory models.

Our framework treats the views of threads as first-class objects, and these determine whether a thread is capable of taking a particular transition, or performing a particular action. Certain actions (e.g., release-acquire and fences) achieve *synchronisation* across threads and these are reflected in axioms such as **MP** and **FNC**, where the view of a thread that synchronises with another is updated. Further abstraction is achieved via the dual notions of β -simulation and interference relations, which allow us to recover critical commutation properties offered by SC, and allow us to define notions such as β -stable properties that are stable under view shifts.

We have achieved abstraction over low-level, model specific details, and shown that it uniformly captures behaviours present in SC, TSO, and RC11-RAR. We have used this scheme to verify key litmus tests from the literature in a generic setting.

APPENDICES

A STORE BUFFERING (UNSYNCHRONISED)

We consider the program template and proof outline for an unsynchronised store buffering example in Figure 16, which comprises shared variables x and y (both initially 0). Thread 1 updates x to 1, then later reads y into r_1 . Thread 2 is similar. This program template is unsynchronised in the sense that there is no guarantee that the write of x in thread 1 (and, similarly, the write of y in thread 2) will be seen by thread 2 (and, similarly, by thread 1).

For this program, the only property we can guarantee is that both r_1 and r_2 have final values of either 0 or 1. We contrast this behaviour with that of Figure 14, which is synchronised by fences in both threads.

PROPOSITION A.1 (UNSYNCHRONISED STORE BUFFERING 🍷). *The proof outline in Figure 16 is valid under SHARED VARS.*

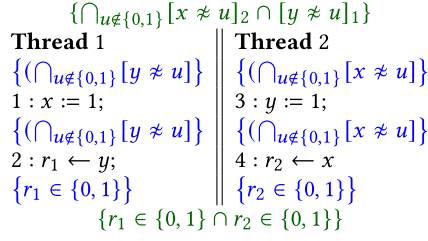


Fig. 16. Proof for unsynchronised store buffering.

B PROOF OF THEOREM 5.1 (SC SATISFIES MSGPASSING+FENCES)

Recall that the parameters for SC are instantiated as follows:

$$\begin{aligned}
T_{SC}(t, a) &= \xrightarrow{a, t}_{SC} \quad fence_{SC} = skip, & Act_{SC} &= \bigcup_{x \in Var_G, v \in Val} \{rd(x, v), wr(x, v), skip\}, \\
vmax_{SC}(t, a) &= \Sigma_{SC}, \quad sync_{SC} = Wr \times Rd \quad interf_{SC}(t, a) = \begin{cases} id & \text{if } a \in Rd \cup \{skip\} \\ \xrightarrow{a, t}_{SC} & \text{otherwise, i.e., } a \in Wr \end{cases}
\end{aligned}$$

We show this instantiation to satisfy all the axioms of MSGPASSING+FENCES. First, note that β for SC is the identity id (as $vmax$ is Σ_{SC}).

B.1 CORE axioms

C1 Trivial as $I \subseteq \Sigma_{SC}$.

C2 $vmax_{SC}(t, a) = \Sigma_{SC}$, hence can be represented by *true*. Since $\beta = id$, **C2** holds if

$$\sigma \xrightarrow{a, t}_{SC} \sigma' \Rightarrow \sigma \xrightarrow{a, t'}_{SC} \sigma',$$

which holds as all three rules of the operational semantics of SC are independent of thread ids.

C3 Case distinction:

- a is a read action or *skip*:
by rules SC-READ and SC-SKIP, we get $\xrightarrow{a, t}_{SC} \subseteq id$, hence $\xrightarrow{a, t}_{SC} \subseteq id \circ id \circ id$.
- a is a write action:
then $\xrightarrow{a, t}_{SC} \subseteq id \circ \xrightarrow{a, t}_{SC} \circ id$ holds.

C4 Case distinction:

- b is a read action or *skip*:
then $\Sigma_{SC} \subseteq \Sigma_{SC} = wlp(id, \Sigma_{SC})$.
- b is a write action:
then $\Sigma_{SC} \subseteq \Sigma_{SC} = wlp(\xrightarrow{b, t}_{SC}, \Sigma_{SC})$ as write actions are always enabled by rule SC-WRITE.

B.2 SHARED VARS axioms

SV1 Case distinction:

- b is a read action or *skip*:
then we trivially get $id \circ \xrightarrow{a, t}_{SC} \subseteq \xrightarrow{a, t}_{SC} \circ id$.

- both are write actions:
as $\sigma[x \mapsto v][y \mapsto u] = \sigma[y \mapsto u][x \mapsto v]$ for $x \neq y$ (which we have when $\text{var}(a) \neq \text{var}(b)$), we get $\xrightarrow{b,t'}_{SC} \circ \xrightarrow{a,t}_{SC} \subseteq \xrightarrow{a,t}_{SC} \circ \xrightarrow{b,t'}_{SC}$.
- a is a read action: then the property follows from $\sigma[x \mapsto u](y) = v$ iff $\sigma(y) = v$ for $x \neq y$.
- a is a skip action: trivial.

SV2 For b a read action, the property holds by interf_{SC} being id . For b a write action, the property holds, because writes are always enabled and hence $\text{wlp}(\xrightarrow{b,t'}_{SC}, \Sigma_{SC}) = \Sigma_{SC}$.

RW1 As $\text{var}(r) = \text{var}(w) = x$, we get $\xrightarrow{w,t'}_{SC} \circ \xrightarrow{r,t}_{SC} = \emptyset$ (the read cannot read a different value directly after the write) and hence $\xrightarrow{w,t'}_{SC} \circ \xrightarrow{r,t}_{SC} \subseteq \xrightarrow{r,t}_{SC} \circ \xrightarrow{w,t'}_{SC}$ trivially holds.

RW2 Trivial as $id \circ \xrightarrow{a,t}_{SC} \subseteq \xrightarrow{a,t}_{SC} \circ id$.

RW3 Trivial as $\Sigma_{SC} \subseteq \Sigma_{SC} = \text{wlp}(id, \Sigma_{SC})$.

RW4 Trivial as some value can always be read from every variable.

RW5 Holds, since all threads can read the value v from x directly after the write.

RW6 Note that $x \uparrow t = \bigcap_{a \in \text{Act}_x} \text{vmax}_{SC}(t, a)$, hence for SC is Σ_{SC} . Then

$$\begin{aligned}
 \bigcup_{v \in \text{Val}} [x \equiv v]_t &= \bigcup_{v \in \text{Val}} \bigcap_{u \neq v} [x \neq u]_t \\
 &= \bigcup_{v \in \text{Val}} \bigcap_{u \neq v} \text{dis}(T(t, \text{Rd}_x[u]), \emptyset) \\
 &= \bigcup_{v \in \text{Val}} \bigcap_{u \neq v} \text{wlp}(T(t, \text{Rd}_x[u]), \emptyset) \\
 &= \bigcup_{v \in \text{Val}} \bigcap_{u \neq v} \{\sigma \mid \sigma(x) \neq u\} \\
 &= \bigcup_{v \in \text{Val}} \{\sigma \mid \sigma(x) = v\} \\
 &= \Sigma_{SC}.
 \end{aligned}$$

RW7 Let $\text{wrval}(w) = \text{rdval}(r) = v$ and $t \neq t'$. Then $\text{wlp}(\xrightarrow{r,t'}_{SC}, \Sigma_{SC}) = \{\sigma \mid \sigma(x) = v\}$. Hence, $\text{wlp}(\xrightarrow{w,t}_{SC}, \text{wlp}(\xrightarrow{r,t'}_{SC}, \Sigma_{SC})) = \Sigma_{SC}$ and the property holds.

B.3 FENCE and MSGPASSING axioms

MP Follows from $\text{wlp}(\xrightarrow{w}_t, \text{wlp}(\xrightarrow{r}_{t'}, \Sigma)) = \Sigma$ for $\text{var}(w) = \text{var}(r)$ and $\text{wrval}(w) = \text{rdval}(r)$.

FNC Holds as $\text{wlp}(id, \Sigma_{SC}) = \Sigma_{SC}$.

C PROOF OF THEOREM 5.3 (PTSO SATISFIES MSGPASSING+FENCES)

We first recap the instantiations for PTSO. $\text{Act}_{PTSO} = \bigcup_{x \in \text{Var}_G, v \in \text{Val}} \{\text{rd}(x, v), \text{wr}(x, v), \text{fence}\}$ and

$$\begin{aligned}
 \Sigma_{PTSO} &= (\text{Var}_G \rightarrow \text{Val}) \times (\text{Tid} \rightarrow (\text{Var}_G \times \text{Val} \times \mathbb{Q})^*), & T_{PTSO}(t, a) &= FL \circ \xrightarrow{a}_t, \\
 I_{PTSO} &= \{(s, \text{wb}) \mid s \in \text{Var}_G \rightarrow \text{Val} \wedge \forall t. \text{wb}_t = \langle \rangle\}, & \text{fence}_{PTSO} &= \text{fence}, \\
 \text{vmax}_{PTSO}(t, a) &= \begin{cases} \text{maxTS}(t, \text{var}(a)) & \text{if } a \in \text{Wr} \cup \text{Rd} \\ \Sigma_{PTSO} & \text{otherwise, i.e., } a = \text{fence} \end{cases}, & \text{sync}_{PTSO} &= \text{Wr} \times \text{Rd}, \\
 \text{interf}_{PTSO}(t, a) &= \begin{cases} id & \text{if } a \in \text{Rd} \cup \{\text{fence}\} \\ \text{ins}(t, a) \circ FL & \text{if } a \in \text{Wr} \end{cases}.
 \end{aligned}$$

We show this instantiation to satisfy all the axioms of MSGPASSING+FENCES. Recall that $FL \triangleq (\bigcup_{t \in \text{Tid}} \xrightarrow{\text{flush}}_t)^*$ and note that β for PTSO contains FL .

C.1 CORE axioms

C1 $I \subseteq \text{vmax}_{\text{PTSO}}(t, a)$: Case distinction:

- $a = \text{fence}$: then it holds, because $\text{vmax}_{\text{PTSO}}(t, \text{fence}) = \Sigma_{\text{PTSO}}$.
- $a = \text{rd}(x, v)$ or $a = \text{wr}(x, v)$: in I all write buffers are empty, hence we have $\text{maxTS}(t', x')$ for all t', x' . As $\text{vmax}_{\text{PTSO}}(t, a) = \text{maxTS}(t, x)$, the property holds.

C2 We need to show $\sigma \in \text{vmax}_{\text{PTSO}}(t, a) \wedge (\sigma, \sigma') \in T_{\text{PTSO}}(t, a) \Rightarrow \exists \sigma'', \tau \in \Sigma. (\sigma, \sigma'') \in T_{\text{PTSO}}(t', a) \wedge (\sigma', \tau) \in \beta \wedge (\sigma'', \tau) \in \beta$. Case distinction:

- $a = \text{fence}$: this means that t 's write buffer is empty in σ' . σ'' is obtained by flushing all the buffers until t 's write buffer is empty, τ is reached by make flushes as to unify σ' and σ'' .
- $a = \text{rd}(x, v)$:
 - Either $\sigma \in \text{vmax}_{\text{PTSO}}(t, a)$, because $x \notin \sigma.\text{wb}_{t'}$ for all t' . Then $\sigma.s(x) = v$ and both t and t' can read v and the same flushes are being done before (or after) the read.
 - Or $\text{vmax}_{\text{PTSO}}(t, a)$, because t contains an entry for x with the largest timestamp q , i.e., $(x, v, q) \in \sigma.\text{wb}_t$. Then σ'' is obtained by flushing write buffer entries until finally (x, v, q) gets flushed and τ is obtained by either doing further flushes in σ' or in σ'' .
- $a = \text{wr}(x, v)$. Both $\xrightarrow{a, t}_{\text{PTSO}}$ and $\xrightarrow{a, t'}_{\text{PTSO}}$ can choose a fresh timestamp, and in particular the same one. Hence, for reaching τ , we need to flush until the write to x done by $\xrightarrow{a, t}_{\text{PTSO}}$ or $\xrightarrow{a, t'}_{\text{PTSO}}$ enters shared memory.

C3 We need to show: $T_{\text{PTSO}}(t, a) \subseteq \beta \circ \text{interf}_{\text{PTSO}}(t, a) \circ \beta$. Note that $T_{\text{PTSO}}(t, a) = FL \circ \xrightarrow{a, t}_{\text{PTSO}}$. Then the property holds, because $\xrightarrow{a, t}_{\text{PTSO}} \subseteq \text{interf}_{\text{PTSO}}(t, a)$ for all actions a .

C4 We need to show: $\text{vmax}_{\text{PTSO}}(t, a) \subseteq \text{wlp}(\text{interf}_{\text{PTSO}}(t, b), \text{vmax}_{\text{PTSO}}(t, a))$. Case distinction:

- $a = \text{fence}$: then $\text{vmax}_{\text{PTSO}}(t, a) = \Sigma_{\text{PTSO}}$ and $\text{wlp}(\text{interf}_{\text{PTSO}}(t, b), \Sigma) = \Sigma$ for all actions b .
- $a = \text{rd}(x, v)$: property holds for $b = \text{fence}$ or $b = \text{wr}(y, u)$ for some $y \neq x$. If $b = \text{wr}(x, u)$, then $\text{interf}_{\text{PTSO}}(t, b)$ could insert (x, u, q) into the write buffer of t with some larger timestamp than (x, v, q') . Still, we have $\text{vmax}_{\text{PTSO}}(t, a)$ then, because $\text{vmax}_{\text{PTSO}}(t, \text{rd}(x, v)) = \text{maxTS}(t, x)$ and thus independent of the actual value v .

C.2 CORE axioms

SV1 We need to show: if $\text{var}(a) \neq \text{var}(b)$, then $\text{interf}_{\text{PTSO}}(t', b) \circ T_{\text{PTSO}}(t, a) \subseteq T_{\text{PTSO}}(t, a) \circ \text{interf}_{\text{PTSO}}(t', b)$. Case distinction on type of b :

- $b = \text{fence}$ or $b = \text{rd}(x, v)$: trivial as $\text{interf}_{\text{PTSO}}(t', b) = \text{id}$.
- $b = \text{wr}(x, v)$: if $\xrightarrow{a, t}_{\text{PTSO}}$ would first require some flushing, then this is possible on both sides as $T_{\text{PTSO}}(t, a) = FL \circ \xrightarrow{a, t}_{\text{PTSO}}$. The only non-trivial case is when $t = t'$ and $a = \text{wr}(y, u)$. However, as $\text{interf}_{\text{PTSO}}(t, b)$ can insert at arbitrary positions in the write buffer, the proof holds.

SV2 We need to show: $\text{vmax}_{\text{PTSO}}(t, a) \subseteq \text{wlp}(\text{interf}_{\text{PTSO}}(t', b), \text{vmax}_{\text{PTSO}}(t, a))$. Case distinction on type of a :

- $a = \text{fence}$, then $\text{vmax}_{\text{PTSO}}(t, a) = \Sigma_{\text{PTSO}}$ and property holds as $\text{wlp}(\text{interf}_{\text{PTSO}}(t', b), \Sigma_{\text{PTSO}}) = \Sigma_{\text{PTSO}}$.
- $a \in \text{Wr} \cup \text{Rd}$, then $\text{vmax}_{\text{PTSO}}(t, a) = \text{maxTS}(t, \text{var}(a))$. The property then follows, because $\text{var}(a) \neq \text{var}(b)$, thus $\text{maxTS}(t, \text{var}(a))$ remains the same.

RW1 We need to show: $\text{interf}_{PTSO}(t', w) \circ T_{PTSO}(t, r) \subseteq T_{PTSO}(t, r) \circ \text{interf}_{PTSO}(t', w)$.

- If $t \neq t'$, then the property holds, because $\text{interf}_{PTSO}(t', w)$ inserts into t' 's write buffer, whereas r reads from t 's write buffer.
- If $t = t'$, then the property holds, because $\text{rdval}(r) \neq \text{wrval}(w)$ (so r does not read the value written by w). Hence, ins (of w) either inserts somewhere in the middle of t 's write buffer (and then the read can read from the end), or at the end (in this case r reads from main memory or the write buffer of another thread once flushed to main memory. The latter is possible as $T_{PTSO}(t, r)$ is defined as $FL \circ \xrightarrow{a, t}_{PTSO}$.

RW2 Holds, as $\text{interf}_{PTSO}(t, r) = \text{id}$ for read actions r .

RW3 Holds, as $\text{interf}_{PTSO}(t, r) = \text{id}$ for read actions r .

RW4 Holds, because $\text{val}_\sigma(t, x)$ is totally defined (there is always a value that can be read).

RW5 Holds, because $T_{PTSO}(t, w)$ inserts the value at the end of t 's write buffer where t can directly read it.

RW6 We have to show that $x_{\uparrow t} \subseteq \bigcup_{v \in \text{Val}} [x \equiv v]_t$. First note that $x_{\uparrow t} = \text{maxTS}(t, \text{var}(a))$ as writes and reads have the same vmax in PTSO. Furthermore, $[x \equiv v]_t$ holds if either

- $x \notin \text{wb}_{t'}$ for all threads t' and $s(x) = v$, or
- there exists q such that $\langle x, v, q \rangle \in \text{wb}_t$ and for all t', q' : if $\langle x, u, q' \rangle \in \text{wb}_{t'}$ for some $u \neq v$, then $q' < q$.

Thus, we have $\text{maxTS}(t, \text{var}(a))$ in both cases.

RW7 Let $\text{wrval}(w) = v$. We first consider $\text{vmax}_{PTSO}(t, w) \cap \text{dis}(T_{PTSO}(t', r))$. Since $w \in \text{Wr}$, we have that $\text{vmax}_{PTSO}(t, w) = \text{maxTS}(t, x)$ (for $x = \text{var}(w)$). For $\text{maxTS}(t, x)$, we have three cases:

- (1) $\text{maxTS}(t, x) = \emptyset$ (because either (1) there is an entry for x in t 's write buffer, but it does not have the maximal timestamp, or (2) there is no entry for x in t 's write buffer, but in the write buffer of another thread), then we are done.
- (2) $\sigma \in \text{maxTS}(t, x)$ and $x \in \sigma.\text{wb}_t$ and its timestamp is maximal over all timestamps for x , but then $\text{dis}(T_{PTSO}(t', r)) = \emptyset$, as the entry in t 's write buffer could first be flushed and then read by t' . Again, we are done.
- (3) $\sigma \in \text{maxTS}(t, x)$ and $x \notin \sigma.\text{wb}_{t'}$ for all t' (including t) and $\sigma.s(x) \neq v$. In that case, t could now execute w (via $T_{PTSO}(t, w)$), then all the write buffer entries with timestamps less than that of w get flushed and finally w , which then gets read by t' (all steps in $T_{PTSO}(t', r)$). After that, no write buffer contains an entry for x anymore, and hence $\text{maxTS}(t', x) = \text{vmax}_{PTSO}(t, a)$.

C.3 MSGPASSING and FENCE axioms

MP Case distinction:

- $b = \text{fence}$: then $\text{vmax}_{PTSO}(t', b) = \Sigma_{PTSO}$, hence the right-hand side of the \subseteq is Σ_{PTSO} and thus trivially fulfilled.
- $b \in \text{Wr} \cup \text{Rd}$: There are several cases for $\text{vmax}_{PTSO}(t, \text{var}(b)) = \text{maxTS}(t, \text{var}(b))$.
 - (1) $\text{maxTS}(t, \text{var}(b)) = \emptyset$, and then we are done.
 - (2) For all t' (including t) $\text{var}(b) \notin \sigma.\text{wb}_{t'}$. Then $\sigma \in \text{maxTS}(t', \text{var}(b))$ holds as well, and this implies that after w and r there is still no entry for $\text{var}(b)$ in any of the write buffers (as $\text{var}(b) \neq \text{var}(w)$), so still $\sigma \in \text{maxTS}(t', \text{var}(b))$.
 - (3) $\text{var}(b) \in \sigma.\text{wb}_t$ and the timestamp in $\sigma.\text{wb}_t$ is maximal for all timestamps for $\text{var}(b)$. For $\sigma \in \text{wlp}(T_{PTSO}(t', r), \text{maxTS}(t', \text{var}(b)))$, the read r needs to be able to read from $\text{var}(r)$ after flushing all $\sigma.\text{wb}_{t'}$ entries to $\text{var}(b)$ to main memory (to get $\text{maxTS}(t', \text{var}(b))$ afterwards). Hence, the timestamps for $\text{var}(b)$ in write buffers are all less than the timestamp

for writes to $var(r)$ with value $rdval(r)$. The write w (on the right-hand side (RHS)) is now inserted into wb_t with a timestamp larger than all other timestamps in wb_t , in particular larger than the timestamp for $var(b)$. Thus, $T_{PTSO}(t', r)$ (on the RHS) can do the same flushes, then $\xrightarrow{t', r}_{PTSO}$ and thereby again reach $maxTS(t', r)$.

FNC Case distinction.

- If $a = fence$, then the property trivially holds as $vmax_{PTSO}(t', a) = \Sigma_{PTSO}$.
- $a \in Wr \cup Rd$: Again, there are three cases for $vmax_{PTSO}(t, a)$.
 - (1) $vmax_{PTSO}(t, a)$ is empty (because either (1) there is an entry for $var(a)$ in t 's write buffer, but it does not have the maximal timestamp, or (2) there is no entry for $var(a)$ in t 's write buffer, but in the write buffer of another thread), then we are done.
 - (2) $var(a) \notin \sigma.wb_{t'}$ for all threads t' (including t). Then the fence action can directly execute to result in state σ' , and afterwards we still have $var(a) \notin \sigma'.wb_{t'}$ for all threads t' .
 - (3) $var(a) \in \sigma.wb_t$ and the timestamp of a for t is maximal. In that case, $T_{PTSO}(t, fence)$ first needs to empty the entire write buffer of t . Due to flushing in order of timestamps, all other writes to $var(a)$ also gets flushed (they have smaller timestamps), hence we afterwards have $maxTS(t', a)$ as no write buffer contains an entry for $var(a)$ anymore.

D PROOF OF THEOREM 5.4 (RC11 SATISFIES MSGPASSING)

We recap the instantiations for RC11-RAR:

$$\begin{aligned}
 \Sigma_{C11} &\triangleq TView \times MView \times Writes, \\
 T_{C11}(t, a) &\triangleq \xrightarrow{a, t}_{C11}, \\
 I_{C11} &\triangleq \left\{ (tview, mview, writes) \mid \exists F \in Var_G \rightarrow Val. \text{let } I_x = (wr(x, F(x)), 0) \text{ in} \right. \\
 &\quad \left. \begin{aligned} &writes = \{I_x \mid x \in Var_G\} \wedge \\ &tview = \lambda t \in Tid. \lambda x \in Var_G. I_x \wedge \\ &mview = \lambda w \in writes. \lambda x \in Var_G. I_x \end{aligned} \right\}, \\
 sync_{C11} &\triangleq Wr_R \times Rd_A, \\
 vmax_{C11}(t, a) &\triangleq \{\sigma \mid \forall (b, q) \in \sigma.writes. var(a) = var(b) \Rightarrow q \leq tst(\sigma.tview_t(var(a)))\}, \\
 interf_{C11}(t, a) &\triangleq \text{if } a \in Rd \text{ then } id \text{ else } ins_w(t, a).
 \end{aligned}$$

The relation β for RC11 arbitrarily advances the views of the system components (thread or a write). Thus, it includes the following transitions:

$$\begin{aligned}
 advanceTV_t(\sigma, \sigma') &= \exists(a, q) \in \sigma.writes. \text{let } x = var(a), tview' = \sigma.tview_t[x := (a, q)] \text{ in} \\
 &\quad q \geq tst(\sigma.tview_t(x)) \wedge \sigma' = \sigma[t := tview'], \\
 advanceMV_w(\sigma, \sigma') &= \exists(a, q) \in \sigma.writes. \text{let } x = var(a), mview' = \sigma.mview_w[x := (a, q)] \text{ in} \\
 &\quad q \geq tst(\sigma.mview_w(x)) \wedge \sigma' = \sigma[w := mview'].
 \end{aligned}$$

D.1 CORE axioms

C1 ($vmax$ initialisation). By definition. In the initial state, for every variable every thread's view points at the sole write.

C2 ($vmax$ maximal determinism). Fix $\sigma \in vmax(t, a)$ and σ' such that $(\sigma, \sigma') \in T(t, a)$. Let $x = var(a)$ and w be the write that a interacts with. There are two cases.

- (1) a is a read. Then a reads from w . Let τ be σ' but with the view of t' advanced using $advanceTV_{t'}$ to “match” that of t in σ' .
 - For the first conjunct of the consequent of **C2**, we can reach the state τ from σ' by advancing the view of t' , i.e., $(\sigma', \tau) \in advanceTV_{t'}$. Note that if $(w, a) \in sync_{C11}$, then for all $y \neq var(a)$, we must update $tview_{t'}(y)$ to $max(\sigma.tview_{t'}(y), \sigma.mview_w(y))$, since

reading from w induces a release-acquire synchronisation. Since this involves repeated application of $advanceTV_{t'}$ only, we have $(\sigma, \tau) \in \beta$.

- For the second conjunct, note that $\sigma \in v\max(t, a)$ implies that for any thread t' , t' 's view of x is no later than that of t . Therefore, t' can also observe w in σ . Now, for some σ'' , pick σ'' such that $(\sigma, \sigma'') \in T_{C11}(t', a)$ and σ'' is the state after t' executes a so that it reads from w . Now, use $advanceTV_t$ to move t 's view forward as needed, until σ'' is identified with τ . Since $(\sigma, \sigma'') \in T_{C11}(t', a)$ and $(\sigma'', \tau) \in \beta$, we have $(\sigma, \tau) \in T_{C11}(t', a) \circ \beta$.
- (2) a is a write. Then a is introduced immediately after w in timestamp order. Let q be the timestamp of a and $u = (a, q)$. Let τ be σ' but with $mview_u(y)$, $tview_t(y)$ and $tview_{t'}(y)$ set to $\max(\sigma'.tview_t(y), \sigma'.tview_{t'}(y))$. This involves application of $advanceMV_u$, $advanceTV_t$ and $advanceTV_{t'}$. Hence, $(\sigma', \tau) \in \beta$ (satisfying the first conjunct of the consequent of C2).

As before, since $\sigma \in v\max_{C11}(t, a)$, t' can also interact with w . For some state σ'' , suppose $(\sigma, \sigma'') \in T_{C11}(t', a)$. Now, we can use $advanceMV_u$, $advanceMV_t$ and $advanceMV_{t'}$ to update $mview_u(y)$ for all y to $\max(\sigma''.tview_t(y), \sigma''.tview_{t'}(y))$. Note that

$$\max(\sigma''.tview_t(y), \sigma''.tview_{t'}(y)) = \max(\sigma'.tview_t(y), \sigma'.tview_{t'}(y)).$$

Thus, $(\sigma'', \tau) \in \beta$, and hence $(\sigma, \tau) \in T_{C11}(t', a) \circ \beta$.

C3 (Over-approximation). There are two cases.

- (1) a is a read. Then each transition in $T_{C11}(t, a)$ only changes t 's view. On the right-hand side, we can use the first β , in particular $advanceTV_t$ to match this view update. Since $interf_{C11}(t, a) = id$ and since β is reflexive, we choose the second instance of β to be id . Thus, $T_{C11}(t, a) \subseteq \beta \circ interf_{C11} \circ \beta$.
- (2) a is a write. Then each transition in $T_{C11}(t, a)$ inserts a write into the state (as ins_W does). Let $x = var(a)$, $(\sigma, \sigma') \in T_{C11}(t, a)$, q be the timestamp of a in σ' and $u = (a, q)$. Suppose (σ, σ') interacts with a write w .

On the RHS, we can use β to advance the view of t to w , yielding a new state τ . Then use $ins_W(t, a)$ to introduce a with timestamp q into τ , yielding a new state τ' . We pick the timestamp of a in τ' to be q . Note that ins_W only advances t 's view on x to u . Moreover, $\tau'.mview_u(x) = u$ and for all $y \neq x$, $\tau'.mview_u(y) = first_{\tau'}(y)$. Thus, we can use the second β to advance $\tau'.mview_u(y)$ for all y to equal $\sigma'.mview_u(y)$. Hence, $(\tau', \sigma') \in \beta$, i.e., $(\sigma, \sigma') \in \beta \circ interf_{C11} \circ \beta$.

C4 (View stability). If a is a read, then $interf_{C11}(t, a) = id$, so we are done. If a is a write, then because \otimes takes the maximal value from each view, we are guaranteed to get to preserve view maximality.

D.2 SHARED VARS axioms

SV1 (Action independence). We perform case analysis on b .

- (1) b is a read. Then $interf_{C11}(t', b) = id$ so the result is trivial.
- (2) b is a write. Suppose $(\sigma, \sigma') \in interf_{C11}(t', b)$ and $(\sigma', \tau) \in T_{C11}(t, a)$. We perform case analysis on t .
 - If $t' \neq t$, then from t 's perspective, σ' contains a new write on $var(b)$. But, since $var(b) \neq var(a)$, a cannot interact with this new write in σ' . Thus, there exists a state σ'' such that $(\sigma, \sigma'') \in T_{C11}(t, a)$ such that
 - for all writes w in σ'' , $\sigma''.mview_w = \tau.mview_w$ and
 - for all $t'' \neq t'$, $\sigma''.tview_{t''} = \tau.tview_{t''}$
 - for all $x \neq var(b)$, $\sigma''.tview_{t'}(x) = \tau.tview_{t'}(x)$
 - $\sigma''.tview_{t'}(var(b)) = \sigma.tview_{t'}(var(b))$
 Thus, $(\sigma'', \tau) \in interf_{C11}(t', b)$, i.e., $(\sigma, \tau) \in T_{C11}(t, a) \circ interf_{C11}(t', b)$.

- If $t' = t$, then the case is similar, except that $\text{interf}_{C11}(b)$ can advance the view of $\text{var}(b)$. But the view of $\text{var}(a)$ is unchanged, so $T_{C11}(t, a)$ is always possible.

SV2 (Action view stability). As usual, the case when b is a read is trivial.

Let b be a write. Since $\text{var}(a) \neq \text{var}(b)$, $\text{vmax}_{C11}(t, a)$ is preserved for any t , even when $\text{interf}_{C11}(t', b)$ installs a maximal write on $\text{var}(b)$.

RW1 (Write independence). Since $\text{rdval}(r) \neq \text{wrval}(w)$, r does not interact with (i.e., read from) w . Suppose $(\sigma, \tau) \in \text{interf}_{C11}(t', w) \circ T_{C11}(t, r)$ where r reads from some $w \neq w'$. Thus, there exists a σ' such that $(\sigma, \sigma') \in T_{C11}(t, r)$ where σ' is the state after r reads from w' in σ . Clearly, we can introduce w in σ' in the same location way as in σ , and doing so results in τ . Thus, $(\sigma, \tau) \in T_{C11}(t, r) \circ \text{interf}_{C11}(t', w)$.

RW2 (Read independence). Trivial, since $\text{interf}_{C11}(t, r) = \text{id}$.

RW3 (Read view stability). Trivial, since $\text{interf}_{C11}(t, r) = \text{id}$.

RW4 (Read enabledness). The right-hand side clearly contains every possible state configuration.

RW5 (Write visibility). A new write is always in the writing thread's visible-writes set.

RW6 (Weak coherence). Since thread t is view maximal on x , all reads of x must only interact with the last write (say w) of x . Thus, all reads of x are guaranteed to return a single value, namely, $\text{wrval}(w)$.

RW7 (Same variable synchronisation). Note that predicates $P \triangleq \text{vmax}_{C11}(t, w) \cap \text{dis}(T_{C11}(t', r))$ and $Q \triangleq \text{vmax}_{C11}(t', a)$ describe the pre- and post-conditions of the transition relation $T_{C11}(t, w) \circ T_{C11}(t', r)$, respectively. Moreover, if $\sigma \in \text{vmax}_{C11}(t, w)$, then $\sigma.\text{tview}_t(\text{var}(w)) = \text{last}_\sigma(x)$, where $\text{last}_\sigma(x)$ returns the last write to x in σ .

For each $\sigma \in P$, we have that

- t is view-maximal on w , and
- r is disabled for t' .

Since we assume $\text{wrval}(w) = \text{rdval}(r)$ and $\text{var}(r) = \text{var}(w) = x$, there is no write $w' \in \text{OW}_\sigma(t', x)$ such that $\text{wrval}(w') = \text{rdval}(r)$. For any σ' such that $(\sigma, \sigma') \in T_{C11}(t, w)$, since t is view-maximal on w , the timestamp of w in σ' is maximal w.r.t. all writes to x in σ' . Moreover, w is observable to t' in σ' . Thus, for any σ'' , such that $(\sigma', \sigma'') \in T_{C11}(t', r)$, we have that $\sigma''.\text{tview}_{t'}(x)$ is maximal, since r must read from w , i.e., $\sigma'' \in \text{vmax}_{C11}(t', a)$.

D.3 MSGPASSING axiom

MP (Message passing). In this proof, let $x = \text{var}(w) = \text{var}(r)$ and $y = \text{var}(b)$.

Note that $P \triangleq \text{vmax}_{C11}(t, b) \cap \text{wlp}(T_{C11}(t', r), \text{vmax}_{C11}(t', b))$ and $Q \triangleq \text{vmax}_{C11}(t', b)$ define the pre and post states of the transition relation $T(t, w) \circ T(t', r)$, respectively. Further note that for any σ , if $\sigma \in \text{wlp}(T(t', r), \text{vmax}_{C11}(t', b))$, then for any σ' , such that $(\sigma, \sigma') \in T(t', r)$, we have $\sigma' \in \text{vmax}_{C11}(t', b)$.

We perform case analysis on the enabledness of $T_{C11}(t', r)$, i.e., on the existence of a write that r can read from.

- (1) There exists a write $u \in \text{OW}_\sigma(t', x)$ such that $\text{rdval}(r) = \text{wrval}(u)$. By P , for any σ' , if $(\sigma, \sigma') \in T_{C11}(t', r)$, we have $\sigma' \in \text{vmax}_{C11}(t', b)$.

Now consider the states after execution of $T(t, w)$, i.e., $(\sigma, \tau) \in T(t, w)$. There are two possibilities.

- $(w, _) \notin \text{OW}_\tau(t', x)$. In this case, $\text{OW}_\tau(t', x) = \text{OW}_\sigma(t', x)$, thus, using P , for all τ' , such that $(\tau, \tau') \in T(t', r)$, we have $\tau' \in \text{vmax}_{C11}(t', r)$.

- $(w, _) \in OW_\tau(t', x)$. In this case, $OW_\tau(t', x) = OW_\sigma(t', x) \cup \{(w, _)\}$. Since $\sigma \in v\max_{C11}(t, b)$, $\tau.mview_{(w, _)}(y) = last_\tau(y)$. For $(\tau, \tau') \in T_{C11}(t', r)$, there are two further possibilities.
 - r reads from a write different from w , i.e., a write in $OW_\sigma(t', x)$. In this case, using P , we have $\tau' \in v\max_{C11}(t, b)$.
 - r reads from w . In this case, since $(w, r) \in sync_{C11} = Wr_R \times Rd_A$ and thus by definition of \otimes , $\tau'.tview_{t'}(y) = last_{\tau'}(y)$, i.e., $\tau' \in v\max_{C11}(t', b)$.
- (2) For all $u \in OW_\sigma(t', x)$, we have $rdval(r) \neq wrval(u)$, i.e., $dis(T_{C11}(t', r))$. Thus, $\sigma \in P$ iff $\sigma \in v\max_{C11}(t, b)$. Now consider the state τ such that $(\sigma, \tau) \in T(t, w)$. As above, there are two possibilities:
 - $(w, _) \notin OW_\tau(t', x)$. Then the transition $T(t', r)$ remains disabled, i.e., there is no τ' such that $(\tau, \tau') \in T(t', r)$, and the result holds trivially.
 - $(w, _) \in OW_\tau(t', x)$. Then, $(\tau, \tau') \in T_{C11}(t', r)$ iff r reads from w . In this case, since $(w, r) \in sync_{C11} = Wr_R \times Rd_A$ and thus by definition of \otimes , $\tau'.tview_{t'}(y) = last_{\tau'}(y)$, i.e., $\tau' \in v\max_{C11}(t', b)$.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

- [1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless model checking for TSO and PSO. *Acta Inf.* 54, 8 (2017), 789–818.
- [2] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *Proceedings of the PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1117–1132. <https://doi.org/10.1145/3314221.3314649>
- [3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017. Context-bounded analysis for POWER. In *Proceedings of the TACAS (Lecture Notes in Computer Science, Vol. 10206)*, Axel Legay and Tiziana Margaria (Eds.). 56–74. https://doi.org/10.1007/978-3-662-54580-5_4
- [4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless model checking for POWER. In *Proceedings of the CAV (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 134–156. https://doi.org/10.1007/978-3-319-41540-6_8
- [5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 135:1–135:29. <https://doi.org/10.1145/3276505>
- [6] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (1996), 66–76. <https://doi.org/10.1109/2.546611>
- [7] Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: An invariance proof method for weak consistency models. In *Proceedings of the POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 3–18.
- [8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software verification for weak memory via program transformation. In *Proceedings of the ESOP (LNCS, Vol. 7792)*, M. Felleisen and P. Gardner (Eds.). Springer, 512–532.
- [9] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial orders for efficient bounded model checking of concurrent software. In *Proceedings of the CAV (LNCS, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 141–157.
- [10] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. <https://doi.org/10.1145/2627752>
- [11] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs*. Springer.
- [12] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. 2011. Getting rid of store-buffers in TSO analysis. In *CAV (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 99–115. https://doi.org/10.1007/978-3-642-22110-1_9
- [13] Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *Proceedings of the POPL*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 235–248. <https://doi.org/10.1145/2429069.2429099>

- [14] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the POPL*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. <https://doi.org/10.1145/1926385.1926394>
- [15] Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. 2022. View-based Owicki-Gries reasoning for persistent x86-TSO. In *Proceedings of the ESOP (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 234–261. https://doi.org/10.1007/978-3-030-99336-8_9
- [16] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent library correctness on the TSO memory model. In *Proceedings of the ESOP (LNCS, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 87–107. https://doi.org/10.1007/978-3-642-28869-2_5
- [17] Kyeongmin Cho, Sung Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping hardware persistency models: View-based and axiomatic persistency models for Intel-x86 and Armv8. In *Proceedings of the PLDI*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 16–31. <https://doi.org/10.1145/3453483.3454027>
- [18] Nicholas Coughlin, Kirsten Winter, and Graeme Smith. 2021. Rely/guarantee reasoning for multicopy atomic weak memory models. In *Proceedings of the FM (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 292–310. https://doi.org/10.1007/978-3-030-90870-6_16
- [19] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-gries reasoning for C11 RAR. In *Proceedings of the ECOOP (LIPIcs)*, Robert Hirschfeld (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- [20] Sadegh Dalvandi, Brijesh Dongol, Simon Doherty, and Heike Wehrheim. 2022. Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. *J. Autom. Reason.* 66, 1 (2022), 141–171. <https://doi.org/10.1007/s10817-021-09610-2>
- [21] Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. 2017. Effective abstractions for verification under relaxed memory models. *Comput. Lang. Syst. Struct.* 47 (2017), 62–76. <https://doi.org/10.1016/j.cl.2016.02.003>
- [22] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.* 23, 2 (2012), 375–382. <https://doi.org/10.1109/TPDS.2011.159>
- [23] Simon Doherty, Sadegh Dalvandi, Brijesh Dongol, and Heike Wehrheim. 2022. Isabelle/HOL files for “Unifying Operational Weak Memory Verification: An Axiomatic Approach.” Retrieved from <https://doi.org/10.6084/m9.figshare.19779415>.
- [24] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *Proceedings of the PPOPP*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 355–365. <https://doi.org/10.1145/3293883.3295702>
- [25] Marko Doko and Viktor Vafeiadis. 2016. A program logic for C11 memory fences. In *Proceedings of the VMCAI (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20
- [26] Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *Proceedings of the ESOP (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17
- [27] Brijesh Dongol, Ian J. Hayes, and Georg Struth. 2016. Convolution as a unifying concept: Applications in separation logic, interval calculi, and concurrency. *ACM Trans. Comput. Log.* 17, 3 (2016), 15:1–15:25. <https://doi.org/10.1145/2874773>
- [28] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the POPL*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 608–621. <https://doi.org/10.1145/2837614.2837615>
- [29] Natalia Gavrilenco, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for weak memory models: Relation analysis for compact SMT encodings. In *Proceedings of the CAV (LNCS, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 355–365. https://doi.org/10.1007/978-3-030-25540-4_19
- [30] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.
- [31] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2011. Concurrent Kleene algebra and its foundations. *J. Log. Algebraic Methods Program.* 80, 6 (2011), 266–296. <https://doi.org/10.1016/j.jlap.2011.04.005>
- [32] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: A simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. <https://doi.org/10.1145/3428262>
- [33] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *Proceedings of the ECOOP (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 17:1–17:29.
- [34] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189.

- [35] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32.
- [36] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 96–110.
- [37] Ori Lahav. 2019. Verification under causally consistent shared memory. *SIGLOG News* 6, 2 (2019), 43–56.
- [38] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries reasoning for weak memory models. In *Proceedings of the ICALP (LNCS, Vol. 9135)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer, 311–323.
- [39] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the PLDI*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. <https://doi.org/10.1145/3062341.3062352>
- [40] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691.
- [41] Leslie Lamport. 1990. *win* and *sin*: Predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 396–428.
- [42] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the PLDI*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 362–376. <https://doi.org/10.1145/3385412.3386010>
- [43] Richard J. Lipton. 1975. Reduction: A new method of proving properties of systems of processes. In *Proceedings of the POPL*, Robert M. Graham, Michael A. Harrison, and John C. Reynolds (Eds.). ACM Press, 78–86. <https://doi.org/10.1145/512976.512985>
- [44] J. Manson, W. Pugh, and S. V. Adve. 2005. The Java memory model. In *Proceedings of the POPL*. ACM, 378–391.
- [45] Bernhard Möller and Georg Struth. 2006. Algebras of modal operators and partial correctness. *Theor. Comput. Sci.* 351, 2 (2006), 221–239. <https://doi.org/10.1016/j.tcs.2005.09.069>
- [46] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. In *Proceedings of the OOPSLA*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 111–128.
- [47] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of the TPHOLS (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27
- [48] Susan S. Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Inf.* 6 (1976), 319–340.
- [49] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular relaxed dependencies in weak memory concurrency. In *Proceedings of the ESOP (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22
- [50] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. 2016. Operational aspects of C/C++ concurrency. Retrieved from arXiv:1606.01400.
- [51] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2018. BMC with memory models as modules. In *Proceedings of the FMCAD*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–9.
- [52] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A simpler and faster operational concurrency model. In *Proceedings of the PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1–15. <https://doi.org/10.1145/3314221.3314624>
- [53] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the PLDI*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. <https://doi.org/10.1145/1993498.1993520>
- [54] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. <https://doi.org/10.1145/1785414.1785443>
- [55] Alexander J. Summers and Peter Müller. 2018. Automating deductive verification for weak-memory programs. In *Proceedings of the TACAS (LNCS, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 190–209.
- [56] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *Proceedings of the ESOP (LNCS, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 357–384.
- [57] Oleg Travkin, Annika Mütze, and Heike Wehrheim. 2013. SPIN as a linearizability checker under weak memory models. In *Proceedings of the HVC (LNCS, Vol. 8244)*, Valeria Bertacco and Axel Legay (Eds.). Springer, 311–326.
- [58] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the OOPSLA*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 691–707.

- [59] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 190–204. Retrieved from <http://dl.acm.org/citation.cfm?id=3009838>.
- [60] Daniel Wright, Mark Batty, and Brijesh Dongol. 2021. Owicki-Gries reasoning for C11 programs with relaxed dependencies. In *Proceedings of the FM (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 237–254. https://doi.org/10.1007/978-3-030-90870-6_13

Received 12 October 2021; revised 29 April 2022; accepted 15 May 2022