Mechanised Operational Reasoning for C11 Programs with Relaxed Dependencies

DANIEL WRIGHT, University of Kent, United Kingdom
SADEGH DALVANDI, University of Surrey, United Kingdom
MARK BATTY, University of Kent, United Kingdom
BRIJESH DONGOL, University of Surrey, United Kingdom

Verification techniques for C11 programs have advanced significantly in recent years with the development of operational semantics and associated logics for increasingly large fragments of C11. However, these semantics and logics have been developed in a restricted setting to avoid the *thin-air-read* problem. In this article, we propose an operational semantics that leverages an intra-thread partial order (called *semantic dependencies*) induced by a recently developed denotational event-structure-based semantics. We prove that our operational semantics is sound and complete with respect to the denotational semantics. We present an associated logic that generalises a recent Owicki–Gries framework for RC11 RAR (repaired C11) with relaxed and release-acquire accesses. We describe the mechanisation of the logic in the Isabelle/HOL theorem prover, which we use to prove correctness of a number of examples.

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

Additional Key Words and Phrases: C11, Owicki-Gries, Verification, Modular Dependencies, Isabelle/HOL

ACM Reference format:
Daniel Wright, Sadegh Dalvandi, Mark Batty, and Brijesh Dongol. 2023. Mechanised Operational Reasoning for C11 Programs with Relaxed Dependencies. *Form. Asp. Comput.* 35, 2, Article 10 (June 2023), 27 pages.
https://doi.org/10.1145/3580285

1 INTRODUCTION

Significant advances have now been made on the semantics of (weak) memory models using a variety of axiomatic (declarative), operational and denotational techniques. Several recent works have therefore focussed on logics and associated verification frameworks for *reasoning* about program executions over weak memory models. These include specialised separation logics [11, 14, 19, 31] and adaptations of classical Owicki–Gries reasoning [9, 21]. At the level of languages, there has been a particular focus on C11 (the 2011 C/C++ standard).

Wright is supported by VeTSS. Batty is supported by EPSRC grants EP/V000470/1 and EP/R032971/1, and the Royal Academy of Engineering. Dalvandi is supported by EPSRC grant EP/R032556/1. Dongol is supported by EPSRC grants EP/V038915/1, EP/R032556/1, EP/R025134/2, VeTSS and ARC Discovery Grant DP190102142.
Authors' addresses: D. Wright and M. Batty, University of Kent, Canterbury CT2 7NZ, United Kingdom; emails: daw29@kent.ac.uk, mbatty@cantab.net; S. Dalvandi and B. Dongol, University of Surrey, Guildford GU2 7XH, United Kingdom; emails: {m.dalvandi, b.dongol}@surrey.ac.uk.

0934-5043/2023/06-ART10 $15.00
https://doi.org/10.1145/3580285

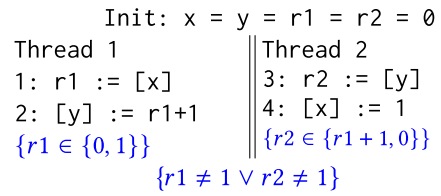

Fig. 1. Load-buffering with a semantic dependency.

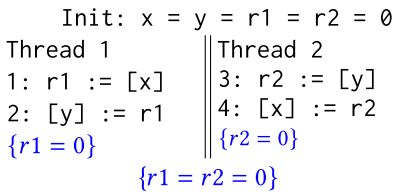

Fig. 2. Load-buffering with two dependencies.

Due to the complexity of C11 [5], many reasoning techniques have restricted themselves to particular fragments of the language by only allowing certain types of memory accesses and/or reordering behaviours. Several works (e.g., References [9, 13, 19, 21]) assume a memory model that guarantees that program order (sequenced-before order) is maintained [22]. While this restriction makes the logics and proofs of program correctness more manageable, it precludes reasoning about observable executions displaying certain real-world phenomena, e.g., those exhibited by the load-buffering litmus test under Power or ARMv7.

Load buffering and the thin-air problem. C11 suffers from the *out-of-thin-air* problem [4], where the language does not impose any ordering between a read and the writes that depend on its value. The intention is to universally permit aggressive compiler optimisation, but in doing so, this accidentally allows writes to take illogical values. The ARM and Power processors allow the relaxed outcome $r1 = 1$ and $r2 = 2$ in Figure 1, where there is nothing to enforce the order of the load and store in the second thread [30]. In Figure 2, each thread reads and then writes the value read—a data dependency from read to write. This dependency makes optimisation impossible, so the relaxed outcome $r1 = r2 = 1$ is forbidden on every combination of compiler and target processor. The specification of C++ erroneously allows the outcome $r1 = r2 = 1$, producing the value 1 "out of thin air."

Formally handling load buffering while avoiding out-of-thin-air behaviours turns out to be an enormously complex task. Although several works [6, 17, 20, 23, 28] have been dedicated to providing semantics for different variations of the load buffering example, many of these have also been shown to be inconsistent with expected behaviours under certain litmus tests [16, 28]. This work builds on top of the recent **Modular Relaxed Dependencies (MRD)** semantics by Paviotti et al. [28]. MRD avoids thin air and aims for compatibility with the existing ISO C and C++ standards.

A key component of MRD is the calculation of a *semantic dependency relation*, which describes when certain reorderings are disallowed. The program in Figure 1 contains only the semantic dependency between lines 1 and 2, whereas the program in Figure 2 contains semantic dependencies between lines 1 and 2 and lines 3 and 4. The program in Figure 1 may therefore execute line 4 before line 3, while the program in Figure 2 must execute both threads in order.

Operational Verification. Although precise, it is not possible to directly reason about programs under MRD, because MRD is a denotational semantics defined over an event structure [32]. This article addresses this gap by developing an operational semantics for MRD, which we then use as a basis for a deductive Owicki–Gries style verification framework. The key idea of our operational semantics is to take the semantic dependency relation as the only order in which programs must be executed, thereby allowing intra-thread reordering. This changes the fundamental meaning of sequential composition, allowing statements that occur "later" in the program to be executed early.

Our semantics is also designed to take non multi-copy atomicity into account, whereby writes are not propagated to all threads at the same time and hence may appear to take effect out-of-order [1, 2]. Note that this phenomenon is distinct from the reordering of operations within a thread (described above), and it is possible to separate the two.

Finally, we develop a logic capable of reasoning about program executions that exhibit both of the phenomena described above. Our logic makes use of the technique by Dalvandi et al. [9] of including assertions that enable reasoning about the "views" of each thread. Since we have concurrent programs, the logic we develop incorporates Owicki–Gries style reasoning for programs, in which assertions are shown to be both locally correct and globally stable (interference free). However, unlike earlier works [9, 21], since we relax thread order, the standard structural approach to Hoare-style proof decomposition is not possible. Instead, we model the system by a transition system whose execution is dictated by the semantic dependency relation.

Extended Contributions. The main motivation for this extended paper to the earlier conference version [33] has been the desire to mechanise our methodology, building on the extensive set of works already available for the RC11 model [9, 10]. In order to keep the MRD denotational model and the corresponding operational semantics as closely related as possible, our prior work [33] used the operational semantics by Doherty et al. [13] as a starting point, resulting in an operational semantics based on execution graphs (cf. References [2, 22]). However, proofs over such models require more effort to mechanise, since certain relations, e.g., the "happens-before" relation, are defined as a transitive reflexive closure of the union of other relations, which requires inductive reasoning and careful case analysis.

In this article, we therefore recast our existing work in terms of a timestamp-based model that already has extensive mechanisation support [9, 10]. We prove that this change in the underlying write-propagation semantics is still sound and complete with respect to MRD, i.e., every execution of the operational semantics is an execution of MRD and vice versa. Additionally, our new semantics makes use of a mapping from events within an event structure to control labels (called a *future*) within a program to further simplify the translation from the denotational to the operational semantics. This fixes the execution order of the underlying program in terms of control labels and allows the underlying event structure to be discarded during the verification. In our prior work, futures were defined in terms of the semantic dependency relation generated by MRD.

As mentioned above, our extension allows our proofs of correctness to be mechanised. Here, we make use of the mechanisation support within Isabelle/HOL developed as part of our earlier work on RC11 RAR [9, 10]. Since the write propagation semantics can be used without change, the only major difference is the way in which programs are executed. In particular, control flow is dictated by the futures generated from the denotational MRD model, as opposed to standard constructs such as branching and sequential composition. We use our framework to encode and verify a number of interesting examples within Isabelle/HOL (see Section 7).

Overview. This article is organised as follows. We recap MRD in Section 2 and the timestamp-based operational semantics for RC11 RAR in Section 3. Then in Section 4, we present a combined semantics, where program order in RC11 is replaced by a more relaxed order defined by MRD. In Section 5, we present the soundness and completeness of our operational semantics. A Hoare-like logic for reasoning about relaxed program execution together with Owicki–Gries-like rules for reasoning about interference is given in Section 6. We present our example proofs in Section 7 and describe the mechanisation in Section 8.

Auxiliary materials. We provide our Isabelle/HOL development as supplementary material to this article [34].

2 MRD AND SEMANTIC DEPENDENCIES

In this section, we review the MRD semantics for a simple C-like while language. This provides a mechanism for defining (in a denotational manner) a relaxed order in which statements within a thread are executed.

Events and actions. Weak memory literature uses a variety of terminology to refer to internal representations of changes to global memory, which complicates attempts to unify multiple models. In the following, we use *events* to refer to the objects created and manipulated by MRD, normally represented as integers. We use *actions* to refer to objects of the form i:R $x\ v$, i:W $x\ v$, referring to a read or write of value v at global location x arising from line i of the input program.

Program syntax. We assume *shared variables* $x, y, z, \ldots$ from a set X, *registers* $r_1, r_2, \ldots$ from a set *Reg*, and *register files* $\rho : Reg \to \text{Val}$, mapping registers to values from a set Val. *Expressions* $e, e_1, e_2, \ldots$ are taken from a set E, whose syntax we do not specify but which can be evaluated w.r.t. a register file using $eval \in E \times (Reg \to \text{Val}) \to \text{Val}$. Thus, $eval(e, \rho)$ is the value of e given the register values in ρ. For basic commands, we have variable assignments (or *stores*) $[y] := e$ and register assignments (or *loads*) $r := [x]$. In this article, we assume that stores and loads are *relaxed* [3, 13, 22] unless they are explicitly specified to be a releasing store (denoted $[y] :=^R e$) or an acquiring load (denoted $r :=^A [x]$). Finally, we assume a simple language of *programs*. The only unconventional aspect of this language is that we require each (variable or register) assignment be decorated with a unique *control label*. This allows a semantics that does not, in general, respect program order to refer to an individual statement without ambiguity. The syntax of commands (for a single thread) is defined by the following grammar, where B is an expression that evaluates to a Boolean and i is a control label. We let *CLabel* denote the set of all control labels. Note that since guards are expressions, they must not mention any shared variables—any guard that relies on a shared memory variable must load its values into a local register prior to evaluating the guard,

$$\begin{aligned} ACom &::= i\colon [x] :=^{[R]} e \mid i\colon r :=^{[A]} [x] \mid i\colon r := e \\ Com &::= ACom \mid Com; Com \mid \textbf{if } B \textbf{ then } Com \textbf{ else } Com \mid \textbf{while } B \textbf{ do } Com \end{aligned}$$

We use $[x] :=^{[R]} e$ to denote that the releasing annotation R is optional (similarly, $r :=^{[A]} [x]$ for the acquiring annotation A). For simplicity, we focus attention on the core atomic features of C11 necessary to probe the thin-air problem and omit more complex instructions such as CASs, fences, non-atomic accesses, and **sequentially consistent (SC)** accesses [5].

We assume a top level parallel composition operator. Thus, a program is of the form $C_1 \| C_2\| \ldots \| C_n$, where $C_i \in Com$. We further assume each atomic command *ACom* in the program has a unique label across all threads and that local registers are unique across threads, i.e., the same register is not used by more than one thread.

Denotational MRD. For the purposes of this article (i.e., the development of the operational semantics and associated program logic), the precise details of the MRD semantics [28] are unimportant. The most important aspect that we use is the set of *semantic dependency* relations that it generates, which precisely characterise the order in which atomic statements are executed.

In MRD, the *denotation* of a program P with an initialisation **Init** is returned by the semantic interpretation function $[\![\textbf{Init}; P]\!]_{n\ \rho\ \kappa}$, where n is a step index used to bound loop iterations, ρ is a register state, and κ is continuation used internally by the interpretation function (details in Reference [28]). This gives a *coherent event structure* of the form

$$(L, S, \vdash, \leq),$$

where

- $L = (E, \sqsubseteq, \#, \Lambda)$ is an *event structure* [32] equipped with a labelling function Λ from events (represented internally as integer identifiers) to actions. The set E contains all events in the structure, $\varepsilon_1 \sqsubseteq \varepsilon_2$ for $\varepsilon_1, \varepsilon_2 \in E$ iff ε_1 is program ordered before ε_2. Events in # cannot happen simultaneously, e.g., two reads of a variable returning different values are in conflict.

- S is a set of *partial executions*, each of which represents a possible interaction of the program with the memory system. Each execution in S is a tuple $(A, \textsc{lk}, \textsc{rf}, \textsc{dp})$, representing a set of events (A, where $A \subseteq E$), lock/unlock order (LK), a *reads-from relation* (RF), and *dependency order* (DP) between operations of *that* execution, and the set of events to be executed. An execution is *complete* if every read in A is linked to a write to the same location of the same value by an RF edge.
- $\vdash$ is a *justification relation* used in the construction of the program's dependency relation.
- $\leq$ is the *preserved program order* (cf. Reference [2]), a subset of program order that is respected by the memory model. For this semantics, we use a definition of preserved program order that preserves edges between accesses to the same location, edges that terminate at a releasing write, and edges that originate at an acquiring read.

MRD then restricts the set of partial executions to those that pass the constraints of an axiomatic concurrency model. MRD can augment any axiomatic concurrency model to support semantic dependencies, but here we work with MRD + RC11 [28], a variant of RC11 [22], where program order is replaced with preserved program order in its NO-THIN-AIR axiom: $\leq \cup \textsc{rf}$ is acyclic. The remaining RC11 axioms require that intra-thread reads-from edges do not contradict a derived relation called *happens before*, composed of program order and reads-from edges between accesses annotated as release and acquire. Furthermore, executions must be *complete*, meaning every read event is the destination of a reads-from edge.

To develop our operational semantics, we only require some of these components. From S, we only require the dependency order. From the labelled event structure, we use the labelling function Λ to connect actions to events. From the coherent event structure, we use $\leq$ to enforce ordering alongside dependency order.

Example 2.1. The event structure in Figure 3 represents the denotation of the program in Figure 1. Note first that each store of a value into a register generates multiple events—one for each possible value. This is because MRD cannot make assertions about which values it may or may not observe during interpretation of the structure, it can only be provided with global value range restrictions prior to running. The execution completeness requirement forbids reads that lack a corresponding write whose value they observe.

Events 2 and 4 are in *conflict* (drawn as a red zigzag) as they represent different potential values being read at line 1. Likewise, events 6, 8, and 10 represent different potential values at line 3. A single execution can observe events 2 and 6 but not events 2 and 4. Events 2 and 3 are in *program order* (represented by the black arrow), i.e., $2 \sqsubseteq 3$. MRD generates a *dependency order* between events 2 and 3 and events 4 and 5 (represented by the yellow arrow). The read at line 1 is stored in register `r1`, which is in turn accessed by the write in line 2, leading to a data dependency. There is no semantic dependency between any events arising from lines 3 and 4, because there is no way for the value read at line 3 to influence the value written at line 4. This means that lines 1 and 2 must be executed in order, but line 4 is free to execute before line 3. For example, consider the traces H_1 and H_2 below,

$$H_1 \;\widehat{=}\; \text{Init}\;\; 2{:}\mathrm{W}\,y\,1\;\; 4{:}\mathrm{W}\,x\,1\;\; 1{:}\mathrm{R}\,x\,1\;\; 3{:}\mathrm{R}\,y\,1$$
$$H_2 \;\widehat{=}\; \text{Init}\;\; 1{:}\mathrm{R}\,x\,1\;\; 4{:}\mathrm{W}\,x\,1\;\; 2{:}\mathrm{W}\,y\,1\;\; 3{:}\mathrm{R}\,y\,1.$$

For the program in Figure 1, the trace H_1 is *disallowed*, since in H_1, the operations at lines 2 and 1 are swapped in a manner inconsistent with the dependency order. The trace H_2 is *allowed*, since lines 3 and 4 are free to execute in any order.

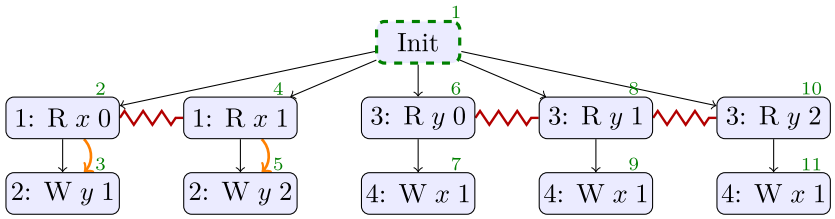

Fig. 3. Coherent event structure for the program in Figure 1.

Table 1. Components of a C11 State

Component	Informal meaning	Initial value
$writes \subseteq \mathsf{Wr} \times \mathbb{Q}$	The writes that have happened so far	$writes_{\mathbf{Init}}$
$tview_t \in \mathsf{Var}_G \to writes$	The viewfront of Thread t	$tview_{\mathbf{Init}}$
$mview_{\hat{w}} \in \mathsf{Var}_G \to writes$	The viewfront of a thread when writing $\hat{w}$	$mview_{\mathbf{Init}}$

3 OPERATIONAL SEMANTICS WITH RELAXED WRITE PROPAGATION

Next, we detail the timestamp-based memory semantics [9], which generates C11 states that describe how writes are propagated within the system. The semantics allows for both relaxed and release-acquire events and the writes themselves are not multi-copy atomic.

C11 State. Table 1 summarises the components of a C11 state. Each global write is represented by a pair $(a, q) \in \mathsf{Wr} \times \mathbb{Q}$, where a is a write action and q is a rational number that we use as a *timestamp* (cf. Reference [19]). The timestamps totally order the writes to each variable; the ordering induced by timestamps is also referred to as the *modification order* [13, 22] or *coherence order* [2]. For each write $\hat{w} = (a, q)$, we denote $\hat{w}$'s timestamp by $tst(\hat{w}) = q$. We also lift the functions *var* and *wrval* to timestamped writes, e.g., $var(\hat{w}) = var(a)$. The set of all writes that have occurred in the execution thus far is recorded in the state component $writes \subseteq \mathsf{Wr} \times \mathbb{Q}$.

Each state must record the writes that are observable to each read. To achieve this, we use two families of functions from global variables to writes, both of which record the *viewfronts* as follows:

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

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

- A function $mview_{\hat{w}}$ that records the *viewfront* of write $\hat{w}$, which is set to be the viewfront of the thread that executed $\hat{w}$ at the time of $\hat{w}$'s execution. We use $mview_{\hat{w}}$ to compute a new value for $tview_t$ if a Thread t *synchronizes* with $\hat{w}$, i.e., if $\hat{w} = (w, q)$, $w \in \mathsf{Wr}_\mathsf{R}$ and another thread executes an $a \in \mathsf{Rd}_\mathsf{A}$ that reads from w.

Initialisation. Table 1 also states how these components are initialised by **Init**. If $\mathsf{Var}_G = \{x_1, \ldots, x_n\}$ and $k_1, \ldots, k_n \in \mathsf{Val}$, then we assume $\mathbf{Init} = \bigwedge_i x_i = k_i$. The initial values of the state components are then defined as follows, where we assume that 0 is the initial timestamp.

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

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

Fig. 4. Transition relation of the memory semantics.

Note that without loss of generality, we assume local registers are initialised with value 0,

$$\begin{aligned} writes_{\mathbf{Init}} &\mathrel{\widehat{=}} \{(0{:}\mathrm{W}\ x_1\ k_1, 0), \ldots, (0{:}\mathrm{W}\ x_n\ k_n, 0)\} \\ tview_{\mathbf{Init}}(x_i) &\mathrel{\widehat{=}} (0{:}\mathrm{W}\ x_i\ k_i, 0) \quad \text{for each } x_i \in \mathsf{Var}_G \\ mview_{\mathbf{Init}} &\mathrel{\widehat{=}} tview_{\mathbf{Init}} \\ \rho_{\mathbf{Init}} &\mathrel{\widehat{=}} (\lambda x \in \mathsf{Var}_L.\ 0). \end{aligned}$$

Transition semantics. The transition relation of our semantics for global reads and writes is given in Figure 4. Each transition $\sigma \overset{a}{\rightsquigarrow}_t \sigma'$ is labelled by an action a and Thread t. The premise of each rule must identify the write action w that the action interacts with. This is made more precise below.

Read **transition by Thread** t**.** Here we assume that

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

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

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

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

For illustration, consider the picture in Figure 5. The x-axis depicts the timestamps of the writes, and the y-axis the variables x, y, and z, which we assume are initialised by writes x_0, y_0, and z_0, respectively. The orange line shows the view of a thread, say, t_1, and the blue line depicts the view of another thread that executes $\hat{w} = (2{:}\mathrm{W}^R\ y\ 42, 3)$. If Thread t_1 performs an acquiring read of y and reads from $\hat{w}$ (i.e., it performs a sychronising read), then Thread t_1's view changes to the diagram on the right, whereby its current viewfront is combined with the viewfront of $\hat{w}$.

Write **transition by Thread** t**.** A write transition must identify the write (w,q) after which a occurs. This w must be observable. We must choose a fresh timestamp $q' \in \mathbb{Q}$ for a, which is formalised by $fresh(q,q')$,

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

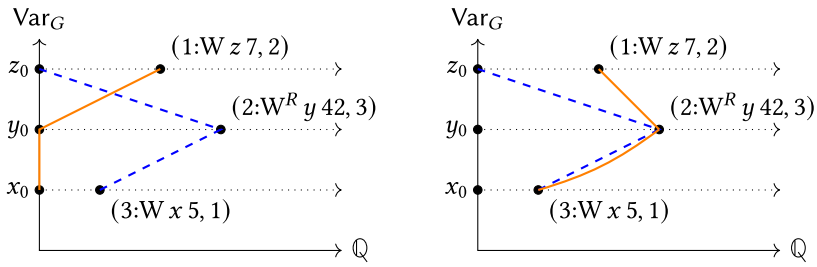

Fig. 5. Illustration of views and view updates: pre-state (left) and post-state (right) after executing $\mathrm{R}^A\, y\, 42$ by thread t_1 (orange) [9].

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

4 OPERATIONAL SEMANTICS OVER MRD

Recall that the MRD semantic interpretation function outputs a coherent event structure $\mu = (L, S, \vdash, \leq)$. In this section, we develop an operational semantics for traversing μ, while generating state configurations that model relaxed write propagation. In essence, this generalises the operational semantics in Section 3 so that threads are executed out-of-order, as allowed by MRD.

4.1 Program Futures

Defining our operational semantics over raw syntax would force us to evaluate statements in program order. Therefore, we convert this syntax into a set of atomic statements. We call this the *atomic set* of C, written $\overline{C}$. MRD evaluates while loops using step-indexing, treating them as finite unwindings of **if-then-else** commands. For these, we generate a fresh unique label for each iteration of the while loop for each of the atomic commands within the loop body. Recall that the parallel composition of commands is modelled by a function from thread identifiers to (sequential) commands. The atomic set of a program P is therefore $\overline{P} \mathrel{\widehat{=}} \lambda t.\, \overline{P(t)}$, which is generated by remapping each thread to the atomic set corresponding to the thread's command.

To retain the ordering recognised during program execution, we introduce *futures*. Essentially, instead of taking our operational steps in program order over the syntax, we can nondeterministically execute any statement that our futures tell us we have executed all necessary predecessors of. Unlike our prior work [33], which defined futures to be an ordered set of actions, we define a mapping from events in the event structure to a pair of type $(\mathit{CLabel}, \mathrm{Val} \cup \{\bot\})$, where $\bot \notin \mathrm{Val}$. A value (different from $\bot$) is used for events corresponding to reads, which may or may not be ordered w.r.t. later statements depending on the value returned. See Section 7 for examples.

Let $\mu = (L, S, \vdash, \leq)$ be a coherent event structure with labelling function Λ_μ, and $X \mathrel{\widehat{=}} (A, \mathrm{LK}, \mathrm{RF}, \mathrm{DP})$ be a partial execution of μ. We define

$$\varphi_X \mathrel{\widehat{=}} \left\{((l_1, v_1), (l_2, v_2)) \mid \exists (\varepsilon_1, \varepsilon_2) \in (\mathrm{DP} \cup \leq).\ \textstyle\bigwedge_{i \in \{1,2\}} \mathit{clab}(\Lambda(\varepsilon_i)) = l_i \wedge \mathit{rdval}(\Lambda(\varepsilon_i)) = v_i\right\}$$
$$\cup \left\{((l, v), (l, v)) \mid \exists e \in A.\ \mathit{clab}(\Lambda(\varepsilon)) = l \wedge \mathit{rdval}(\Lambda(\varepsilon)) = v\right\},$$

$$\frac{n = eval(e,\rho) \quad \rho' = \rho[r := n]}{(l\colon r := e, \rho) \xrightarrow{\tau}_t (\mathbf{skip}, \rho')} \qquad \frac{n = eval(e,\rho) \quad a = \mathrm{W}^{[R]}\, x\, n}{(l\colon [x] :=^{[R]} e, \rho) \xrightarrow{a}_t (\mathbf{skip}, \rho)}$$

$$\frac{a = \mathrm{R}^{[A]}\, x\, n \quad \rho' = \rho[r := n]}{(l\colon r :=^{[A]} [x], \rho) \xrightarrow{a}_t (\mathbf{skip}, \rho')}$$

Fig. 6. Uninterpreted operational semantics of reads and writes.

where for an action a, $clab(a)$ returns the control label of a and $rdval$ returns the value read by a. For simplicity, we use notation l^v to represent the label–value pair (l, v). If a does not perform any reads, then $rdval(a) = \bot$. Thus, the set of futures corresponding to μ is

$$F_\mu \mathrel{\widehat{=}} \{\varphi_T \mid T \in S\}.$$

The aim of this new definition of futures is to generate orders over program syntax so that the underlying event structure can be discarded when reasoning about correctness, which was not possible in our prior work [33]. Next, we define a future-based transition relation that utilises this new definition.

Let F be a set of futures and $\preceq \in F$ be a future. We say that a label–value pair l^v is *available* in $\preceq$ iff l^v is minimal in $\preceq$, i.e., for all k^u if $k^u \neq l^v$, then $k^u \not\preceq l^v$. If l^v is available in $\preceq$, then the *future of* l^v *in* $\preceq$, denoted $\preceq_{\setminus l^v}$, is the future with all label–value pairs corresponding to l^v removed.

We lift this to a set of futures F to describe the *candidate futures of* l^v *in* F, denoted $l^v \triangleright F$:

$$l^v \triangleright F = \{\preceq_{\setminus l^v} \mid \preceq \in F \wedge l^v \text{ is available in } F\}.$$

Note that l^v is *enabled* in F iff $l^v \triangleright F \neq \emptyset$.

Essentially, $l^v \triangleright F$ consumes an atomic statement from each of the futures in F provided that a is available, discarding all futures in which l^v is not available. Intuitively, if an event is minimal in a program future, then it may be executed immediately. If it is not minimal, then its predecessors must be executed first.

4.2 Future-based Transition Relation

First, we link the syntax of atomic statements (reads and writes) to the actions using the transition rules in Figure 6, where $f[k := v]$ denotes function updates in which $f(k)$ is updated to v. A local evaluation $i\colon r := e$ generates a silent τ action, which has no effect on the shared memory, i.e., is defined by the rule

$$\textsc{Silent}\ \frac{}{\sigma \overset{\tau}{\rightsquigarrow}_t \sigma}$$

Our operational semantics is defined by the transition relation in Future-Step given below. The transition relation is defined over $(\overline{Q}, (\sigma, \rho), F)$, where $\overline{Q}$ is the atomic set corresponding to the program text, (σ, ρ) is a configuration, and F is the current set of *futures*. The transition executes a statement $l : s$ from $\overline{Q}(t)$ by generating an action a and new register file according to $\xrightarrow{a}_t$ and shared memory state according to $\overset{a}{\rightsquigarrow}_t$. Note that we require $rdval(a) = v$ and that l^v is enabled in the set of futures F. Below, we use $\uplus$ to denote disjoint union

$$\textsc{Future-Step}\ \frac{\overline{Q}(t) = \overline{C} \uplus \{l\colon s\} \qquad l^v \triangleright F \neq \emptyset \qquad (l\colon s, \rho) \xrightarrow{a}_t (\mathbf{skip}, \rho') \qquad \sigma \overset{a}{\rightsquigarrow}_t \sigma' \qquad rdval(a) = v}{(\overline{Q}, (\sigma, \rho), F) \Longrightarrow (\overline{Q}[t := \overline{C}], (\sigma', \rho'), l^v \triangleright F)}$$

Once a minimal statement of some thread is chosen, it is checked for consistency and added to the set of executed events. The set of futures is pruned to remove the chosen statement and to exclude any futures that are incompatible with the chosen statement. The operational semantics continues until it has consumed all of the futures.

4.3 Example

Recall the program in Figure 1 and its event structure representation in Figure 3. Let $\Delta_A = \{(x, x) \mid x \in A\}$ be the diagonal of set A. We first derive our set of futures from this structure,

$$F \;\widehat{=}\; \{\{1^u \prec 2\} \cup \Delta_{\{3^v,4\}} \mid u \in \{0,1\} \wedge v \in \{0,1,2\}\},$$

where we drop the value corresponding to a control label when the value corresponding to the label is $\perp$, i.e., simply write l for $l^\perp$. Furthermore, we let $\{1^u \prec 2\}$ denote the future $\Delta_{\{1^u,2\}} \cup \{(1^u, 2)\}$. The atomic set of the program is

$$\overline{P} \;=\; \begin{Bmatrix} 1 \mapsto \{1 : \mathsf{r1} := [\mathsf{x}], 2 : [\mathsf{y}] := \mathsf{r1} + 1\}, \\ 2 \mapsto \{3 : \mathsf{r2} := [\mathsf{y}], 4 : [\mathsf{x}] := 1\} \end{Bmatrix}.$$

The initial configuration is (σ_0, ρ_0), where σ_0 is the initial state as described in Section 3 and $\rho_0 = \{1 \mapsto \{r1 \mapsto 0\}, 2 \mapsto \{r2 \mapsto 0\}\}$.

To find out which events we can execute, we check our futures set. We cannot yet execute line 2, since it requires that line 1 (with some value) is executed first. We can, however, execute any other line. Suppose we execute line 4, which corresponds to $4 : [\mathsf{x}] := 1$ in $\overline{P}(2)$. To use the transition relation $\Longrightarrow$, we need to do the following:

(1) determine the corresponding action a and new thread-local state using $\xrightarrow{a}_2$;
(2) generate a new memory state using $\overset{a}{\rightsquigarrow}$; and
(3) check that 4 is available in the current set of futures.

For (1), we can only create one action $a = 4{:}\mathrm{W}\,x\,1$ and the local state is unchanged. For (2), we generate a new global state τ using the WRITE rule in Figure 4. For (3), we take our candidate futures $4 \triangleright F$, which results in the new future set

$$F_4 \;=\; \{\{1^u \prec 2\} \cup \Delta_{\{3^v\}} \mid u \in \{0,1\} \wedge v \in \{0,1,2\}\}$$

and sub-program

$$\overline{P} \;=\; \begin{Bmatrix} 1 \mapsto \{1 : \mathsf{r1} := [\mathsf{x}], 2 : [\mathsf{y}] := \mathsf{r1} + 1\}, \\ 2 \mapsto \{3 : \mathsf{r2} := [\mathsf{y}]\} \end{Bmatrix}.$$

Now consider the execution of line 1. Step (1) generates a read event $a = 1{:}\mathrm{R}\,x\,v$. The read value v must be compatible with the set of futures as well as the memory semantics. In this case, the transitions corresponding to $v \in \{0,1\}$ are both possible, since 1^0 and 1^1 are both minimal in F_4. Moreover, $\tau.OW(1,x) = \{(0{:}\mathrm{W}\,x\,0, \_), (4{:}\mathrm{W}\,x\,1, \_)\}$ (ignoring the timestamps).

5 EQUIVALENCE BETWEEN MRD AND THE OPERATIONAL SEMANTICS

We now establish equivalence of our operational semantics and the MRD denotational semantics. To link the denotational model with the timestamp-based operational semantics, we must extend the state with additional auxiliary information that records the behaviours of reads (see Figure 7).

We first record some more information in our operational state, in particular which writes are observed by reads. We name this the *reads from* relation rf, and write that $w \xrightarrow{\mathsf{rf}} r$ if a write w stores the value observed by read r.

$$\textsc{Tracked Read}\ \frac{a \in \{\mathrm{l{:}R}\ x\ n, \mathrm{l{:}R}^{A}\ x\ n\} \qquad (w,q) \in \hat{\sigma}.OW(t,x) \qquad wrval(w) = n \qquad tview'_t = \begin{cases} \hat{\sigma}.tview_t \otimes \hat{\sigma}.mview_{(w,q)} & \text{if } (w,a) \in \mathsf{Wr_R} \times \mathsf{Rd_A} \\ \hat{\sigma}.tview_t[x := (w,q)] & \text{otherwise} \end{cases}}{\hat{\sigma} \overset{a}{\rightsquigarrow}_t \hat{\sigma}[tview_t := tview'_t,\ \mathsf{rf} := \mathsf{rf} \cup \{(w,a)\},\ reads := reads \cup \{a\}]}$$

Fig. 7. Transition relation with rf tracking.

We also need to track which read events of MRD have been executed, thus we introduce a second state component, *reads*, that is updated by the execution of each read action. We let

$$\sigma.actions \ \widehat{=}\ \mathbf{dom}(\sigma.writes) \cup \sigma.reads$$

be the set of all actions in the state σ. Note that actions are guaranteed to be unique, since we include the control label corresponding to the action. Further note that elements of $\sigma.writes$ are pairs of the form (a,q), where a is a write action and q is a timestamp.

To disambiguate the standard semantics from the tracked-read semantics, we use $\hat{\sigma}$ to refer to states of the tracked read semantics. Additionally, given a state $\hat{\sigma}$, one can derive the *modification order*, mo, by ordering all elements in the *writes* set by timestamp, meaning whenever we have (w_1,q_1) and (w_2,q_2) in *writes* such that $q_1 < q_2$, we have $(w_1,w_2) \in \mathsf{mo}$.

Theorem 5.1 (Soundness and Completeness). *Every execution generated by the* MRD *model can be generated by the operational semantics, and every final state generated by the operational semantics corresponds to a complete execution of the* MRD *semantics.*

We define this correspondence between MRD executions and states of the operational semantics, $\sim$, as follows: $(A, \textsc{rf}, \textsc{dp}, \textsc{rf}) \sim \hat{\sigma}$ iff $\Lambda(A) = \hat{\sigma}.actions \wedge \Lambda(\textsc{rf}) = \hat{\sigma}.\mathsf{rf}$.

The operational semantics of Section 4 is sound and complete with respect to MRD. We establish these properties here in Lemmas 5.2 and 5.3, respectively. We use Lemma 5.4 in our proof of completeness—it allows us to reorder steps of the operational model in cases where the memory behaviour does not constrain the ordering.

5.1 Operational Executions are Event-structure Executions

Here we show that every state reachable in the operational semantics corresponds to an execution of the MRD model. The existence of an execution with a corresponding set of events follows from the conversion of executions into program futures (via the φ_X function) and the requirement that we follow a program future whenever we take an operational step. We then show that every rf relation built by the operational semantics is covered by an $\textsc{rf}$ relation in a corresponding execution in MRD.

Lemma 5.2. *Given some program P such that*

$$(\overline{P}, (\hat{\sigma}_{\mathbf{Init}}, \rho_{\mathbf{Init}}), F) \Longrightarrow^{*} ((\lambda t : TID.\ \emptyset), (\hat{\sigma}, \rho), \{\emptyset\})$$

over the tracked read semantics. There exists an execution $(A, \textsc{lk}, \textsc{rf}, \textsc{dp})$ in $[\![\mathbf{Init}; P]\!]_{n\ \rho_{\mathbf{Init}}\ \emptyset}$ such that $(A, \textsc{lk}, \textsc{rf}, \textsc{dp}) \sim \hat{\sigma}$.

Proof. Let $D = \hat{\sigma}.actions$, $\mathsf{rf}_D = \hat{\sigma}.\mathsf{rf}$ and $f \in F$ be the future that we followed in the operational execution. By definition of F, we must have that $f = \varphi_X$ for some execution $X = (E, \textsc{lk}, \textsc{rf}, \textsc{dp})$ and $\Lambda(E) = D$.

We now show by contradiction that there must be some execution that has an $\textsc{rf}$ relation such that $\Lambda(\textsc{rf}) = \mathsf{rf}_D$. To arrive at a complete rf_D relation that does not correspond to any $\textsc{rf}$, there must either be an edge in rf_D, which is absent from all $\textsc{rf}$ or an edge in $\textsc{rf}$ that is absent from rf_D.

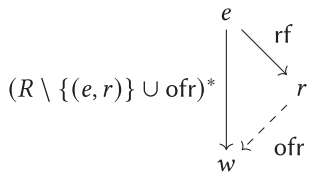

Fig. 8. Construction of an $\xrightarrow{\text{ofr}}$ edge.

We know that MRD exhaustively generates RF edges that do not create a cycle in RF ∪ DP ∪ ≤. There cannot be an edge in RF that is simply absent from rf_D with no new edge added, as this would imply that rf_D is incomplete, i.e., there exists a read in $D \notin \mathbf{ran}(\text{rf}_D)$. This is forbidden by the read rule: Every read event in D must have a corresponding edge in rf_D.

If an edge is absent from all RF, then it must always cause a RF ∪ DP ∪ ≤ cycle. Given that futures are ordered by DP ∪ ≤ and the definition of availability within a future only allows us to execute events in this order, we can only generate rf_D edges from earlier writes to later reads. To create a cycle, however, we would need to relate a later write to an earlier read. This is forbidden by the availability requirement, thus there must be some rf that is equal to rf_D. □

5.2 The Operational Semantics Covers All Event-structure Executions

Here we show that, given an execution of the MRD model, we can find an operational state after stepping through the program that contains the same events with the same reads-from edges. We do this by establishing a total order over the events in the execution and proving inductively that we are able to take operational steps in this order and that every such step results in an operational state corresponding to a restriction of the MRD execution.

In the following, we define the restriction of X to event set B, given $X = (A, \text{LK}, \text{DP}, \text{RF})$ and $B \subseteq A$, as $X_{|B} = (B, \text{LK}_{|B}, \text{DP}_{|B}, \text{RF}_{|B})$.

Lemma 5.3. *Let $X = (A, LK, RF, DP)$ be a complete execution of $[\![\mathbf{Init}; P]\!]_{n\ \rho_{\text{Init}}\ \emptyset} = (L, S, \vdash, \leq)$ and $\xrightarrow{R}$ be the relation $DP \cup RF \cup \leq$. Then both of the following hold:*

(1) *There exists a relation $\xrightarrow{R'}$ disjoint from $\xrightarrow{R}$ such that $\xrightarrow{R \cup R'}$ is total and acyclic over E, giving the chain $e_1 \xrightarrow{R \cup R'} e_2 \xrightarrow{R \cup R'} \cdots \xrightarrow{R \cup R'} e_k$.*

(2) *There exists an execution of the operational semantics $\hat{\sigma}_{\mathbf{Init}} \overset{\Lambda(e_1)}{\rightsquigarrow} \hat{\sigma}_1 \overset{\Lambda(e_2)}{\rightsquigarrow} \cdots \overset{\Lambda(e_k)}{\rightsquigarrow} \hat{\sigma}_k$ over P such that $X_k \sim \hat{\sigma}_k$, where $X_k = X_{\{e_1, e_2, \ldots e_k\}}$.*

Proof. To begin, we capture ordering implicit in R from reads to writes at the same location in the *operational from-reads* relation $\xrightarrow{\text{ofr}}$:

$$\text{ofr} \mathrel{\widehat{=}} \left\{ (r : \mathrm{R}\, x\, v_1, w : \mathrm{W}\, x\, v_2) \mid \exists e.\ e \xrightarrow{\text{rf}} r \wedge e \xrightarrow{R \setminus \{(e,r)\} \cup \text{ofr}}{}^{*}\ w \right\}.$$

This means that if $w_1 \xrightarrow{R \cup \text{ofr}}{}^{*} w_2$ and $w_1 \xrightarrow{\text{rf}} r$, then $r \xrightarrow{\text{ofr}} w_2$, illustrated in Figure 8. Intuitively, a read will always be ofr-before any write to the same location that must occur after the write from which it read.

We establish that $(R \cup \text{ofr})^*$ must be acyclic. Define a partially constructed ofr as follows:

$$\text{ofr}_n \mathrel{\widehat{=}} \left\{ (r : \mathrm{R}\, x\, v_1, w : \mathrm{W}\, x\, v_2) \mid \exists e.\ e \xrightarrow{\text{rf}} r \wedge e \xrightarrow{R \setminus \{(e,r)\} \cup \text{ofr}_{n-1}}{}^{*}\ w \right\}$$

$$\text{ofr}_0 \mathrel{\widehat{=}} \left\{ (r : \mathrm{R}\, x\, v_1, w : \mathrm{W}\, x\, v_2) \mid \exists e.\ e \xrightarrow{\text{rf}} r \wedge e \xrightarrow{R \setminus \{(e,r)\}}{}^{*}\ w \right\}.$$

At ofr_0, given R^* is promised by MRD's acyclicity check to be acyclic, there cannot be an edge from w to e or r to e. This means the only way to cause a cycle between these three events is to create an edge from w to r, which is the inverse of the edge we actually create. The acyclicity of $(R \cup \text{ofr}_n)^*$ follows from the same reasoning using the acyclicity of $(R \cup \text{ofr}_{n-1})^*$.

We proceed by induction over $\xrightarrow{R \cup \text{ofr} \cup R'}$.

Base case. We take any arbitrary $\xrightarrow{R'}$ that makes $\xrightarrow{R \cup \text{ofr} \cup R'}{}^*$ total and retains acyclicity, and take the set of mo-minimal writes to be E. Each of these initialising writes will set a unique global variable to initial value as defined by **Init**. Let $e_1..e_m$ be these writes in the arbitrary order given by R', as a set of writes to disjoint locations will be unordered by $R \cup \text{ofr}$: The DP and ofr edges relate reads to writes, $\leq$ relates same-location accesses, and rf relates writes to reads.

Our initial state $\hat{\sigma}_{\text{Init}}$ is the tuple $(writes_{\text{Init}}, tview_{\text{Init}}, mview_{\text{Init}}, \emptyset, \emptyset)$. We let $\Lambda(\{e_1..e_m\}) = writes_{\text{Init}}$, as both of these sets contain one event or action, respectively, $\mathrm{W}\,x\,0$ for each $x \in \text{Var}_G$, and have $\Lambda(\emptyset) = \emptyset$ for the respective rf relations. Therefore, $X_m \sim \hat{\sigma}_{\text{Init}}$.

It remains to show that $X_{n-1} \sim \hat{\sigma}_{n-1} \Rightarrow X_n \sim \hat{\sigma}_n$ for $n \leq k$. There are two cases for the event label for e_n: a read event or a write event.

Inductive read case. In the case where e_n is a read, we must use the semantic rule for appending a new read to state $\hat{\sigma}_{n-1}$. This requires an event w to exist such that $\Lambda(w)$ is in D, that $\Lambda \in \sigma_{n-1}.OW(tid(\Lambda(e_n)), var(\Lambda(e_n)))$, and that $var(w) = var(e_n) \wedge wrval(w) = wrval(e_n)$. If we can find such an event, then $(\Lambda(w), \Lambda(e_n))$ will be in rf_D. This means we also require $(w, e_n) \in \text{RF}_E$.

Given X is complete, there must be some write w' such that $w' \xrightarrow{\text{rf}} e_n$. It must also be true that $w' \xrightarrow{R \cup \text{ofr} \cup R'}{}^* e_n$, due to the inclusion of rf in $\xrightarrow{R}$. Therefore, w' will be in $e_1..e_{n-1}$ and we can use w' as our target w.

It remains to show that w will be in the observable writes set. We know that $\Lambda(w)$ will be in $\sigma_{n-1}.OW(tid(\Lambda(e_n)), var(\Lambda(e_n)))$ if there are no events in the intersection of D and $\sigma_{n-1}.tview_{tid(e_n)}(x)$ with a greater timestamp value. If $\Lambda(w)$ is maximal in mo, then there will be no events in D with a greater timestamp value.

Suppose $\Lambda(w)$ is not maximal and some event w'' exists in X_{n-1} such that $(\Lambda(w), \Lambda(w'')) \in \text{mo}$. By the construction of mo, it must also be true that $w \xrightarrow{R \cup \text{ofr} \cup R'}{}^* w''$.

If there is at least one $\xrightarrow{R'}$ edge connecting them, then by Lemma 5.4 there exists another choice of $\xrightarrow{R'}$ such that w'' is added to $\hat{\sigma}_n$ before w and (w, w'') is no longer in mo.

If there are no $\xrightarrow{R'}$ edges between them, then $w \xrightarrow{R \cup \text{ofr}} w''$. If $w \xrightarrow{R \cup \text{ofr}}{}^* w''$ and $w \xrightarrow{\text{rf}} e_n$, then $e_n \xrightarrow{\text{ofr}} w''$. This means that w'' cannot be in $e_1..e_n$, and therefore $\Lambda(w'')$ cannot be in D. By this and the above, whenever w is not maximal in mo we must be able to find a new R' such that it becomes maximal. This gives us that $\Lambda(w)$ is in $\sigma_{n-1}.OW(tid(\Lambda(e_n)), var(\Lambda(e_n)))$.

Inductive write case. If e_n is a non-initialising write, meaning there is some event $w \in X$ that is a write to the same variable as e_n, then we must show that w is in $\sigma_{n-1}.OW(tid(\Lambda(e_n)), var(\Lambda(e_n)))$. This follows by the same reasoning as in the read case. □

Lemma 5.4. *If $e_i \xrightarrow{R \cup R'}{}^* e_j \wedge e_i \not\xrightarrow{R}{}^* e_j$ in the chain $e_1..e_n$, then there exists a valid choice of relation $\xrightarrow{R''}$ such that $e_j \xrightarrow{R \cup R''}{}^* e_i$, the union $\xrightarrow{R \cup R''}$ is acyclic, and there exists an event e'_n in $e_1..e_n$ such that*

(1) $e_1..e_n$ *under* $\xrightarrow{R \cup R'}$ *is equal to* $e_1..e'_n$ *under* $\xrightarrow{R \cup R''}$

(2) σ'_n, *the result of running the operational semantics under* $\xrightarrow{R \cup R''}$, *is equal to* σ_n *excepting relation edges between* e_i *and* e_j.

Proof. Begin by enforcing that $e_j \xrightarrow{R''} e_i$. A cycle in $\xrightarrow{R \cup R''}$ would have the shape $e_i \xrightarrow{R \cup R''} e_j$. As we can trivially prevent $e_i \xrightarrow{R''}{}^{*} e_j$ by construction, this relation edge must contain one or more event pairs e and e' such that $e \xrightarrow{R}{}^{*} e'$. This event pair can be moved $\xrightarrow{R''}$ before e_j unless $e_j \xrightarrow{R}{}^{*} e$, and it can be moved $\xrightarrow{R''}$ after e_i unless $e' \xrightarrow{R}{}^{*} e_i$. If both of these are true, then $e_i \xrightarrow{R}{}^{*} e_j$, which we know is untrue.

To find e'_n, choose any element that is not equal to e_j from the set of events that are maximal in $\xrightarrow{R}$ and make it maximal in $\xrightarrow{R''}$. There must be some element in this set that is not e_j, as otherwise we would have $\forall e.e \xrightarrow{R} e_j$ and it would again be true that $e_i \xrightarrow{R}{}^{*} e_j$.

What remains is to create two arbitrary orderings R''_1 and R''_2 such that $e_1 \xrightarrow{R \cup R''_1} e_j$ and $e_j \xrightarrow{R \cup R''_2} e_n$ and both $\xrightarrow{R \cup R''_1}$ and $\xrightarrow{R \cup R''_2}$ are acyclic. Partition all remaining events into those that are and are not $\xrightarrow{R}$ before e_j, then construct $\xrightarrow{R''_1}$ over the first and $\xrightarrow{R''_2}$ over the second. These are individually trivial by the acyclicity of $\xrightarrow{R}$, and their union must be acyclic as their event sets are disjoint.

It is trivial to establish that $e_1..e_n = e_1..e_{n'}$ by the fact that all relation edges in X are preserved in $\xrightarrow{R}$. To show that σ_n is equivalent to $\sigma_{n'}$ excepting (e_i, e_j) edges, it suffices to show that if $(e, e') \in \text{mo}$, then $(e, e') \in \text{mo}'$, as rf edges are constrained by $\xrightarrow{R}$. Events are only relocated around points e_i and e_j, so any such edges would have the shape (e, e_j) or (e_i, e) in σ_n and become (e_j, e) or (e, e_i), respectively. Focusing on the first case, any such reordering is only *required* if $e_j \xrightarrow{R}{}^{*} e$, as this forces e to be placed $\xrightarrow{R \cup R''}{}^{*}$ after e_j in our construction of $\xrightarrow{R''}$. This makes the initial mo edge impossible to observe. The same reasoning holds for the (e_i, e) edge. □

6 HOARE LOGIC AND OWICKI–GRIES REASONING

Recall that the standard Owicki–Gries methodology [27] decomposes proofs of parallel programs into two cases:

- *local correctness* conditions, which define correctness of an assertion with respect to an individual thread, and
- *non-interference* conditions, which ensures stability of an assertion under the execution of statements in other threads.

The standard Owicki–Gries methodology has been shown to be applicable to a weak memory setting with relaxed write propagation (but without relaxed program order) [9]. Note that an alternative characterisation (also in a model without relaxed program order) has been given in Reference [21].[1] Unfortunately, the unrestricted C11 semantics captured by MRD relaxes program order and hence sequential composition to allow behaviours such as those in Figure 1, which requires a fundamental shift in Hoare-style proof decomposition. We show that the modular Owicki–Gries rules for reasoning about concurrent threads remain unchanged.

[1]This characterisation uses standard assertions but assumes a non-standard interpretation of Hoare-triples and introduces a stronger interference freedom check. In fact, for the model in Reference [21], the introduction of auxiliary variables is unsound.

Like Dalvandi et al. [9], we assume assertions are predicates over state configurations. In the current article, the operational semantics is dictated by the set of futures generated by MRD, and thus we require two modifications to the classical meaning. First, like prior work [9], we assume predicates are over state configurations, which include (local) register files and (shared) event graphs. This takes into account relaxed write propagation. Second, we introduce Hoare-triples with futures generated by MRD, which takes into account relaxed program execution.

In the development below, we refer to the *preprocessed form* of a program P given by $\mathcal{P}(P) = (\mu, \overline{P})$, where μ is the coherent event structure corresponding to P and $\overline{P}$ the set normal form of P. We let F_μ be the initial set of futures corresponding to μ. P terminates when the set of futures only contains the empty set.

Definition 6.1 (Hoare Triple). Suppose X and Y are predicates over state configurations, P is a concurrent program, and $\mathcal{P}(P) = (\mu, \overline{P})$. The semantics of a Hoare triple is given by $\{X\}\ \mathbf{Init}; P\ \{Y\}$, where

$$\{X\}\ \mathbf{Init}; P\ \{Y\} \ \widehat{=}\ X(\sigma_{\mathbf{Init}}, \rho_{\mathbf{Init}}) \\ \wedge(\forall \mathbb{C}, \mathbb{C}'.\ X(\mathbb{C}) \wedge ((\overline{P}, \mathbb{C}, F_\mu) \Longrightarrow^* (\emptyset, \mathbb{C}', \{\emptyset\}))\ \Rightarrow\ Y(\mathbb{C}')).$$

That is, $\{X\}\ P\ \{Y\}$ holds iff for every pair of state configurations $\mathbb{C}, \mathbb{C}'$, assuming $X(\mathbb{C})$ holds and we execute $\overline{P}$ with respect to the future F_μ until $\overline{P}$ terminates in $\mathbb{C}'$, we have $Y(\mathbb{C}')$.

Although Definition 6.1 provides meaning for a Hoare triple, we still require a method for decomposing the proof outline. To this end, we introduce the concept of a *future predicate*, which is a predicate parameterised by both futures and configuration states. To make use of future predicates, we introduce a notion of a Hoare triple for programs in set normal form.

For the definitions below, assume P is a program and $\mathcal{P}(P) = (\mu, \overline{P})$. We say an atomic set $\overline{Q}$ is a *sub-program* of an atomic set $\overline{P}$ iff for all t, $\overline{Q}(t) \subseteq \overline{P}(t)$. We say a set of futures F' is a *sub-future* of set of futures F iff for each $f' \in F'$ there exists an $f \in F$ such that f' is an up-closed subset of f. For example, if $F = \{\{1 \prec 2, 3, 4\}, \{1 \prec 2, 3\}\}$, then $\{\{2, 4\}, \{1 \prec 2\}\}$ is a sub-future of F, but $\{\{1, 3\}\}$ is not. We say the sub-program $\overline{Q}$ corresponds to a sub-future F iff $\mathit{labels}(\overline{Q}) = \mathit{labels}(F)$, where we assume *labels* returns the set of all control labels of its argument.

Definition 6.2 (Hoare Triple (Single Step)). Suppose I and I' are future predicates, P is a program and $\mathcal{P}(P) = (\mu, \overline{P})$. If G is a sub-future of F_μ, corresponding to a sub-program $\overline{Q}$ of $\overline{P}$, then we define

$$\{I\}_G\ \overline{Q}\ \{I'\} \ \widehat{=}\ \forall \mathbb{C}, \mathbb{C}', G'.\ I(G)(\mathbb{C}) \wedge ((\overline{Q}, \mathbb{C}, G) \Longrightarrow (\overline{Q}', \mathbb{C}', G'))\ \Rightarrow\ I'(G')(\mathbb{C}').$$

We say I is *future stable* for $(F, \overline{P})$ iff for all sub-futures G of F with corresponding sub-programs $\overline{Q}$ of $\overline{P}$, we have $\{I\}_G\ \overline{Q}\ \{I\}$.

Lemma 6.3 (Invariant). *Suppose X and Y are configuration-state predicates, P is a program and $\mathcal{P}(P) = (\mu, \overline{P})$. If $X \Rightarrow I(F_\mu)$, $I(\{\emptyset\}) \Rightarrow Y$, and I is future stable for $(F_\mu, \overline{P})$, then $\{X\}\mathbf{Init}; P\{Y\}$ provided $X(\sigma_{\mathbf{Init}}, \rho_{\mathbf{Init}})$.*

By construction, the sets of futures corresponding to different threads are disjoint, i.e., for each future $\preceq \in F_\mu$, we have that $\preceq = \bigcup_t \preceq_{|t}$, where $\preceq_{|t}$ denotes the future $\preceq$ restricted to control labels of thread t. The only possible inter-thread dependency in MRD is via the reads from relation [28], which does not contribute to the set of futures. This observation leads to a technique for an Owicki–Gries-like modular proof technique for decomposing the monolithic invariant I into an invariant per thread.

We define $F_{|t} = \{\preceq_{|t} \mid \preceq \in F\}$. Then, we obtain the following lemma for decomposing invariants in the same way as Owicki and Gries.

Lemma 6.4 (Owicki–Gries). *For each thread t, let I_t be a future predicate corresponding to t. If $X(\sigma_{\text{Init}}, \rho_{\text{Init}})$, $X \Rightarrow \forall t.\ I_t(F_{\mu\,|t})$, and $\forall t.\ I_t(\{\emptyset\}) \Rightarrow Y$, then $\{X\}P\{Y\}$ holds provided both of the following hold.*

(1) *For all threads t, I_t is future stable for $(F_{\mu\,|t}, \overline{P})$.* *(local correctness)*
(2) *For all threads t_1, t_2 such that $t_1 \neq t_2$, sub-futures F_1 of $F_{\mu\,|t_1}$, and F_2 of $F_{\mu\,|t_2}$, if $\overline{Q}$ corresponds to F_2, then we have $\{I_{t_1}(F_1) \wedge I_{t_2}\}_{F_2}\ \overline{Q}\ \{I_{t_1}(F_1)\}$.* *(global correctness)*

Thus, we establish $\{X\}P\{Y\}$ through a series of smaller proof obligations. We require that

(1) the initialisation of the program guarantees X,
(2) whenever X holds then for each thread t, I_t holds for the initial future $F_{\mu\,|t}$,
(3) if $I_t(\{\emptyset\})$ holds for all t, then the post-condition Y holds,
(4) each I_t is maintained by the execution of each thread, and
(5) I_t at each sub-future of t is stable with respect to steps of another thread.

7 VERIFICATION EXAMPLES

With the verification framework now in place, we present a correctness proof for the program in Figure 1, as well as three additional other programs. First, in Section 7.1, we present assertions for reasoning about state-configurations.

7.1 View-based Assertions

As we can see from Figure 6, the states that we use are *configurations*, which are pairs of the form (σ, ρ), where σ is a tagged action graph representing the shared state and ρ is a mapping from threads to register files representing the local state. Like prior work [9], we use assertions that describe the *views* of each thread, recalling that due to relaxed write propagation, the views of each thread may be different. Note that the formalisation of a state in prior work is done using a timestamp-based semantics [9]. Nevertheless, the principles for defining thread-view assertions also carry over to our setting of tagged action graphs.

In this article, the programs we consider are relatively simple, and, hence, we only use two types of view assertions: *synchronised view*, denoted $[x = v]_t$, which holds iff both Thread t observes the last write to x and this write updates the value of x to v, and *possible view*, denoted $[x \approx v]_t$, which holds iff t may observe a write to x with value v. Formally, we define the following. If W is a set of (timestamped) writes, x is a variable and $view : \text{Var}_G \to (\text{Wr} \times \mathbb{Q})$ is a view function, then we define the following:

$$last(W, x, w) \ \widehat{=}\ \hat{w} \in W \wedge (\forall \hat{w}' \in W.\ tst(\hat{w}') \leq tst(\hat{w}))$$
$$dview(view, W, x, n) \ \widehat{=}\ last(W, x, view(x)) \wedge wrval(view(x)) = n.$$

Thus $last(W, x, w)$ iff w is the write to x in W with the maximal timestamp and $dview(view, W, x, n)$ iff $view(x)$ is the last write to x in W and the value of this write is n.

We then define the following assertions for $x, y \in \text{Var}_G$ and $u, v \in \text{Val}$ (see References [9, 10] for details):

$$[x = v]_t(\sigma, \rho) \ \widehat{=}\ dview(\sigma.tview_t, \sigma.writes, x, v)$$
$$[x \approx v]_t(\sigma, \rho) \ \widehat{=}\ \exists \hat{w} \in \sigma.OW(t, x).\ wrval(\hat{w}) = v$$
$$(x = u)[y = v]_t(\sigma, \rho) \ \widehat{=}\ \forall (w, q) \in \sigma.OW(t, x).\ wrval(w) = u \Rightarrow w \in \text{Wr}_{\text{R}} \wedge dview(\sigma.mview_{(w,q)}, \sigma.writes, y, v).$$

Thus the definite observation assertion $[x = v]_t(\sigma, \rho)$ holds iff Thread t currently sees the last write to x that produces the value v. The possible observation assertion $[x \approx v]_t(\sigma, \rho)$ holds iff t can see

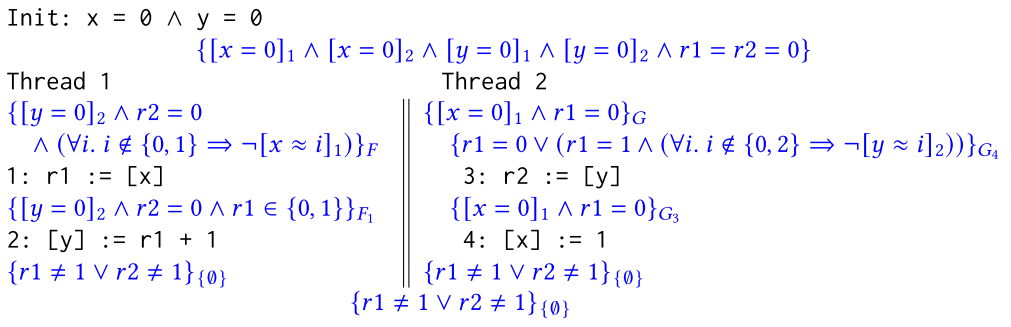

Fig. 9. Proof outline for load buffering with semantic dependencies.

some write to x that writes v. Finally the conditional observation assertion $(x = u)[y = v]_t(\sigma, \rho)$ holds iff for every write, w, to x that t can observe with value u, we have that w is a releasing write and the modification view of w over $\sigma.writes$ sees the last write to y with value v.

If $(x = u)[y = v]_t(\sigma, \rho)$ holds, then we can guarantee that if t reads x with value u via an acquiring read, and then it subsequently sees the last write to y with value v. Further details of these assertions may be found in References [9, 10].

Next we discuss a series of examples that make use of these assertions.

7.2 LB+sdep

The interleaved program text and future-indexed assertions for this example are given in Figure 9.

To reduce the domain of our future predicate, we collapse the event-based futures used by the operational semantics into sets of label-based futures. An event future F_E can be converted into a label future F_L by applying the labelling function to all events in F_E. This makes the futures $\{3\}$ and $\{5\}$ equivalent, as both are instances of $\{2{:}\mathrm{W}\ y\ 2\}$, and thus $I(\{3\}) = I(\{5\})$. This is not always a valid step, as some events that share labels may not be related by $\preceq$ in the same way. Thus, the technique can only be used if the label-based representation describes exactly the futures generated by MRD, which is the case for our example.

We describe our initial set of label futures as $\{\{1^u \prec 2\} \cup \Delta_{\{3^v,4\}} \mid u \in \{0,1\} \wedge v \in \{0,1,2\}\}$. We verify that this is a valid step: All futures generated by MRD are described by these labels, and they do not describe any potential futures not generated by MRD. We partition this into $F = \{\{1^u \prec 2\} \mid u \in \{0,1\}\}$ and $G = \{\Delta_{\{3^v,4\}} \mid v \in \{0,1,2\}\}$ representing the future sets of Thread 1 and Thread 2, respectively. We let $F_1 = \{\Delta_{\{2\}}\}$ be the future set after executing line 1, $G_3 = \{\Delta_{\{4\}}\}$ be the future set after executing line 3 (reading some value for y) and $G_4 = \{\Delta_{\{3^v\}} \mid v \in \{0,1,2\}\}$ be the future set after executing line 4.

Our future predicate I must output a configuration predicate for every sub-future of the initial set. Our partitioning fully describes these sub-futures, so we now attach our assertions to these futures.

To simplify the visualisation, we interleave the future predicate components with the program to provide Hoare-style pre/post-assertions, and place the assertion above a line of code if that line of code is contained in the applied future. We use indentation to denote that an assertion applies to more than one future. In this example G contains both lines 3 and 4; hence both lines are indented w.r.t. the first assertion in Thread 2. We apply Lemma 6.4, and in the discussion below, we describe the local and global correctness checks.

For local correctness in Thread 1, we must establish that $\{I\}_F\{1,2\}\{I\}_{\{\emptyset\}}$. By the definition of the future F, line 1 must be executed before line 2. This means we only need to verify that $\{I\}_F\{1\}\{I\}_{F_1}$ and $\{I\}_{F_1}\{2\}\{I\}_{\{\emptyset\}}$, which is identical to a standard Hoare logic proof and relatively uninteresting.

In Thread 2, no order is imposed between lines 3 and 4. This means to establish local correctness we must check that

- $\{I\}_G$ `3: r2 := y` $\{I\}_{G_3}$
- $\{I\}_G$ `4: x := 1` $\{I\}_{G_4}$
- $\{I\}_{G_3}$ `4: x := 1` $\{I\}_{\{\emptyset\}}$
- $\{I\}_{G_4}$ `3: r2 := y` $\{I\}_{\{\emptyset\}}$

The first three are trivial: line 3 modifies neither x nor $r1$, and line 4 does not modify $r1$. For the final step, the first disjunct of $\{I\}_{G_4}$ is the same as the first disjunct of $\{I\}_{\{\emptyset\}}$, and the second disjunct ensures that we cannot observe $r2 = 1$ after executing line 3.

For global correctness, we must check that every assertion in Thread 1 continues to hold after every line of Thread 2, and vice versa. These checks are also straightforward, so we omit a detailed discussion. The only noteworthy aspect is that for line 3 (and similarly, line 4), our precondition is the conjunction $\{I\}_{G_4} \wedge \{I\}_G$, as both G and G_3 are subfutures corresponding to line 3.

7.3 Partial Conditional

Our next example is the partial conditional example, whose proof outline is given in Figure 10 and corresponding event structure generated by MRD in Figure 11. We name the initial future sets for this program F for Thread 1 and G for Thread 2. For the sub-futures used in each assertion, we annotate the events that have been executed to reach that sub-future via subscript. For instance,

$$\{r1 = 1 \wedge r2 = 1 \wedge I\}_{G_{3^1,4^1}},$$

corresponds to the assertion that holds after executing lines 3 and 4, where both reads return the value 1. Similarly,

$$\{r1 = 1 \wedge r2 = 1 \wedge [w = 0]_2 \wedge [z = 1]_2\}_{G_{3^1,4^1,6}}$$

represents the assertion that holds after executing lines 3 and 4 (both returning the value 1) as well the write at line 6.

The program contains no dependencies in Thread 1 and non-trivial dependencies in Thread 2. The semantic dependencies we gain from program analysis in that thread are smaller than the dependencies we would gain from syntactic analysis, because there is some redundancy in the guards that a compiler would be willing to optimise away.

- The writes on lines 5 and 8 are identical, and their guards only differ in the value stored in register $r2$. Since between them they cover every possible value we could witness in $r2$, we can say that the action [w] := 1 at both line 5 and line 8 is always dependent on line 3 and never dependent on line 4.
- Likewise, the writes on lines 6 and 7 are identical, and since their guards similarly cover every value in $r1$ but require a specific one in $r2$ we can say that they are always dependent on line 4 and never on line 3.

This program is equivalent to writing into w if the value 1 is observed in x and separately writing into z if value 1 is observed in y, and the dependency arrows in Figure 11 reflect this.

Here we use a superscript notation in our program futures to distinguish events by values. An event where line 3 observes the value 1 is referred to as 3^1, and events where it reads 2 referred to as 3^2, and likewise for line 4.

This asymmetric dependency is made clearer by looking at the event structure generated for the program, shown in Figure 11. The dependency edges shown in the futures are represented as yellow arrows. They show that we cannot write 1 to w without observing 1 at x, and that we cannot write 1 to z without observing 1 at y.

Init: x = 0 ∧ y = 0

Thread 1

$\{[x = 0]_2 \wedge [y = 0]_2\}_F$
$\{[x = 0]_2 \wedge [y \in \{0, 1\}]_2\}_{F_2}$
1: [x] := 1
$\{([x \approx 0]_2 \vee [x \approx 1]_2)$
$\wedge [y = 0]_2\}_{F_1}$
2: [y] := 1
$\{([x \approx 0]_2 \vee [x \approx 1]_2)$
$\wedge ([y \approx 0]_2 \vee [y \approx 1]_2)\}_{\{\emptyset\}}$

Thread 2

$\{I\}_G$
$\{r1 = 0 \wedge r2 = n \wedge I\}_{G_{4^n}}$
3: r1 := [x] ;
$\{r1 = m \wedge r2 = 0 \wedge I\}_{G_{3^m}}$
4: r2 := [y] ;
$\{r1 = m \wedge r2 = n \wedge I\}_{G_{3^m,4^n}}$
if r1 = 1 & r2 = 1 {
$\{r1 = 1 \wedge r2 = 1 \wedge I\}_{G_{3^1,4^1}}$
$\{r1 = 1 \wedge r2 = 1 \wedge [w = 0]_2 \wedge [z = 1]_2\}_{G_{3^1,4^1,6}}$
$\{r1 = 0 \wedge r2 = 1 \wedge [w = 0]_2 \wedge [z = 1]_2\}_{G_{4^1,6}}$
5: [w] := 1 ;
$\{r1 = 1 \wedge r2 = 1 \wedge [w = 1]_2 \wedge [z = 0]_2\}_{G_{3^1,4^1,5}}$
$\{r1 = 1 \wedge r2 = 0 \wedge [w = 1]_2 \wedge [z = 0]_2\}_{G_{3^1,5}}$
6: [z] := 1 ; }
if r1 = 0 & r2 = 1 {
$\{r1 = 0 \wedge r2 = 1 \wedge I\}_{G_{3^0,4^1}}$
7: [z] := 1 }
if r1 = 1 & r2 = 0
$\{r1 = 1 \wedge r2 = 0 \wedge I\}_{G_{3^1,4^0}}$
8: [w] := 1 ;
$\{[w = r1]_2 \wedge [z = r2]_2\}_{\{\emptyset\}}$

$\{[w = r1]_2 \wedge [z = r2]_2\}_{\{\emptyset\}}$

Fig. 10. Partial conditional: $F \mathrel{\widehat{=}} \{\Delta_{\{1,2\}}\}$ $G \mathrel{\widehat{=}} \{\{3^1 \prec 5, 4^1 \prec 6\}, \{3^1 \prec 8\} \cup \Delta_{\{4^0\}}, \{4^1 \prec 7\} \cup \Delta_{\{3^0\}}, \Delta_{\{3^0,4^0\}}\}$ and $I \mathrel{\widehat{=}} [w = 0]_2 \wedge [z = 0]_2$.

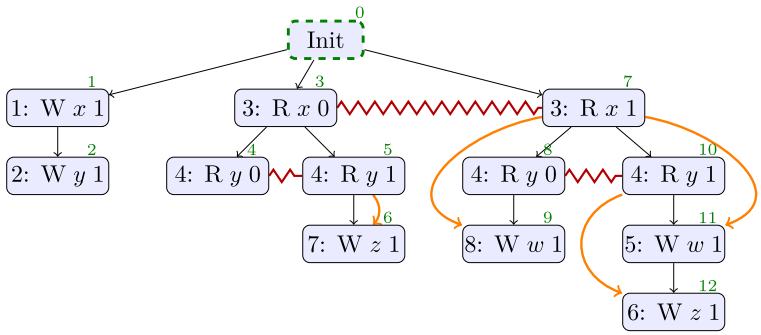

Fig. 11. Event structure generated by the program in Figure 10, including the dependency edges for the writes in Thread 2 (represented by the orange arrows).

7.4 Random Number Generator

We now look at the **random number generator (RNG)** litmus test (see Figure 12) from Reference [6], which in a poorly defined semantics has the potential to terminate so that line 6 is executed. Without any notion of semantic dependency forcing events into a partial order, a semantics could choose to execute the write on line 6 early, then observe this write at line 1 to load the value 99 into register 1 and eventually propagate the value 100 from y into x to be observed at line 5. This

```
Init: x = 0 ∧ y = 0
```

Thread 1	Thread 2	Thread 3
$\{[y = 0]_2 \wedge [y = 0]_3 \wedge r2 = 0$	$\{[x = 0]_{1,2,3}$	$\{\forall i \notin \{0,1\} \Rightarrow \neg[x \approx i]_3$
$\wedge\, \forall i.\, i \neq 0 \Rightarrow \neg[x \approx i]_1\}_F$	$\wedge\, \forall i.\, i \notin \{0,1\} \Rightarrow \neg[y \approx i]_2\}_G$	$\wedge\, \neg[x \approx 99]_{1,2,3}\}_H$
`1: r1 := [x]`	`3: r2 := [y]`	`5: r3 := [x]`
$\{[y = 0]_3 \wedge r1 = 0\}_{F_1}$	$\{[x = 0]_{1,2,3} \wedge r2 \in \{0,1\}\}_{G_3}$	**if** `r3 = 100` **then**
`2: [y] := r1 + 1`	`4: [x] := r2`	$\bot$
$\{true\}_{\{\emptyset\}}$	$\{true\}_{\{\emptyset\}}$	`6: [x] := 99`
		$\{\neg[x \approx 99]_{1,2,3}\}_{\{\emptyset\}}$

$$\{\neg[x \approx 99]_{1,2,3}\}_{\{\emptyset\}}$$

Fig. 12. RNG, where $F \mathrel{\widehat{=}} \{\{1 \prec 2\}\}$, $G \mathrel{\widehat{=}} \{\{3 \prec 4\}\}$, $H \mathrel{\widehat{=}} \{\Delta_{\{5^0\}}, \Delta_{\{5^1\}}\}$.

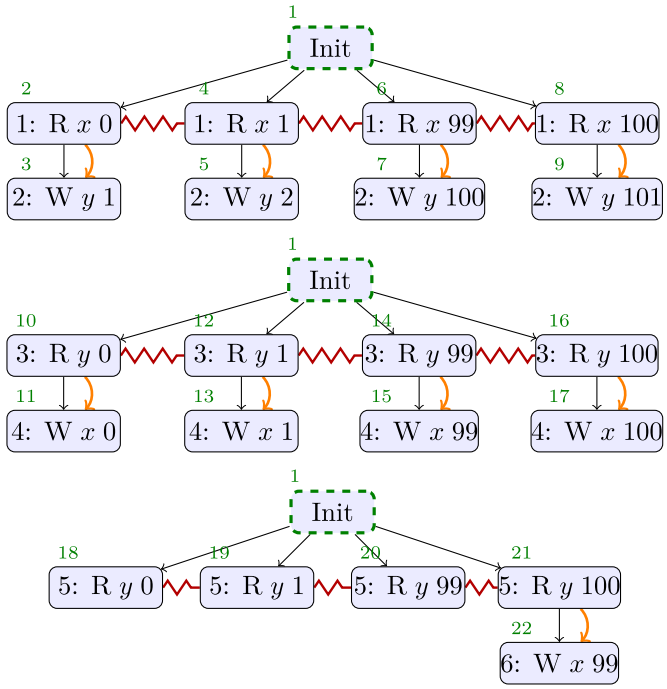

Fig. 13. Event structure generated by the program in Figure 12, with each thread displayed separately.

outcome is technically permitted by ISO C++ [4]. Our proof outline shows that this cannot be the case in our semantics, i.e., the program satisfies the postcondition $\neg[x \approx 99]_{1,2,3}$.

The program is similar to Figure 9 but contains an additional semantic dependency between lines 3 and 4, forcing in-order execution. The event structure generated from this program is shown in Figure 13. We include an event corresponding to line 6, but it never appears in a future, because it is not contained in any complete execution. For an execution to be complete, each read must take its value from a write in the same execution, and there cannot be a cycle in semantic dependency and the reads-from relation in any execution allowed by the model. There is only one way to include event 6 in a complete execution: Event 6 must read from event 22, event 16 from event 7, and event 21 from event 17. Combining the orange semantic dependency arrows on the diagram with

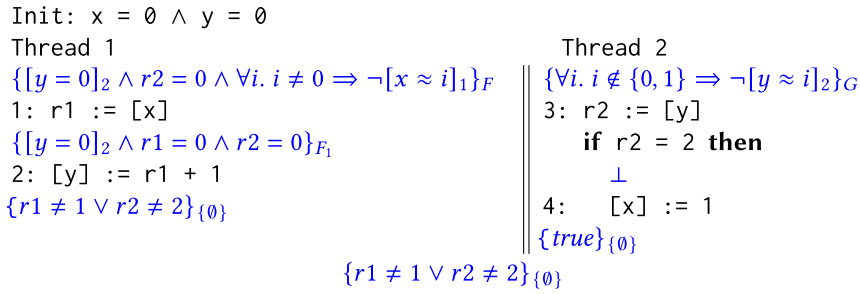

Fig. 14. Simple LB+data-addr+ctrl, where $F \mathrel{\widehat{=}} \{\{1 \prec 2\}\}$ and $G \mathrel{\widehat{=}} \{\Delta_{\{3^0\}}, \Delta_{\{3^1\}}\}$.

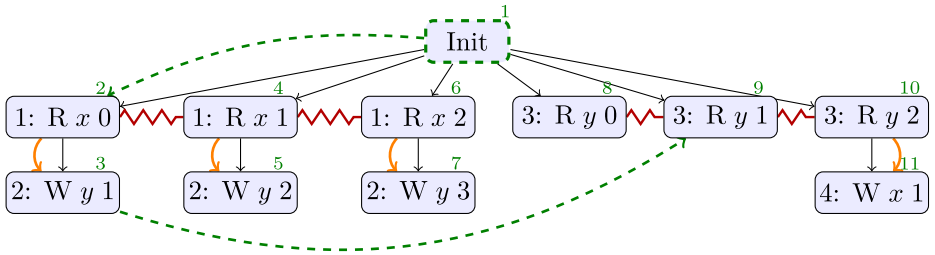

Fig. 15. Event structure for the program in Figure 14, including the reads-from edges (dashed green lines) for an execution observing 1 at y in Thread 2. Note that we cannot construct a complete execution that reads 2, as Init can only write 0 to a location.

the resulting reads-from arrows forms a cycle, and thus the execution is discarded. We indicate this in our proof outline with a $\bot$ replacing an expected precondition for line 6. The following theorem has been verified in Isabelle/HOL.

Theorem 7.1. *The proof outline in Figure 12 is valid.*

7.5 LB+data-add+ctrl

The next two examples (Figures 14 and 16) are designed to test whether a chain of semantic dependencies on data, addresses, and control are properly modelled by the semantics. These examples are variations of the program in Section 7.4, where we guarantee that the branch in Thread 2 is not taken. The difference is that its "environment" comprises Thread 1 that comprises a read of x followed by a write to a variable that Thread 2 reads. Since Thread 1 contains no out-of-order behaviours, it is impossible for Thread 2 to read the value 2.

For Figure 14 as we can see by the corresponding event structure Figure 15, although MRD seemingly generates an event 3:R y 2 followed by 4:W x 1, when considering the reads-from relation together with the RC11 axioms, there is no complete execution that contains such a future.

In Figure 16, Thread 1 contains a longer dependency chain. In particular, it includes chaining of the dependency through thread-local registers as well as a thread-local read of a global variable. Our notion of semantic dependency correctly represents a linear order for this example, and, consequently, this creates no additional proof burden compared to a standard proof in RC11 [9, 10].

The proofs of both programs are straightforward, since they do not contain any intra-thread re-orderings. Again, we have shown correctness of both proof outlines using our existing Isabelle/HOL formalisation [10].

Theorem 7.2. *The proof outlines in Figures 14 and 16 are valid.*

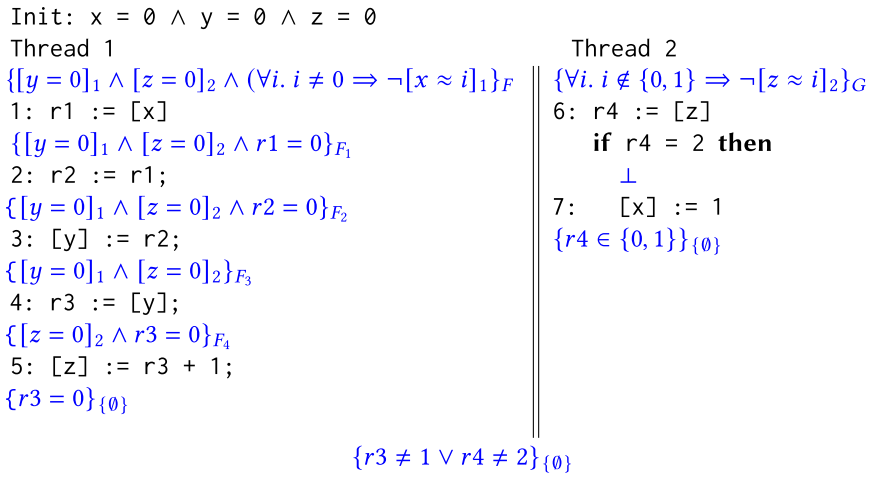

Fig. 16. LB+data-add+ctrl, where $F \mathrel{\widehat{=}} \{\{1 \prec 2 \prec 3 \prec 4 \prec 5\}\}$ and $G \mathrel{\widehat{=}} \{\Delta_{\{6^0\}}, \Delta_{\{6^1\}}\}$.

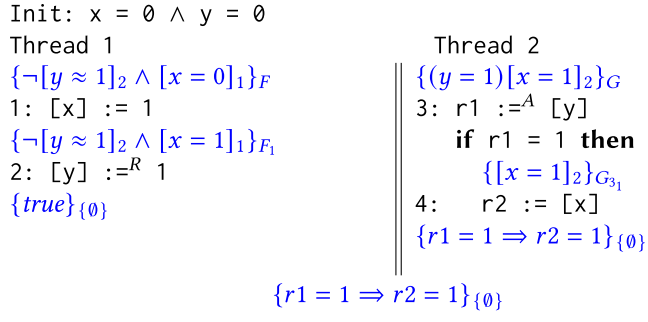

Fig. 17. Message passing, where $F \mathrel{\widehat{=}} \{\{1 \prec 2\}\}$ and $G \mathrel{\widehat{=}} \{\Delta_{\{3^0\}}, \{3^1 \prec 4^1\}\}$.

7.6 MP+rel+acq

Our final example is the message passing example that handles release-acquire synchronisation. We make use of MRD to establish order between the statements. Here, program order between control labels 1 and 2 is preserved by MRD, since 2 contains a releasing write. Similarly, program order between lines 3 and 4 is preserved, since line 3 contains an acquiring read. Thus, we obtain the futures indicated in Figure 17. Since all program order is preserved, the weak memory view assertions in the proof outline from our earlier work [9, 10] can be used without change.

Thread 1 assigns 1 to x using a relaxed write then sets the value of y (also known as the flag) to 1 using a releasing write. Thread 2 reads the value of y using an acquiring read, and if this read sees the write performed by Thread 1, proceeds to read x. C11 guarantees that the releasing write and the corresponding acquiring read *synchronise*. Once synchronisation has occured, the stale (initial value) of x should no longer be visible to Thread 2. Thus, upon termination if $r1 = 1$ holds, then $r2 = 1$ must hold.

We record the transfer of information using the conditional observation assertion $(y = 1)[x = 1]_2$ in Thread 2. Details of this assertion may be found in our prior work [9, 10].

8 ISABELLE/HOL MECHANISATION

Our mechanisation adapts our existing Isabelle/HOL development [9, 10]. The aim of our previous Isabelle/HOL development was to provide a mechanised verification environment for RC11

programs where thread orders were assumed to be preserved. This assumption does not hold anymore in the context of the current work where re-ordering in a thread is allowed in certain situations.

We adapted our previous development by encoding the future-based transition relation introduced in Section 4.2 to replace the program-counter-based transition relation in Reference [9]. Since the underlying memory model semantics remains the same and only the program semantics is changed (to allow thread re-ordering), all the view-based assertions and proof rules in References [9, 10] are still valid and can be reused.

To use our verification environment, one needs to calculate future sets, using MRD or another memory model, and then encode them in Isabelle. The encoded future sets will allow us to generate proof obligations for the Owicki–Gries local correctness and non-interference proof.

To illustrate our mechanisation we use the example in Figure 14. We have the following two future sets for this example: $F = \{\{1 \prec 2\}\}$ and $G = \{\Delta_{\{3^0\}}, \Delta_{\{3^1\}}\}$. We model line numbers 1–4 as `L1`–`L4` using a datatype called `PC`.

A future is a partial order of pairs of type `C` $\widehat{=}$ `PC × (V option)`. Here, `V option` defines an `option` type that may either carry the value "`Some v`" corresponding a read of value `v` or "`None`" if the event at the given line is a write. The type of a future set is defined as `((C × C) set) set`. We define the future sets F and G (of Figure 14) in Isabelle/HOL as follows:

```
F = { {((L1, None), (L2, None))} }
G = { {((L3, Some 0), (L3, Some 0))},  {((L3, Some 1), (L3, Some 1))} }
```

For simplicity, in the Isabelle/HOL encoding, when constructing the initial future sets, we omit reflexive pairs in a future if they already appear in some pair of the future. For instance, in `F`, we omit the pairs `((L1, None), (L1, None))` and `((L2, None), (L2, None))`. After executing `(L1, None)`, we would be left with a new future `{((L2, None), (L2, None))}`.

We associate proof outlines with future sets in the same way as the examples presented in previous sections. Execution of each line of the code will lead to a new future set based on the memory operation (i.e., read or write) associated with that line of the code and the returned value for reads.

We then define a number of lemmas to generate local correctness and non-interference proof obligations from the encoded programs and proof outlines. For each thread, we define a local correctness lemma that shows if a minimal line of code (w.r.t. the current future set) is executed, then it establishes the assertion associated with the new future set. The local correctness lemma for Thread 1 of Figure 14 is encoded in Isabelle/HOL as follows[2]:

```
lemma po_local_t1:
  assumes f ∈ {F, F1}
      and c ∈ f
      and minimal p c
      and next_future p f f'
      and prog_future_inv f s
      and prog t1 p s s'
    shows prog_future_inv f' s'
```

where `f` is the current future set, `c` is a member of the future set, `p` is the minimal event of `c`, `s` is the pre-state, and `s'` is the post-state resulted from execution of `p`. We assume `prog` is the transition

[2]We leave out some minor details of the Isabelle/HOL syntax.

relation that models program execution for p and program_future_inv is a function that returns the assertions associated with a given future set f in a specific state s.

The above lemma gives rise to a number of proof obligations depending on different cases for f, c, and p. For instance, for the case f = F (i.e., f is the the initial future set) we will have c = {((L1, None), (L2, None))}, and p = (L1, None). This means that we have to show that prog t1 p s s′ establishes prog_future_inv f′ s′ if executed in a state such that prog_future_inv f s holds, where f′ is obtained from f by removing p from f (as defined by next_future p f f′). In particular, for f = F and p as instantiated above we have f′ = F_1.

The global correctness lemma is similar and takes the following form:

```
lemma po_global_t1:
  assumes f ∈ {F, F₁, F∅}
      and g ∈ {G, G₃₂}
      and c ∈ g
      and minimal p c
      and prog_future_inv f s
      and prog_future_inv g s
      and prog t2 p s s'
    shows prog_future_inv f s'
```

representing the execution of steps of t1 preserving the invariants of t2.

9 RELATED WORK

Work based on assumptions unsound in C++. Most logics for weak memory are based on simplifying assumptions that exclude thin-air reads by ensuring program order is respected, even when no semantic dependency exists [9, 13, 14, 19, 21]. These assumptions incorrectly exclude the relaxed outcome of load buffering tests like Figure 1, introducing unsoundness when applied to languages like C++: Compiling Figure 1 for an ARM processor produces code that does exhibit the relaxed behaviour. The logic of Lundberg et al. [24] correctly discards many of this spurious program ordering, but it does not handle concurrency. The CDSChecker tool of Norris and Dempsky [26] aims for high-performance model checking of concurrent C++ code. Its mechanism relaxes constraints on where reads must read from to model load-buffering behaviour. This mechanism is known to deviate from the C++ Standard, but it has not been shown to be consistent with a concurrency model that solves the thin-air problem. We provide an operational semantics of C++ that solves the thin-air problem, where prior attempts use simplifying assumptions like those of prior logics [9, 25].

Thin-air-free semantics. Each of the concurrency definitions that solves the out-of-thin-air problem is remarkably complex [6, 17, 20, 23, 28, 29] and so, too, is MRD. We choose to base our logic on MRD, because MRD's semantic dependency relation hides much of the complexity of the model that calculates it. Where previously we would rely on program ordering, now we consider the semantic dependency provided by MRD. Jeffrey et al. offer an alternative calculation of semantic dependency that validates additional compiler optimisations based on *pomsets with predicate transformers* [18], albeit with an even higher computational cost for evaluating the model. Semantic dependency is sufficiently opaque in our logic and semantics that we can accommodate this new definition and future developments of semantic dependency trivially.

Logics for thin-air-free models. There are four logics built above concurrency models that solve the thin-air problem: Three of these are very simple and apply to only a handful of examples [16, 17, 20], and one is a separation logic built above the Promising Semantics [31]. Unfortunately, the

Promising Semantics allows unwanted out-of-thin-air behaviour, forbidden by MRD [28], and as a result may not support type safety [16].

Recent logics for other memory models. Recent works have focussed on rely-guarantee reasoning at the hardware level for both multicopy atomic [7] and non-multicopy atomic memory models [8]. Here, a rely condition is used to capture reorderings permitted in hardware to show that sequentially consistent rely-guarantee proofs are preserved. However, it is currently unclear whether the semantic dependency relation can be captured by the reordering-based semantics that these frameworks are based on. Another line of work [12] aims to unify weak memory reasoning over hardware and programming languages using an axiomatically defined memory model abstraction. However, the only instances that have been developed are SC, TSO, and RC11 (where program order is preserved). Integrating semantic dependencies into such abstract memory models is an interesting direction for future work.

10 CONCLUSIONS AND FUTURE WORK

The subtle behaviour of concurrent programs written in optimised languages necessitates good support for reasoning, but existing logics make unsound assumptions that rule out compiler optimisations or use underlying concurrency models that admit out of thin-air behaviour (see Section 9). We present a logic built above MRD. MRD has been recognised as a potential solution to the thin-air problem in C and C++ by the ISO [15] and is the best guess at a C++ model that allows optimisation and forbids thin-air behaviours.

We follow a typical path for constructing a logic and start with an operational semantics that we show equivalent to MRD and then, as much as possible, follow the reasoning style of Owicki–Gries. In each case, we diverge from a typical development, because we cannot adopt program order into our reasoning system and instead must follow semantic dependency. Our logic supplants linear program order with a partial order, and where traditional pre- and post-conditions are indexed by the program counter, and here we index them by their position in the partial order. This approach will work with any memory model that can provide a partially ordered structure over individual program actions.

The challenge in using the logic presented here is in managing the multitude of proof obligations that follow from the branching structure and lack of order in semantic dependency. This problem represents an avenue for further work: It may be possible to obviate the need for some of these additional proof obligations within the logic or to provide tools to more conveniently manage them.

ACKNOWLEDGMENTS

We thank Simon Doherty for discussions on an earlier version of this work.

REFERENCES

[1] S. V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: A tutorial. *IEEE Comput.* 29, 12 (1996), 66–76.

[2] J. Alglave, L. Maranget, and M. 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.

[3] M. Batty, M. Dodds, and A. Gotsman. 2013. Library abstraction for C/C++ concurrency. In *POPL'14*, R. Giacobazzi and R. Cousot (Eds.). ACM, 235–248.

[4] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. 2015. The problem of programming language concurrency semantics. In *ESOP'15 (Lecture Notes in Computer Science)*, J. Vitek (Ed.), Vol. 9032. Springer, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2011. Mathematizing C++ concurrency. In *POPL'11*, T. Ball and M. Sagiv (Eds.). ACM, 55–66.

[6] S. Chakraborty and V. Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383
[7] N. Coughlin, K. Winter, and G. Smith. 2021. Rely/Guarantee reasoning for multicopy atomic weak memory models. In *FM'21 (Lecture Notes in Computer Science)*, M. Huisman, C. S. Pasareanu, and N. Zhan (Eds.), Vol. 13047. Springer, 292–310. https://doi.org/10.1007/978-3-030-90870-6_16
[8] N. Coughlin, K. Winter, and G. Smith. 2022. Compositional reasoning for non-multicopy atomic architectures. *Form. Asp. Comput.* (December 2022). https://doi.org/10.1145/3574137
[9] S. Dalvandi, S. Doherty, B. Dongol, and H. Wehrheim. 2020. Owicki-gries reasoning for C11 RAR. In *ECOOP'20 (LIPIcs)*, R. Hirschfeld and T. Pape (Eds.), Vol. 166. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 11:1–11:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.11
[10] S. Dalvandi, B. Dongol, S. Doherty, and H. 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
[11] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, (2020), 34:1–34:29. https://doi.org/10.1145/3371102
[12] S. Doherty, S. Dalvandi, B. Dongol, and H. Wehrheim. 2022. Unifying operational weak memory verification: An axiomatic approach. *ACM Trans. Comput. Log.* 23, 4 (2022), 27:1–27:39. https://doi.org/10.1145/3545117
[13] S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. 2019. Verifying C11 programs operationally. In *PPoPP'19*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 355–365. https://doi.org/10.1145/3293883.3295702
[14] M. Doko and V. Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *ESOP'17*. 448–475.
[15] O. Giroux. 2019. ISO WG21 SG1 Concurrency Subgroup Vote. (2019). https://github.com/cplusplus/papers/issues/554.
[16] R. Jagadeesan, A. Jeffrey, and J. Riely. 2020. Pomsets with preconditions: A simple model of relaxed memory. *Proc. ACM Program. Lang.* 4 (2020), 194:1–194:30. https://doi.org/10.1145/3428262
[17] A. Jeffrey and J. Riely. 2019. On thin air reads: Towards an event structures model of relaxed memory. *Log. Methods Comput. Sci.* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:33)2019
[18] A. Jeffrey, J. Riely, M. Batty, S. Cooksey, I. Kaysin, and A. Podkopaev. 2022. The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, Article 54 (January 2022), 30 pages. https://doi.org/10.1145/3498716
[19] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP'17*. 17:1–17:29.
[20] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL'17*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. http://dl.acm.org/citation.cfm?id=3009850.
[21] O. Lahav and V. Vafeiadis. 2015. Owicki-gries reasoning for weak memory models. In *ICALP'15 (LNCS)*, Vol. 9135. Springer, 311–323.
[22] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI'17*, A. Cohen and M. T. Vechev (Eds.). ACM, 618–632.
[23] S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis. 2020. Promising 2.0: Global optimizations in relaxed memory concurrency. In *PLDI'20*, A. F. Donaldson and E. Torlak (Eds.). ACM, 362–376. https://doi.org/10.1145/3385412.3386010
[24] D. Lundberg, R. Guanciale, A. Lindner, and M. Dam. 2020. Hoare-style logic for unstructured programs. In *Software Engineering and Formal Methods*, F. de Boer and A. Cerone (Eds.).
[25] K. Nienhuis, K. Memarian, and P. Sewell. 2016. An operational semantics for C/C++11 concurrency. In *OOPSLA'16*. ACM, 111–128.
[26] B. Norris and B. Demsky. 2013. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA'13*. Association for Computing Machinery, New York, NY, 131–150. https://doi.org/10.1145/2509136.2509514
[27] S. S. Owicki and D. Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Inf.* 6 (1976), 319–340. https://doi.org/10.1007/BF00268134
[28] M. Paviotti, S. Cooksey, A. Paradis, D. Wright, S. Owens, and M. Batty. 2020. Modular relaxed dependencies in weak memory concurrency. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 599–625.
[29] J. Pichon-Pharabod and P. Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL'16*, R. Bodík and R. Majumdar (Eds.). ACM, 622–633. https://doi.org/10.1145/2837614.2837616
[30] A. Podkopaev, O. Lahav, and V. Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3 (2019), 69:1–69:31. https://doi.org/10.1145/3290382

[31] K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP'18 (Lecture Notes in Computer Science)*, A. Ahmed (Ed.), Vol. 10801. Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[32] G. Winskel. 1986. Event structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course (Lecture Notes in Computer Science)*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.), Vol. 255. Springer, 325–392. https://doi.org/10.1007/3-540-17906-2_31

[33] D. Wright, M. Batty, and B. Dongol. 2021. Owicki-gries reasoning for C11 programs with relaxed dependencies. In *FM'21 (Lecture Notes in Computer Science)*, M. Huisman, C. S. Pasareanu, and N. Zhan (Eds.), Vol. 13047. Springer, 237–254. https://doi.org/10.1007/978-3-030-90870-6_13

[34] D. Wright, S. Dalvandi, B. Dongol, and M. Batty. 2022. Isabelle/HOL files for "Mechanised Operational Reasoning for C11 Programs with Relaxed Dependencies." (Nov. 2022). https://doi.org/10.6084/m9.figshare.21507636

Received 31 March 2022; revised 24 November 2022; accepted 5 January 2023