




A Rely-Guarantee Framework for Proving Deadlock Freedom under Causal Consistency^{*}

Brijesh Dongol¹, Ori Lahav², and Heike Wehrheim³

¹ University of Surrey, Guildford, UK

² Tel Aviv University, Tel Aviv, Israel

³ Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany

Abstract. Jones’ rely-guarantee framework (originally developed to enable reasoning about partial correctness) has been extended in several works to additionally enable reasoning about deadlock freedom. However, these frameworks were originally developed for the strong memory model known as sequential consistency (SC). Under SC, all threads are assumed to read from the most recent write to each shared location, which is too strong for most modern multithreaded systems. In recent work, we have shown that rules for rely-guarantee can be adapted to cope with weaker causally consistent memory models (e.g., strong release-acquire), while preserving most of its core rules. This framework is modular and can be instantiated to different memory models. The only adaptation necessary is at the level of atomic statements (reads and writes), which require introduction of new assertions and associated proof rules that must be proved sound with respect to the operational semantics of the memory model at hand. In this paper, we show that it is possible to also further extend the framework to cope with deadlock freedom under causal consistency, taking inspiration from the aforementioned extensions to rely-guarantee reasoning for SC. We exemplify our technique on a short but challenging protocol for synchronising initialisation.

1 Introduction

Jones’ *Rely-Guarantee* (RG) framework [14] is a seminal technique for compositional proofs of shared-variable concurrent programs. It allows one to express the semantics of a program using standard *pre-* and *postconditions* as well as a *rely* condition (describing the assumptions about the program’s environment) and a *guarantee* condition (which abstractly characterises the behaviour of the program). The rely and guarantee conditions enable one to *compose* parallel

^{*} Dongol is supported by EPSRC grants EP/Y036425/1, EP/X037142/1, EP/X015149/1, EP/V038915/1, EP/R025134/2 and VeTSS. Lahav is supported by the Israel Science Foundation (grant 814/22) and by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement no. 851811). Wehrheim is supported by the German Research Council DFG (project no. 467386514).

components, replacing a global interference freedom check, as required by the Owicki-Gries method [24], with rely/guarantee checks.

RG has been extensively studied over the years, e.g., through the lens of generalised abstract frameworks [8] and program algebras [22]; localised reasoning techniques [11, 31]; applications to specific weak memory models [4, 5, 18, 26]; extensions with temporal logic specifications [12], etc. We study rely-guarantee in the context of causally consistent memory models, which covers a large class of weak memory models including SC, TSO, and fragments of C11. While some works [4, 5, 26] have considered the use of *standard* (i.e., sequentially consistent) rely-guarantee frameworks to capture the reorderings allowed by weak memory, our point of departure is the parametric encoding in [18]. This encoding shows that it is possible to reuse the generic decomposition rules of rely-guarantee [33] unchanged across different memory models. This allows a verifier to focus on assertions and proof rules for atomic commands such as reads and writes, which comprises the only model-specific aspect of the framework.

The main contribution of this paper is a further extension of the parametric RG framework [18] with additional techniques for reasoning about deadlock freedom. In particular, like Xu et al. [33], we assume that a program is specified by a 5-tuple of the form $(P, \mathcal{R}, U, \mathcal{G}, Q)$, where P and Q are the pre and postconditions, respectively, \mathcal{R} and \mathcal{G} are the rely and guarantee conditions, respectively, and U is a so called *run* predicate used to ensure that the component in question is able to take a step. This notion refines the per-component *wait* condition used to ensure absence of deadlock developed by Stølen [30].⁴

The main mechanisms of our reasoning framework borrow from Xu et al. [33], but we enable reasoning generically about (weak) memory models. As such, our run predicate characterises deadlock in terms of the actions that the program can take, *as well as* the actions of the underlying memory system. We also develop a simpler rule for parallel composition, which reduces the complexity of composing run predicates. We exemplify these techniques for the *strong release-acquire* (SRA) memory model [16–18], though stress that it is possible to use the framework to reason in other memory models too.

Overview. This paper is organised as follows. In §2, we motivate the approach by presenting two simple examples executing on the SC and SRA memory models. §3 presents the generalised syntax and semantics of our approach, which is parametric over a memory model, recapping our earlier work [18]. §4 presents the extended generic rely-guarantee technique. §5 recaps the SRA memory model and §6 presents a logic for SRA, which includes high-level assertions for memory triples. We present an application of our reasoning framework in §7.

2 Motivating Examples

Weak memory models allow threads to read *stale values* of locations that are older than the last value written for that location. The effects of weak mem-

⁴ We refer the reader to [33, pg 166] and [25] for details on the differences between run and wait.

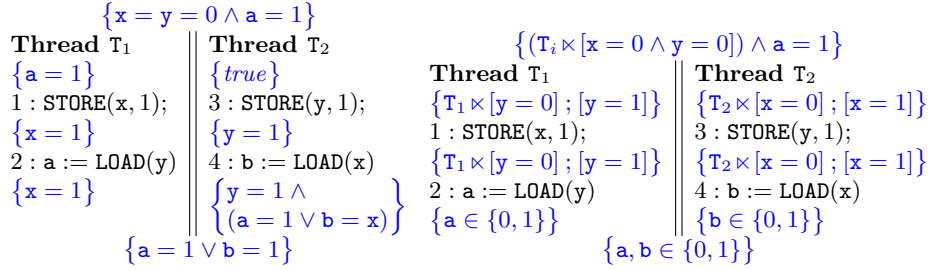


Fig. 1. Store buffering in SC [20]

Fig. 2. Store buffering in SRA can only establish a weaker postcondition

ory on partial correctness have been extensively studied, with many techniques extending well-established frameworks such as Owicki-Gries [3, 6, 7, 20] and RG reasoning [18] as well as different types of separation logics [15]. Loosely speaking, these techniques assume a more general notion of a state (that describe how and when threads can read stale values), and high-level assertions that characterise such states. A natural question to consider how deadlock is affected by a weak memory model. In particular, is it possible to compositionally prove deadlock freedom under weak memory where threads may read stale values?

To make the remaining discussion more concrete, consider the programs in Figures 1 to 4, which present the well-known *store-buffering* (SB) litmus test [2], as well as a novel litmus test that we will refer to as *store buffering with waits* (SBW).

2.1 Store Buffering

For the SB example, using the SRA memory model instead of SC changes the properties that one can infer about the program. In particular, under SC (Fig. 1), one can prove the postcondition⁵ $a = 1 \vee b = 1$, but under SRA (Fig. 2), this is not possible and only the weaker postcondition $a, b \in \{0, 1\}$ holds.

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. Line 1 establishes $x = 1$ and line 2 preserves $x = 1$ since it only reads from y and updates the local register a . **(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 $\mathcal{R}_1 \triangleq \{a = 1, x = 1\}$. **(3)** Thread T_1 guarantees to its concurrent environment (i.e., T_2) that its only interferences are $\text{STORE}(x, 1)$, $a := \text{LOAD}(y)$. We formally capture this condition by a guarantee set $\mathcal{G}_1 \triangleq \{\{true\} T_1 \mapsto \text{STORE}(x, 1), \{x = 1\} T_1 \mapsto a := \text{LOAD}(y)\}$, where each element is a command executed by a thread guarded by a precondition. **(4)** The rely/guarantee assertions corresponding to T_2 are similar. **(5)** To perform the parallel composition, $\langle \mathcal{R}_1, \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2, \mathcal{G}_2 \rangle$ must 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

⁵ The proof outline taken from [20].

is, if $G = \{P\} \tau \mapsto c$, we require the Hoare triple $\{P \cap R\} \tau \mapsto c \{R\}$ to hold. For Fig. 2, 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 unchanged).

This exact technique also applies to the SRA program in Fig. 2 but applied to assertions over the weak-memory state (see [3,6,7,18,32]). To understand the differences between Fig. 1 and Fig. 2, one must first develop a basic understanding of the SRA semantics.

Following [18], in SRA, every memory state records sequences of store mappings (from shared variables to values) that each thread may observe. For example, assuming all variables are initialised to 0, if T_1 executed its code until completion before T_2 even started (so under SC the memory state is the store $\{x \mapsto 1, y \mapsto 0\}$), we may reach the SRA state in which T_1 ’s potential consists of a single store $\{x \mapsto 1, y \mapsto 0\}$, the value of the register a is 0, and T_2 ’s potential is the sequence of stores: $\langle \{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\} \rangle$, which captures the stores that T_2 may observe in the order it may observe them. Note that even though the second store $\{x \mapsto 1, y \mapsto 0\}$ is visible to T_2 , it is not forced to read from this store, and may read the value 0 for x by reading from the first store. This allows the program to terminate with $a = b = 0$, which is not possible under SC.

To capture the above informal reasoning in a Hoare logic, we have designed a new form of assertion capturing the possible locally observable sequences of stores, rather than a single 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, $[x = 1]$ is satisfied by all store sequences in which every store maps x to 1, whereas $[y \neq 1]; [x = 1]$ is satisfied by all store sequences that can be split into a (possibly empty) prefix whose value for y is not 1 followed by a (possibly empty) suffix whose value for x is 1. Assertions of the form $\tau \times I$ state that the potential of thread τ includes only store sequences that satisfy I .

The first assertion in T_1 is implied by the initial condition since $\tau \times I$ implies $\tau \times I; J$. Local correctness of the precondition of line 2 holds because the store does not modify the free variable y in $P \triangleq T_1 \times [y = 0]; [y = 1]$. This enables us to establish the postcondition $a \in \{0, 1\}$ of line 2 since the precondition P indicates that T_1 may read either 0 or 1 for y . As in SC memory, we must also check interference freedom of the assertions. The only relevant assertion is the precondition of line 1 (and line 2), and this is straightforward to establish using the WR-OTHER rule from Fig. 10 (which we will discuss in subsequent sections).

2.2 Store Buffering with Waits

We extend our earlier work [18] and introduce an *await* statement, $\text{AWAIT}(x = e)$, to synchronise concurrent threads. Within $\text{AWAIT}(x = e)$, we assume x is a shared location that is used in the guard condition, $x = e$. The await statement causes the executing thread to *block* until it can read a value for x so that $x = e$ evaluates to *true*.

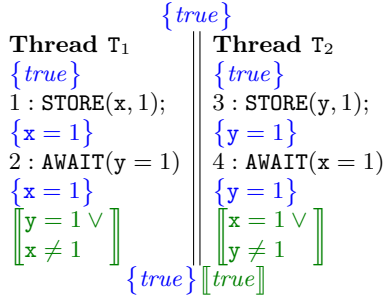


Fig. 3. Store buffering with waits in SC

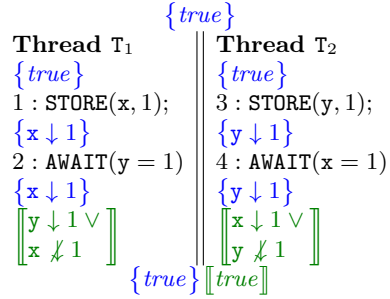


Fig. 4. Store buffering with waits in SRA

Here, we consider SBW, executing under SC (Fig. 3) and SRA (Fig. 4). SBW trivially guarantees the postcondition $true$. Additionally, following Xu et al [33], we introduce a run predicate, denoted by $\llbracket U \rrbracket$, that describe the states in which the program is free from deadlock. In both Figures 3 and 4 the run predicate $true$ indicates that the programs overall are deadlock free.

Note that unlike Xu et al [33], our treatment is generic over memory models. Thus, our semantics of run predicates take into account weak memory effects. In particular, we capture the fact that all weak memory models comprise “internal steps” that model propagation of writes between threads. That is, in every state in which the run predicate holds, the program must be able to execute a program statement possibly after a finite number of propagation steps. This relaxed notion of deadlock freedom is *required* for weak memory since writes executed by a thread are not immediately visible to other threads. For example, consider a thread that is waiting on a lock that is held by another thread. After the thread holding the lock releases it, other threads will only be notified that the lock is free after this information is propagated via internal steps.

It turns out that in our generalised model, proof rules for decomposing deadlock-freedom arguments similar to Xu et al [33] continue to apply. We develop a simpler (and more general) rule for parallel composition. In particular, the overall run predicate of the concurrent program, U , is the union of run predicates U_1 and U_2 , provided that both $Q_1 \cap U_1 \subseteq U_2$ and $Q_2 \cap U_2 \subseteq U_1$ hold, where Q_i is the postcondition of thread T_i .

We now describe the construction of the run predicate of thread T_1 in Fig. 3. First, the run predicates of lines 1 and 2 are $true$ and $y = 1$, respectively. These generate RG judgements comprising run predicates of the form $U \vee \neg P$, where U and P are the run predicate and the precondition of each line. For lines 1 and 2, these are $true$ and $y = 1 \vee x \neq 1$, respectively. Finally, we have a rule of consequence that allows run predicates to be weakened (allowing the run predicate of line 1 to be replaced by $y = 1 \vee x \neq 1$), and a sequential composition rule that allows judgements with the same run predicate to be composed, leading to the overall predicate $y = 1 \vee x \neq 1$ for thread T_1 .

The situation in Fig. 4 is similar, but slightly more subtle. First, recall that SRA states are sequences of stores that each thread can observe. The run pred-

$$\begin{array}{ll}
\text{values} & v \in \text{Val} = \{0, 1, \dots\} \\
\text{local registers} & r \in \text{Reg} = \{\mathbf{a}, \mathbf{b}, \dots\} \\
\text{shared variables} & x, y \in \text{Loc} = \{\mathbf{x}, \mathbf{y}, \dots\} \\
\text{thread identifiers} & \tau, \pi \in \text{Tid} = \{\mathbf{T}_0, \mathbf{T}_1, \dots\} \\
e ::= & r \mid v \mid e + e \mid e = e \mid \neg e \mid e \wedge e \mid e \vee e \mid \dots \\
c ::= & r := e \mid \text{STORE}(x, e) \mid r := \text{LOAD}(x) \mid \text{AWAIT}(x, (\lambda r. e)) \mid \dots \\
\tilde{c} ::= & c \mid \langle c, r := e \rangle \\
C ::= & \tilde{c} \mid \text{SKIP} \mid C ; C \mid \text{IF } e \text{ THEN } C \text{ ELSE } C \mid \text{WHILE } e \text{ DO } C
\end{array}$$

Fig. 5. Program syntax

icates for such sequences are defined in terms of the *last value* of each location, formalised by $x \downarrow v$ and $x \not\downarrow v$, which hold iff the value of x in the last state of the memory sequence is and is not v , respectively. One aspect of the SRA memory model is that all threads *agree* on the last value of every variable, and hence, unlike the assertions in Fig. 2, the last-value assertions can omit thread-specific information. The pre- and postcondition of line 2 state that the last value of \mathbf{x} is 1, and their correctness can be proven correct using standard RG reasoning. The run predicate for thread \mathbf{T}_1 is also similar to SB under SC, and states that $\mathbf{y} \downarrow 1 \vee \mathbf{x} \not\downarrow 1$, which is exactly a disjunction comprising the guard of the await statement and the negation of the await statement’s precondition. Since the run condition of the await is $\mathbf{y} \downarrow 1$, the run condition $\mathbf{y} \downarrow 1 \vee \mathbf{x} \not\downarrow 1$ can be established using the COM rule in Fig. 7.

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. 5, is mostly standard, comprising primitive (atomic) commands c and compound commands C . The non-standard components are instrumented commands $\langle c, r := e \rangle$, which are meant to atomically execute a primitive command c and an assignment $r := e$. Such instructions are needed to support auxiliary (a.k.a. ghost) variables in RG proofs. We additionally include a blocking await statement $\text{AWAIT}(x, \lambda r. e)$, which waits until the executing thread can read a value v of x such that $e(r := v)$ (the expression obtained by substituting v for r in e) evaluates to “true”. By the semantics of $\text{AWAIT}(x, \lambda r. e)$ (see Fig. 6), the program is able to take a step if it can read a value of x so that e evaluates to true. Otherwise, the program is blocked. Note that by construction, $\text{AWAIT}(x, \lambda r. e)$ disallows await statements over more than one shared location — this restriction is necessary to accommodate weak memory semantics. We introduce syntactic sugar $\text{AWAIT}(x = e)$ for $\text{AWAIT}(x, \lambda r. r = e)$.

To simplify the formalisation, we elide read-modify-write instructions, which are used to implement atomic commands such as compare-and-swap and fetch-and-add. The interested reader may wish to refer to our earlier work [18].

We also simplify the presentation by assuming top-level parallelism throughout and model a program as a mapping from thread identifiers to commands.

$$\begin{array}{c}
\frac{\gamma' = \gamma[r \mapsto \gamma(e)]}{r := e \gg \gamma \xrightarrow{\varepsilon} \gamma'} \quad \frac{l = \mathbf{W}(x, \gamma(e))}{\mathbf{STORE}(x, e) \gg \gamma \xrightarrow{l} \gamma} \quad \frac{l = \mathbf{R}(x, v) \quad \gamma' = \gamma[r \mapsto v]}{r := \mathbf{LOAD}(x) \gg \gamma \xrightarrow{l} \gamma'} \\
\\
\frac{c \gg \gamma \xrightarrow{l_\varepsilon} \gamma' \quad v = \gamma'(e) \quad \gamma'' = \gamma'[r \mapsto v]}{\langle c, r := e \rangle \gg \gamma \xrightarrow{l_\varepsilon} \gamma''} \quad \frac{l = \mathbf{R}(x, v) \quad \gamma(e(r := v))}{\mathbf{AWAIT}(x, \lambda r. e) \gg \gamma \xrightarrow{l} \gamma} \\
\\
\hline
\frac{\tilde{c} \gg \gamma \xrightarrow{l_\varepsilon} \gamma'}{\langle \tilde{c}, \gamma \rangle \xrightarrow{l_\varepsilon} \langle \mathbf{SKIP}, \gamma' \rangle} \quad \frac{\langle C_1, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C'_1, \gamma' \rangle}{\langle C_1 ; C_2, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C'_1 ; C_2, \gamma' \rangle} \quad \frac{}{\langle \mathbf{SKIP} ; C_2, \gamma \rangle \xrightarrow{\varepsilon} \langle C_2, \gamma \rangle} \\
\\
\hline
\frac{\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle}{\langle C_0 \uplus \{\tau \mapsto C\}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle C_0 \uplus \{\tau \mapsto C'\}, \gamma' \rangle}
\end{array}$$

Fig. 6. Small-step semantics of instrumented primitive commands $\tilde{c} \gg \gamma \xrightarrow{l_\varepsilon} \gamma'$ (top); commands $\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle$ (middle); and programs $\langle C, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle C', \gamma' \rangle$ (bottom). The semantics of **IF** and **WHILE** are elided and may be found in [18].

Thus, we write programs as sets of the form $\{\tau_1 \mapsto C_1, \dots, \tau_n \mapsto C_n\}$. (Our earlier work [18] allows dynamic thread creation but this creates additional formal overhead that detracts from the main ideas within this paper.)

Program semantics. We provide small-step operational semantics to commands that are independent of the memory system. The semantics of (instrumented) primitive commands, commands and programs are given in Fig. 6. To connect this semantics to a given memory system, its steps are instrumented with labels, which can be either a read label $\mathbf{R}(x, v_{\mathbf{R}})$ or a write label $\mathbf{W}(x, v_{\mathbf{W}})$, where $x \in \mathbf{Loc}$, $v_{\mathbf{R}}, v_{\mathbf{W}} \in \mathbf{Val}$. We denote by \mathbf{Lab} the set of all labels. In turn, in program configurations we record the current *register store* $\gamma : \mathbf{Reg} \rightarrow \mathbf{Val}$. Register stores are extended to expressions as expected. We denote by Γ the set of all register stores. Steps for commands take the form $\langle C, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle C', \gamma' \rangle$, where C and C' are programs, γ and γ' are register stores, and $\langle \tau : l_\varepsilon \rangle$ (with $\tau \in \mathbf{Tid}$ and $l_\varepsilon \in \mathbf{Lab} \cup \{\varepsilon\}$) is a *command transition label*.

Memory semantics. To give semantics to programs under a memory model, we synchronise 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 $\mathcal{M}.\mathbf{Q}$, and steps denoted by $\rightarrow_{\mathcal{M}}$. The transition labels of general memory system \mathcal{M} consist of non-silent program transition labels (elements of $\mathbf{Tid} \times \mathbf{Lab}$) and a (disjoint) set $\mathcal{M}.\mathbf{\Theta}$ of internal memory actions, which is again left parametric (used, e.g., for memory-internal propagation of values).

Example 1. The memory system that guarantees sequential consistency is denoted here by **SC**. This memory system simply tracks the most recent value

written to each variable and has no internal transitions ($\text{SC}.\Theta = \emptyset$). Formally, it is defined by $\text{SC}.\mathbf{Q} \triangleq \text{Loc} \rightarrow \text{Val}$ and \rightarrow_{SC} is given by:

$$\frac{l = \mathbf{R}(x, v_{\mathbf{R}}) \quad m(x) = v_{\mathbf{R}}}{m \xrightarrow{\tau, l}_{\text{SC}} m} \quad \frac{l = \mathbf{W}(x, v_{\mathbf{W}}) \quad m' = m[x \mapsto v_{\mathbf{W}}]}{m \xrightarrow{\tau, l}_{\text{SC}} m'}$$

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

Definition 1. 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 $(\text{Tid} \times (\text{Lab} \cup \{\varepsilon\})) \uplus \mathcal{M}.\Theta$; states are triples of the form $\langle \mathcal{C}, \gamma, m \rangle$ where \mathcal{C} is a program, γ is a register store, and $m \in \mathcal{M}.\mathbf{Q}$; and the transitions are “synchronised transitions” of the program and the memory system, using labels to decide what to synchronise on. These transitions are formalised by the following rules:

$$\frac{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l} \langle \mathcal{C}', \gamma' \rangle \quad l \in \text{Lab} \quad m \xrightarrow{\tau, l}_{\mathcal{M}} m'}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\tau, l}_{\overline{\mathcal{M}}} \langle \mathcal{C}', \gamma', m' \rangle} \quad \frac{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, \varepsilon} \langle \mathcal{C}', \gamma' \rangle}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\tau, \varepsilon}_{\overline{\mathcal{M}}} \langle \mathcal{C}', \gamma', m \rangle} \quad \frac{\theta \in \mathcal{M}.\Theta \quad m \xrightarrow{\theta}_{\mathcal{M}} m'}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\theta}_{\overline{\mathcal{M}}} \langle \mathcal{C}, \gamma, m' \rangle}$$

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 at 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 command specifications (similar to Hoare triples) for primitive commands. Thus, for each instance of the framework (for a specific memory system), one 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 judgements. We let \mathcal{M} be an arbitrary memory system and $\Sigma_{\mathcal{M}} \triangleq \Gamma \times \mathcal{M}.\mathbf{Q}$. Properties of programs \mathcal{C} are stated via *RG judgements*:

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

where $P, Q, U \subseteq \Sigma_{\mathcal{M}}$ are the precondition, postcondition and run condition, respectively, $\mathcal{R} \subseteq \mathcal{P}(\Sigma_{\mathcal{M}})$ is a set of relies, and \mathcal{G} is a set of *guarded commands* (the guarantees) in the style of [20]. Each guarded command takes the form $\{G\} \tau \mapsto \tilde{c}$, where $G \subseteq \Sigma_{\mathcal{M}}$ and \tilde{c} is an (instrumented) primitive command.

Interpretation of RG judgements. RG judgements $\mathcal{C} \text{ sat}_{\mathcal{M}} (P, \mathcal{R}, U, \mathcal{G}, Q)$ state that an execution 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 never block (deadlock) in states of U , will – when \mathcal{C} terminates – 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 judgements are interpreted on *computations* of programs. Computations arise from runs of the concurrent system (see Def. 1) by abstracting away from concrete transition labels and including arbitrary “environment transitions” representing steps of the concurrent context. The transitions in a computation are divided into three sorts:

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

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

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

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

In this paper (going beyond our previous work on RG reasoning for causally consistent memory models [18]), we are specifically interested in proving absence of deadlocks in computations.

Definition 2. A configuration $\langle \mathcal{C}, \gamma, m \rangle$ is called *deadlock* when there is some $\tau \in \text{Tid}$ such that $\mathcal{C}(\tau) \neq \text{SKIP}$ and there is no configuration \mathcal{Y} such that $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\text{mem}}^* \mathcal{Y} \xrightarrow{\text{cmp}} \mathcal{Y}$.

A deadlock is thus a configuration in which there is at least one non-terminated thread τ , but there are no possible component steps — neither in this state nor after having executed a finite number of (internal) memory transitions. In our programming language, deadlocks can only arise when threads reach await statements. In SC, there are no internal memory transitions and deadlock simply means that the await’s condition is not fulfilled in the current state. For weak memory models, this is different. A thread τ might be blocked in some state because it can just read a stale value for some shared variable x ; however, it might become unblocked once some memory model internal steps have occurred. Such a situation occurs in the SBW example (Fig. 4):

Example 2. Consider the await statement in thread T_1 and a state in which the store instruction $\text{STORE}(y, 1)$ of thread T_2 has already occurred. Thread T_1 might still see the (now stale) initial value for y (which is 0) and hence its $\text{AWAIT}(y = 1)$ is blocked. However, if after a number a memory internal steps the current value of y becomes observable for T_1 (and this is the case for the memory model SRA), this will not be considered a deadlock in the memory model.

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

Definition 3. 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 $\mathcal{C} \text{ sat}_{\mathcal{M}} (P, \mathcal{R}, U, \mathcal{G}, Q)$ an RG judgement.

- ξ admits precondition P if $\langle \gamma_0, m_0 \rangle \in P$.
- ξ admits rely \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 $a_{i+1} = \text{env}$.
- ξ admits run condition U if when ξ is finite and $\langle \gamma_{\text{last}(\xi)}, m_{\text{last}(\xi)} \rangle \in U$, then $\langle \mathcal{C}_{\text{last}(\xi)}, \gamma_{\text{last}(\xi)}, m_{\text{last}(\xi)} \rangle$ is not a deadlock.
- ξ admits guarantee \mathcal{G} if for every $i \geq 0$ with $a_{i+1} = \text{cmp}$ and $\langle \gamma_i, m_i \rangle \neq \langle \gamma_{i+1}, m_{i+1} \rangle$ there exists $\{P\} \tau \mapsto \tilde{c} \in \mathcal{G}$ such that $\langle \gamma_i, m_i \rangle \in P$ and for some $l_\varepsilon \in \text{Lab} \cup \{\varepsilon\}$, we have $\langle \{\tau \mapsto \tilde{c}\}, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon} \langle \{\tau \mapsto \text{SKIP}\}, \gamma_{i+1}, m_{i+1} \rangle$.
- ξ admits postcondition Q if $\langle \gamma_{\text{last}(\xi)}, m_{\text{last}(\xi)} \rangle \in Q$ whenever ξ is finite and $\mathcal{C}_{\text{last}(\xi)}(\tau) = \text{SKIP}$ for every $\tau \in \text{dom}(\mathcal{C}_{\text{last}(\xi)})$.

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

Then, *validity* of a judgement is defined as

$$\models \mathcal{C} \text{ sat}_{\mathcal{M}} (P, \mathcal{R}, U, \mathcal{G}, Q) \stackrel{\Delta}{\Leftrightarrow} \text{Comp}(\mathcal{C}) \cap \text{Assume}(P, \mathcal{R}) \subseteq \text{Commit}(U, \mathcal{G}, Q)$$

We say that a program \mathcal{C} *can deadlock from P under \mathcal{R}* if there is a computation $\xi \in \text{Comp}(\mathcal{C}) \cap \text{Assume}(P, \mathcal{R})$ that contains a deadlock, otherwise it is *deadlock-free relative to P and \mathcal{R}* .

The following proposition states that, to guarantee deadlock freedom, it is sufficient to prove validity of a judgement under the weakest possible run predicate and postcondition.

Proposition 1. *A program \mathcal{C} is deadlock-free relative to P and \mathcal{R} if there exists a guarantee \mathcal{G} such that $\mathcal{C} \text{ sat}_{\mathcal{M}} (P, \mathcal{R}, \Sigma, \mathcal{G}, \Sigma)$.*

Command specifications. Our proof rules build on *command specifications*, which specify pre- and postconditions as well as run conditions for primitive commands for a memory system \mathcal{M} .

Definition 4. A *command specification for a memory system \mathcal{M}* is a tuple of the form $\{P\} \tau \mapsto \tilde{c} \{Q\} \llbracket U \rrbracket$, where $P, Q, U \subseteq \Sigma_{\mathcal{M}}$, $\tau \in \text{Tid}$, and \tilde{c} is an instrumented primitive command. A command specification for \mathcal{M} is *valid*, denoted

$$\begin{array}{c}
\text{SKIP} \frac{}{\{\tau \mapsto \text{SKIP}\} \underline{\text{sat}}_{\mathcal{M}}(P, \{P\}, \Sigma, \emptyset, P)} \qquad \text{COM} \frac{\mathcal{M} \vDash \{P\} \tau \mapsto \tilde{c} \{Q\} \llbracket U \rrbracket}{\{\tau \mapsto \tilde{c}\} \underline{\text{sat}}_{\mathcal{M}}(P, \{P, Q\}, (\Sigma \setminus P) \cup U, \{\{P\} \tau \mapsto \tilde{c}\}, Q)} \\
\\
\text{SEQ} \frac{\frac{\{\tau \mapsto C_1\} \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}, U, \mathcal{G}, R) \quad \{\tau \mapsto C_2\} \underline{\text{sat}}_{\mathcal{M}}(R, \mathcal{R}, U, \mathcal{G}, Q)}{\{\tau \mapsto C_1 ; C_2\} \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}, U, \mathcal{G}, Q)} \quad \text{CONSQ} \frac{\frac{\{\tau \mapsto C\} \underline{\text{sat}}_{\mathcal{M}}(P', \mathcal{R}', U', \mathcal{G}', Q') \quad P \subseteq P' \quad \mathcal{R} \leq \mathcal{R}' \quad U \subseteq U' \quad \mathcal{G}' \leq \mathcal{G} \quad Q' \subseteq Q}{\{\tau \mapsto C\} \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}, U, \mathcal{G}, Q)}}{} \\
\\
\text{PAR} \frac{\frac{\frac{\{\tau_1 \mapsto C_1\} \underline{\text{sat}}_{\mathcal{M}}(P_1, \mathcal{R}_1, U_1, \mathcal{G}_1, Q_1) \quad \{\tau_2 \mapsto C_2\} \underline{\text{sat}}_{\mathcal{M}}(P_2, \mathcal{R}_2, U_2, \mathcal{G}_2, Q_2)}{P \subseteq P_1 \cap P_2 \quad Q_1 \cap Q_2 \subseteq Q \quad \langle \mathcal{R}_1, \mathcal{G}_1 \rangle \text{ and } \langle \mathcal{R}_2, \mathcal{G}_2 \rangle \text{ are non-interfering} \quad Q_1 \cap U_1 \subseteq U_2 \quad Q_2 \cap U_2 \subseteq U_1}}{\{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\} \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P, Q\}, U_1 \cup U_2, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}}{}
\end{array}$$

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

by $\mathcal{M} \vDash \{P\} \tau \mapsto \tilde{c} \{Q\} \llbracket U \rrbracket$, when the following *progress* condition holds for every $\langle \gamma, m \rangle \in U$ and some configuration \mathcal{Y} :

$$\exists m'. m \xrightarrow{\theta_1 \dots \theta_{n-1}}_{\mathcal{M}} m' \wedge \langle \tau \mapsto \tilde{c}, \gamma, m' \rangle \xrightarrow{\tau, l_\varepsilon}_{\mathcal{M}} \mathcal{Y}$$

for some $\theta_1, \dots, \theta_{n-1} \in \mathcal{M}.\Theta$, $n \geq 1$, and some $l_\varepsilon \in \text{Lab} \cup \{\varepsilon\}$, and the following *safety* condition holds for every $\langle \gamma, m \rangle \in P$, $\gamma' \in \Gamma$ and $m' \in \mathcal{M}.\mathbf{Q}$, if

$$\langle \{\tau \mapsto \tilde{c}\}, \gamma, m \rangle \xrightarrow{\tau, l_\varepsilon}_{\mathcal{M}} \langle \{\tau \mapsto \text{SKIP}\}, \gamma', m' \rangle$$

for some $l_\varepsilon \in \text{Lab} \cup \{\varepsilon\}$, then $\langle \gamma', m' \rangle \in Q$.

Example 3. For the memory system SC introduced in Ex. 1, we have, e.g., command specifications of the form $\text{SC} \vDash \{e(r := x)\} \tau \mapsto r := \text{LOAD}(x) \{e\} \llbracket \text{true} \rrbracket$ (where $e(r := x)$ is the expression e with all occurrences of r replaced by x) and $\text{SC} \vDash \{P\} \text{AWAIT}(x, \lambda r. e) \{P\} \llbracket e(r := x) \rrbracket$.

RG proof rules. We aim at proof rules for deriving valid RG judgements. Figure 7 lists (semantic) proof rules based on externally provided command specifications. These rules mostly follow RG reasoning for sequential consistency (as of [33]). Again, proof rules for IF and WHILE are elided.

Two rules (COM and PAR) are of specific importance for reasoning about deadlocks. Rule COM builds on command specifications and transfers pre- and postconditions in a command specification to the RG judgement. The run condition in this judgement states the computations of $\tau \mapsto \tilde{c}$ neither block in states in which the precondition is not satisfied (because such states will not be reached in computations starting with P) nor in states of the run condition in the command specification.

Rule PAR for parallel composition combines judgements 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 (5) in §2. Formally, non-interference is defined as follows:

Definition 5. Rely-guarantee pairs $\langle \mathcal{R}_1, \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2, \mathcal{G}_2 \rangle$ are *non-interfering* if $\mathcal{M} \models \{R \cap P\} \tau \mapsto \tilde{c} \{R\} \llbracket false \rrbracket$ holds for every $R \in \mathcal{R}_1$ and $\{P\} \tau \mapsto \tilde{c} \in \mathcal{G}_2$, and similarly for every $R \in \mathcal{R}_2$ and $\{P\} \tau \mapsto \tilde{c} \in \mathcal{G}_1$.

The condition on run predicates within rule PAR slightly deviates from [33]⁶. Given that components C_i ($i \in \{1, 2\}$) do not block in states of U_i , we can deduce $U_1 \cup U_2$ as run condition for the parallel composition when we furthermore ensure $Q_1 \cap U_1 \subseteq U_2$ and $Q_2 \cap U_2 \subseteq U_1$. This can intuitively be understood as ensuring that neither component blocks in states of $U_1 \cup U_2$: First of all, in a state in U_1 C_1 does not block (and dually, in a state in U_2 , C_2 does not block). Now consider a state in U_1 in which C_1 has already terminated. In this case it has established its postcondition Q_1 which together with U_1 implies the state to be in U_2 (and hence C_2 to not block). The dual argument applies to states in U_2 .

Example 4. We exemplify rule PAR on SBW (Fig. 3). Let C_i be the program of thread T_i and $\mathcal{C} : T_i \mapsto C_i$, $i \in \{1, 2\}$. We employ the following guarantee conditions:

$$\begin{aligned} \mathcal{G}_1 &\triangleq \{\{true\}T_1 \mapsto \text{STORE}(x, 1), \{x = 1\}T_1 \mapsto \text{AWAIT}(y = 1)\} \\ \mathcal{G}_2 &\triangleq \{\{true\}T_2 \mapsto \text{STORE}(y, 1), \{y = 1\}T_2 \mapsto \text{AWAIT}(x = 1)\} \end{aligned}$$

The RG judgements for T_1 and T_2 are derivable using rules of Fig. 7 and command specifications as of Ex. 3:

$$\begin{aligned} T_1 \mapsto C_1 &\text{ sat}_{\text{SC}} (true, \{true, x = 1\}, y = 1 \vee x \neq 1, \mathcal{G}_1, x = 1) \\ T_2 \mapsto C_2 &\text{ sat}_{\text{SC}} (true, \{true, y = 1\}, x = 1 \vee y \neq 1, \mathcal{G}_2, y = 1) \end{aligned}$$

Combining these using rule PAR we get:

$$\mathcal{C} \text{ sat}_{\text{SC}} (true, \{true, x = 1, y = 1\}, true, \mathcal{G}_1 \cup \mathcal{G}_2, true)$$

which proves deadlock freedom of the parallel composition relative to the precondition $true$ and the rely $\{true, x = 1, y = 1\}$.

As usual, our proof calculus also includes a rule of consequence CONSQ to be able to strengthen or weaken assertions in RG judgements. Due to the non-standard form of relies and guarantees, we need a specific condition on them, different from subset inclusion. For relies $\mathcal{R}, \mathcal{R}'$ and guarantees $\mathcal{G}, \mathcal{G}'$ we employ the notation

$$\begin{aligned} \mathcal{R} \leq \mathcal{R}' &\text{ iff } \forall R' \in \mathcal{R}', \sigma \in R', \sigma' \in \Sigma. (\forall R \in \mathcal{R}. \sigma \in R \Rightarrow \sigma' \in R) \Rightarrow \sigma' \in R' \\ \mathcal{G}' \leq \mathcal{G} &\text{ iff } \forall \{P'\} \pi \mapsto \tilde{c} \in \mathcal{G}'. \exists P. P' \subseteq P \wedge \{P\} \pi \mapsto \tilde{c} \in \mathcal{G} \end{aligned}$$

to express that relies can be strengthened and guarantees weakened. As an example for relies, we get (formulated in terms of SC-like assertions) $\{x > 2\} \not\leq \{(x > 2) \wedge (x < 5)\}$ and $\{y = 1, x > 2\} \leq \{(y = 1) \wedge (x > 2)\}$. Note that both relations are reflexive.

⁶ The form in [33] can be derived from our rule.

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

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

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. We write $\vdash C \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}, U, \mathcal{G}, Q)$ for provability of a judgement using the semantic rules presented above.

Theorem 1 (Soundness).

$$\vdash C \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}, U, \mathcal{G}, Q) \text{ implies } \models C \underline{\text{sat}}_{\mathcal{M}}(P, \mathcal{R}, U, \mathcal{G}, Q).$$

5 Potential-based Memory System for SRA

We recap the potential-based semantics for Strong Release-Acquire (SRA) as far as necessary for understanding our RG logic (in §6) and the specific aspect of reasoning about deadlock freedom. Our semantics is based on the one in [18] which in turn builds on [16, 17], with certain adaptations to make it better suited for Hoare-style reasoning. Note that for simplicity we do not consider read-modify-write instructions here.

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 6. A *potential store* is a function $\delta : \text{Loc} \rightarrow \text{Val} \times \text{Tid}$. We write $\text{val}(\delta(x))$ and $\text{tid}(\delta(x))$ to retrieve the different components of $\delta(x)$.

Having $\delta(x) = \langle v, \tau \rangle$ allows a thread 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 characterise the SRA model).

Notation 7 Lists over an alphabet A are written as $L = a_1 \cdot \dots \cdot a_n$ where $a_1, \dots, a_n \in A$. 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 .

Potential stores are collected in *potential store lists*: A (potential) store list $L \in \mathcal{L}$ 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. SRA states (SRA.Q) consist of *potential mappings* from threads to sets of store lists.

Definition 8. A *potential D* is a non-empty set of potential store lists. A *potential mapping* is a function $\mathcal{D} : \text{Tid} \rightarrow \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\}$ that maps thread identifiers to potentials such that all lists agree on the very final potential store (i.e., $L_1[|L_1|] = L_2[|L_2|]$ for every $L_1 \in \mathcal{D}(\tau_1), L_2 \in \mathcal{D}(\tau_2), \tau_1, \tau_2 \in \text{Tid}$).

$$\begin{array}{c}
\forall L' \in \mathcal{D}'(\tau). \exists L \in \mathcal{D}(\tau). L' = L[x \mapsto \langle v_w, \tau \rangle] \\
\forall \pi \in \text{dom}(\mathcal{D}) \setminus \{\tau\}, L' \in \mathcal{D}'(\pi). \exists L_0, L_1. \\
L_0 \cdot L_1 \in \mathcal{D}(\pi) \wedge L_1 \in \mathcal{D}(\tau) \wedge \\
L' = L_0 \cdot L_1[x \mapsto \langle v_w, \tau \rangle] \\
\text{WRITE} \frac{}{\mathcal{D} \xrightarrow{\tau, W(x, v_w)}_{\text{SRA}} \mathcal{D}'} \\
\\
\text{LOSE} \frac{\mathcal{D}' \sqsubseteq \mathcal{D}}{\mathcal{D} \xrightarrow{\varepsilon}_{\text{SRA}} \mathcal{D}'} \\
\\
\text{READ} \frac{\exists \pi. \forall L \in \mathcal{D}(\tau). \\
\text{val}(L[1](x)) = v_r \wedge \\
\text{tid}(L[1](x)) = \pi}{\mathcal{D} \xrightarrow{\tau, R(x, v_r)}_{\text{SRA}} \mathcal{D}} \\
\\
\text{DUP} \frac{\mathcal{D} \preceq \mathcal{D}'}{\mathcal{D} \xrightarrow{\varepsilon}_{\text{SRA}} \mathcal{D}'}
\end{array}$$

Fig. 8. Semantics of SRA, defining $\delta[x \mapsto \langle v, \tau \rangle](y) = \langle v, \tau \rangle$ if $y = x$ and $\delta(y)$ else, pointwise lifted to lists

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.

Definition 9. 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 and L' and L agree on the final store (i.e., $L'[|L'|] = L[|L|]$);
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 $\tau \in \text{dom}(\mathcal{D})$.

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. Rule READ details read steps: reading has to take place on the first potential store of the lists, and to this end all lists of the reading thread τ have to contain the same first element. Most of the complexity is left for the WRITE step. It updates to the new written value for the writer thread τ . Additionally, for every other thread, it updates a *suffix* (L_1) of the store list with the new value. To guarantee causal consistency this updated suffix cannot be arbitrary: it has to be in the potential of the writer thread ($L_1 \in \mathcal{D}(\tau)$).

Example 5. Consider again the SB litmus test of Fig. 2 and store lists

$$L = \left[\begin{array}{l} \mathbf{x} \mapsto \langle 1, T_1 \rangle \\ \mathbf{y} \mapsto \langle 0, T_0 \rangle \end{array} \right] \cdot \left[\begin{array}{l} \mathbf{x} \mapsto \langle 1, T_1 \rangle \\ \mathbf{y} \mapsto \langle 1, T_2 \rangle \end{array} \right] \quad L' = \left[\begin{array}{l} \mathbf{x} \mapsto \langle 1, T_1 \rangle \\ \mathbf{y} \mapsto \langle 1, T_2 \rangle \end{array} \right].$$

Thread T_1 might have a potential L after the execution of both store instructions, and can now read y to be 0, i.e., $L \xrightarrow{T_1, R(y, 0)}_{\text{SRA}} L$ is possible. Another possible step is to take LOSE next changing the list to L' from which a transition $L' \xrightarrow{T_1, R(y, 1)}_{\text{SRA}} L'$ is possible.

<i>extended expressions</i>	$E ::= e \mid x \mid E + E \mid \neg E \mid E \wedge E \mid \dots$
<i>interval assertions</i>	$I ::= [E] \mid I ; I \mid I \wedge I \mid I \vee I$
<i>assertions</i>	$\varphi, \psi ::= \tau \times I \mid e \mid x \downarrow \lambda r. e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$

Fig. 9. Assertions of $\text{Piccolo}_{\text{DF}}$

6 Program Logic

For the instantiation of our RG framework to SRA, we next (1) introduce the assertions of our logic $\text{Piccolo}_{\text{DF}}$ and (2) specify command specifications for $\text{Piccolo}_{\text{DF}}$. $\text{Piccolo}_{\text{DF}}$ extends our prior logic, Piccolo [18] with a new assertion $x \downarrow \lambda r. e$ for describing the last value of a location x .

Syntax and semantics. Figure 9 gives the grammar of $\text{Piccolo}_{\text{DF}}$. We base it on *extended expressions* which—besides registers—can also involve locations. 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 $(I_1 ; \dots ; I_n)$ using the “;” operator (called “chop”). In turn, $\tau \times I$ means that all store lists in τ ’s potential satisfy I . In addition to the logic proposed in [18], we employ the assertion $x \downarrow \lambda r. e$ to describe properties about the *last* value of a shared variable x . Since all threads agree on the very last potential store, this assertion is not thread specific. For an assertion φ , we let $fv(\varphi) \subseteq \text{Reg} \cup \text{Loc} \cup \text{Tid}$ be the set of registers, locations and thread identifiers occurring in φ .

Example 6. Consider again the store buffering litmus test SB (Fig. 2). We would like to express that T_1 can potentially see both values of y , 0 and 1, and that it observes these in some order, namely 0 before 1. This can be specified as $T_1 \times [y = 0] ; [y = 1]$. Note that both intervals can also be empty, i.e., the formula also describes states in which T_1 can only observe y to be 0 or only to be 1. In SBW (Fig. 4) we are just interested in the *last* values of shared locations (to see whether await instructions are blocked) and specify this, for instance, as $x \downarrow \lambda r. r = 1$ (denoted $x \downarrow 1$).

An assertion φ describes a set of register stores coupled with SRA states:

Definition 10. Let γ be a register store, δ a potential store, L a store list, and \mathcal{D} a potential mapping. We let $\llbracket e \rrbracket_{\langle \gamma, \delta \rangle} = \gamma(e)$ and $\llbracket x \rrbracket_{\langle \gamma, \delta \rangle} = \delta(x)$. 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 $\llbracket E \rrbracket_{\langle \gamma, \delta \rangle} = \text{true}$ 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. $\langle \gamma, \mathcal{D} \rangle \models \tau \times I$ if $\langle \gamma, L \rangle \models I$ for every $L \in \mathcal{D}(\tau)$.
5. $\langle \gamma, \mathcal{D} \rangle \models e$ if $\gamma(e) = \text{true}$.
6. $\langle \gamma, \mathcal{D} \rangle \models x \downarrow \lambda r. e$ if $\llbracket e(r := x) \rrbracket_{\langle \gamma, L \llbracket L \rrbracket \rangle} = \text{true}$ for all $\tau \in \text{Tid}, L \in \mathcal{D}(\tau)$.
7. $\langle \gamma, \mathcal{D} \rangle \models \varphi_1 \wedge \varphi_2$ if $\langle \gamma, \mathcal{D} \rangle \models \varphi_1$ and $\langle \gamma, \mathcal{D} \rangle \models \varphi_2$ (similarly for \vee).

Assumption	Pre	Command	Post	Run	Reference
	$\{\varphi(r := e)\}$	$\tau \mapsto r := e$	$\{\varphi\}$	$\llbracket true \rrbracket$	SUBST-ASGN
$x \notin fv(\varphi)$	$\{\varphi\}$	$\tau \mapsto \text{STORE}(x, e)$	$\{\varphi\}$	$\llbracket true \rrbracket$	STABLE-WR
$r \notin fv(\varphi)$	$\{\varphi\}$	$\tau \mapsto r := \text{LOAD}(x)$	$\{\varphi\}$	$\llbracket true \rrbracket$	STABLE-LD
	$\{true\}$	$\tau \mapsto \text{STORE}(x, e)$	$\{\tau \times [x = e]\}$	$\llbracket true \rrbracket$	WR-OWN
	$\{\pi \times I\}$	$\tau \mapsto \text{STORE}(x, e)$	$\{\pi \times I; [x = e]\}$	$\llbracket true \rrbracket$	WR-OTHER
	$\{\varphi\}$	$\tau \mapsto \text{AWAIT}(x, \lambda r. e)$	$\{\varphi\}$	$\llbracket x \downarrow \lambda r. e \rrbracket$	AWAIT
$u \neq v$	$\{\tau \times [x = u]\}$	$\tau \mapsto \text{AWAIT}(x = v)$	$\{false\}$	$\llbracket x \downarrow \lambda r. e \rrbracket$	AWAIT-BLOCK

Fig. 10. Piccolo_{DF} axioms for (instrumented) commands, assuming $\tau \neq \pi$

Note that with \wedge and \vee as well as negation on expressions,⁷ 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 threads $\tau, \pi \in \text{Tid}$:

- For $\varphi_1 = \tau \times [E_1^\tau]; \dots; [E_n^\tau]$ and $\varphi_2 = \pi \times [E_1^\pi]; \dots; [E_m^\pi]$: 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 $\langle \gamma, \mathcal{D} \rangle \models \tau \times [e(r := x)]$, then we also get $\langle \gamma, \mathcal{D} \rangle \models x \downarrow \lambda r. e$.

All assertions are preserved by the step LOSE (as well as the second memory model internal step DUP). This stability is required by our RG framework (condition (mem))⁸. Stability is achieved here because negations occur on the level of (simple) expressions only (e.g., we cannot have $\neg(\tau \times [x = v])$), 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 2. *If $\langle \gamma, \mathcal{D} \rangle \models \varphi$ and $\mathcal{D} \xrightarrow{\varepsilon}_{\text{SRA}} \mathcal{D}'$, then $\langle \gamma, \mathcal{D}' \rangle \models \varphi$.*

Axioms. Assertions in Piccolo_{DF} describe sets of states, thus can be used to formulate axioms (cf. [13]) of the RG proof system. Figure 10 gives the axioms for the different primitive instructions.

Example 7. We employ the axioms to show some proof steps for SBW (Fig. 4), namely the derivation of an RG judgement for thread T_1 . First, for the store instruction we employ axiom WR-OWN and obtain

$$\{true\} T_1 \mapsto \text{STORE}(x, 1) \{T_1 \times [x = 1]\} \llbracket true \rrbracket.$$

Next we employ rule COM of Figure 7 and the rule of consequence (using (a) $T_1 \times [x = 1]$ implies $x \downarrow 1$ for the post condition and (b) $x \not\downarrow 1 \vee y \downarrow 1$ implies $true$ for the run condition) and obtain the RG judgement:

⁷ Negation just occurs on the level of simple expressions e which is sufficient for calculating $P \setminus [e]$ required in rules IF and WHILE.

⁸ Such stability requirements are also common to other reasoning techniques for weak memory models, e.g. [9].

$$\begin{aligned} T_1 \mapsto \text{STORE}(x, 1) \text{ sat } \text{SRA} \\ (true, \{true, x \downarrow 1\}, x \not\downarrow 1 \vee y \downarrow 1, \{\{true\}T_1 \mapsto \text{STORE}(x, 1)\}, x \downarrow 1). \end{aligned}$$

Note that, formally, $x \not\downarrow 1$ is shorthand for $x \downarrow (\lambda r. r \neq 1)$. Using axiom **AWAIT**, we obtain the command specification

$$\{x \downarrow 1\} T_1 \mapsto \text{AWAIT}(y = 1) \{x \downarrow 1\} \llbracket y \downarrow 1 \rrbracket.$$

Again lifting this to RG judgements using rule **COM**, we obtain

$$\begin{aligned} T_1 \mapsto \text{AWAIT}(y = 1) \text{ sat } \text{SRA} \\ (x \downarrow 1, \{x \downarrow 1\}, (x \not\downarrow 1) \vee (y \downarrow 1), \{\{x \downarrow 1\}T_1 \mapsto \text{AWAIT}(y = 1)\}, x \downarrow 1). \end{aligned}$$

Next using the rule for sequential composition **SEQ**, we get a judgement for the entire program of thread T_1 which has a run condition $\llbracket (x \not\downarrow 1) \vee (y \downarrow 1) \rrbracket$.

In addition to the axioms above, we use two rules for load instructions. The first rule loads a value from a single interval:

$$\text{LD-SINGLE} \frac{}{\{\tau \times [(e \wedge E)(r := x)]\} \tau \mapsto r := \text{LOAD}(x) \{(e \wedge \tau \times [E])\} \llbracket true \rrbracket}$$

The second rule is a *shift* rule for load instructions when several intervals occur in the precondition:

$$\text{LD-SHIFT} \frac{\{\tau \times I\} \tau \mapsto r := \text{LOAD}(x) \{\psi\} \llbracket true \rrbracket}{\{\tau \times [(e \wedge E)(r := x)]; I\} \tau \mapsto r := \text{LOAD}(x) \{(e \wedge \tau \times [E]; I) \vee \psi\} \llbracket true \rrbracket}$$

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-SINGLE** and **LD-SHIFT** employ backward substitution.

Example 8. We exemplify rules **LD-SINGLE** and **LD-SHIFT** on a proof step of example **SB** in Figure 2. Our objective is to derive a command specification for the load instruction (number 2) in thread T_1 . Using rule **LD-SINGLE** and weakening of the postcondition, we get $\{T_1 \times [y = 1]\} T_1 \mapsto a := \text{LOAD}(y) \{a = 1\} \llbracket true \rrbracket$. We use this as the hypothesis for rule **LD-SHIFT** and another weakening of the postcondition and obtain

$$\{T_1 \times [y = 1]; [y = 1]\} T_1 \mapsto a := \text{LOAD}(y) \{a = 0 \vee a = 1\} \llbracket true \rrbracket$$

For instrumented primitive commands we employ the following rule:

$$\text{INSTR} \frac{\{\psi_0\} \tau \mapsto c \{\psi_1\} \llbracket \varphi \rrbracket \quad \{\psi_1\} \tau \mapsto r := e \{\psi_2\} \llbracket true \rrbracket}{\{\psi_0\} \tau \mapsto \langle c, r := e \rangle \{\psi_2\} \llbracket \varphi \rrbracket}$$

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

Lemma 1. *If $\vdash_{\text{Piccolo}_{\text{DF}}} \{\varphi\} \tau \mapsto \alpha \{\psi\} \llbracket U \rrbracket$ (i.e., a **Piccolo_{DF}** command specification is derivable) and $\Phi = \{\Gamma \mid \Gamma \models \varphi\}$, $\Psi = \{\Gamma \mid \Gamma \models \psi\}$ and $\mathbf{U} = \{\Gamma \mid \Gamma \models U\}$, where $\Gamma = \langle \gamma, \mathcal{D} \rangle$, then $\text{SRA} \models \{\Phi\} \tau \mapsto \alpha \{\Psi\} \llbracket \mathbf{U} \rrbracket$.*

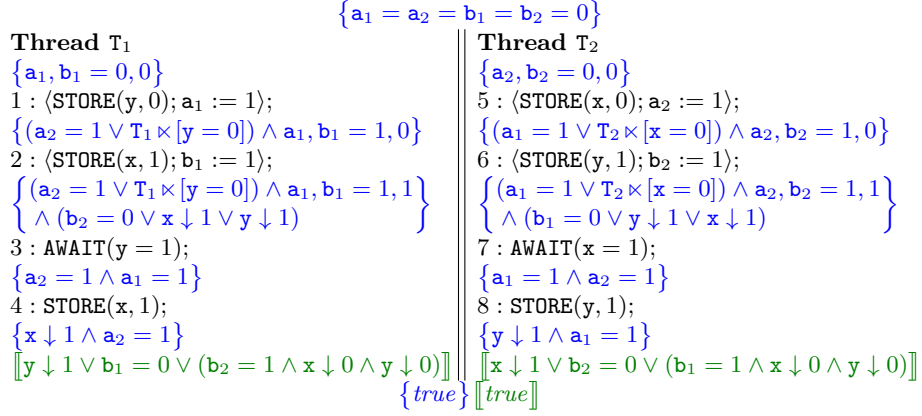


Fig. 11. Initialisation protocol in SRA

7 Initialisation Protocol

Our main example is an initialisation protocol [10], which is a program that aims to synchronise the initialisation of two parallel threads (see Fig. 11). The protocol ensures that each thread only continues execution if it can be sure that the other thread has executed its initialising code. Our protocol is that of Feijen and van Gasteren [10] using two shared variables x and y for synchronisation. Thread T_1 sets y to 0 (ensuring that it will wait at line 3). Additionally it sets x to 1 both immediately before and after the await to stop the other thread (T_2) waiting. Thread T_2 is symmetric.

The proof outline, as given in Fig. 11 requires the introduction of two auxiliary variables in each thread. Auxiliary variable a_1 is used to record that thread T_1 has completed its execution, and b_1 is used to record whether T_1 is waiting to execute its await statement. Under SRA, the proof outline aims to establish two main properties.

1. When both threads are waiting to execute their await statement, one of the guards of the two threads becomes enabled (preconditions $b_2 = 0 \vee x \downarrow 1 \vee y \downarrow 1$ and $b_1 = 0 \vee y \downarrow 1 \vee x \downarrow 1$ together with preconditions $b_1 = b_2 = 1$ of lines 3 and 7).
2. When one of the threads has terminated, the guard of the other thread must eventually become enabled (postconditions $x \downarrow 1$ and $y \downarrow 1$ of lines 4 and 8, respectively).

The remaining of the assertions are introduced to support these properties.

The overall run predicate of each thread T_i is taken as a predicate that ensures that the guard of the only blocking (await) statement becomes true, disjoined with a condition that implies the negation of its precondition. It is straightforward to show that the disjunction of the two run predicates in the two threads evaluates to *true*.

8 Conclusions and Related Work

We have shown that Jones' rely/guarantee technique can be extended to address deadlock freedom under a family of weak memory models. We have focussed on SRA, but similar techniques apply for variants of causal consistency, such as WRA from [17] and LRA from [29], as well as to other memory models with operational semantics. It is also possible to extend similar techniques for Owicki-Gries reasoning for weak memory models developed in prior works [6, 7, 32].

Recent works have examined notions of fairness and liveness for weak memory models. Lahav et al. [19] introduces a notion of memory fairness which is required in addition to scheduler fairness to ensure writes are propagated to other threads. Oberhauser et al. [23] apply model checking techniques to detect liveness violations in weak memory synchronisation algorithms. Abdulla et al. [1] study a different notion of fairness, and describe how verification under such assumptions can be rephrased as a reachability problem. It is interesting to extend the RG paradigm to consider other progress properties besides deadlock freedom, like, e.g., [21, 27, 28] do assuming SC. Another line for future work is the automation of the reasoning in our proposed logic.

References

1. Abdulla, P.A., Atig, M.F., Godbole, A., Krishna, S., Vahanwala, M.: Overcoming memory weakness with unified fairness - systematic verification of liveness in weak memory models. In: CAV. LNCS, vol. 13964, pp. 184–205. Springer (2023). https://doi.org/10.1007/978-3-031-37706-8_10
2. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM TOPLAS **36**(2), 7:1–7:74 (2014), <https://doi.org/10.1145/2627752>
3. 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
4. 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
5. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. For. Asp. Comp. (2022), <https://doi.org/10.1145/3574137>
6. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries reasoning for C11 RAR. In: ECOOP. LIPIcs, vol. 166, pp. 11:1–11:26. Leibniz-Zentrum für Informatik (2020), <https://doi.org/10.4230/LIPIcs.ECOOP.2020.11>
7. Dalvandi, S., Dongol, B., Doherty, S., Wehrheim, H.: Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. Journal of Automated Reasoning **66**(1), 141–171 (2022), <https://doi.org/10.1007/s10817-021-09610-2>
8. 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>
9. Doherty, S., Dalvandi, S., Dongol, B., Wehrheim, H.: Unifying operational weak memory verification: An axiomatic approach. ACM TOCL **23**(4), 27:1–27:39 (2022), <https://doi.org/10.1145/3545117>

10. Feijen, W.H.J., van Gasteren, A.J.M.: On a Method of Multiprogramming. Springer (1999). <https://doi.org/10.1007/978-1-4757-3126-2>
11. Feng, X.: Local rely-guarantee reasoning. In: POPL. pp. 315–327. ACM (2009). <https://doi.org/10.1145/1480881.1480922>
12. Hayes, I.J., Jones, C.B., Meinicke, L.A.: Specifying and reasoning about shared-variable concurrency. In: Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday. LNCS, vol. 14080, pp. 110–135. Springer (2023). https://doi.org/10.1007/978-3-031-40436-8_5
13. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969), <https://doi.org/10.1145/363235.363259>
14. Jones, C.B.: Tentative steps toward a development method for interfering programs. *ACM TOPLAS* **5**(4), 596–619 (1983), <https://doi.org/10.1145/69575.69577>
15. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017), <https://doi.org/10.4230/LIPIcs.ECOOP.2017.17>
16. 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>
17. Lahav, O., Boker, U.: What’s decidable about causally consistent shared memory? *ACM TOPLAS* **44**(2), 8:1–8:55 (2022), <https://doi.org/10.1145/3505273>
18. Lahav, O., Dongol, B., Wehrheim, H.: Rely-guarantee reasoning for causally consistent shared memory. In: CAV. LNCS, vol. 13964, pp. 206–229. Springer (2023). https://doi.org/10.1007/978-3-031-37706-8_11
19. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making weak memory models fair. *Proc. ACM Program. Lang.* **5**(OOPSLA), 1–27 (2021). <https://doi.org/10.1145/3485475>
20. 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
21. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. *SIGPLAN Not.* **51**(1), 385–399 (jan 2016), <https://doi.org/10.1145/2914770.2837635>
22. Meinicke, L.A., Hayes, I.J.: Using cylindric algebra to support local variables in rely/guarantee concurrency. In: *FormaliSE*. pp. 108–119. IEEE (2023). <https://doi.org/10.1109/FORMALISE58978.2023.00019>
23. Oberhauser, J., Chehab, R.L.d.L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., Vafeiadis, V.: Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In: *ASPLOS*. p. 530–545. ACM, New York, NY, USA (2021), <https://doi.org/10.1145/3445814.3446748>
24. 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>
25. Qiwen, X.: A theory of state-based parallel programming. Ph.D. thesis, University of Oxford, UK (1992)
26. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: *VSTTE*. LNCS, vol. 6217, pp. 55–70 (2010), https://doi.org/10.1007/978-3-642-15057-9_4
27. 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>
28. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: iFM. LNCS, vol. 9681, pp. 193–209. Springer (2016), https://doi.org/10.1007/978-3-319-33693-0_13
 29. Singh, A.K., Lahav, O.: Decidable verification under localized release-acquire concurrency. In: TACAS. LNCS, vol. 14572, pp. 235–254. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_12, https://doi.org/10.1007/978-3-031-57256-2_12
 30. Stølen, K.: Development of Parallel Programs on Shared Data-structures. Ph.D. thesis, Computer Science Department, Manchester University (1990)
 31. 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
 32. 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
 33. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. For. Asp. Comp. **9**(2), 149–174 (1997), <https://doi.org/10.1007/BF01211617>