Reasoning about Promises in Weak Memory Models with Event Structures (Extended Version)*

Heike Wehrheim\textsuperscript{1}, Lara Bargmann\textsuperscript{1}, and Brijesh Dongol\textsuperscript{2}

\textsuperscript{1} University of Oldenburg, Oldenburg, Germany
\textsuperscript{2} University of Surrey, Guildford, UK

\textbf{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.

\textbf{Keywords:} Weak memory models, promises, event structures, incorrectness logic.

\section{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 \cite{36}, C11 \cite{2,27}, Power \cite{35} or ARM \cite{16}. 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 \cite{14,21,31} avoid modelling specific hardware components and instead define the semantics in terms of views of thread on the shared state. Promises \cite{22,24} are employed in operational semantics as a way of capturing out-of-order writes while still executing operations in thread order. A promise (w.r.t. a value $\kappa$ and a shared location $x$) of a thread $\tau$ states that $\tau$ will eventually write value $\kappa$ 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 \cite{18} structural proof calculi (and their extensions to concurrency by Owicki and Gries \cite{29}) for weak memory models. Owicki-Gries-like proof calculi have been proposed by a number of researchers \cite{11,12,23,41}, and have also recently been given for non-volatile memory \cite{3,33}. Svendsen et al. \cite{37} have developed a separation

* 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.
logic for promises for the C11 memory model. Wright et al. [41] 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 [30].

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 [28]. 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 [6, 17, 40], 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 [31]. 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.

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 \( a = 1 \land b = 1 \).

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 \( \bar{E} \) which are flow event structures [40]. 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
To reason about the set of reachable final states of the concurrent program, we again construct the parallel composition of the event structures, synchronising read with their corresponding fulfill events. In Fig. 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 \texttt{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 \texttt{a}, obtaining the assertions shown. Note that here the event structure contains an additional fence event, \texttt{fnc}, that is ordered after \texttt{bar(a,y)}, causally ordered after \texttt{rd2(y,1)}, which states that the view of register \texttt{a} is at least that of the read of \texttt{y}. Execution of line 2 then adds a fulfill event with label \texttt{ff1(x,1)} to the assertion, which is not ordered with any other event except \texttt{ini}. Symmetric assertions can be generated for Thread 2.

1 of Fig. 1, the pre-assertion of the load only contains an event for initial writes (labelled \texttt{ini}), yet the load may read the value 1 for \texttt{y} from a promised write, described by the event labelled \texttt{rd2(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 \texttt{bar(a,y)}, causally ordered after \texttt{rd2(y,1)}, which states that the view of register \texttt{a} is at least that of the read of \texttt{y}. Execution of line 2 then adds a fulfill event with label \texttt{ff1(x,1)} to the assertion, which is not ordered with any other event except \texttt{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 \texttt{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 \texttt{a}, obtaining the assertions shown. Note that here the event structure contains an additional fence event, \texttt{fnc}, that is ordered after \texttt{bar(a,y)}, and before \texttt{ff1(x,1)}. Similarly, for Thread 2 loading 1 into \texttt{b}, we would obtain a symmetric set of assertions. Here, the parallel composition of local assertions is however not \textit{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 \( \rightarrow \) arrow denotes ordering and we now give event names together with labels.

\[
\mathcal{E}_1: \quad e_{\text{ini}} : \text{ini} \rightarrow e_1 : \text{rd2(y,1)} \rightarrow e_2 : \text{bar(a,y)} \rightarrow e_3 : \text{fnc} \rightarrow e_4 : \text{ff1(x,1)}
\]

\[
\mathcal{E}_2: \quad f_{\text{ini}} : \text{ini} \rightarrow f_1 : \text{rd1(x,1)} \rightarrow f_2 : \text{bar(b,x)} \rightarrow f_3 : \text{fnc} \rightarrow f_4 : \text{ff2(y,1)}
\]

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 \( \mathcal{E}_1 \parallel \mathcal{E}_2 \)).
This composition of event structure is built similar to [17], allowing events of the parallel composition to be lifted from the sub-components. These are events of the form \((e_i, \ast)\) and \((\ast, f_i)\). The parallel composition also contains synchronised read/fulfill events, e.g., \((e_1, f_4)\) depicts a read synchronised with the fulfill (write) \(ff_1(y, 1)\). 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 \((e_1, \ast)\) and \((\ast, f_4)\).

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 \(\rightarrow\), 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 \((e_1, \ast)\) cannot include 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: \((e_1, f_4) \rightarrow (e_2, \ast) \rightarrow (e_3, \ast) \rightarrow (e_4, f_1) \rightarrow (\ast, f_2) \rightarrow (\ast, f_3) \rightarrow (e_1, f_4)\). 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 \(a = 1 \land b = 1\) 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. [31]. 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 [22, 24, 31]. 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 \(x, y \in \text{Loc}\) be the set of shared locations, \(\kappa \in \text{Val}\) the set of values, \(\tau \in \text{Tid}\) the set of thread identifiers and \(a, b \in \text{Reg}\) local registers. Our sequential language encompasses the following constructs:

\[
rv ::= \kappa \mid a \quad st ::= \text{skip} \mid a ::= \text{load}\ x \mid \text{store}\ x \ rv \mid a ::= \eta \mid \text{dmb} \\
S ::= st \mid S; S \mid \text{asm}\ \beta \mid S + S \mid S^\ast
\]

where \(\eta \in \text{Exp}\) is an arithmetic and \(\beta \in \text{BExp}\) is a boolean expressions, both over (local) registers only. We assume \(S^\ast = \exists n \in \mathbb{N}. S^n\), where \(S^0 \equiv \text{skip}\) and \(S^n \equiv S; S^{n-1}\). We use abbreviations: \(\text{while } \beta \text{ do } S = (\text{asm}\ \beta; S)^\ast\); \(\text{asm} \neg \beta\) and \(\text{if } \beta \text{ then } S_1 \text{ else } S_2 = (\text{asm}\ \beta; S_1) + (\text{asm} \neg \beta; S_2)\), where \(\text{asm}\ \beta\) is a command that tests whether \(\beta\) holds.
Threaded state. We let $\mathcal{T}State$ be the set of all timestamped states and $\mathcal{M}emory$ the set of all memory states, both of which we make more precise below. A thread $T \in \mathcal{Thread}$ is an element of $S \times \mathcal{T}State$, a concurrent program is a mapping $T \in \mathcal{T}Pool \equiv \mathcal{T}id \rightarrow \mathcal{Thread}$ and a concurrent program state is a pair $(T, M) \in \mathcal{T}Pool \times \mathcal{M}emory$. We let $R(\tau), \tau \in \mathcal{T}id$, be the set of registers occurring in the program of $T(\tau)$. 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 $t \cup t' \equiv \max(t, t')$ and generalise this to sets of timestamps using $\bigcup_{t \in \mathbb{T}} t = 0$. A memory is a sequence of write messages of type $W_r \equiv (\mathcal{L}oc \times \mathcal{V}al \times \mathcal{T}id) \cup \{\text{ini}\}$, 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 $w \equiv (x, \kappa, \tau)$ using $\{x := \kappa\}_\tau$ and let $w.loc = x, w.val = \kappa$ and $w.tid = \tau$. For a memory $M$ and thread $\tau \in \mathcal{T}id$, we let $M_\tau \subseteq \mathbb{T}$ be the set of timestamps of entries of $\tau$ in $M$, i.e. $\{t \in \mathbb{T} \mid M(t).t id = \tau\}$. $M_\tau$ is used to determine the promise set of each $\tau$. We write $\text{tids}(M)$ 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 $M \oplus w$.

A thread state $ts \in \mathcal{T}State$ consists of the following components: a set of (non-fulfilled) promises $\text{prom} \in 2^T$, a coherence view of each location, $\text{coh} : \mathcal{L}oc \rightarrow \mathbb{T}$, the value and view of each register, $\text{regs} : \mathcal{R}eg \rightarrow \mathcal{V}al \times \mathbb{T}$, a read view $\text{v}_{\text{read}} : \mathbb{T}$, two write views $w_{\text{old}}, w_{\text{new}} : \mathbb{T}$ and a condition view $\text{v}_{\text{C}} : \mathbb{T}$. We write $\text{regs}(a)$ as $\kappa@t$ and also let $\text{v}_{\text{a}}$ be this view of register $a$. Finally, the evaluation of an expression $\eta$ with respect to a register assignment $\text{regs}$, $\llbracket \eta \rrbracket_{\text{regs}} \in \mathcal{V}al \times \mathbb{T}$, is defined as follows:

$$\llbracket \kappa \rrbracket_{\text{regs}} \equiv \kappa@0 \text{ for } \kappa \in \mathcal{V}al, \quad \llbracket a \rrbracket_{\text{regs}} \equiv \text{regs}(a) \text{ for } a \in \mathcal{R}eg,$$

$$\llbracket \eta \circ \eta \rrbracket_{\text{regs}} \equiv (\kappa_1@\eta_1 \oplus \kappa_2@\eta_2)(v_1 \ominus v_2) \text{ with } \llbracket \eta_1 \rrbracket_{\text{regs}} = \kappa_1@\eta_1, \llbracket \eta_2 \rrbracket_{\text{regs}} = \kappa_2@\eta_2$$

Note that this evaluation is with respect to the register function $\text{regs}$ 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

$$M_{\text{ini}} \equiv \{\text{ini}\} \quad ts_{\text{ini}} \equiv \begin{cases} \text{prom} = \{\}, \text{v}_{\text{read}} = w_{\text{new}} = w_{\text{old}} = \text{v}_{\text{C}} = 0, \text{coh} = (\lambda x. 0), \text{regs} = (\lambda a. 0@0) \end{cases}$$

where $ts_{\text{ini}}$ 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 $T$ is locally in its initial state iff for each thread $\tau$, we have $\pi_2(T(\tau)) = ts_{\text{ini}}$, where $\pi_i$ projects the $i$th component of a tuple. Given that $T$ is in its initial state, the initial concurrent program state is given by $(T, M_{\text{ini}})$.

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 $\kappa$ to $x$) of a thread at timestamp $t$, and to this end has to ensure that views $w_{\text{new}}, \text{v}_{\text{C}}, \text{coh}(x)$ as well as that of the value/register are less than $t$. It removes $t$ from the thread’s promise set and updates $\text{coh}(x)$ and $w_{\text{old}}$ (as to ensure dependencies with fences). Rule PROMISE simply adds an arbitrary new promise at the end of memory. FENCE ensures views $w_{\text{read}}$ and $w_{\text{new}}$ are updated. This rule for instance guarantees that store operations separated by barriers $\text{dmb}$ 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).
Note that the thread identifier in fulfill actions it is the thread executing the fulfill (and having made the corresponding promise).

Actions on a location \( x \) only make steps when all promises can eventually be fulfilled. Like \( \langle \text{Event structures} \rangle \), steps can be done at the beginning without losing any of the reachable states (see Lemma 31). In our semantics, all promise steps can be done at the beginning without losing any of the reachable states (see Lemma 1).

\[ \langle \text{store} x \text{ rv}, ts, M \rangle \xrightarrow{\text{store}(x, r, v)} \langle \text{skip}, ts', M \rangle \]

\[ \langle \text{store} x \text{ rv}, ts, M \rangle \xrightarrow{\text{store}(x, r, v)} \langle \text{skip}, ts', M \rangle \]

Fig. 3. Operational semantics (Atomic statement rules)

Finally, we say that \( \langle T, M \rangle \) is certifiable (used in PROGRAM STEP) if there is some \( T', M' \) such that \( \langle T, M \rangle \rightarrow^* \langle T', M' \rangle \) and \( T'.prom = \emptyset \). Certifiability ensures that a concurrent program can only make steps when all promises can eventually be fulfilled. Like [31], in our semantics, all promise steps can be done at the beginning without losing any of the reachable states (see Lemma 1).

### 4 Event Structures

Event structures [5, 6, 17, 40] 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 [17].

**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:

\[
\begin{align*}
\text{Act}^s & \equiv \bigcup_{\tau \in \text{Tid}, \kappa \in \mathcal{V}A} \{ \text{rd}_e(x, \kappa), \text{ff}_f(x, \kappa) \} \cup \{ \text{init} \} \\
\text{Act}^n & \equiv \bigcup_{\tau \in \text{Tid}} \{ \text{fnc}_r \} \\
\text{Act}^{\mathcal{A}} & \equiv \bigcup_{a \in \text{Loc}, \eta \in \mathcal{V}A} \{ \text{bar}(a, x), \text{bar}(a, \eta) \} \\
\text{Act}^{\mathcal{T}} & \equiv \bigcup_{\tau \in \text{Tid}, \beta \in \mathcal{B}E} \{ \text{fnc}_r(\beta) \}
\end{align*}
\]

Actions on a location \( x \) can be read actions \( \text{rd}_e(\cdot, \cdot) \), fulfill actions \( \text{ff}_f(\cdot, \cdot) \) or the initialization \( \text{init} \). Note that the thread identifier \( \tau \) 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 $Act^{wr}$ all fulfill actions. To record loading into register $a$, we use so called bar actions $\text{bar}(a, \cdot)$. The action $\text{fnc}$ occurs when a $\text{dmb}$ statement is executed and $\text{tst}(\cdot)$ describes the execution of some $\text{asm}$ statement.

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

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

For $L \subseteq \text{Loc}$, we write $e \stackrel{L}{\rightarrow} f$ to denote $e \rightarrow f$ 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 $\text{ini}$ be the event structure $\{(\text{ini}), \emptyset, \emptyset, \emptyset, \emptyset\}$ with $\ell(\text{ini}) = \text{ini}$. Given an event structure $E = (E, \rightarrow, \#, \Lambda, \ell)$, we similarly to actions define its set of actions labelled with specific actions as $\text{Rd}(E)$, $\text{Rd}^{\ast}(E)$, $\text{Rd}^{\ast}(E)$, $\text{Fr}(E)$, $\text{Fr}^{\ast}(E)$ and $\text{Fr}^{\ast}(E)$ via the labelling function $\ell$. For an event $e$ labelled with an action $\in Act^{\ast}(\text{ini})$, we let $e.\text{loc} = x$. We slightly abuse notation so that $e.\text{loc} = x$ for all $x$. We furthermore define $\text{last}_{\alpha}(E)$, $\alpha \in Act$, to be the last event in flow order labelled $\alpha$, i.e., $\text{last}_{\alpha}(E) = e$ if $\ell(e) = \alpha$ and for all $e'$ such that $e' \neq e$ and $\ell(e') = \alpha$, we have $e' \rightarrow e$. Moreover, $\text{last}_{\alpha}(E) = \perp$ if no event labelled $\alpha$ exists. We lift $\text{last}$ to sets of actions by $\text{last}_{\alpha}(E) = \{\text{last}_{\alpha}(E) | \alpha \in Act\}$. An event structure $E$ is **sequential** if all events are flow-ordered: $\forall e, e' \in E, e \neq e' : e \rightarrow e' \vee e' \rightarrow e$. We let $S$ 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 $E = (E, \rightarrow, \#, \Lambda, \ell)$ satisfies the following properties: (1) $C$ is cycle-free: $E \cap (C \times C)^{+}$ 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 $e \in C$, $d \rightarrow e$ and $d \notin C$, then there exists $f \in C$ such that $d \# f$ and $f \rightarrow e$.

We let $\text{Conf}(E)$ be the set of configurations of $E$. We identify a configuration with the (conflict-free) event structure $E_{C}$ which is $E$ 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 $\tau$) to synchronize with reads from promises of $\tau$. Similarly to CCS [26], we model this synchronisation via complementary actions where $\text{rd}_{\tau}(x, \kappa) = \text{ff}_{\tau}(x, \kappa)$ and vice versa, and $\overline{a} = a$. 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 $E_{1}, \ldots, E_{n}$, as follows, where $E_{i}^{*}$ denotes $E_{i} \cup \{\ast\}$.

$$ sync(E_{1}, \ldots, E_{n}) \equiv \left\{ (e_{1}, e_{2}, \ldots, e_{n}) \in E_{1}^{*} \times E_{2}^{*} \times \cdots \times E_{n}^{*} \mid \exists i, \ell_{i}(e_{i}) \in Act^{\ast} \land \forall j \neq i. e_{j} \neq \ast \Rightarrow \ell_{i}(e_{i}) = \ell_{j}(e_{j}) \land (\exists j \neq i. e_{j} \neq \ast) \right\} $$

$$ \cup \{ (e_{1}^{\delta}, e_{2}^{\delta}, \ldots, e_{n}^{\delta}) | (e_{1}^{\delta}, e_{2}^{\delta}, \ldots, e_{n}^{\delta}) \} \} $$
An event \( e \) might also occur unsynchronized in a parallel composition (which is then written as \((*,\ldots,*,e,\ldots,*\))

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 \( \Delta \)-axiom of Castellani and Zhang [7] 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 \( S \times S \times \ldots S \) 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 4.** \( \) Let \( E_1, E_2, \ldots, E_n \) be event structures for threads \( \tau_1, \tau_2, \ldots, \tau_n \), respectively. The parallel composition \( E = E_1 || E_2 || \ldots || E_n \) is the event structure \( (E, \rightarrow, \#, \Lambda, \ell) \) with

\[
\begin{align*}
- E &= \text{sync}(E_1, E_2, \ldots, E_n) \cup \left( \bigcup_i \{ (*, \ldots, *, e_i, \ldots, *) \} \right) \\
- (e_1, e_2, \ldots, e_n) &\rightarrow (d_1, d_2, \ldots, d_n) \text{ iff } \exists i. e_i \neq d_i, \\
- \Lambda((e_1, e_2, \ldots, e_n), (d_1, d_2, \ldots, d_n)) &= \bigcup_i \Lambda(e_i, d_i), \\
- (e_1, e_2, \ldots, e_n) &\#(d_1, d_2, \ldots, d_n) \text{ iff} \\
- \bullet &\exists i. e_i \neq d_i, \text{ or} \\
- \bullet &\exists i, j. e_i = d_i \wedge e_i \neq * \wedge e_j \neq d_j \quad \text{(inherits conflicts)} \\
- \bullet &\ell(e_1, e_2, \ldots, e_n) = \begin{cases} 
\ell(e_i) & \text{if } (e_1, e_2, \ldots, e_n) \in \text{sync}(E_1, E_2, \ldots, E_n) \wedge \ell(e_i) = \ell (\ldots) \\
\ell(e_i) & \text{if } (e_1, e_2, \ldots, e_n) \notin \text{sync}(E_1, E_2, \ldots, E_n) \wedge e_i \neq * 
\end{cases} \quad \text{(conflicts on differently paired events)}
\end{align*}
\]

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 \( E_i \) to \( E_n \) are synchronization if \( \pi_i(\text{sync}(E_1, \ldots, E_n)) \supseteq \text{Rd}(E_i), i \in \{1, \ldots, n\} \) (all the reads have a synchronization with a fulfill). The configuration (describing an execution of the parallel composition of threads) which we extract from \( \text{Conf}(E_1 || \ldots || E_n) \) 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 \( E_C = (E_C, \rightarrow_C, \#, \Lambda_C, \ell_C) \) corresponding to a configuration \( C \in \text{Conf}(E_1 || \ldots || E_n) \) is interference free if

1. \( C \) is thread-covering: \( \forall i \in \{1, \ldots, n\} : \pi_i(E_C) = E_i, \)
2. \( C \) is memory-consistent linearizable: there exists a total order \( \prec \subseteq \text{Act}^r(E_C) \times \text{Act}^r(E_C) \) among reads, fulfills and the ini event such that

\[
\begin{align*}
- \rightarrow_C^+ \cap (\text{Act}^r(E_C) \times \text{Act}^r(E_C)) &\subseteq \prec \quad \text{and} \\
- \forall d, e, f \in E_C: d \rightarrow f \wedge d \prec e \prec f &\Rightarrow e.\text{loc} \notin L,
\end{align*}
\]
3. \( C \) contains no unsynchronized reads: there is no event in \( E_C \) of the form \((*,*,\ldots,*,e_i,*\ldots,*\)) where \( e_i \in \text{Rd}(E_i) \).
Example 1. Consider the two event structures given next (which belong to a message passing program with barriers, see Section 5).

\[ e_{\text{ini}} : \text{ini} \rightarrow e_1 : \text{ff}_1(x, 5) \]
\[ e_2 : \text{fn}_1 \rightarrow e_3 : \text{ff}_1(y, 1) \]
\[ f_{\text{ini}} : \text{ini} \rightarrow f_1 : \text{rd}_1(f, 1) \rightarrow f_2 : \text{bar}(a, y) \]
\[ f_{\text{ini}} : \text{ini} \rightarrow f_3 : \text{bar}(b, x) \]

Their parallel composition gives the following event structure:

\[ (e_{\text{ini}}, f_{\text{ini}}) : \text{ini} \rightarrow (e_1, \ast) : \text{ff}_1(x, 5) \rightarrow (e_2, \ast) : \text{fn}_1 \rightarrow (e_3, \ast) : \text{ff}_1(y, 1) \]
\[ (e_1, \ast) : \text{ff}_1(x, 5) \rightarrow (e_2, \ast) : \text{fn}_1 \rightarrow (e_3, \ast) : \text{ff}_1(y, 1) \]
\[ (e_3, f_1) : \text{ff}_1(y, 1) \rightarrow (\ast, f_2) : \text{bar}(a, y) \]
\[ (\ast, f_3) : \text{bar}(b, x) \]

This event structure has no interference-free configuration. To satisfy the conditions “thread-covering” and “no unsynchronised reads”, we must include event \((e_1, f_1)\). This means that the only possible configuration must also include the downclosure \((e_1, \ast)\) and \((e_2, \ast)\). However, together with the location restriction \(\{x\}\) on the edge \(((e_{\text{ini}}, f_{\text{ini}}), (e_3, f_1))\), the resulting event structure is not memory-consistent linearizable, since it contains a sequence \((e_{\text{ini}}, f_{\text{ini}}) \rightarrow (e_1, \ast) \rightarrow (e_2, \ast) \rightarrow (e_3, f_1)\), where \((e_1, \ast)\) corresponds to a fulfilled write on \(x\) that is forbidden by the edge \(((e_{\text{ini}}, f_{\text{ini}}), (e_3, f_1))\).

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 \(E\), where \(E\) 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 \(\tau\) can have fence and fulfill events of \(\tau\), 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 \(E\) – together with some memory \(M\) – allow us to compute the current views of threads.

A local assertion of a thread \(\tau\) defines constraints on the global memory \(M\) (the ordering of writes and their values) as well as the views of \(\tau\): An assertion \(E\) describes a set of states \([E] = \{\langle tS, M \rangle \in (\text{Tid} \rightarrow \text{TState}) \times \text{Memory} \mid \langle tS, M \rangle \text{ matches } E\}\) where “matches” is defined by conditions (1)-(4) below.

(1) \(M\) is consistent with the fulfill and read events of \(E\).

There exists a total mapping \(\psi : \text{Ff}(E) \cup \text{Rd}(E) \cup \{e_{\text{ini}}\} \rightarrow \text{dom}(M)\) which
\[ prFnc_n(\mathcal{E}) = \{ e \in Rd(\mathcal{E}) \cup \text{Fs}(\mathcal{E}) \cup \{ e_{\text{in}} \} \mid \exists e' \in \text{last}n, (\mathcal{E}) : e \rightarrow e' \} \]
\[ prBar_n(\mathcal{E}) = \{ e \in Rd(\mathcal{E}) \cup \text{Fs}(\mathcal{E}) \cup \{ e_{\text{in}} \} \mid \exists e' \in \text{last}bar_{(a)}(\mathcal{E}) : e \rightarrow e' \} \]
\[ prBar_n(\mathcal{E}) = \bigcup_{a \in R(\tau)} prBar_n(\mathcal{E}) \]
\[ prBar^+_n(\mathcal{E}) = prBar_n(\mathcal{E}) \cap \text{Act}^+(\mathcal{E}) \]
\[ prTst_n(\mathcal{E}) = \{ e \in Rd(\mathcal{E}) \cup \text{Fs}(\mathcal{E}) \cup \{ e_{\text{in}} \} \mid \exists e' \in \text{last}tst_{(a)}(\mathcal{E}) : e \rightarrow e' \} \]

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

1. initializes at zero: the one event \(e_{\text{in}}\) labeled ini is mapped to 0.
2. is consecutive for every thread \(\tau:\)
   - for all \(e \in \text{Fs}(\mathcal{E}), t \in \mathbb{T} \text{ s.t. } M(t) = (x := \kappa), t < \psi(e)\) and \(e.loc = x\), there exists \(d \in \text{Fs}(\mathcal{E})\) such that \(\psi(d) = t\).
3. preserves content: if \(\psi(e) = t \neq 0\) and \(M(t) = (x := \kappa)\), then \(\ell(e) \in \{\text{ff}(x, \kappa), \text{rd}_{(x, \kappa)}\}\),
4. preserves flows: \(\forall e, e' \in \text{dom}(M) : e \rightarrow e' \Rightarrow \psi(e) < \psi(e')\),
5. and preserves memory constraints:
   \[ \forall d, e \in \text{dom}(M), L \subseteq \text{Loc} \text{ s.t. } d \xrightarrow{L} e, \forall t \in \mathbb{T} \text{ s.t. } \psi(d) < t < \psi(e) : M(t).loc \neq d.loc. \]

The mapping \(\psi\) 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 \(\psi(e_{\text{in}}) = 0\)).

(2) The open (non-fulfilled) promises of a thread \(\tau\) are the entries of \(\tau\) in \(M\) which are not fulfilled, i.e., \(ts(\tau).\text{prom} = M_{\tau} \setminus \psi(\text{Fs}(\mathcal{E}))\).

(3) The views of a thread \(\tau\) are consistent with mapping \(\psi\) and \(M\).

Letting \(ts = ts(\tau), a \in R(\tau)\) and \(x \in \text{Loc}\), we have

\[ ts.v_C = \bigcup_{e \in \text{prTst}_{(a)}(\mathcal{E})} \psi(e) \]

\[ ts.coh(x) = \bigcup_{e \in \text{Fs}(\mathcal{E}) \cup \text{prBar}_{(a)}(\mathcal{E})} \psi(e) \]

\[ ts.v_{\text{wOld}} = \bigcup_{e \in \text{Fs}(\mathcal{E})} \psi(e) \]

\[ ts.v_{\text{wNew}} = \bigcup_{e \in \text{prFnc}_{(a)}(\mathcal{E}) \setminus \text{Fs}(\mathcal{E}) \cup \text{prBar}_{(a)}(\mathcal{E})} \psi(e) \]

\[ zts.v_{\text{a}} = \bigcup_{e \in \text{prBar}_{(a)}(\mathcal{E})} \psi(e) \]

\[ ts.v_{\text{read}} = \bigcup_{e \in \left(\text{prFnc}_{(a)}(\mathcal{E}) \setminus \text{Fs}(\mathcal{E})\right) \cup \text{prBar}_{(a)}(\mathcal{E})} \psi(e) \]

(4) The values of registers \(R(\tau)\) of thread \(\tau\) agree with values in \(\mathcal{E}\).

For \(a \in \text{REG}, ts.reg(a) = \kappa \otimes v_a\) with \(\kappa = \llbracket a \rrbracket_{\mathcal{E}}\) (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 \(\text{bar}(a, \cdot)\) (on \(x\)) or (3) the value of the expression \(\eta\) in a last \(\text{bar}(a, \eta)\)) 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 \(\text{ff}_1(y, 1)\), so \(\psi\)
The definition of these operators has to ensure that they capture the dependencies between views that are local to threads and describe a single execution of the thread, hence are conflict-free. 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.

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

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 the following 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 $E \oplus \mathbb{f}_r(x, \kappa)$ has to introduce a flow from the last test event to the newly added fulfill event.

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 $[E]$) 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 $[E]$) and $v_{read} \cup coh(x)$. This is achieved by inserting an additional location restriction $x$ via the operator $rstr^r_x(E)$ to the following (potentially already $L$-labelled) flows (thus getting the restriction $L \cup \{x\}$):

$$\{e \xrightarrow{L} e' \mid e' \in (prFnc_r(E) \cap \mathbb{F}_r(E)) \cup prBar_r(E) \cup \mathbb{F}^r_x(E)\}.$$  

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 $E'$) 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: $a := \text{load } y$; |
| 2: dmb; | 5: $b := \text{load } x$; |
| 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 need to apply rule PR-READNEW for

\[
\begin{align*}
\text{PR-WRITE} & \quad \text{PR-WRITER} & \quad \text{PR-FENCE} \\
\{E\} \text{ store } x \kappa & \quad \{E\} \text{ store } a \kappa & \quad \{E\} \text{ dmb}_a \{E \oplus \mathbb{f}_r(x, \kappa)\} \\
\text{PR-READEX} & \quad \text{PR-READNEW} \\
& \quad e = \text{last}_{\text{Act}}(E) & \quad E' = (E', \rightarrow', A', \ell') \in S \\
& \quad \ell(e) \in (\text{rd}_{\ell'}(x, \kappa), \mathbb{f}_r(x, \kappa), \text{ini}) & \quad \ell'(E') \subseteq \text{Act}^{\text{rd}} \setminus \text{Act}_r \\
\{E\} \text{ a := load } x & \quad [rstr^r_x(E \oplus \text{bar}(a, x))] & \quad \{E\} \text{ a := load } x \{[E \oplus E'] \oplus \text{bar}(a, x)\} \\
\text{PR-REGISTERS} & \quad \text{PR-ASSUME} & \quad \text{PR-CHOICE} \\
\{E\} \text{ a := } \eta & \quad [\beta]_E = \text{true} & \quad \{E\}_1 S_1 + S_2 \{E\}_2 \\
\{E\} \text{ a := } \eta \{E \oplus \text{bar}(a, \eta)\} & \quad [E] \text{ asm } \beta \{E \oplus \text{tst}_r(\beta)\} & \quad \{E\}_1 S_1 + S_2 \{E\}_2 \\
\{E\} \text{ S } \{E\} & \quad \{E\}_1 S_1 + S_2 \{E\}_2 \\
\{E\}_1 S_1; S_2 \{E\}_3 & \quad \{E\}_1 S_1; S_2 \{E\}_2 \\
\{E\}_1 S_1; S_2 \{E\}_3 & \quad \{E\}_1 S_1; S_2 \{E\}_2 \\
\end{align*}
\]

Fig. 7. Local proof rules for a thread $\tau$.  

12 Heike Wehrheim, Lara Bargmann, and Brijesh Dongol
the first load giving us
\[
ini \rightarrow rd_1(x, 5) \rightarrow rd_1(y, 1) \rightarrow bar(a, y)
\]
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.

\[
\forall i \in \{1, \ldots, n\}. \ [ini] S_i [\mathcal{E}_i] \quad \mathcal{E}_1, \ldots, \mathcal{E}_n \text{ synchronisable} \\
\text{interference free} \quad C \in Conf(\mathcal{E}_1 || \ldots || \mathcal{E}_n) \\
\implies [ini] S_1 || \ldots || S_n [\mathcal{E}_C]
\]

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 \((a = 1 \land b = 0)\) (as calculated by \([a]_E\) and \([b]_E\) taking the value of the last fulfill or ini event prior to the last bar event on \(a\) and \(b\), respectively).

\[
\begin{align*}
\text{Thread 1} & \\
[ini] \quad 1 : \text{store} x 5; & \quad [ini] \quad 4 : a := \text{load} y; \\
& \quad 2 : \text{store} y 1; & \quad [ini] \quad 5 : b := \text{load} x; \\
\begin{align*}
& \quad \text{ini } \xrightarrow{} \text{ff}_1(x, 5) \\
& \quad \text{ini } \xrightarrow{} \text{ff}_1(y, 1) \\
\end{align*} & \begin{align*}
& \quad \text{ini } \xrightarrow{} \text{bar}(a, y) \\
\end{align*} \\
\end{align*}
\]

\[
\begin{align*}
\text{Thread 2} & \\
[ini] \quad & \begin{align*}
& \quad [ini] \quad \text{rd}_1(y, 1) \rightarrow \text{bar}(a, y) \\
& \quad [ini] \quad \text{bar}(b, x) \\
\end{align*} \\
& \begin{align*}
& \quad \text{bar}(b, x) \\
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} : & \begin{align*}
& \quad \text{ini } \xrightarrow{} \text{bar}(b, x) \\
& \quad \text{ini } \xrightarrow{} \text{ff}_1(x, 5) \\
\end{align*} \\
\end{align*}
\]

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 appendix.

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 \([\text{Ini}]\), \([S_i, E_i], i \in \{1, \ldots, n\}\), be proof outlines of threads \(\tau_1\) to \(\tau_n\) such that \(E_i\) to \(E_n\) are synchronizable and let \(T_0\) be an initial thread pool with \(T_0(\tau_i) = (S_i, ts_{\text{ini}})\) and \(M_0 = M_{\text{ini}}\).

Then, for every thread pool \(T\) with \(T(\tau_i) = (\text{skip}, ts_i)\), interference free configuration \(C \in \text{Conf}(E_1 \| \ldots \| E_n)\) and memory \(M\) such that \((ts_i, M) \in \llbracket E_i \rrbracket\), \(\text{tids}(M) = \{\tau_1, \ldots, \tau_n\}\) and \(ts_i.prom = \emptyset, i \in \{1, \ldots, n\}\), we have \(\langle T_0, M_0 \rangle \rightarrow^* \langle T, M \rangle\).

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

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

Then there are local proof outlines \([\text{Ini}]\), \([S_k, E_k]\) of threads \(\tau_k, k \in \{1, \ldots, n\}\), such that \(E_1\) to \(E_n\) are synchronizable and there exists an interference free configuration \(C \in \text{Conf}(E_1 \| \ldots \| E_n)\) with \(\langle T, M \rangle \in \llbracket E_C \rrbracket\).

6 Related Work

The first semantics of weak memory models employing promises has been proposed by Kang et al. in 2017 [22] 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. [31]. In particular, like [31] 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, 10, 11, 13–15, 23, 34, 38]. The only one explicitly dealing with promises in the semantics is the proposal of Svendsen et al. [37]. 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 [22]), 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 [8, 19, 20], but not for reasoning. Wright et al [41] 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 [30], which is a partial order over a thread’s execution. An Owicky-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 [28], with a predecessor approach with (almost) the same principles by de Vries and Koutavas [39]. The first extension of incorrectness logic to concurrent programs has been proposed by Raad et al. in the form of an incorrectness separation logic [32] which is however not compositional.

Colvin [9] 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 \(\langle p \rangle s \langle q \rangle = \neg\{\neg p\} s \neg\{\neg q\}\), 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. Boehm, H., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI. pp. 68–78. ACM (2008). https://doi.org/10.1145/1375581.1375591, https://doi.org/10.1145/1375581.1375591
5. 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
6. 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
7. 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
8. 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
9. 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
10. 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
11. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries Reasoning for C11 RAR. In: Hirsfeld, 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
12. 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

13. 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

14. 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

15. 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

16. 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: Bodik, R., Majumdar, R. (eds.) POPL. pp. 608–621. ACM (2016). https://doi.org/10.1145/2837614.2837615

17. 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

18. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)

19. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with precondition: a simple model of relaxed memory. Proc. ACM Program. Lang. 4(OOPSLA), 194:1–194:30 (2020). https://doi.org/10.1145/3428262

20. 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

21. 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

22. 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

23. 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

24. 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

25. Maranget, L., Sarkar, S., Sewell, P.: A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf (2012), accessed: May, 2022

26. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)

27. 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

28. 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

29. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. 6, 319–340 (1976)
30. 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
31. 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
32. 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
33. 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
34. 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
35. Sarkar, S., Sewell, P., Aghvami, 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
36. 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
37. 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
38. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: Hosking, A.L., Engst, P.T., Lopes, C.V. (eds.) OOPSLA. pp. 867–884. ACM (2013). https://doi.org/10.1145/2509136.2509532
39. de Vries, E., Koutavas, V.: Reverse Hoare Logic. In: Bâthe, W., 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
40. 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
41. 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

A Further examples

**RRC – Read Read Coherence** In the read read coherence litmus test the first two threads store different values to \( x \), while the other two threads load \( x \) into different registers. The outcome \( (a = 1 \land b = 2 \land c = 2 \land d = 1) \) should not be allowed and indeed, the parallel composition of local proof outlines does not give us an interference-free configuration. Here, we also give the event names in the assertions.
Here, the parallel composition of the local assertion event structures contains a cycle

\[
\begin{align*}
(e_{\text{init}}, e_{\text{init}}, e_{\text{init}}, e_{\text{init}}) &\xrightarrow{(*, *, *, e_8)} (e_{1}, *, e_3, e_9) &\xrightarrow{(*, *, e_4, *)} \\
(*, *, *, e_{10}) &\xrightarrow{(*, e_2, e_5, e_7)} (*, *, e_6, *)
\end{align*}
\]

and therefore has no interference free configuration.

**WRC – Write-to-Read Causality** The next litmus test is the WRC example, originally formulated by Boehm and Adve [4] and appearing here in the form of [25].

We see that due to the lack of a barrier in between statements 2 and 3 in Thread 2, the final outcome \((a = 1 \land b = 1 \land c = 0)\) is reachable.
SB – Store buffering

Next, we take a look at the standard SB example. We see that the outcome $(a = 0 \land b = 0) \checkmark$ which is possible in lots of weak memory models is also possible here, and can be proven to be reachable.

IRIW – Independent Reads of Independent Writes

In the IRIW litmus test we have two threads writing to shared variables $x$ and $y$, respectively, and two other threads both reading both variables, but in a different order. The outcome $(a = 1 \land b = 0 \land c = 1 \land d = 0) \times$ is not allowed by our semantics.
And indeed, while we can construct local event structures for each thread, the one candidate for a configuration of their parallel composition is

\[
\begin{bmatrix}
\text{ini} \\
\{x\} \\
\{y\} \\
\text{bar}(b, y) \\
\text{ff}_3(y, 1) \\
\text{bar}(c, y) \\
\text{bar}(d, x) \\
\text{ff}_1(x, 1) \\
\text{bar}(a, x)
\end{bmatrix}.
\]

But this configuration is not interference free (because of the location restrictions there is no valid linearization of the read and fulfill events), and we hence (correctly) cannot show outcome \((a = 1 \land b = 0 \land c = 1 \land d = 0)\) \(\not\in\) to be reachable.

### B Proofs of Soundness and Completeness

#### B.1 Properties of the Operational Semantics

We start with two properties of the operational semantics. The first ensures that in any trace, the promises can be reordered so that they are executed at the start of the trace without affecting the final outcome.

**Lemma 1.** For every trace \(tr\) there exists a trace \(tr'\) (ending in the same final state) such that in \(tr'\) all promise actions of \(tr\) occur at the beginning followed by the non-promise actions of \(tr\) (in the same order as in \(tr\)).

**Proof.** Let \((\langle T_0, M_0 \rangle \xrightarrow{op_1} \langle T_1, M_1 \rangle \xrightarrow{op_2} \ldots \xrightarrow{op_k} \langle T_k, M_k \rangle)\) be a trace \(tr\) with \(I = \{i \mid op_i = \text{rd} (\cdot, \cdot)\}\) being the set of all promises. Let \(i_1 < i_2 < \ldots\) be the elements of \(I\). We now want to swap \(op_{i_1}\) with the operation before it, until \(op_{i_1}\) is the first one in the trace. After that we do the same with \(op_{i_2}\) until it is the second operation and so on. For that we need to show that we can swap a promise with a fence, fulfill or read operation before it.

Let \(op_i\) be a promise operation, \(l\) its location and \(op_{i-1}\) the operation in front of \(op_i\). If we promise something, we change \(prom\) to \(prom \cup \{\mid M_{i-1} \mid + 1\}\) in \(ts\) and add one element to the memory. In case that \(op_{i-1}\) is a fence operation we can reorder \(op_i\) and \(op_{i-1}\), because a fence operation changes different attributes of \(ts\) (\(v_{\text{read}}\) and \(v_{\text{write}}\)) and has no effects on the memory.

If \(op_{i-1}\) is a fulfill operation, it fulfills a different promise. Hence the condition

\[v_{\text{write}} \cup ts.\text{cof}(l) \sqcup v_a < t\]

still applies after the reordering, because \(t\) depends on the promise \(op_{i-1}\) is fulfilling. The changes of \(\text{cof}\) and \(v_{\text{write}}\) are not related to the changes caused by \(op_i\). Additionally the element deleted by \(op_{i-1}\) in \(prom\) is not the same as the element that \(op_i\) adds. So we can also reorder a promise and a fulfill.
In case that \( op_{i-1} \) is a read operation, \( op_{i-1} \) changes other attributes of \( ts \) then \( op_i \) would. If \( op_{i-1} \) reads from a location \( l' \), which is different from \( l \), the condition
\[
\forall t'. \, t < t' \leq (ts.\text{read} \cup ts.\text{coh}(l)) \Rightarrow M(t').\text{loc} \neq l
\]
holds. However, when \( op_{i-1} \) reads from \( l \), we need to have a deeper look at \( v_{\text{read}} \) and \( \text{coh} \). Both will only be replaced by timestamps of entries in \( M_{i-1} \). Those timestamps are always smaller than \(|M_{i-1}| + 1\), so the condition is still true. Hence we can reorder a read and a promise operation.

That means we can reorder the operations of our execution in a way that all promises are at the beginning and the final state stays the same.

The second property ensures that views of each component monotonically increase after each transition. Note that this differs from the semantics of Pulte et al., where register views might decrease after the execution of a write to a local register [31].

**Lemma 2 (View monotonicity).** Suppose \( \langle\langle S, ts\rangle, M\rangle \xrightarrow{\Delta E \tau} \langle\langle S', ts'\rangle, M'\rangle \). Then for view \( v \in \{v_{\text{read}}, v_{w\text{New}}, v_{w\text{Old}}, v_C\} \), we have \( ts.v < ts'.v \). Moreover, for all locations \( x \), we have \( ts.\text{coh}(x) \leq ts'.\text{coh}(x) \), and for all registers \( a \), we have that if \( ts.\text{regs}(a) = \kappa @v \) and \( ts'.\text{regs}(a) = \kappa' @v' \), then \( v \leq v' \).

**Proof.** Let \( \langle\langle S, ts\rangle, M\rangle \xrightarrow{\Delta E \tau} \langle\langle S', ts'\rangle, M'\rangle \).

- \( v_C \) only changes for \( op = \text{asm}(bexp) \): \( ts'.v_C = ts.v_C \cup ts.v \geq ts.v_C \)
- \( v_{w\text{Old}} \) only changes for \( op = \text{ff}(x, \kappa) \): \( ts'.v_{w\text{Old}} = ts.v_{w\text{Old}} \cup t \geq ts.v_{w\text{Old}} \)
- \( \text{coh}(x) \) changes in two cases:
  - for \( op = \text{rd}(x, \kappa) \): \( ts'.\text{coh}(x) = ts.\text{coh}(x) \cup v_{\text{post}} \geq ts.\text{coh}(x) \)
  - for \( op = \text{ff}(x, \kappa) \): \( ts'.\text{coh}(x) = t > ts.\text{coh}(x) \) because \( ts.v_{w\text{New}} \cup ts.v_C \cup ts.\text{coh}(x) \cup v_a \leq t \)
- \( v_{\text{read}} \) changes in two cases:
  - for \( op = \text{rd}(x, \kappa) \): \( ts'.v_{\text{read}} = v_{\text{post}} = ts.v_{\text{read}} \cup t \geq ts.v_{\text{read}} \)
  - for \( op = \text{fnc} \): \( ts'.v_{\text{read}} = ts.v_{\text{read}} \cup ts.v_{w\text{Old}} \geq ts.v_{\text{read}} \)
- \( v_{w\text{New}} \) only changes for \( op = \text{fnc} \): \( ts'.v_{w\text{New}} = ts.v_{w\text{New}} \cup ts.v_{w\text{Old}} \geq ts.v_{w\text{New}} \). By looking at \( v_{\text{read}} \) we see that \( ts'.v_{\text{read}} \geq ts.v_{\text{read}} \) holds in all cases of \( op \). Since \( v_{\text{read}} \) changes whenever \( v_{w\text{New}} \) does and both are set to the same value we get \( ts.v_{\text{read}} = ts.v_{w\text{New}} \). Therefore \( ts'.v_{w\text{New}} \geq ts.v_{w\text{New}} \).
- For \( \text{regs}(a) = \kappa @v \) we first show that \( ts.v_a \leq ts.v_{\text{read}} \) is an invariant. We know that \( v_a \) only changes for \( op = \text{rd}(x, \kappa) \) and \( op = \text{lst}(a, \exp) \) whereas \( v_{\text{read}} \) changes for \( op = \text{rd}(x, \kappa) \) and \( op = \text{fnc} \). For \( op = \text{rd}(x, \kappa) \) \( v_a \) and \( v_{\text{read}} \) are set to the same value. Since \( ts'.v_{\text{read}} \geq ts.v_{\text{read}}, \) \( v_a \) can not get bigger than \( v_{\text{read}} \) after a register operation. Therefore our invariant holds and we get:
  - for \( op = \text{rd}(x, \kappa) \): \( ts'.v_a = ts.v_{\text{read}} \cup t \geq ts.v_{\text{read}} \geq ts.v_a \)
  - since \( op = \text{lst}(a, \exp) \) changes \( v_a \) to the maximum of \( v_a \) and the view of \( \exp \), \( ts'.v_a \geq ts.v_a \)

**B.2 Soundness**

Next we prove soundness of our proof rules. We first prove correctness of each of the atomic rules from Fig. 7.
Proposition 1 (Soundness of rule PR-WriteC). Let \( \langle ts', M \rangle \in \llbracket E \oplus \text{ff}_\tau(x, \kappa) \rrbracket \). Then there exists \( \langle ts, M \rangle \in \llbracket E \rrbracket \) such that

\[ \langle \text{store } x \kappa, ts \rangle, M \rangle \xrightarrow{\text{ff}_\tau(x, \kappa)} \langle \text{skip}, ts' \rangle, M \rangle. \]

Proof. We write \( E' \) for \( E \oplus \text{ff}_\tau(x, \kappa) \). Let \( \langle ts', M \rangle \in \llbracket E' \rrbracket \) and let \( e' \) be the event such that \( \ell(e') = \text{ff}_\tau(x, \kappa) \). This means that there exists some mapping \( \psi': \text{Ff}(E') \cup \text{Rd}(E') \to \text{dom}(M) \) that shows \( \langle ts, M \rangle \) satisfies \( E' \). In particular, \( \psi'(e') = t \) implies that \( M(t) = \langle x := \kappa \rangle_\tau \). We show that there exists some \( \langle ts, M \rangle \in \llbracket E \rrbracket \) such that the following holds (eliding the program statements):

\[ \langle ts, M \rangle \xrightarrow{\text{ff}_\tau(x, \kappa)} \langle ts', M \rangle \]

Note that \( M \) is unchanged and \( t \) in the rule \text{Fulfill} is instantiated to \( ts'.\text{coh}(x) \). We define \( ts \) to be the following by using \( \psi' \):

- \( ts'.\text{prom} := ts'.\text{prom} \cup \{ ts'.\text{coh}(x) \} \)
- \( ts'.\text{coh}(x) := \bigcup_{e \in \text{Ff}(E')} \psi'(e) \)
- \( ts.v_{w\text{Old}} := \bigcup_{e \in \text{Ff}(E')} \psi'(e) \)
- \( ts.v_a := 0 \)
- \( ts.v_C := ts'.v_C \)
- \( ts.v_{\text{read}} := ts'.v_{\text{read}} \)

First, we show \( \langle ts, M \rangle \in \llbracket E \rrbracket \) by checking the conditions in Section 5.1.

1. We define \( \psi = \{ e' \} \triangleleft \psi' \), where \( \triangleleft \) denotes domain anti-restriction. Clearly \( M \) remains consistent with \( E \).
2. This is trivial. We have one additional open promise in \( ts'.\text{prom} \), but this is precisely the fulfilled write in \( \text{E}' \).
3. –

\[ ts.v_C = ts'.v_C \]

by definition

\[ = \bigcup_{e \in \text{prTst}_\tau(E')} \psi'(e) \]

unfolding definition of \( ts'.v_C \)

\[ = \bigcup_{e \in \text{prTst}_\tau(E')} \psi'(e) \]

\( \text{prTst}_\tau(E') = \text{prTst}_\tau(E) \)

\[ = \bigcup_{e \in \text{prTst}_\tau(E)} \psi(e) \]

\( \text{prTst}_\tau(E) \subseteq \text{dom}(\psi) \)

\[ ts.\text{coh}(x) = \bigcup_{e \in \text{Ff}(E) \cup \text{prBar}_\tau(E)} \psi'(e) \]

definition

\[ = \bigcup_{e \in \text{Ff}(E) \cup \text{prBar}_\tau(E)} \psi(e) \]

\( \text{Ff}_\tau(E) \cup \text{prBar}_\tau(E) \subseteq \text{dom}(\psi) \)
Reasoning about Promises in Weak Memory Models with Event Structures (Extended Version)

\( ts.v_{w\text{Old}} = \bigsqcup_{e \in \text{Ff}(\mathcal{E})} \psi'(e) \quad \text{definition} \)
\( = \bigsqcup_{e \in \text{Ff}(\mathcal{E})} \psi(e) \qquad \text{Ff}(\mathcal{E}) \subseteq \text{dom}(\psi) \)

\( ts.v_{w\text{New}} = ts'.v_{w\text{New}} \quad \text{by definition} \)
\( = \bigsqcup_{e \in \text{prFnc}_\tau(\mathcal{E}') \cap (\text{Ff}_\tau(\mathcal{E}') \cup \text{prBar}_\tau(\mathcal{E}'))} \psi(e) \quad \text{since prFnc}_\tau(\mathcal{E}') = \text{prFnc}_\tau(\mathcal{E}) \)
\( = \bigsqcup_{e \in \text{prFnc}_\tau(\mathcal{E}) \cap (\text{Ff}_\tau(\mathcal{E}) \cup \text{prBar}_\tau(\mathcal{E}))} \psi(e) \quad \text{logic} \)

\( ts.v_{\text{read}} = ts.v_{\text{read}} \quad \text{by definition} \)
\( = \bigsqcup_{e \in (\text{prFnc}_\tau(\mathcal{E}') \cap \text{Ff}_\tau(\mathcal{E}')) \cup \text{prBar}_\tau(\mathcal{E})} \psi(e) \quad \text{unfolding} \)
\( = \bigsqcup_{e \in \text{prFnc}_\tau(\mathcal{E}) \cap (\text{Ff}_\tau(\mathcal{E}) \cup \text{prBar}_\tau(\mathcal{E}))} \psi(e) \quad \text{logic} \)

4. Trivial since no register value is modified.

Recall:
\( \mathcal{E} \oplus \text{ff}_\tau(x, \kappa) = (E \cup \{e\}, \rightarrow \cup \{(e', e) \mid e' \in \text{last}_{\text{Act}}^{x \cup \{\text{inc}_{\cdot}, \text{tst}_{\cdot}\}}(\mathcal{E}), \Lambda, \ell[e \mapsto \text{ff}_\tau(x, \kappa)]\}) \)

Next, we have \( (ts, M) \xrightarrow{\text{ff}_\tau(x, \kappa)} (ts', M) \) as follows:
- \( ts'.\text{prom} = ts.\text{prom} \setminus \{t\} \) by definition
- \( \text{regs}(\kappa) = \kappa@0 \) because \( ts.v_a = ts'.v_a \) and \( ts.\text{regs}(a) = ts'.\text{regs}(a) \)
- \( M(t) = \langle x := \kappa \rangle \), by definition
- \( ts.v_{w\text{New}} \sqcup ts.v_C \sqcup ts.\text{coh}(x) \sqcup ts.v_a < ts'.\text{coh}(x) \). We prove maximality of \( ts'.\text{coh}(x) \) for each component:
  - \( ts.v_{w\text{New}} < ts'.\text{coh}(x) \)
    Recall that \( ts.v_{w\text{New}} = \bigsqcup_{e \in \text{prFnc}_\tau(\mathcal{E}) \cap (\text{Ff}_\tau(\mathcal{E}) \cup \text{prBar}_\tau(\mathcal{E}))} \psi(e) \). By definition of \( \mathcal{E} \oplus^n \text{ff}_\tau(x, \kappa) \), we have that the event corresponding to \( \text{ff}_\tau(x, \kappa) \) is causally after each \( e \in \text{prFnc}_\tau(\mathcal{E}) \). Thus, \( ts.v_{w\text{New}} < ts'.\text{coh}(x) \).
• \(ts.v_C < ts'.coh(x)\).
  Recall that \(ts.v_C = \bigcup_{e \in prTst_r(\mathcal{E})} \psi(e)\). By definition of \(\mathcal{E} \oplus^a \mathcal{F}_r(x, \kappa, \kappa)\), we have that the event corresponding to \(\langle x, \kappa, e \rangle\) is causally after each \(e \in prTst_r(\mathcal{E})\). Thus \(ts.v_C < ts'.coh(x)\).
• \(ts.coh(x) < ts'.coh(x)\) because by definition of \(\mathcal{E} \oplus^a \mathcal{F}_r(x, \kappa, \kappa)\) we have a causal edge from each \(e \in \text{Act}^x\) to the event corresponding to \(\mathcal{F}_r(x, \kappa)\).
• \(ts.v_a < ts'.coh(x)\) because \(v_a = v_\kappa = 0\).

**Proposition 2 (Soundness of rule PR-WriteR).** Let \(\langle ts', M \rangle \in [\mathcal{E} \oplus^a \mathcal{F}_r(x, \kappa)\] and \([a]_\mathcal{E} = \kappa\). Then there exists \(\langle ts, M \rangle \in [\mathcal{E}]\) such that
\[\langle \text{store } x \ a,ts \rangle, M \xrightarrow{\mathcal{F}_r(x, \kappa)} \langle \text{skip}, ts' \rangle, M\].

**Proof.** We write \(\mathcal{E}'\) for \(\mathcal{E} \oplus^a \mathcal{F}_r(x, \kappa)\). Let \(\langle ts', M \rangle \in [\mathcal{E}]\) and let \(e'\) be the event such that \(\ell(e') = \mathcal{F}_r(x, \kappa)\). This means that there exists some mapping \(\psi' : \mathcal{F}(\mathcal{E}') \cup \text{Rd}(\mathcal{E}') \to \text{dom}(M)\) that shows \(\langle ts, M \rangle\) satisfies \(\mathcal{E}'\). In particular, \(\psi'(e') = t\) implies that \(M(t) = \langle x := \kappa \rangle_r\). We show that there exists some \(\langle ts, M \rangle \in [\mathcal{E}]\) such that the following holds (eliding the program statements):
\[\langle ts, M \rangle \xrightarrow{\mathcal{F}_r(x, \kappa)} \langle ts', M \rangle\]

Note that \(M\) is unchanged and \(t\) in the rule Fulfill is instantiated to \(ts'.coh(x)\). We define \(ts\) to be the following by using \(\psi'\):
\[
\begin{align*}
  ts.prom := ts'.prom \cup \{ts'.coh(x)\} & &
  ts.coh(x) := \bigcup_{e \in \mathcal{F}(\mathcal{E})} \psi'(e) \\
  ts.v_{wOld} := \bigcup_{e \in \mathcal{F}(\mathcal{E})} \psi'(e) & &
  ts.v_{wNew} := ts'.v_{wNew} \\
  ts.v_a := ts'.v_a & &
  ts.v_{read} := ts'.v_{read} \\
  ts.v_C := ts'.v_C
\end{align*}
\]

First, we show \(\langle ts, M \rangle \in [\mathcal{E}]\) by checking the conditions in Section 5.1.

1. We define \(\psi = \{e' \lor \psi'\}, \) where \(\lor\) denotes domain anti-restriction. Clearly \(M\) remains consistent with \(\mathcal{E}\).
2. This is trivial. We have one additional open promise in \(ts.prom\), but this is precisely the fulfilled write in \(\mathcal{E}'\).
3. –
\[
\begin{align*}
  ts.v_C &= ts'.v_C & \text{by definition} \\
  &= \bigcup_{e \in prTst_r(\mathcal{E}')} \psi(e) & \text{unfolding definition of } ts'.v_C \\
  &= \bigcup_{e \in prTst_r(\mathcal{E})} \psi'(e) & \text{pr_tst}(\mathcal{E}') = \text{pr_tst}(\mathcal{E}) \\
  &= \bigcup_{e \in prTst_r(\mathcal{E})} \psi(e) & \text{pr_tst}(\mathcal{E}) \subseteq \text{dom}(\psi)
\end{align*}
\]
\[ ts.co(h(x) = \bigcup_{e \in \text{Ff}(E) \cup \text{prBar}_a(E)} \psi'(e) \quad \text{definition} \]
\[ = \bigcup_{e \in \text{Ff}(E) \cup \text{prBar}_a(E)} \psi(e) \quad \text{Ff}(E) \cup \text{prBar}_a(E) \subseteq \text{dom}(\psi) \]

\[ ts.vwOld = \bigcup_{e \in \text{Ff}(E)} \psi'(e) \quad \text{definition} \]
\[ = \bigcup_{e \in \text{Ff}(E)} \psi(e) \quad \text{Ff}(E) \subseteq \text{dom}(\psi) \]

\[ ts.vwNew = ts'.vwNew \quad \text{by definition} \]
\[ = \bigcup_{e \in \text{prFnc}_a(E') \cap (\text{Ff}(E') \cup \text{prBar}_a(E'))} \psi(e) \quad \text{unfolding definition} \]
\[ = \bigcup_{e \in \text{prFnc}_a(E) \cap (\text{Ff}(E) \cup \text{prBar}_a(E))} \psi(e) \quad \text{since prFnc}_a(E') = prFnc_a(E) \]
\[ = \bigcup_{e \in \text{prFnc}_a(E) \cap (\text{Ff}(E) \cup \text{prBar}_a(E))} \psi(e) \quad \text{logic} \]

\[ ts.v_a = ts'.v_a \quad \text{by definition} \]
\[ = \bigcup_{e \in \text{prBar}_a(E')} \psi(e) \quad \text{unfolding definition} \]
\[ = \bigcup_{e \in \text{prBar}_a(E)} \psi(e) \quad \text{since prBar}_a(E') = prBar_a(E) \]

\[ ts.v_{read} = ts.v_{read} \quad \text{by definition} \]
\[ = \bigcup_{e \in (\text{prFnc}_a(E') \cap \text{Ff}(E')) \cup \text{prBar}_a(E')} \psi(e) \quad \text{unfolding} \]
\[ = \bigcup_{e \in (\text{prFnc}_a(E) \cap \text{Ff}(E) \cup \text{prBar}_a(E))} \psi(e) \quad \text{since prFnc}_a(E') = prFnc_a(E) \quad \text{and prBar}_a(E') = prBar_a(E) \]
\[ = \bigcup_{e \in (\text{prFnc}_a(E) \cap \text{Ff}(E) \cup \text{prBar}_a(E))} \psi(e) \quad \text{logic} \]

4. Trivial since no register value is modified.
Recall:
\[ E \oplus a \mathbb{ff}_r(x, \kappa) \]
\[ = (E \cup \{ e \}, \vdash \{ (e', e) \mid e' \in \text{last}_{t \in \text{Act}^+} \cup \{ \text{func}, \text{bar}(a, x) \}) (E), A, t[e \mapsto \mathbb{ff}_r(x, \kappa)] ) \]

Next, we have \((ts, M) \xrightarrow{\mathbb{ff}_r(x, \kappa)} (ts', M)\) as follows:
- \(ts'.prom = ts.prom \setminus \{ t \}\) by definition
- \(\text{regs}(a) = \kappa \ominus v_a\) because \(ts.v_a = ts'.v_a\) and \(ts.\text{regs}(a) = ts'.\text{regs}(a)\),
- \(M(t) = \{ x := \kappa \}\) by definition
- \(ts.v_{wNew} \cup ts.v_C \cup ts.\text{coh}(x) \cup ts.v_a < ts'.\text{coh}(x)\). We prove maximality of \(ts'.\text{coh}(x)\) for each component:
  - \(ts.v_{wNew} < ts'.\text{coh}(x)\)
    Recall that \(ts.v_{wNew} = \bigcup_{e \in \text{prFunc}(E) \cap \text{prBar}_r(E)} \psi(e)\). By definition of \(E \oplus a \mathbb{ff}_r(x, \kappa)\),
    we have that the event corresponding to \(\mathbb{ff}_r(x, \kappa)\) is causally after each \(e \in \text{prFunc}(E)\).
    Thus, \(ts.v_{wNew} < ts'.\text{coh}(x)\).
  - \(ts.v_C < ts'.\text{coh}(x)\).
    Recall that \(ts.v_C = \bigcup_{e \in \text{prTst}(E)} \psi(e)\). By definition of \(E \oplus a \mathbb{ff}_r(x, \kappa)\), we have that the event corresponding to \(\mathbb{ff}_r(x, \kappa)\) is causally after each \(e \in \text{prTst}(E)\).
    Thus \(ts.v_C < ts'.\text{coh}(x)\).
  - \(ts.\text{coh}(x) < ts'.\text{coh}(x)\) because by definition of \(E \oplus a \mathbb{ff}_r(x, \kappa)\) we have a causal edge from each \(e \in \text{Act}^+\) to the event corresponding to \(\mathbb{ff}_r(x, \kappa)\).
  - \(ts.v_a < ts'.\text{coh}(x)\) same Because
    \[ \bigcup_{e \in \text{prBar}_r(E)} \psi(e) = \bigcup_{e \in \text{prBar}_r(E')} \psi(e) \]

Moreover \(\mathbb{ff}_r(x, \kappa)\) goes at the end and \(\text{prBar}_r(E) = \text{prBar}_r(E')\)

**Proposition 3 (Soundness of rule PR-ReadEx).** Let \((ts', M) \in \mathbb{rstr}^x_r(E \oplus \text{bar}(a, x))\), where \(e\) is the event in rule PR-READEx. Then there exists \((ts, M) \in [E]\) such that
\[ \langle [a := \text{load} x, ts], M \rangle \xrightarrow{r_d(x, \kappa)} \langle \text{skip}, ts', M \rangle. \]

**Proof.** For the proof, we elide the program statements. We write \(E'\) for \(\mathbb{rstr}^x_r(E \oplus \text{bar}(a, x))\). Let \((ts', M) \in [E']\). This means that there exists some mapping \(\psi': \text{FF}(E') \cup \text{Rd}(E') \cup \{ e_{\text{init}} \} \rightarrow \text{dom}(M)\) such that \(\psi'\) can be used to show that \((ts', M)\) satisfies \(E'\), in particular by preservation of memory constraints for the location-restricted red flows for all timestamps \(t'\), events \(f\) such that \(\psi'(e) < t' < \psi'(f)\) and \(e \rightarrow f\) and \(f \in (\text{prFunc}(E') \cap \text{ff}_{r}(E')) \cup \text{prBar}_r(E') \cup \text{ff}_{r}(E')\) we have \(M(t').loc \neq e.loc\).

We now show that there exists some \((ts, M) \in [E]\) such that
\[ \langle ts, M \rangle \xrightarrow{r_d(x, \kappa)} \langle ts', M \rangle \]

Note that \(M\) stays the same as we do not have any further promises. We define \(ts\) to be the following (using \(\psi := \psi'\)):
\[
\begin{align*}
\text{ts.prom} & := ts'.prom \\
\text{ts.v}_{wNew} & := ts'.v_{wNew} \\
\text{ts.coah}(x) & := \bigcup_{e \in \text{ff}_{r}(E') \cup \text{prBar}_r(E')} \psi(e) \\
\text{ts.v}_{\text{read}} & := \bigcup_{e \in (\text{prFunc}(E) \cap \text{ff}_{r}(E)) \cup \text{prBar}_r(E')} \psi(e) \\
\text{ts.v}_{wOld} & := ts'.v_{wOld} \\
\text{ts.v}_b & := ts'.v_b \quad (b \neq a) \\
\text{ts.v}_a & := \bigcup_{e \in \text{prBar}_r(E)} \psi(e) \\
\text{ts.v}_C & := ts'.v_C
\end{align*}
\]
Since $E'$ neither has additional fulfill nor read events nor fences nor tests and $M$ is the same and $prFnc_{+}(E) = prFnc_{+}(E')$, we by definition get $(t_s, M) \in [E]$. Next, we need to determine the timestamp $t$ for reading, which in this case is $\psi(e)$. Since $\psi$ preserves contents and $\ell(e) \in \{rd_r(x, \kappa), \overline{ff}_r(x, \kappa), ini\}$, we get $M(t).loc = x$ and $M(t).val = \kappa$ ($\kappa = 0$ in case of reading initial values). Now consider some timestamp $t'$ such that $t < t' \leq t_s.v_{\text{read}} \sqcup t_s.coh(x)$. Let $f$ be the event such that $\psi(f) = t_s.v_{\text{read}} \sqcup t_s.coh(x)$. This implies (definition of $t_s.coh(x)$ and $t_s.v_{\text{read}}$) that $f \in Ff_{r}^{x}(E) \cup \overline{prBar}_{r}^{x}(E) \cup (prFnc_{+}(E) \cap Ff_{r}(E)) \cup \overline{prBar}_{r}(E)$. Thus we also get $f \in (prFnc_{+}(E') \cap Ff_{r}(E')) \cup \overline{prBar}_{r}(E') \cup Ff_{r}(E')$. By preservation of memory constraints we thus have $M(t').loc \neq e.loc$.

Next to the views. We need to show the correct afterstate for three views being changed by the read rule: $v_a, coh(x)$ and $v_{\text{read}}$.

- $v_a$ is set to $t_s.v_{\text{read}} \sqcup t$. We get

$$t_s.v_{\text{read}} \sqcup t = \bigcup_{e \in \{prFnc_{+}(E) \cap Ff_{r}(E)\} \cup \overline{prBar}_{r}(E)} \psi(e) \sqcup t$$

- $coh(x)$ is set to $coh(x) \sqcup v_{\text{read}} \sqcup t$. We get

$$t_s.co(x) \sqcup t_s.v_{\text{read}} \sqcup t = \bigcup_{e \in \{Ff_{r}(E) \cup \overline{prBar}_{r}^{x}(E)\} \cup (prFnc_{+}(E) \cap Ff_{r}(E)) \cup \overline{prBar}_{r}(E)} \psi(e) \sqcup \bigcup_{e \in \{prFnc_{+}(E') \cap Ff_{r}(E')\} \cup \overline{prBar}_{r}(E')} \psi(e) \sqcup t$$

This holds as the new bar event is flow after all other bars and all fences (by definition of $\oplus$).

- $v_{\text{read}}$ is set to $v_{\text{read}} \sqcup t$. We get

$$t_s.v_{\text{read}} \sqcup t = \bigcup_{e \in \{prFnc_{+}(E) \cap Ff_{r}(E)\} \cup \overline{prBar}_{r}(E)} \psi(e) \sqcup t$$

Finally, we look at the value in register $a$ which is set to $\kappa$ (which is $M(t).val$). $[a]_{E'} = \kappa$ as the reading event $e$ is in $last_{Act^{x}}(E')$ (as this is equal to $last_{Act^{x}}(E)$ and reading from last events is...
required by the hypothesis of rule PR-ReadEx and the event is in \( \mathit{prBar}_a(\mathcal{E}') \) (by definition of \( \oplus \)).

**Proposition 4 (Soundness of rule PR-ReadNew).** Let \( \langle ts', M \rangle \in [(\mathcal{E} \oplus \mathcal{E}') \oplus \mathit{bar}(a, x)] \), \( \mathcal{E}' \) as of rule PR-ReadNew. Then there exists \( \langle ts, M \rangle \in [\mathcal{E}] \) such that \( \langle \langle a := \mathit{load}(x, ts), M \rangle \mathit{r}(x, \kappa)_r, \langle \mathit{skip}, ts', M \rangle \rangle \).

**Proof.** For the proof, we elide the program statements. We write \( \mathcal{E}_{\mathit{after}} \) for \( (\mathcal{E} \oplus \mathcal{E}') \oplus \mathit{bar}(a, x) \). Let \( \langle ts', M \rangle \in [\mathcal{E}_{\mathit{after}}] \). This means that there exists some mapping \( \psi' : \mathit{Ff}(\mathcal{E}_{\mathit{after}}) \cup \mathit{Rd}(\mathcal{E}_{\mathit{after}}) \cup \mathit{e_m} \rightarrow \mathit{dom}(M) \) such that \( \psi' \) can be used to show that \( \langle ts', M \rangle \) satisfies \( \mathcal{E}_{\mathit{after}} \). Let \( e \) be the event in \( \mathcal{E}' \) which we read from and let \( t = \psi'(e) \). We set \( ts \) to be the thread state which we construct from \( \mathcal{E} \) and \( \mathcal{E}' \) using \( \psi \) for which we take \( \psi'|_e \). By definition of \( \mathcal{E}_{\mathit{after}} \), \( e \) is flow-after all fulfills of \( x \), all fences plus reads in \( \mathcal{E} \) which are followed by a bar event (and \( e \) not in \( \mathcal{E} \)). As \( \psi' \) is preserving flows, all events \( f \in (\mathit{prFnc}_r(\mathcal{E}) \cap \mathit{Ff}(\mathcal{E}_{\mathit{after}})) \cup \mathit{prBar}(\mathcal{E}') \) which determine \( ts, v_{\mathit{read}} \) have a timestamp less than \( t \), hence \( ts, v_{\mathit{read}} < t \). Similarly, all events \( f \in \mathit{FF}_r^+(\mathcal{E}) \cup \mathit{prBar}_r^+(\mathcal{E}) \) which determine \( ts, \mathit{coh}(x) \) have a timestamp less than \( t \) and hence \( ts, \mathit{coh}(x) < t \). Hence the condition of the read rule in the operational semantics about locations is trivially fulfilled. Moreover, \( v_{\mathit{post}} \) is \( t \). Thus, \( v_a \) as well as \( v_{\mathit{read}} \) and \( \mathit{coh}(x) \) are set to \( t \), and this then coincides with \( ts', v_a = \bigcup_{f \in \mathit{prBar}_r(\mathcal{E}_{\mathit{after}})} \psi'(f) \) (and \( e \) is the maximal event in this set), with \( ts', v_{\mathit{read}} \) (similarly, because \( e \) is the maximal event in \( (\mathit{prFnc}_r(\mathcal{E}_{\mathit{after}}) \cap \mathit{FF}_r(\mathcal{E}_{\mathit{after}})) \cup \mathit{prBar}(\mathcal{E}_{\mathit{after}}) \) and with \( ts', \mathit{coh}(x) \) (similarly, because \( e \) is the maximal event in \( \mathit{Ff}(\mathcal{E}_{\mathit{after}}) \cup \mathit{prBar}_r(\mathcal{E}_{\mathit{after}}) \), respectively. Views \( v_{\mathit{wOld}}, v_{\mathit{wNew}} \) and \( v_{\mathit{C}} \) keep their values as of the operational semantics, and indeed calculating their views from \( \mathcal{E} \) and \( \mathcal{E}_{\mathit{after}} \) gives the same value as we neither introduce new test events (this would change \( v_{\mathit{C}} \)) nor new fulfills (would change \( v_{\mathit{wOld}} \)) nor fences (would change \( v_{\mathit{wNew}} \)).

Finally, we look at the value in register \( a \) which is set to \( \kappa \) (which is \( M(t).v_{\mathit{val}} \)). \( \mathbb{E}_{\mathit{after}} = \kappa \) as the reading event \( e \) with label \( \mathit{rd}_r(x, \kappa) \) of \( \mathit{last}_{\mathit{Act}}(\mathcal{E}_{\mathit{after}}) \cap \mathit{prBar}_a(\mathcal{E}_{\mathit{after}}) \).

**Proposition 5 (Soundness of rule PR-Fence).** Let \( \langle ts', M \rangle \in [(\mathcal{E} \oplus \mathit{fnC}_r)] \). Then there exists \( \langle ts, M \rangle \in [\mathcal{E}] \) such that \( \langle \langle \mathit{dmb}, ts \rangle, M \rangle \mathit{fnC}_r(\langle \mathit{skip}, ts' \rangle, M) \rangle \).

**Proof.** For the proof, we elide the program statements. We write \( \mathcal{E}' \) for \( (\mathcal{E} \oplus \mathit{fnC}_r) \). Let \( \langle ts', M \rangle \in [\mathcal{E}'] \). This means that there exists some mapping \( \psi' : \mathit{Ff}(\mathcal{E}') \cup \mathit{Rd}(\mathcal{E}') \cup \mathit{e_m} \rightarrow \mathit{dom}(M) \) such that \( \psi' \) can be used to show that \( \langle ts, M \rangle \) satisfies \( \mathcal{E}' \). We now show that there exists some \( \langle ts, M \rangle \in [\mathcal{E}] \) such that

\[
\langle ts, M \rangle \mathit{fnC}_r(\langle ts', M \rangle) \]

Note that \( M \) stays the same as we do not have any further promises. We define \( ts \) to be the following by using \( \psi := \psi' \):

\[
\begin{align*}
ts.prom &:= ts'.prom \\
ts.v_{wOld} &:= ts'.v_{wOld} \\
ts.v_{wNew} &:= \bigcup_{e \in \mathit{prFnc}_r(\mathcal{E}) \cap \mathit{Ff}(\mathcal{E}) \cup \mathit{prBar}_r(\mathcal{E})} \psi'(e) \\
ts.v_{\mathit{read}} &:= \bigcup_{e \in (\mathit{prFnc}_r(\mathcal{E}) \cap \mathit{Ff}(\mathcal{E})) \cup \mathit{prBar}_r(\mathcal{E})} \psi'(e) \\
\end{align*}
\]

Since \( \mathcal{E}' \) neither has additional fulfill, read nor bar events, \( M \) is the same and \( \mathit{prBar}_r(\mathcal{E}) = \mathit{prBar}_r(\mathcal{E}') \) is the same, we by definition get \( \langle ts, M \rangle \in [\mathcal{E}] \). Next to the views. We need to show the
correct afterstate for two views being changed by the fence rule: \( v_{w\text{New}} \) and \( v_{\text{read}} \). Both are set to \( ts.v_{\text{read}} \sqcup ts.v_{\text{wOld}} \).

- For \( v_{\text{read}} \) we get

\[
\begin{align*}
    ts.v_{\text{read}} \sqcup ts.v_{\text{wOld}} &= \bigcup_{e \in (prFnc_\tau(\mathcal{E}) \cap \mathcal{Ff}_\tau(\mathcal{E})) \cup prBar_\tau(\mathcal{E})} \psi(e) \\
    &= \bigcup_{e \in (prFnc_\tau(\mathcal{E}) \cap \mathcal{Ff}_\tau(\mathcal{E})) \cup prBar_\tau(\mathcal{E})} \psi(e) \\
    &= \bigcup_{e \in (prFnc_\tau(\mathcal{E}) \cap \mathcal{Ff}_\tau(\mathcal{E}')) \cup prBar_\tau(\mathcal{E})} \psi(e) \\
    &= \bigcup_{e \in (prFnc_\tau(\mathcal{E}) \cap \mathcal{Ff}_\tau(\mathcal{E}')) \cup prBar_\tau(\mathcal{E})} \psi(e) \\
    &= ts'.v_{\text{read}}
\end{align*}
\]

as \( \oplus \text{fnc}_\tau \) introduces a flow from all last events to the new fence and so \( \mathcal{Ff}_\tau(\mathcal{E}) = prFnc_\tau(\mathcal{E'}) \cap \mathcal{Ff}_\tau(\mathcal{E}') \).

- For \( v_{w\text{New}} \) we get

\[
\begin{align*}
    ts.v_{\text{read}} \sqcup ts.v_{\text{wOld}} &= \bigcup_{e \in (prFnc_\tau(\mathcal{E}) \cap \mathcal{Ff}_\tau(\mathcal{E}')) \cup prBar_\tau(\mathcal{E}) \cup \mathcal{Ff}_\tau(\mathcal{E})} \psi(e) \\
    &= \bigcup_{e \in prBar_\tau(\mathcal{E}) \cup \mathcal{Ff}_\tau(\mathcal{E})} \psi(e) \\
    &= \bigcup_{e \in prBar_\tau(\mathcal{E}') \cup \mathcal{Ff}_\tau(\mathcal{E}') \cup \mathcal{Ff}_\tau(\mathcal{E}') \cup \mathcal{Ff}_\tau(\mathcal{E}')} \psi(e) \\
    &= \bigcup_{e \in prFnc_\tau(\mathcal{E}) \cap (prBar_\tau(\mathcal{E}') \cup \mathcal{Ff}_\tau(\mathcal{E}'))} \psi(e) \\
    &= ts'.v_{w\text{New}}
\end{align*}
\]

as \( prBar_\tau(\mathcal{E}') \) and \( \mathcal{Ff}_\tau(\mathcal{E}') \) are subsets of \( prFnc_\tau(\mathcal{E}') \).

**Proposition 6 (Soundness of rule PR-Register).** Suppose that \( \langle ts', M \rangle \in \llbracket \mathcal{E} \oplus \text{bar}(a, \exp) \rrbracket \).

Then there exists \( \langle ts, M \rangle \in \llbracket \mathcal{E} \rrbracket \) such that

\[
\langle a := \exp, ts \rangle, M \xrightarrow{\text{lst}(a, \exp)} \langle \text{skip}, ts' \rangle, M \rangle.
\]

**Proof.** For the proof, we elide the program statements. We write \( \mathcal{E}' \) for \( \mathcal{E} \oplus \text{bar}(a, \exp) \). Let \( \langle ts', M \rangle \in \llbracket \mathcal{E}' \rrbracket \). This means that there exists some mapping \( \psi' : \mathcal{Ff}(\mathcal{E}') \cup \text{Rd}(\mathcal{E}') \cup \text{Ini} \rightarrow \text{dom}(M) \) such that \( \psi' \) can be used to show that \( \langle ts, M \rangle \) satisfies \( \mathcal{E}' \). We now show that there exists some \( \langle ts, M \rangle \in \llbracket \mathcal{E} \rrbracket \) such that

\[
\langle ts, M \rangle \xrightarrow{\text{lst}(a, \exp)} \langle ts', M \rangle
\]
Note that $M$ stays the same as we do not have any further promises. We define $ts$ to be the following by using $\psi := \psi'$:

$$
\begin{align*}
ts.prom & := ts'.prom \\
\underline{\text{ts.coh}}(x) & := ts'.\underline{\text{co}}h(x) \\
ts.v_{\text{wOld}} & := ts'.v_{\text{wOld}} \\
ts.v_{\text{wNew}} & := ts'.v_{\text{wNew}} \\
ts.v_a & := \bigcup_{e \in \text{prBar}_a(E)} \psi(e) \\
ts.v_{\text{read}} & := ts'.v_{\text{read}}
\end{align*}
$$

We need to show the correct afterstate for $v_a$ and $\text{reg}(a)$ being changed by the register rule.

- For $v_a$ we get

$$
\begin{align*}
\text{ts}_a \cup \bigcup_{a' \in R(exp)} \text{ts}_{a'} & = \bigcup_{e \in \text{prBar}_a(E)} \psi(e) \cup \bigcup_{a' \in R(exp)} \bigcup_{e \in \text{prBar}_a'(E)} \psi(e) \\
& = \bigcup_{e \in \text{prBar}_a(E) \cup \bigcup_{a' \in R(exp)} \text{prBar}_a'(E)} \psi(e) \\
& = \bigcup_{e \in \text{prBar}_a'(E)} \psi'(e)
\end{align*}
$$

as $\bigoplus \text{bar}(a, exp)$ introduces for every $b \in R(exp) \cup \{a\}$ a flow from the last $\text{bar}(b, \cdot)$ to $\text{bar}(a, exp)$ and so $\text{prBar}_a(E') = \text{prBar}_a(E) \cup \bigcup_{a' \in R(exp)} \text{prBar}_a'(E)$.

- For $\text{reg}(a)$ we need to show $[a]_{E'} = \kappa$ with $[a]_{\text{reg}} = \kappa \oplus v$. Let $exp = e_1 op_1 \ldots op_{k-1} e_k$ with $[e_i]_{\text{reg}} = \kappa_i \oplus v_i$. We get

$$
\begin{align*}
[a]_{\text{reg}} & = [exp]_{\text{reg}} \\
& = [e_1 op_1 \ldots op_{k-1} e_k]_{\text{reg}} \\
& = (\kappa_1 [op_1] \ldots [op_{k-1}] \kappa_k) \oplus (v_1 \downarrow \ldots \downarrow v_k)
\end{align*}
$$

and

$$
\begin{align*}
[a]_{E'} & = [exp]_{E'} \\
& = [e_1 op_1 \ldots op_{k-1} e_k]_{E'} \\
& = [e_1]_{E'} [op_1] \ldots [op_{k-1}] [e_1]_{E'} \\
& = \kappa_1 [op_1] \ldots [op_{k-1}] \kappa_k
\end{align*}
$$

as $[e_i]_{E'} = [e_i]_E$ and $[e_i]_E = [e_i]_{\text{reg}}$ for all $1 \leq i \leq k$.

**Proposition 7 (Soundness of rule PR-Assume).** Suppose that $(ts', M) \in [E \oplus \text{st}_r(bexp)]$ such that $[bexp]_E = \text{true}$. Then there exists $(ts, M) \in [E]$ such that $(\langle \text{assume} bexp, ts \rangle, M) \xrightarrow{\text{asm}(bexp)}_r (\langle \text{skip}, ts' \rangle, M)$. 

Proposition 8. Let \( E_1, \ldots, E_n \) be local assertions of threads \( \tau_1, \ldots, \tau_n \), respectively. If \( E_1, \ldots, E_n \) are synchronizable and \( C \) is an interference free configuration of \( Conf(E_1||\ldots||E_n) \), then there exists some memory \( M \) and thread states \( ts_1, \ldots, ts_n \) (of threads \( \tau_1, \ldots, \tau_n \), respectively) such that \( \langle ts_i, M \rangle \in [E_C] \), \( i = 1, \ldots, n \).

Proof. Let \( C \) be an interference free configuration of \( E_1||\ldots||E_n \). Let \( \prec \) be the total memory consistent linearizable order. We assume it to be \( e_i \prec e_1 \prec \ldots \prec e_m \) for \( E_C = \{ e_i, e_1, \ldots, e_m \} \). We now construct the following memory \( M \):

\[
M(0) = ini
\]
\[
M(i) = \langle x, \kappa \rangle_\tau \quad \text{if } \ell(e_i) \in \{ ff_\tau(x, \kappa), rd_\tau(x, \kappa) \}
\]

With this, we let \( \psi \) map \( e_i \) to 0 and \( e_i \) to \( i \). By construction and \( C \) being interference free, \( \psi \) is (i) initializing at zero, (ii) consecutive, (iii) content preserving, (iv) flow preserving, and (v) memory-constraints preserving. The thread states \( ts_1, \ldots, ts_n \) can then simply be calculated from \( M, \psi \) and \( C \).

The second proposition is a sort of compositionality result of parallel composition with respect to the global state (views and memory) of a program and its projection onto the threads and their individual proof outlines.
We have to show that this $\psi$ is well defined and total.

- it is total, because $C$ is thread-covering,
- well defined:
  assume exists another, different event $(e'_1, \ldots, e_i, \ldots, e'_n) \in E_C$. Since $e_i \neq *$, we have (by the definition of parallel composition) that
  $$(e_1, \ldots, e_i, \ldots, e_n) \# (e'_1, \ldots, e_i, \ldots, e'_n)$$
  and then by definition of configuration we cannot have both events in $C$.

We further need to show that $\psi_i$ satisfies the conditions (i) to (v) of $\llbracket E_i \rrbracket$.

1. it initializes at zero: by construction,
2. consecutive for $\tau_i$: follows because $\psi$ is consecutive,
3. content preserving: by construction,
4. flow order preserving: let $e_i \rightarrow^+ d_i$. Because $C$ is thread-covering, there exist events $(e_1, \ldots, e_i, \ldots, e_n)$, $(d_1, \ldots, d_i, \ldots, d_n) \in E_C$ such that
   $$(e_1, \ldots, e_i, \ldots, e_n) \rightarrow^+ (d_1, \ldots, d_i, \ldots, d_n).$$
   Hence $\psi(e_1, \ldots, e_i, \ldots, e_n) < \psi(d_1, \ldots, d_i, \ldots, d_n)$ which implies $\psi_i(e_i) < \psi_i(d_i)$ by construction.
5. memory constraints preserving: let $d_i \xrightarrow{L, i} e_i$, then there exists
   $$(d_1, \ldots, d_i, \ldots, d_n) \xrightarrow{L'} (e_1, \ldots, e_i, \ldots, e_n)$$
   for some $L' \supset L$. Hence for all $t$ such that $\psi(d_1, \ldots, d_i, \ldots, d_n) < t < \psi(e_1, \ldots, e_i, \ldots, e_n)$ we get $M(t).loc \neq (d_1, \ldots, d_i, \ldots, d_n).loc$. Hence we also have $\forall t: \psi_i(d_i) < t < \psi_i(e_i): M(t).loc \neq d_i.loc$.

We further need to show that the promises are ok:
We have $ts_i.prom = M_{\tau_i} \setminus \psi(\mathbb{F}_\tau_i(E_C))$. By the definition of labelling in parallel composition and the fact that all $E_{\overline{j}}$, $j \neq i$, contain no events labelled $\mathbb{F}_\tau_i(\cdot, \cdot)$, we get that for all $(e_1, \ldots, e_i, \ldots, e_n) \in E_C$ s.t. $l(e_1, \ldots, e_i, \ldots, e_n) = \mathbb{F}_\tau_i(\cdot, \cdot)$ we have $t_i(e_i) = \mathbb{F}_\tau_i(\cdot, \cdot)$. Furthermore $\psi(e_1, \ldots, e_i, \ldots, e_n) = \psi_i(e_i)$. Hence $M_{\tau_i} \setminus \psi(\mathbb{F}_\tau_i(E_C)) = M_{\tau_i} \setminus \psi_i(\mathbb{F}_\tau_i(E_i))$.

Next to the views:
First $v_C$. We have
$$ts_i.v_C = \bigcup_{(e_1, \ldots, e_n) \in prTst_{\tau_i}(E_C)} \psi(e_1, \ldots, e_n).$$

This holds because the proof rules only introduce fulfill events of a thread $\tau$ in the proof outline of $\tau$.
Recall that \( prTst_{\tau_i}(E_C) = \{(e_1, \ldots, e_n) \in \mathbb{Rd}(E_C) \cup \mathbb{FF}(E_C) \cup \{e_m\} \mid \exists (e'_1, \ldots, e'_n) \in last_{\tau_i}(E_C) : (e_1, \ldots, e_n) \rightarrow^+(e'_1, \ldots, e'_n)\}. \) First, note that \( e'_j = *, j \neq i \) (since this is a test-labelled event which is not synchronized). Second, note that we do not necessarily have \( e_i \rightarrow_i^+ e'_i \). However, there exists a sequence \((e_i^1, \ldots, e_i^1), \ldots, (e_i^n, \ldots, e_i^n)\) such that

\[
(e_1, \ldots, e_n) = (e_1^1, \ldots, e_n^1) \rightarrow \cdots \rightarrow (e_1^n, \ldots, e_n^n) = (e_1^1, \ldots, e_n^1)
\]

with some \( j, 1 \leq j \leq m-1 \) s.t. \( (e_i^1, \ldots, e_i^j) \in \text{sync}(E_1, \ldots, E_n) \) and

\[
e_i^j \rightarrow_{i-1} \cdots \rightarrow_{1} e_i^j.
\]

We furthermore have \( \psi(e_i^1, \ldots, e_i^j) = \psi_i(e_i^j) \) and \( \psi(e_1, \ldots, e_n) < \psi(e_i^1, \ldots, e_i^j) \) (by \( \psi \) being flow preserving). Thus \( \bigcup_{e \in prTst_{\tau_i}(E_C)} \psi_i(e) = \bigcup_{e \in prTst_{\tau_i}(E)} \psi_i(e) \).

The other views have a similar reasoning, because \( prBar \) and \( prFnc \) are also sets of events prior to an event of the form \((*, \ldots, *, *, *, *)\). Moreover, if \( (e_1, \ldots, e_n) \in \mathbb{FF}(E_C) \), then \( e_i \in \mathbb{FF}(E_i) \).

Finally, the register values. We need to show that \([a]_{E_C} = [a]_{E_i} \) for all registers \( a \in \mathbb{R}(\tau_i) \). Recall that \([a]_{E_C}\) is calculated via \( prBar_{\tau_i} \). As \( a \) is local to \( \tau_i \), an ordering with respect to a \( \text{bar}(a, \_)-\)labelled event has to come from the event structure \( E_i \). Hence, the same value is calculated in \( E_C \) and \( E_i \).

These two propositions next help us to establish our first main result which is the soundness of the rule of parallel composition.

**Theorem 1.** Let \([\text{lni}] S_i [E_i], i \in \{1, \ldots, n\}, \) be proof outlines of threads \( \tau_1 \) to \( \tau_n \) such that \( E_1 \) to \( E_n \) are synchronizable and let \( T_0 \) be an initial thread pool with \( T_0(\tau_i) = (S_i, \text{ts} s_0) \) and \( M_0 = M_{\text{ini}} \).

Then, for every thread pool \( T \) with \( T(\tau_i) = (\text{skip}, \text{ts} s_i) \), interference free configuration \( C \in \text{Conf}(E_1 | \cdots | E_n) \) and memory \( M \) such that \( \langle ts_i, M \rangle \in [E_C], \text{tids}(M) = \{\tau_1, \ldots, \tau_n\} \) and \( ts_i.\text{prom} = \emptyset, i \in \{1, \ldots, n\} \), we have \( \langle T_0, M_0 \rangle \rightarrow^* \langle T, M \rangle \).

**Proof.** Note first that by Proposition 8 such states \( \langle ts_i, M \rangle \) always exist for interference free configurations.

For the proof, we basically show the existence of an execution of this form:

\[
\langle T_0, M_0 \rangle \xrightarrow{\text{all promises}} \langle T_0^p, M \rangle \xrightarrow{\text{the statements}} \langle T, M \rangle
\]

First note, that Lemma 1 tells us that such a trace which first executes all promises and then the remaining operations exists. We first show \( \langle T_0, M_0 \rangle \rightarrow^* \langle T_0^p, M \rangle \). For this, we let \( T_0^p \) be the thread pool coinciding with \( T_0 \) except for the promise sets which are \( ts_i.\text{prom} = M_{\tau_i}, i \in \{1, \ldots, n\} \). Assume \#M = m, \( M(j) = (x := \kappa_j) \) and let \( op_j = \text{prm}_\tau(x, \kappa), 1 \leq j \leq m \). Then

\[
\langle T_0, M_0 \rangle \xrightarrow{\text{op}_1 \cdots \text{op}_m} \langle T_0^p, M \rangle
\]

(promises are made in the order as given by \( M \)) and all these steps are certified as the promises will finally be empty.

Next, we consider the steps of statements. For this, first let \( ts_i^p \) be \( T_0^p(\tau_i).\text{tstate} \) be the state of thread \( \tau_i \) after having made the promises. Let \( op_1^i \ldots op_h^i, i \in \{1, \ldots, n\}, \) and \( op_j^i \neq \text{rd}_{\tau_i}(\_ \_ \_), \) be the sequence of non-promise operations such that \( S_i \xrightarrow{op_1^i \cdots op_h^i} \text{skip}, i \in \{1, \ldots, n\} \). From
Hence also \( \langle ts_i, M \rangle \rightarrow^* \langle T, M \rangle \).

### B.3 Completeness

Finally, we prove completeness of our proof calculus. We must prove that for every trace of a program we can construct local event structures, whereby the parallel composition of the local event structures contains an interference-free configuration. We construct these local event structures by induction following the execution trace.

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

Then there are local proof outlines \([\text{ ini}]; S_k [\mathcal{E}_k]\) of threads \( \tau_k \), \( k \in \{1, \ldots, n\} \), such that \( \mathcal{E}_1 \) to \( \mathcal{E}_n \) are synchronous and there exists an interference free configuration \( C \in \text{Conf}(\mathcal{E}_1||\ldots||\mathcal{E}_n) \) with \( \langle T, M \rangle \in [\mathcal{E}_C] \).

For the proof, we need a number of well-formedness conditions on the event structures our proof rules construct. First of all, we slightly extend the action labels of events: instead of \( \text{rd}_r(x, \kappa) \) we use \( \text{rd}_r^d(x, \kappa) \) to denote the timestamp \( t \) of the read value in memory, and we similarly use \( \text{ff}_r^d(x, \kappa) \). In a trace \( tr = (T_0, M_0) \stackrel{op_1}{\rightarrow} \ldots \stackrel{op_k}{\rightarrow} \langle T, M \rangle \) we define \( T_{rd}(tr, \tau) \) to be the set of timestamps that thread \( \tau \) reads from, similarly \( T_{ff}(tr, \tau) \) for the timestamps of the fulfills. With these extensions at hand, we say that an event structure \( \mathcal{E} = \langle E, \rightarrow, \#; \Lambda, \ell \rangle \) of thread \( \tau \) is timestamp closed if the following holds: \( \forall t, t' \in T_{rd}(tr, \tau), e \in E, \text{ if } \ell(e) = \text{rd}_r^d(\cdot, \cdot) \text{ and } t' < t, t' \neq 0, \text{ then there exists an } e' \in E \text{ with } \ell(e') \in \{\text{rd}_r^d(\cdot, \cdot), \text{ff}_r^d(\cdot, \cdot)\}; \mathcal{E} \text{ is timestamp ordered if } \forall t, t' \in T_{rd}(tr, \tau) \cup T_{ff}(tr, \tau) \text{ we have: if } t' < t, \text{ then } t'-\text{labelled events cannot be flow-after } t \text{ labelled events}; \mathcal{E} \text{ contains a modification order for all locations } x \text{ if for all } x \in \text{Loc}, \rightarrow^+|_{\text{Act}^x \times \text{Act}^x} \text{ is a total order. Finally, we need some conditions on the read events occurring in the event structures. Recall that the rule PR-READNEW may introduce a number of read events (ordered in a chain) of which only the last in the chain is used for reading at that time. This reading is marked by attaching a bar event flow after the read. We define a read event } e \text{ in } \mathcal{E} \text{ to be unbarred, } \text{ubr}_E(e), \text{ if } \ell(e) = \text{rd}_r(x, \cdot) \wedge \neg \exists e' \text{ s.t. } \ell(e') = \text{bar}_r(x, \cdot) \wedge e' \rightarrow e' e'. \text{ We say that all read chains end in a bar in } \mathcal{E} \text{ if the following holds: } \forall e \in E : \text{ubr}_E(e) \Rightarrow \exists m, \exists e_1, \ldots, e_m \in \text{Rd}(\mathcal{E}), e_1 = e, e_i \rightarrow e_{i+1} \wedge \neg \text{ubr}(e_m). \)
Proof. Assume \( tr : (T_0, M_0) \xrightarrow{op_1} (T_1, M_1) \ldots \xrightarrow{op_m} (T_m, M_m) = (T, M) \) is the trace such that all promises are done at the beginning, and let \( op_m \) be the last such promise operation leading to \( (T_j, M_j) \). Note first that \( M_j = M_{j+1} = \ldots = M_m = M \) (only promises change the memory). We now inductively construct the event structures \( E^i_k \) of every thread \( k, 1 \leq k \leq n \) (and by this the proof outlines) for all indizes \( i = j \) to \( m \). In every step, the event structures \( E^i_k \) are (i) timestamp closed, (ii) timestamp ordered, (iii) contain a modification order, (iv) all read chains end in a bar and moreover (v) \( (T_i(\tau_k), M) \in [E^i_k] \) using the mapping \( \psi : \text{Ff}(E^i_k) \cup \text{Rd}(E^i_k) \cup \{ e_{ini} \} \to \text{dom}(M) \) to be

\[
e_{ini} \mapsto 0
\]

\[
\text{rd}^i(\cdot, \cdot) \mapsto t
\]

\[
\text{ff}^i(\cdot, \cdot) \mapsto t
\]

**Induction Start** \( i = j : E^i_k = \text{Ini} \). As the event structure \( \text{Ini} \) contains just a single event, it is timestamp closed, timestamp ordered and contains a modification order. Moreover, \( (T_j(\tau_1), M) \in [\text{Ini}] \) because the event \( e_{ini} \) gets mapped to 0 by \( \psi \), and hence all views are 0, and the promise set contains all entries in \( M \) of \( \tau_1 \).

**Induction Step** Assume \( E^i_k \) with properties (i) to (iv) has been constructed. We now have different cases depending on the next operation \( op_{i+1} \). Let \( T_i(\tau_k) = (S^i, ts^i) \).

1. \( op_{i+1} \) is not an operation of \( \tau_k \):
   
   Then \( E^{i+1}_k = E^i_k \). As the event structure is not changing and the state of thread \( \tau_k \) is not changing, all properties are trivially preserved.

2. \( op_{i+1} \) is a dmb
   
   Then \( E^{i+1}_k = E^i_k \oplus \text{fnr}_{\tau_k} \). As the new event is neither a read nor a fulfill (which would have timestamps), event structure \( E^{i+1}_k \) is still timestamp closed, timestamp ordered, contains a modification order and all read chains end in bar. As for the views: there are two views affected by the fence operation, \( v_{\text{read}} \) and \( v_{\text{wNew}} \). We only consider these here. First, \( v_{\text{read}} \).
   
   By the operational semantics, we know that \( ts^{i+1}, v_{\text{read}} = ts^i, v_{\text{read}} \sqcup ts^i, v_{\text{wOld}} \). We need to show that this fits to the semantics of \( E^{i+1}_k \) given the semantics of \( E^i_k \) and operation \( \oplus \).
   
   \[
   ts^i, v_{\text{read}} \sqcup ts^i, v_{\text{wOld}} = \bigcup_{e \in \text{prFnc}_{\tau_k}(E^i_k) \cup \text{prFf}_{\tau_k}(E^i_k)} \psi(e) \sqcup \bigcup_{e \in \text{prBar}_{\tau_k}(E^i_k)} \psi(e)
   \]
   
   \[
   = \bigcup_{e \in \text{Act}_{\tau_k}(E^i_k)} \psi(e)
   \]
   
   \[
   = \bigcup_{e \in \text{Act}_{\tau_k}(E^i_k) \cup \text{prBar}_{\tau_k}(E^i_k)} \psi(e)
   \]
   
   \[
   = \bigcup_{e \in \text{prFnc}_{\tau_k}(E^{i+1}_k) \cup \text{prFf}_{\tau_k}(E^{i+1}_k)} \psi(e)
   \]
   
   The last term corresponds to \( ts^{i+1}, v_{\text{read}} \) when calculated from \( E^{i+1}_k \). For \( v_{\text{wNew}} \) we get almost the same reasoning as the new fence event is placed flow after all but test events.

3. \( op_{i+1} = \text{ff}_{\tau_k}(x, \kappa) \) is an operation of thread \( \tau_k \):
   
   By the operational semantics, this transition results from a statement \( \text{store} x a \) for some register \( a \) such that \( ts^i.reg(r) = \kappa @ v_{\text{a}} \). Let \( t \) be the timestamp of the promise in \( M \) being fulfilled, i.e. \( t \in ts^i.prom \). Since \( (T_i(\tau_k), M_i) \in [E^i_k] \) (by induction hypothesis), we
get \(v_{wOld} = \kappa\). Hence proof rule WRITE is applicable and we let \(E_k^{i+1} = E_k^+ \uplus \text{ff}_{tst_k} (x, \kappa)\). We show properties (i) to (v). By induction hypothesis, \(E_k^{i+1}\) is timestamp closed as the additional event is not a read event. It is timestamp ordered as the new event is added after all events \(e'\) with \(\ell(e') \in \text{Act}^{	ext{op}}\), and we know by the operational semantics that \(ts^i.\text{coh}(x) < t\), and by \([E_k^i]\) that \(ts^i.\text{coh}(x)\) is the maximum of all fulfills and reads to \(x\). It moreover maintains the existence of a modification order for \(x\) due to the same reasons. All read chains end in bar because no new reads are added.

Next to the state after the fulfill: the operational semantics changes the view \(\text{coh}(x)\) and \(v_{wOld}\) as well as \(\text{prom}\) and keeps the rest. We need to show that this after state is in \([E_k^{i+1}]\).

First, \(\text{coh}(x) = t\) (by operational semantics) and this would be derived from \(E_k^{i+1}\) as the new event \(\text{ff}_{tst_k} (x, \kappa)\) is maximal on all \(x\)-labelled events. Also, since this event is now in the event structure, \(t\) is not in \(\text{prom}\) anymore. Finally, \(v_{wOld}\) is derived from the event structure as the maximum of all timestamps of fulfills. This includes \(t\) and hence \(v_{wOld} := v_{wOld} \uplus t\) holds. For the remaining views: \(v_c\) is unchanged because no new test events are added; \(v_{wNew}\) is unchanged because no fence is added; all \(v_i\)’s are kept as no bar events are added and \(v_{\text{read}}\) is unchanged as neither fences nor bars are added.

4. \(op_{t+1} = \text{lst}(a, \text{exp})\) is an operation of thread \(\tau_k\):

Then \(E_k^{i+1} = E_k^+ \uplus \text{bar}(a, \text{exp})\). As the new event is neither a read nor a fulfill, \(E_k^{i+1}\) is still timestamp closed, timestamp ordered, contains a modification order and all read chains end in bar. Further \([\text{exp}]_{\mathcal{E}} = [\text{exp}]_{\{\text{ts}^{i+1}, \text{regs}\} \uplus v}\) for some view \(v\) and \(ts^{i+1}.\text{vars} = \bigcup_{e \in \text{prBar} \cap E_k^{i+1}} \psi(e)\). Both equations follow directly from the construction in the proof of Proposition 6.

5. \(op_{t+1} = \text{asm}(\beta)\) is an operation of thread \(\tau_k\):

Since \((T, (\tau_k), M_i) \in \[E_k^i\] and \([\beta]_{\text{ts}^i, \text{regs}} = \text{true} \uplus \text{bar} \uplus \text{prom} \uplus \text{bar} \uplus \text{prom} \uplus \text{bar}\) for some view \(v\), we get \(\[\beta]\)_{\mathcal{E}} = \text{true}. Hence proof rule \text{Assume} is applicable and we let \(E_k^{i+1} = E_k^+ \uplus \text{tst}_{tst_k}(\beta)\). As the new event is neither a read nor a fulfill, \(E_k^{i+1}\) is still timestamp closed, timestamp ordered, contains a modification order and all read chains end in bar. Now we only need to show that \(ts^{i+1}.\text{vars} = \bigcup_{e \in \text{prTst}_{tst_k} (E_k^{i+1})} \psi(e)\) which follows directly from the construction in the proof of Proposition 7.

6. \(op_{t+1} = \text{rd}(x, \kappa)\) is a transition of thread \(\tau_k\) executing instruction \(a := \text{load} x\) thereby reading from timestamp \(t\):

Let \(T = \{t_1, \ldots, t_a\}\) be the set of timestamps that thread \(\tau_k\) will read in \(\text{tr}\) after and including \(op_{t+1}\) for which there is furthermore no timestamp in \(T_{\text{ff}}(tr, \tau_k)\). Let \(T_{\leq t} = \{t' \in T \mid t' \leq t\}\) and let \(t^1 < t^2 < \ldots < t^p\) be an ordering of that set. Note that \(t^p = t\).

By timestamp closedness, there is some index \(t\), \(0 \leq t \leq p\), such that all timestamps \(t' \in T_{\text{rd}}(tr, \tau_k)\) with \(t' \leq t\) have read or fulfill events in \(E_k^i\). Two cases need to be considered: If \(t = p\), i.e. read or fulfill events for all \(T_{\leq t}\) are in \(E_k^i\), we apply rule \text{PR-READ}. We let \(e' = \text{last}_{\text{Act}^e}(E_k^i)\). By existence of a modification order, \(e'\) is uniquely defined. We need to show that the timestamp of \(e'\) is \(t\). Assume the contrary, i.e. \(e'\) has a timestamp \(t' \neq t\). Let \(e\) be the event in \(E_k^i\) with timestamp \(t\). \(e\) is also an event on location \(x\), i.e. by modification order and \(e' \in \text{last}_{\text{Act}^e}(E_k^i)\), we have to have \(e \rightarrow e'\). By timestamp ordering, we thus have \(t < t'\).

By the operational semantics, we get \(t' > ts^i.\text{v}_{\text{read}} \uplus ts^i.\text{coh}(x)\). As \((ts^i, M) \in \[E_k^i\]\), \(e'\) has to be an unbarred read (because in case of a fulfill, \(ts^i.\text{coh}(x) = t'\); in case of a barred read \(ts^i.\text{v}_{\text{read}} \geq t'\)). However, in \(E_k^i\) all read chains end in bar, hence \(ts^i.\text{v}_{\text{read}}\) is greater or equal the timestamp of the last event in the chain, and hence greater or equal \(t'\).

Contradiction. Hence the timestamp of \(\text{last}_{\text{Act}^e}(E_k^i)\) is \(t\).
We now let \( e = \text{last}_{\mathit{Act}}(E_k^i) \) to match the notation of the rule. As \( \langle ts^i, M \rangle \in [E_k^i] \) using the specific \( \psi \) and \( M(t) \) being \( (x := k)^t \), \( e \) is labelled \( \mathit{rd}^u_k(x, \kappa) \), \( \mathit{ff}^u_k(x, \kappa) \) or \( \mathit{ini} \) (with \( \kappa = 0 \)). We then let \( E_k^{i+1} = \mathit{rest}^c(E_k^i \oplus \mathit{bar}(a, x)) \). We need to show this new event structure to satisfy all properties. First, as we neither add new read nor new fulfill events, \( E_k^{i+1} \) is still timestamp closed, timestamp ordered and contains a modification order. As we only add additional bar events, all read chains continue to end in a bar. We need to show \( \langle ts^{i+1}, M \rangle \in [E_k^{i+1}] \). The reasoning for promises and views follows that used in the proof of Proposition 3. Finally, we need to take a look at the new location restriction in \( E_k^{i+1} \). By the operational semantics, we get

\[
\forall t', t < t' \leq ts^i \text{ : } v_{\mathit{read}} \cup ts^i.\mathit{coh}(x) \Rightarrow M(t') \neq x
\]

By \( \langle ts^i, M \rangle \in [E_k^i] \), we thus get

\[
\forall t', t < t' \leq \bigcup_{f \in (\mathit{prFnc}(E_k^i) \cup \mathit{ff}_{ts}(E_k^i) \cup \mathit{prBar}_{ts}(E_k^i) \cup \mathit{ff}_{ts}(E_k^i))} \psi(f) \Rightarrow M(t') \neq x
\]

Hence \( M \) satisfies the additional memory constraint placed by \( \mathit{rest}^c(E_k^i \oplus \mathit{bar}(a, x)) \).

Second case. If \( \ell < p \), we apply rule \( \mathit{PR-ReadNew} \) and let \( E' \) be the event structure with \( E' = \{ e^{i+1}, \ldots, e^p \} \). \( e^{i+1} \rightarrow \ldots \rightarrow e^p \). \( \ell'(E^m) = \mathit{rd}^u_k(x, \kappa) \).

By construction, \( E' \) is sequential, \( \ell'(E') \subseteq \mathit{Act}^{\mathit{rd}^u_k} \). \( \text{last}_{\mathit{Act}}(E') = e \) is an event labelled \( \mathit{rd}^u_k(x, \kappa) \) and hence rule \( \mathit{PR-ReadNew} \) is applicable. We now let \( E_k^{i+1} = (E_k^i \oplus E') \oplus \mathit{bar}(a, x) \).

Next to the properties. In \( E_k^{i+1} \) all read chains end in a bar (as the additional read chain ends in \( e \) followed by \( \mathit{bar}(a, x) \); \( E_k^{i+1} \) contains a modification order (as \( E' \) is sequential and hence contains a modification order, \( E_k^i \) contains a modification order and \( E_k^i \oplus E' \) orders all events on same locations). \( E_k^{i+1} \) is timestamp ordered: \( E' \) is timestamp ordered by construction, \( E_k^i \) is timestamp ordered by induction, now assume that here is some event \( f \in E_k^i \) and \( g \in E' \) with timestamps \( t_f \) and \( t_g \) such that \( f \rightarrow g \) but \( t_g > t_f \). By \( \oplus \) on event structures, there is some location \( y \) such that \( f \) is a fulfill on \( y \) and \( g \) a read of \( y \). As \( \langle ts^i, M \rangle \in [E_k^i] \), \( ts^i.\mathit{coh}(y) \geq t_f \).

However, then the trace \( tr \) cannot later have a read of \( y \) from a timestamp earlier than \( t_f \) (by view monotonicity and operational semantics). Hence \( E_k^{i+1} \) is timestamp ordered. Timestamp closedness follows by construction of \( E' \) which assumes there are already events in \( E_k^i \) for all timestamps \( t' \leq t' \).

It remains to be shown that \( \langle ts^{i+1}, M \rangle \in [E_k^{i+1}] \). Note first that \( t \) is larger than all timestamps of all events in \( E_k^i \). As \( \langle ts^i, M \rangle \in [E_k^i] \), we thus have

\[
ts^i.\mathit{v}_{\mathit{read}} < t \land ts^i.\mathit{v}_u < t \land \forall y \in \mathit{Loc} : ts^i.\mathit{coh}(y) < t
\]

Hence \( ts^{i+1}.\mathit{v}_{\mathit{read}} = ts^{i+1}.\mathit{v}_u = ts^{i+1}.\mathit{coh}(x) = t \). This is consistent with \( E_k^{i+1} \) because \( E_k^{i+1} \) has a bar event directly flow after the maximal read event \( e^p \) in \( E' \) which has timestamp \( t \).

Views \( v_C, v_{\mathit{wOld}}, v_{\mathit{wNew}}, \mathit{coh}(y) \) for \( y \neq x \) do not change as neither new test nor fulfill nor fence nor \( \mathit{bar}(\cdot, y) \) events are added. Finally, \( ts^{i+1}.\mathit{prom} = ts^i.\mathit{prom} \) and this still matches \( E_k^{i+1} \) as no fulfill events are added.

We next let \( E_k = E_k^m, 1 \leq k \leq n \). We need to show that \( E_1 \) to \( E_n \) are synchronisable and that there exists some interference free configuration \( C \in (E_1| \ldots | E_n) \).

\( E_1 \) to \( E_n \) are synchronisable because the read events in the event structure correspond to the read operations occurring in the execution, they read from memory and every entry in memory has a fulfill operation in the execution (as all promise sets of threads are empty at the end) and
hence a fulfill event in the event structures.
We construct the configuration $C$, i.e. a set of events. Note that we here mix events and their labellings.

1. $(e_{i_1}, \ldots, e_{i_n}) \in E_C$.
2. Let $t \in \text{dom}(M) \setminus \{0\}$. For every $t$ we construct a single event $e_t = (e_1, \ldots, e_n)$ as follows:
   (i) $e_t = *$ if $t \notin T_{rd}(\tau_k) \cup T_{ff}(\tau_k)$.
   (ii) $e_t = \text{ff}_x^x(x, \kappa)$ if $t \in T_{ff}(\tau_k)$.
   (iii) $e_t = \text{rd}_x^x(x, \kappa)$ if $t \in T_{rd}(\tau_k)$ and $t \in T_{ff}(\tau_i)$ else.
3. for every $e_t \in \mathcal{E}_k$ with $\ell_k(e_t) \in \{\text{fn}_{r_k}, \text{st}_{r_k}, \text{bar}(a, \cdot) \mid a \in R(\tau_k)\}$, we add one event
   $(\ast, \ldots, e_t, \ldots, \ast)$ to $C$.

We first show that $C \in \text{Conf}(\mathcal{E}_1||\ldots||\mathcal{E}_k)$, i.e. it is a configuration.

- $C$ is cycle-free:
  By construction, all $\mathcal{E}_k$ are cycle-free. So, cycles may only arise because of synchronization.
  Assume there are timestamps $t_1, t_2$ and events $(d_1, \ldots, d_n)$ (labelled with $t_1$) and $(e_1, \ldots, e_n)$
  (labelled with $t_2$) such that $(d_1, \ldots, d_n) \rightarrow^+ (e_1, \ldots, e_n)$ and $(e_1, \ldots, e_n) \rightarrow^+ (d_1, \ldots, d_n)$.
  This ordering has to come from some $\mathcal{E}_k$, hence contradicts timestamp orderedness of all $\mathcal{E}_k$.

- $C$ is conflict free:
  This holds because all $\mathcal{E}_k$ are conflict-free and no event gets paired with two or more different events.

- $C$ is left-closed up to conflicts. Let $(d_1, \ldots, d_n), (e_1, \ldots, e_n) \in E_{\mathcal{E}_1||\ldots||\mathcal{E}_n}$ such that we have
  $(d_1, \ldots, d_n) \rightarrow (e_1, \ldots, e_n)$ and $(e_1, \ldots, e_n) \in C$, however, $(d_1, \ldots, d_n) \notin C$. Assume
  w.l.o.g. that the flow is coming from $d_k \rightarrow e_k$. By construction of $C$, the event $(d_1, \ldots, d_n)$
  has to have either (1) $\ell_k(d_k) = \text{rd}_x^x(\cdot, \cdot)$ and there exists some $j$ such that $d_j = *$
  but $t \in T_{rd}(\tau_j)$ or $d_1 = *$ or (2) $\ell_k(d_k) = \text{ff}_x^x(\cdot, \cdot)$ and exists some $j$ such that $d_j = *$
  but $t \in T_{rd}(\tau_j)$ (otherwise the construction of $C$ would have included the event). In both cases
  $e_t \#(d_1, \ldots, d_n)$ and $e_t \rightarrow (e_1, \ldots, e_n)$ guaranteeing left-closedness up to conflicts.

Next, we look at interference freedom. Note that by construction $C$ is thread-covering. It contains no unsynchronised reads as the read events in the trace need to read from a promise in memory, hence there has to exist a fulfill. For the ordering $\lessdot$ we now take the ordering as induced by the timestamps on events. By timestamp ordering we then get that $\rightarrow_k^C \cap (\mathcal{E}_C \times \mathcal{E}_C) \subseteq \lessdot$. Note furthermore that by construction $(\mathcal{T}(\tau_k), M) \in \llbracket \mathcal{E}_k \rrbracket$,
$k \in \{1, \ldots, n\}$, using the mapping $\psi$ as detailed above. Hence, the location restrictions are also met.