Verifying C11 Programs Operationally

Simon Doherty
Department of Computer Science
University of Sheffield
s.doherty@sheffield.ac.uk

Brijesh Dongol
Department of Computer Science
University of Surrey
b.dongol@surrey.ac.uk

Heike Wehrheim
Department of Computer Science
University of Paderborn
wehrheim@upb.de

John Derrick
Department of Computer Science
University of Sheffield
j.derrick@sheffield.ac.uk

Abstract

This paper develops an operational semantics for a release-acquire fragment of the C11 memory model with relaxed accesses. We show that the semantics is both sound and complete with respect to the axiomatic model. The semantics relies on a per-thread notion of observability, which allows one to reason about a weak memory C11 program in program order. On top of this, we develop a proof calculus for invariant-based reasoning, which we use to verify the release-acquire version of Peterson's mutual exclusion algorithm.

Keywords Operational semantics, C11, Verification, Soundness and Completeness

1 Introduction

Intensive research on the correctness of shared-memory concurrent programs over the last three decades has resulted in a variety of tools and techniques. However, the vast majority of these have been developed on the assumption of *sequential consistency* [21]. Programs running on modern hardware execute using *weak memory models* [2], requiring many of these techniques to be reworked.

This paper is focused on the C11 memory model, which has been the topic of several recent papers (e.g., [4–6, 8, 11, 14, 15, 17, 20, 22, 23]). Typically the C11 memory model is described using an axiomatic semantics [4–6, 20] via a two-step procedure. (1) Construct *candidate executions* of a program comprising low-level (e.g., read/write operations) in which reads may return an arbitrary value. (2) Apply a number of axioms over the memory model to rule out invalid candidate executions. Such axioms may state, for instance, that every read is validated by a write that has written the value read. Of particular interest are axioms that exclude certain cycles from arising. However precise, axiomatic definitions are unsuitable for program verification (in particular, those involving invariant-based reasoning), which requires one to consider the step-wise execution of a program. There has therefore been a substantial effort to develop an operational semantics: for weak memory models in general [14, 15, 18] and for C11 specifically [22, 23].

Our key goal in this paper is to develop an operational model that supports verification of weak memory C11 programs. Like many programming languages, C11 has several advanced features, e.g., speculation, that contributes to the complexity of the logics for reasoning about them. Some operational models (e.g., [23]) attempt to deal with the full complexity of the language and its behaviour. Other models focus on a well-behaved and well-understood fragment (e.g., [15, 18]). In order to support an intuitive verification method, we take the latter course. We do *not* handle some-forms of speculation (thin-air reads), release sequences, non-atomic accesses or sequentially consistent accesses. This leaves us with the so-called *RAR fragment* [5] of C11 (see Section 4.1), where $\mathsf{sb} \cup \mathsf{rf}$ is acyclic, and thus dependencies between operations are easier to manage. All read/write/update operations are either *relaxed* or *synchronised* via release-acquire annotations. Acyclicity of $\mathsf{sb} \cup \mathsf{rf}$ precludes behaviours allowed by hardware architectures such as Power [19]. Thus, to ensure programs proved correct by our logic remain sound, one must ensure adequate fencing of independent instructions during compilation (see [19] for details).

This paper comprises three main contributions. The first contribution is an operational semantics for the RAR fragment that we prove to be both *sound* and *complete* with respect to the axiomatic definition. Our semantics (like [15, 25]) allows each thread to have its own (per-thread) *observations* of memory. We build on the recently proposed *extended coherence order* [20] (which is the transitive closure of the *communication relation* in [3]). The extended coherence order describes the order of reads and writes to a variable (see Example 3.3), which in turn enables one to define how events may be introduced in a valid C11 execution without violating validity of the axioms.

We combine the extended coherence order with the causality relation of C11 (formalised by *happens-before*) to define the set of writes already *encountered* by each thread. This set is in turn used to define the writes observable by the thread (see Section 3.2). Our operational semantics naturally

PPoPP'19, February 16 – 20, 2019, Washington, DC, USA
2019.

builds on observability: reads are validated on-the-fly (as opposed to a post-hoc manner in the axiomatic semantics). Thus, each state constructed using the transition relations of our operational semantics is a valid C11 state (see Section 4.2). Moreover, we show that any candidate execution that is valid according to the axiomatic semantics can be generated by our operational semantics.

The second contribution is a verification technique that builds on the operational semantics to enable inductive reasoning over the program steps. One difficulty in using an operational semantics of weak-memory to support verification is the fact that the state spaces of such operational models are far more complicated than the state space that one would use for a verification over sequentially consistent memory, where the shared store can be represented using a simple mapping from variables to values. We address this issue by developing a notation that builds on conventional reasoning (over sequentially consistent memory). For example, we include assertions that ensure a thread will read a particular value in a C11 state and assertions that ensure happens-before order between writes to different variables. The former is analogous to equations on variables and their values in the conventional setting; the latter has no direct analogue in a sequentially consistent setting (the closest analogue is the use of auxiliary variables [24] to record whether certain operations have already occurred).

Our third contribution is the demonstration of the utility of our verification method by proving the mutual exclusion property of a C11 version of Peterson's algorithm [27].

2 Command Language

This section describes our command language and defines its *uninterpreted operational semantics*; namely, an operational semantics that generate the read, write or update *action* for each step of the corresponding command. These actions are in turn used to generate state transitions in Section 3, where the reads and writes are interpreted in a C11 state. Such a decoupled approach is inspired by the approach taken by Lahav et al. [17].

2.1 Syntax

The syntax of commands (for a single thread) is defined by the following grammar, where *Exp* and *Com* define expressions and commands, respectively. We assume that $\ominus$ is a unary operator (e.g., $\neg$), $\otimes$ is a binary operator (e.g., $\wedge$, $\vee$), B is an expression (of type *Exp*) that evaluates to a boolean, x is a variable (of type Var) and n is a value (of type Val).

$$
\begin{aligned}
\mathit{Exp} &::= \mathsf{Val} \mid \mathit{Exp}^{\mathsf{A}} \mid \ominus \mathit{Exp} \mid \mathit{Exp} \otimes \mathit{Exp} \\
\mathit{Com} &::= \textbf{skip} \mid x.\textbf{swap}(n)^{\mathsf{RA}} \mid x := \mathit{Exp} \mid x :=^{\mathsf{R}} \mathit{Exp} \mid \\
&\qquad \mathit{Com}; \mathit{Com} \mid \textbf{if } B \textbf{ then } \mathit{Com} \textbf{ else } \mathit{Com} \mid \\
&\qquad \textbf{while } B \textbf{ do } \mathit{Com}
\end{aligned}
$$

Algorithm 1 Peterson's algorithm with release-acquire

Init: $\mathit{flag}_1 = \mathit{false} \wedge \mathit{flag}_2 = \mathit{false} \wedge \mathit{turn} = 1$

	thread 1		thread 2
1:	**thread** 1	1:	**thread** 2
2:	$\mathit{flag}_1 := \mathit{true}$;	2:	$\mathit{flag}_2 := \mathit{true}$;
3:	$\mathit{turn}.\textbf{swap}(2)^{\mathsf{RA}}$;	3:	$\mathit{turn}.\textbf{swap}(1)^{\mathsf{RA}}$;
4:	**while** $(\mathit{flag}_2 = \mathit{true})^{\mathsf{A}} \wedge \mathit{turn} = 2$ **do skip**	4:	**while** $(\mathit{flag}_1 = \mathit{true})^{\mathsf{A}} \wedge \mathit{turn} = 1$ **do skip**
5:	Critical section ;	5:	Critical section ;
6:	$\mathit{flag}_1 :=^{\mathsf{R}} \mathit{false}$;	6:	$\mathit{flag}_2 :=^{\mathsf{R}} \mathit{false}$;

Commands have their standard meanings. The only exceptions are the synchronising annotations, *release* R, *acquire* A and *release-acquire* RA (which we describe in detail below), and the command **swap**, which generates a read-modify-write update event, atomically swapping the variable x with value n. Note that (for simplicity), we only present a release-acquire version of the **swap** operation, but leave in the RA annotation for emphasis. Furthermore, we assume unannotated accesses are *relaxed*, i.e., data races do not give rise to undefined behaviour; however it is straightforward to extend the semantics to incorporate non-atomic accesses (which potentially generate undefined behaviour).

Example 2.1 (Peterson's algorithm)**.** The running example for this paper will be the classic Peterson's mutual exclusion algorithm for two threads (see Algorithm 1) implemented using release-acquire annotations (this algorithm is taken from [27]). As with Peterson's original algorithm, variable flag_i is used to indicate whether thread i intends to enter its critical section and a shared variable *turn* is used to "give way" when both threads intend to enter their critical sections at the same time.

The difference in the C11 implementation is with the synchronisation annotations. The flag variable is set to *true* (line 2) using relaxed atomics (which does not induce any synchronisation), but is set to *false* (line 6) using a release annotation. The intention of the latter is to synchronise this write to *flag* with the read of *flag* at line 4 in the other thread. The value of *turn* is set using a **swap** command, which induces release-acquire synchronisation. Note that the read of *turn* within the guard of the busy wait loop (line 4) is relaxed. However, as we shall see, the algorithm still satisfies the mutual exclusion property.

2.2 Uninterpreted semantics

The uninterpreted operational semantics of commands is given by a relation $\longrightarrow \subseteq \mathit{Com} \times \mathsf{Act}_\tau \times \mathit{Com}$, where

$$
\mathsf{Act} = \bigcup_{x \in \mathsf{Var}; m, n \in \mathsf{Val}} \{rd(x,n), rd^{\mathsf{A}}(x,n), wr(x,n), wr^{\mathsf{R}}(x,n), upd^{\mathsf{RA}}(x,m,n)\}
$$

$\tau \notin \mathsf{Act}$ is a *silent action* and $\mathsf{Act}_\tau = \mathsf{Act} \cup \{\tau\}$. We write $C \xrightarrow{a} C'$ for $(C, a, C') \in \longrightarrow$.

$$\frac{x \in fv(E) \quad n \in \mathsf{Val} \quad a = rd(x,n)}{eval(E, a, E[n/x])} \qquad \frac{x \in fv(E) \quad n \in \mathsf{Val} \quad a = rd^{\mathsf{A}}(x,n)}{eval(E^{\mathsf{A}}, a, E[n/x])} \qquad \frac{fv(E) \neq \emptyset \quad eval(E, a, E')}{eval(\ominus E, a, \ominus E')} \qquad \frac{fv(E_1) \neq \emptyset \quad eval(E_1, a, E_1')}{eval(E_1 \otimes E_2, a, E_1' \otimes E_2)} \qquad \frac{fv(E_1) = \emptyset \quad eval(E_2, a, E_2')}{eval(E_1 \otimes E_2, a, E_1 \otimes E_2')}$$

Figure 1. Expression evaluation

An expression evaluation step is formalised by a relation $eval(E, a, E')$, where E, E' are expressions and a is a *read action* that is generated by the evaluation step (see Figure 1). We assume $fv(E)$ returns the set of free variables in E. Note that $eval(E, a, E')$ is only defined when $fv(E) \neq \emptyset$. Moreover, in the presence of a binary operator, expression evaluation is assumed to take place from left to right. The notation $E[n/x]$ stands for expression E with variable x replaced by value n.

The uninterpreted operational semantics for commands is given in Figure 2. Again, most of these rules are straightforward. We assume $[\![E]\!]$ denotes the value of (variable-free) expression E. An assignment $x := E$ generates a read action whenever $fv(E) \neq \emptyset$ and a write action whenever $fv(E) = \emptyset$. A **swap** command generates an update action, and guard evaluation either generates a read or a silent action.

Note that the uninterpreted operational semantics allows any value to be read. Thus, we have the following property.

Proposition 2.2. *For all $m, m' \in \mathsf{Val}$, if $C \xrightarrow{rd(x,m)} C'$, then $C \xrightarrow{rd(x,m')} C'$; if $C \xrightarrow{rd^{\mathsf{A}}(x,m)} C'$, then $C \xrightarrow{rd^{\mathsf{A}}(x,m')} C'$; and if $C \xrightarrow{upd^{\mathsf{RA}}(x,m,n)} C'$, then $C \xrightarrow{upd^{\mathsf{RA}}(x,m',n)} C'$.*

For simplicity, we assume concurrency at the top level only. We let T be the set of all threads and use function of type $Prog : T \to Com$ to model a program comprising multiple threads. The uninterpreted operational semantics of a program is given by a relation $\longrightarrow \subseteq Prog \times T \times \mathsf{Act}_\tau \times Prog$ (using overloading). As before, we write $P \xrightarrow{a}_t P'$ for $(P, t, a, P') \in \longrightarrow$. An evaluation step of a program P is given by the rule Prog (Figure 2), which relies on the uninterpreted operational semantics of a command to generate an action a and command C from the command $P(t)$. The program after taking a transition is the program P but with t mapped to the new command C.

Since threads execute independently in the uninterpreted semantics, all actions commute.

Proposition 2.3. *If $P \xrightarrow{a_1}_{t_1} P_1$ and $P_1 \xrightarrow{a_2}_{t_2} P'$ and $t_1 \neq t_2$, then there exists a P_2 such that $P \xrightarrow{a_2}_{t_2} P_2$ and $P_2 \xrightarrow{a_1}_{t_1} P'$.*

3 An Operational Semantics for RAR C11

We now extend the semantics from Section 2 and interpret read, write and update actions in the C11 memory model. We develop an operational semantics that takes inspiration from the axiomatic descriptions [5, 6, 20]. In Section 4.2, we show that the operational model is in fact equivalent to a reformulation (inspired by [20]) of the *RAR fragment* of the RC11 semantics [5].

We formalise C11 states in Section 3.1 and define an operational *event semantics* based on observability (Section 3.2). This event semantics in turn gives rise to an *interpreted semantics* (Section 3.3).

3.1 C11 States and Basic Orders

The formalisation in this section follows the existing literature on axiomatic C11 semantics [6, 20]. First we give some preliminary definitions.

Notation. For an action $a \in \mathsf{Act}$, we let $var(a) \in \mathsf{Var}$ be the variable read (or written to), $rdval(a) \in \mathsf{Val}$ be the value read and $wrval(a) \in \mathsf{Val}$ be the value written. We extend actions to *events* of type $Evt = G \times \mathsf{Act}_\tau \times T$, where G is the set of *tags* used to uniquely identify events in an execution. For an event (g, a, t), where g is a tag, a is an action, and t is a thread identifier, we define $tag(e) = g$, $act(e) = a$, $tid(e) = t$, and (using lifting) $var(e) = var(act(e))$, $wrval(e) = wrval(act(e))$, $rdval(e) = rdval(act(e))$. For a relation $R \subseteq Evt \times Evt$, we let $R_{|t}$ and $R_{|v}$ be the restriction of R to events of thread t, and variable v, respectively.

We let U denote the RMW update events, and distinguish the sets $\mathsf{Wr_R} \supseteq \mathsf{U}$ (write release), $\mathsf{Rd_A} \supseteq \mathsf{U}$ (read acquire), $\mathsf{Wr_X}$ (write relaxed) and $\mathsf{Rd_X}$ (read relaxed). Finally, we define $\mathsf{Rd} = \mathsf{Rd_A} \cup \mathsf{Rd_X}$ (all reads) and $\mathsf{Wr} = \mathsf{Wr_R} \cup \mathsf{Wr_X}$ (all writes).

Definition 3.1. A *C11 state* is a triple $\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ comprising a set of events D paired with a *sequenced-before* relation $\mathsf{sb} \subseteq D \times D$, a *reads-from* relation $\mathsf{rf} \subseteq \mathsf{Wr} \times \mathsf{Rd}$ and a *modification order* $\mathsf{mo} \subseteq \mathsf{Wr} \times \mathsf{Wr}$.

We let Σ denote the set of all C11 states. The three relations in a C11 state $((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ reflect different relationships between operations. The sequenced-before relation sb records the program order within one thread; $\mathsf{sb}_{|t}$ is a strict total order for each thread t. The reads-from relation rf provides the justification for the values being read: every read must have a corresponding action that writes the value being read. The modification order mo describes an ordering of the writes on variables; $\mathsf{mo}_{|v}$ is a strict total order for each variable $v \in \mathsf{Var}$.

Weak memory models are often defined in terms of a *happens-before* order (denoted hb), which formalises a notion of causality. In C11, an event occuring in a thread before another event in the same thread induces *sequenced-before* order (denoted sb), which in turn induces happens before order. Moreover, reads-from edges induce happens-before order when the corresponding actions in the edge are *synchronising actions* (i.e., a release and an acquire). This is formalised by an additional *synchronises-with* relation (denoted

$$\frac{eval(E, a, E')}{x := E \xrightarrow{a} x := E'} \qquad \frac{fv(E) = \emptyset \quad a = wr(x, [\![E]\!])}{x := E \xrightarrow{a} \mathbf{skip}} \qquad \frac{fv(E) = \emptyset \quad a = wr^{\mathsf{R}}(x, [\![E]\!])}{x :=^{\mathsf{R}} E \xrightarrow{a} \mathbf{skip}} \qquad \frac{}{\mathbf{skip}; C \xrightarrow{\tau} C} \qquad \frac{C_1 \xrightarrow{a} C_1'}{C_1; C_2 \xrightarrow{a} C_1'; C_2}$$

$$\frac{m, n \in \mathsf{Val} \quad a = upd^{\mathsf{RA}}(x, m, n)}{x.\mathbf{swap}(n)^{\mathsf{RA}} \xrightarrow{a} \mathbf{skip}} \qquad \frac{eval(B, a, B')}{\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \xrightarrow{a} \mathbf{if}\ B'\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2} \qquad \frac{[\![B]\!] = true}{\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \xrightarrow{\tau} C_1} \qquad \frac{[\![B]\!] = false}{\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \xrightarrow{\tau} C_2}$$

$$\frac{eval(B, a, B')}{\mathbf{while}\ B\ \mathbf{do}\ C \xrightarrow{a} \mathbf{while}\ B'\ \mathbf{do}\ C} \qquad \frac{[\![B]\!] = true}{\mathbf{while}\ B\ \mathbf{do}\ C \xrightarrow{\tau} C; \mathbf{while}\ B\ \mathbf{do}\ C} \qquad \frac{[\![B]\!] = false}{\mathbf{while}\ B\ \mathbf{do}\ C \xrightarrow{\tau} \mathbf{skip}}$$

$$\textsc{Prog}\ \frac{P(t) \xrightarrow{a} C}{P \xrightarrow{a}_t P[t \mapsto C]}$$

Figure 2. Uninterpreted operational semantics of commands and programs

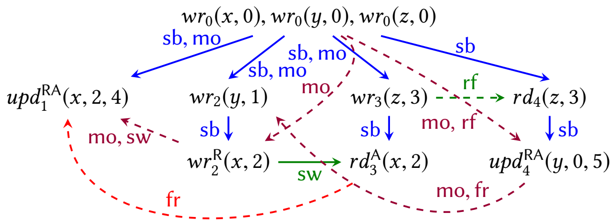
sw). Formally, we define

$$\mathsf{sw} = \mathsf{rf} \cap (\mathsf{Wr_R} \times \mathsf{Rd_A}) \qquad\qquad \mathsf{hb} = (\mathsf{sb} \cup \mathsf{sw})^+ .$$

As is standard in the literature, we assume all variables are initialised by a special thread $0 \in T$. Define the set of *initialising writes* to be $\mathsf{IWr} = \{w \in \mathsf{Wr} \mid tid(w) = 0\}$. The initial states of our operational model are those of the form $\sigma_0 = ((I, \emptyset), \emptyset, \emptyset)$ where $I \subseteq \mathsf{IWr}$, and for each variable x, there is exactly one write $w \in I$ such that $var(w) = x$. For a state $\sigma = ((D, _), _, _)$, let $I_\sigma = D \cap \mathsf{IWr}$.

The relation $\mathsf{fr} = (\mathsf{rf}^{-1}; \mathsf{mo}) \backslash Id$ (where ; is relational composition) is the "from-read" relation[1] that relates each read to all writes that are mo-after the write the read has read from. We must subtract Id (identity) edges from $\mathsf{rf}^{-1}; \mathsf{mo}$ to cope with update events, which have the potential to induce reflexivity in fr [5, 20].

Example 3.2. An example C11 state is given below, where threads 1-4 have executed some actions. Since the actions are unique, we elide the tags from each event, and we identify the thread id with the action itself, e.g., $wr_1(y, 1)$ is the action $wr(y, 1)$ executed by thread 1.

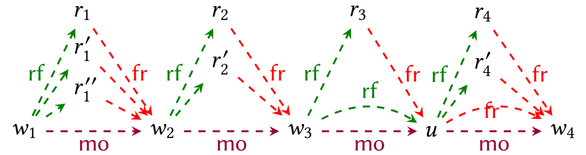


The initialising writes are sb-before all thread actions, but are not ordered amongst themselves. Relation sb also describes the order for each thread. Relation mo describes the order of modifications for each variable. The unsynchronised read $rd_4(z, 3)$ is justified by the rf from $wr_3(z, 3)$, whereas the synchronised read $rd_3^{\mathsf{A}}(x, 2)$ is justified by the sw from $wr_2^{\mathsf{R}}(x, 2)$ and fixed before $upd_1^{\mathsf{RA}}(x, 2, 4)$ via the fr relation. Update events are related by both mo and rf to the immediately preceding write, and possibly related to later writes/updates by mo and fr. If the write being read is releasing, then an update induces an sw (e.g., see $upd_1^{\mathsf{RA}}(x, 2, 4)$). □

In addition, our semantics uses the *extended coherence order*[2] [20], denoted eco, which is an order that fixes the order of reads and writes to each variable (see Example 3.3 below). Formally we define:

$$\mathsf{eco} = (\mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf})^+$$

Example 3.3. For executions of a C11 program, eco over a single variable takes the following form, where $w_1, \ldots, w_5$ are writes and r_1, r_1' etc are reads and u is an update.



Reads r_1, r_1' and r_1'' read from the write w_1, inducing from-read edges to w_2 (the write that immediately follows w_1 in mo). The update u induces an rf from w_3 (the write event immediately before u in mo) and an fr to w_4 (the write event immediately after u in mo). □

3.2 Event Semantics and Observability

Recalling that Σ denotes the set of all possible C11 states and Wr is the set of all writes (including updates), each step of the event semantics is formalised by the transition relation $\rightsquigarrow_{RA} \subseteq \Sigma \times \mathsf{Wr}_\perp \times Evt \times \Sigma$ (see Figure 3), where we have $\mathsf{Wr}_\perp = \mathsf{Wr} \cup \{\perp\}$ and $\perp \notin \mathsf{Wr}$. Again, we write $\sigma \overset{w,e}{\rightsquigarrow}_{RA} \sigma'$ for $(\sigma, w, e, \sigma') \in \rightsquigarrow_{RA}$.

For each rule $\sigma \overset{w,e}{\rightsquigarrow}_{RA} \sigma'$, w is the *write being observed* by the event e. Strictly speaking, the event semantics could be defined without the w. However, making this observed write explicit is useful for the verification (Section 5).

[1] fr is also referred to as "reads-before" [20].

[2] The non-transitive version of this order is commonly referred to as the com order [3].

$$\textsc{Read}\ \frac{\begin{array}{c} a \in \{rd(x,n), rd^{\mathsf{A}}(x,n)\} \quad var(w) = x \quad wrval(w) = n \\ w \in OW_\sigma(t) \quad \mathsf{rf}' = \mathsf{rf} \cup \{(w,e)\} \quad \mathsf{mo}' = \mathsf{mo} \end{array}}{((D,\mathsf{sb}),\mathsf{rf},\mathsf{mo}) \overset{w,e}{\rightsquigarrow}_{RA} ((D,\mathsf{sb})+e, \mathsf{rf}', \mathsf{mo}')}$$

$$\textsc{Write}\ \frac{\begin{array}{c} a \in \{wr(x,n), wr^{\mathsf{R}}(x,n)\} \quad w \in OW_\sigma(t) \backslash CW_\sigma \\ var(w) = x \quad \mathsf{rf}' = \mathsf{rf} \quad \mathsf{mo}' = \mathsf{mo}[w,e] \end{array}}{((D,\mathsf{sb}),\mathsf{rf},\mathsf{mo}) \overset{w,e}{\rightsquigarrow}_{RA} ((D,\mathsf{sb})+e, \mathsf{rf}', \mathsf{mo}')}$$

$$\textsc{RMW}\ \frac{\begin{array}{c} a = upd^{\mathsf{RA}}(x,m,n) \quad w \in OW_\sigma(t) \backslash CW_\sigma \quad var(w) = x \\ wrval(w) = m \quad \mathsf{rf}' = \mathsf{rf} \cup \{(w,e)\} \quad \mathsf{mo}' = \mathsf{mo}[w,e] \end{array}}{((D,\mathsf{sb}),\mathsf{rf},\mathsf{mo}) \overset{w,e}{\rightsquigarrow}_{RA} ((D,\mathsf{sb})+e, \mathsf{rf}', \mathsf{mo}')}$$

Figure 3. Event semantics assuming $\sigma = ((D,\mathsf{sb}),\mathsf{rf},\mathsf{mo})$, $e = (g,a,t)$ and $g \notin tags(D)$

We now describe each of the rules in Figure 3. Executing each event e updates (D,sb) to:

$$(D,\mathsf{sb}) + e = \begin{pmatrix} D \cup \{e\}, \\ \mathsf{sb} \cup (\{e' \in D \mid tid(e') \in \{tid(e), 0\}\} \times \{e\}) \end{pmatrix}$$

Thus, the initial writes are sb-prior to every non-initialising event. Relations rf and mo are updated according to the write events in D that are observable to the thread executing the given event. To this end, we must distinguish three sets of writes: *encountered writes* and *observable writes*, which are specific to each thread, and *covered writes*, which are the set of writes that are immediately followed, in reads-from order, by an update event.

The set of *encountered writes* are the writes that thread t is aware of (either directly or indirectly) in state $\sigma = ((D,\mathsf{sb}),\mathsf{rf},\mathsf{mo})$, and are given by:

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

where $R^?$ is the reflexive closure of relation R. Thus, for each $w \in EW_\sigma(t)$, there must exist an event e of thread t such that w is either eco- or hb- or eco; hb-prior to e. Note that $EW_\sigma(t) = \emptyset$ if the thread t has not executed any actions; as soon as the thread executes its first action, we have $I \subseteq EW_\sigma(t)$.

From these, we determine the *observable writes*, which are the writes that thread t can observe in its next read. These are defined as:

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

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

Finally, to guarantee *atomicity* of the update events, there cannot be any write operations (in modification order) between the write that an update reads from and the write of the update itself. We therefore define the set of *covered writes* as follows:

$$CW_\sigma = \{w \in \mathsf{Wr} \cap D \mid \exists u \in \mathsf{U}.\ (w,u) \in \mathsf{rf}\}$$

Example 3.4. Consider the C11 state σ in Example 3.2. Given that $I = \{wr_0(x,0), wr_0(y,0), wr_0(z,0)\}$ is the set of initialising writes, the encountered writes for each thread are as follows:

$$EW_\sigma(1) = I \cup \{wr_2^{\mathsf{R}}(x,2), upd_1^{\mathsf{RA}}(x,2,4)\}$$
$$EW_\sigma(2) = I \cup \{wr_2(y,1), wr_2^{\mathsf{R}}(x,2), upd_4^{\mathsf{RA}}(y,0,5)\}$$
$$EW_\sigma(3) = I \cup \{wr_2(y,1), wr_2^{\mathsf{R}}(x,2), wr_3(z,3), upd_4^{\mathsf{RA}}(y,0,5)\}$$
$$EW_\sigma(4) = I \cup \{wr_3(z,3), upd_4^{\mathsf{RA}}(y,0,5)\}$$

The observable writes are hence

$$\begin{aligned} OW_\sigma(1) &= \{wr_0(y,0), wr_0(z,0), wr_2(y,1), wr_3(z,3), \\ &\qquad upd_1^{\mathsf{RA}}(x,2,4), upd_4^{\mathsf{RA}}(y,0,5)\} \\ OW_\sigma(2) &= \{wr_0(z,0), wr_2(y,1), wr_3(z,3), upd_1^{\mathsf{RA}}(x,2,4)\} \\ OW_\sigma(3) &= \{wr_2(y,1), wr_2^{\mathsf{R}}(x,2), wr_3(z,3), upd_1^{\mathsf{RA}}(x,2,4)\} \\ OW_\sigma(4) &= \{wr_0(x,0), wr_2(y,1), wr_2^{\mathsf{R}}(x,2), wr_3(z,3), \\ &\qquad upd_1^{\mathsf{RA}}(x,2,4), upd_4^{\mathsf{RA}}(y,0,5)\} \end{aligned}$$

The covered writes are $CW_\sigma = \{wr_0(y,0), wr_2^{\mathsf{R}}(x,2)\}$. □

Observable writes are used to resolve the read events in each thread. Namely, a thread t can read from any write event in $OW_\sigma(t)$. This is reflected in the Read rule, where the rf component is updated to record an rf from some observable write w to the read event e, provided w writes to the variable that e reads and the value read matches the value written.

To explain the write and update semantics, we require some more formal machinery. The observable and covered writes together determine the allowable updates to the mo relation after executing a write event. Unlike SC, a write event to variable x is not simply appended to the end of $\mathsf{mo}_{|x}$. Instead we allow a thread t that performs a write e (or update) to x to insert e after any observable write w in $\mathsf{mo}_{|x}$ that is not a covered write. This condition is sufficient to ensure no cyclic dependencies arise as a result of performing the write.

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 write event e directly after a write w in mo is given by

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

The rules Write and RMW update mo in the same way. For the write event e executed by thread t, they pick some w that writes to the same variable as e, is observable to t and not covered by an update event, then insert e immediately after w in mo.

Example 3.5. For the execution in Example 3.4, no thread may introduce a write between $wr_1^{\mathsf{R}}(x,2)$ and $upd_1^{\mathsf{RA}}(x,4,5)$, or between $wr_0(y,0)$ and $upd_5^{\mathsf{RA}}(y,0,7)$.

3.3 Interpreted Semantics

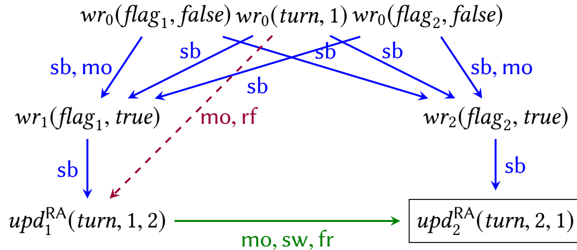
We now combine the event semantics with the uninterpreted semantics to give an interpreted semantics for the language in Section 2 overall. We give two generic rules that allows different memory models to be plugged in for the event semantics.

To this end, we define a *configuration* to be a pair (P, σ), consisting of a program P and a state σ of the memory model. The command part of a configuration triggers events that are agnostic to values. However, the memory model will only allow certain values in read events. This idea is captured by the following two rules combining the uninterpreted program semantics (i.e., rule Prog) from Section 2.2 and an event semantics in some memory model M:

$$\frac{P \xrightarrow{\tau}_t P'}{(P, \sigma) \overset{\bot, \tau}{\Longrightarrow}_M (P', \sigma)} \qquad \frac{P \xrightarrow{a}_t P' \quad \sigma \overset{w, e}{\rightsquigarrow}_M \sigma' \quad act(e) = a \quad tid(e) = t}{(P, \sigma) \overset{w, e}{\Longrightarrow}_M (P', \sigma')}$$

The first rule describes a τ-step and does not change the state. The second states that a thread can execute action a in the current state σ only if the event semantics of the memory model in consideration permits it.

Example 3.6. Consider the state of Peterson's algorithm (Algorithm 1) in RA C11 that results when thread 1 has reached the guard at line 4, and thread 2 is about to execute line 3. Execution of this step introduces the boxed event $upd_2^{RA}(turn, 2, 1)$. (We use the box for emphasis; it does not carry any special semantic meaning.)



In the state *without* the boxed event, thread 2 can read from $wr_0(turn, 1)$ via a read event, but it cannot do so via an update because $wr_0(turn, 1)$ is covered by the existing update $upd_1^{RA}(turn, 1, 2)$. Hence the update of thread 1 (when the event in the box occurs) updates *turn* from 2 to 1, which creates mo, sw and fr edges from $upd_1^{RA}(turn, 1, 2)$.

Now consider a continuation from the state with the boxed event, where the threads read the values in their respective guards. Thread 2 has encountered $wr_1(flag_1, true)$, and hence, is no longer able to observe $wr_0(flag_1, false)$. Similarly, since thread 2 has encountered $upd_2^{RA}(turn, 2, 1)$ it is no longer able to observe $wr_0(turn, 1)$ or $upd_1^{RA}(turn, 1, 2)$. We therefore conclude that thread 2's guard will evaluate to true, causing it to spin at line 4. In contrast, thread 1 can read from either $wr_0(flag_2, false)$ or $wr_2(flag_2, true)$ since it has not yet encountered the event $wr_2(flag_2, true)$. Similarly, since it has not yet encountered $upd_2^{RA}(turn, 2, 1)$, it can read from both $upd_1^{RA}(turn, 1, 2)$ and $upd_2^{RA}(turn, 2, 1)$. Thread 1 therefore could spin at line 4 or exit the busy loop. Note that once it has read a new value for $flag_2$ or *turn*, the previous value (in mo-order) can no longer be read.

This example demonstrates how the basic synchronisation principle of Peterson's algorithm is guaranteed by the release-acquire annotations. Namely, (1) the updates on *turn* are totally ordered via hb due to the release-acquire annotation on statement **swap**, and (2) the thread that is first to execute **swap**, may miss to see that the other thread has set its flag.

4 Validation of Operational Semantics

We now justify our operational semantics by showing it to be sound and complete with respect to an existing axiomatic version of the C11 memory model. There are several versions of the C11 axiomatic semantics that might be regarded as both standard and complete [5, 6, 20]. Our semantics deals only with the release, acquire and relaxed annotations on operations. We call this the *RAR* fragment of C11. The standard C11 semantics also specifies the behaviour of operations carrying *sequentially consistent* and *non-atomic* annotations. We ignore these annotations here. Our semantics closely resembles the RAR fragment of [5] and [20]. Like [5], we use the convention that update operations are represented as a single event, rather than a read/write pair. Like [20], we adopt the constraint that sb ∪ rf is acyclic, and make use of the extended coherence order.[3] The axiomatic semantics is given in Section 4.1. Soundness and completeness of the memory model is presented in Section 4.2.

4.1 Background: RAR Fragment of RC11

Axiomatic semantics start with *pre-executions*, which are candidates for valid C11 executions. A number of axioms are used to define which of these candidates are considered real executions. Pre-executions only contain a set of events and program order (as represented by the sequenced-before relation). We call such a pair (D, sb) a *pre-execution state*. New events can be added to pre-execution states using the + operator in the same way as in Figure 3. Thus, if $(D, \mathsf{sb}) \overset{\bot, e}{\rightsquigarrow}_{PE} (D', \mathsf{sb}')$ then $(D', \mathsf{sb}') = (D, \mathsf{sb}) + e$. These pre-execution steps are combined with the steps of a program as before, i.e., using the rules in Section 3.3, i.e., we replace $\overset{w, e}{\rightsquigarrow}_M$ by $\overset{\bot, e}{\rightsquigarrow}_{PE}$. Since the first event in $\overset{\bot, e}{\rightsquigarrow}_{PE}$ is always $\bot$ (no write events are observed), we write $\overset{e}{\rightsquigarrow}_{PE}$ for $\overset{\bot, e}{\rightsquigarrow}_{PE}$.

Proposition 4.1. *If $\gamma \overset{e_1}{\rightsquigarrow}_{PE} \gamma_1$ and $\gamma_1 \overset{e_2}{\rightsquigarrow}_{PE} \gamma'$ and $tid(e_1) \neq tid(e_2)$, then there exists a γ_2 s.t. $\gamma \overset{e_2}{\rightsquigarrow}_{PE} \gamma_2$ and $\gamma_2 \overset{e_1}{\rightsquigarrow}_{PE} \gamma'$.*

[3] In the full version of this paper [7], we prove that our axiomatic model is equivalent to a variant of the RAR fragment of [5]. This proof is supported by a mechanisation in Memalloy [26], which shows our models is equivalent to the RAR fragment for models upto size 7. The associated `.cat` files have been submitted as supplementary material.

Once a candidate pre-execution (D, sb) is computed, it is augmented with the relations rf and mo.

Definition 4.2. A C11 execution $((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ is *valid* iff each of the following axioms hold:

SB-Total. Sequenced-before is a total order over the events of each (non-initialising) thread and orders all initialising writes before all other events. Formally, for any $e, e' \in D$,

$$((e, e') \in \mathsf{sb} \Rightarrow tid(e) = 0 \lor tid(e) = tid(e')) \land$$
$$(tid(e) = 0 \land tid(e') \neq 0 \Rightarrow (e, e') \in \mathsf{sb}) \land$$
$$(tid(e) \neq 0 \land tid(e) = tid(e') \land e \neq e' \Rightarrow (e, e') \in \mathsf{sb} \cup \mathsf{sb}^{-1}) .$$

MO-Valid. Modification order is a strict order on $\mathsf{Wr} \cap D$ consisting of a disjoint union of relations $\{\mathsf{mo}_{|x}\}_{x \in \mathsf{Var}}$ which are themselves total. That is, for any $w, w' \in \mathsf{Wr} \cap D$,

$$((w, w') \in \mathsf{mo} \Rightarrow var(e) = var(e')) \land$$
$$(tid(w) = 0 \land tid(w') \neq 0 \land var(w) = var(w') \Rightarrow (w, w') \in \mathsf{mo}) \land$$
$$(tid(w) \neq 0 \land tid(w') \neq 0 \land var(w) = var(w') \land w \neq w' \Rightarrow (w, w') \in \mathsf{mo} \cup \mathsf{mo}^{-1}) .$$

RF-Complete. Each read matches exactly one write in the execution, i.e., for every $e \in \mathsf{Rd} \cap D$ there is exactly one $w \in \mathsf{Wr} \cap D$ such that $(w, e) \in \mathsf{rf}$, and for every $(e, e') \in \mathsf{rf}$,

$$e \in \mathsf{Wr} \land e' \in \mathsf{Rd} \land var(e) = var(e') \land wrval(e) = rdval(e').$$

No-Thin-Air. The relation $\mathsf{sb} \cup \mathsf{rf}$ is acyclic.

Coherence. The relations $\mathsf{hb}; \mathsf{eco}^?$ and eco are irreflexive.

Definition 4.3. A pre-execution state γ is *justifiable* iff there exist relations rf and mo such that $(\gamma, \mathsf{rf}, \mathsf{mo})$ is valid.

4.2 Soundness and Completeness

Having defined a new operational semantics for C11, the next step is now the comparison with the existing axiomatic semantics. In the following, we prove the before given operational and axiomatic semantics to be equal. We start by showing that the executions of the operational semantics are all consistent.

Theorem 4.4. *Let $\sigma = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ be a C11 state reachable from σ_0 using relation $\rightsquigarrow_{RA}$. Then σ satisfies* SB-Total, MO-Valid, RF-Complete, No-Thin-Air *and* Coherence.

We next show that all consistent executions of a program are reachable in our operational semantics. We do so in two steps. First, we consider the runs of a program on the memory model. Since the axiomatic semantics in its pre-execution allows for reads before the appropriate writes, not every sequence of events possible for pre-executions is also possible in the operational semantics.

Example 4.5. Consider the following simple program with two threads.

thread 1: $z := x$ **thread** 2: $x := 5$

We have mapping $P_0 = \{1 \mapsto z := x, 2 \mapsto x := 5\}$. The following pre-execution is possible:

$$\delta_0 \xRightarrow{rd_1(x,5)}_{PE} \delta_1 \xRightarrow{wr_1(z,5)}_{PE} \delta_2 \xRightarrow{wr_2(x,5)}_{PE} \delta_3$$

where $\delta_i = (P_i, \gamma_i)$. The pre-execution state δ_3 can be justified using the following C11 state

$$wr_2(x,5) \xrightarrow{\mathsf{rf}} rd_1(x,5) \xrightarrow{\mathsf{sb}} wr_1(z,5)$$

The sequence of events is however not possible in the $\rightsquigarrow_{RA}$ semantics since we cannot have a read without the prior write that it reads from, and hence the first transition cannot be emulated. Still, the operational semantics can reach the same final C11 state by executing

$$(P_0, \sigma_0) \xRightarrow{wr_2(x,5)}_{RA} (P'_1, \sigma'_1) \xRightarrow{rd_1(x,5)}_{RA} (P'_2, \sigma'_2) \xRightarrow{wr_1(z,5)}_{RA} (P_3, \sigma_3)$$

which is also a sequence of steps in $\Longrightarrow_{PE}$. □

The "reordering" of events described in Example 4.5 is always possible: for every sequence of steps of pre-executions, we can find a corresponding permutation of these steps in which reads are ordered after their writes (and the program order within threads is preserved).

Putting together Propositions 2.3 and 4.1, we have the following result.

Proposition 4.6. *If $(P, \gamma) \xRightarrow{e_1}_{PE} (P_1, \gamma_1)$ and $(P_1, \gamma_1) \xRightarrow{e_2}_{PE} (P', \gamma')$ where $tid(e_1) \neq tid(e_2)$, then there exists a program P_2 and a pre-execution state γ_2 such that $(P, \gamma) \xRightarrow{e_2}_{PE} (P_2, \gamma_2)$ and $(P_2, \gamma_2) \xRightarrow{e_1}_{PE} (P', \gamma')$.*

This proposition is used to prove a permutation theorem for independent elements. We say that sequence $e_1 e_2 \dots e_n$ is a *linearization* of a strict order $\prec$ iff $\mathbf{dom}(\prec) \cup \mathbf{ran}(\prec) = \{e_1, e_2, \dots, e_n\}$ and for any e_i, e_j, we have $e_i \prec e_j \Rightarrow i < j$.

Lemma 4.7. *Let $Q = (P_0, \gamma_0) \xRightarrow{e_1}_{PE} (P_1, \gamma_1) \xRightarrow{e_2}_{PE} \dots \xRightarrow{e_k}_{PE} (P_k, \gamma_k)$ and $\gamma_k = (D_k, \mathsf{sb}_k)$. Then for every linearization $f_1, \dots, f_k$ of sb_k, there exist programs $P'_1, \dots, P'_{n-1}$ and pre-execution states $\gamma'_1, \dots, \gamma'_{n-1}$ such that*

$$(P_0, \gamma_0) \xRightarrow{f_1}_{PE} (P'_1, \gamma'_1) \xRightarrow{f_2}_{PE} \dots \xRightarrow{f_k}_{PE} (P_k, \gamma_k) .$$

We now show that for every justifiable pre-execution there is an execution of the C11 semantics that ends in the C11 state justifying the pre-execution. The theorem uses a notion that restricts pre-executions and C11 executions to a set of events. For a set of events $E \subseteq D$, we define:

$$(D, \mathsf{sb})_{\downarrow E} = (E, \mathsf{sb} \cap (E \times E))$$
$$(\gamma, \mathsf{rf}, \mathsf{mo})_{\downarrow E} = (\gamma_{\downarrow E}, \mathsf{rf} \cap (E \times E), \mathsf{mo} \cap (E \times E))$$

In the completeness proof, we assume that the given pre-execution sequence $(P_0, \gamma_0) \xRightarrow{e_1}_{PE} (P_1, \gamma_1) \xRightarrow{e_2}_{PE} \dots \xRightarrow{e_k}_{PE}$

(P_k, γ_k) has been reordered such that $e_1 \dots e_k$ is a linearization of $\mathsf{sb}_k \cup \mathsf{rf}_k$, where rf_k is the reads-from relation used in the justification of γ_k. Such a linearization is possible since $\mathsf{sb}_k \cup \mathsf{rf}_k$ is acyclic (axiom No-Thin-Air).

Theorem 4.8. *Suppose* $(P_0, \gamma_0) \overset{e_1}{\Longrightarrow}_{PE} (P_1, \gamma_1) \overset{e_2}{\Longrightarrow}_{PE} \dots \overset{e_k}{\Longrightarrow}_{PE} (P_k, \gamma_k)$ *such that* $\gamma_k = (D_k, \mathsf{sb}_k)$ *is justifiable with justification* $\sigma_k = (\gamma_k, \mathsf{rf}_k, \mathsf{mo}_k)$ *and* $e_1, \dots, e_k$ *is a linearization of* $\mathsf{sb}_k \cup \mathsf{rf}_k$. *Then* $(P_0, \sigma_0) \overset{e_1}{\Longrightarrow}_{RA} (P_1, \sigma_1) \overset{e_2}{\Longrightarrow}_{RA} \dots \overset{e_k}{\Longrightarrow}_{RA} (P_k, \sigma_k)$, *where* $\sigma_i = (\gamma_k, \mathsf{rf}_k, \mathsf{mo}_k)_{\downarrow\{e_1,\dots,e_i\}}$, $0 < i < k$.

5 Verification

We now describe our verification method (Section 5.1), building on the operational semantics. In Section 5.2, we apply it to our case study, Peterson's mutual exclusion algorithm.

5.1 Verification Method

Our verification method is built around two kinds of assertions for describing states of the operational semantics. The first kind, *determinate-value* assertions, are used to describe the values that a read operation might return. As such, these assertions are analogous to equations that specify the values of variables in a conventional (i.e., sequentially consistent) setting in which the state of an algorithm can be represented as a store that maps variables to values. The second kind of assertion, *variable-ordering* assertions, has no direct analogue in the conventional setting. Variable-ordering assertions provide a way to describe how information about a variable propagates between threads.

Determinate-values. In the following, we assume that $\sigma = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ is a valid C11 state. We let $\sigma.last(x)$ be the write or update to x in D that is not succeeded by another write or update in $\mathsf{mo}_{|x}$. Note that $\sigma.last(x)$ is well-defined in any valid state σ. When X is a set of operations and x is a variable, $X_{|x} = \{e \in X \mid var(e) = x\}$. For the determinate value assertions, consider some thread t and variable x. In some states of the operational semantics, there is exactly *one* write that t can read-from when reading x. This is true precisely when $OW_\sigma(t)_{|x} = \{\sigma.last(x)\}$ (recall that $\sigma.last(x)$ is never covered, and so $\sigma.last(x)$ can always be observed in a transition). Under such a condition, the value returned by a read of x in thread t must be $wrval(\sigma.last(x))$. This ultimately provides us with a weak memory analogue of an equation asserting that a given variable has a given value in a conventional sequentially consistent setting.

Definition 5.1. Let t be a thread, σ a state and v a value. The *determinate-value* assertion $x \overset{\sigma}{=}_t v$ holds iff

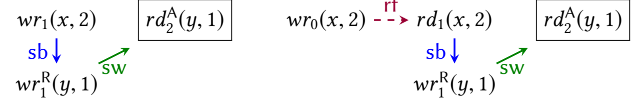
$$OW_\sigma(t)_{|x} = \{\sigma.last(x)\} \quad (1)$$

$$v = wrval(\sigma.last(x)) \quad (2)$$

$$\sigma.last(x) \in I_\sigma \cup \{e \mid \exists e'.\ tid(e) = t \wedge (e, e') \in \sigma.\mathsf{hb}^?\} \quad (3)$$

Condition (3) states that $\sigma.last(x)$ is either an operation of the initialising thread, an operation of t, or happens-before an operation of t.

Example 5.2. To illustrate the determinate-value assertion, consider the two states below. In each case, assume there are writes (not shown) to variable x that are mo-prior to the write to x. Also assume that each write is the last write in mo order.



For the state on the left, after the boxed operation, thread 2 satisfies $x \overset{\sigma}{=}_2 2$, but for the state on the right, thread 2 does not. In each case, the only write to x that thread 2 can observe is the illustrated write to x, but thread 2 satisfies a corresponding determinate value assertion only on the left state. This is because on the left we have $(wr_1(x,2), rd_2(x,2)) \in \mathsf{hb}$, but the unsychronised rf edge on the right means that there is no analogous hb edge.

In our verification, determinate-value assertions support clean interaction with variable-ordering assertions, which we describe shortly. Note that because our operational model prevents update operations reading from covered writes (see Section 3), i.e., are more restricted than read operations, an update operation on a variable x may only be able to read from the last write to x even if $x \overset{\sigma}{=} v$ is false for all v. Below, we show how to handle important instances of this situation.

The next two lemmas are immediate from the definition of $\overset{\sigma}{=}_t$. Lemma 5.3 below ensures that the value returned by a reading transition using the semantics in Figure 3 is consistent with the determinate-value assertion. Lemma 5.4 ensures that when a determinate-value assertion holds for two threads reading from the same variable, they return the same values for the variable.

Lemma 5.3 (Determinate-Value Read)**.** *For any* Read *or* RMW *transition* $(P, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (P', \sigma')$, *if* $var(e) \overset{\sigma}{=}_{tid(e)} v$, *then* $rdval(e) = v$.

Lemma 5.4 (Determinate-Value Agreement)**.** *For threads* t, t', *variable* x, *and values* v, v', *if* $x \overset{\sigma}{=}_t v$ *and* $x \overset{\sigma}{=}_{t'} v'$ *then* $v = v'$, *and thus* t *and* t' *agree on the value of* x.

Determinate-value assertions differ from their conventional counterparts in that they are *relative to a particular thread.* It is almost definitive of weak-memory systems that distinct threads can have different views of the memory state.

Variable-ordering. How can we ensure that distinct threads can agree on (or share) sufficient determinate-value assertions to support a verification? We address this problem by using another class of assertion: *variable-order* assertions, which orders two variables whenever the last writes to the variables are causally (i.e., hb) ordered.

Definition 5.5. Let x, y be variables and σ a state with hb relation σ.hb. The *variable-order* assertion $x \xrightarrow{\sigma} y$ holds iff $(\sigma.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb}$.

For example, the state σ in the left of Example 5.2 without the boxed event satisfies $x \xrightarrow{\sigma} y$. When $x \xrightarrow{\sigma} y$, a determinate-value assertion $x \stackrel{\sigma}{=}_t v$ can be "copied" to another thread t', whenever t' performs an *acquiring* read that reads-from the last modification of y and this write is *releasing*. It is easy to see that in a state σ' after such a synchronisation, $\sigma'.last(x)$ is happens-before an operation of t', and thus $x \stackrel{\sigma'}{=}_{t'} v$.

Inference rules. Figure 4 presents a set of rules that precisely captures reasoning principles for determinate-value and variable-order assertions. The "copying" of determinate value assertions is captured in rule Transfer[4]. For the left state in Example 5.2 we can see this copying: when the boxed event $rd_2^{\mathsf{A}}(y, 1)$ occurs (leading to state σ'), the determinate value assertion $x \stackrel{\sigma}{=}_1 2$ is "copied" to thread 2 giving $x \stackrel{\sigma}{=}_2 2$ by rule Transfer. Rule WOrd shows how we introduce variable ordering assertions: a variable ordering assertion can be introduced every time a thread writes to one variable (y in the rule), while having a determinate value assertion on another variable (x in the rule). Note that this rule would not be sound, without Condition (3) of Definition 5.1: since the existence of an hb edge from $\sigma'.last(x)$ to $\sigma'.last(y)$.

Last modification transitions. Observe that the rules in Figure 4 are all conditioned on the modification that is observed in the transition being the last modification to the given variable. Thus, we must be able to prove that a given read or update observes the last modification. There are several ways to do this. It is easy to see that if $x \stackrel{\sigma}{=}_t v$ for some thread t in some state σ then t can only read the last write to x. We formalise this claim in Lemma 5.6 below, and in our case study we show how to use it in verification.

Update operations provide another way to guarantee that a given operation observes the last modification at a given variable. Given a C11 state $((D, _), _, \mathsf{mo})$, an *update-only* variable is any variable x such that for all modifications $m \in D$ with $x = var(m)$, either m is an update or $m \in \mathsf{IWr}$. Note that initially, every variable is an update only variable. In the operational semantics, update-only variables have the property that any new update or write can only be added to the *end* of the modification order. This is a consequence of the fact that for such a variable, any modification but the last is covered. Thus, we have the following lemma.

Lemma 5.6 (Last Modification Transition)**.** *Let $t = tid(e)$ and $x = var(e)$ for some event e. For any reachable transition $(P, \sigma) \xRightarrow{m,e}_{RA} (P', \sigma')$, $m = \sigma.last(x)$ if either $x \stackrel{\sigma}{=}_t v$, for some value v, or x is an update only variable in σ.*

In other cases, other kinds of invariants can be used to guarantee this last-modification property.

[4]We show soundness of these proof rules in the full version.

Example 5.7. Consider the following message-passing interaction between two threads:

Init: $f = 0 \land d = 0$

thread 1	**thread** 2
1 : $d := 5$;	1 : **while** $!f^{\mathsf{A}}$ **do skip**;
2 : $f :=^{\mathsf{R}} 1$;	2 : $r := d$;

Here, thread 1 sets the data variable d to 5, and then indicates that the data is ready by setting the flag variable f to 1. Thread 2 awaits this condition, and then consumes the data. In order to show that this simple program is correct, we must be able to prove that thread 2 always reads the correct value at line 2.

We sketch a proof that for any state σ', where thread 2 is at line 2, we have $d \stackrel{\sigma'}{=}_2 5$. First note that this program satisfies the invariant that for each write w satisfying $var(w) = f$ and $wrval(w) = 1$, w is a releasing write of thread 1 and $last(f) = w$. Using rules NoMod, ModLast and WOrd, after executing line 2 of thread 1, the resulting state σ satisfies $d \stackrel{\sigma}{=}_1 5$ and $d \xrightarrow{\sigma} f$. This fact, along with the invariant above satisfy the premises of the Transfer rule where x is d and y is f. Thus, when the loop exits, into state σ', we have $d \stackrel{\sigma'}{=}_2 5$, as required.

Equipped with these techniques, we now show that Peterson's algorithm with the synchronisation annotations as given in Section 2 guarantees mutual exclusion.

5.2 An Example Verification: Peterson's Algorithm

We turn now to the verification of the version of Peterson's Mutual Exclusion algorithm given in Algorithm 1. Our verification consists of proving a *mutual exclusion invariant* (Theorem 5.8) stating that there is no reachable state in which both processes are in their respective critical sections.

To state our invariants, we make use of an auxiliary *program counter* function, which for each thread, returns the line number of Algorithm 1 that the thread is currently executing. More precisely, for each configuration (P, σ) of Peterson's algorithm, and t a thread with $t \in \{1, 2\}$, the expression $P.pc_t$ returns i when $P(t)$ is the part of the program starting on line i.

The mutual exclusion property for Algorithm 1 is proved in Theorem 5.8, which relies on the following invariants.

$$turn \text{ is an update-only location} \tag{4}$$

$$turn \stackrel{\sigma}{=}_1 2 \lor turn \stackrel{\sigma}{=}_2 1 \tag{5}$$

$$P.pc_t \in \{3,4,5,6\} \implies flag_t \stackrel{\sigma}{=}_t true \tag{6}$$

$$P.pc_t \in \{4,5,6\} \implies flag_t \xrightarrow{\sigma} turn \tag{7}$$

$$P.pc_t \in \{4,5,6\} \land P.pc_{\hat{t}} \in \{4,5,6\} \implies flag_{\hat{t}} \stackrel{\sigma}{=}_t true \lor turn \stackrel{\sigma}{=}_{\hat{t}} t \tag{8}$$

$$P.pc_t = 5 \land P.pc_{\hat{t}} \in \{4,5,6\} \implies turn \stackrel{\sigma}{=}_{\hat{t}} t \tag{9}$$

$$P.pc_t = 2 \implies flag_t \stackrel{\sigma}{=} false \tag{10}$$

$$\textsc{Init}\ \frac{\sigma_0 = ((I, \emptyset), \emptyset, \emptyset) \quad I \subseteq \mathsf{IWr}}{x \overset{\sigma_0}{=}_t wrval(\sigma_0.last(x))} \qquad \textsc{ModLast}\ \frac{x = var(e) \quad e \in \mathsf{Wr}_{|x} \quad m = \sigma.last(x)}{x \overset{\sigma'}{=}_{tid(e)} wrval(e)}$$

$$\textsc{Transfer}\ \frac{y = var(e) \quad x \overset{\sigma}{\to} y \quad x \overset{\sigma}{=}_t v \quad (m, e) \in \mathsf{sw} \quad m = \sigma.last(y)}{x \overset{\sigma'}{=}_{tid(e)} v} \qquad \textsc{UOrd}\ \frac{m \in \mathsf{Wr}_{\mathsf{R}|y} \quad e \in \mathsf{U}_{|y} \quad x \overset{\sigma}{\to} y}{x \overset{\sigma'}{\to} y}$$

$$\textsc{NoMod}\ \frac{e \notin \mathsf{Wr}_{|x} \quad x \overset{\sigma}{=}_t v}{x \overset{\sigma'}{=}_t v} \qquad \textsc{AcqRd}\ \frac{x = var(e) \quad e \in \mathsf{Rd}_{\mathsf{A}|x} \quad m \in \mathsf{Wr}_{\mathsf{R}|x} \quad m = \sigma.last(x)}{x \overset{\sigma'}{=}_{tid(e)} rdval(e)}$$

$$\textsc{WOrd}\ \frac{x \neq y \quad e \in \mathsf{Wr}_{|y} \quad x \overset{\sigma}{=}_{tid(e)} v \quad m = \sigma.last(y)}{x \overset{\sigma'}{\to} y} \qquad \textsc{NoModOrd}\ \frac{e \notin \mathsf{Wr}_{|\{x,y\}} \quad x \overset{\sigma}{\to} y}{x \overset{\sigma'}{\to} y}$$

Figure 4. Rules for determinate-value and variable-order assertions. We assume σ, m, e, σ' satisfy $(_, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (_, \sigma')$.

As in the classical (sequentially consistent) setting, we prove that these invariants hold for the initial configuration and for each transition of the algorithm. For space reasons we only provide details for one of these cases, i.e., where the first test at line 4 is evaluated to false, causing it to enter the critical section.[5]

Proof. We consider the first test at line 4, $flag_t = false$, in the case where the test fails (the success case is very simple). Let (P', σ') be the configuration after the step in question Assume that $P.pc_t = 4$, $P'.pc_t = 5$, and $e = R_t(flag_{\hat{t}}, false)$.

Because e is not a write and the value of $pc_{\hat{t}}$ does not change, it is easy to use the NoMod and NoModOrd rules to show that each invariant except for (9) is preserved. We now prove that (9) is preserved. We do so by proving that $turn \overset{\sigma'}{=}_{\hat{t}} t$ under the assumption that $P'.pc_{\hat{t}} \in \{4, 5, 6\}$. Because $P.pc_{\hat{t}} = P'.pc_{\hat{t}}$, we have $P.pc_{\hat{t}} \in \{4, 5, 6\}$. Furthermore, by Lemma 5.3 and the fact that $e = R_t(flag_{\hat{t}}, false)$ and $m = \sigma.last(flag_{\hat{t}})$ the assertion $flag_{\hat{t}} \overset{\sigma}{=}_t true$ is false. Thus by Invariant 8, we have $turn \overset{\sigma}{=}_{\hat{t}} t$. Then, from rule NoMod, and the fact that e is not a write, we have $turn \overset{\sigma'}{=}_{\hat{t}} t$, as required. □

These invariants are sufficient to prove that Peterson's Algorithm satisfies the mutual exclusion property.

Theorem 5.8 (Mutual exclusion). *For each reachable configuration (P, σ), $P.pc_1 \neq 5 \vee P.pc_2 \neq 5$.*

Proof. Assume that $P.pc_1 = 5$ and $P.pc_2 = 5$. Then, by Property (9) above, we have $turn \overset{\sigma}{=}_1 2$ and $turn \overset{\sigma}{=}_2 1$. But this is impossible by Lemma 5.4. Thus, $P.pc_1 \neq 5$ or $P.pc_2 \neq 5$. □

6 Conclusion and Related Work

We have developed an operational semantics for the RAR fragment of the C11 memory model, which has been shown to be both sound and complete with respect to the axiomatic description. Thus, every state generated by the operational semantics is guaranteed to be one allowed by the axiomatic semantics. Moreover, any execution that is valid with respect to the axiomatic semantics can be generated by the operational semantics. Our semantics relies on a thread-local view of observability[6], which is defined in terms of eco and hb orders. We have developed a proof technique for our operational semantics with a notation that follows conventional proofs of sequentially consistent memory as much as possible. Finally, we have applied this technique to an example verification.

There is a large body of related work; here, we provide a brief snapshot. There are several works aimed at providing operational semantics for a larger subset of C11, including models that aim to address the so-called thin-air problem (that we rule out by the No-Thin-Air axiom), which invariably lead to more complex semantics. Nienhuis et al. [23] provide a semantics that supports inductive reasoning, but they are forced to consider an order that does not include sb. This complicates a verification technique that follows program order. Kang et al. [15] develop an operational model aimed at handling cycles in sb ∪ rf. Again, their sophisticated model handles a larger subset of the C11 language, but at the cost of a more complicated state space and transition relation. Lahav et al. [17] provide an operational model for a stronger release-acquire model, where sb ∪ rf ∪ mo is required to be acyclic.

Kang et al. [15] provide a basic program logic for proving invariants; using their semantics in verification remains an open problem. Jagadeesan et al. [12] develop an operational semantics capable of coping with out-of-order executions for the Java memory model. However, their work focusses on supporting Java compiler optimisations and they do not consider program verification. One avenue for future work is to see how our notions of determinate-value and variable-ordering assertions might be applied to verification in a more sophisticated semantics [12, 15].

Concurrent separation logic (CSL) provides a different approach to verification, and several frameworks have been developed for dealing with C11-style weak memory [8, 9, 13, 14]. These frameworks typically deal with a fragment of C11 containing release/acquire operations but that is not comparable to the fragment of our model. Weak-memory

[5] The full proof is available in the extended version [7].

[6] Our notion of observability differs from those defined in [10, 14].

CSL has been a very active area of research for several years, and we refer the reader to the introduction of [14] for an excellent review.

Finally, recent works have focused on model checking approaches [1, 16], where validation is aimed at efficient consistency checking of the standard axiomatic semantics.

Acknowledgments

This work has been supported by EPSRC grants EP/R032556/1 and EP/M017044/1, and DFG grant WE 2290/12-1. We thank John Wickerson for helpful discussions. We also thank our anonymous reviewers and shepherd (Viktor Vafeiadis) for their comments, which have helped improve the paper.

References

[1] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *PACMPL* 2, OOPSLA (2018), 135:1–135:29.

[2] S. V. Adve and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996), 66–76.

[3] 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.

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

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

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2011. Mathematizing C++ concurrency. In *POPL*, T. Ball and M. Sagiv (Eds.). ACM, 55–66.

[7] S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. 2016. Verifying C11 Programs Operationally. *CoRR* abs/1811.09143 (2016). arXiv:1811.09143 (Full version with proofs.).

[8] M. Doko and V. Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI (LNCS)*, Vol. 9583. Springer, 413–430.

[9] M. Doko and V. Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP*. 448–475.

[10] D. S. Fava, M. Steffen, and V. Stolz. 2018. Operational Semantics of a Weak Memory Model with Channel Synchronization. In *FM (LNCS)*, Vol. 10951. Springer, 258–276.

[11] M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira. 2016. Reasoning about Fences and Relaxed Atomics. In *PDP*. IEEE Computer Society, 520–527.

[12] R. Jagadeesan, C. Pitcher, and J. Riely. 2010. Generative Operational Semantics for Relaxed Memory Models. In *ESOP (LNCS)*, Vol. 6012. Springer, 307–326.

[13] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650.

[14] 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*. 17:1–17:29.

[15] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM, 175–189.

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

[17] O. Lahav, N. Giannarakis, and V. Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*, R. Bodík and R. Majumdar (Eds.). ACM, 649–662.

[18] O. Lahav and V. Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *ICALP (2) (LNCS)*, Vol. 9135. Springer, 311–323.

[19] O. Lahav and V. Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM (LNCS)*, J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou (Eds.), Vol. 9995. 479–495.

[20] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*, A. Cohen and M. T. Vechev (Eds.). ACM, 618–632.

[21] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

[22] C. Lidbury and A. F. Donaldson. 2017. Dynamic race detection for C++11. In *POPL*. ACM, 443–457.

[23] K. Nienhuis, K. Memarian, and P. Sewell. 2016. An operational semantics for C/C++11 concurrency. In *OOPSLA*. ACM, 111–128.

[24] S. S. Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs.* Garland Publishing, New York.

[25] A. Podkopaev, I. Sergey, and A. Nanevski. 2016. Operational Aspects of C/C++ Concurrency. *CoRR* abs/1606.01400 (2016). arXiv:1606.01400

[26] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. 2017. Automatically comparing memory consistency models. In *POPL*. ACM, 190–204.

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

A Proofs of Section 4.2

Theorem 4.4. *Let* $\sigma = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ *be a C11 state reachable from* σ_0 *using relation* $\rightsquigarrow_{RA}$. *Then* σ *satisfies* SB-Total, MO-Valid, RF-Complete, No-Thin-Air *and* Coherence.

Proof of Theorem 4.4. By induction on the number of steps executed to reach σ.

Induction base. The initial state σ_0 satisfies all conditions as all relations are empty and there are no read event in σ_0.

Induction step. Let σ_i be a C11 state reachable in i steps that fulfils the C11 consistency conditions. Let $\sigma_i \overset{e}{\rightsquigarrow}_{C11} \sigma_{i+1}$. We need to show that σ_{i+1} satisfies all conditions.

SB-Total: Follows from definition of + and induction hypothesis.

MO-Valid: Follows from definition of $\mathsf{mo}[w, e]$ and induction hypothesis.

RF-Complete: Follows from rules Read and RMW and induction hypothesis.

No-Thin-Air: Let $\sigma_i = ((D_i, \mathsf{sb}_i), \mathsf{rf}_i, \mathsf{mo}_i)$, assume (by induction hypothesis) that $\mathsf{sb}_i \cup \mathsf{rf}_i$ is acyclic and consider the introduction of element e to σ_i. For each rule in Figure 3, e is maximal in sb_{i+1}. Thus $\mathsf{sb}_{i+1} \cup \mathsf{rf}_i$ is acyclic. Moreover, if e is a write $\mathsf{rf}_{i+1} = \mathsf{rf}_i$, and if e is a read or an update, e is maximal in rf_{i+1}, and hence, $\mathsf{sb}_{i+1} \cup \mathsf{rf}_{i+1}$ is acyclic.

Coherence. Assume $\mathsf{hb}_i; \mathsf{eco}_i^?$ is irreflexive. Consider case distinction on the type of event e.

- e is a read event. This introduces edges $(e', e) \in \mathsf{sb}_{i+1}$, $(w, e) \in \mathsf{rf}_{i+1}$ and $(e, w') \in \mathsf{fr}_{i+1}$ for each w' such that $(w, w') \in \mathsf{mo}_i$. If we have a cycle in $\mathsf{hb}_{i+1}; \mathsf{eco}_{i+1}^?$, the cycle has to pass through e, and therefore also leave e

via some $(e, w') \in \mathsf{fr}_{i+1}$ edge (since these are the only outgoing edges from e). There are two cases:

1. There is a path with edges $(w', e'') \in \mathsf{eco}^?_{i+1}$ and $(e'', e') \in \mathsf{hb}^?_{i+1}$ for some $e'' \in D_i$.
2. There is a path with edges $(w', e'') \in \mathsf{eco}^?_{i+1}$ and $(e'', w) \in \mathsf{hb}^?_{i+1}$ and $(w, e) \in \mathsf{sw}^?_{i+1}$ (due to the events w and e synchronising via R and A synchronisation).

Both cases potentially create a reflexive edge via the composition of edges $(e'', e) \in \mathsf{hb}_{i+1}$ and $(e, e'') \in \mathsf{eco}_{i+1}$. However, we now have $w' \in EW_{\sigma_i}(t)$ and since $(w, w') \in \mathsf{mo}$, we have $w \notin OW_{\sigma_i}(t)$ and hence the edge $(w, e) \in \mathsf{rf}_{i+1}$ cannot exist, giving rise to a contradiction.

- e is a write event. This introduces edges $(e', e) \in \mathsf{sb}_{i+1}$, $(w, e) \in \mathsf{mo}_{i+1}$ and $(r, e) \in \mathsf{fr}_{i+1}$ for any read r in D_i that reads $var(e)$. If e is maximal in mo_{i+1} we are done as $\mathsf{hb}_{i+1}; \mathsf{eco}^?_{i+1}$ cannot be reflexive. Otherwise, there must be an edge that leaves e via an edge $(e, w') \in \mathsf{mo}_{i+1}$. We have two cases:
  1. There is a path with edges $(w', e'') \in \mathsf{eco}^?_{i+1}$ and $(e'', e') \in \mathsf{hb}^?_{i+1}$. This potentially creates a reflexive edge, via the composition of edges $(e'', e) \in \mathsf{hb}_{i+1}$ and $(e, e'') \in \mathsf{eco}_{i+1}$. However, this would mean we have $w \notin OW_{\sigma_i}(t)$, and hence the edge $(w, e) \in \mathsf{mo}_{i+1}$ cannot exist.
  2. There is a path with edges $(w', e'') \in \mathsf{eco}^?_{i+1}$ and $(e'', w) \in \mathsf{hb}_{i+1}$. This potentially creates a reflexive edge, via the composition of edges $(e'', w) \in \mathsf{hb}_{i+1}$ and $(w, e'') \in \mathsf{eco}_{i+1}$. However, this also means that we have edges $(w, w') \in \mathsf{mo}_i$, $(w', e'') \in \mathsf{eco}^?_i$ and $(e'', w) \in \mathsf{hb}_i$, i.e., $\mathsf{hb}_i; \mathsf{eco}^?_i$ is reflexive, which is a contradiction.
  3. There is a path with edges $(w', e'') \in \mathsf{eco}^?_{i+1}$ and $(e'', r) \in \mathsf{hb}_{i+1}$. This case is similar to the one above.
- e is an update event. This introduces edges $(e', e) \in \mathsf{sb}_{i+1}$, $(w, e) \in \mathsf{rf}_{i+1}$, $(w, e) \in \mathsf{mo}_{i+1}$. The proof is similar to the proofs of the read and write cases.

We now show eco_{i+1} is irreflexive, assuming eco_i is irreflexive. We perform case analysis on the type of event e.

- e is a read event. This introduces eco edges $(w, e) \in \mathsf{rf}_{i+1}$ and $(e, w') \in \mathsf{fr}_{i+1}$ for each w' such that $(w, w') \in \mathsf{mo}_i$. If eco_{i+1} is reflexive, we must have an edge $(w', w) \in \mathsf{eco}_{i+1}$. But this means we have edges $(w', w) \in \mathsf{eco}_i$ and $(w, w') \in \mathsf{mo}_i \subseteq \mathsf{eco}_{i+1}$, i.e., eco_i is reflexive, which is a contradiction.
- e is a write event. This introduces eco edges $(w, e) \in \mathsf{mo}_{i+1}$ and $(r, e) \in \mathsf{fr}_{i+1}$. If e is maximal in mo_{i+1} we are done as eco_{i+1} cannot be reflexive. Otherwise, there is an path that leaves e via an edge $(e, w') \in \mathsf{mo}_{i+1}$. Then we either have a path with edge $(w', w) \in \mathsf{eco}_{i+1}$ or a path with edge $(w', r) \in \mathsf{eco}_{i+1}$. However, both contradict the assumption that eco_i is irreflexive.
- e is an update event. This introduces eco edges $(w, e) \in \mathsf{rf}_{i+1}$, $(w, e) \in \mathsf{mo}_{i+1}$, as well as $(e, w') \in \mathsf{mo}_{i+1}$ and $(e, w') \in \mathsf{fr}_{i+1}$ for each w' such that $(w, w') \in \mathsf{mo}_i$. If eco_{i+1} is reflexive, we must have an edge $(w', w) \in \mathsf{eco}_{i+1}$. But this means we have edges $(w', w) \in \mathsf{eco}_i$ and $(w, w') \in \mathsf{mo}_i \subseteq \mathsf{eco}_{i+1}$, i.e., eco_i is reflexive, which is a contradiction.

□

Lemma 4.7. *Let* $Q = (P_0, \gamma_0) \overset{e_1}{\Longrightarrow}_{PE} (P_1, \gamma_1) \overset{e_2}{\Longrightarrow}_{PE} \dots \overset{e_k}{\Longrightarrow}_{PE} (P_k, \gamma_k)$ *and* $\gamma_k = (D_k, \mathsf{sb}_k)$. *Then for every linearization* $f_1, \dots, f_k$ *of* sb_k, *there exist programs* $P'_1, \dots, P'_{n-1}$ *and pre-execution states* $\gamma'_1, \dots, \gamma'_{n-1}$ *such that*

$$(P_0, \gamma_0) \overset{f_1}{\Longrightarrow}_{PE} (P'_1, \gamma'_1) \overset{f_2}{\Longrightarrow}_{PE} \dots \overset{f_k}{\Longrightarrow}_{PE} (P_k, \gamma_k) .$$

Proof of Lemma 4.7. Let $F = f_1, \dots, f_k$ and $E = e_1, \dots, e_k$, and let δ_i represent the pair (P_i, γ_i). Suppose f_1, the first element of F is the element e_i, the ith element of E. We show that Q can be transformed into a valid sequence of PE steps such that e_i is the first event considered. By definition, we have that Q is the sequence:

$$\delta_0 \overset{e_1}{\Longrightarrow}_{PE} \dots \delta_{i-2} \overset{e_{i-1}}{\Longrightarrow}_{PE} \delta_{i-1} \overset{e_i}{\Longrightarrow}_{PE} \delta_i \dots \overset{e_k}{\Longrightarrow}_{PE} \delta_k .$$

Since both E and F are a linearizations of sb_k, we have the property:

$$\forall j.\ 0 \le j \le i-1 \Rightarrow tid(e_j) \ne tid(e_i) \tag{11}$$

By (11), we have in particular that $tid(e_{i-1}) \ne tid(e_i)$. Thus by Proposition 4.6 there must exist a δ'_{i-1} such that $\delta_{i-2} \overset{e_i}{\Longrightarrow}_{PE} \delta'_{i-1} \overset{e_{i-1}}{\Longrightarrow}_{PE} \delta_i$, and hence, a valid pre-execution sequence

$$\delta_0 \overset{e_1}{\Longrightarrow}_{PE} \dots \delta_{i-2} \overset{e_i}{\Longrightarrow}_{PE} \delta'_{i-1} \overset{e_{i-1}}{\Longrightarrow}_{PE} \delta_i \dots \overset{e_k}{\Longrightarrow}_{PE} \delta_k .$$

Again by property (11), we have that $tid(e_{i-2}) \ne tid(e_i)$ and the process above can be repeated so that we obtain:

$$\delta_0 \overset{e_1}{\Longrightarrow}_{PE} \dots \overset{e_i}{\Longrightarrow}_{PE} \delta'_{i-2} \overset{e_{i-2}}{\Longrightarrow}_{PE} \delta'_{i-1} \overset{e_{i-1}}{\Longrightarrow}_{PE} \delta_i \dots \overset{e_k}{\Longrightarrow}_{PE} \delta_k .$$

Further repeating this process, we obtain:

$$\delta_0 \overset{e_i}{\Longrightarrow}_{PE} \delta'_1 \overset{e_1}{\Longrightarrow}_{PE} \delta'_2 \dots \delta'_{i-2} \overset{e_{i-2}}{\Longrightarrow}_{PE} \delta'_{i-1} \overset{e_{i-1}}{\Longrightarrow}_{PE} \delta_i \overset{e_{i+1}}{\Longrightarrow}_{PE} \dots \overset{e_k}{\Longrightarrow}_{PE} \delta_k .$$

which (since $e_i = f_1$) is equivalent to:

$$\delta_0 \overset{f_1}{\Longrightarrow}_{PE} \delta'_1 \overset{e_1}{\Longrightarrow}_{PE} \delta'_2 \dots \delta'_{i-2} \overset{e_{i-2}}{\Longrightarrow}_{PE} \delta'_{i-1} \overset{e_{i-1}}{\Longrightarrow}_{PE} \delta_i \overset{e_{i+1}}{\Longrightarrow}_{PE} \dots \overset{e_k}{\Longrightarrow}_{PE} \delta_k .$$

We can now repeat the entire process for f_2 using the property and percolate the element it corresponds to in E it to the correct position in F since $f_2 \ne e_i$, i.e. f_2 corresponds to an element in $\{e_1, \dots e_k\} \backslash \{e_i\}$. Assuming that f_2 corresponds to position i' in E, we have property $\forall j.\ 1 \le j \le i'-1 \Rightarrow tid(e_j) \ne tid(e_{i'})$ (analogous to (11)), where the lower index

is increased by 1 and upper index is adjusted to $i' - 1$. Once f_2 is in position, we can repeat for f_3 and so forth. □ □

We now show that for every justifiable pre-execution there is an execution of the C11 semantics that ends in the C11 state justifying the pre-execution. The theorem uses a notion that restricts pre-executions and C11 executions to a set of events. For a set of events $E \subseteq D$, we define:

$$(D, \mathsf{sb})_{\downarrow E} = (E, \mathsf{sb} \cap (E \times E))$$
$$(\gamma, \mathsf{rf}, \mathsf{mo})_{\downarrow E} = (\gamma_{\downarrow E}, \mathsf{rf} \cap (E \times E), \mathsf{mo} \cap (E \times E))$$

In the completeness proof, we assume that the given pre-execution sequence $(P_0, \gamma_0) \stackrel{e_1}{\Longrightarrow}_{PE} (P_1, \gamma_1) \stackrel{e_2}{\Longrightarrow}_{PE} \dots \stackrel{e_k}{\Longrightarrow}_{PE} (P_k, \gamma_k)$ has been reordered such that $e_1 \dots e_k$ is a linearization of $\mathsf{sb}_k \cup \mathsf{rf}_k$, where rf_k is the reads-from relation used in the justification of γ_k. Such a linearization is possible since $\mathsf{sb}_k \cup \mathsf{rf}_k$ is acyclic (axiom No-Thin-Air).

Theorem 4.8. *Suppose* $(P_0, \gamma_0) \stackrel{e_1}{\Longrightarrow}_{PE} (P_1, \gamma_1) \stackrel{e_2}{\Longrightarrow}_{PE} \dots \stackrel{e_k}{\Longrightarrow}_{PE} (P_k, \gamma_k)$ *such that* $\gamma_k = (D_k, \mathsf{sb}_k)$ *is justifiable with justification* $\sigma_k = (\gamma_k, \mathsf{rf}_k, \mathsf{mo}_k)$ *and* $e_1, \dots, e_k$ *is a linearization of* $\mathsf{sb}_k \cup \mathsf{rf}_k$. *Then*

$$(P_0, \sigma_0) \stackrel{e_1}{\Longrightarrow}_{RA} (P_1, \sigma_1) \stackrel{e_2}{\Longrightarrow}_{RA} \dots \stackrel{e_k}{\Longrightarrow}_{RA} (P_k, \sigma_k)$$

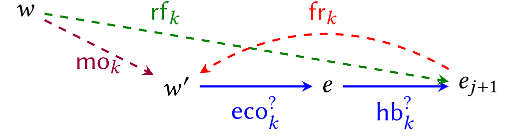
where $\sigma_i = (\gamma_k, \mathsf{rf}_k, \mathsf{mo}_k)_{\downarrow\{e_1,\dots,e_i\}}$, $0 < i < k$.

Proof of Theorem 4.8. By induction on the number of steps.
Base case. The initial configurations agree and hence the claim holds for 0 steps.

Induction step. Let the above claim hold for sequences up to length j. We perform a case split on the type of event $e_{j+1} \notin D_j$.

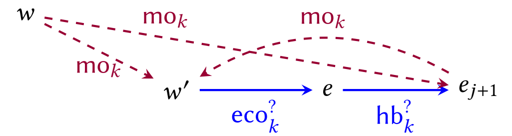
1. e_{j+1} is a read event of a thread t, i.e., $tid(e_{j+1}) = t$. Let w be the write event that e_{j+1} reads from, i.e., $(w, e_{j+1}) \in \mathsf{rf}_k$. We known $w \in D_j$ since we consider elements in $\mathsf{sb}_k \cup \mathsf{rf}_k$ order.
   We need to show that $w \in OW_{\sigma_j}(t)$. The proof is by contradiction. Assume $w \notin OW_{\sigma_j}(t)$, then there exists a $w' \in EW_{\sigma_j}(t)$ such that $(w, w') \in \mathsf{mo}_j$. Hence $(e_{j+1}, w') \in \mathsf{fr}_k$ and there exists some e such that $(w', e) \in \mathsf{eco}_k^?$ and $(e, e_{j+1}) \in \mathsf{hb}_k^?$. There are three possibilities:
   - $(w', e), (e, e_{j+1}) \in Id$. This is an immediate contradiction since it implies $w' = e_{j+1}$.
   - $(w', e) \in \mathsf{eco}_k$ and $(e, e_{j+1}) \in Id$, i.e., $e = e_{j+1}$. This contradicts the assumption that eco_k is irreflexive.
   - $(w', e) \in \mathsf{eco}_k^?$ and $(e, e_{j+1}) \in \mathsf{hb}_k$. We then have $(e_{j+1}, e) \in \mathsf{eco}_k$ resulting in a contradiction to the assumption that $\mathsf{hb}_k; \mathsf{eco}_k^?$ is irreflexive.

   The contradictory scenario is depicted by the following diagram:

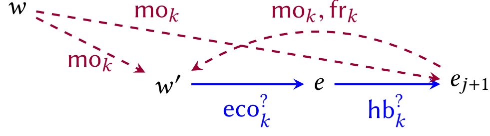


2. Suppose e_{j+1} is a write event and w is the immediate predecessor of e_{j+1} in mo_k. Note that w may either be a write or an update event. We must show that it is possible to take a $\stackrel{e_{j+1}}{\rightsquigarrow}_{RA}$ step such that e_{j+1} is placed immediately after w. To this end, we must show that $w \in OW_{\sigma_j}(t)$.
   Suppose not, i.e., $w \notin OW_{\sigma_j}(t)$. Then there exists an event $w' \in EW_{\sigma_j}(t)$ such that $(w, w') \in \mathsf{mo}$ and an event e such that $(w', e) \in \mathsf{eco}_k^?$ and $(e, e_{j+1}) \in \mathsf{hb}_k^?$. Since we have assumed w is an immediate predecessor of e_{j+1} in mo_k and that $(w, w') \in \mathsf{mo}_k$, we must have (e_{j+1}, w'). There are three possibilities:
   - $(w', e), (e, e_{j+1}) \in Id$. This is an immediate contradiction since it implies $w' = e_{j+1}$ and we have assumed $w' \in D_j$, $e_{j+1} \notin D_j$.
   - $(w', e) \in \mathsf{eco}_k$ and $(e, e_{j+1}) \in Id$, i.e., $e = e_{j+1}$. This contradicts the assumption that eco_k is irreflexive.
   - $(w', e) \in \mathsf{eco}_k^?$ and $(e, e_{j+1}) \in \mathsf{hb}_k$. We then have $(e_{j+1}, e) \in \mathsf{eco}_k$ resulting in a contradiction to the assumption that $\mathsf{hb}_k; \mathsf{eco}_k^?$ is irreflexive.

   The contradictory scenario is depicted by the following diagram:



   We also need to show that w selected is not covered. Again assume the contrary: there exists some update event u such that $(w, u) \in \mathsf{rf}_k$. Then $(w, u) \in \mathsf{mo}_k$ as well. Hence there is an edge $(u, e_{j+1}) \in \mathsf{fr}_k$. Since the update u and e_{j+1} write to the same location, they need to be mo-ordered. Here we have two cases:
   - If $(u, e_{j+1}) \in \mathsf{mo}_k$, then w is not the immediate predecessor of e_{j+1} in mo_k.
   - If $(e_{j+1}, u) \in \mathsf{mo}_k$, then the fr_k edge and the mo_k edge together form a cyle, contradicting irreflexivity of eco_k.
3. Suppose e_{j+1} is an update event and w is the immediate predecessor of e_{j+1} in mo_k. We must show that it is possible to take a $\stackrel{e_{j+1}}{\rightsquigarrow}_{RA}$ step such that e_{j+1} is placed immediately after w. This case is a combination of the read and write cases, namely if we assume $w \notin OW_{\sigma_j}(t)$, then there must exist a w' and e as shown in the diagram below, which is a contradiction.



□

B Proofs for Section 5

B.1 Proofs of lemmas

Lemma 5.3. (Determinate-Value Read) *For any* READ *or* RMW *transition* $(P, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (P', \sigma')$, *if* $var(e) \overset{\sigma}{=}_{tid(e)} v$, *then* $rdval(e) = v$.

Proof. By the definition of $var(e) \overset{\sigma}{=}_{tid(e)} v$, we know $m = \sigma.last(x)$. Both the READ or RMW transitions stipulate that the value read is the $rdval(e) = wrval(m) = v$. □

Lemma 5.4. (Determinate-Value Agreement) *For any threads* t, t' *location* x, *and values* v, v', *if* $x \overset{\sigma}{=}_t v$ *and* $x \overset{\sigma}{=}_{t'} v'$ *then* $v = v'$, *and thus* t *and* t' *agree on the value of* x.

Proof. By $x \overset{\sigma}{=}_t v$ and $x \overset{\sigma}{=}_{t'} v'$, we have $OW_\sigma(t) = OW_\sigma(t') = \{\sigma.last(x)\}$, and thus $v = v'$. □

Lemma 5.6. *Let* $t = tid(e)$ *and* $x = var(e)$ *for some event* e. *For any reachable transition* $(P, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (P', \sigma')$, $m = \sigma.last(x)$ *if either of the following conditions hold:*

1. $x \overset{\sigma}{=}_t v$, *for some value* v, *or*
2. x *is an update only location in* σ.

Proof. If property 1 holds, then $OW_\sigma(t)_{|x} = \{\sigma.last(x)\}$, and thus $m = \sigma.last(x)$. If property 2 holds, then every modification to x is covered, except the last. Thus, because m is not covered, $m = \sigma.last(x)$. □

B.2 Soundness of determinate-value and variable-order assertions

For simplicity, we copy Figure 4 as Figure 5. We refer to the set in condition (3) of Definition 5.1 as the *happens-before cone* of t in σ, and hence define:

$$\sigma.hbc(t) = I_\sigma \cup \{e \mid \exists e'.\ tid(e) = t \wedge (e, e') \in \sigma.\mathsf{hb}^?\}$$

Lemma B.1. INIT *is valid.*

Proof. We have $\sigma_0 = ((I, \emptyset), \emptyset, \emptyset)$. We have three sub proofs

1. Since $\mathsf{mo} = \emptyset$, we have $OW_{\sigma_0}(t) = I$, and also $|I_{|x}| = 1$ and hence $I_{|x} = \{\sigma_0.last(x)\} = OW_{\sigma_0}(t)_{|x}$.
2. Trivial by definition.
3. Immediate since $\sigma.last(x) \in I$.

□

Lemma B.2 (Establish Determinate-Values). *For any reachable transition* $(P, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (P', \sigma')$, *the rules* NOMOD, MODLAST, TRANSFER *and* ACQREAD *from Figure 5 hold.*

Proof.
NOMOD. This is easy to check since $\sigma.last(x) = \sigma'.last(x)$ and $OW_\sigma(t)_{|x} = OW_{\sigma'}(t)_{|x}$.

MODLAST. Since $m = \sigma.last(x)$, the new modification e is added to the end of $\mathsf{mo}_{|x}$, so that $\sigma'.last(x) = e$. Because $e \in OW_{\sigma'}(t)_{|x}$ and e is mo-after every other modification in $\sigma'.\mathsf{mo}_{|x}$, $OW_{\sigma'}(t)_{|x} = \{e\}$. Finally, because $tid(e) = t$, $e \in \{e \mid \exists e'.\ tid(e) = t \wedge (e, e') \in \sigma.\mathsf{hb}^?\}$.

TRANSFER. First note that because $x \overset{\sigma}{\rightarrow} y$ we have

$$(\sigma.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb}\ . \tag{12}$$

Because hb is irreflexive, we also have $x \neq y$, and thus by NOMOD:

$$\sigma'.last(x) = \sigma.last(x)\ . \tag{13}$$

By (12) and (13), we have $(\sigma'.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb}$. Moreover,

	$(\sigma'.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb}$	
$\Rightarrow$	$(\sigma'.last(x), \sigma.last(y)) \in \sigma'.\mathsf{hb}$	because $\sigma.\mathsf{hb} \subseteq \sigma'.\mathsf{hb}$
$\Rightarrow$	$(\sigma.last(y), e) \in \sigma'.\mathsf{hb} \wedge$	assumption
	$(\sigma'.last(x), e) \in \sigma'.\mathsf{hb}$	$(\sigma.last(y), e) \in \mathsf{sw}$

Therefore,

$$\sigma'.last(x) \in \sigma'.hbc(tid(e))$$

This proves the third property of the determinate-value assertion. We prove the two remaining properties of the determinate-value assertion:

- Since $\sigma'.last(x) \in \sigma'.hbc(tid(e))$, we have $\sigma'.last(x) \in EW_\sigma(t)$, and therefore $\sigma'.OW(t)_{|x} = \{\sigma'.last(x)\}$.
- By (13), $wrval(\sigma'.last(x)) = v$ iff $wrval(\sigma.last(x)) = v$, which is true because $x \overset{\sigma}{=}_{t'} v$.

ACQRD. We know $\sigma'.\mathsf{mo}_{|x} = \sigma.\mathsf{mo}_{|x}$ and thus $\sigma'.last(x) = \sigma.last(x)$. Therefore by the assumption $m = \sigma.last(x)$, we have $m = \sigma'.last(x)$. Because $m \in EW_{\sigma'}(t)$ and m is maximal in $\sigma'.\mathsf{mo}_{|x}$ we have $\sigma'.OW(t)_{|x} = \{m\} = \{\sigma'.last(x)\}$ by definition of $OW_{\sigma'}$. The fact that $rdval(e) = wrval(m)$ follows from the premises of rules READ and RMW. Finally, we have $(m, e) \in \mathsf{sw} \subseteq \sigma'.\mathsf{hb}$ thus $\sigma'.last(x) \in \sigma.hbc(t)$. □

Lemma B.3 (Establish Location-Order)**.** *For any reachable transition* $(P, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (P', \sigma')$, *the rules* WRITEORD *and* NOMODORD *hold.*

Proof.
WRITEORD. Note that $\sigma.last(x) = \sigma'.last(x)$ since $x \neq y$ and $e \in \mathsf{Wr}_{|y}$. By $x \overset{\sigma}{=}_{tid(e)} v$, we have $\sigma.last(x) \in \sigma.hbc(tid(e))$. Expanding the definition of hbc and reformulating slightly, we see that

$$\sigma.hbc(tid(e)) = I_\sigma \cup \{e' \mid tid(e') = tid(e)\} \cup \{e' \mid \exists e''.\ tid(e'') = tid(e) \wedge (e', e'') \in \sigma.\mathsf{hb}\}$$

$$\textsc{Init}\ \frac{\sigma_0 = ((I, \emptyset), \emptyset, \emptyset) \quad I \subseteq \mathsf{IWr}}{x \overset{\sigma_0}{=}_t wrval(\sigma_0.last(x))} \qquad \textsc{ModLast}\ \frac{x = var(e) \quad e \in \mathsf{Wr}_{|x} \quad m = \sigma.last(x)}{x \overset{\sigma'}{=}_{tid(e)} wrval(e)}$$

$$\textsc{Transfer}\ \frac{y = var(e) \quad x \overset{\sigma}{\rightarrow} y \quad x \overset{\sigma}{=}_t v \quad (m,e) \in \mathsf{sw} \quad m = \sigma.last(y)}{x \overset{\sigma'}{=}_{tid(e)} v} \qquad \textsc{UOrd}\ \frac{m \in \mathsf{Wr}_{\mathsf{R}|y} \quad e \in \mathsf{U}_{|y} \quad x \overset{\sigma}{\rightarrow} y}{x \overset{\sigma'}{\rightarrow} y}$$

$$\textsc{NoMod}\ \frac{e \notin \mathsf{Wr}_{|x} \quad x \overset{\sigma}{=}_t v}{x \overset{\sigma'}{=}_t v} \qquad \textsc{AcqRd}\ \frac{x = var(e) \quad e \in \mathsf{Rd}_{\mathsf{A}|x} \quad m \in \mathsf{Wr}_{\mathsf{R}|x} \quad m = \sigma.last(x)}{x \overset{\sigma'}{=}_{tid(e)} rdval(e)}$$

$$\textsc{WOrd}\ \frac{x \neq y \quad e \in \mathsf{Wr}_{|y} \quad x \overset{\sigma}{=}_{tid(e)} v \quad m = \sigma.last(y)}{x \overset{\sigma'}{\rightarrow} y} \qquad \textsc{NoModOrd}\ \frac{e \notin \mathsf{Wr}_{|\{x,y\}} \quad x \overset{\sigma}{\rightarrow} y}{x \overset{\sigma'}{\rightarrow} y}$$

Figure 5. Rules for determinate-value and variable-order assertions. We assume σ, m, e, σ' satisfy $(_, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (_, \sigma')$.

Thus, there are three cases to consider. 1. $\sigma.last(x) \in I_\sigma$. In this case, we get $x \overset{\sigma'}{\rightarrow} y$ since we will have

$$(\sigma'.last(x), \sigma'.last(y)) \in \sigma'.\mathsf{sb} \subseteq \sigma'.\mathsf{hb}.$$

2. $tid(\sigma.last(x)) = tid(e)$. By transition rules Write and RMW of our operational semantics, $(\sigma.last(x), e) \in \sigma'.\mathsf{sb}$ and hence $(\sigma'.last(x), e) \in \sigma'.\mathsf{hb}$, or equivalently $x \overset{\sigma'}{\rightarrow} y$.
3. There exists an e' with $tid(e') = tid(e)$ and $(\sigma.last(x), e') \in \sigma.\mathsf{hb}$. Because $\sigma'.last(x) = \sigma.last(x)$ and $\sigma.\mathsf{hb} \subseteq \sigma'.\mathsf{hb}$ we have $(\sigma'.last(x), e') \in \sigma'.\mathsf{hb}$. By the modification transitions of our operational semantics we also have $(e', e) \in \sigma'.\mathsf{sb}$, and thus (putting the two together) we have $(\sigma'.last(x), e) \in \sigma'.\mathsf{hb}$ as required.

NoModOrd. This is easy to check since because e is not a modification of x or y we have $\sigma'.last(x) = \sigma.last(x)$ and $\sigma'.last(y) = \sigma.last(y)$. Now, because $(\sigma.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb}$ (the content of $x \overset{\sigma}{\rightarrow} y$) and the fact that $\sigma.\mathsf{hb} \subseteq \sigma'.\mathsf{hb}$, it follows that $(\sigma'.last(x), \sigma'.last(y)) \in \sigma'.\mathsf{hb}$, or equivalently $x \overset{\sigma'}{\rightarrow} y$.

UpdOrd. There are two cases to consider. First, assume $m \neq \sigma.last(y)$. In this case, $\sigma'.last(y) = \sigma.last(y)$. Because $x \overset{\sigma}{\rightarrow} y$, we have $(\sigma.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb} \subseteq \sigma'.\mathsf{hb}$, and thus $(\sigma.last(x), \sigma'.last(y)) \in \sigma'.\mathsf{hb}$. Because $x \overset{\sigma}{\rightarrow} y$ (by the irreflexivity of hb), x and y are distinct variables and thus $e \notin \mathsf{Wr}_{|x}$. Therefore, $\sigma'.last(x) = \sigma.last(x)$, so $(\sigma.last(x), \sigma'.last(y)) \in \sigma'.\mathsf{hb}$ as required.

Second, assume $m = \sigma.last(y)$. In this case $\sigma'.last(y) = e$. Furthermore, because $m \in \mathsf{Wr}_{R|y}$ and e is an update (which is acquiring) we have $(m, e) \in \mathsf{sw}$ and therefore $(m, e) \in \sigma'.\mathsf{hb}$. Because $x \overset{\sigma}{\rightarrow} y$ we have $(\sigma.last(x), \sigma.last(y)) \in \sigma.\mathsf{hb} \subseteq \sigma'.\mathsf{hb}$ and thus, $(\sigma.last(x), m) \in \sigma'.\mathsf{hb}$ so $(\sigma.last(x), e) \in \sigma'.\mathsf{hb}$ by transitivity. Finally, because $\sigma'.last(x) = \sigma.last(x)$ and $\sigma'.last(y) = e$ we have $(\sigma'.last(x), \sigma'.last(y)) \in \sigma'.\mathsf{hb}$ as required. □

C Relationship with Canonical C11

In this section, we describe the relationship between the version of the C11 semantics given in Section 4 and that of [5], on which it is closely based. The semantics of [5] uses a notion of candidate execution as described below. We focus on the relationship between our notion of *validity* (Definition 4.2) of this paper, and their notion of *consistency*, which we call *canonical consistency* in this appendix. We prove that, for any candidate execution, validity *without the* NoThinAir *axiom* and a version of canonical consistency (described below) are equivalent.

Batty et al [5] use a notion of *candidate execution* (Definition 7, [5]), which gives certain well-formedness conditions on executions. For the purposes of this appendix, we define *candidate executions* as follows:

Definition C.1 (Candidate Execution)**.** A tuple $((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ is a *candidate execution* if it satisfies the conjunction of the conditions RF-Complete, MO-Valid and SB-Total of Definition 4.2 in our current paper.

Minor variations in presentation prevent us from claiming that the definition just given is strictly equivalent to Definition 7 of [5]. Principally, [5] employs an equivalence relation to determine when two operations are on the same thread, whereas we index operations with a thread identifier. Another difference is that Batty et al. [5] define the hb relation such that initialising writes are hb-prior to all other events, whereas we stipulate that initialising writes are sb-prior to all other events (thus ensuring the hb-ordering indirectly). With these caveats aside, the definition of candidate execution given here is essentially the same as that of [5].

Let $(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ be a candidate execution. As is true for all versions of the C11 memory model, canonical consistency is defined in terms of the happens-before relation, which in turn is defined in terms of the synchronises-with relation. The synchronises-with relation of [5], which we call *canonical synchronises-with* and denote by sw_C is slightly larger than our definition

$$\mathsf{sw} \subseteq \mathsf{sw}_C$$

The extra edges in sw_C relate to the so-called *release sequences*, which we have ignored in our presentation. The effect of this relaxation is that our version defines a weaker semantics, with more valid executions.

The happens-before relation in [5], which we call *canonical happens-before* and denote hb_C, is defined as follows

$$\mathsf{hb}_C = (\mathsf{sb} \cup (I \times \neg I) \cup \mathsf{sw}_C)^+$$

where $\neg I$ is the complement of the set of initialising writes. In our version of the semantics, $I \times \neg I \subseteq \mathsf{sb}$, and thus $\mathsf{sb} \cup (I \times \neg I) = \mathsf{sb}$ so

$$\mathsf{hb}_C = (\mathsf{sb} \cup \mathsf{sw}_C)^+$$

similar to our definition. Thus, because $\mathsf{sw} \subseteq \mathsf{sw}_C$, $\mathsf{hb} \subseteq \mathsf{hb}_C$.

We now present the definition of consistency given in [5] as it relates to the RAR fragment.

Definition C.2 (Canonical RAR Consistency)**.** A candidate execution $\mathbb{D} = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ is canonically consistent if all the following conditions hold

$$\begin{aligned} &\mathit{irrefl}(\mathsf{hb}_C) && \text{(HB-C)} \\ &\mathit{irrefl}((\mathsf{rf}^{-1})^?; \mathsf{mo}; \mathsf{rf}^?; \mathsf{hb}_C) && \text{(COH-C)} \\ &\mathit{irrefl}(\mathsf{rf}; \mathsf{hb}_C) && \text{(RF-C)} \\ &\mathit{irrefl}(\mathsf{rf} \cup (\mathsf{mo}; \mathsf{mo}; \mathsf{rf}^{-1}) \cup (\mathsf{mo}; \mathsf{rf})) && \text{(UPD-C)} \end{aligned}$$

where hb_C is defined from $\mathbb{D}$ as above.

To account for the fact that $\mathsf{sw} \subseteq \mathsf{sw}_C$, and thus $\mathsf{hb} \subseteq \mathsf{hb}_C$, we give a slightly weaker notion of canonical consistency, called *weak canonical RAR consistency*, which we prove equivalent to our notion of validity. This weaker condition is obtained from the stronger by replacing hb_C by hb. Also, to simplify presentation of the proof, we split the condition RF-C into two conditions: one called RF, and one called RFI that explicitly requires the irreflexivity of the rf relation. This second change is purely presentational, and does not change the strength of the semantics.

Definition C.3 (Weak Canonical RAR Consistency)**.** A candidate execution $\mathbb{D} = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$ is canonically consistent if all the following conditions hold

$$\begin{aligned} &\mathit{irrefl}(\mathsf{hb}) && \text{(HB)} \\ &\mathit{irrefl}((\mathsf{rf}^{-1})^?; \mathsf{mo}; \mathsf{rf}^?; \mathsf{hb}) && \text{(COH)} \\ &\mathit{irrefl}(\mathsf{rf}; \mathsf{hb}) && \text{(RF)} \\ &\mathit{irrefl}(\mathsf{rf}) && \text{(RFI)} \\ &\mathit{irrefl}((\mathsf{mo}; \mathsf{mo}; \mathsf{rf}^{-1}) \cup (\mathsf{mo}; \mathsf{rf})) && \text{(UPD)} \end{aligned}$$

where hb is defined from $\mathbb{D}$ as usual.

As we shall see, the validity condition $\mathit{irrefl}(\mathsf{eco}^?; \mathsf{hb})$ (as used in the coherence condition of Definition 4.2 in our paper) captures the collective effect of conditions HB, COH and RF. The condition UPD, which we sometimes call *update atomicity*, requires that each update appears in mo-order immediately after the write that the update reads from. As we shall see, the validity condition $\mathit{irrefl}(\mathsf{eco})$ implies update atomicity, and for any candidate execution, the update atomicity property implies $\mathit{irrefl}(\mathsf{eco})$.

The following lemma follows easily from the fact that $\mathsf{hb} \subseteq \mathsf{hb}_C$.

Lemma C.4. *For any candidate execution*

$$\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo}),$$

if $\mathbb{D}$ is canonical consistent, then it is weakly canonical consistent.

From now on, we consider only weak canonical consistency. Thus, when we refer to properties HB, COH, RF, and UPD we mean those of *weak* canonical consistency.

For the remainder of the section, we work towards proving the following theorem

Theorem C.5. *For any candidate execution*

$$\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo}),$$

$\mathbb{D}$ is weakly canonical consistent iff $\mathbb{D}$ satisfies Coherence *of Definition 4.2 on page 7.*

As we shall see, much of our proof is about reformulating the various relations and axioms that make-up the canonical memory model.

The following lemma provides a more convenient form for the UPD property.

Lemma C.6. *For any candidate execution*

$$\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo}),$$

the UPD *condition (that is, $\mathit{irrefl}((\mathsf{mo}; \mathsf{mo}; \mathsf{rf}^{-1}) \cup (\mathsf{mo}; \mathsf{rf}))$) is equivalent to $\mathit{irrefl}(\mathsf{fr}; \mathsf{mo}) \wedge \mathit{irrefl}(\mathsf{rf}; \mathsf{mo})$.*

Proof. First note that for any relations r, s we have both

$$\mathit{irrefl}(r; s) \Leftrightarrow \mathit{irrefl}(s; r)$$
$$\mathit{irrefl}(r \cup s) \Leftrightarrow \mathit{irrefl}(r) \wedge \mathit{irrefl}(s).$$

Using these equivalences, UPD is equivalent to

$$\mathit{irrefl}(\mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{mo}) \wedge \mathit{irrefl}(\mathsf{rf}; \mathsf{mo}).$$

It remains to show that $\mathit{irrefl}(\mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{mo})$ is equivalent to $\mathit{irrefl}(\mathsf{fr}; \mathsf{mo})$. Because $\mathsf{fr} \subseteq \mathsf{rf}^{-1}; \mathsf{mo}$, we have

$$\mathsf{fr}; \mathsf{mo} \subseteq \mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{mo}$$

and thus if $\mathit{irrefl}(\mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{mo})$ then $\mathit{irrefl}(\mathsf{fr}; \mathsf{mo})$.

Finally, we show that if there is a cycle in $\mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{mo}$ then there is also one in $\mathsf{fr}; \mathsf{mo}$. Assume that $(x, x) \in \mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{mo}$. Then there is some y such that $(x, y) \in \mathsf{rf}^{-1}; \mathsf{mo}$ and $(y, x) \in \mathsf{mo}$. There are two cases to consider. In the first case, $y = x$. But this is impossible because then we would have $(x, x) \in \mathsf{mo}$, contrary to the irreflexivity of mo. In the second case, $x \neq y$, but then $(x, y) \in (\mathsf{rf}^{-1}; \mathsf{mo}) - \mathit{Id} = \mathsf{fr}$ so $(x, x) \in \mathsf{fr}; \mathsf{mo}$ and we are done. □

The first lemma says that each update operation can only read from its immediate mo predecessor.

Lemma C.7 (Update orderings)**.** *For any candidate execution $(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$, satisfying* UPD *the following properties hold for any update $u \in D$ and event $x \in D$:*

i

$$(u, x) \in \mathsf{fr} \implies (u, x) \in \mathsf{mo}$$

ii

$$(x, u) \in \mathsf{rf} \implies (x, u) \in \mathsf{mo}$$

Proof. Note first that mo must order u and x (in some direction). This is because $var(u) = var(x)$, u is a modification, x is a modification (because it either has an incoming fr edge or an outgoing rf edge) and mo is total over modifications to the same location. Therefore, it is sufficient to derive a contradiction from the assumption that mo orders the two operations the "wrong" way.

Assume for Property i that $(u, x) \in \mathsf{fr}$ and $(x, u) \in \mathsf{mo}$. But then $(u, u) \in \mathsf{fr}; \mathsf{mo}$ contrary to the UPD property, as formulated in Lemma C.6.

Assume for Property ii that $(x, u) \in \mathsf{rf}$ and $(u, x) \in \mathsf{mo}$. But then $(x, x) \in \mathsf{rf}; \mathsf{mo}$ contrary to the UPD property, as formulated in Lemma C.6. ◻

We next need some properties about the structure of eco.

Lemma C.8 (Coherence inclusions)**.** *For any candidate execution* $(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$*, that satisfies the* UPD *property the following inclusions hold:*

i $\mathsf{rf}; \mathsf{fr} \subseteq \mathsf{mo}$
ii $\mathsf{rf}; \mathsf{mo} \subseteq \mathsf{mo}$
iii $\mathsf{rf}; \mathsf{rf} \subseteq \mathsf{mo}; \mathsf{rf}$
iv $\mathsf{mo}; \mathsf{fr} \subseteq \mathsf{mo}$
v $\mathsf{fr}; \mathsf{mo} \subseteq \mathsf{fr}$
vi $\mathsf{fr}; \mathsf{fr} \subseteq \mathsf{fr}$

Proof.

- i) Consider $(x, y) \in \mathsf{rf}$ and $(y, z) \in \mathsf{fr}$. Because rf is one-to-many, $\mathsf{rf}^{-1}(y) = x$. Because $(y, z) \in \mathsf{fr}$, $(\mathsf{rf}^{-1}(y), z) \in \mathsf{mo}$. Therefore, $(x, z) \in \mathsf{mo}$ as required.
- ii) Consider $(x, y) \in \mathsf{rf}$ and $(y, z) \in \mathsf{mo}$. Because y has an incoming rf edge it is a read, because it has an outgoing mo edge, it is a modification, and so y is an update. Thus, by Lemma C.7ii, $(x, y) \in \mathsf{mo}$ and then the result follows by transitivity.
- iii) Consider $(x, y) \in \mathsf{rf}$ and $(y, z) \in \mathsf{rf}$. Because y has both incoming and outgoing rf edges, it is an update. Thus, by Lemma C.7ii, $(x, y) \in \mathsf{mo}$ and then the result is immediate.
- iv) Consider $(x, y) \in \mathsf{mo}$ and $(y, z) \in \mathsf{fr}$. Because y has an incoming mo edge it is a modification, because it has an outgoing fr edge, it is a read, and thus y is an update. Thus, by Lemma C.7i, $(y, z) \in \mathsf{mo}$ and now $(x, z) \in \mathsf{mo}$ by transitivity.
- v) Let $(x, z) \in \mathsf{fr}; \mathsf{mo}$. Thus, there is some $y \neq x$ such that $(x, y) \in \mathsf{rf}^{-1}; \mathsf{mo}$ and $(y, z) \in \mathsf{mo}$. Let w be unique such that $(w, x) \in \mathsf{rf}$ and so $(w, y) \in \mathsf{mo}$. By transitivity of mo we have $(w, z) \in \mathsf{mo}$ and thus $(x, z) \in \mathsf{rf}^{-1}; \mathsf{mo}$. It remains to show that $x \neq z$. Assume otherwise. Then x is an update (because it has both an incoming rf edge and an incoming mo edge). Now, by Lemma C.7i, because $(x, y) \in \mathsf{fr}$, $(x, y) \in \mathsf{mo}$ and thus $(x, z) \in \mathsf{mo}$. But now $x \neq y$ by the irreflexivity of mo.
- vi) Consider $(x, y) \in \mathsf{fr}$ and $(y, z) \in \mathsf{fr}$. Because y has an incoming fr edge it is a modification, and because y has an outgoing fr edge it is a read. Thus, y is an update and by Lemma C.7i, $(y, z) \in \mathsf{mo}$. So now $(x, z) \in \mathsf{fr}; \mathsf{mo}$ so by Property v of this Lemma, $(x, z) \in \mathsf{fr}$.

◻

The next lemma presents a "closed-form" for the eco relation, in which eco is defined without use of a transitive closure, providing a simple set of cases that must be considered when analysing the relation. This is inspired by a similar expression in [20].

Lemma C.9 (eco cases)**.** *For any semi-consistent execution* $(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$*, with update atomicity*

$$\mathsf{eco} = \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr} \cup (\mathsf{mo}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf})$$

Proof. Let

$$\mathsf{eco}' = \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr} \cup (\mathsf{mo}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf})$$

We show that $\mathsf{eco}' = \mathsf{eco}$.

Recall that $\mathsf{eco} = (\mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf})^+$. It is easy to see that $\mathsf{eco}' \subseteq \mathsf{eco}$ (as each option in the union defining eco' is included in one or two steps of eco).

We prove that $\mathsf{eco} \subseteq \mathsf{eco}'$ by induction.

Let p be a (nonempty) path through the transitive closure eco. Thus for each i such that $i + 1 < |p|$ (where $|p|$ is the length of p) $(p_i, p_{i+1}) \in \mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf}$ (indexing from 0). We prove by induction on the length of p that $(p_0, p_{|p|-1}) \in \mathsf{eco}'$, which is sufficient to show that $\mathsf{eco} \subseteq \mathsf{eco}'$. For the base case, p contains two elements, p_0 and p_1, so we must prove that $(p_0, p_1) \in \mathsf{eco}'$. But this follows from the fact that

$$\mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf} \subseteq \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr} \cup (\mathsf{mo}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf})$$

which is clear by inspection. For the induction, assume there is some p' and x such that $p' = p; \langle x \rangle$ and both

$$(p_0, p_{|p|-1}) \in \mathsf{eco}'$$
$$(p_{|p|-1}, x) \in \mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf}$$

We must prove that $(p_0, x) \in \mathsf{eco}'$. It is sufficient to show that

$$\mathsf{eco}'; (\mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf}) \subseteq \mathsf{eco}'$$

But by distributivity of ; over $\cup$, this is equivalent to

$$(\mathsf{eco}'; \mathsf{fr}) \cup (\mathsf{eco}'; \mathsf{mo}) \cup (\mathsf{eco}'; \mathsf{rf}) \subseteq \mathsf{eco}'$$

Expanding the definition of eco' and applying distributivity once again we obtain 15 cases to check: five options in the union defining eco' combined with each of the three relations fr, mo, rf. The cases, and their proofs are as follows:

- $\mathsf{rf}; \mathsf{fr} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{rf}; \mathsf{fr} &\subseteq \mathsf{mo} && \text{by Lemma C.8i} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{rf};\mathsf{mo} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{rf};\mathsf{mo} &\subseteq \mathsf{mo} && \text{by Lemma C.8ii} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{rf};\mathsf{rf} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{rf};\mathsf{rf} &\subseteq \mathsf{mo};\mathsf{rf} && \text{by Lemma C.8iii} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{mo};\mathsf{fr} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{mo};\mathsf{fr} &\subseteq \mathsf{mo} && \text{by Lemma C.8iv} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{mo};\mathsf{mo} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{mo};\mathsf{mo} &\subseteq \mathsf{mo} && \text{by transitivity} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{mo};\mathsf{rf} \subseteq \mathsf{eco}'$. But this is true by definition of eco'.
- $\mathsf{fr};\mathsf{fr} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{fr};\mathsf{fr} &\subseteq \mathsf{fr} && \text{by Lemma C.8vi} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{fr};\mathsf{mo} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{fr};\mathsf{mo} &\subseteq \mathsf{fr} && \text{by Lemma C.8v} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{fr};\mathsf{rf} \subseteq \mathsf{eco}'$. But this is true by definition of eco'.
- $\mathsf{mo};\mathsf{rf};\mathsf{fr} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{mo};\mathsf{rf};\mathsf{fr} &\subseteq \mathsf{mo};\mathsf{mo} && \text{by Lemma C.8i} \\ &\subseteq \mathsf{mo} && \text{by transitivity} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{mo};\mathsf{rf};\mathsf{mo} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{mo};\mathsf{rf};\mathsf{mo} &\subseteq \mathsf{mo};\mathsf{mo} && \text{by Lemma C.8ii} \\ &\subseteq \mathsf{mo} && \text{by transitivity} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{mo};\mathsf{rf};\mathsf{rf} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{mo};\mathsf{rf};\mathsf{rf} &\subseteq \mathsf{mo};\mathsf{mo};\mathsf{rf} && \text{by Lemma C.8iii} \\ &\subseteq \mathsf{mo};\mathsf{rf} && \text{by transitivity} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{fr};\mathsf{rf};\mathsf{fr} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{fr};\mathsf{rf};\mathsf{fr} &\subseteq \mathsf{fr};\mathsf{mo} && \text{by Lemma C.8i} \\ &\subseteq \mathsf{fr} && \text{by Lemma C.8v} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{fr};\mathsf{rf};\mathsf{mo} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{fr};\mathsf{rf};\mathsf{mo} &\subseteq \mathsf{fr};\mathsf{mo} && \text{by Lemma C.8ii} \\ &\subseteq \mathsf{fr} && \text{by Lemma C.8v} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

- $\mathsf{fr};\mathsf{rf};\mathsf{rf} \subseteq \mathsf{eco}'$. But

$$\begin{aligned} \mathsf{fr};\mathsf{rf};\mathsf{rf} &\subseteq \mathsf{fr};\mathsf{mo};\mathsf{rf} && \text{by Lemma C.8iii} \\ &\subseteq \mathsf{fr};\mathsf{rf} && \text{by Lemma C.8v} \\ &\subseteq \mathsf{eco}' && \text{by definition of } \mathsf{eco}' \end{aligned}$$

This completes our proof. □

Lemma C.10 (Weak Canonical RAR Consistency implies eco-irreflexivity)**.** *For a candidate execution*

$$\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo}),$$

if $\mathbb{D}$ *is weakly canonical consistent then* $\mathbb{D}$ *satisfies irrefl*(eco).

Proof. Recall from Lemma C.9 that

$$\mathsf{eco} = \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr} \cup (\mathsf{mo};\mathsf{rf}) \cup (\mathsf{fr};\mathsf{rf})$$

Assume for a contradiction that there is some $(x, x) \in \mathsf{eco}$. There are five cases to consider: one for each option of the union. It cannot be that $(x, x) \in \mathsf{rf}$, $(x, x) \in \mathsf{fr}$, $(x, x) \in \mathsf{mo}$ edges, because all these relations are irreflexive. Thus the pair (x, x) must appear in one of the following: $\mathsf{mo};\mathsf{rf}$ or $\mathsf{fr};\mathsf{rf}$. We treat each case separately.

In the first case, we have $(x, x) \in \mathsf{mo};\mathsf{rf}$ for some $x \in D$. The relation $\mathsf{mo};\mathsf{rf}$ goes from modifying operations to reading operations, so again x must be an update. Let w' be the modification satisfying $(x, w') \in \mathsf{mo}$ and $(w', x) \in \mathsf{rf}$ (the existence of this operation is guaranteed by the definition of relational composition). But now, by Property Lemma C.7i, $(w', x) \in \mathsf{mo}$ and thus $(x, x) \in \mathsf{mo}$ contrary to the irreflexivity of mo.

In the second case, we have $(x, x) \in \mathsf{fr};\mathsf{rf}$. Let w be the modification satisfying $(x, w) \in \mathsf{fr}$ and $(w, x) \in \mathsf{rf}$ (again, the existence of this modification is guaranteed by relational composition). Because

$$\mathsf{fr} = \mathsf{rf}^{-1};\mathsf{mo} \setminus Id$$

there is some modification w' satisfying $(w', x) \in \mathsf{rf}$, $(w', w) \in \mathsf{mo}$ and $w' \neq w$. But because rf is one-to-many $\mathsf{rf}^{-1}(x) = w$ and $\mathsf{rf}^{-1}(x) = w'$, and thus $w = w'$, a contradiction. This completes our proof. □

Lemma C.11. *For a candidate execution*

$$\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo}),$$

irrefl(eco; hb) *is equivalent to the conjunction of* COH *and* RF *(defined in Definition C.3).*

Proof. Let $R = (\mathsf{rf}^{-1})?;\mathsf{mo};\mathsf{rf}^{?}$ so that COH is equivalent to *irrefl*(R; hb). Now, note that

$$\begin{aligned} (\mathsf{rf}^{-1})?;\mathsf{mo};\mathsf{rf}^{?} &= (\mathsf{mo} \cup \mathsf{fr});(\mathsf{rf} \cup Id) && (14) \\ & \quad \text{def of fr, ref. clos.} \\ &= (\mathsf{mo} \cup (\mathsf{mo};\mathsf{rf}) \cup \mathsf{fr} \cup (\mathsf{fr};\mathsf{rf})) && (15) \\ & \quad \text{distrib of ; over } \cup \end{aligned}$$

Therefore, recalling from Lemma C.9 that

$$\mathsf{eco} = \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr} \cup (\mathsf{mo};\mathsf{rf}) \cup (\mathsf{fr};\mathsf{rf})$$

we have eco $= R \cup$ rf. Thus, because ; distributes over $\cup$, eco; hb $= (R;$ hb$) \cup$ (rf; hb), and so *irrefl*(eco; hb) is equivalent to *irrefl*(R; hb $\cup$ rf; hb). But, because *irrefl*($r \cup s$) $\Leftrightarrow$ *irrefl*(r) $\wedge$ *irrefl*(s) for any relations r, s, this is equivalent to the conjunction of COH and RF. □

Lemma C.12 (Weak Canonical RAR Consistency implies eco; hb-irreflexivity)**.** *For a candidate execution*

$$\mathbb{D} = ((D, \mathrm{sb}), \mathrm{rf}, \mathrm{mo}),$$

whenever $\mathbb{D}$ *is weakly canonical consistent, we have that* $\mathbb{D}$ *satisfies irrefl*(eco$^?$; hb).

Proof. First, note that Property HB of weakly canonical consistency ensures that *irrefl*(hb), so if there is a cycle in eco$^?$; hb then there is a cycle in eco; hb (so we must actually take an eco step). We prove that this later is impossible. But by Lemma C.11, because $\mathbb{D}$ satisfies COH and RF we have *irrefl*(eco; hb) as required. □

Lemma C.13 (Coherence implies canonical coherence)**.** *For a candidate execution* $\mathbb{D} = ((D, \mathrm{sb}), \mathrm{rf}, \mathrm{mo})$*, if* $\mathbb{D}$ *satisfies irrefl*(eco$^?$; hb) *then* $\mathbb{D}$ *satisfies all of HB, COH, and RF above.*

Proof. Assume *irrefl*(eco$^?$; hb). Because *irrefl*(eco$^?$; hb) and hb $\subseteq$ eco$^?$; hb we have *irrefl*(hb) as required for HB.

Because *irrefl*(eco$^?$; hb) and eco; hb $\subseteq$ eco$^?$; hb we have *irrefl*(eco; hb). By Lemma C.11, this implies the conjunction of COH and RF. This completes our proof.

□

Lemma C.14 (Coherence implies Update Atomicity)**.** *For a candidate execution* $\mathbb{D} = ((D, \mathrm{sb}), \mathrm{rf}, \mathrm{mo})$*, if* $\mathbb{D}$ *satisfies irrefl*(eco$^?$) *then* $\mathbb{D}$ *satisfies the update atomicity property UPD.*

Proof. By Lemma C.6, UPD is equivalent to *irrefl*(fr; mo) $\wedge$ *irrefl*(rf; mo). But fr; mo $\subseteq$ eco, so because *irrefl*(eco) we have *irrefl*(fr; mo). Likewise, rf; mo $\subseteq$ eco so *irrefl*(rf; mo). This is sufficient to prove UPD. □

The four lemmas C.10, C.12, C.13 and C.13 together imply C.15 can now be used to prove our main theorem.

Theorem C.15. *For any candidate execution*

$$\mathbb{D} = ((D, \mathrm{sb}), \mathrm{rf}, \mathrm{mo}),$$

$\mathbb{D}$ *is weakly canonical consistent iff* $\mathbb{D}$ *satisfies* SC *Coherence of Definition 4.2 on page 7.*

Proof. For the left-to-right direction, see Lemmas C.10 and C.12. For the right-to-left direction, see Lemmas C.13 and C.13. □

D Proof of Peterson's algorithm

A configuration (P, σ) is an *initial configuration of Peterson's algorithm* if σ is an initial state of our semantics and the following conditions hold:

$$P.pc_t = 2 \tag{16}$$

$$wrval(\sigma.last(turn)) \in \{1, 2\} \tag{17}$$

$$wrval(\sigma.last(flag_t)) = false \tag{18}$$

for each $t \in \{1, 2\}$. The last condition here is not strictly necessary for our proof, but it is needed to ensure that Peterson's algorithm makes progress.

Lemma D.1 (Peterson's C11 Invariants)**.** *If* (P, σ) *is a state reachable of Peterson's algorithm, then* (P, σ) *satisfies the following for each* $t, \hat{t} \in \{1, 2\}$.

$$turn \textit{ is an update-only location} \tag{19}$$

$$turn \overset{\sigma}{=}_1 2 \vee turn \overset{\sigma}{=}_2 1 \tag{20}$$

$$P.pc_t \in \{3, 4, 5, 6\} \implies flag_t \overset{\sigma}{=}_t true \tag{21}$$

$$P.pc_t \in \{4, 5, 6\} \implies flag_t \overset{\sigma}{\to} turn \tag{22}$$

$$P.pc_t \in \{4, 5, 6\} \wedge P.pc_{\hat{t}} \in \{4, 5, 6\} \implies flag_{\hat{t}} \overset{\sigma}{=}_t true \vee turn \overset{\sigma}{=}_{\hat{t}} t \tag{23}$$

$$P.pc_t = 5 \wedge P.pc_{\hat{t}} \in \{4, 5, 6\} \implies turn \overset{\sigma}{=}_{\hat{t}} t \tag{24}$$

$$P.pc_t = 2 \implies flag_t \overset{\sigma}{=} false \tag{25}$$

Proof. We first prove that each property holds in the initial configuration. Let (P, σ) be an initial state. Thus, for each t, $\sigma.pc_t = 2$. This is sufficient to show that all of the invariants 21, 22, 23 and 24 are true initially, as these invariants all assume that at least one thread t has $P.pc_t \neq 2$. We show that each remaining invariant holds as follows:

(19) By definition, every location is update-only in an initial state

(20) This follows from Init, and the initial condition 17, $turn \overset{\sigma}{=} 0$ or $turn \overset{\sigma}{=} 1$.

(25) This follows from Init, and the initial condition

$$wrval(\sigma_0.last(flag_t)) = false.$$

We now prove, for each transition that each property is preserved. Fix a transition $(P, \sigma) \overset{m,e}{\Longrightarrow}_{RA} (P', \sigma')$ with (P, σ) satisfying the invariants of Figure D.1. Also, fix a thread t (thus fixing $\hat{t}$), which is the thread executing the operation represented by the transition. For each transition, we prove that each invariant is preserved. Where appropriate, we do so for both t and $\hat{t}$. Invariants applied to $\hat{t}$ are marked with a primed label. We ignore the execution of line 5, as the critical section does not modify the variables used in Peterson's algorithm.

Case 1: $P.pc_t = 2$, and $P'.pc_t = 3$ and $e = W_t(flag_t, true)$. It follows from Lemma 5.6, and invariant 25 that

$$m = \sigma.last(flag_t).$$

(19) [*turn* is an update-only location in σ']. This follows because *turn* is an update-only location in σ, and $e \notin \mathrm{Wr}_{|turn}$.

(20) [$turn \overset{\sigma'}{=}_1 2 \vee turn \overset{\sigma'}{=}_2 1$]. From the rule NoMod, and the fact that $e \notin \mathsf{Wr}_{|turn}$ it follows that if $turn \overset{\sigma}{=}_1 2$ (resp. $turn \overset{\sigma}{=}_2 1$) then $turn \overset{\sigma'}{=}_1 2$ (resp. $turn \overset{\sigma'}{=}_2 1$), which is sufficient to prove that the invariant is preserved.

(21) [$P'.pc_t \in \{3,4,5,6\} \implies flag_t \overset{\sigma'}{=}_t true$]. From rule ModLast, the fact that $e \in \mathsf{Wr}_{|flag_t}$ and that $m = \sigma.last(flag_t)$ it follows that $flag_t \overset{\sigma'}{=} wrval(e) = true$.

(21') [$P'.pc_{\hat{t}} \in \{3,4,5,6\} \implies flag_{\hat{t}} \overset{\sigma'}{=}_{\hat{t}} true$]. From rule NoMod and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}}$, it follows that if $flag_{\hat{t}} \overset{\sigma}{=}_{\hat{t}} true$, then $flag_{\hat{t}} \overset{\sigma'}{=}_{\hat{t}} true$, which is sufficient to prove that the invariant is preserved.

(22) [$P'.pc_t \in \{4,5,6\} \implies flag_t \xrightarrow{\sigma'} turn$]. Similar to the proof for Invariant 21, $P'.pc_t = 2 \notin \{4,5,6\}$.

(22') [$P'.pc_{\hat{t}} \in \{4,5,6\} \implies flag_{\hat{t}} \xrightarrow{\sigma'} turn$]. From rule NoMod-Ord and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}} \cup \mathsf{Wr}_{|turn}$, it follows that if $flag_{\hat{t}} \xrightarrow{\sigma} turn$, then $flag_{\hat{t}} \xrightarrow{\sigma'} turn$, which is sufficient to prove that the invariant is preserved.

(23) [$P'.pc_t \in \{4,5,6\} \wedge P'.pc_{\hat{t}} \in \{4,5,6\} \implies flag_{\hat{t}} \overset{\sigma'}{=}_t true \vee turn \overset{\sigma'}{=}_{\hat{t}} t$]. As before, it is sufficient that $P'.pc_t \notin \{4,5\}$.

(23') [$P'.pc_{\hat{t}} \in \{4,5,6\} \wedge P'.pc_t \in \{4,5,6\} \implies flag_t \overset{\sigma'}{=}_{\hat{t}} true \vee turn \overset{\sigma'}{=}_t t$]. It is again sufficient that $P'.pc_t \notin \{4,5,6\}$.

(24) [$P'.pc_t = 5 \wedge P'.pc_{\hat{t}} \in \{4,5,6\} \implies turn \overset{\sigma'}{=}_{\hat{t}} t$]. It is sufficient that $P'.pc_t \neq 5$.

(24') [$P'.pc_{\hat{t}} = 5 \wedge P'.pc_t \in \{4,5,6\} \implies turn \overset{\sigma'}{=}_t t$]. It is again sufficient that $P'.pc_t \notin \{4,5,6\}$.

(25) [$P.pc_t = 2 \implies flag_t \overset{\sigma}{=} false$]. It is sufficient that $P'.pc_t = 3$.

(25') $P.pc_{\hat{t}} = 2 \implies flag_{\hat{t}} \overset{\sigma}{=} false$. From rule NoMod and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}}$, it follows that if $flag_{\hat{t}} \overset{\sigma}{=} false$, then $flag_{\hat{t}} \overset{\sigma'}{=} false$, which is sufficient to prove that the invariant is preserved.

For the remaining cases, we do not explicitly state the invariant that we are proving. The mapping from labels to invariants remains as above.

Case 2: $P.pc_t = 3$, and $P'.pc_t = 4$ and $e = U_t(turn, v, \hat{t})$ for some v. By Lemma 5.6, because e is an update and $turn$ is an update-only location, $m = \sigma.last(turn)$.

(19) This follows because e is an update.

(20) From the rule ModLast, and that $e \in \mathsf{Wr}_{|turn}$ and $m = \sigma.last(turn)$ it follows that $turn \overset{\sigma'}{=}_t wrval(e) = \hat{t}$, which is sufficient.

(21) Note that by Invariant 21 applied to (P, σ), we have $flag_t \overset{\sigma}{=}_t true$. Then, from the rule NoMod, and the fact that $e \notin \mathsf{Wr}_{|flag_t}$ it follows that $flag_t \overset{\sigma'}{=}_t true$ as required.

(21') From rule NoMod and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}}$, it follows that if $flag_{\hat{t}} \overset{\sigma}{=}_{\hat{t}} true$, then $flag_{\hat{t}} \overset{\sigma'}{=}_{\hat{t}} true$, which is sufficient to prove that the invariant is preserved.

(22) Note that by Invariant 21 applied to (P, σ), we have $flag_t \overset{\sigma}{=}_t true$. Then, from the rule WriteOrd, and the fact that $turn$ and $flag_t$ are distinct variables, $e \in \mathsf{Wr}_{|turn}$ and $m = \sigma.last(turn)$, we have $flag_t \xrightarrow{\sigma'} turn$ as required.

(22') Note first that because $turn$ is update only, m is an update and thus $m \in \mathsf{Wr}_{R|turn}$. Then, from rule UpdOrd and the fact that $e \in \mathsf{U}_{|turn}$ it follows that if $flag_{\hat{t}} \xrightarrow{\sigma} turn$, then $flag_{\hat{t}} \xrightarrow{\sigma'} turn$, which is sufficient to prove that the invariant is preserved.

(23) In the proof that this transition preserves Invariant 20, we proved that $turn \overset{\sigma'}{=}_t wrval(e) = \hat{t}$, which is sufficient to prove that this current invariant is preserved.

(23') Again, we know that $turn \overset{\sigma'}{=}_t wrval(e) = \hat{t}$, which is sufficient to prove that this invariant is preserved (bearing in mind that t and $\hat{t}$ are transposed in this invariant).

(24) It is sufficient that $P'.pc_t \neq 5$.

(24') Again, the fact that $turn \overset{\sigma'}{=}_t wrval(e) = \hat{t}$ is enough.

(25) It is sufficient that $P'.pc_t \neq 2$.

(25') From rule NoMod and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}}$, it follows that if $flag_{\hat{t}} false$, then $flag_{\hat{t}} \overset{\sigma'}{=}_{\hat{t}} false$, which is sufficient to prove that the invariant is preserved.

Case 3: In this case, we consider the first test at line 4 $flag_t = false$. If this test returns $true$, then nothing about the state changes except that t moves to the second test in the condition. Because nothing about the state is changing, application of the rules NoMod and NoModOrd can be used to show that all the invariants are preserved in a standard way. Therefore, we only consider in detail the situation when the test returns $false$. Thus, assume that $P.pc_t = 4$, and $P'.pc_t = 5$, and $e = R_t(flag_{\hat{t}}, false)$.

Because e is not a write and the value of $pc_{\hat{t}}$ does not change, it is straightforward to use the rules NoMod and NoModOrd to show that each invariant except for 24 and 24' are preserved. Because it is simpler, we first prove that 24' is preserved. Briefly, $P'.pc_{\hat{t}} = P.pc_{\hat{t}}$, and because $e \notin \mathsf{Wr}_{|flag_t}$ we have $flag_t \overset{\sigma}{=}_{\hat{t}} true \implies flag_t \overset{\sigma'}{=}_{\hat{t}} true$ (by rule NoMod), and because $e \notin \mathsf{Wr}_{|turn}$ we have $turn \overset{\sigma}{=}_{\hat{t}} \hat{t} \implies turn \overset{\sigma'}{=}_{\hat{t}} \hat{t}$. These three properties are sufficient to show that 24' is preserved.

We now prove that 24 is preserved. We do so by proving that $turn \overset{\sigma'}{=}_{\hat{t}} t$ under the assumption that $P'.pc_{\hat{t}} \in \{4,5,6\}$. Because $P.pc_{\hat{t}} = P'.pc_{\hat{t}}$, we have $P.pc_{\hat{t}} \in \{4,5,6\}$. Thus, because $P.pc_t = 4$, Invariant 23 guarantees that

$$flag_{\hat{t}} \overset{\sigma}{=}_t true \vee turn \overset{\sigma}{=}_{\hat{t}} = t$$

But the disjunct $flag_{\hat{t}} \overset{\sigma}{=}_t$ must be false. If it were true, Lemma 5.3 the read e would have to return *true*, contrary to the hypothesis that $e = R_t(flag_{\hat{t}}, false)$. Thus $turn \overset{\sigma}{=}_{\hat{t}} = t$. Then, from rule NoMod, and the fact that e is not a write, we have $turn \overset{\sigma'}{=}_{\hat{t}}\ t$ as required.

Case 4: In this case, we consider the second test at line 4 $turn = \hat{t}$. As before, if this test returns true, then all the invariants are straight-forwardly preserved. So assume that $P.pc_t = 4$, and $P'.pc_t = 5$, and $e = R_t(turn, t)$.

Again, because e is not a write and the value of $pc_{\hat{t}}$ does not change, it is easy to show that each invariant except for 24 is preserved. We show that Invariant 24 is preserved by proving that $turn \overset{\sigma'}{=}_{\hat{t}}\ t$. By Lemma 5.3, and the fact that $e = R_t(turn, t)$ the assertion $turn \overset{\sigma}{=}_t\ \hat{t}$ is false. Thus, by Invariant 20, $turn \overset{\sigma}{=}_{\hat{t}}\ t$. Then, from rule NoMod, and the fact that e is not a write, we have $turn \overset{\sigma'}{=}_{\hat{t}}\ t$ as required.

Case 5: $P.pc_t = 6$, and $P'.pc_t = 2$ and $e = W_t(flag_t, false)$. It follows from Lemma 5.6, and Invariant 21 that

$$m = \sigma.last(flag_t).$$

(19) This invariant is preserved because $e \notin \mathsf{Wr}_{|turn}$.
(20) This invariant is preserved by rule NoMod and the fact that $e \notin \mathsf{Wr}_{|turn}$.
(21) Note that $P'.pc_t \notin \{3, 4, 5, 6\}$, which is sufficient to show that this invariant is preserved.
(21') This invariant is preserved by rule NoMod and the fact that $e \notin \mathsf{Wr}_{flag_{\hat{t}}}$.
(22) Note that $P'.pc_t \notin \{4, 5\}$, which is sufficient to show that this invariant is preserved.
(22') This invariant is preserved by rule NoMod and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}} \cup \mathsf{Wr}_{|turn}$.
(23) Note that $P'.pc_t \notin \{4, 5, 6\}$, which is sufficient to show that this invariant is preserved.
(23') Again, it is sufficient to note that $P'.pc_t \notin \{4, 5, 6\}$ (bearing in mind that t and $\hat{t}$ are transposed in 23').
(24) This invariant is preserved by rule NoMod and the fact that $e \notin \mathsf{Wr}_{|turn}$.
(24') This invariant is preserved by rule NoMod and the fact that $e \notin \mathsf{Wr}_{|turn}$.
(25) From rule ModLast and the fact that $e \in \mathsf{Wr}_{|flag_t}$, $m = \sigma.last(flag_t)$ and $wrval(e) = false$, we have $flag_t \overset{\sigma'}{=}_t$ *false* as required.
(25') This invariant is preserved by rule NoMod and the fact that $e \notin \mathsf{Wr}_{|flag_{\hat{t}}}$.

This completes our proof. □

E Mechanisation in MemAlloy

The `.cat` files for MemAlloy

https://github.com/johnwickerson/memalloy

are given below.

- `c11_rar.cat` contains our formalisation of the RAR fragment.
- `c11_simp_2.cat` is the same as `c11_simp.cat`, distributed with MemAlloy, but imports `c11_base_rar.cat` instead of `c11_base.cat`.
- `c11_base_rar.cat` contains definitions common to both `c11_rar.cat` and `c11_simp_2.cat`. It is essentially the file distributed with MemAlloy, but with a simplified sw relation that ignores release sequences. It also omits events such as SC events that are not part of our C11 model.

No differences were found between `c11_rar.cat` and `c11_simp_2.cat` for models up to size 7.

As a sanity check, both `c11_rar.cat` and `c11_simp_2.cat` were compared against `c11_lidbury.cat`, which formalises Lidbury and Donaldson [22], revealing the exact same sets of counterexamples.

E.1 File `c11_rar.cat`

```
(* This file imports c11_rar_base.cat and
rephrases the acyclicity axiom in terms of eco *)

"C"
include "c11_rar_base.cat"

let eco = (rf | co | fr)+

irreflexive hb as hb_irr
irreflexive hb ; eco as hb_eco_irr
irreflexive eco as eco_irr
```

E.2 File `c11_simp_2.cat`

```
(* This is the C11 file distributed with
Memalloy, but imports c11_rar_base.cat
instead of c11_base.cat *)

"C"
include "c11_rar_base.cat"

acyclic scp as Ssimp
```

E.3 File c11_base_rar.cat

```
"C"
include "basic.cat"

(* Modifications to c11_base.cat *)

(* synchronises with (sw) simplified to
elide release sequences *)
let sw = [REL]; rf; [ACQ]

empty [SC] as omitSC
```

```
empty [NAL] as omitNAL
empty [F] as omitF
empty [R & W] \ [REL & ACQ] as RAOnlyRMW

(* Definitions below are from c11_base.cat *)

let fsb = [F]; po
let sbf = po; [F]

(* release sequence *)
let rs = poloc*; rf*

(* happens before *)
let hb = (po | sw)+
let hbl = hb & loc

(* conflict *)
let cnf = (((W*M) | (M*W)) & loc) \ id

(* data race *)
let dr = (cnf \ (A*A)) \ thd \ (hb | hb^-1)
undefined_unless empty dr as Dr

(* unsequenced race *)
let ur = (cnf & thd) \ (po | po^-1)
undefined_unless empty ur as Ur

(* coherence, etc *)
acyclic hbl | rf | co | fr as HbCom

(* no "if(r==0)" *)
deadness_requires empty if_zero as No_If_Zero

(* no unsequenced races *)
deadness_requires empty ur as Dead_Ur

(* coherence edges are forced *)
deadness_requires empty unforced_co as Forced_Co

(* external control dependency *)
let cde = ((rf \ thd) | ctrl)* ; ctrl
(* dependable release sequence *)
let drs = rs \ ([R]; !cde)
(* dependable synchronises-with *)
let dsw =
sw & (((fsb?; [REL]; drs?) \ (!ctrl; !cde)) ; rf)

(* dependable happens-before *)
let dhb = po?; (dsw;ctrl)*
(* self-satisfying cycle *)
let ssc = id & cde
(* potential data race *)
let pdr = cnf \ (A*A)
(* reads-from on non-atomic location *)
let narf = rf & (NAL*NAL)

deadness_requires
empty pdr \ (dhb | dhb^-1 | narf;ssc | ssc;narf^-1)
as Dead_Pdr

let scb = fsb?; (co | fr | hb); sbf?
let scp = (scb & (SC * SC)) \ id
```