Owicki-Gries Reasoning for C11 Programs with Relaxed Dependencies

Daniel Wright[1(✉)], Mark Batty[1], and Brijesh Dongol[2]

[1] University of Kent, Canterbury, UK
daw29@kent.ac.uk
[2] University of Surrey, Guildford, UK

Abstract. Deductive 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 paper, 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 (repaired C11), and demonstrate the use of this logic over several example proofs.

1 Introduction

Significant advances have now been made on the semantics of (weak) memory models using a variety of axiomatic (aka 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 [9,11,15,26] and adaptations of classical Owicki-Gries reasoning [7,17]. At the level of languages, there has been a particular focus on C11 (the 2011 C/C++ standard).

Due to the complexity of C11 [4], 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., [7,10,15,17]) assume a memory model that guarantees that program order (aka sequenced-before order) is maintained [18]. While this restriction makes the logics and proofs of program correctness more manageable, it precludes reasoning

We thank Simon Doherty for discussions on an earlier version of this work. Wright is supported by VeTSS. Batty is supported by EPSRC grants EP/V000470/1 and EP/R032971/1, and the Royal Academy of Engineering. Dongol is supported by EPSRC grants EP/V038915/1, EP/R032556/1, EP/R025134/2, VeTSS and ARC Discovery Grant DP190102142.

M. Huisman et al. (Eds.): FM 2021, LNCS 13047, pp. 237–254, 2021.
https://doi.org/10.1007/978-3-030-90870-6_13

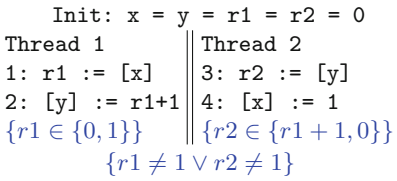

Fig. 1. Load-buffering with a semantic dependency

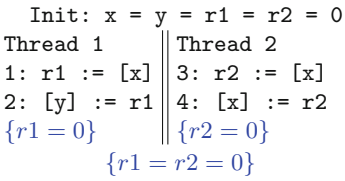

Fig. 2. Load-buffering with two dependencies

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 [5], 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 Fig. 1, where there is nothing to enforce the order of the load and store in the second thread [25]. In Fig. 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. 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 [6,14,16,19,23] 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 [13,23]. This work builds on top of the recent MRD (Modular Relaxed Dependencies) semantics by Paviotti et al. [23]. 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 Fig. 1 contains only the semantic dependency between lines 1 and 2, whereas the program in Fig. 2 contains semantic dependencies between lines 1 and 2, and lines 3 and 4. The program in Fig. 1 may therefore execute line 4 before line 3, while the program in Fig. 2 must execute both threads in order.

Contributions. Although precise, there is currently no direct mechanism for reasoning about programs under MRD because MRD is a denotational semantics defined over an event structure [27]. This paper 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 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. Here, we adapt the operational model of weak memory effects by Doherty et al. [10] so that it follows semantic dependency rather than the more restrictive thread order used in earlier works [7,10,15,17].

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. [7] 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 [7,17], since we relax thread order, the standard approach to Hoare-style proof decomposition is not possible.

Overview. This paper is organised as follows. We recap MRD in Sect. 2 and an operational semantics for RC11 in Sect. 3. Then in Sect. 4, we present a combined semantics, where program order in RC11 is replaced by a more relaxed order defined by MRD. A Hoare-like logic for reasoning about relaxed program execution together with Owicki-Gries-like rules for reasoning about interference is given in Sect. 5. We present an example proof in Sect. 6.

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, which precludes development of a program logic.

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 $\mathrm{i{:}R}\,x\,v$, $\mathrm{i{:}W}\,x\,v$, 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* $\rho : Reg \to Val$ 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 $eval \in E \times (Reg \to Val) \to Val$. 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 paper, we assume that stores and loads are *relaxed* [3,10,18] 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 which 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. 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.

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

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 CAS, fences, non-atomic accesses and SC accesses [4].

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.

Denotational MRD. For the purposes of this paper (i.e., the development of the operational semantics and associated program logic), the precise details of the MRD semantics [23] 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 is returned by the semantic interpretation function $[\![P]\!]$. This gives a *coherent event structure* of the form:

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

where

- $L = (E, \sqsubseteq, \#, Lab)$ is an *event structure* [27] equipped with a labelling function Lab from events (represented internally as integer identifiers) to actions. The set E contains all events in the structure, $a_1 \sqsubseteq a_2$ for $a_1, a_2 \in E$ iff a_1 is program ordered before a_2. Events in $\#$ cannot happen simultaneously, e.g., two reads of a variable returning different values are in conflict.
- $S = (A, \text{LK}, \text{RF}, \text{DP})$ 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 of relations representing *lock/unlock order*, a *reads-from relation* and *dependency order* 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* (c.f., [2]), a subset of the program order of F that is respected by the memory model.

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 *Lab* to connect actions to events. From the coherent event structure, we use $\leq$ to enforce ordering alongside dependency order[1].

Example 1. The event structure in Fig. 3 represents the denotation of the program in Fig. 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. Instead, each read must indicate a write whose value it is observing during the axiomatic checks, subject to various coherence restrictions. A read will only appear in a *complete execution* if it can satisfy this requirement.

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). 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.

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

For the program in Fig. 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.

3 Operational Semantics with Relaxed Write Propagation

To reason about the relaxed propagation of writes, Doherty et al. [10] build an operational semantics that is equivalent to a declarative (aka axiomatic)

[1] MRD also defines a set of axioms that describes when a particular execution is consistent with a denotation. We do not discuss these in detail here, but they are used in the soundness and completeness proofs.

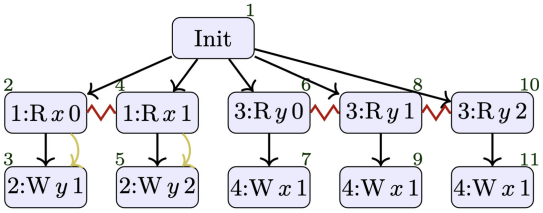

Fig. 3. Coherent event structure for the program in Fig. 1

semantics [18], which defines consistency of *tagged action graphs*. The operational semantics allows the stepwise construction of such graphs, in contrast to a declarative approach which only considers complete executions that are either accepted or rejected by the axioms of the memory model. We do not discuss the declarative semantics here, focusing instead on the operational model, which we generalise in Sect. 4. We also note that like other prior works [15,17,18], the existing operational semantics [10] assumes program order within a thread is maintained.

Formally, *tagged actions* are triples (g, a, t), where g is a tag (uniquely identifying the action); a is a read or write action (potentially annotated as a releasing write or acquiring read), and t is a thread (corresponding to the thread that issued the action). We let TA be the set of all tagged actions, $\mathsf{Wr}, \mathsf{Wr}_R, \mathsf{Rd}, \mathsf{Rd}_A \subseteq TA$ be the set of write, releasing write, read and acquiring reads, respectively. Note that Wr is the set of all writes, including those from Wr_R (similarly, Rd).

A *tagged action graph* is a tuple $(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ where D is a set of tagged actions, which may correspond to read or write actions, and sb, rf and mo are relations over D. Here, sb is the *sequenced before* relation, where $b_1\ \mathsf{sb}\ b_2$ iff b_1 and b_2 are tagged actions of the same thread and b_1 is executed before b_2. $\mathsf{rf} \subseteq \mathsf{Wr} \times \mathsf{Rd}$ is the *reads-from* relation [2] relating each write to the read that reads from that write. Finally, $\mathsf{mo} \subseteq \mathsf{Wr} \times \mathsf{Wr}$ denotes *modification order* (aka coherence order), which is the order in which writes occur in the system, and hence the order in which writes must be seen by all threads. Note that if $b_1\ \mathsf{mo}\ b_2$, then b_1 and b_2 must act on the same variable. Moreover, $\mathsf{mo}|x$ is a total order, where $\mathsf{mo}|x$ is the relation mo restricted to writes of variable x.

We assume tagged action graphs are initialised with writes corresponding to the initialisation of the program, and relations sb, rf and mo are initially empty.

Following Lahav et al. [18], the so-called repairing or restricted C11 model (which is the model in [7,10,15]) instantiates sequence-before order to the *program order* relation [2]. This disallows statements within a thread from being executed out-of-order (although writes may be propagated to other threads in a relaxed manner). In Sect. 4, we present an alternative instantiation of sb using the semantic dependencies generated by MRD to enable out-of-order executions within a thread can be considered.

To characterise the operational semantics, we must define three further relations: *happens before*, denoted hb (which captures a notion of causality); *from read*, denoted fr (which relates each read to the write that overwrites the value read), and *extended coherence order*, denoted eco (which fixes the order of writes and reads). Formally we have:

$$\mathsf{hb} = (\mathsf{sb} \cup (\mathsf{rf} \cap \mathsf{Wr}_R \times \mathsf{Rd}_A))^+ \qquad \mathsf{fr} = (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus Id \qquad \mathsf{eco} = (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr})^+$$

where Id is the identity relation, ; denotes relational composition, and $^+$ denotes transitive closure. Note that there is only a happens-before relation between a write-read pair related by rf if the write is releasing and the read is acquiring.

$$\textsc{Read}\ \frac{\begin{array}{ccccc} b = (g, a, t) & g \notin tags(D) & a \in \{\mathrm{i{:}R}\, x\, n, \mathrm{i{:}R}^A\, x\, n\} \\ \sigma = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) & w \in OW_\sigma(t) & var(w) = x & wrval(w) = n \end{array}}{(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \overset{b}{\rightsquigarrow} (D \cup \{b\}, \mathsf{sb} +_D b, \mathsf{rf} \cup \{(w, b)\}, \mathsf{mo})}$$

$$\textsc{Write}\ \frac{\begin{array}{ccccc} b = (g, a, t) & g \notin tags(D) & a \in \{\mathrm{i{:}W}\, x\, n, \mathrm{i{:}W}^R\, x\, n\} \\ \sigma = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) & w \in OW_\sigma(t) & var(w) = x \end{array}}{(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \overset{b}{\rightsquigarrow} (D \cup \{b\}, \mathsf{sb} +_D b, \mathsf{rf}, \mathsf{mo}[w, b])}$$

Fig. 4. Memory semantics

With these basic relations in place, we are now in a position to define the transition relation governing the operational rules for read and write actions. The rules themselves are given in Fig. 4. Assuming σ denotes the set of all tagged action graphs, each transition is a relation $\rightsquigarrow \subseteq \Sigma \times TA \times \Sigma$. We write $\sigma \overset{b}{\rightsquigarrow} \sigma'$ to denote $(\sigma, b, \sigma') \in \rightsquigarrow$.

To accommodate relaxed propagation of writes, the semantics allows different threads to have different views of the system, formalised by a set of *observable writes*. These are in turn defined in terms of a set of *encountered writes*, denoted $EW_\sigma(t)$, which are the writes that thread t is aware of (either directly or indirectly) in state σ:

$$EW_\sigma(t) = \{w \in \mathsf{Wr} \cap D_\sigma \mid \exists b \in D_\sigma.\ tid(b) = t \wedge (w, b) \in \mathsf{eco}_\sigma^?; \mathsf{hb}_\sigma^?\}$$

Here $R^?$ is the reflexive closure of relation R and tid returns the thread identifier of the given tagged action. Thus, for each $w \in EW_\sigma(t)$, there must exist a tagged action b of thread t such that w is either eco-, hb- or $\mathsf{eco}; \mathsf{hb}$-prior to b. From these we determine the *observable writes*, which are the writes that thread t can observe in its next read. These are defined as:

$$OW_\sigma(t) = \{w \in \mathsf{Wr} \cap D_\sigma \mid \forall w' \in EW_\sigma(t).\ (w, w') \notin \mathsf{mo}_\sigma\}$$

$$\frac{n = eval(e, \gamma(t)) \quad a = \mathrm{i{:}W}^{[R]}\, x\, n}{(i\colon [x] :=^{[R]} e, \gamma) \xrightarrow{a}_t (\mathbf{skip}, \gamma)} \qquad \frac{a = \mathrm{i{:}R}^{[A]}\, x\, n \quad \rho' = \gamma(t)[r := n]}{(i\colon r :=^{[A]} [x], \rho) \xrightarrow{a}_t (\mathbf{skip}, \gamma[t := \rho'])}$$

$$\textsc{Prog}\ \frac{(P(t), \gamma) \xrightarrow{a}_t (C, \gamma')}{(P, \gamma) \xrightarrow{a}_t (P[t := C], \gamma')} \qquad \textsc{Full}\ \frac{b = (g, a, t) \quad (P, \gamma) \xrightarrow{a}_t (P', \gamma') \quad \sigma \overset{b}{\rightsquigarrow} \sigma'}{(P, (\sigma, \gamma)) \overset{b}{\Longrightarrow} (P', (\sigma', \gamma'))}$$

Fig. 5. Interpreted operational semantics of programs (sample)

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

We now describe each of the rules in Fig. 4. Relations rf and mo are updated according to the write actions in D that are observable to the thread t executing the given action. A read action b may read from any write $w \in OW_\sigma(t)$. In the post state, we obtain a new tagged action set $D \cup \{b\}$, sequenced-before relation $\mathsf{sb} +_D b$ (defined below) and reads-from relation $\mathsf{rf} \cup \{(w, b)\}$. Relation mo is unmodified. Formally, $\mathsf{sb} +_D b$ introduces b at the end of sb for thread t:

$$\mathsf{sb} +_D b = \mathsf{sb} \cup (\{b' \in D \mid tid(b') \in \{tid(b), 0\}\} \times \{b\})$$

Note that we assume that initialisation is carried out by a unique thread with id 0, and that we require the set D as an input to cope with initialisation where sb may be empty for the given thread.

The write rule is similar except that it leaves rf unchanged and updates mo to $\mathsf{mo}[w, b]$. Given that $R[x]$ is the relational image of x in R, we define $R_{\Downarrow x} = \{x\} \cup R^{-1}[x]$ to be the set of all elements in R that relate to x (inclusive). The insertion of a tagged write action b directly after a w in mo is given by

$$\mathsf{mo}[w, b] = \mathsf{mo} \cup (\mathsf{mo}_{\Downarrow w} \times \{b\}) \cup (\{b\} \times \mathsf{mo}[w])$$

Thus, $\mathsf{mo}[w, b]$ effectively introduces b immediately after the w in mo.

The final component of the operational semantics is a set of rules that link the program syntax in Sect. 2 with the memory semantics. In prior work, this was through a set of operational rules that generate the actions associated with each atomic statement, combined with the rules in Fig. 4 to formalise the evolution of states. A sample of these rules for reads and writes from memory are given in Fig. 5. These are then lifted to the level of programs using the rules Prog and Full, where the transition rule $\overset{b}{\Longrightarrow}$ combines the thread-local semantics $\xrightarrow{a}_t$ and global-state semantics $\overset{b}{\rightsquigarrow}$. Note that we model parallel composition as functions from thread identifiers to commands, thus $P[t := C]$ represents the program, where the command for thread t is updated to C.

The rules in [10] generate actions corresponding to program syntax (i.e., $\xrightarrow{a}_t$) in *program order*. In this paper, we follow a different approach—the actions will

be generated by (and executed in) the order defined by MRD. This therefore allows both reordering of program statements and relaxed write propagation.

4 Operational Semantics over MRD

Recall that the MRD semantic interpretation function $[\![P]\!]$ 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 Sect. 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 $\lambda t.\ \overline{P(t)}$.

To retain the ordering recognised during program execution, we introduce *futures*, which are sets of MRD events partially ordered by the semantic dependency and preserved program order relations. Essentially, instead of taking our operational steps in program order over the syntax, we can nondeterministically execute any statement which our futures tell us we have executed all necessary predecessors of.

For an execution $S = (A, \text{LK}, \text{RF}, \text{DP})$ of an event structure μ, we can construct an initial future $f = (A, \preceq)$, where $\preceq = \text{DP} \cup \leq_{|A}$ and $\leq_{|A}$ is the preserved program order $\leq$ (see Sect. 2) restricted to events in A. We say that an action a is *available* in a future $(K, \preceq)$ iff there exists some event g with label a such that $g \in K$ and g is minimal in $\preceq$, i.e., for all events $g' \in K$, $g' \not\preceq g$. If a is available in f, then the *future of* a *in* f, denoted $a \triangleright f$, is the future

$$a \triangleright f \iff (K', \preceq_{|K'}) \qquad \text{where } K' = K \setminus \{g \mid \lambda_\mu(g) = a\}$$

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

$$a \triangleright F = \{a \triangleright f \mid f \in F \wedge a \text{ available in } f\}$$

Note that a is *enabled* in F iff $a \triangleright F \neq \emptyset$.

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

4.2 Future-Based Transition Relation

Our operational semantics is defined by the transition relation in FUTURE-STEP given below, which generalises FULL in Fig. 5. The transition relation is defined over $(\overline{Q}, (\sigma, \gamma), F)$, where $\overline{Q}$ is the atomic set corresponding to the program text, (σ, γ) is a configuration and F is a set of *futures*. The rule generalises FULL in the obvious way, i.e., by evolving the configuration as allowed by $\overset{(g,a,t)}{\rightsquigarrow}$ and $\xrightarrow{a}_t$ and consuming an available action a in F. Below, we use $\uplus$ to denote disjoint union and $f[k := v]$ to denote function updates where $f(k)$ is updated to v.

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

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

The following theorem establishes equivalence of our operational semantics and the MRD denotational semantics.

Theorem 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.*

4.3 Example

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

$$\begin{array}{lll} \{2 \prec 3, 6, 7\} & \{2 \prec 3, 8, 9\} & \{2 \prec 3, 10, 11\} \\ \{4 \prec 5, 6, 7\} & \{4 \prec 5, 8, 9\} & \{4 \prec 5, 10, 11\} \end{array}$$

where $\{2 \prec 3, 6, 7\}$ represents the future $(\{2, 3, 6, 7\}, 2 \prec 3 \cup \Delta_{\{2,3,6,7\}})$. The atomic set of the program is

$$\overline{P} = \left\{ \begin{array}{l} 1 \mapsto \{1 : \mathtt{r1} := [\mathtt{x}], 2 : [\mathtt{y}] := \mathtt{r1} + 1\}, \\ 2 \mapsto \{3 : \mathtt{r2} := [\mathtt{y}], 4 : [\mathtt{x}] := 1\} \end{array} \right\}$$

The initial configuration is (σ_0, γ_0), where $\sigma_0 = (\{(0_x, 0{:}\mathrm{W}\, x\, 0, 0), (0_y, 0{:}\mathrm{W}\, y\, 0, 0)\}, \emptyset, \emptyset, \emptyset)$ and $\gamma_0 = \{1 \mapsto \{r1 \mapsto 0\}, 2 \mapsto \{r2 \mapsto 0\}\}$, assuming the initialising thread has identifier 0.

To find out which events we can execute, we check our futures set. We cannot execute events 3 or 5, which both have pre-requisite events that have not yet

been executed. These events are the only available events generated by line 2, so we cannot attempt to execute line 2. We can, however, execute any other line. Suppose we execute line 4, which corresponds to $4 : [x] := 1$ in $\overline{P}(2)$. To use the transition relation $\stackrel{\mu}{\Longrightarrow}$, we need to do the following: (1) determine the corresponding action, a and new thread-local state using $\xrightarrow{a}_2$; (2) generate a tagged action $b = (g, a, 2)$ for a fresh tag g and a new global state using $\overset{b}{\rightsquigarrow}$; and (3) check that a 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 using the WRITE rule in Fig. 4 (full details elided). For (3), we take our candidate futures $a \triangleright F$ by examining which MRD events have the label $4{:}\mathrm{W}\,x\,1$—in this case events 7, 9, and 11. All futures can execute one of these events so the new future set contains all futures in F, each minus the set $\{7, 9, 11\}$.

5 Hoare Logic and Owicki-Gries Reasoning

Recall that the standard Owicki-Gries methodology [22] 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) [7]. Note that an alternative characterisation (also in a model without relaxed program order) has been given in [17][2]. Unfortunately, the unrestricted C11 semantics captured by MRD relaxes program order and hence sequential composition in order to allow behaviours such as those in Fig. 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.

Like Dalvandi et al. [7], we assume assertions are predicates over state configurations. In the current paper, the operational semantics is dictated by the set of futures generated by MRD, thus we require two modifications to the classical meaning. First, like prior work [7], 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 $[\![P]\!]$ and $\overline{P}$ the set normal form of P. We let F_μ be the initial set of futures corresponding to μ.

[2] 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 [17], the introduction of auxiliary variables is unsound.

Definition 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 $\{X\}\ Init; P\ \{Y\}$, where*

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

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 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 $labels(\overline{Q}) = labels(\{Lab_\mu[\pi_1 f] \mid f \in F\})$, where we assume *labels* returns the set of all labels of its argument, π_1 is the project of the first component of the given argument, and $R[S]$ is the relational image of set S over relation R.

Definition 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 $\overline{P}$, we define*

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

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 1 (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 $\{X\} Init; P \{Y\}$ provided $Init \Rightarrow X$.*

By construction, the sets of futures corresponding to different threads are disjoint, i.e., for each future $f \in F_\mu$, we have that $f = \bigcup_t f_{|t}$, where $f_{|t}$ denotes the future f restricted to events of thread t. The only possible inter-thread dependency in MRD is via the reads from relation [23], 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 $F_{|t} = \{f_{|t} \mid f \in F\}$. Then, we obtain the following lemma for decomposing invariants in the same way as Owicki and Gries.

Lemma 2 (Owicki-Gries). *For each thread t, let I_t be a future predicate corresponding to t. If Init $\Rightarrow X$, $X \Rightarrow \forall t.\ I_t(F_{\mu|t})$, 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*[3] $\{I_{t_1}(F_1) \wedge I_{t_2}\}_{F_2}\ \overline{Q}\ \{I_{t_1}(F_1)\}$. *(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.

6 A Verification Example

With the verification framework now in place, we present a correctness proof for the program in Fig. 1. The program and invariant are given in Fig. 6. First, in Sect. 6.1, we present assertions for reasoning about state-configurations.

6.1 View-Based Assertions

As we can see from Fig. 5, the states that we use are *configurations*, which are pairs of the form (σ, γ), where σ is an tagged action graph representing the shared state and γ is mapping from threads to register files representing the local state. Like prior work [7], 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 a time-stamp based semantics [7]. Nevertheless, the principles for defining thread-view assertions also carry over to our setting of tagged action graphs.

In this paper, 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

$$[x = v]_t(\sigma, \gamma) \mathrel{\widehat{=}} \exists w.\ OW_\sigma(t)_{|x} = \{w\} \wedge wrval(w) = v$$
$$[x \approx v]_t(\sigma, \gamma) \mathrel{\widehat{=}} \exists w \in OW_\sigma(t)_{|x}.\ wrval(w) = v$$

where $OW_\sigma(t)|x$ denotes the writes in $OW_\sigma(t)$ restricted to the variable x. Recall that, by definition, for any state σ generated by the operational semantics, the last write to each variable in mo order is observable to every thread. Examples of these assertions in the context of an Owicki-Gries-style proof outline is given in Fig. 6.

[3] Technically speaking, each instance of $I_{t_1}(F_1)$ in the Hoare-triple is a function $\lambda x.\ I_{t_1}(F_1)$.

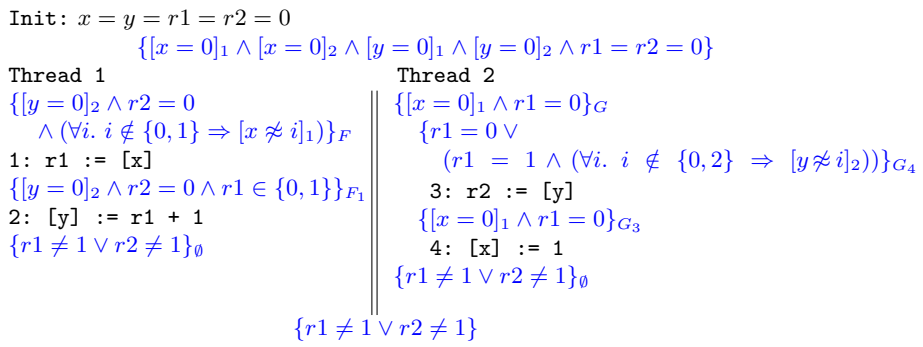

Fig. 6. Proof outline for load buffering with semantic dependencies

6.2 Example Proof

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 $\{2{:}\mathrm{W}\, y\, 2\}$, thus $I(\{3\}) = I(\{5\})$. This isn't always a valid step, as some events which 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 $\{\{1_u < 2, 3_v, 4\} \mid u \in \{0, 1\} \wedge v \in \{0, 1, 2\}\}$, using the notation $\{i_v\}$ to refer to an action with the line number i with a read that returns the value v. 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 $F = \{\{1_u < 2\} \mid u \in \{0, 1\}\}$ and $G = \{\{3_v, 4\} \mid v \in \{0, 1, 2\}\}$ representing the future sets of threads 1 and 2, respectively. We let $F_1 = \{\{2\}\}$ be the future set after executing line 1, $G_3 = \{\{4\}\}$ be the future set after executing line 3 (reading some value for y), and $G_4 = \{\{3_v\} \mid v \in \{0, 1, 2\}\}$ 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 2, 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 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.

The proof described above has actually been encoded and checked using our existing Isabelle/HOL development [7,8] by manually encoding the re-ordering in thread 2. We aim to develop full mechanisation support as future work.

7 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 [7,10, 11,15,17]. These assumptions incorrectly exclude the relaxed outcome of load buffering tests like Fig. 1, introducing unsoundness when applied to languages like C++: compiling Fig. 1 for an ARM processor produces code that does exhibit the relaxed behaviour. The logic of Lundberg et al. [20] correctly discards many of this spurious program ordering, but it does not handle concurrency. We provide an operational semantics of C++ that solves the thin air problem, where prior attempts use simplifying assumptions like those of prior logics [7,21].

Thin-Air-Free Semantics. Each of the concurrency definitions that solves the out-of-thin-air problem is remarkably complex [6,14,16,19,23,24], 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.

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 [13,14,16], and one is a separation logic built

above the Promising Semantics [26]. Unfortunately, the Promising Semantics allows unwanted out-of-thin-air behaviour, forbidden by MRD [23], and as a result may not support type safety [13].

8 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 Sect. 7). 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 [12] 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 far a 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, 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.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
2. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
3. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: Giacobazzi, R., Cousot, R. (eds.) POPL, pp. 235–248. ACM (2013)
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 55–66. ACM (2011)
5. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_12
6. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. Proc. ACM Program. Lang. **3**(POPL), 70:1–70:28 (2019). https://doi.org/10.1145/3290383

7. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries reasoning for C11 RAR. In: Hirschfeld, R., Pape, T. (eds.) ECOOP. LIPIcs, vol. 166, pp. 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.11
8. Dalvandi, S., Dongol, B., Doherty, S.: Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. CoRR abs/2004.02983 (2020). https://arxiv.org/abs/2004.02983
9. Dang, H., Jourdan, J., Kaiser, J., Dreyer, D.: Rustbelt meets relaxed memory. Proc. ACM Program. Lang. **4**(POPL), 34:1–34:29 (2020), https://doi.org/10.1145/3371102
10. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) PPoPP, pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702
11. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: ESOP, pp. 448–475 (2017)
12. Giroux, O.: ISO WG21 SG1 concurrency subgroup vote unanimously approved: OOTA is a major problem for C++, modular relaxed dependencies is the best path forward we have seen, and we wish to continue to pursue this direction (2019). https://github.com/cplusplus/papers/issues/554#issuecomment-551899923
13. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with preconditions: a simple model of relaxed memory. Proc. ACM Program. Lang. **4**(OOPSLA), 194:1–194:30 (2020). https://doi.org/10.1145/3428262
14. Jeffrey, A., Riely, J.: 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
15. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in iris. In: ECOOP, pp. 17:1–17:29 (2017)
16. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) POPL, pp. 175–189. ACM (2017). http://dl.acm.org/citation.cfm?id=3009850
17. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
18. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Cohen, A., Vechev, M.T. (eds.) PLDI, pp. 618–632. ACM (2017)
19. Lee, S., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C., Lahav, O., Vafeiadis, V.: Promising 2.0: global optimizations in relaxed memory concurrency. In: Donaldson, A.F., Torlak, E. (eds.) PLDI, pp. 362–376. ACM (2020). https://doi.org/10.1145/3385412.3386010
20. Lundberg, D., Guanciale, R., Lindner, A., Dam, M.: Hoare-style logic for unstructured programs. In: de Boer, F., Cerone, A. (eds.) Software Engineering and Formal Methods (2020)
21. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for C/C++11 concurrency. In: OOPSLA, pp. 111–128. ACM (2016)
22. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134

23. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: Müller, P. (ed.) Programming Languages and Systems, pp. 599–625. Springer International Publishing, Cham (2020)
24. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22, January, 2016, pp. 622–633. ACM (2016). https://doi.org/10.1145/2837614.2837616
25. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. **3**(POPL), 69:1–69:31 (2019). https://doi.org/10.1145/3290382
26. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 357–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_13
27. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_31