Reasoning about Promises in Weak Memory Models with Event Structures*

Heike Wehrheim[0000−0002−2385−7512][1], Lara Bargmann[1], and Brijesh Dongol[0000−0003−0446−3507][2]

[1] University of Oldenburg, Oldenburg, Germany
[2] University of Surrey, Guildford, UK

Abstract. Modern processors such as ARMv8 and RISC-V allow executions in which independent instructions within a process may be reordered. To cope with such phenomena, so called *promising* semantics have been developed, which permit threads to read values that have not yet been written. Each promise is a speculative update that is later validated (fulfilled) by an actual write. Promising semantics are operational, providing a pathway for developing proof calculi. In this paper, we develop an incorrectness-style logic, resulting in a framework for reasoning about state reachability. Like incorrectness logic, our assertions are *underapproximating*, since the set of all valid promises are not known at the start of execution. Our logic uses *event structures* as assertions to compactly represent the ordering among events such as promised and fulfilled writes. We prove soundness and completeness of our proof calculus and demonstrate its applicability by proving reachability properties of standard weak memory litmus tests.

Keywords: Weak memory models, promises, event structures, incorrectness logic

1 Introduction

In recent years, numerous works have looked into semantics for weak memory models for various hardware architectures or languages, e.g. for x86-TSO [34], C11 [2, 25], Power [33] or ARM [15]. Such semantics typically can be classified as either being declarative (aka axiomatic) or operational. Operational semantics furthermore can be divided into those following a microarchitectural style (providing formalizations of the actual hardware architecture) and those trying to abstract from architectures. Most notably, *view-based* semantics [13, 20, 29] avoid modelling specific hardware components and instead define the semantics in terms of *views* of thread on the shared state. *Promises* [21, 23] are employed in operational semantics as a way of capturing out-of-order writes while still

* Wehrheim and Bargmann are supported by DFG-WE2290/14-1. Dongol is supported by EPSRC grants EP/V038915/1, EP/R032556/1, EP/R025134/2, VeTSS and ARC Discovery Grant DP190102142.

executing operations in thread order. A promise (w.r.t. a value κ and a shared location x) of a thread τ states that τ will eventually write value κ onto location x. All promised writes then need to be *fulfilled* (i.e., justified) in the future of a program run, but other threads can read from promises before they are fulfilled.

Our interest here is the development and use of Hoare-style [17] structural proof calculi (and their extensions to concurrency by Owicki and Gries [27]) for weak memory models. Owicki-Gries-like proof calculi have been proposed by a number of researchers [10, 11, 22, 40], and have also recently been given for non-volatile memory [3, 31]. Svendsen et al. [35] have developed a separation logic for promises for the C11 memory model. Wright et al. [40] have developed an Owicki-Gries proof system for out-of-order writes (as allowed by promises), but rely on pre-processing via the denotational MRD framework [28].

All of these proposals follow Hoare's principle of providing *safety* proofs. In particular, a Hoare triple $\{p\}S\{q\}$ describes the fact that an execution of program S starting in a state satisfying p is either non-terminating, or terminates in a state satisfying q (over-approximating the final states). However, for weak memory models, we often want to prove *reachability*, i.e. under-approximate the set of final states, like in the recent proposal of O'Hearn's incorrectness logic [26]. Here, a triple $[p]\,S\,[q]$ describes the possibility of program S reaching all states satisfying q when started in a state satisfying p. A verification technique supporting these *reachability* triples enables one to reason about executions that deviate from the expected sequentially consistent behaviour of concurrent programs.

Contributions. In this paper, we present a reachability proof calculus for concurrent programs where the semantics of the weak memory model is based on promises. The specific challenges therein lay in (i) capturing the meaning of promises as writes which will only happen in the future but can nevertheless already be read from, and (ii) appropriately describing the required ordering (and concurrency) between promises and fulfills as fixed by the concurrent program under consideration. We address these challenges via the following contributions. (1) We develop a program logic based on assertions which are (flow) *event structures* [5, 16, 39], employing parallel composition of event structure and synchronization as a means of determining whether all promises read from have eventually been fulfilled. (2) We extend the theory of flow event structures with the notion of a flow label to capture the behaviours observed in weak memory models. (3) We develop the first *compositional* proof rule for a concurrent reachability (incorrectness) logic. (4) We prove soundness and completeness of this novel event-structure based proof calculus. (5) Finally, we demonstrate its applicability on a number of litmus tests.

Overview. In Section 2, we provide a concrete overview via a motivating example and in Section 3, we present the memory model that we use. Our model is a simplified (strengthened) version of the ARMv8/RISC-V semantics of Pulte et al [29]. In Section 4, we present an extended theory for event structures (specifically an extension of flow event structures) that has been designed to enable reasoning about relaxed memory models. We describe our reasoning methodology and provide examples verifying common litmus tests in Section 5.

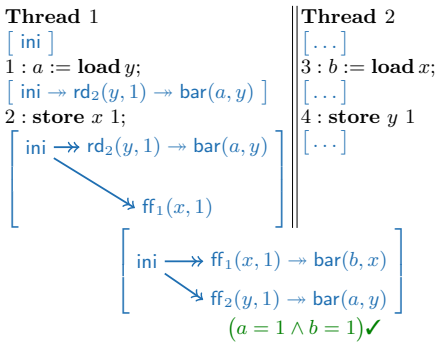


Fig. 1. Reachability for load buffering

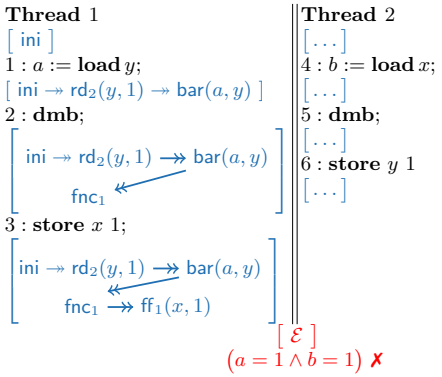


Fig. 2. Load buffering with barriers

2 Motivating Examples

Consider the program in Fig. 1, which describes the load buffering litmus test. Thread 1 (similarly thread 2) loads the value of y (sim. x) into register a (sim. b), then updates x (sim. y) to 1. Since there are no dependencies between lines 1 and 2, and similarly between lines 3 and 4, architectures such as ARMv8 and RISC-V allow the stores in both threads to be reordered with the loads. Thus the program allows the final outcome $a = 1 \wedge b = 1$.

This phenomenon is captured by promising semantics by allowing each thread to "promise" their respective stores, then later fulfilling them. In the meantime, other threads may read from promised writes. Our assertions within a thread reflect this semantics via assertions $\mathcal{E}$ which are *flow event structures* [39]. The events and their partial order reflect program executions, and in particular describe the various *views* which threads have on shared state.

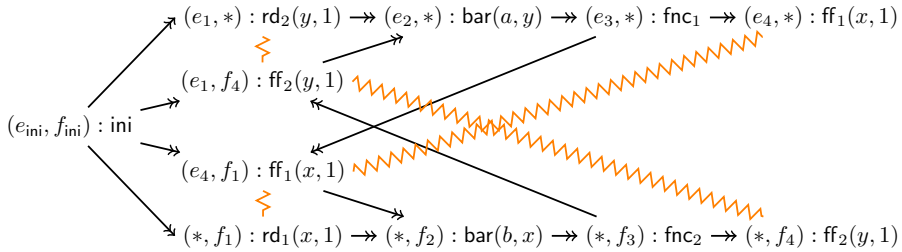
The proof outlines (i.e., program texts with assertions) of individual threads may first of all contain read events for arbitrary promises, i.e. describe the reading of arbitrary values. In Thread 1 of Fig. 1, the pre-assertion of the load only contains an event for initial writes (labelled ini), yet the load may read the value 1 for y from a promised write, described by the event labelled $\mathsf{rd}_2(y, 1)$ in the post assertion. The semantics generates dependencies if the same register is used (perhaps indirectly) by a read and a later write. This is captured in our assertions using the event labelled $\mathsf{bar}(a, y)$, causally ordered after $\mathsf{rd}_2(y, 1)$, which states that the view of register a is at least that of the read of y. Execution of line 2 then adds a fulfill event with label $\mathsf{ff}_1(x, 1)$ to the assertion, which is not ordered with any other event except ini. Symmetric assertions can be generated for Thread 2. To obtain an assertion describing the combined execution, we compose the final event structures of both threads to obtain a "postcondition" of the program. For this, we use parallel composition of event structures, synchronising read with their corresponding fulfill events. In Fig. 1, both reads are valid since the promises that these reads rely on can be fulfilled in the composition without creating cyclic dependencies.

Fig. 2 presents a variation of the program in Fig. 1, which includes additional barriers **dmb** (fences) between the load and store in each thread, preventing their reordering. Again we build a proof outline for an execution in which Thread 1 loads 1 into a, obtaining the assertions shown. Note that here the event structure contains an additional fence event, fnc, that is ordered after $\mathsf{bar}(a, y)$ and before $\mathsf{ff}_1(x, 1)$. Similarly, for Thread 2 loading 1 into b, we would obtain a symmetric set of assertions. Here, the parallel composition of local assertions is however not *interference free* (see below): the promises that threads 1 and 2 have read from cannot be fulfilled in this concurrent program. More detailedly, let $\mathcal{E}_1$ and $\mathcal{E}_2$ below be the (final) event structures of threads 1 and 2, respectively, where $\twoheadrightarrow$ arrow denotes ordering and we now give event names together with labels.

$\mathcal{E}_1$: $e_{\mathsf{ini}} : \mathsf{ini} \twoheadrightarrow e_1 : \mathsf{rd}_2(y, 1) \twoheadrightarrow e_2 : \mathsf{bar}(a, y) \twoheadrightarrow e_3 : \mathsf{fnc}_1 \twoheadrightarrow e_4 : \mathsf{ff}_1(x, 1)$

$\mathcal{E}_2$: $f_{\mathsf{ini}} : \mathsf{ini} \twoheadrightarrow f_1 : \mathsf{rd}_1(x, 1) \twoheadrightarrow f_2 : \mathsf{bar}(b, x) \twoheadrightarrow f_3 : \mathsf{fnc}_2 \twoheadrightarrow f_4 : \mathsf{ff}_2(y, 1)$

To reason about the set of reachable final states of the concurrent program, we again construct the parallel composition of $\mathcal{E}_1$ and $\mathcal{E}_2$ (denoted $\mathcal{E}_1 \| \mathcal{E}_2$):



This composition of event structure is built similar to [16], allowing events of the parallel composition to be lifted from the sub-components. These are events of the form $(e_i, *)$ and $(*, f_i)$. The parallel composition also contains *synchronised* read/fulfill events, e.g., (e_1, f_4) depicts a read synchronised with the fulfill (write) $\mathsf{ff}_1(y, 1)$. We inherit order in the composition from the constituent event structures. Moreover, to prevent the same event occurring more than once in an "execution" of $\mathcal{E}_1 \| \mathcal{E}_2$, we use the *conflict* relation (zigzagged line). Thus, the synchronised event (e_1, f_4) conflicts with both $(e_1, *)$ and $(*, f_4)$.

The final step in proving is the generation of a valid interference free *configuration* of the parallel composition, which is a subset of the event structure satisfying certain conditions, including acyclicity of $\twoheadrightarrow$, absence of conflicts and absence of unsynchronised reads (ensuring the fulfillment of all promises read from). It turns out that for the event structure above, it is impossible to generate such a configuration. The event $(e_1, *)$ cannot be included since it is an unsynchronised read. Therefore, (e_1, f_4) must be included. However, by the definition of a configuration, this also means that the downclosure of (e_1, f_4) must be included, which results in a cycle: $(e_1, f_4) \twoheadrightarrow (e_2, *) \twoheadrightarrow (e_3, *) \twoheadrightarrow (e_4, f_1) \twoheadrightarrow (*, f_2) \twoheadrightarrow (*, f_3) \twoheadrightarrow (e_1, f_4)$. Since $\mathcal{E}_1 \| \mathcal{E}_2$ has no interference free configurations, the proof outline is not valid and in fact, a final state with $a = 1 \wedge b = 1$ is unreachable here.

3 A Weak Memory Semantics with Promises

We develop a promising semantics inspired by the recent view-based operational semantics by Pulte et al. [29]. We have reduced architecture-specific details, allowing us to focus on the interaction between promises and thread views. Our notion of a *promise* coincides with earlier works [21,23,29]. Threads can promise to write certain values on shared locations and other threads can read from this promise even before the actual write has occurred. All promises however need to be fulfilled at the end of the program execution.

Syntax. Let $x, y \in \textsc{Loc}$ be the set of *shared locations*, $\kappa \in \textsc{Val}$ the set of *values*, $\tau \in \textsc{Tid}$ the set of *thread identifiers* and $a, b \in \textsc{Reg}$ *local registers*. Our sequential language encompasses the following constructs:

$$rv ::= \kappa \mid a \qquad st ::= \mathbf{skip} \mid a := \mathbf{load}\, x \mid \mathbf{store}\ x\ rv \mid a := \eta \mid \mathbf{dmb}$$
$$S ::= st \mid S; S \mid \mathbf{asm}\, \beta \mid S + S \mid S^*$$

where $\eta \in \textsc{Exp}$ is an arithmetic and $\beta \in \textsc{BExp}$ is a boolean expressions, both over (local) registers only. We assume $S^* = \exists n \in \mathbb{N}.\ S^n$, where $S^0 \mathrel{\widehat{=}} \mathbf{skip}$ and $S^n \mathrel{\widehat{=}} S; S^{n-1}$. We use abbreviations: $\mathbf{while}\, \beta\, \mathbf{do}\, S = (\mathbf{asm}\, \beta; S)^*; \mathbf{asm}\, \neg\beta$ and $\mathbf{if}\, \beta\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2 = (\mathbf{asm}\, \beta; S_1) + (\mathbf{asm}\, \neg\beta; S_2)$, where $\mathbf{asm}\, \beta$ is a command that tests whether β holds.

Timestamped state. We let *TState* be the set of all *timestamped states* and *Memory* the set of all memory states, both of which we make more precise below. A *thread* $T \in \mathit{Thread}$ is an element of $S \times \mathit{TState}$, a *concurrent program* is a mapping $\boldsymbol{T} \in \mathit{TPool} \mathrel{\widehat{=}} \textsc{Tid} \to \mathit{Thread}$ and a *concurrent program state* is a pair $\langle \boldsymbol{T}, M \rangle \in \mathit{TPool} \times \mathit{Memory}$. We let $R(\tau)$, $\tau \in \textsc{Tid}$, be the set of registers occurring in the program of $\boldsymbol{T}(\tau)$. We assume $R(\tau) \cap R(\tau') = \emptyset$ whenever $\tau \neq \tau'$.

Threads will make *promises* for writes at particular timestamps. Timestamps $t \in \mathbb{T}$ are natural numbers. We define $t \sqcup t' \mathrel{\widehat{=}} \mathit{max}(t, t')$ and generalise this to sets of timestamps using $\bigsqcup_{t \in T} t$, where $\bigsqcup_{t \in \emptyset} t = 0$. A *memory* is a sequence of write messages of type $\mathsf{Wr} \mathrel{\widehat{=}} (\textsc{Loc} \times \textsc{Val} \times \textsc{Tid}) \cup \{\mathsf{ini}\}$, where ini is a special write message denoting initialisation. The position of a write in the sequence fixes its timestamp. We assume all variables are initialised with value 0.

We denote a write $w \mathrel{\widehat{=}} (x, \kappa, \tau)$ using $\langle x := \kappa \rangle_\tau$ and let $w.loc = x, w.val = \kappa$ and $w.tid = \tau$. For a memory M and thread $\tau \in \textsc{Tid}$, we let $M_\tau \subseteq \mathbb{T}$ be the set of timestamps of entries of τ in M, i.e. $\{t \in \mathbb{T} \mid M(t).tid = \tau\}$. M_τ is used to determine the promise set of each τ. We write $\mathit{tids}(M)$ to denote the set of threads with entries in M. New messages w are appended at the end of the memory, which we write as $M +\!\!+ w$.

A thread state $ts \in \mathit{TState}$ consists of the following components: a set of (non-fulfilled) *promises* $\mathit{prom} \in 2^{\mathbb{T}}$, a *coherence* view of each location, $\mathit{coh} : \textsc{Loc} \to \mathbb{T}$, the value and view of each register, $\mathit{regs} : \textsc{Reg} \to \textsc{Val} \times \mathbb{T}$, a read view $v_{\mathit{read}} : \mathbb{T}$, two write views $v_{\mathit{wOld}}, v_{\mathit{wNew}} : \mathbb{T}$ and a condition view $v_C : \mathbb{T}$. We write $\mathit{regs}(a)$ as $\kappa @ v$ and also let v_a be this view v of register a. Finally, the evaluation of an expression η with respect to a register assignment regs, $[\![\eta]\!]_{\mathit{regs}} \in \textsc{Val} \times \mathbb{T}$, is

defined as follows:

$$[\![\kappa]\!]_{regs} \mathrel{\widehat{=}} \kappa@0 \text{ for } \kappa \in \textsc{Val}, \qquad [\![a]\!]_{regs} \mathrel{\widehat{=}} regs(a) \text{ for } a \in \textsc{Reg},$$
$$[\![\eta_1\, op\, \eta_2]\!]_{regs} \mathrel{\widehat{=}} (\kappa_1 [\![op]\!] \kappa_2)@(v_1 \sqcup v_2) \text{ with } [\![\eta_1]\!]_{regs} = \kappa_1@v_1, [\![\eta_2]\!]_{regs} = \kappa_2@v_2$$

Note that this evaluation is with respect to the register function *regs* and this calculates both the value of the expression and the maximal view of the registers within the expression.

To define the initial state of a program, we let

$$M_{\mathsf{ini}} \mathrel{\widehat{=}} \langle \mathsf{ini} \rangle \qquad ts_{\mathsf{ini}} \mathrel{\widehat{=}} \begin{bmatrix} prom = \{\}, v_{read} = v_{wNew} = v_{wOld} = v_C = 0, \\ coh = (\lambda x.\ 0), regs = (\lambda a.\ 0@0) \end{bmatrix}$$

where ts_{ini} is a record initialising the promises to the empty set, each view to 0, the coherence function to a map from locations to timestamp 0, and the register function to a map from registers to value 0 with timestamp 0. We say that a program $\boldsymbol{T}$ is locally in its initial state iff for each thread τ, we have $\pi_2(\boldsymbol{T}(\tau)) = ts_{\mathsf{ini}}$, where π_i projects the ith component of a tuple. Given that $\boldsymbol{T}$ is in its initial state, the initial concurrent program state is given by $\langle \boldsymbol{T}, M_{\mathsf{ini}} \rangle$.

The rules of the operational semantics (except for standard rules for program constructs) are given in Fig. 3. The two key rules are the Read and Fulfill rule. Read identifies a timestamp t to read a value for x from such that in between t and the maximum of read view and coherence of x, there are no further promises to x in memory M. It updates read view, coherence of x and the view of the register involved in the load as to ensure preservation of dependencies. Fulfill fulfills an already made promise (to write κ to x) of a thread at timestamp t, and to this end has to ensure that views $v_{wNew}, v_C, coh(x)$ as well as that of the value/register are less than t. It removes t from the thread's promise set and updates $coh(x)$ and v_{wOld} (as to ensure dependencies with fences). Rule Promise simply adds an arbitrary new promise at the end of memory. Fence ensures views v_{read} and v_{wNew} are updated. This rule for instance guarantees that store operations separated by barriers **dmb** can only be fulfilled in that order, i.e. the write of the first store cannot be promised to happen later than the write of the second store (more precisely, such promises cannot be fulfilled).

Finally, we say that $\langle T, M \rangle$ is *certifiable* (used in Program Step) if there is some T', M' such that $\langle T, M \rangle \rightarrow_\tau^* \langle T', M' \rangle$ and $T'.prom = \emptyset$. Certifiability ensures that a concurrent program can only make steps when all promises can eventually be fulfilled. Like [29], in our semantics, all promise steps can be done at the beginning without losing any of the reachable states.

4 Event Structures

Event structures [4, 5, 16, 39] are models of concurrent systems which compactly represent (concurrent) executions. Here, we use flow event structures because of their ease in defining a compositional parallel composition [16].

$$\textsc{Promise}\quad \frac{w.tid = \tau \quad w.loc = x \quad w.val = \kappa \quad t = |M| + 1 \quad ts' = ts[prom \mapsto ts.prom \cup \{t\}]}{\langle\langle S, ts\rangle, M\rangle \xrightarrow{prm(x,\kappa)}_\tau \langle\langle S, ts'\rangle, M +\!\!+ w\rangle}$$

$$\textsc{Fence}\quad \frac{v = ts.v_{read} \sqcup ts.v_{wOld} \quad ts' = ts\begin{bmatrix} v_{read} \mapsto v \\ v_{wNew} \mapsto v \end{bmatrix}}{\langle\langle \mathbf{dmb}, ts\rangle, M\rangle \xrightarrow{\mathsf{fnc}}_\tau \langle\langle \mathbf{skip}, ts'\rangle, M\rangle}$$

$$\textsc{Read}\quad \frac{M(t).loc = x \quad M(t).val = \kappa \quad \forall t'.\ t < t' \leq (ts.v_{read} \sqcup ts.coh(x)) \Rightarrow M(t').loc \neq x \quad v_{post} = ts.v_{read} \sqcup t \quad ts' = ts\begin{bmatrix} regs(a) \mapsto \kappa@v_{post}, \\ coh(x) \mapsto ts.coh(x) \sqcup v_{post}, \\ v_{read} \mapsto v_{post} \end{bmatrix}}{\langle\langle a := \mathbf{load}\, x, ts\rangle, M\rangle \xrightarrow{rd(x,\kappa)}_\tau \langle\langle \mathbf{skip}, ts'\rangle, M\rangle}$$

$$\textsc{Fulfill}\quad \frac{t \in ts.prom \quad [\![rv]\!]_{ts.regs} = \kappa@v_{rv} \quad M(t) = \langle x := \kappa\rangle_\tau \quad ts.v_{wNew} \sqcup ts.v_C \sqcup ts.coh(x) \sqcup v_{rv} < t \quad ts' = ts\begin{bmatrix} prom \mapsto ts.prom \setminus \{t\}, \\ coh(x) \mapsto t, \\ v_{wOld} \mapsto v_{wOld} \sqcup t \end{bmatrix}}{\langle\langle \mathbf{store}\ x\ rv, ts\rangle, M\rangle \xrightarrow{ff(x,\kappa)}_\tau \langle\langle \mathbf{skip}, ts'\rangle, M\rangle}$$

$$\textsc{Register}\quad \frac{regs(a) = \kappa_a@u \quad [\![\eta]\!]_{ts.regs} = \kappa@v \quad ts' = ts\left[\, regs(a) \mapsto \kappa@(u \sqcup v)\,\right]}{\langle\langle a := \eta, ts\rangle, M\rangle \xrightarrow{lst(a,\eta)}_\tau \langle\langle \mathbf{skip}, ts'\rangle, M\rangle}$$

$$\textsc{Assume}\quad \frac{[\![\beta]\!]_{ts.regs} = \mathsf{true}@v \quad ts' = ts[v_C \mapsto ts.v_C \sqcup v]}{\langle\langle \mathbf{asm}\,\beta, ts\rangle, M\rangle \xrightarrow{asm(\beta)}_\tau \langle\langle \mathbf{skip}, ts'\rangle, M\rangle}$$

$$\textsc{Program Step}\quad \frac{\langle \boldsymbol{T}(\tau), M\rangle \xrightarrow{op}_\tau \langle T', M\rangle \quad \langle T', M\rangle \text{ certifiable}}{\langle \boldsymbol{T}, M\rangle \xrightarrow{op}_\tau \langle \boldsymbol{T}[\tau \mapsto T'], M'\rangle}$$

Fig. 3. Operational semantics (Atomic statement rules)

Notation. Event structures consist of sets of *events* $d, e, f \in E$. Events will be labelled with *actions* which are here specific to our usage and give us information about program executions:

$$Act^x \mathrel{\widehat{=}} \bigcup_{\tau \in \textsc{Tid}, \kappa \in \textsc{Val}} \{\mathsf{rd}_\tau(x,\kappa), \mathsf{ff}_\tau(x,\kappa)\} \cup \{\mathsf{ini}\} \qquad Act^{\mathsf{fnc}} \mathrel{\widehat{=}} \bigcup_{\tau \in \textsc{Tid}} \{\mathsf{fnc}_\tau\}$$

$$Act^a \mathrel{\widehat{=}} \bigcup_{x \in \textsc{Loc}, \eta \in \textsc{Exp}} \{\mathsf{bar}(a,x), \mathsf{bar}(a,\eta)\} \qquad Act^{\mathsf{tst}} \mathrel{\widehat{=}} \bigcup_{\tau \in \textsc{Tid}, \beta \in \textsc{BExp}} \{\mathsf{tst}_\tau(\beta)\}$$

Actions on a location x can be *read* actions $\mathsf{rd}_\tau(\cdot,\cdot)$, *fulfill* actions $\mathsf{ff}_\tau(\cdot,\cdot)$ or the initialization ini. Note that the thread identifier τ in read actions is the id of the thread having made the promise and in fulfill actions it is the thread executing the fulfill (and having made the corresponding promise). We let Act^{rd} denote all read and Act^{ff} all fulfill actions. To record loading into register a, we use so called *bar* actions $\mathsf{bar}(a,\cdot)$. The action fnc occurs when a **dmb** statement is executed and $\mathsf{tst}.(\cdot)$ describes the execution of some **asm** statement.

We often lift notations to sets of locations $L \subseteq \textsc{Loc}$ or sets of registers $R \subseteq \textsc{Reg}$. For example, $Act^L = \bigcup_{x \in L} Act^x$. The overall set of actions is $Act = Act^{\textsc{Loc}} \cup Act^{\textsc{Reg}} \cup Act^{\mathsf{fnc}} \cup Act^{\mathsf{tst}}$.

Definition 1. *A* location-coloured flow event structure *(short: event structure)* $\mathcal{E} = (E, \twoheadrightarrow, \#, \Lambda, \ell)$ *labelled over a set of actions Act consists of a finite set of events* E, *an irreflexive* flow relation $\twoheadrightarrow \subseteq E \times E$, *a* location restriction *function* $\Lambda : E \times E \to 2^{\text{Loc}}$, *a symmetric* conflict relation $\# \subseteq E \times E$, *and a* labelling function $\ell : E \to Act$.

For $L \subseteq \text{Loc}$, we write $e \overset{L}{\twoheadrightarrow} f$ to denote $e \twoheadrightarrow f$ and $\Lambda(e, f) = L$. The location restrictions are employed to reflect the application condition of rule Read within the event structure: it tells us that there is no write to $x \in L$ in between e and f, where e and f will eventually be mapped to timestamps in memory.

We let Ini be the event structure $(\{e_{\mathsf{ini}}\}, \emptyset, \emptyset, \emptyset, \ell)$ with $\ell(e_{\mathsf{ini}}) = \mathsf{ini}$. Given an event structure $\mathcal{E} = (E, \twoheadrightarrow, \#, \Lambda, \ell)$, we – similarly to actions – define its set of events labelled with specific actions as $\mathsf{Rd}(\mathcal{E})$, $\mathsf{Rd}_\tau(\mathcal{E})$, $\mathsf{Rd}^x_\tau(\mathcal{E})$, $\mathsf{Ff}(\mathcal{E})$, $\mathsf{Ff}_\tau(\mathcal{E})$ and $\mathsf{Ff}^x_\tau(\mathcal{E})$ via the labelling function ℓ. For an event e labelled with an action in $Act^x \backslash \{\mathsf{ini}\}$, we let $e.loc = x$. We slightly abuse notation so that $e_{\mathsf{ini}}.loc = x$ for all x. We furthermore define $last_\alpha(\mathcal{E})$, $\alpha \in Act$, to be the last event in flow order labelled α, i.e., $last_\alpha(\mathcal{E}) = e$ if $\ell(e) = \alpha$ and for all e' such that $e' \neq e$ and $\ell(e') = \alpha$, we have $e' \twoheadrightarrow^+ e$. Moreover, $last_\alpha(\mathcal{E}) = \bot$ if no event labelled α exists. We lift *last* to sets of actions by $last_A(\mathcal{E}) = \{last_\alpha(\mathcal{E}) \mid \alpha \in Act\}$. An event structure $\mathcal{E}$ is *sequential* if all events are flow-ordered: $\forall e, e' \in E, e \neq e' : e \twoheadrightarrow^+ e' \vee e' \twoheadrightarrow^+ e$. We let $\mathcal{S}$ be the set of sequential event structures.

An event structure describes (several) concurrent executions in compact form. One execution is therein given as a configuration.

Definition 2. *A* configuration $C \subseteq E$ *of an event structure* $\mathcal{E} = (E, \twoheadrightarrow, \#, \Lambda, \ell)$ *satisfies the following properties: (1) C is cycle-free:* $(\twoheadrightarrow \cap (C \times C))^+$ *is irreflexive, (2) C is conflict-free:* $\# \cap (C \times C) = \emptyset$, *(3) C is left-closed up to conflicts:* $\forall d, e \in E$, *if* $e \in C$, $d \twoheadrightarrow e$ *and* $d \notin C$, *then there exists* $f \in C$ *such that* $d \# f$ *and* $f \twoheadrightarrow e$.

We let $Conf(\mathcal{E})$ be the set of configurations of $\mathcal{E}$. We identify a configuration with the (conflict-free) event structure $\mathcal{E}_C$ which is $\mathcal{E}$ restricted to events of C.

Our intention is to use event structures to record information about the local history of each thread, in particular the promises of other threads which they have read from. Eventually (i.e., when combining local event structures) all promises read from need to be fulfilled. This is captured by our notion of parallel composition which requires fulfills (of a thread τ) to *synchronize* with reads from promises of τ. Similary to CCS [24], we model this synchronisation via *complementary* actions where $\overline{\mathsf{rd}_\tau(x, \kappa)} = \mathsf{ff}_\tau(x, \kappa)$ and vice versa, and $\overline{\overline{a}} = a$. Contrary to CCS, the synchronisation does not create internal actions, but keeps the fulfill labels (as to still see what promise a fulfill belonged to).

We first define the *synchronising events* of n event structures $\mathcal{E}_1, \ldots, \mathcal{E}_n$, as follows, where E_{i*} denotes $E_i \cup \{*\}$.

$$sync(\mathcal{E}_1, \ldots, \mathcal{E}_n) \mathrel{\widehat{=}} \left\{ \begin{array}{l} (e_1, e_2, \ldots, e_n) \in E_{1*} \times E_{2*} \times \cdots \times E_{n*} \mid \\ \exists i.\ \ell_i(e_i) \in Act^{\mathsf{ff}} \wedge (\forall j \neq i.\ e_j \neq * \Rightarrow \overline{\ell_i(e_i)} = \ell_j(e_j)) \wedge \\ \quad (\exists j \neq i.\ e_j \neq *) \end{array} \right\} \\ \cup \{(e^1_{\mathsf{ini}}, e^2_{\mathsf{ini}}, \ldots, e^n_{\mathsf{ini}})\}$$

An event e might also occur unsynchronized in a parallel composition (which is then written as $(*, \dots, *, e, *, \dots, *)$.

Note that since we aim to reason about reachability of states (underapproximation), we just need parallel composition for *conflict-free* event structures, i.e. for event structures describing a single execution. Thus the Δ-axiom of Castellani and Zhang [6] which they impose in order to get compositionality is trivially fulfilled for our application. Next, we still first of all define parallel composition of arbitrary event structures.

We let $\times_i S$ denote the product $S \times S \times \dots S$ generating a tuple of length i. If $i \leq 0$, we let $\times_i S = \bot$. Finally, we let $\bot \times S = S \times \bot = S$.

Definition 3 (Parallel composition). *Let $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_n$ be event structures for threads $\tau_1, \tau_2, \dots, \tau_n$, respectively. The* parallel composition $\mathcal{E} = \mathcal{E}_1 || \mathcal{E}_2 || \dots || \mathcal{E}_n$ *is the event structure $(E, \twoheadrightarrow, \#, \Lambda, \ell)$ with*

- $E = sync(\mathcal{E}_1, \mathcal{E}_2, \dots \mathcal{E}_n) \cup \left(\bigcup_i (\times_{i-1}\{*\}) \times (E_i \setminus \{e^i_{\mathsf{ini}}\}) \times (\times_{n-i}\{*\}) \right)$
- $(e_1, e_2, \dots, e_n) \twoheadrightarrow (d_1, d_2, \dots, d_n)$ *iff* $\exists i.\ e_i \twoheadrightarrow_i d_i$,
- $\Lambda((e_1, e_2, \dots, e_n), (d_1, d_2, \dots, d_n)) = \bigcup_i \Lambda(e_i, d_i)$,
- $(e_1, e_2, \dots, e_n) \# (d_1, d_2, \dots, d_n)$ *iff*
  - $\exists i.\ e_i \#_i d_i$, *or* *(inherit conflicts)*
  - $\exists i, j.\ e_i = d_i \wedge e_i \neq * \wedge e_j \neq d_j$ *(conflicts on differently paired events),*
- *Labels:*

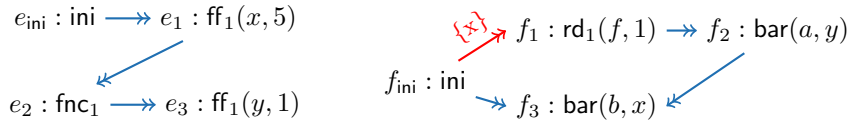
$$\ell(e_1, e_2, \dots e_n) = \begin{cases} \mathsf{ini} & \text{if } (e_1, e_2, \dots e_n) = (e^1_{\mathsf{ini}}, e^2_{\mathsf{ini}}, \dots e^n_{\mathsf{ini}}) \\ \ell(e_i) & \text{if } (e_1, e_2, \dots e_n) \in sync(\mathcal{E}_1, \mathcal{E}_2, \dots \mathcal{E}_n) \wedge \ell(e_i) = \mathsf{ff}.(\cdot, \cdot) \\ \ell(e_i) & \text{if } (e_1, e_2, \dots e_n) \notin sync(\mathcal{E}_1, \mathcal{E}_2, \dots \mathcal{E}_n) \wedge e_i \neq * \end{cases}$$

Parallel composition of event structures is used to combine local proof outlines of threads. This combination is only possible if enough synchronization partners are available. Event structures $\mathcal{E}_1$ to $\mathcal{E}_n$ are *synchronizable* if $\pi_i(sync(\mathcal{E}_1, \dots, \mathcal{E}_n)) \supseteq \mathsf{Rd}(E_i)$, $i \in \{1, \dots, n\}$ (all the reads have a synchronization with a fulfill). The configuration (describing an execution of the parallel composition of threads) which we extract from $Conf(\mathcal{E}_1 || \dots || \mathcal{E}_n)$ furthermore has to guarantee that no events from the local proof outlines are lost and that the local assertions make no contradictory assumptions about the contents of memory.

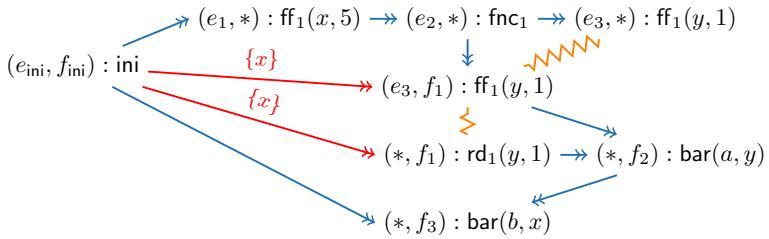
Definition 4. *The event structure $\mathcal{E}_C = (E_C, \twoheadrightarrow_C, \emptyset, \Lambda_C, \ell_C)$ corresponding to a configuration $C \in Conf(\mathcal{E}_1 || \dots || \mathcal{E}_n)$ is* interference free *if*

1. *C is* thread-covering*: $\forall i \in \{1, \dots, n\} : \pi_i(E_C) = E_i$,*
2. *C is* memory-consistent linearizable*: there exists a total order $\prec \subseteq Act^x(E_C) \times Act^x(E_C)$ among reads, fulfills and the* ini *event such that*
   - $\twoheadrightarrow_C{}^+ \cap (Act^x(E_C) \times Act^x(E_C)) \subseteq \prec$ *and*
   - $\forall d, e, f \in E_C$*:* $d \xrightarrow{L} f \wedge d \prec e \prec f \implies e.loc \notin L$,
3. *C contains* no unsynchronised reads*: there is no event in E_C of the form $(*, *, \dots, *, e_i, * \dots, *)$, where $e_i \in \mathsf{Rd}(\mathcal{E}_i)$.*

Example 1. Consider the two event structures given next (which belong to a message passing program with barriers, see Section 5).



Their parallel composition gives the following event structure:



This event structure has no interference-free configuration. To satisfy the conditions "thread-covering" and "no unsynchronised reads", we must include event (e_3, f_1). This means the only possible configuration must also include the down-closure $(e_1, *)$ and $(e_2, *)$. However, together with the location restriction $\{x\}$ on the edge $((e_{\mathsf{ini}}, f_{\mathsf{ini}}), (e_3, f_1))$, the resulting event structure is not memory-consistent linearizable, since it contains a sequence $(e_{\mathsf{ini}}, f_{\mathsf{ini}}) \twoheadrightarrow (e_1, *) \twoheadrightarrow (e_2, *) \twoheadrightarrow (e_3, f_1)$, where $(e_1, *)$ corresponds to a fulfilled write on x that is forbidden by the edge $((e_{\mathsf{ini}}, f_{\mathsf{ini}}) \overset{\{x\}}{\twoheadrightarrow} (e_3, f_1))$. Conceptually, this means that we cannot find a memory M which matches the constraints on its contents given in the event structure.

5 Reasoning

Our overall objective is the design of a proof calculus for reasoning about the *reachability* of certain final states of concurrent programs. A concurrent program state describes the values of registers and shared variables, the contents of memory and the views of threads. During reasoning, we employ event structures as *assertions* in proof outlines. They abstract from the concrete state in neither giving the exact contents of memory nor the timestamps of thread views.

5.1 Semantics of Assertions

Local assertions in the proof outlines of single threads take the form $\mathcal{E}$, where $\mathcal{E}$ is a conflict-free event structure (i.e., $\# = \emptyset$). The event structure is conflict-free because it describes a *single* execution of the thread (reachability logic). An assertion for a thread τ can have fence and fulfill events of τ, read events reading from (promises of) threads $\tau' \neq \tau$ as well as bar and test events over registers of $R(\tau)$. The events in $\mathcal{E}$ – together with some memory M – allow us to compute the current views of threads. Figure 4 gives some definitions for calculating views.

$$prFnc_\tau(\mathcal{E}) = \{e \in \mathsf{Rd}(\mathcal{E}) \cup \mathsf{Ff}(\mathcal{E}) \cup \{e_{\mathsf{ini}}\} \mid \exists e' \in last_{\mathsf{fnc}_\tau}(\mathcal{E}) : e \twoheadrightarrow^+ e'\}$$
$$prBar_a(\mathcal{E}) = \{e \in \mathsf{Rd}(\mathcal{E}) \cup \mathsf{Ff}(\mathcal{E}) \cup \{e_{\mathsf{ini}}\} \mid \exists e' \in last_{\mathsf{bar}(a,\cdot)}(\mathcal{E}) : e \twoheadrightarrow^+ e'\}$$
$$prBar_\tau(\mathcal{E}) = \bigcup_{a \in R(\tau)} prBar_a(\mathcal{E})$$
$$prBar^x_\tau(\mathcal{E}) = prBar_\tau(\mathcal{E}) \cap Act^x(\mathcal{E})$$
$$prTst_\tau(\mathcal{E}) = \{e \in \mathsf{Rd}(\mathcal{E}) \cup \mathsf{Ff}(\mathcal{E}) \cup \{e_{\mathsf{ini}}\} \mid \exists e' \in last_{\mathsf{tst}_\tau(\cdot)}(\mathcal{E}) : e \twoheadrightarrow^+ e'\}$$

Fig. 4. Determining the decisive reads and writes prior to an event ($\mathcal{E} = (E, \twoheadrightarrow, \#, \Lambda, \ell)$ event structure, $\tau \in \textsc{Tid}$, $a \in \textsc{Reg}, x \in \textsc{Loc}$)

A local assertion of a thread τ defines constraints on the global memory M (the ordering of writes and their values) as well as the views of τ: An assertion $\mathcal{E}$ describes a set of states $[\![\mathcal{E}]\!] = \{\langle \boldsymbol{ts}, M\rangle \in (\textsc{Tid} \to \mathit{TState}) \times \mathit{Memory} \mid \langle \boldsymbol{ts}, M\rangle \text{ matches } \mathcal{E}\}$ where "matches" is defined by conditions **(1)**-**(4)** below.
(1) M is consistent with the fulfill and read events of $\mathcal{E}$.
There exists a total mapping $\psi : \mathsf{Ff}(\mathcal{E}) \cup \mathsf{Rd}(\mathcal{E}) \cup \{e_{\mathsf{ini}}\} \to dom(M)$ which

1. *initializes at zero*: the one event e_{ini} labelled ini is mapped to 0,
2. is *consecutive* for every thread τ:
   for all $e \in \mathsf{Ff}_\tau(\mathcal{E})$, $t \in \mathbb{T}$ s.t. $M(t) = \langle x := \kappa\rangle_\tau$, $t < \psi(e)$ and $e.loc = x$, there exists $d \in \mathsf{Ff}_\tau(\mathcal{E})$ such that $\psi(d) = t$,
3. *preserves content*: if $\psi(e) = t \neq 0$ and $M(t) = \langle x := \kappa\rangle_\tau$, then $\ell(e) \in \{\mathsf{ff}_\tau(x,\kappa), \mathsf{rd}_\tau(x,\kappa)\}$,
4. *preserves flows*: $\forall e, e' \in dom(M) : e \twoheadrightarrow^+_{\mathcal{E}} e' \Rightarrow \psi(e) < \psi(e')$,
5. and *preserves memory constraints*:
   $\forall d, e \in dom(M)$, $L \subseteq \textsc{Loc}$ s.t. $d \overset{L}{\twoheadrightarrow} e$, $\forall t \in \mathbb{T}$ s.t. $\psi(d) < t < \psi(e)$: $M(t).loc \neq d.loc$.

The mapping ψ is used to assign timestamps to read and fulfill events. We therefore will later also talk about the *timestamp of an event* (depending on such a mapping). Note that the event structure Ini is consistent with all memories M (using mapping $\psi(e_{\mathsf{ini}}) = 0$).
(2) The open (non-fulfilled) promises of a thread τ are the entries of τ in M which are not fulfilled, i.e., $\boldsymbol{ts}(\tau).prom = M_\tau \setminus \psi(\mathsf{Ff}_\tau(\mathcal{E}))$.
(3) The views of a thread τ are consistent with mapping ψ and M.
Letting $ts = \boldsymbol{ts}(\tau)$, $a \in R(\tau)$ and $x \in \textsc{Loc}$, we have

$$ts.v_C = \bigsqcup_{e \in prTst_\tau(\mathcal{E})} \psi(e) \qquad ts.coh(x) = \bigsqcup_{e \in \mathsf{Ff}^x_\tau(\mathcal{E}) \cup prBar^x_\tau(\mathcal{E})} \psi(e)$$
$$ts.v_{wOld} = \bigsqcup_{e \in \mathsf{Ff}_\tau(\mathcal{E})} \psi(e) \qquad ts.v_{wNew} = \bigsqcup_{e \in prFnc_\tau(\mathcal{E}) \cap \left(\mathsf{Ff}_\tau(\mathcal{E}) \cup prBar_\tau(\mathcal{E})\right)} \psi(e)$$
$$zts.v_a = \bigsqcup_{e \in prBar_a(\mathcal{E})} \psi(e) \qquad ts.v_{read} = \bigsqcup_{e \in \left(prFnc_\tau(\mathcal{E}) \cap \mathsf{Ff}_\tau(\mathcal{E})\right) \cup prBar_\tau(\mathcal{E})} \psi(e)$$

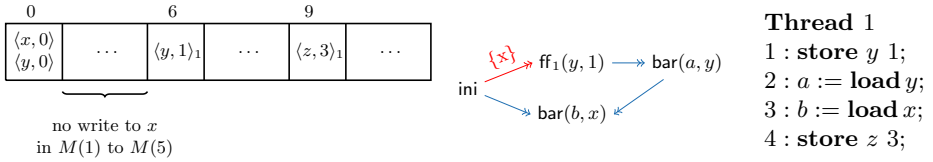


Fig. 5. Example memory M (left) for event structure $\mathcal{E}$ (middle) describing an execution of statements 1, 2 and 3 in thread 1 (right).
State ts of thread 1: $prom = \{9\}, v_C = v_{wNew} = coh(z) = 0, v_a = v_b = coh(y) = coh(x) = v_{wOld} = v_{read} = 6$, using mapping $\psi : \mathsf{ini} \mapsto 0, \mathsf{ff}_1(y,1) \mapsto 6$.

(4) The values of registers $R(\tau)$ of thread τ agree with values in $\mathcal{E}$.
For $a \in \textsc{Reg}$, $ts.regs(a) = \kappa @ v_a$ with $\kappa = [\![a]\!]_{\mathcal{E}}$ (where the semantics of a register a in $\mathcal{E}$ is (1) 0 if no bar event for a is in $\mathcal{E}$ or (2) the value of a read or fulfill to x prior to the last $\mathsf{bar}(a, \cdot)$ (on x) or (3) the value of the expression η in a last $\mathsf{bar}(a, \eta)$) and v_a as defined above.

Figure 5 gives an example for the definition of "matches". On the right hand side we see the program of thread 1. It first stores 1 to y, then loads the values of y and x into registers a and b, respectively, and finally stores 3 to z. The event structure in the middle gives the assertion reached after statement 3, i.e. *before* the final store operation. The memory M on the left hand side matches this event structure: There are promises for the event ini at $M(0)$ as well as for event $\mathsf{ff}_1(y,1)$, so ψ maps ini to 0 and $\mathsf{ff}_1(y,1)$ to 6. The colored location restriction in the event structure furthermore requires not to have any promises to x in between 0 and 6. As there is one more promise of thread 1 in M, not yet covered by the event structure, we can derive $1.prom = \{9\}$.

5.2 Proof Rules

Essentially, assertions describe the events which have already happened together with their orderings plus further constraints. The initial assertion in proof outlines is always the event structure Ini. Then, the proof rules successively add new events to the event structure when e.g. reading from or writing to shared variables. We however *never* add events for promises; rather, threads can first of all assume arbitrary promises of other threads having been made which they can read from. The overall interference freedom constraint guarantees that these local assumptions about promises are met at the end.

For adding new events, we use a number of $\oplus$-operators, detailed in Fig. 6. The event structures in there are local to threads and describe a single execution of the thread, hence are conflict-free. The definition of these operators has to ensure that they capture the dependencies between views as defined by the operational semantics. For example, rule Fulfill requires (among others) the timestamp t to be larger than control view v_C, hence $\mathcal{E} \oplus \mathsf{ff}_\tau(x, \kappa)$ has to introduce a flow from the last test event to the newly added fulfill event.

$$
\begin{aligned}
\mathcal{E} \oplus \mathsf{ff}_\tau(x,\kappa) &= (E_e, \twoheadrightarrow \cup \{(e',e) \mid e' \in last_{Act^x \cup \{\mathsf{fnc}_\tau, \mathsf{tst}_\tau(\cdot)\}}(\mathcal{E})\}, \Lambda, \ell[e \mapsto \mathsf{ff}_\tau(x,\kappa)])\\
\mathcal{E} \oplus^a \mathsf{ff}_\tau(x,\kappa) &= (E_e, \twoheadrightarrow \cup \{(e',e) \mid e' \in last_{Act^x \cup \{\mathsf{fnc}_\tau, \mathsf{tst}_\tau(\cdot), \mathsf{bar}(a,\cdot)\}}(\mathcal{E})\},\\
&\qquad\qquad \Lambda, \ell[e \mapsto \mathsf{ff}_\tau(x,\kappa)])\\
\mathcal{E} \oplus \mathsf{bar}(a,x) &= (E_e, \twoheadrightarrow \cup \{(e',e) \mid e' \in last_{Act^x \cup \{\mathsf{fnc}_\tau, \mathsf{bar}(\cdot,\cdot)\}}(\mathcal{E})\}, \Lambda, \ell[e \mapsto \mathsf{bar}(a,x)])\\
\mathcal{E} \oplus \mathsf{bar}(a,\eta) &= (E_e, \twoheadrightarrow \cup \left\{ \begin{array}{l}(e',e) \mid \\ \exists b \in R(\eta) \cup \{a\} : e' = last_{\mathsf{bar}(b,\cdot)}(\mathcal{E})\end{array} \right\}, \Lambda, \ell[e \mapsto \mathsf{bar}(a,\eta)])\\
\mathcal{E} \oplus \mathsf{fnc}_\tau &= (E_e, \twoheadrightarrow \cup \{(e',e) \mid e' \in last_{Act \setminus \{\mathsf{tst}_\tau(\cdot)\}}(\mathcal{E})\}, \Lambda, \ell[e \mapsto \mathsf{fnc}_\tau])\\
\mathcal{E} \oplus \mathsf{tst}_\tau(\beta) &= (E_e, \twoheadrightarrow \cup \{(e',e) \mid \exists a \in R(\beta).\ e' \in last_{\mathsf{bar}(a,\cdot)}(\mathcal{E})\}, \Lambda, \ell[e \mapsto \mathsf{tst}_\tau(\beta)])\\
\mathcal{E} \oplus \mathcal{E}' &= \begin{pmatrix} E \cup E', \\ \twoheadrightarrow \cup \twoheadrightarrow' \cup \left\{ \begin{array}{l}(e,e') \in E \times E' \mid \exists x \in \textsc{Loc}. \\ e = last_{\{\mathsf{fnc}, \mathsf{bar}(\cdot,\cdot), \mathsf{ff}.(x,\cdot)\}}(\mathcal{E}) \wedge \ell(e') \in Act^x \end{array} \right\}, \\ \Lambda \cup \Lambda', \ell \cup \ell' \end{pmatrix}
\end{aligned}
$$

Fig. 6. Operations for adding events to a conflict-free event structure $\mathcal{E} = (E, \twoheadrightarrow, \Lambda, \ell)$, where $e \notin E$ is a fresh event and $E_e = E \cup \{e\}$, $\mathcal{E}' = (E', \twoheadrightarrow', \Lambda', \ell')$, $E \cap E' = \emptyset$)

Figure 7 gives the proof rules for building local proof outlines of threads. Most of the rules (i.e., PR-Write, PR-WriteR, PR-Fence, PR-Registers and PR-Assume) just add one new event to the event structure recording the occurrence of a particular program statement. More complex are the two read rules: PR-ReadEx is applied for load statements reading from x when the event structure already contains an event e describing (in the sense of $[\![\mathcal{E}]\!]$) the entry in memory to read from; this can be a read, fulfill or the ini event. In this case, the event structure after the load has to reflect the applicability condition of rule Read: no entries in memory to x in between t (the timestamp of e in $[\![\mathcal{E}]\!]$) and $v_{read} \sqcup coh(x)$. This is achieved by inserting an additional location restriction x via the operator $rstr_e^x(\mathcal{E})$ to the following (potentially already L-labelled) flows (thus getting the restriction $L \cup \{x\}$):

$$\{e \overset{L}{\twoheadrightarrow} e' \mid e' \in (prFnc_\tau(\mathcal{E}) \cap \mathsf{Ff}_\tau(\mathcal{E})) \cup prBar_\tau(\mathcal{E}) \cup \mathsf{Ff}_\tau^x(\mathcal{E})\}\ .$$

Rule PR-ReadNew on the other hand introduces new read events into an event structure upon a load statement. The rule can directly introduce an entire *sequence* of read events (i.e., add a sequential event structure $\mathcal{E}'$) as to enable later reads from memory entries which are prior to the entry of the current read (described by event e in the rule). This is required for message passing idioms like in the following program.

Thread 1	**Thread** 2
1 : **store** x 5;	4 : $a :=$ **load** y;
2 : `dmb`;	5 : $b :=$ **load** x;
3 : **store** y 1;	

Here, due to the fence in Thread 1, Thread 2 – after having read y to be 1 – can only read x to be 5. When constructing the proof outline for Thread 2, we

$$\textsc{PR-Write}\ \frac{}{[\mathcal{E}]\ \mathbf{store}\ x\ \kappa\ [\mathcal{E} \oplus \mathsf{ff}_\tau(x,\kappa)]} \qquad \textsc{PR-WriteR}\ \frac{[\![a]\!]_\mathcal{E} = \kappa}{[\mathcal{E}]\ \mathbf{store}\ x\ a\ [\mathcal{E} \oplus^a \mathsf{ff}_\tau(x,\kappa)]} \qquad \textsc{PR-Fence}\ \frac{}{[\mathcal{E}]\ \mathbf{dmb}_\tau\ [\mathcal{E} \oplus \mathsf{fnc}_\tau]}$$

$$\textsc{PR-ReadEx}\ \frac{e = last_{Act^x}(\mathcal{E}) \quad \ell(e) \in \{\mathsf{rd}_{\tau'}(x,\kappa), \mathsf{ff}_\tau(x,\kappa), \mathsf{ini}\}}{[\mathcal{E}]\ a := \mathbf{load}\ x\ [rstr_e^x(\mathcal{E} \oplus \mathsf{bar}(a,x))]} \qquad \textsc{PR-ReadNew}\ \frac{\mathcal{E}' = (E', \twoheadrightarrow', \Lambda', \ell') \in \mathcal{S} \quad \ell'(E') \subseteq Act^{Rd} \setminus Act_\tau \quad last_{Act}(\mathcal{E}') = e, \ell'(e) = \mathsf{rd}_{\tau'}(x,\kappa)}{[\mathcal{E}]\ a := \mathbf{load}\ x\ [(\mathcal{E} \oplus \mathcal{E}') \oplus \mathsf{bar}(a,x)]}$$

$$\textsc{PR-Registers}\ \frac{}{[\mathcal{E}]\ a := \eta\ [\mathcal{E} \oplus \mathsf{bar}(a,\eta)]} \qquad \textsc{PR-Assume}\ \frac{[\![\beta]\!]_\mathcal{E} = \mathsf{true}}{[\mathcal{E}]\ \mathbf{asm}\ \beta\ [\mathcal{E} \oplus \mathsf{tst}_\tau(\beta)]} \qquad \textsc{PR-Choice}\ \frac{[\mathcal{E}_1]\ S_i\ [\mathcal{E}_2]}{[\mathcal{E}_1]\ S_1 + S_2\ [\mathcal{E}_2]}$$

$$\textsc{PR-Sequencing}\ \frac{[\mathcal{E}_1]\ S_1\ [\mathcal{E}_2] \qquad [\mathcal{E}_2]\ S_1\ [\mathcal{E}_3]}{[\mathcal{E}_1]\ S_1; S_2\ [\mathcal{E}_3]} \qquad \textsc{PR-IterateZero}\ \frac{}{[\mathcal{E}]\ S^0\ [\mathcal{E}]} \qquad \textsc{PR-IterateNonZero}\ \frac{n > 0 \qquad [\mathcal{E}_1]\ S; S^{n-1}\ [\mathcal{E}_2]}{[\mathcal{E}_1]\ S^n\ [\mathcal{E}_2]}$$

Fig. 7. Local proof rules for a thread τ

need to apply rule PR-ReadNew for the first load giving us

$$\mathsf{ini} \twoheadrightarrow \mathsf{rd}_1(x,5) \twoheadrightarrow \mathsf{rd}_1(y,1) \twoheadrightarrow \mathsf{bar}(a,y)$$

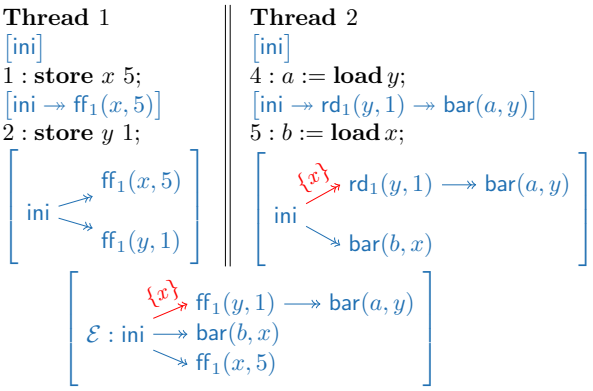
as assertion after statement 4. For the subsequent load we can then apply proof rule PR-ReadEx. Note that we could also construct a local proof outline having the load in line 4 read from ini. This would then give us the two event structures of Example 1 which we, however, have already seen to not allow for an interference free configuration of their parallel composition.

Finally, we have a proof rule for parallel composition which combines local event structures when they are synchronisable and the resulting configuration is interference free.

$$\textsc{Parallel}\ \frac{\forall i \in \{1,\dots,n\}.\ [\mathsf{Ini}]\ S_i\ [\mathcal{E}_i] \qquad \mathcal{E}_1,\dots,\mathcal{E}_n \text{ synchronisable} \qquad \text{interference free } C \in Conf(\mathcal{E}_1||\dots||\mathcal{E}_n)}{[\mathsf{Ini}]\ S_1||\dots||S_n\ [\mathcal{E}_C]}$$

This rule ensures that (1) all synchronization constraints are met (i.e., the promises that threads want to read from have been made) and (2) there is a configuration C of the combined event structure which is interference free.

Example 2. Next, we give a complete proof outline for the message passing litmus test *without* a barrier in the writing thread. We see that here message passing is not guaranteed (i.e., reading y to be 1 does not "pass the message" that x is 5 from Thread 1 to 2) and we can actually reach a final state with $(a = 1 \wedge b = 0)$ (as calculated by $[\![a]\!]_\mathcal{E}$ and $[\![b]\!]_\mathcal{E}$ taking the value of the last fulfill or ini event prior to the last bar event on a and b, respectively).



5.3 Soundness and Completeness

Due to lack of space, we can neither discuss soundness nor completeness of our proof calculus in some more detail here. Proofs can be found in the extended version [38].

Soundness requires proving all local proof rules correct plus showing the correctness of rule Parallel as of Theorem 1 below. It states that whenever we find an interference free configuration in the parallel composition of synchronizable event structures in a locally sound proof outline, all thread states and memory contents matching this configuration are actually reachable by the concurrent program.

Theorem 1. *Let* [Ini] S_i [$\mathcal{E}_i$], $i \in \{1, \ldots, n\}$, *be proof outlines of threads* τ_1 *to* τ_n *such that* $\mathcal{E}_1$ *to* $\mathcal{E}_n$ *are synchronizable and let* $\boldsymbol{T_0}$ *be an initial thread pool with* $\boldsymbol{T_0}(\tau_i) = (S_i, ts_{\mathsf{ini}})$ *and* $M_0 = M_{\mathsf{ini}}$.

Then, for every thread pool $\boldsymbol{T}$ *with* $\boldsymbol{T}(\tau_i) = (\mathbf{skip}, ts_i)$, *interference free configuration* $C \in Conf(\mathcal{E}_1 || \ldots || \mathcal{E}_n)$ *and memory* M *such that* $\langle ts_i, M\rangle \in [\![\mathcal{E}_C]\!]$, $tids(M) = \{\tau_1, \ldots, \tau_n\}$ *and* $ts_i.prom = \emptyset$, $i \in \{1, \ldots, n\}$, *we have* $\langle \boldsymbol{T_0}, M_0\rangle \rightarrow^* \langle \boldsymbol{T}, M\rangle$.

Our second main result is the *completeness* of the proof calculus: whenever there is an execution of a concurrent program, our proof calculus allows to show the reachability of its final state. More specifically, for every trace of a concurrent program we find local proof outlines with synchronizable event structures and an interference free configuration describing the final state of the trace.

Theorem 2. *Let* $\langle \boldsymbol{T_0}, M_0\rangle \rightarrow^* \langle \boldsymbol{T}, M\rangle$ *be a trace of a concurrent program over threads* $\tau_1, \ldots, \tau_n$ *such that* $\boldsymbol{T_0}$ *is the initial thread pool with* $\boldsymbol{T_0}(\tau_k) = (S_k, ts_{\mathsf{ini}})$, $M_0 = M_{\mathsf{ini}}$ *and* $\boldsymbol{T}$ *the final thread pool with* $\boldsymbol{T}(\tau_k) = (\mathbf{skip}, ts_k)$ *and* $ts_k.prom = \emptyset$, $k \in \{1, \ldots, n\}$.

Then there are local proof outlines [Ini] S_k [$\mathcal{E}_k$] *of threads* τ_k, $k \in \{1, \ldots, n\}$, *such that* $\mathcal{E}_1$ *to* $\mathcal{E}_n$ *are synchronizable and there exists an interference free configuration* $C \in Conf(\mathcal{E}_1 || \ldots || \mathcal{E}_n)$ *with* $\langle \boldsymbol{T}, M\rangle \in [\![\mathcal{E}_C]\!]$.

6 Related Work

The first semantics of weak memory models employing *promises* has been proposed by Kang et al. in 2017 [21] for building an operational semantics which allows modelling of read-write reordering while at the same time disallows out-of-thin-air behaviours. Our semantics here is a slightly simplified version of the promising semantics of ARMv8 given by Pulte et al. [29]. In particular, like [29] all program traces can be reordered so that the promise steps are all at the beginning which is a key property required for the soundness of our proof calculus.

There are already several proposals for program logics for weak memory e.g. [1, 9, 10, 12–14, 22, 32, 36]. The only one explicitly dealing with promises in the semantics is the proposal of Svendsen et al. [35]. They develop a safety proof calculus whereas we are interested in reachability. Their logic furthermore has to deal with promises occurring at any program step (as they show soundness with respect to the promising semantics of [21]), whereas we rely on all promises being made at the beginning.

Partial order models of concurrency have already been used for giving the semantics of memory models [7,18,19], but not for reasoning. Wright et al [40] take the approach of using a *semantic dependency* relation, which is a partial order generated through an event structure representation of a C/C++ program [28], which is a partial order over a thread's execution. An Owicki-Gries logic is provided to reason directly over such partial orders. Incorrectness logic as used for proving reachability properties of *sequential* programs has been introduced by O'Hearn [26], with a predecessor approach with (almost) the same principles by de Vries and Koutavas [37]. The first extension of incorrectness logic to concurrent programs has been proposed by Raad et al. in the form of an incorrectness separation logic [30] which is however not compositional.

Colvin [8] defines a semantics based on a reordering relation for several hardware memory models, which is then lifted to a Hoare calculus. This is then rephrased into a reachability property by defining triples $\langle\langle p \rangle\rangle\ s\ \langle\langle q \rangle\rangle = \neg\{p\}\ s\ \{\neg q\}$, which states that it is possible for s to reach q if execution starts in a state satisfying p. Note that this is weaker than O'Hearn's notion of incompleteness, which states that all states satisfying q are reachable from an execution starting in a state satisfying p.

7 Conclusion

In this paper, we have proposed a reachability (incorrectness) logic for concurrent programs running on weak memory models. The reasoning technique is based on assertions which are event structures abstractly describing the contents of memory and the views of all threads. We have proven soundness and completeness of the proof calculus, and have demonstrated its applicability by proving the outcomes of some standard litmus tests to be reachable.

Acknowledgements. We thank Christopher Pulte for clarifying one aspect of the Register rule of ARMv8's operational semantics to us and Sadegh Dalvandi for initial discussions on the semantics.

References

1. Alglave, J., Cousot, P.: Ogre and Pythia: an invariance proof method for weak consistency models. In: Castagna, G., Gordon, A.D. (eds.) POPL. pp. 3–18. ACM (2017). https://doi.org/10.1145/3009837.3009883, https://doi.org/10.1145/3009837.3009883
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL. pp. 55–66. ACM (2011). https://doi.org/10.1145/1926385.1926394, https://doi.org/10.1145/1926385.1926394
3. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-Based Owicki-Gries Reasoning for Persistent x86-TSO. In: ESOP. Lecture Notes in Computer Science, vol. 13240, pp. 234–261. Springer (2022)
4. Boudol, G., Castellani, I.: On the semantics of concurrency: Partial orders and transition systems. In: Ehrig, H., Kowalski, R.A., Levi, G., Montanari, U. (eds.) TAPSOFT'87. Lecture Notes in Computer Science, vol. 249, pp. 123–137. Springer (1987). https://doi.org/10.1007/3-540-17660-8_52, https://doi.org/10.1007/3-540-17660-8_52
5. Boudol, G., Castellani, I.: Flow models of distributed computations: Three equivalent semantics for CCS. Inf. Comput. **114**(2), 247–314 (1994). https://doi.org/10.1006/inco.1994.1088, https://doi.org/10.1006/inco.1994.1088
6. Castellani, I., Zhang, G.: Parallel product of event structures. Theor. Comput. Sci. **179**(1-2), 203–215 (1997). https://doi.org/10.1016/S0304-3975(96)00104-1, https://doi.org/10.1016/S0304-3975(96)00104-1
7. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290383, https://doi.org/10.1145/3290383
8. Colvin, R.J.: Parallelized sequential composition and hardware weak memory models. In: Calinescu, R., Pasareanu, C.S. (eds.) SEFM 2021. Lecture Notes in Computer Science, vol. 13085, pp. 201–221. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_12, https://doi.org/10.1007/978-3-030-92124-8_12
9. Coughlin, N., Winter, K., Smith, G.: Rely/Guarantee Reasoning for Multicopy Atomic Weak Memory Models. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) FM. Lecture Notes in Computer Science, vol. 13047, pp. 292–310. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_16, https://doi.org/10.1007/978-3-030-90870-6_16
10. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries Reasoning for C11 RAR. In: Hirschfeld, R., Pape, T. (eds.) ECOOP. LIPIcs, vol. 166, pp. 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.11, https://doi.org/10.4230/LIPIcs.ECOOP.2020.11
11. Dalvandi, S., Dongol, B., Doherty, S., Wehrheim, H.: Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. J. Autom. Reason. **66**(1), 141–171 (2022). https://doi.org/10.1007/s10817-021-09610-2, https://doi.org/10.1007/s10817-021-09610-2
12. Doherty, S., Dalvandi, S., Dongol, B., Wehrheim, H.: Unifying operational weak memory verification: An axiomatic approach. ACM Trans. Comput. Log. **23**(4), 27:1–27:39 (2022). https://doi.org/10.1145/3545117, https://doi.org/10.1145/3545117

13. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) PPoPP. pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702, https://doi.org/10.1145/3293883.3295702
14. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 413–430. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_20, https://doi.org/10.1007/978-3-662-49122-5_20
15. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: Bodík, R., Majumdar, R. (eds.) POPL. pp. 608–621. ACM (2016). https://doi.org/10.1145/2837614.2837615, https://doi.org/10.1145/2837614.2837615
16. van Glabbeek, R.J., Goltz, U.: Well-behaved flow event structures for parallel composition and action refinement. Theor. Comput. Sci. **311**(1-3), 463–478 (2004). https://doi.org/10.1016/j.tcs.2003.10.031, https://doi.org/10.1016/j.tcs.2003.10.031
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
18. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with preconditions: a simple model of relaxed memory. Proc. ACM Program. Lang. **4**(OOPSLA), 194:1–194:30 (2020). https://doi.org/10.1145/3428262, https://doi.org/10.1145/3428262
19. Jeffrey, A., Riely, J., Batty, M., Cooksey, S., Kaysin, I., Podkopaev, A.: The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. Proc. ACM Program. Lang. **6**(POPL), 1–30 (2022). https://doi.org/10.1145/3498716, https://doi.org/10.1145/3498716
20. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In: Müller, P. (ed.) ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.17, https://doi.org/10.4230/LIPIcs.ECOOP.2017.17
21. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) POPL. pp. 175–189. ACM (2017). https://doi.org/10.1145/3009837.3009850, https://doi.org/10.1145/3009837.3009850
22. Lahav, O., Vafeiadis, V.: Owicki-Gries Reasoning for Weak Memory Models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP. Lecture Notes in Computer Science, vol. 9135, pp. 311–323. Springer (2015). https://doi.org/10.1007/978-3-662-47666-6_25, https://doi.org/10.1007/978-3-662-47666-6_25
23. Lee, S., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C., Lahav, O., Vafeiadis, V.: Promising 2.0: global optimizations in relaxed memory concurrency. In: Donaldson, A.F., Torlak, E. (eds.) PLDI. pp. 362–376. ACM (2020). https://doi.org/10.1145/3385412.3386010, https://doi.org/10.1145/3385412.3386010
24. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
25. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for C/C++11 concurrency. In: Visser, E., Smaragdakis, Y. (eds.) OOPSLA. pp. 111–128. ACM (2016). https://doi.org/10.1145/2983990.2983997, https://doi.org/10.1145/2983990.2983997

26. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 10:1–10:32 (2020). https://doi.org/10.1145/3371078, https://doi.org/10.1145/3371078
27. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
28. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: Müller, P. (ed.) ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 599–625. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_22, https://doi.org/10.1007/978-3-030-44914-8_22
29. Pulte, C., Pichon-Pharabod, J., Kang, J., Lee, S.H., Hur, C.: Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In: McKinley, K.S., Fisher, K. (eds.) PLDI. pp. 1–15. ACM (2019). https://doi.org/10.1145/3314221.3314624, https://doi.org/10.1145/3314221.3314624
30. Raad, A., Berdine, J., Dreyer, D., O'Hearn, P.W.: Concurrent incorrectness separation logic. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498695, https://doi.org/10.1145/3498695
31. Raad, A., Lahav, O., Vafeiadis, V.: Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA), 151:1–151:28 (2020). https://doi.org/10.1145/3428219, https://doi.org/10.1145/3428219
32. Ridge, T.: A rely-guarantee proof system for x86-tso. In: Leavens, G.T., O'Hearn, P.W., Rajamani, S.K. (eds.) VSTTE. Lecture Notes in Computer Science, vol. 6217, pp. 55–70. Springer (2010). https://doi.org/10.1007/978-3-642-15057-9_4, https://doi.org/10.1007/978-3-642-15057-9_4
33. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Hall, M.W., Padua, D.A. (eds.) PLDI. pp. 175–186. ACM (2011). https://doi.org/10.1145/1993498.1993520, https://doi.org/10.1145/1993498.1993520
34. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010). https://doi.org/10.1145/1785414.1785443, https://doi.org/10.1145/1785414.1785443
35. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 357–384. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_13, https://doi.org/10.1007/978-3-319-89884-1_13
36. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) OOPSLA. pp. 867–884. ACM (2013). https://doi.org/10.1145/2509136.2509532, https://doi.org/10.1145/2509136.2509532
37. de Vries, E., Koutavas, V.: Reverse Hoare Logic. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM. Lecture Notes in Computer Science, vol. 7041, pp. 155–171. Springer (2011). https://doi.org/10.1007/978-3-642-24690-6_12, https://doi.org/10.1007/978-3-642-24690-6_12
38. Wehrheim, H., Bargmann, L., Dongol, B.: Reasoning about promises in weak memory models with event structures (extended version) (2022). https://doi.org/10.48550/ARXIV.2211.16330, https://arxiv.org/abs/2211.16330

39. Winskel, G.: An introduction to event structures. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. Lecture Notes in Computer Science, vol. 354, pp. 364–397. Springer (1988). https://doi.org/10.1007/BFb0013026, https://doi.org/10.1007/BFb0013026
40. Wright, D., Batty, M., Dongol, B.: Owicki-Gries reasoning for C11 programs with relaxed dependencies. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) FM. Lecture Notes in Computer Science, vol. 13047, pp. 237–254. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_13, https://doi.org/10.1007/978-3-030-90870-6_13