Rely-Guarantee Reasoning for Causally Consistent Shared Memory[*]

Ori Lahav[1], Brijesh Dongol[2], and Heike Wehrheim[3]

[1] Tel Aviv University, Tel Aviv, Israel
[2] University of Surrey, Guildford, UK
[3] University of Oldenburg, Oldenburg, Germany

Abstract. Rely-guarantee (RG) is a highly influential compositional proof technique for concurrent programs, which was originally developed assuming a sequentially consistent shared memory. In this paper, we first generalize RG to make it parametric with respect to the underlying memory model by introducing an RG framework that is applicable to any model axiomatically characterized by Hoare triples. Second, we instantiate this framework for reasoning about concurrent programs under *causally consistent memory*, which is formulated using a recently proposed *potential-based* operational semantics, thereby providing the first reasoning technique for such semantics. The proposed program logic, which we call Piccolo, employs a novel assertion language allowing one to specify ordered sequences of states that each thread may reach. We employ Piccolo for multiple litmus tests, as well as for an adaptation of Peterson's algorithm for mutual exclusion to causally consistent memory.

1 Introduction

Rely-guarantee (RG) is a fundamental compositional proof technique for concurrent programs [21,48]. Each program component P is specified using *rely* and *guarantee* conditions, which means that P can tolerate any environment interference that follows its rely condition, and generate only interference included in its guarantee condition. Two components can be composed in parallel provided that the rely of each component agrees with the guarantee of the other.

The original RG framework and its soundness proof have assumed a sequentially consistent (SC) memory [33], which is unrealistic in modern processor architectures and programming languages. Nevertheless, the main principles behind RG are not at all specific for SC. Accordingly, our first main contribution, is to formally decouple the underlying memory model from the RG proof principles, by proposing a generic RG framework parametric in the input memory model.

To do so, we assume that the underlying memory model is axiomatized by Hoare triples specifying pre- and postconditions on memory states for each primitive operation (e.g., loads and stores). This enables the formal development of RG-based logics for different shared memory models as instances of one framework, where all build on a uniform soundness infrastructure of the RG rules (e.g., for sequential and parallel composition), but employ different specialized assertions to describe the possible memory states, where specific soundness arguments are only needed for primitive memory operations.

The second contribution of this paper is an instance of the general RG framework for *causally consistent shared memory*. The latter stands for a family of wide-spread and well-studied memory models weaker than SC, which are sufficiently strong for implementing a variety of synchronization idioms [6, 12, 26]. Intuitively, unlike SC, causal consistency allows different threads to observe writes to memory in different orders, as long as they agree on the order of writes that are causally related. This concept can be formalized in multiple ways, and here we target a strong form of causal consistency, called *strong release-acquire* (SRA) [28, 31] (and equivalent to "causal convergence" from [12]), which is a slight strengthening of the well-known release-acquire (RA) model (used by C/C++11). (The variants of causal consistency only differ for programs with write/write races [10, 28], which are rather rare in practice.)

Our starting point for axiomatizing SRA as Hoare triples is the *potential-based* operational semantics of SRA, which was recently introduced with the goal of establishing the decidability of control state reachability under this model [27, 28] (in contrast to undecidability under RA [1]). Unlike more standard presentations of weak memory models whose states record information about the *past* (e.g., in the form of store buffers containing executed writes before they are globally visible [36], partially ordered execution graphs [8, 20, 31], or collections of timestamped messages and thread views [11, 16, 17, 23, 25, 47]), the states of the potential-based model track possible *futures* ascribing what sequences of observations each thread can perform. We find this approach to be a particularly appealing candidate for Hoare-style reasoning which would naturally generalize SC-based reasoning. Intuitively, while an assertion in SC specifies possible observations at a given program point, an assertion in a potential-based model should specify possible *sequences* of observations.

To pursue this direction, we introduce a novel assertion language, resembling temporal logics, which allows one to express properties of sequences of states. For instance, our assertions can express that a certain thread may currently read $x = 0$, but it will have to read $x = 1$ once it reads $y = 1$. Then, we provide Hoare triples for SRA in this assertion language, and incorporate them in the general RG framework. The resulting program logic, which we call Piccolo, provides a novel approach to reason on concurrent programs under causal consistency, which allows for simple and direct proofs, and, we believe, may constitute a basis for automation in the future.
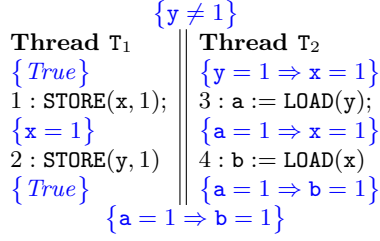
$$\{y \neq 1\}$$

Thread T_1	**Thread** T_2
$\{\mathit{True}\}$	$\{y = 1 \Rightarrow x = 1\}$
$1 : \texttt{STORE}(x,1);$	$3 : a := \texttt{LOAD}(y);$
$\{x = 1\}$	$\{a = 1 \Rightarrow x = 1\}$
$2 : \texttt{STORE}(y,1)$	$4 : b := \texttt{LOAD}(x)$
$\{\mathit{True}\}$	$\{a = 1 \Rightarrow b = 1\}$

$$\{a = 1 \Rightarrow b = 1\}$$

Fig. 1. Message passing in SC

$$\{T_0 \ltimes [y \neq 1]\}$$

Thread T_1	**Thread** T_2
$\{\mathit{True}\}$	$\{T_2 \ltimes [y \neq 1] \,;\, [x = 1]\}$
$1 : \texttt{STORE}(x,1);$	$3 : a := \texttt{LOAD}(y);$
$\{T_1 \ltimes [x = 1]\}$	$\{a = 1 \Rightarrow T_2 \ltimes [x = 1]\}$
$2 : \texttt{STORE}(y,1)$	$4 : b := \texttt{LOAD}(x)$
$\{\mathit{True}\}$	$\{a = 1 \Rightarrow b = 1\}$

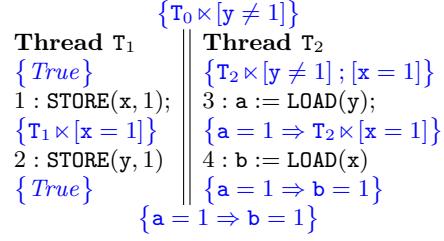$$\{a = 1 \Rightarrow b = 1\}$$

Fig. 2. Message passing in SRA

2 Motivating Example

To make our discussion concrete, consider the message passing program (MP) in Figures 1 and 2, comprising shared variables x and y and local registers a and b. The proof outline in Fig. 1 assumes SC, whereas Fig. 2 assumes SRA. In both cases, at the end of the execution, we show that if a is 1, then b must also be 1. We use these examples to explain the two main concepts introduced in this paper: (*i*) a generic RG framework and (*ii*) its instantiation with a potential-focused assertion system that enables reasoning under SRA.

Rely-Guarantee. The proof outline in Fig. 1 can be read as an RG derivation:

1. Thread T_1 locally establishes its postcondition when starting from any state that satisfies its precondition. This is trivial since its postcondition is *True*.
2. Thread T_1 relies on the fact that its used assertions are *stable* w.r.t. interference from its environment. We formally capture this condition by a rely set $\mathcal{R}_1 \triangleq \{\mathit{True}, x = 1\}$.
3. Thread T_1 guarantees to its concurrent environment that its only interferences are $\texttt{STORE}(x,1)$ and $\texttt{STORE}(y,1)$, and furthermore that $\texttt{STORE}(y,1)$ is only performed when $x = 1$ holds. We formally capture this condition by a guarantee set $\mathcal{G}_1 \triangleq \{\{\mathit{True}\}\ T_1 \mapsto \texttt{STORE}(x,1), \{x = 1\}\ T_1 \mapsto \texttt{STORE}(y,1)\}$, where each element is a command guarded by a precondition.
4. Thread T_2 locally establishes its postcondition when starting from any state that satisfies its precondition. This is straightforward using standard Hoare rules for assignment and sequential composition.
5. Thread T_2's rely set is again obtained by collecting all the assertions used in its proof: $\mathcal{R}_2 \triangleq \{y = 1 \Rightarrow x = 1, a = 1 \Rightarrow x = 1, a = 1 \Rightarrow b = 1\}$. Indeed, the local reasoning for T_2 needs all these assertions to be stable under the environment interference.
6. Thread T_2's guarantee set is given by:

$$\mathcal{G}_2 \triangleq \{\{y = 1 \Rightarrow x = 1\}\ T_2 \mapsto a := \texttt{LOAD}(y), \{a = 1 \Rightarrow x = 1\}\ T_2 \mapsto b := \texttt{LOAD}(x)\}$$

7. To perform the parallel composition, $\langle \mathcal{R}_1, \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2, \mathcal{G}_2 \rangle$ should be *non-interfering*. This involves showing that each $R \in \mathcal{R}_i$ is *stable* under each $G \in \mathcal{G}_j$ for $i \neq j$. That is, if $G = \{P\}\ \tau \mapsto c$, we require the Hoare triple

$\{P \cap R\} \ \tau \mapsto c \ \{R\}$ to hold. In this case, these proof obligations are straightforward to discharge using Hoare's assignment axiom (and is trivial for $i = 1$ and $j = 2$ since load instructions leave the memory intact).

Remark 1. Classical treatments of RG involve two related ideas [21]: (1) specifying a component by rely and guarantee conditions (together with standard pre- and postconditions); and (2) taking the relies and guarantees to be binary relations over states. Our approach adopts (1) but not (2). Thus, it can be seen as an RG presentation of the Owicki-Gries method [37], as was previously done in [32]. We have not observed an advantage for using binary relations in our examples, but the framework can be straightforwardly modified to do so.

Now, observe that substantial aspects of the above reasoning are *not* directly tied with SC. This includes the Hoare rules for compound commands (such as sequential composition above), the idea of specifying a thread using collections of stable rely assertions and guaranteed guarded primitive commands, and the non-interference condition for parallel composition. To carry out this generalization, we assume that we are provided an assertion language whose assertions are interpreted as *sets of memory states* (which can be much more involved than simple mappings of variables to values), and a set of valid Hoare triples for the primitive instructions. The latter is used for checking validity of primitive triples, (e.g., $\{P\} \ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{x}, 1) \ \{Q\}$), as well as non-interference conditions (e.g., $\{P \cap R\} \ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{x}, 1) \ \{R\}$). In §4, we present this generalization, and establish the soundness of RG principles independently of the memory model.

Potential-based reasoning. The second contribution of our work is an application of the above to develop a logic for a potential-based operational semantics that captures SRA. In this semantics every memory state records sequences of store mappings (from shared variables to values) that each thread may observe. For example, assuming all variables are initialized to 0, if $\mathtt{T}_1$ executed its code until completion before $\mathtt{T}_2$ even started (so under SC the memory state is the store $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}$), we may reach the SRA state in which $\mathtt{T}_1$'s potential consists of one store $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}$, and $\mathtt{T}_2$'s potential is the sequence of stores:

$$\langle \{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 0\}, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\} \rangle,$$

which captures the stores that $\mathtt{T}_2$ may observe in the order it may observe them. Naturally, potentials are *lossy* allowing threads to non-deterministically lose a subsequence of the current store sequence, so they can progress in their sequences. Thus, $\mathtt{T}_2$ can read 1 from $\mathtt{y}$ only after it loses the first two stores in its potential, and from this point on it can only read 1 from $\mathtt{x}$. Now, one can see that *all* potentials of $\mathtt{T}_2$ at its initial program point are, in fact, subsequences of the above sequence (regardless of where $\mathtt{T}_1$ is), and conclude that $\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1$ holds when $\mathtt{T}_2$ terminates.

To capture the above informal reasoning in a Hoare logic, we designed a new form of assertions capturing possible locally observable sequences of stores, rather than one global store, which can be seen as a restricted fragment of linear

temporal logic. The proof outline using these assertions is given in Fig. 2. In particular, $[\mathtt{x} = 1]$ is satisfied by all store sequences in which every store maps $\mathtt{x}$ to 1, whereas $[\mathtt{y} \neq 1] \, ; [\mathtt{x} = 1]$ is satisfied by all store sequences that can be split into a (possibly empty) prefix whose value for $\mathtt{y}$ is not 1 followed by a (possibly empty) suffix whose value for $\mathtt{x}$ is 1. Assertions of the form $\tau \ltimes I$ state that the potential of thread τ includes only store sequences that satisfy I.

The first assertion of $\mathtt{T}_2$ is implied by the initial condition, $\mathtt{T}_0 \ltimes [\mathtt{y} \neq 1]$, since the potential of the parent thread $\mathtt{T}_0$ is inherited by the forked child threads and $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1]$ implies $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] \, ; I$ for any I. Moreover, $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] \, ; [\mathtt{x} = 1]$ is preserved by (i) line 1 because writing 1 to $\mathtt{x}$ leaves $[\mathtt{y} \neq 1]$ unchanged and re-establishes $[\mathtt{x} = 1]$; and (ii) line 2 because the semantics for SRA ensures that after reading 1 from $\mathtt{y}$ by $\mathtt{T}_2$, the thread $\mathtt{T}_2$ is confined by $\mathtt{T}_1$'s potential just before it wrote 1 to $\mathtt{y}$, which has to satisfy the precondition $\mathtt{T}_1 \ltimes [\mathtt{x} = 1]$. (SRA allows to update the other threads' potential only when the suffix of the potential after the update is observable by the writer thread.)

In §6 we formalize these arguments as Hoare rules for the primitive instructions, whose soundness is checked using the potential-based operational semantics and the interpretation of the assertion language. Finally, Piccolo is obtained by incorporating these Hoare rules in the general RG framework.

Remark 2. Our presentation of the potential-based semantics for SRA (fully presented in §5) deviates from the original one in [28], where it was called loSRA. The most crucial difference is that while loSRA's potentials consist of lists of per-location read options, our potentials consist of lists of *stores* assigning a value to every variable. (This is similar in spirit to the adaptation of load buffers for TSO [4,5] to snapshot buffers in [2]). Additionally, unlike loSRA, we disallow empty potential lists, require that the potentials of the different threads agree on the very last value to each location, and handle read-modify-write (RMW) instructions differently. We employed these modifications to loSRA as we observed that direct reasoning on loSRA states is rather unnatural and counterintuitive, as loSRA allows traces that *block* a thread from reading any value from certain locations (which cannot happen in the version we formulate). For example, a direct interpretation of our assertions over loSRA states would allow states in which $\tau \ltimes [x = v]$ and $\tau \ltimes [x \neq v]$ both hold (when τ does not have any option to read from x), while these assertions are naturally contradictory when interpreted on top of our modified SRA semantics. To establish confidence in the new potential-based semantics we have proved *in Coq* its equivalence to the standard execution-graph based semantics of SRA (over 5K lines of Coq proofs) [29].

3 Preliminaries: Syntax and Semantics

In this section we describe the underlying program language, leaving the shared-memory semantics parametric.

Syntax. The syntax of programs, given in Fig. 3, is mostly standard, comprising primitive (atomic) commands c and compound commands C. The non-standard

$$values \quad v \in \mathsf{Val} = \{0, 1, ...\} \qquad shared\ variables \quad x, y \in \mathsf{Loc} = \{\mathtt{x}, \mathtt{y}, ...\}$$
$$local\ registers \quad r \in \mathsf{Reg} = \{\mathtt{a}, \mathtt{b}, ...\} \quad thread\ identifiers \quad \tau, \pi \in \mathsf{Tid} = \{\mathtt{T_0}, \mathtt{T_1}, ...\}$$

$$e ::= r \mid v \mid e + e \mid e = e \mid \neg e \mid e \wedge e \mid e \vee e \mid ...$$
$$c ::= r := e \mid \mathtt{STORE}(x, e) \mid r := \mathtt{LOAD}(x) \mid \mathtt{SWAP}(x, e) \qquad \tilde{c} ::= \langle c, \vec{r} := \vec{e} \rangle$$
$$C ::= c \mid \tilde{c} \mid \mathtt{skip} \mid C\,;C \mid \mathtt{if}\ e\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\ e\ \mathtt{do}\ C \mid C\ ^{\tau_1}||^{\tau_2}\ C$$

Fig. 3. Program syntax

$$\frac{\gamma' = \gamma[r \mapsto \gamma(e)]}{r := e \gg \gamma \xrightarrow{\varepsilon} \gamma'} \qquad \frac{l = \mathtt{W}(x, \gamma(e))}{\mathtt{STORE}(x, e) \gg \gamma \xrightarrow{l} \gamma} \qquad \frac{l = \mathtt{R}(x, v) \qquad \gamma' = \gamma[r \mapsto v]}{r := \mathtt{LOAD}(x) \gg \gamma \xrightarrow{l} \gamma'}$$

$$\frac{l = \mathtt{RMW}(x, v, \gamma(e))}{\mathtt{SWAP}(x, e) \gg \gamma \xrightarrow{l} \gamma} \qquad \frac{\begin{array}{c} c \gg \gamma \xrightarrow{l_\varepsilon} \gamma_0 \\ r_1 := e_1 \gg \gamma_0 \xrightarrow{\varepsilon} \gamma_1 \quad ... \quad r_n := e_n \gg \gamma_{n-1} \xrightarrow{\varepsilon} \gamma_n \end{array}}{\langle c, \langle r_1, ..., r_n \rangle := \langle e_1, ..., e_n \rangle \rangle \gg \gamma \xrightarrow{l_\varepsilon} \gamma_n}$$

Fig. 4. Small-step semantics of (instrumented) primitive commands $(\tilde{c} \gg \gamma \xrightarrow{l_\varepsilon} \gamma')$

components are instrumented commands $\tilde{c}$, which are meant to atomically execute a primitive command c and a (multiple) assignment $\vec{r} := \vec{e}$. Such instructions are needed to support auxiliary (a.k.a. ghost) variables in RG proofs. In addition, $\mathtt{SWAP}$ (a.k.a. atomic exchange) is an example of an RMW instruction. For brevity, other standard RMW instructions, such as $\mathtt{FADD}$ and $\mathtt{CAS}$, are omitted.

Unlike many weak memory models that only support top-level parallelism, we include dynamic thread creation via commands of the form $C_1\ ^{\tau_1}||^{\tau_2}\ C_2$ that forks two threads named τ_1 and τ_2 that execute the commands C_1 and C_2, respectively. Each C_i may itself comprise further parallel compositions. Since thread identifiers are explicit, we require commands to be *well formed*. Let $\mathsf{Tid}(C)$ be the set of all thread identifiers that appear in C. A command C is *well formed*, denoted $\mathsf{wf}(C)$, if parallel compositions inside employ disjoint sets of thread identifiers. This notion is formally defined by induction on the structure of commands, with the only interesting case being $\mathsf{wf}(C_1\ ^{\tau_1}||^{\tau_2}\ C_2)$ if $\mathsf{wf}(C_1) \wedge \mathsf{wf}(C_2) \wedge \tau_1 \neq \tau_2 \wedge \mathsf{Tid}(C_1) \cap \mathsf{Tid}(C_2) = \emptyset$.

Program semantics. We provide small-step operational semantics to commands independently of the memory system. To connect this semantics to a given memory system, its steps are instrumented with labels, as defined next.

Definition 1. A *label* l takes one of the following forms: a read $\mathtt{R}(x, v_{\mathtt{R}})$, a write $\mathtt{W}(x, v_{\mathtt{W}})$, a read-modify-write $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$, a fork $\mathtt{FORK}(\tau_1, \tau_2)$, or a join $\mathtt{JOIN}(\tau_1, \tau_2)$, where $x \in \mathsf{Loc}$, $v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathsf{Val}$, and $\tau_1, \tau_2 \in \mathsf{Tid}$. We denote by Lab the set of all labels.

$$\frac{\tilde{c} \gg \gamma \xrightarrow{l_\varepsilon} \gamma'}{\langle \tilde{c}, \gamma \rangle \xrightarrow{l_\varepsilon} \langle \mathtt{skip}, \gamma' \rangle} \qquad \frac{\langle C_1, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C_1', \gamma' \rangle}{\langle C_1 \,;\, C_2, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C_1' \,;\, C_2, \gamma' \rangle} \qquad \frac{}{\langle \mathtt{skip} \,;\, C_2, \gamma \rangle \xrightarrow{\varepsilon} \langle C_2, \gamma \rangle}$$

$$\frac{\gamma(e) = true \Rightarrow i = 1 \qquad \gamma(e) \neq true \Rightarrow i = 2}{\langle \mathtt{if}\ e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, \gamma \rangle \xrightarrow{\varepsilon} \langle C_i, \gamma \rangle} \qquad \frac{C' = \mathtt{if}\ e\ \mathtt{then}\ (C \,;\, \mathtt{while}\ e\ \mathtt{do}\ C)\ \mathtt{else}\ \mathtt{skip}}{\langle \mathtt{while}\ e\ \mathtt{do}\ C, \gamma \rangle \xrightarrow{\varepsilon} \langle C', \gamma \rangle}$$

Fig. 5. Small-step semantics of commands ($\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle$)

$$\frac{\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle}{\langle \mathcal{C}_0 \uplus \{\tau \mapsto C\}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle \mathcal{C}_0 \uplus \{\tau \mapsto C'\}, \gamma \rangle} \qquad \frac{\begin{array}{c} \mathcal{C}(\tau) = C_1\ {}^{\tau_1}||^{\tau_2}\ C_2 \\ \tau_1 \notin dom(\mathcal{C}) \qquad \tau_2 \notin dom(\mathcal{C}) \\ l = \mathtt{FORK}(\tau_1, \tau_2) \\ \mathcal{C}' = \{\tau_1 \mapsto C_1, \tau_2 \mapsto C_2\} \end{array}}{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l} \langle \mathcal{C} \uplus \mathcal{C}', \gamma \rangle} \qquad \frac{\begin{array}{c} \mathcal{C} = \left\{\begin{array}{l} \tau \mapsto C_1\ {}^{\tau_1}||^{\tau_2}\ C_2, \\ \tau_1 \mapsto \mathtt{skip}, \tau_2 \mapsto \mathtt{skip} \end{array}\right\} \\ l = \mathtt{JOIN}(\tau_1, \tau_2) \\ \mathcal{C}' = \{\tau \mapsto \mathtt{skip}\} \end{array}}{\langle \mathcal{C}_0 \uplus \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l} \langle \mathcal{C}_0 \uplus \mathcal{C}', \gamma \rangle}$$

Fig. 6. Small-step semantics of command pools ($\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle \mathcal{C}', \gamma' \rangle$)

Definition 2. A *register store* is a mapping $\gamma : \mathsf{Reg} \to \mathsf{Val}$. Register stores are extended to expressions as expected. We denote by Γ the set of all register stores.

The semantics of (instrumented) primitive commands is given in Fig. 4. Using this definition, the semantics of commands is given in Fig. 5. Its steps are of the form $\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle$ where C and C' are commands, γ and γ' are register stores, and $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$ (ε denotes a thread internal step). We lift this semantics to *command pools* as follows.

Definition 3. A *command pool* is a non-empty partial function $\mathcal{C}$ from thread identifiers to commands, such that the following hold:

1. $\mathsf{Tid}(\mathcal{C}(\tau_1)) \cap \mathsf{Tid}(\mathcal{C}(\tau_2)) = \emptyset$ for every $\tau_1 \neq \tau_2$ in $dom(\mathcal{C})$.
2. $\tau \notin \mathsf{Tid}(\mathcal{C}(\tau))$ for every $\tau \in dom(\mathcal{C})$.

We write command pools as sets of the form $\{\tau_1 \mapsto C_1, \dots, \tau_n \mapsto C_n\}$.

Steps for command pools are given in Fig. 6. They take the form $\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle \mathcal{C}', \gamma' \rangle$, where $\mathcal{C}$ and $\mathcal{C}'$ are command pools, γ and γ' are register stores, and $\langle \tau : l_\varepsilon \rangle$ (with $\tau \in \mathsf{Tid}$ and $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$) is a *command transition label*.

Memory semantics. To give semantics to programs under a memory model, we synchronize the transitions of a command C with a memory system. We leave the memory system parametric, and assume that it is represented by a labeled transition system (LTS) $\mathcal{M}$ with set of states denoted by $\mathcal{M}.\mathtt{Q}$, and steps denoted by $\to_\mathcal{M}$. The transition labels of general memory system $\mathcal{M}$ consist of non-silent program transition labels (elements of $\mathsf{Tid} \times \mathsf{Lab}$) and a (disjoint) set $\mathcal{M}.\Theta$ of internal memory actions, which is again left parametric (used, e.g., for memory-internal propagation of values).

Example 1. The simple memory system that guarantees sequential consistency is denoted here by SC. This memory system tracks the most recent value written to each variable and has no internal transitions ($\mathsf{SC}.\Theta = \emptyset$). Formally, it is defined by $\mathsf{SC}.\mathbb{Q} \triangleq \mathsf{Loc} \to \mathsf{Val}$ and $\to_{\mathsf{SC}}$ is given by:

$$
\frac{\begin{array}{c}l = \mathtt{R}(x, v_{\mathtt{R}})\\ m(x) = v_{\mathtt{R}}\end{array}}{m \xrightarrow{\tau, l}_{\mathsf{SC}} m} \qquad \frac{\begin{array}{c}l = \mathtt{W}(x, v_{\mathtt{W}})\\ m' = m[x \mapsto v_{\mathtt{W}}]\end{array}}{m \xrightarrow{\tau, l}_{\mathsf{SC}} m'} \qquad \frac{\begin{array}{c}l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})\\ m(x) = v_{\mathtt{R}}\\ m' = m[x \mapsto v_{\mathtt{W}}]\end{array}}{m \xrightarrow{\tau, l}_{\mathsf{SC}} m'} \qquad \frac{l \in \{\mathtt{FORK}(\_,\_), \mathtt{JOIN}(\_,\_)\}}{m \xrightarrow{\tau, l}_{\mathsf{SC}} m}
$$

The composition of a program with a general memory system is defined next.

Definition 4. The *concurrent system* induced by a memory system $\mathcal{M}$, denoted by $\overline{\mathcal{M}}$, is the LTS whose transition labels are the elements of $(\mathsf{Tid} \times (\mathsf{Lab} \cup \{\varepsilon\})) \uplus \mathcal{M}.\Theta$; states are triples of the form $\langle \mathcal{C}, \gamma, m \rangle$ where $\mathcal{C}$ is a command pool, γ is a register store, and $m \in \mathcal{M}.\mathbb{Q}$; and the transitions are "synchronized transitions" of the program and the memory system, using labels to decide what to synchronize on, formally given by:

$$
\frac{\begin{array}{c}\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l} \langle \mathcal{C}', \gamma' \rangle\\ l \in \mathsf{Lab} \qquad m \xrightarrow{\tau, l}_{\mathcal{M}} m'\end{array}}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\tau, l}_{\overline{\mathcal{M}}} \langle \mathcal{C}', \gamma', m' \rangle} \qquad \frac{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, \varepsilon} \langle \mathcal{C}', \gamma' \rangle}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\tau, \varepsilon}_{\overline{\mathcal{M}}} \langle \mathcal{C}', \gamma', m \rangle} \qquad \frac{\begin{array}{c}\theta \in \mathcal{M}.\Theta\\ m \xrightarrow{\theta}_{\mathcal{M}} m'\end{array}}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\theta}_{\overline{\mathcal{M}}} \langle \mathcal{C}, \gamma, m' \rangle}
$$

4 Generic Rely-Guarantee Reasoning

In this section we present our generic RG framework. Rather than committing to a specific assertion language, our reasoning principles apply on the *semantic level*, using sets of states instead of syntactic assertions. The structure of proofs still follows program structure, thereby retaining RG's compositionality. By doing so, we decouple the semantic insights of RG reasoning from a concrete syntax. Next, we present proof rules serving as blueprints for memory model specific proof systems. An instantiation of this blueprint requires lifting the semantic principles to syntactic ones. More specifically, it requires

1. a language with (a) concrete assertions for specifying sets of states and (b) operators that match operations on sets of states (like $\wedge$ matches $\cap$); and
2. sound Hoare triples for primitive commands.

Thus, each instance of the framework (for a specific memory system) is left with the task of identifying useful abstractions on states, as well as a suitable formalism, for making the generic semantic framework into a proof system.

RG judgments. We let $\mathcal{M}$ be an arbitrary memory system and $\Sigma_{\mathcal{M}} \triangleq \Gamma \times \mathcal{M}.\mathbb{Q}$. Properties of programs $\mathcal{C}$ are stated via *RG judgments*:

$$
\mathcal{C} \; \underline{sat}_{\mathcal{M}} \; (P, \mathcal{R}, \mathcal{G}, Q)
$$

where $P, Q \subseteq \Sigma_\mathcal{M}$, $\mathcal{R} \subseteq \mathcal{P}(\Sigma_\mathcal{M})$, and $\mathcal{G}$ is a set of *guarded commands*, each of which takes the form $\{G\}\ \tau \mapsto \alpha$, where $G \subseteq \Sigma_\mathcal{M}$ and α is either an (instrumented) primitive command $\tilde{c}$ or a fork/join label (of the form $\texttt{FORK}(\tau_1, \tau_2)$ or $\texttt{JOIN}(\tau_1, \tau_2)$). The latter is needed for considering the effect of forks and joins on the memory state.

Interpretation of RG judgments. RG judgments $\mathcal{C}\ \underline{sat}_\mathcal{M}\ (P, \mathcal{R}, \mathcal{G}, Q)$ state that a terminating run of $\mathcal{C}$ starting from a state in P, under any concurrent context whose transitions preserve each of the sets of states in $\mathcal{R}$, will end in a state in Q and perform only transitions contained in $\mathcal{G}$. To formally define this statement, following the standard model for RG, these judgments are interpreted on *computations* of programs. Computations arise from runs of the concurrent system (see Def. 4) by abstracting away from concrete transition labels and including arbitrary "environment transitions" representing steps of the concurrent context. We have:

- *Component* transitions of the form $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\texttt{cmp}} \langle \mathcal{C}', \gamma', m' \rangle$.
- *Memory* transitions, which correspond to internal memory steps (labeled with $\theta \in \mathcal{M}.\boldsymbol{\Theta}$), of the form $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\texttt{mem}} \langle \mathcal{C}, \gamma, m' \rangle$.
- *Environment* transitions of the form $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\texttt{env}} \langle \mathcal{C}, \gamma', m' \rangle$.

Note that memory transitions do not occur in the classical RG presentation (since SC does not have internal memory actions).

A *computation* is a (potentially infinite) sequence

$$\xi = \langle \mathcal{C}_0, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \mathcal{C}_1, \gamma_1, m_1 \rangle \xrightarrow{a_2} \dots$$

with $a_i \in \{\texttt{cmp}, \texttt{env}, \texttt{mem}\}$. We let $\langle \mathcal{C}_{\mathsf{last}(\xi)}, \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle$ denotes its last element, when ξ is finite. We say that ξ is a computation *of a command pool* $\mathcal{C}$ when $\mathcal{C}_0 = \mathcal{C}$ and for every $i \geq 0$:

- If $a_i = \texttt{cmp}$, then $\langle \mathcal{C}_i, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon}_{\overline{\mathcal{M}}} \langle \mathcal{C}_{i+1}, \gamma_{i+1}, m_{i+1} \rangle$ for some $\tau \in \mathsf{Tid}$ and $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$.
- If $a_i = \texttt{mem}$, then $\langle \mathcal{C}_i, \gamma_i, m_i \rangle \xrightarrow{\theta}_{\overline{\mathcal{M}}} \langle \mathcal{C}_{i+1}, \gamma_{i+1}, m_{i+1} \rangle$ for some $\theta \in \mathcal{M}.\boldsymbol{\Theta}$.

We denote by $Comp(\mathcal{C})$ the set of all computations of a command pool $\mathcal{C}$.

To define validity of RG judgments, we use the following definition.

Definition 5. Let $\xi = \langle \mathcal{C}_0, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \mathcal{C}_1, \gamma_1, m_1 \rangle \xrightarrow{a_2} \dots$ be a computation, and $\mathcal{C}\ \underline{sat}_\mathcal{M}\ (P, \mathcal{R}, \mathcal{G}, Q)$ an RG-judgment.

- ξ admits P if $\langle \gamma_0, m_0 \rangle \in P$.
- ξ admits $\mathcal{R}$ if $\langle \gamma_i, m_i \rangle \in R \Rightarrow \langle \gamma_{i+1}, m_{i+1} \rangle \in R$ for every $R \in \mathcal{R}$ and $i \geq 0$ with $a_{i+1} = \texttt{env}$.
- ξ admits $\mathcal{G}$ if for every $i \geq 0$ with $a_{i+1} = \texttt{cmp}$ and $\langle \gamma_i, m_i \rangle \neq \langle \gamma_{i+1}, m_{i+1} \rangle$ there exists $\{P\}\ \tau \mapsto \alpha \in \mathcal{G}$ such that $\langle \gamma_i, m_i \rangle \in P$ and
  - if $\alpha = \tilde{c}$ is an instrumented primitive command, then for some $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$, we have $\langle \{\tau \mapsto \tilde{c}\}, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon}_{\overline{\mathcal{M}}} \langle \{\tau \mapsto \texttt{skip}\}, \gamma_{i+1}, m_{i+1} \rangle$

$$\text{SKIP} \quad \frac{}{\{\tau \mapsto \texttt{skip}\} \ \underline{sat}_{\mathcal{M}} \ (P, \{P\}, \emptyset, P)}$$

$$\text{COM} \quad \frac{\mathcal{M} \vDash \{P\} \ \tau \mapsto \tilde{c} \ \{Q\}}{\{\tau \mapsto \tilde{c}\} \ \underline{sat}_{\mathcal{M}} \ (P, \{P,Q\}, \{\{P\} \ \tau \mapsto \tilde{c}\}, Q)}$$

$$\text{SEQ} \quad \frac{\{\tau \mapsto C_1\} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R}_1, \mathcal{G}_1, R) \qquad \{\tau \mapsto C_2\} \ \underline{sat}_{\mathcal{M}} \ (R, \mathcal{R}_2, \mathcal{G}_2, Q)}{\{\tau \mapsto C_1 \, ; C_2\} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}$$

$$\text{IF} \quad \frac{\{\tau \mapsto C_1\} \ \underline{sat}_{\mathcal{M}} \ (P \cap \llbracket e \rrbracket, \mathcal{R}_1, \mathcal{G}_1, Q) \qquad \{\tau \mapsto C_2\} \ \underline{sat}_{\mathcal{M}} \ (P \setminus \llbracket e \rrbracket, \mathcal{R}_2, \mathcal{G}_2, Q)}{\{\tau \mapsto \texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2\} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P\}, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}$$

$$\text{WHILE} \quad \frac{P \setminus \llbracket e \rrbracket \subseteq Q \qquad \{\tau \mapsto C\} \ \underline{sat}_{\mathcal{M}} \ (P \cap \llbracket e \rrbracket, \mathcal{R}, \mathcal{G}, P)}{\{\tau \mapsto \texttt{while } e \texttt{ do } C\} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R} \cup \{P, Q\}, \mathcal{G}, Q)}$$

$$\text{PAR} \quad \frac{\begin{array}{c}\{\tau_1 \mapsto C_1\} \ \underline{sat}_{\mathcal{M}} \ (P_1, \mathcal{R}_1, \mathcal{G}_1, Q_1) \qquad \{\tau_2 \mapsto C_2\} \ \underline{sat}_{\mathcal{M}} \ (P_2, \mathcal{R}_2, \mathcal{G}_2, Q_2) \\ P \subseteq P_1 \cap P_2 \qquad Q_1 \cap Q_2 \subseteq Q \qquad \langle \mathcal{R}_1, \mathcal{G}_1 \rangle \text{ and } \langle \mathcal{R}_2, \mathcal{G}_2 \rangle \text{ are non-interfering}\end{array}}{\{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P\}, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}$$

$$\text{FORK-JOIN} \quad \frac{\begin{array}{c}\mathcal{M} \vDash \{P\} \ \tau \mapsto \texttt{FORK}(\tau_1, \tau_2) \ \{P'\} \qquad \mathcal{M} \vDash \{Q'\} \ \tau \mapsto \texttt{JOIN}(\tau_1, \tau_2) \ \{Q\} \\ \{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\} \ \underline{sat}_{\mathcal{M}} \ (P', \mathcal{R}, \mathcal{G}, Q') \\ \mathcal{G}' = \mathcal{G} \cup \{\{P\} \ \tau \mapsto \texttt{FORK}(\tau_1, \tau_2), \{Q'\} \ \tau \mapsto \texttt{JOIN}(\tau_1, \tau_2)\}\end{array}}{\{\tau \mapsto C_1 \ ^{\tau_1}||^{\tau_2} \ C_2\} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R} \cup \{P, Q\}, \mathcal{G}', Q)}$$

Fig. 7. Generic sequential RG proof rules (letting $\llbracket e \rrbracket = \{\langle \gamma, m \rangle \mid \gamma(e) = \textit{true}\}$)

- if $\alpha \in \{\texttt{FORK}(\tau_1, \tau_2), \texttt{JOIN}(\tau_1, \tau_2)\}$, then $m_i \xrightarrow{\tau, \alpha}_{\mathcal{M}} m_{i+1}$ and $\gamma_i = \gamma_{i+1}$.
- ξ admits Q if $\langle \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle \in Q$ whenever ξ is finite and $\mathcal{C}_{\mathsf{last}(\xi)}(\tau) = \texttt{skip}$ for every $\tau \in dom(\mathcal{C}_{\mathsf{last}(\xi)})$.

We denote by $\mathsf{Assume}(P, \mathcal{R})$ the set of all computations that admit P and $\mathcal{R}$, and by $\mathsf{Commit}(\mathcal{G}, Q)$ the set of all computations that admit $\mathcal{G}$ and Q.

Then, *validity* of a judgment if defined as

$$\vDash \mathcal{C} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R}, \mathcal{G}, Q) \stackrel{\triangle}{\Leftrightarrow} \mathit{Comp}(\mathcal{C}) \cap \mathsf{Assume}(P, \mathcal{R}) \subseteq \mathsf{Commit}(\mathcal{G}, Q)$$

Memory triples. Our proof rules build on *memory triples*, which specify pre- and postconditions for primitive commands for a memory system $\mathcal{M}$.

Definition 6. A *memory triple for a memory system* $\mathcal{M}$ is a tuple of the form $\{P\} \ \tau \mapsto \alpha \ \{Q\}$, where $P, Q \subseteq \Sigma_{\mathcal{M}}$, $\tau \in \mathsf{Tid}$, and α is either an instrumented primitive command, a fork label, or a join label. A memory triple for $\mathcal{M}$ is *valid*, denoted by $\mathcal{M} \vDash \{P\} \ \tau \mapsto \alpha \ \{Q\}$, if the following hold for every $\langle \gamma, m \rangle \in P$, $\gamma' \in \Gamma$ and $m' \in \mathcal{M}.\mathtt{Q}$:

- if α is an instrumented primitive command and $\langle \{\tau \mapsto \alpha\}, \gamma, m \rangle \xrightarrow{\tau, l_\varepsilon}_{\overline{\mathcal{M}}} \langle \{\tau \mapsto \texttt{skip}\}, \gamma', m' \rangle$ for some $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$, then $\langle \gamma', m' \rangle \in Q$.

10

– If $\alpha \in \{\texttt{FORK}(\tau_1, \tau_2), \texttt{JOIN}(\tau_1, \tau_2)\}$ and $m \xrightarrow{\tau, \alpha}_{\mathcal{M}} m'$, then $\langle \gamma, m' \rangle \in Q$.

Example 2. For the memory system SC introduced in Ex. 1, we have, e.g., memory triples of the form $\mathsf{SC} \vDash \{e(r := x)\} \; \tau \mapsto r := \texttt{LOAD}(x) \; \{e\}$ (where $e(r := x)$ is the expression e with all occurrences of r replaced by x).

RG proof rules. We aim at proof rules deriving valid RG judgments. Figure 7 lists (semantic) proof rules based on externally provided memory triples. These rules basically follows RG reasoning for sequential consistency. For example, rule SEQ states that RG judgments of commands C_1 and C_2 can be combined when the postcondition of C_1 and the precondition of C_2 agree, thereby uniting their relies and guarantees. Rule COM builds on memory triples. The rule PAR for parallel composition combines judgments for two components when their relies and guarantees are *non-interfering*. Intuitively speaking, this means that each of the assertions that each thread relied on for establishing its proof is preserved when applying any of the assignments collected in the guarantee set of the other thread. An example of non-interfering rely-guarantee pairs is given in step 7 in §2. Formally, non-interference is defined as follows:

Definition 7. Rely-guarantee pairs $\langle \mathcal{R}_1, \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2, \mathcal{G}_2 \rangle$ are *non-interfering* if $\mathcal{M} \vDash \{R \cap P\} \; \tau \mapsto \alpha \; \{R\}$ holds for every $R \in \mathcal{R}_1$ and $\{P\} \; \tau \mapsto \alpha \in \mathcal{G}_2$, and similarly for every $R \in \mathcal{R}_2$ and $\{P\} \; \tau \mapsto \alpha \in \mathcal{G}_1$.

In turn, FORK-JOIN combines the proof of a parallel composition with proofs of fork and join steps (which may also affect the memory state). Note that the guarantees also involve guarded commands with FORK and JOIN labels.

Additional rules for consequence and introduction of auxiliary variables are elided here (they are similar to their SC counterparts), and provided in the extended version of this paper [30].

Soundness. To establish soundness of the above system we need an additional requirement regarding the internal memory transitions (for SC this closure vacuously holds as there are no such transitions). We require all relies in $\mathcal{R}$ to be *stable under internal memory transitions*, i.e. for $R \in \mathcal{R}$ we require

$$\forall \gamma, m, m', \theta \in \mathcal{M}.\mathbf{\Theta}. \, m \xrightarrow{\theta}_{\mathcal{M}} m' \Rightarrow (\langle \gamma, m \rangle \in R \Rightarrow \langle \gamma, m' \rangle \in R) \qquad \text{(mem)}$$

This condition is needed since the memory system can non-deterministically take its internal steps, and the component's proof has to be stable under such steps.

With this requirement, we are able to establish soundness. The proof, which generally follows [48] is is given in the extended version of this paper [30]. We write $\vdash \mathcal{C} \; \underline{sat}_{\mathcal{M}} \; (P, \mathcal{R}, \mathcal{G}, Q)$ for provability of a judgment using the semantic rules presented above.

Theorem 1 (Soundness). $\vdash C \; \underline{sat}_{\mathcal{M}} \; (P, \mathcal{R}, \mathcal{G}, Q) \implies \vDash C \; \underline{sat}_{\mathcal{M}} \; (P, \mathcal{R}, \mathcal{G}, Q)$.

5   Potential-based Memory System for SRA

In this section we present the potential-based semantics for Strong Release-Acquire (SRA), for which we develop a novel RG logic. Our semantics is based on the one in [27, 28], with certain adaptations to make it better suited for Hoare-style reasoning (see Remark 2).

In weak memory models, threads typically have different views of the shared memory. In SRA, we refer to a memory snapshot that a thread may observe as a *potential store*:

Definition 8. A *potential store* is a function $\delta : \mathsf{Loc} \to \mathsf{Val} \times \{\mathtt{R}, \mathtt{RMW}\} \times \mathsf{Tid}$. We write $\mathtt{val}(\delta(x))$, $\mathtt{rmw}(\delta(x))$, and $\mathtt{tid}(\delta(x))$ to retrieve the different components of $\delta(x)$. We denote by Δ the set of all potential stores.

Having $\delta(x) = \langle v, \mathtt{R}, \tau \rangle$ allows to read the value v from x (and further ascribes that this read reads from a write performed by thread τ, which is technically needed to properly characterize the SRA model). In turn, having $\delta(x) = \langle v, \mathtt{RMW}, \tau \rangle$ further allows to perform an RMW instruction that atomically reads and modifies x.

Potential stores are collected in *potential store lists* describing the values which can (potentially) be read and in what order.

Notation 9 Lists over an alphabet A are written as $L = a_1 \cdot ... \cdot a_n$ where $a_1, ..., a_n \in A$. We also use $\cdot$ to concatenate lists, and write $L[i]$ for the i'th element of L and $|L|$ for the length of L.

A *(potential) store list* is a finite sequence of potential stores ascribing a possible sequence of stores that a thread can observe, in the order it will observe them. The RMW-flags in these lists have to satisfy certain conditions: once the flag for a location is set, it remains set in the rest of the list; and the flag must be set at the end of the list. Formally, store lists are defined as follows.

Definition 10. A *store list* $L \in \mathcal{L}$ is a non-empty finite sequence of potential stores with *monotone RMW-flags* ending with an RMW, that is: for all $x \in \mathsf{Loc}$,

1. if $\mathtt{rmw}(L[i](x)) = \mathtt{RMW}$, then $\mathtt{rmw}(L[j](x)) = \mathtt{RMW}$ for every $i < j \leq |L|$, and
2. $\mathtt{rmw}(L[|L|](x)) = \mathtt{RMW}$.

Now, SRA states (SRA.Q) consist of *potential mappings* that assign potentials to threads as defined next.

Definition 11. A *potential* D is a non-empty set of potential store lists. A *potential mapping* is a function $\mathcal{D} : \mathsf{Tid} \rightharpoonup \mathcal{P}(\mathcal{L}) \backslash \{\emptyset\}$ that maps thread identifiers to potentials such that all lists agree on the very final potential store (that is: $L_1[|L_1|] = L_2[|L_2|]$ whenever $L_1 \in \mathcal{D}(\tau_1)$ and $L_2 \in \mathcal{D}(\tau_2)$).

These potential mappings are "lossy" meaning that potential stores can be arbitrarily dropped. In particular, dropping the first store in a list enables reading from the second. This is formally done by transitioning from a state $\mathcal{D}$ to a "smaller" state $\mathcal{D}'$ as defined next.

12

WRITE
$$\frac{\begin{array}{c} \forall L' \in \mathcal{D}'(\tau).\, \exists L \in \mathcal{D}(\tau).\, L' = L[x \mapsto \langle v_{\mathtt{W}}, \mathtt{RMW}, \tau\rangle] \\ \forall \pi \in dom(\mathcal{D}) \setminus \{\tau\},\, L' \in \mathcal{D}'(\pi).\, \exists L_0, L_1. \\ L_0 \cdot L_1 \in \mathcal{D}(\pi) \wedge L_1 \in \mathcal{D}(\tau) \wedge \\ L' = L_0[x \mapsto \mathtt{R}] \cdot L_1[x \mapsto \langle v_{\mathtt{W}}, \mathtt{RMW}, \tau\rangle] \end{array}}{\mathcal{D} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{SRA}} \mathcal{D}'}$$

LOSE
$$\frac{\mathcal{D}' \sqsubseteq \mathcal{D}}{\mathcal{D} \xrightarrow{\varepsilon}_{\mathsf{SRA}} \mathcal{D}'}$$

DUP
$$\frac{\mathcal{D} \preceq \mathcal{D}'}{\mathcal{D} \xrightarrow{\varepsilon}_{\mathsf{SRA}} \mathcal{D}'}$$

READ
$$\frac{\exists \pi.\, \forall L \in \mathcal{D}(\tau).\, \mathtt{val}(L[1](x)) = v_{\mathtt{R}} \wedge \mathtt{tid}(L[1](x)) = \pi}{\mathcal{D} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\mathsf{SRA}} \mathcal{D}}$$

RMW
$$\frac{\forall L \in \mathcal{D}(\tau).\, \mathtt{rmw}(L[1](x)) = \mathtt{RMW} \quad \mathcal{D} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\mathsf{SRA}} \mathcal{D} \quad \mathcal{D} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{SRA}} \mathcal{D}'}{\mathcal{D} \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})}_{\mathsf{SRA}} \mathcal{D}'}$$

FORK
$$\frac{\mathcal{D}_{\mathrm{new}} = \{\tau_1 \mapsto \mathcal{D}(\tau), \tau_2 \mapsto \mathcal{D}(\tau)\} \quad \mathcal{D}' = \mathcal{D}|_{dom(\mathcal{D}) \setminus \{\tau\}} \uplus \mathcal{D}_{\mathrm{new}}}{\mathcal{D} \xrightarrow{\tau, \mathtt{FORK}(\tau_1, \tau_2)}_{\mathsf{SRA}} \mathcal{D}'}$$

JOIN
$$\frac{\mathcal{D}_{\mathrm{new}} = \{\tau \mapsto \mathcal{D}(\tau_1) \cap \mathcal{D}(\tau_2)\} \quad \mathcal{D}' = \mathcal{D}|_{dom(\mathcal{D}) \setminus \{\tau_1, \tau_2\}} \uplus \mathcal{D}_{\mathrm{new}}}{\mathcal{D} \xrightarrow{\tau, \mathtt{JOIN}(\tau_1, \tau_2)}_{\mathsf{SRA}} \mathcal{D}'}$$

Fig. 8. Steps of SRA (defining $\delta[x \mapsto \langle v, u, \tau\rangle](y) = \langle v, u, \tau\rangle$ if $y = x$ and $\delta(y)$ else, and $\delta[x \mapsto \mathtt{R}]$ to set all RMW-flags for x to R; both pointwise lifted to lists)

Definition 12. The (overloaded) partial order $\sqsubseteq$ is defined as follows:

1. on potential store lists: $L' \sqsubseteq L$ if L' is a nonempty subsequence of L;
2. on potentials: $D' \sqsubseteq D$ if $\forall L' \in D'.\, \exists L \in D.\, L' \sqsubseteq L$;
3. on potential mappings: $\mathcal{D}' \sqsubseteq \mathcal{D}$ if $\mathcal{D}'(\tau) \sqsubseteq \mathcal{D}(\tau)$ for every $\tau \in dom(\mathcal{D})$.

We also define $L \preceq L'$ if L' is obtained from L by duplication of some stores (e.g., $\delta_1 \cdot \delta_2 \cdot \delta_3 \preceq \delta_1 \cdot \delta_2 \cdot \delta_2 \cdot \delta_3$). This is lifted to potential mappings as expected.

Figure 8 defines the transitions of SRA. The LOSE and DUP steps account for losing and duplication in potentials. Note that these are both internal memory transitions (required to preserve relies as of (mem)). The FORK and JOIN steps distribute potentials on forked threads and join them at the end. The READ step obtains its value from the first store in the lists of the potential of the reader, provided that all these lists agree on that value and the writer thread identifier. RMW steps atomically perform a read and a write step where the read is restricted to an RMW-marked entry.

Most of the complexity is left for the WRITE step. It updates to the new written value for the writer thread τ. For every other thread, it updates a *suffix* (L_1) of the store list with the new value. For guaranteeing causal consistency this updated suffix cannot be arbitrary: it has to be in the potential of the writer thread ($L_1 \in \mathcal{D}(\tau)$). This is the key to achieving the "shared-memory causality principle" of [28], which ensures causal consistency.

Example 3. Consider again the MP program from Fig. 2. After the initial fork step, threads T_1 and T_2 may have the following store list in their potentials:

$$L = \begin{bmatrix} x \mapsto \langle 0, \text{RMW}, T_0 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 0, \text{RMW}, T_0 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 0, \text{RMW}, T_0 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix}.$$

Then, $\text{STORE}(x, 1)$ by T_1 can generate the following store list for T_2:

$$L_2 = \begin{bmatrix} x \mapsto \langle 0, \text{R}, T_0 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix}.$$

Thus T_2 keeps the possibility of reading the "old" value of x. For T_1 this is different: the model allows the writing thread to only see its new value of x and all entries for x in the store list are updated. Thus, for T_1 we obtain store list

$$L_1 = \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 0, \text{RMW}, T_0 \rangle \end{bmatrix}.$$

Next, when T_1 executes $\text{STORE}(y, 1)$, again, the value for y has to be updated to 1 in T_1 yielding

$$L_1' = \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_0 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix}.$$

For T_2 the write step may change L_2 to

$$L_2' = \begin{bmatrix} x \mapsto \langle 0, \text{R}, T_0 \rangle \\ y \mapsto \langle 0, \text{R}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 0, \text{R}, T_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix}.$$

Thus, thread T_2 can still see the old values, or lose the prefix of its list and see the new values. Importantly, it cannot read 1 from y *and then* 0 from x. Note that $\text{STORE}(y, 1)$ by T_1 *cannot* modify L_2 to the list

$$L_2'' = \begin{bmatrix} x \mapsto \langle 0, \text{R}, T_0 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} x \mapsto \langle 1, \text{RMW}, T_1 \rangle \\ y \mapsto \langle 1, \text{RMW}, T_1 \rangle \end{bmatrix},$$

as it requires T_1 to have L_2 in its *own potential*. This models the intended semantics of message passing under causal consistency.

The next theorem establishes the equivalence of SRA as defined above and opSRA from [28], which is an (operational version of) the standard strong release-acquire declarative semantics [26,31]. (As a corollary, we obtain the equivalence between the potential-based system from [28] and the variant we define in this paper.)

Our notion of equivalence employed in the theorem is *trace equivalence*. We let a trace of a memory system be a sequence of transition labels, ignoring ε transitions, and consider traces of SRA starting from an initial state $\lambda \tau \in \{T_1, ..., T_N\}. \{\langle \lambda x. \langle 0, \text{RMW}, T_0 \rangle \rangle\}$ and traces of opSRA starting from the initial execution graph that consists of a write event to every location writing 0 by a distinguished initialization thread T_0.

extended expressions	E	$::= e \mid x \mid \mathtt{R}(x) \mid E + E \mid \neg E \mid E \wedge E \mid \ldots$
interval assertions	I	$::= [E] \mid I\,;I \mid I \wedge I \mid I \vee I$
assertions	φ, ψ	$::= \tau \ltimes I \mid e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$

Fig. 9. Assertions of Piccolo

Theorem 2. *A trace is generated by* SRA *iff it is generated by* opSRA.

The proof is of this theorem is by simulation arguments (forward simulation in one direction and backward for the converse). It is mechanized in Coq [29]. The mechanized proof does not consider fork and join steps, but they can be straightforwardly added.

6   Program Logic

For the instantiation of our RG framework to SRA, we next (1) introduce the assertions of the logic Piccolo and (2) specify memory triples for Piccolo. Our logic is inspired by *interval logics* like Moszkowski's ITL [35] or duration calculus [13].

Syntax and semantics. Figure 9 gives the grammar of Piccolo. We base it on *extended expressions* which—besides registers—can also involve locations as well as expressions of the form $\mathtt{R}(x)$ (to indicate RMW-flag R). Extended expressions E can hold on entire *intervals* of a store list (denoted $[E]$). Store lists can be split into intervals satisfying different interval expressions $(I_1\,;\ldots;I_n)$ using the ";" operator (called "chop"). In turn, $\tau \ltimes I$ means that all store lists in τ's potential satisfy I. For an assertion φ, we let $fv(\varphi) \subseteq \mathsf{Reg} \cup \mathsf{Loc} \cup \mathsf{Tid}$ be the set of registers, locations and thread identifiers occurring in φ, and write $\mathtt{R}(x) \in \varphi$ to indicate that the term $\mathtt{R}(x)$ occurs in φ.

As an example consider again MP (Fig. 2). We would like to express that $\mathtt{T}_2$ upon seeing y to be 1 cannot see the old value 0 of x anymore. In Piccolo this is expressed as $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1]\,;[\mathtt{x} = 1]$: the store lists of $\mathtt{T}_2$ can be split into two intervals (one possibly empty), the first satisfying $\mathtt{y} \neq 1$ and the second $\mathtt{x} = 1$.

Formally, an assertion φ describes register stores coupled with SRA states:

Definition 13. Let γ be a register store, δ a potential store, L a store list, and $\mathcal{D}$ a potential mapping. We let $[\![e]\!]_{\langle \gamma, \delta \rangle} = \gamma(e)$, $[\![x]\!]_{\langle \gamma, \delta \rangle} = \delta(x)$, and $[\![\mathtt{R}(x)]\!]_{\langle \gamma, \delta \rangle} =$ if $\mathtt{rmw}(\delta(x)) = \mathtt{R}$ then *true* else *false*. The extension of this notation to any extended expression E is standard. The validity of assertions in $\langle \gamma, \mathcal{D} \rangle$, denoted by $\langle \gamma, \mathcal{D} \rangle \models \varphi$, is defined as follows:

1. $\langle \gamma, L \rangle \models [E]$ if $[\![E]\!]_{\langle \gamma, \delta \rangle} = true$ for every $\delta \in L$.
2. $\langle \gamma, L \rangle \models I_1\,;I_2$ if $\langle \gamma, L_1 \rangle \models I_1$ and $\langle \gamma, L_2 \rangle \models I_2$ for some (possibly empty) L_1 and L_2 such that $L = L_1 \cdot L_2$.
3. $\langle \gamma, L \rangle \models I_1 \wedge I_2$ if $\langle \gamma, L \rangle \models I_1$ and $\langle \gamma, L \rangle \models I_2$ (similarly for $\vee$).
4. $\langle \gamma, \mathcal{D} \rangle \models \tau \ltimes I$ if $\langle \gamma, L \rangle \models I$ for every $L \in \mathcal{D}(\tau)$.
5. $\langle \gamma, \mathcal{D} \rangle \models e$ if $\gamma(e) = true$.

Assumption	Pre Command Post	Reference
	$\{\varphi(r := e)\}$ $\tau \mapsto r := e$ $\{\varphi\}$	SUBST-ASGN
$x \notin fv(\varphi)$	$\{\varphi\}$ $\tau \mapsto \mathtt{WRITE}(x, e)$ $\{\varphi\}$	STABLE-WR
$r \notin fv(\varphi)$	$\{\varphi\}\tau \mapsto r := \mathtt{LOAD}(x)\{\varphi\}$	STABLE-LD
$\tau \notin fv(\varphi)$	$\{\varphi\}$ $\tau \mapsto \mathtt{FORK}(\tau_1, \tau_2)$ $\{\varphi\}$	STABLE-FORK
$\tau \notin fv(\varphi)$	$\{\varphi\}$ $\tau \mapsto \mathtt{JOIN}(\tau_1, \tau_2)$ $\{\varphi\}$	STABLE-JOIN
	$\{e \wedge \tau \ltimes I\}$ $\tau \mapsto \mathtt{FORK}(\tau_1, \tau_2)$ $\{e \wedge \tau_1 \ltimes I \wedge \tau_2 \ltimes I\}$	FORK
	$\{e \wedge \tau_1 \ltimes I \wedge \tau_2 \ltimes I\}$ $\tau \mapsto \mathtt{JOIN}(\tau_1, \tau_2)$ $\{e \wedge \tau \ltimes I\}$	JOIN
	$\{True\}$ $\tau \mapsto \mathtt{WRITE}(x, e)$ $\{\tau \ltimes [x = e]\}$	WR-OWN
$\mathtt{R}(x) \notin I$	$\{\tau \ltimes I\}$ $\tau \mapsto \mathtt{WRITE}(x, e)$ $\{\tau \ltimes (I \wedge [\mathtt{R}(x)]) \,;\, [x = e]\}$	WR-OTHER-1
$x \notin fv(I_\tau),$ $\mathtt{R}(x) \notin I$	$\{\tau \ltimes I_\tau \wedge \pi \ltimes I \,;\, I_\tau\}$ $\tau \mapsto \mathtt{WRITE}(x, e)$ $\{\pi \ltimes I \,;\, I_\tau\}$	WR-OTHER-2
$x \notin fv(I_\tau)$	$\{\tau \ltimes I_\tau\}$ $\tau \mapsto \mathtt{WRITE}(x, e)$ $\{\pi \ltimes [\mathtt{R}(x)] \,;\, I_\tau\}$	WR-OTHER-3
$x \notin fv(I)$	$\{\tau \ltimes [\mathtt{R}(x)] \,;\, I\}$ $\tau \mapsto \mathtt{SWAP}(x, e)$ $\{\tau \ltimes I\}$	SWAP-SKIP

Fig. 10. Memory triples for Piccolo using $\mathtt{WRITE} \in \{\mathtt{SWAP}, \mathtt{STORE}\}$ and assuming $\tau \neq \pi$

6. $\langle \gamma, \mathcal{D} \rangle \models \varphi_1 \wedge \varphi_2$ if $\langle \gamma, \mathcal{D} \rangle \models \varphi_1$ and $\langle \gamma, \mathcal{D} \rangle \models \varphi_2$ (similarly for $\vee$).

Note that with $\wedge$ and $\vee$ as well as negation on expressions,[4] the logic provides the operators on sets of states necessary for an instantiation of our RG framework. Further, the requirements from SRA states guarantee certain properties:

- For $\varphi_1 = \tau \ltimes [E_1^\tau] \,;\, ... \,;\, [E_n^\tau]$ and $\varphi_2 = \pi \ltimes [E_1^\pi] \,;\, ... \,;\, [E_m^\pi]$: if $E_i^\tau \wedge E_j^\pi \Rightarrow False$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$, then $\varphi_1 \wedge \varphi_2 \Rightarrow False$ (follows from the fact that all lists in potentials are non-empty and agree on the last store).
- If $\langle \gamma, \mathcal{D} \rangle \models \tau \ltimes [\mathtt{R}(x)] \,;\, [E]$, then every list $L \in \mathcal{D}(\tau)$ contains a non-empty suffix satisfying E (since all lists have to end with RMW-flags set on).

All assertions are preserved by steps LOSE and DUP. This stability is required by our RG framework (condition (mem))[5]. Stability is achieved here because negations occur on the level of (simple) expressions only (e.g., we cannot have $\neg(\tau \ltimes [x = v])$, meaning that τ must have a store in its potential whose value for x is not v, which would not be stable under LOSE).

Proposition 1. *If $\langle \gamma, \mathcal{D} \rangle \models \varphi$ and $\mathcal{D} \xrightarrow{\varepsilon}_{\mathsf{SRA}} \mathcal{D}'$, then $\langle \gamma, \mathcal{D}' \rangle \models \varphi$.*

Memory triples. Assertions in Piccolo describe sets of states, thus can be used to formulate memory triples. Figure 10 gives the base triples for the different primitive instructions.

We see the standard SC rule of assignment (SUBST-ASGN) for registers followed by a number of stability rules detailing when assertions are not affected by instructions. Axioms FORK and JOIN describe the transfer of properties from forking thread to forked threads and back.

---

[4] Negation just occurs on the level of simple expressions e which is sufficient for calculating $P \setminus [\![e]\!]$ required in rules IF and WHILE.

[5] Such stability requirements are also common to other reasoning techniques for weak memory models, e.g., [19].

The next four axioms in the table concern write instructions (either `SWAP` or `STORE`). They reflect the semantics of writing in SRA: (1) In the writer thread τ all stores in all lists get updated (axiom WR-OWN). Other threads π will have (2) their lists being split into "old" values for x with R flag and the new value for x (WR-OTHER-1), (3) properties (expressed as I_τ) of suffixes of lists being preserved when the writing thread satisfies the same properties (WR-OTHER-2) and (4) their lists consisting of R-accesses to x followed by properties of the writer (WR-OTHER-3). The last axiom concerns `SWAP` only: as it can only read from store entries marked as `RMW` it discards intervals satisfying $[R(x)]$.

Example 4. We employ the axioms for showing one proof step for MP, namely one pair in the non-interference check of the rely $\mathcal{R}_2$ of T_2 with respect to the guarantees $\mathcal{G}_1$ of T_1:

$$\{T_2 \ltimes [y \neq 1] \, ; [x = 1] \wedge T_1 \ltimes [x = 1]\} \; T_1 \mapsto \texttt{STORE}(x, 1) \; \{T_2 \ltimes [y \neq 1] \, ; [x = 1]\}$$

By taking I_τ to be $[x = 1]$, this is an instance of WR-OTHER-2.

In addition to the axioms above, we use a *shift* rule for load instructions:

$$\text{LD-SHIFT} \; \frac{\{\tau \ltimes I\} \; \tau \mapsto r := \texttt{LOAD}(x) \; \{\psi\}}{\{\tau \ltimes [(e \wedge E)(r := x)] \, ; I\} \; \tau \mapsto r := \texttt{LOAD}(x) \; \{(e \wedge \tau \ltimes [E]; I) \vee \psi\}}$$

A load instruction reads from the first store in the lists, however, if the list satisfying $[(e \wedge E)(r := x)]$ in $[(e \wedge E)(r := x)] \, ; I$ is empty, it reads from a list satisfying I. The shift rule for `LOAD` puts this shifting to next stores into a proof rule. Like the standard Hoare rule SUBST-ASGN, LD-SHIFT employs backward substitution.

Example 5. We exemplify rule LD-SHIFT on another proof step of example MP, one for local correctness of T_2:

$$\{T_2 \ltimes [y \neq 1] \, ; [x = 1]\} \; T_2 \mapsto a := \texttt{LOAD}(y) \; \{a = 1 \Rightarrow T_2 \ltimes [x = 1]\}$$

From axiom STABLE-LD we get $\{T_2 \ltimes [x = 1]\} \; T_2 \mapsto a := \texttt{LOAD}(y) \; \{T_2 \ltimes [x = 1]\}$. We obtain $\{T_2 \ltimes [y \neq 1] \, ; [x = 1]\} \; T_2 \mapsto a := \texttt{LOAD}(y) \; \{a \neq 1 \vee T_2 \ltimes [x = 1]\}$ using the former as premise for LD-SHIFT.

In addition, we include the standard conjunction, disjunction and consequence rules of Hoare logic. For instrumented primitive commands we employ the following rule:

$$\text{INSTR} \; \frac{\{\psi_0\} \; \tau \mapsto c \; \{\psi_1\} \quad \{\psi_1\} \; \tau \mapsto r_1 := e_1 \; \{\psi_2\} \ldots \{\psi_{n-1}\} \; \tau \mapsto r_n := e_n \; \{\psi_n\}}{\{\psi_0\} \; \tau \mapsto \langle c, \langle r_1, \ldots, r_n \rangle := \langle e_1, \ldots, e_n \rangle \rangle \; \{\psi_n\}}$$

Finally, it can be shown that all triples derivable from axioms and rules are valid memory triples.

Lemma 1. *If a* Piccolo *memory triple is derivable,* $\vdash_{\textsf{Piccolo}} \{\varphi\} \; \tau \mapsto \alpha \; \{\psi\}$, *then* $\textsf{SRA} \vDash \{\{\langle \gamma, \mathcal{D} \rangle \mid \langle \gamma, \mathcal{D} \rangle \vDash \varphi\}\} \; \tau \mapsto \alpha \; \{\{\langle \gamma, \mathcal{D} \rangle \mid \langle \gamma, \mathcal{D} \rangle \vDash \psi\}\}$.
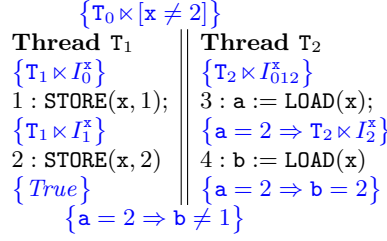
$$\{T_0 \ltimes [x \neq 2]\}$$

Thread T_1	**Thread** T_2
$\{T_1 \ltimes I_0^x\}$	$\{T_2 \ltimes I_{012}^x\}$
$1 : \text{STORE}(x, 1);$	$3 : a := \text{LOAD}(x);$
$\{T_1 \ltimes I_1^x\}$	$\{a = 2 \Rightarrow T_2 \ltimes I_2^x\}$
$2 : \text{STORE}(x, 2)$	$4 : b := \text{LOAD}(x)$
$\{True\}$	$\{a = 2 \Rightarrow b = 2\}$

$$\{a = 2 \Rightarrow b \neq 1\}$$

Fig. 11. RRC for two threads (a.k.a. CoRR0)

$$\{T_0 \ltimes I_0^x\}$$

Thread T_1	**Thread** T_2	**Thread** T_3	**Thread** T_4
$\{T_1 \ltimes I_{02}^x\}$	$\{T_1 \ltimes I_{01}^x\}$	$\{T_3 \ltimes (I_{012}^x \vee I_{021}^x)\}$	$\{T_4 \ltimes (I_{012}^x \vee I_{021}^x)\}$
$1 : \text{STORE}(x, 1)$	$2 : \text{STORE}(x, 2)$	$3 : a := \text{LOAD}(x);$	$5 : c := \text{LOAD}(x);$
$\{True\}$	$\{True\}$	$\{a = 2 \Rightarrow T_3 \ltimes I_{21}^x\}$	$\{c = 1 \Rightarrow T_4 \ltimes I_{12}^x\}$
		$4 : b := \text{LOAD}(x)$	$6 : d := \text{LOAD}(x)$
		$\{\langle a, b \rangle = \langle 2, 1 \rangle \Rightarrow T_3 \ltimes I_1^x\}$	$\left\{ \begin{array}{l} \langle c, d \rangle = \langle 1, 2 \rangle \Rightarrow \\ \qquad\qquad T_4 \ltimes I_2^x \end{array} \right\}$

$$\{\langle a, b \rangle = \langle 2, 1 \rangle \Rightarrow \langle c, d \rangle \neq \langle 1, 2 \rangle\}$$
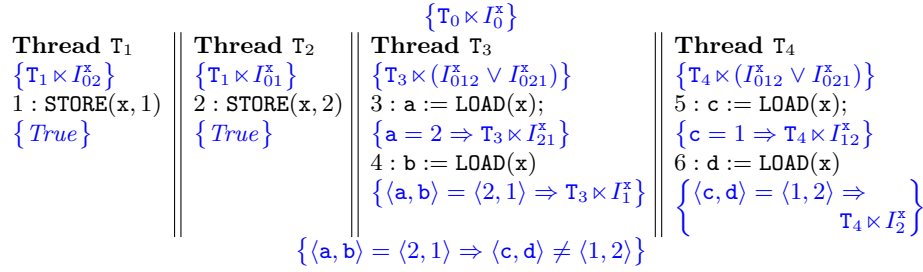
Fig. 12. RRC for four threads (a.k.a. CoRR2)

7 Examples

We discuss examples verified in Piccolo. Additional examples can be found in the extended version of this paper [30].

Coherence. We provide two coherence examples in Figures 11 and 12, using the notation $I_{v_1 v_2 \dots v_n}^x = [x = v_1]\,;[x = v_2]\,; \dots ;[x = v_n]$. Fig. 11 enforces an ordering on writes to the shared location x on thread T_1. The postcondition guarantees that after reading the second write, thread T_2 cannot read from the first. Fig. 12 is similar, but the writes to x occur on two different threads. The postcondition of the program guarantees that the two different threads agree on the order of the writes. In particular if one reading thread (here T_3) sees the value 2 then 1, it is impossible for the other reading thread (here T_4) to see 1 then 2.

Potential assertions provide a compact and intuitive mechanism for reasoning, e.g., in Fig. 11, the precondition of line 3 precisely expresses the order of values available to thread T_2. This presents an improvement over view-based assertions [16], which required a separate set of assertions to encode write order.

Peterson's algorithm. Figure 13 shows Peterson's algorithm for implementing mutual exclusion for two threads [38] together with Piccolo assertions. We depict only the code of thread T_1. Thread T_2 is symmetric. A third thread T_3 is assumed stopping the other two threads at an arbitrary point in time. We use do C until e as a shorthand for C ; while e do C. For correctness under

Thread T_1

$\{\neg a_1 \wedge \neg a_2 \wedge mx_1 = 0\}$

$\texttt{while } \neg\texttt{stop do} \quad \{\neg a_1 \wedge (\neg a_2 \vee T_1 \ltimes [R(\texttt{turn})] \,; [\texttt{flag}_2])\}$

$1 : \quad \texttt{STORE}(\texttt{flag}_1, true); \{\neg a_1 \wedge T_1 \ltimes [\texttt{flag}_1] \wedge (\neg a_2 \vee T_1 \ltimes [R(\texttt{turn})] \,; [\texttt{flag}_2])\}$

$2 : \quad \langle\texttt{SWAP}(\texttt{turn}, 2); a_1 := true\rangle;$

$3 : \quad \texttt{do } \{a_1 \wedge (\neg a_2 \vee T_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1] \vee P)\}$

$4 : \quad\quad \texttt{fl}_1 := \texttt{LOAD}(\texttt{flag}_2); \{a_1 \wedge (\neg a_2 \vee (\texttt{fl}_1 \wedge T_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1]) \vee P)\}$

$5 : \quad\quad \texttt{tu}_1 := \texttt{LOAD}(\texttt{turn}); \{a_1 \wedge (\neg a_2 \vee (\texttt{fl}_1 \wedge \texttt{tu}_1 \neq 1 \wedge T_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1]) \vee P)\}$

$6 : \quad \texttt{until } \neg\texttt{fl}_1 \vee (\texttt{tu}_1 = 1); \{a_1 \wedge (\neg a_2 \vee P)\}$

$7 : \quad \texttt{STORE}(\texttt{cs}, \bot); \{a_1 \wedge (\neg a_2 \vee P)\}$

$8 : \quad \texttt{STORE}(\texttt{cs}, 0); \{T_1 \ltimes [\texttt{cs} = 0] \wedge a_1 \wedge (\neg a_2 \vee P)\}$

$9 : \quad \texttt{mx}_1 := \texttt{LOAD}(\texttt{cs}); \{mx_1 = 0 \wedge a_1 \wedge (\neg a_2 \vee P)\}$

$10 : \quad \langle\texttt{STORE}(\texttt{flag}_1, 0); a_1 := false\rangle$

$\{mx_1 = 0\}$

Fig. 13. Peterson's algorithm, where $P = T_1 \ltimes [R(\texttt{turn})] \,; [\texttt{flag}_2 \wedge \texttt{turn} = 1]$. Thread T_2 is symmetric and we assume a stopper thread T_3 that sets $\texttt{stop}$ to $true$.

SRA, all accesses to the shared variable $\texttt{turn}$ are via a $\texttt{SWAP}$, which ensures that $\texttt{turn}$ behaves like an SC variable.

Correctness is encoded via registers mx_1 and mx_2 into which the contents of shared variable $\texttt{cs}$ is loaded. Mutual exclusion should guarantee both registers to be 0. Thus neither threads should ever be able to read $\texttt{cs}$ to be $\bot$ (as stored in line 7). The proof (like the associated SC proof in [9]) introduces auxiliary variables a_1 and a_2. Variable a_i is initially *false*, set to *true* when a thread T_i has performed its swap, and back to *false* when T_i completes.

Once again potentials provide convenient mechanisms for reasoning about the interactions between the two threads. For example, the assertion $T_1 \ltimes [R(\texttt{turn})] \,;$ $[\texttt{flag}_2]$ in the precondition of line 2 encapsulates the idea that an RMW on $\texttt{turn}$ (via $\texttt{SWAP}(\texttt{turn}, 2)$) must read from a state in which $\texttt{flag}_2$ holds, allowing us to establish $T_1 \ltimes [\texttt{flag}_2]$ as a postcondition (using the axiom SWAP-SKIP). We obtain disjunct $T_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1]$ after additionally applying WR-OWN.

8 Discussion, Related and Future Work

Previous RG-like logics provided ad-hoc solutions for other concrete memory models such as x86-TSO and C/C++11 [11,16,17,32,39,40,47]. These approaches established soundness of the proposed logic with an ad-hoc proof that couples together memory and thread transitions. We believe that these logics can be formulated in our proposed general RG framework (which will require extensions to other memory operations such as fences).

Moreover, Owicki-Gries logics for different fragments of the C11 memory model [16, 17, 47] used specialized assertions over the underlying view-based semantics. These include *conditional-view assertion* (enabling reasoning about MP), and *value-order* (enabling reasoning about coherence). Both types of assertions are special cases of the potential-based assertions of Piccolo.

Ridge [40] presents an RG reasoning technique tailored to x86-TSO, treating the write buffers in TSO architectures as threads whose steps have to preserve relies. This is similar to our notion of stability of relies under internal memory transitions. Ridge moreover allows to have memory-model specific assertions (e.g., on the contents of write buffers).

The OGRA logic [32] for Release-Acquire (which is slightly weaker form of causal consistency compared to SRA studied in this paper) takes a different approach, which cannot be directly handled in our framework. It employs simple SC-like assertions at the price of having a non-standard non-interference condition which require a stronger form of stability.

Coughlin et al. [14,15] provide an RG reasoning technique for weak memory models with a semantics defined in terms of *reordering relations* (on instructions). They study both multicopy and non-multicopy atomic architectures, but in all models, the rely-guarantee assertions are interpreted over SC.

Schellhorn et al. [41] develop a framework that extends ITL with a compositional interleaving operator, enabling proof decomposition using RG rules. Each interval represents a sequence of states, strictly alternating between program and environment actions (which may be a skip action). This work is radically different from ours since (1) their states are interpreted using a standard SC semantics, and (2) their intervals represent an *entire execution* of a command as well the interference from the environment while executing that command.

Under SC, rely-guarantee was combined with separation logic [44,46], which allows the powerful synergy of reasoning using stable invariants (as in rely-guarantee) and ownership transfer (as in concurrent separation logic). It is interesting to study a combination of our RG framework with concurrent separation logics for weak memory models, such as [43,45].

Other works have studied the decidability of verification for causal consistency models. In work preceding the potential-based SRA model [28], Abdulla et al. [1] show that verification under RA is undecidable. In other work, Abdulla et al. [3] show that the reachability problem under TSO remains decidable for systems with dynamic thread creation. Investigating this question under SRA is an interesting topic for future work.

Finally, the spirit of our generic approach is similar to Iris [22], Views [18], Ogre and Pythia [7], the work of Ponce de León et al. [34], and recent axiomatic characterizations of weak memory reasoning [19], which all aim to provide a *generic* framework that can be instantiated to underlying semantics.

In the future we are interested in automating the reasoning in Piccolo, starting from automatically checking for validity of program derivations (using, e.g., SMT solvers for specialised theories of sequences or strings [24,42]), and, including, more ambitiously, synthesizing appropriate Piccolo invariants.

References

1. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.N.: Verification of programs under the release-acquire semantics. In: PLDI. pp. 1117–1132. ACM (2019), https://doi.org/10.1145/3314221.3314649

2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Kumar, K.N., Saivasan, P.: Deciding reachability under persistent x86-tso. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021), `https://doi.org/10.1145/3434337`

3. Abdulla, P.A., Atig, M.F., Bouajjani, A., Kumar, K.N., Saivasan, P.: Verifying reachability for TSO programs with dynamic thread creation. In: NETYS. LNCS, vol. 13464, pp. 283–300. Springer (2022), `https://doi.org/10.1007/978-3-031-17436-0_19`

4. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: The benefits of duality in verifying concurrent programs under TSO. In: CONCUR. LIPIcs, vol. 59, pp. 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), `https://doi.org/10.4230/LIPIcs.CONCUR.2016.5`

5. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: A load-buffer semantics for total store ordering. Log. Methods Comput. Sci. **14**(1) (2018), `https://doi.org/10.23638/LMCS-14(1:9)2018`

6. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Comput. **9**(1), 37–49 (1995), `https://doi.org/10.1007/BF01784241`

7. 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`

8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014), `https://doi.org/10.1145/2627752`

9. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009), `https://doi.org/10.1007/978-1-84882-745-5`

10. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. Log. Methods Comput. Sci. **17**(1) (2021), `https://lmcs.episciences.org/7149`

11. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-Based Owicki-Gries Reasoning for Persistent x86-TSO. In: ESOP. LNCS, vol. 13240, pp. 234–261. Springer (2022), `https://doi.org/10.1007/978-3-030-99336-8_9`

12. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: POPL. pp. 626–638. ACM (2017), `https://doi.org/10.1145/3009837.3009888`

13. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991), `https://doi.org/10.1016/0020-0190(91)90122-X`

14. Coughlin, N., Winter, K., Smith, G.: Rely/guarantee reasoning for multicopy atomic weak memory models. In: FM. LNCS, vol. 13047, pp. 292–310. Springer (2021), `https://doi.org/10.1007/978-3-030-90870-6_16`

15. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. Form. Asp. Comput. (dec 2022), `https://doi.org/10.1145/3574137`

16. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries Reasoning for C11 RAR. In: 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`

17. 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`

18. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300. ACM (2013), https://doi.org/10.1145/2429069.2429104
19. 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
20. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: PPoPP. pp. 355–365. ACM (2019), https://doi.org/10.1145/3293883.3295702
21. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983), https://doi.org/10.1145/69575.69577
22. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018), https://doi.org/10.1017/S0956796818000151
23. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In: 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
24. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: CertiStr: a certified string solver. In: CPP. pp. 210–224. ACM (2022), https://doi.org/10.1145/3497775.3503691
25. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL. pp. 175–189. ACM (2017), https://doi.org/10.1145/3009837.3009850
26. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2), 43–56 (2019), https://doi.org/10.1145/3326938.3326942
27. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: PLDI. pp. 211–226. ACM (2020), https://doi.org/10.1145/3385412.3385966
28. Lahav, O., Boker, U.: What's Decidable About Causally Consistent Shared Memory? ACM Trans. Program. Lang. Syst. **44**(2), 8:1–8:55 (2022), https://doi.org/10.1145/3505273
29. Lahav, O., Dongol, B., Wehrheim, H.: Artifact: Rely-guarantee reasoning for causally consistent shared memory. Zenodo (2023), https://doi.org/10.5281/zenodo.7929646
30. Lahav, O., Dongol, B., Wehrheim, H.: Rely-guarantee reasoning for causally consistent shared memory (extended version) (2023), https://doi.org/10.48550/arXiv.2305.08486
31. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL. pp. 649–662. ACM (2016), https://doi.org/10.1145/2837614.2837643
32. Lahav, O., Vafeiadis, V.: Owicki-Gries Reasoning for Weak Memory Models. In: ICALP. LNCS, vol. 9135, pp. 311–323. Springer (2015), https://doi.org/10.1007/978-3-662-47666-6_25
33. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979), https://doi.org/10.1109/TC.1979.1675439
34. de León, H.P., Furbach, F., Heljanko, K., Meyer, R.: BMC with memory models as modules. In: FMCAD. pp. 1–9. IEEE (2018), https://doi.org/10.23919/FMCAD.2018.8603021

35. Moszkowski, B.C.: A complete axiom system for propositional interval temporal logic with infinite time. Log. Methods Comput. Sci. **8**(3) (2012), `https://doi.org/10.2168/LMCS-8(3:10)2012`

36. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: TPHOLs. LNCS, vol. 5674, pp. 391–407. Springer (2009), `https://doi.org/10.1007/978-3-642-03359-9_27`

37. Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica **6**, 319–340 (1976), `https://doi.org/10.1007/BF00268134`

38. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)

39. 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`

40. Ridge, T.: A Rely-Guarantee Proof System for x86-TSO. In: VSTTE. LNCS, vol. 6217, pp. 55–70. Springer (2010), `https://doi.org/10.1007/978-3-642-15057-9_4`

41. Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: A temporal logic framework for compositional reasoning about interleaved programs. Ann. Math. Artif. Intell. **71**(1-3), 131–174 (2014), `https://doi.org/10.1007/s10472-013-9389-z`

42. Sheng, Y., Nötzli, A., Reynolds, A., Zohar, Y., Dill, D.L., Grieskamp, W., Park, J., Qadeer, S., Barrett, C.W., Tinelli, C.: Reasoning about vectors using an SMT theory of sequences. In: IJCAR. LNCS, vol. 13385, pp. 125–143. Springer (2022), `https://doi.org/10.1007/978-3-031-10769-6_9`

43. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP. LNCS, vol. 10801, pp. 357–384. Springer (2018), `https://doi.org/10.1007/978-3-319-89884-1_13`

44. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge, UK (2008), `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221`

45. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA. pp. 867–884. ACM (2013), `https://doi.org/10.1145/2509136.2509532`

46. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. LNCS, vol. 4703, pp. 256–271. Springer (2007), `https://doi.org/10.1007/978-3-540-74407-8_18`

47. Wright, D., Batty, M., Dongol, B.: Owicki-Gries Reasoning for C11 Programs with Relaxed Dependencies. In: FM. LNCS, vol. 13047, pp. 237–254. Springer (2021), `https://doi.org/10.1007/978-3-030-90870-6_13`

48. Xu, Q., de Roever, W.P., He, J.: The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. Formal Aspects Comput. **9**(2), 149–174 (1997), `https://doi.org/10.1007/BF01211617`

A Auxiliary Variables and Rule of Consequence

Here, we provide the necessary definitions and the rule for auxiliary variables.

Definition 14. For a set $Z \subseteq \mathsf{Reg}$, two register stores γ and γ' are *Z-equivalent*, denoted by $\gamma =_Z \gamma'$, if $\gamma(r) = \gamma'(r)$ for every $r \in Z$. This equivalence is lifted to pairs of register stores and memory states by defining $\langle \gamma, m \rangle =_Z \langle \gamma', m' \rangle$ if $\gamma =_Z \gamma'$ and $m = m'$.

Definition 15. The removal of a set $Z \subseteq \mathsf{Reg}$ from a multi-assignment $\vec{r} := \vec{e}$, denoted by $\mathsf{rem}(\vec{r} := \vec{e}, Z)$, is the multi-assignment $\langle r_{j_1}, ... , r_{j_m} \rangle := \langle e_{j_1}, ... , e_{j_n} \rangle$ where $j_1 < ... < j_m$ is an enumeration of $\{1 \le j \le n \mid r_j \in Z\}$. This removal is only defined if no register in Z occurs in $\langle e_{j_1}, ... , e_{j_n} \rangle$. This notation is extended to commands in the expected way by applying removal inside the second component of instrumented primitive commands. It is only defined if all registers in Z only occur as instrumentation, i.e., in the second component of instrumented primitive commands.

Definition 16. A set $P \subseteq \Sigma$ of states is *independent of a set $Z \subseteq \mathsf{Reg}$*, denoted by $\mathsf{ind}(P, Z)$, if for every two states σ and $\sigma' \in \Sigma_{\mathcal{M}}$ with $\sigma =_{\mathsf{Reg} \setminus Z} \sigma'$, we have $\sigma \in P \iff \sigma' \in P$. This notion is lifted to guarded commands by defining $\mathsf{ind}(\{P\}\ \tau \mapsto c, Z) \overset{\triangle}{\iff} \mathsf{ind}(P, Z)$, and to sets of sets of states (or of guarded commands) by requiring that every element of the set is independent of Z.

The rule of auxiliary variables then takes the following form:

$\textsc{Aux}$
$$\frac{
\begin{array}{c}
\{\tau \mapsto C'\}\ \underline{sat}_{\mathcal{M}}\ (P', \mathcal{R}', \mathcal{G}, Q) \\
C = \mathsf{rem}(C', Z) \\
\mathsf{ind}(P, Z) \quad \mathsf{ind}(\mathcal{R}, Z) \quad \mathsf{ind}(\mathcal{G}, Z) \quad \mathsf{ind}(Q, Z) \\
\forall \sigma \in P.\, \exists \sigma' \in P'.\, \sigma =_{\mathsf{Reg} \setminus Z} \sigma' \\
\forall \sigma_1, \sigma_2.\, (\forall R \in \mathcal{R}.\, \sigma_1 \in R \Rightarrow \sigma_2 \in R) \Rightarrow \\
\forall \sigma_1'.\, \sigma_1 =_{\mathsf{Reg} \setminus Z} \sigma_1' \Rightarrow \exists \sigma_2'.\, \sigma_2 =_{\mathsf{Reg} \setminus Z} \sigma_2' \wedge (\forall R' \in \mathcal{R}'.\, \sigma_1' \in R' \Rightarrow \sigma_2' \in R')
\end{array}
}{
\{\tau \mapsto C\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)
}$$

The proof rules also contain a rule of consequence:

$\textsc{consq}$
$$\frac{
\begin{array}{c}
\{\tau \mapsto C\}\ \underline{sat}_{\mathcal{M}}\ (P', \mathcal{R}', \mathcal{G}', Q') \\
P \subseteq P' \quad \forall R' \in \mathcal{R}', \sigma \in R', \sigma' \in \Sigma.\, (\forall R \in \mathcal{R}.\, \sigma \in R \Rightarrow \sigma' \in R) \Rightarrow \sigma' \in R' \\
Q' \subseteq Q \quad \forall \{P'\}\ \pi \mapsto \tilde{c} \in \mathcal{G}'.\, \exists P.\, P' \subseteq P \wedge \{P\}\ \pi \mapsto \tilde{c} \in \mathcal{G}
\end{array}
}{
\{\tau \mapsto C\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)
}$$

B Soundness of RG Proof Rules

Theorem 1 (Soundness). $\vdash C\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q) \implies\ \vDash C\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)$.

Proof. By induction on the structure of rules. Next, we consider the rules in Fig. 7 plus the rule for Fork/Join and consequence. The rules for parallel composition and auxiliary variables are treated below.

Skip Let $\xi \in Comp(\{\tau \mapsto \texttt{skip}\}) \cap \mathsf{Assume}(P, \{P\})$. We need to prove that $\xi \in \mathsf{Commit}(\emptyset, P)$.

The computation ξ has mem and env transitions only. By (mem) and $\xi \in \mathsf{Assume}(P, \{P\})$, both type of transitions preserve P. Hence, in case ξ is finite, $\langle \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle \in P$ and furthermore $\mathcal{C}_{\mathsf{last}(\xi)}(\tau') = \texttt{skip}$ for every $\tau' \in dom(\mathcal{C}_{\mathsf{last}(\xi)})$ as $dom(\mathcal{C}_{\mathsf{last}(\xi)}) = \{\tau\}$. ξ also admits $\mathcal{G}$ as $\mathcal{G} = \emptyset$.

Instrumented primitive commands Let $\xi \in Comp(\{\tau \mapsto \tilde{c}\}) \cap \mathsf{Assume}(P, \{P, Q\})$.

We need to prove that $\xi \in \mathsf{Commit}(\{P\}\ \tau \mapsto \tilde{c}, Q)$.

By the operational semantics, ξ has to take the form

$$\langle \{\tau \mapsto \tilde{c}\}, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \{\tau \mapsto \tilde{c}\}, \gamma_1, m_1 \rangle \xrightarrow{a_2} ... \xrightarrow{a_j} \langle \{\tau \mapsto \texttt{skip}\}, \gamma_j, m_j \rangle ...$$

with $a_j = \texttt{cmp}$, $a_i \in \{\texttt{mem}, \texttt{env}\}$ for all $i, 1 \le i \le j - 1 \vee i \ge j + 1$, and for all $m \ge j + 1$ the command map is $\{\tau \mapsto \texttt{skip}\}$. By $\xi \in \mathsf{Assume}(P, \{P, Q\})$ and (mem), $\langle \gamma_0, m_0 \rangle$ and also $\langle \gamma_{j-1}, m_{j-1} \rangle \in P$. By $\mathcal{M} \vDash \{P\}\ \tau \mapsto \tilde{c}\ \{Q\}$, $\langle \gamma_j, m_j \rangle \in Q$. By $\xi \in \mathsf{Assume}(P, \{P, Q\})$, $\langle \gamma_m, m_m \rangle \in Q$ for all $m \ge j$. Hence, if ξ is finite, $\langle \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle \in Q$ and $\mathcal{C}_{\mathsf{last}(\xi)}(\tau) = \texttt{skip}$ for every $\tau \in dom(\mathcal{C}_{\mathsf{last}(\xi)})$. Finally, ξ admits $\mathcal{G}$ as a_j is the only component step executing c in a state satisfying P.

Sequential composition Let $\xi \in Comp(\{\tau \mapsto C_1 \, ; C_2\}) \cap \mathsf{Assume}(P, \mathcal{R}_1 \cup \mathcal{R}_2)$. We need to prove that $\xi \in \mathsf{Commit}(\mathcal{G}_1 \cup \mathcal{G}_2, Q)$. The computation $\xi = \langle \mathcal{C}_0, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \mathcal{C}_1, \gamma_1, m_1 \rangle \xrightarrow{a_2} ...$ takes one of two forms:

(1) Either (the C_1 part is infinite) $\mathcal{C}_i = \{\tau \mapsto C_1^i \, ; C_2\}$, $i \ge 0$, and $\xi' = \langle \{\tau \mapsto C_1\}, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \{\tau \mapsto C_1^1\}, \gamma_1, m_1 \rangle \xrightarrow{a_2} ...$ is in $Comp(\{\tau \mapsto C_1\}) \cap \mathsf{Assume}(P, \mathcal{R}_1)$ (as $\mathcal{R}_1 \subseteq \mathcal{R}_1 \cup \mathcal{R}_2$),

or (2) (C_1 part finite, C_2 part finite or infinite) there exists k s.t. $\mathcal{C}_k = \{\tau \mapsto \texttt{skip} \, ; C_2\}$, $\mathcal{C}_{k+1} = \{\tau \mapsto C_2\}$ and $a_{k+1} = \varepsilon$.

First case: By $\{\tau \mapsto C_1\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}_1, \mathcal{G}_1, mid)$, $\xi' \in \mathsf{Commit}(\mathcal{G}_1, mid)$. Hence $\xi' \in \mathsf{Commit}(\mathcal{G}_1 \cup \mathcal{G}_2, mid)$, and as ξ' and ξ are infinite, $\xi \in \mathsf{Commit}(\mathcal{G}_1 \cup \mathcal{G}_2, Q)$ (no final state).

Second case: We split ξ into ξ_1 and ξ_2 by letting ξ_1 run until state k, $\langle \{\tau \mapsto \texttt{skip} \, ; C_2\}, \gamma_k, m_k \rangle$, and have ξ_2 start in state $k+1$, $\langle \{\tau \mapsto C_2\}, \gamma_{k+1}, m_{k+1} \rangle$. From ξ_1 we construct computation ξ_1' like for case (1). Again, $\xi_1' \in Comp(\{\tau \mapsto C_1\}) \cap \mathsf{Assume}(P, \mathcal{R}_1)$. As ξ_1' is finite and by $\{\tau \mapsto C_1\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}_1, \mathcal{G}_1, mid)$, $\xi_1' \in \mathsf{Commit}(\mathcal{G}_1, mid)$, hence $\langle \gamma_k, m_k \rangle \in mid$ and so is $\langle \gamma_{k+1}, m_{k+1} \rangle$ (as the ε-step of $\texttt{skip}$ does not change registers nor memory). Furthermore, all component steps in ξ_1' satisfy $\mathcal{G}_1$ (and thus $\mathcal{G}_1 \cup \mathcal{G}_2$) and so do the same steps in ξ. Then, $\xi_2 \in Comp(\{\tau \mapsto C_2\}) \cap \mathsf{Assume}(mid, \mathcal{R}_2)$, hence by $\{\tau \mapsto C_2\}\ \underline{sat}_{\mathcal{M}}\ (mid, \mathcal{R}_2, \mathcal{G}_2, Q)$, $\xi_2 \in \mathsf{Commit}(\mathcal{G}_2, Q)$. Thus, if ξ_2 is finite, $\langle \gamma_{\mathsf{last}(\xi_2)}, m_{\mathsf{last}(\xi_2)} \rangle \in Q$ and $\mathcal{C}_{\mathsf{last}(\xi)}(\tau) = \texttt{skip}$. Hence $\xi \in \mathsf{Commit}(\mathcal{G}_1 \cup \mathcal{G}_2, Q)$.

If Let $\xi \in Comp(\{\tau \mapsto \texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2\}) \cap \mathsf{Assume}(P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P\})$. Need to show $\xi \in \mathsf{Commit}(\mathcal{G}_1 \cup \mathcal{G}_2, Q)$. Then there exists a k s.t. $\mathcal{C}_k = \{\tau \mapsto \texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2\}$, $a_{k+1} = \varepsilon$ and for all $i \le k$, step $a_i \ne \texttt{cmp}$,

25

hence $\langle \gamma_k, m_k \rangle \in P$ (as P is in the relies). Then two cases: (1) $\gamma_k \in [\![e]\!]$ or is not. The cases are dual and we just consider the first one.

In that case, $\mathcal{C}_{k+1} = \{\tau \mapsto C_1\}$. Furthermore, $\langle \gamma_{k+1}, m_{k+1} \rangle \in P \cap [\![e]\!] \times \mathcal{M}.\mathtt{Q}$ (the ε-step of if neither changing registers nor memory). All further component steps of ξ now satisfy $\mathcal{G}_1$ (by $\{\tau \mapsto C_1\}$ $\underline{sat}_{\mathcal{M}}$ $(P \cap ([\![e]\!] \times \mathcal{M}.\mathtt{Q}), \mathcal{R}_1, \mathcal{G}_1, Q)$. Hence, if ξ is finite, $\langle \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle \in Q$ and $\mathcal{C}_{\mathsf{last}(\xi)}(\tau) = \mathtt{skip}$ and thus $\xi \in \mathsf{Commit}(\mathcal{G}_1, Q)$ and hence in $\mathsf{Commit}(\mathcal{G}_1 \cup \mathcal{G}_2, Q)$.

While Let $\xi \in \{\tau \mapsto \mathtt{while}\ e\ \mathtt{do}\ C\} \cap \mathsf{Assume}(P, \mathcal{R} \cup \{P, Q\})$. We need to show $\xi \in \mathsf{Commit}(\mathcal{G}, Q)$.

We let $\mathcal{C}_w = \mathtt{while}\ e\ \mathtt{do}\ C$, $\mathcal{C}_{if} = \mathtt{if}\ e\ \mathtt{then}\ (C\ ;\ \mathtt{while}\ e\ \mathtt{do}\ C)\ \mathtt{else}\ \mathtt{skip}$, $\mathcal{C}_s = \{\tau \mapsto C\ ;\ \mathtt{while}\ e\ \mathtt{do}\ C\}$ and $\mathcal{C}_{end} = \{\tau \mapsto \mathtt{skip}\ ;\ \mathtt{while}\ e\ \mathtt{do}\ C\}$. By the operational semantics, the computation ξ takes the following form or a prefix of it followed by non-component steps:

$$\langle \mathcal{C}_w, \gamma_0, m_0 \rangle \to \ldots \to \langle \mathcal{C}_w, \gamma_{k_1}, m_{k_1} \rangle \xrightarrow{\mathtt{cmp}} \langle \mathcal{C}_{if}, \gamma_{k_1+1}, m_{k_1+1} \rangle \to \ldots \to$$
$$\langle \mathcal{C}_{if}, \gamma_{m_1}, m_{m_1} \rangle \xrightarrow{\mathtt{cmp}} \langle \mathcal{C}_s, \gamma_{m_1+1}, m_{m_1+1} \rangle \to \ldots \to \langle \mathcal{C}_{end}, \gamma_{k_2-1}, m_{k_2-1} \rangle$$
$$\xrightarrow{\mathtt{cmp}} \langle \mathcal{C}_w, \gamma_{k_2}, m_{k_2} \rangle \to \ldots \to \langle \mathcal{C}_{if}, \gamma_{m_2}, m_{m_2} \rangle \xrightarrow{\mathtt{cmp}} \langle \mathcal{C}_s, \gamma_{m_2+1}, m_{m_2+1} \rangle$$
$$\to \ldots \to \langle \mathcal{C}_w, \gamma_{k_r}, m_{k_r} \rangle \xrightarrow{\mathtt{cmp}} \langle \mathcal{C}_{if}, \gamma_{k_r+1}, m_{k_r+1} \rangle \to \ldots \to$$

which goes on like this forever, or eventually reaches $\langle \mathcal{C}_{if}, \gamma_{m_l}, m_{m_l} \rangle$ with $\langle \gamma_{m_l}, m_{m_l} \rangle \notin [\![e]\!] \times \mathcal{M}.\mathtt{Q}$ and $\mathcal{C}_{m_l+1} = \{\tau \mapsto \mathtt{skip}\}$. Afterwards, the computation either has $\mathtt{env}$ or $\mathtt{mem}$ steps forever, or ξ is final. By the operational semantics and $\{\tau \mapsto C\}$ $\underline{sat}_{\mathcal{M}}$ $(P \cap ([\![e]\!] \times \mathcal{M}.\mathtt{Q}), \mathcal{R}, \mathcal{G}, P)$ we get the following properties:

$$\forall i, 1 \leq i < l: \quad \gamma_{m_i}(e) = 1$$
$$\forall i, 1 \leq i < l: \quad \langle \gamma_{k_i}, m_{k_i} \rangle \in P \cap ([\![e]\!] \times \mathcal{M}.\mathtt{Q})$$
$$\langle \gamma_{m_l}, m_{m_l} \rangle \in P \setminus ([\![e]\!] \times \mathcal{M}.\mathtt{Q})$$

All components steps of ξ are in $\mathcal{G}$, either because they are steps of C or steps belonging to component steps labelled ε which correspond to unfolding the while, evaluating the condition in if or moving from one part of a sequential composition to the next). By $P \setminus ([\![e]\!] \times \mathcal{M}.\mathtt{Q}) \subseteq post$, $\langle \gamma_{m_l}, m_{m_l} \rangle \in Q$. Furthermore, by $\xi \in \mathsf{Assume}(P, \mathcal{R} \cup \{P, Q\})$, $\langle \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle \in Q$ and $\mathcal{C}_{\mathsf{last}(\xi)}(\tau) = \mathtt{skip}$.

Consequence Let $\xi \in Comp(\{\tau \mapsto C\}) \cap \mathsf{Assume}(P, \mathcal{R} \cup \{P, Q\})$. We need to show $\xi \in \mathsf{Commit}(\mathcal{G}, Q)$.

As $P \subseteq P'$, $\xi \in \mathsf{Assume}(P', \mathcal{R} \cup \{P, Q\})$.

From ξ admitting $\mathcal{R}$, we get $\forall R \in \mathcal{R}, \forall i \geq 0$ with $a_{i+1} = \mathtt{env}$: $\langle \gamma_i, m_i \rangle \in R \Rightarrow \langle \gamma_{i+1}, m_{i+1} \rangle \in R$. Now let $R' \in \mathcal{R}'$ and assume $\langle \gamma_i, m_i \rangle \in R'$. By the condition on relies in the rule, we then get $\langle \gamma_{i+1}, m_{i+1} \rangle \in R'$. Hence $\xi \in \mathsf{Assume}(P, \mathcal{R}')$.

By $\{\tau \mapsto C\}$ $\underline{sat}_{\mathcal{M}}$ $(P', \mathcal{R}', \mathcal{G}', Q')$, we get $\xi \in \mathsf{Commit}(\mathcal{G}', Q')$. Thus for all $i \geq 0$ and $a_{i+1} = \mathtt{cmp}$, there exists $\{P'\}$ $\tau \mapsto \tilde{c} \in \mathcal{G}'$ such that $\langle \gamma_i, m_i \rangle \in P'$ and $\langle \{\tau \mapsto \tilde{c}\}, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon}_{\overline{\mathcal{M}}} \langle \{\tau \mapsto \mathtt{skip}\}, \gamma_{i+1}, m_{i+1} \rangle$ for some $l_\varepsilon \in$

Lab $\cup \{\varepsilon\}$ (and similar for Fork/Join steps). By the condition on guarantees there is thus some $\{P\}\ \tau \mapsto \tilde{c} \in \mathcal{G}$ such that $\langle \gamma_i, m_i \rangle \in P$ and $\langle \{\tau \mapsto \tilde{c}\}, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon}_{\overline{\mathcal{M}}} \langle \{\tau \mapsto \mathtt{skip}\}, \gamma_{i+1}, m_{i+1} \rangle$ for some $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$. Thus ξ admits $\mathcal{G}$. The rest follows from $Q' \subseteq Q$.

Fork/Join Let $\xi \in Comp(\{\tau \mapsto C_1 \ {}^{\tau_1}||^{\tau_2}\ C_2\}) \cap \mathsf{Assume}(P, \mathcal{R} \cup \{P, Q\})$. We need to show that $\xi \in \mathsf{Commit}(\mathcal{G}, Q)$. For the computation ξ, assume there is an index k such that $\mathcal{C}_k = \{\tau \mapsto C_1 \ {}^{\tau_1}||^{\tau_2}\ C_2\}$, $\mathcal{C}_{k+1} = \{\tau \mapsto C_1 \ {}^{\tau_1}||^{\tau_2}\ C_2\} \uplus \{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\}$ and the step a_{k+1} belongs to the component transition $\mathtt{FORK}(\tau_1, \tau_2)$. Moreover, all steps prior to $k+1$ are non-$\mathtt{cmp}$ steps. Then $\langle \gamma_k, m_k \rangle \in P$ (by P being contained in the rely) and by

$$\mathcal{M} \vDash \{P\}\ \tau \mapsto \mathtt{FORK}(\tau_1, \tau_2)\ \{P'\}$$

we get $\langle \gamma_{k+1}, m_{k+1} \rangle \in P'$. Now two cases: (1) either there is some $m > k$ such that $\mathcal{C}_m = \{\tau \mapsto C_1 \ {}^{\tau_1}||^{\tau_2}\ \} \uplus \{\tau_1 \mapsto \mathtt{skip}\} \uplus \{\tau_2 \mapsto \mathtt{skip}\}$ and the step $a_{m+1} = \mathtt{cmp}$, namely that belonging to $\mathtt{JOIN}(\tau_1, \tau_2)$, and then $\mathcal{C}_{m+1} = \{\tau \mapsto \mathtt{skip}\}$, or (2) no such step exists and thus ξ is infinite.

For case (1), $\langle \gamma_m, m_m \rangle \in Q'$ and hence by $\mathcal{M} \vDash \{Q'\}\ \tau \mapsto \mathtt{JOIN}(\tau_1, \tau_2)\ \{Q\}$, we get $\langle \gamma_{m+1}, m_{m+1} \rangle \in Q$ (and so are all further states because Q is in the rely). In both cases, all component steps fulfill $\mathcal{G}$ because they are either component steps of C_1 or C_2 or fork or join. Hence, $\xi \in \mathsf{Commit}(\mathcal{G} \cup \{\tau \mapsto \mathtt{FORK}(\tau_1, \tau_2), \tau \mapsto \mathtt{JOIN}(\tau_1, \tau_2)\}, Q)$.

Soundness of parallel composition. For parallel composition, we take a computation ξ and make a projection onto the threads τ_1 and τ_2 (plus their forked threads). We let $T_i \subseteq \mathsf{Tid}$ contain τ_i, $i = 1, 2$, plus threads (recursively) forked by it. We construct computations ξ_1, ξ_2 by replacing in $\xi = \langle \mathcal{C}_0, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \mathcal{C}_1, \gamma_1, m_1 \rangle \xrightarrow{a_2} \ldots$ the command maps $\mathcal{C}_k$ by $\mathcal{C}_k|_{T_i}$, $i = 1, 2$ (making $dom(\mathcal{C}_k) = T_i$). All $\mathtt{cmp}$-steps of T_2 become $\mathtt{env}$-steps in ξ_1, and vice versa, other steps stay as they are. We get

$$\xi_i \in Comp(\{\tau_i \mapsto C_i\}), i = 1, 2$$

Then the following lemma is required for later showing soundness of the rule of parallel composition (to avoid circular reasoning). It is analogous to one of Xu et al. [48].

Lemma 2. *Let* $\xi \in Comp(\{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\}) \cap \mathsf{Assume}(P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P, Q\})$ *and* ξ_1, ξ_2 *be the projections onto threads* τ_1, τ_2, *respectively. Assume furthermore* $\{\tau_i \mapsto C_i\}\ \underline{sat}_{\mathcal{M}}\ (P_i, \mathcal{R}_i, \mathcal{G}_i, Q_i)$, $(P_1, \mathcal{R}_1, \mathcal{G}_1, Q_1)$ *and* $(P_2, \mathcal{R}_2, \mathcal{G}_2, Q_2)$ *to be non-interfering and* $P \subseteq P_1 \cap P_2$. *Then* ξ_1 *admits* $\mathcal{G}_1$ *and* ξ_2 *admits* $\mathcal{G}_2$.

Proof. First of all, ξ_i admits P_i by $P \subseteq P_1 \cap P_2$. Now proof by contradiction. Assume j to be the smallest index such that for the transition $\xrightarrow{a_{j+1}=\mathtt{cmp}}$ in some ξ_i there is no $\{p\}\ \tau_i \mapsto \tilde{c} \in \mathcal{G}_i$ such that $\langle \gamma_j, m_j \rangle \in p$ and $\langle \gamma_j, m_j \rangle \xrightarrow{c} \langle \gamma_{j+1}, m_{j+1} \rangle$. Without loss of generality assume this to be thread $i = 1$. We now consider the computation ξ_1^{j+1} which is ξ_1 up to $\langle \mathcal{C}_1^{j+1}, \gamma_{j+1}, m_{j+1} \rangle$. $\xi_1^{j+1} \in$

$Comp(\{\tau_1 \mapsto C_1\})$. We now consider the env-steps in ξ_1^{j+1}. These are either env-steps in ξ or cmp-steps in ξ_2. In the first case, they preserve all $R \in \mathcal{R}_1 \cup \mathcal{R}_2$. In the latter case, there exists some $\{p'\}\ \tau_2 \mapsto \tilde{c}' \in \mathcal{G}_2$ which justifies the step (if not, we would have found a smaller index). By non-interference, this step preserves all $R \in \mathcal{R}_1$. Hence, $\xi_1^{j+1} \in \mathsf{Assume}(P_1, \mathcal{R}_1)$. By $\{\tau_1 \mapsto C_1\}\ \underline{sat}_\mathcal{M}\ (P_1, \mathcal{R}_1, \mathcal{G}_1, Q_1)$, we thus get $\xi_1^{j+1} \in \mathsf{Commit}(\mathcal{G}_1, Q_1)$ which gives the contradiction.

With this lemma at hand, we can show soundness of the rule for parallel composition.

Proof. Let $\xi \in Comp(\{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\}) \cap \mathsf{Assume}(P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P, Q\})$. By Lemma 2 and its proof, we get $\xi_i \in \mathsf{Assume}(P_i, \mathcal{R}_i)$ for the two projections ξ_1 and ξ_2 of ξ. Hence, $\xi_i \in \mathsf{Commit}(\mathcal{G}_i, Q_i), i = 1, 2$. Furthermore, if ξ is finite, so are ξ_1 and ξ_2. Then $\langle \gamma_{\mathsf{last}(\xi_i)}, m_{\mathsf{last}(\xi)} \rangle \in Q_i$ and $\mathcal{C}_{\mathsf{last}(\xi_i)}(\tau_i) = \mathtt{skip},\ i = 1, 2$. By $Q_1 \cap Q_2 \subseteq Q$, this final state also satisfies Q.

Soundness of auxiliary variables. We start by showing a proposition about transitions to not be affected by auxiliary variables.

Proposition 2. *Let $Z \subseteq \mathsf{Reg}$ be a set of auxiliary registers, γ_1, γ_2 register stores with $\gamma_1 =_{\mathsf{Reg} \setminus Z} \gamma_2$ and let $\langle \{\tau \mapsto \mathsf{rem}(\tilde{c}, Z)\}, \gamma_1, m \rangle \xrightarrow{\tau, l} \langle \{\tau \mapsto \mathtt{skip}\}, \gamma_1', m' \rangle$. Then for all γ_2' with $\langle \{\tau \mapsto \tilde{c}\}, \gamma_2, m \rangle \xrightarrow{\tau, l} \langle \{\tau \mapsto \mathtt{skip}\}, \gamma_2', m' \rangle$ we have $\gamma_1' =_{\mathsf{Reg} \setminus Z} \gamma_2'$.*

Proof. Proof of the soundness of the rule of auxiliary variables.
Let $\xi \in Comp(\{\tau \mapsto C\}) \cap \mathsf{Assume}(P, \mathcal{R})$. We need to show that $\xi \in \mathsf{Commit}(\mathcal{G}, Q)$. Assume ξ to be

$$\langle \{\tau \mapsto C_0\}, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \{\tau \mapsto C_1\}, \gamma_1, m_1 \rangle \xrightarrow{a_2} \dots \to \dots$$

with $C_0 = C$. Let $Z \subseteq \mathsf{Reg}$ be a set of auxiliary registers and C' be an arbitrary command with $\mathsf{rem}(C', Z) = C$ and $\{\tau \mapsto C'\}\ \underline{sat}_\mathcal{M}\ (P', \mathcal{R}', \mathcal{G}, Q)$. By the lifting of Proposition 2 to arbitrary commands, there exists a computation $\xi' \in Comp(\{\tau \mapsto C'\})$ with

$$\langle \{\tau \mapsto C_0'\}, \gamma_0', m_0 \rangle \xrightarrow{a_1} \langle \{\tau \mapsto C_1'\}, \gamma_1', m_1 \rangle \xrightarrow{a_2} \dots \to \dots$$

such that $C_0' = C'$, $\mathsf{rem}(C_i', Z) = C_i$ for all $i \geq 0$ and $\gamma_i =_{\mathsf{Reg} \setminus Z} \gamma_i'$ for all $i \geq 0$. By $\mathsf{ind}(P, Z)$, $\gamma_0' \in P$. We choose γ_0' to be a register store satisfying P' (which exists by condition $\forall \sigma \in P. \exists \sigma' \in P'. \sigma =_{\mathsf{Reg} \setminus Z} \sigma'$). For the environment steps see below. We get $\xi' \in \mathsf{Assume}(P', \mathcal{R})$ (using $\mathsf{ind}(\mathcal{R}, Z)$).
By ξ' admitting $\mathcal{R}$, we know that for all $R \in \mathcal{R}$, $\langle \gamma_i, m_i \rangle \xrightarrow{a_{i+1}} \langle \gamma_{i+1}, m_{i+1} \rangle$ with $a_{i+1} = \mathtt{env}$, the condition $\langle \gamma_i, m_i \rangle \in R \Rightarrow \langle \gamma_{i+1}, m_{i+1} \rangle \in R$ holds. Now take arbitrary $R' \in \mathcal{R}'$ and let $\langle \gamma_i', m_i \rangle \in R'$. By the condition on relies, we know the existence of $\langle \gamma_2, m_{i+1} \rangle$ with $\langle \gamma_{i+1}, m_{i+1} \rangle =_{\mathsf{Reg} \setminus Z} \langle \gamma_2, m_{i+1} \rangle$ and $\langle \gamma_2, m_{i+1} \rangle \in R'$. We take this to be $\langle \gamma_{i+1}', m_{i+1} \rangle$. Hence $\xi' \in \mathsf{Assume}(P', \mathcal{R}')$. By $\{\tau \mapsto C'\}\ \underline{sat}_\mathcal{M}\ (P', \mathcal{R}', \mathcal{G}, Q)$ we get $\xi' \in \mathsf{Commit}(\mathcal{G}, Q)$. By $\mathsf{ind}(\mathcal{G}, Z)$ and $\mathsf{ind}(Q, Z)$ we get $\xi \in \mathsf{Commit}(\mathcal{G}, Q)$.

C More Examples

C.1 Load-buffering (LB)

We present a proof outline for LB.

$$\{a = 0 \land b = 0\}$$

Thread 1	**Thread** 2
$\{T_2 \ltimes [y = 0] \land b = 0\}$	$\{T_1 \ltimes [x = 0] \land a = 0\}$
$1 : a := \texttt{LOAD(x)};$	$3 : b := \texttt{LOAD(y)};$
$\{T_2 \ltimes [y = 0] \land b = 0\}$	$\{T_1 \ltimes [x = 0] \land a = 0\}$
$2 : \texttt{STORE(y, 1)};$	$4 : \texttt{STORE(x, 1)};$
$\{a = 0 \lor b = 0\}$	$\{a = 0 \lor b = 0\}$

$$\{a = 0 \lor b = 0\}$$

C.2 Two writers on two locations (2+2W)

We present a proof outline for 2+2W.

$$\{a = b = \hat{c} = 0\}$$

Thread T_1
$\{a = 0\}$
$1 : \texttt{STORE(x, 1)};$
$\{True\}$
$2 : \texttt{STORE(y, 2)};$
$\left\{ T_1 \ltimes [y = 2] \lor \left(\begin{array}{l}(\hat{c} = 1 \Rightarrow T_2 \ltimes [x = 2]) \land \\ (b \neq 0 \Rightarrow b = 2)\end{array} \right) \right\}$
$3 : a := \texttt{LOAD(y)};$
$\{True\}$

Thread T_2
$\{b = \hat{c} = 0\}$
$4 : \texttt{STORE(y, 1)};$
$\{True\}$
$5 : \langle \texttt{STORE(x, 2)}, \hat{c} := 1 \rangle;$
$\{T_2 \ltimes [x \neq 0] \land (a = 1 \Rightarrow T_2 \ltimes [x = 2]) \land \hat{c} = 1\}$
$6 : b := \texttt{LOAD(x)};$
$\{b \neq 0 \land (a = 1 \Rightarrow b = 2)\}$

$$\{a = 1 \Rightarrow b = 2\}$$

C.3 Store-buffering with fences (SB)

SC-fences are modeled as RMWs to an otherwise unused location f. The auxiliary variable $\hat{c}$ is used to remember the order of the fence instructions.

$$\{\hat{c} = 0\}$$

Thread T_1
$\{\hat{c} = 0 \lor (\hat{c} = 2 \land T_1 \ltimes [R(f)] \, ; [y = 1])\}$
$1 : \texttt{STORE(x, 1)};$
$\left\{ \begin{array}{l} T_1 \ltimes [x = 1] \land \\ (\hat{c} = 0 \lor (\hat{c} = 2 \land T_1 \ltimes [R(f)] \, ; [y = 1])) \end{array} \right\}$
$2 : \langle \texttt{SWAP(f, 0)}, \hat{c} := 10\hat{c} + 1 \rangle;$
$\{\hat{c} = 1 \lor \hat{c} = 12 \lor (\hat{c} = 21 \land T_1 \ltimes [y = 1])\}$
$3 : a := \texttt{LOAD(x)};$
$\{\hat{c} = 1 \lor \hat{c} = 12 \lor (\hat{c} = 21 \land b = 1)\}$

Thread T_2
$\{\hat{c} = 0 \lor (\hat{c} = 1 \land T_2 \ltimes [R(f)] \, ; [x = 1])\}$
$4 : \texttt{STORE(y, 1)};$
$\left\{ \begin{array}{l} T_2 \ltimes [y = 1] \land \\ (\hat{c} = 0 \lor (\hat{c} = 1 \land T_2 \ltimes [R(f)] \, ; [x = 1])) \end{array} \right\}$
$5 : \langle \texttt{SWAP(f, 0)}, \hat{c} := 10\hat{c} + 2 \rangle;$
$\{\hat{c} = 2 \lor \hat{c} = 21 \lor (\hat{c} = 12 \land T_2 \ltimes [x = 1])\}$
$6 : b := \texttt{LOAD(y)};$
$\{\hat{c} = 2 \lor \hat{c} = 21 \lor (\hat{c} = 12 \land a = 1)\}$

$$\{\hat{a} = 1 \lor \hat{b} = 1\}$$