Compositional reasoning for non-multicopy atomic architectures

NICHOLAS COUGHLIN*, Defence Science and Technology Group, Australia; School of ITEE, The University of Queensland, Australia
KIRSTEN WINTER, Defence Science and Technology Group, Australia; School of ITEE, The University of Queensland, Australia
GRAEME SMITH, Defence Science and Technology Group, Australia; School of ITEE, The University of Queensland, Australia

Rely/guarantee reasoning provides a compositional approach to reasoning about concurrent programs. However, such reasoning traditionally assumes a sequentially consistent memory model and hence is unsound on modern hardware in the presence of data races. In this paper, we present a rely/guarantee-based approach for *non-multicopy atomic* weak memory models, i.e., where a thread's stores are not simultaneously propagated to all other threads and hence are not observable by other threads at the same time. Such memory models include those of the earlier versions of the ARM processor as well as the POWER processor.

This paper builds on our approach to compositional reasoning for *multicopy atomic* architectures, i.e., where a thread's stores are simultaneously propagated to all other threads. In that context, an operational semantics can be based on thread-local instruction reordering. We exploit this to provide an efficient compositional proof technique in which weak memory behaviour can be shown to preserve rely/guarantee reasoning on a sequentially consistent memory model. To achieve this, we introduce a side-condition, *reordering interference freedom* on each thread, reducing the complexity of weak memory to checks over pairs of reorderable instructions.

In this paper we extend our approach to non-multicopy atomic weak memory models. We utilise the idea of reordering interference freedom between parallel components. This by itself would break compositionality but serves as a vehicle to derive a refined compatibility check between rely and guarantee conditions which takes into account the effects of propagations of stores that are only partial, i.e., not covering all threads. All aspects of our approach have been encoded and proved sound in Isabelle/HOL.

CCS Concepts: • **Theory of computation → Logic and verification**; **Hoare logic**; **Automated reasoning**.

Additional Key Words and Phrases: Verification, rely/guarantee reasoning, weak memory models, non-multicopy atomicity

ACM Reference Format:
Nicholas Coughlin, Kirsten Winter, and Graeme Smith. 2022. Compositional reasoning for non-multicopy atomic architectures. 1, 1 (August 2022), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Nicholas Coughlin, Defence Science and Technology Group, Australia; School of ITEE, The University of Queensland, Australia, nicholas.coughlin@uqconnect.edu.au; Kirsten Winter, Defence Science and Technology Group, Australia; School of ITEE, The University of Queensland, Australia, kirsten@itee.uq.edu.au; Graeme Smith, Defence Science and Technology Group, Australia; School of ITEE, The University of Queensland, Australia, smith@itee.uq.edu.au.

XXXX-XXXX/2022/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Reasoning about concurrent programs with interference over shared resources is a complex task. The interleaving of all thread behaviours leads to an exponential explosion in observable behaviour. Rely/guarantee reasoning [14] is one approach to reduce the complexity of the verification task. It enables reasoning about one thread at a time by considering an abstraction of the thread's environment given as a *rely condition* on shared resources. This abstraction is justified by proving that all other threads in the environment guarantee the assumed rely condition. The approach limits the interference between threads to the effects of the rely condition (specified as a relation over states).

Xu et al. [37] show how rely/guarantee reasoning can be used to allow reasoning over individual threads in a concurrent program using Hoare logic [13]. We introduce a similar approach in [36] to allow thread-local reasoning, in the context of information flow security, using weakest precondition calculation [10]. These approaches work equally well for concurrent programs executed on weak memory models under the implicit assumption that the code is data-race free. This is a reasonable assumption given that most programmers avoid data races due to them leading to unexpected behaviour when the code's execution is optimised under the weak memory model of the compiler [4, 15] or underlying hardware [3, 7, 30]. However, data races may be introduced inadvertently by programmers, or programmers may introduce data races for efficiency reasons, as seen in non-blocking algorithms [22]. These algorithms appear regularly in the low-level code of operating systems, e.g., seqlock [5] is used routinely in the Linux kernel, and software libraries, e.g., the Michael-Scott queue [21] is used as the basis for Java's ConcurrentLinkedQueue in java.util.concurrent.

This paper defines a proof system for rely/guarantee reasoning that is parameterised by the weak memory model under consideration. In a previous publication [8] we restricted our focus to those memory models that are *multicopy atomic*, i.e., where a thread's stores become observable to all other threads at the same time. This includes the memory models of x86-TSO [29], ARMv8 [26] and RISC-V [35] processor architectures, but not POWER [28], older ARM [11] processors nor C11 [4]. As shown by Colvin and Smith [7], multicopy atomic memory models can be captured in terms of instruction reordering. That is, they can be characterised by a reordering relation over pairs of instructions in a thread's code, indicating when two instructions may execute out-of-order. This has been validated against the same sets of litmus test used to validate the widely accepted weak memory semantics of Alglave et al. [3].

Consequently, the implications of weak memory can be captured thread-locally, enabling compositional reasoning. However, thread-local reasoning under such a semantics is non-trivial. Instruction reordering introduces interference within a single thread, similar to the effects of interference between concurrent threads and equally hard to reason about. For instance, a thread with n reorderable instructions may have $n!$ behaviours due to possible reordering. To tackle such complexity, we exploit the fact that many of these instructions will not influence the behaviour of others. We reduce the verification burden to a standard rely/guarantee judgement [37], over a sequentially consistent memory model, and a consideration of the pair-wise interference between reorderable instructions in a thread, totalling $n(n-1)/2$ pairs given n reorderable instructions. The resulting proof technique has been automated and shown to be sound on both a simple while language and an abstraction of ARMv8 assembly code using Isabelle/HOL [24] (see https://bitbucket.org/wmmif/wmm-rg).

This paper extends the work of [8] in that it additionally provides an approach to compositional reasoning for *non-multicopy atomic* architectures. For non-multicopy atomic processors such as POWER and older versions of ARM, the semantics of Colvin and Smith refers to a *storage subsystem* to capture each component's *view* of the global memory. This view depends on the *propagations*

of writes performed by the hardware. It is this view of a component that provides the point of reference for the rely/guarantee reasoning.

To capture the semantics of propagations which deliver a particular view, one can utilise the notion of instruction reorderings between components. However, reasoning about such reorderings can not be performed thread-locally, and the compositionality of the approach would be lost. Instead we reason over reorderings of an instruction with behaviours of the rely condition, which abstractly represents the behaviours of the instructions of other components. By lifting the argument of reordering interference freedom between components to the abstract level, compositionality is maintained. We show how this global reordering interference freedom manifests itself in our theory as a specialisation of the compatibility between guarantee and rely conditions that is standard in rely/guarantee reasoning [14].

There are a number of approaches for verifying concurrent code under weak memory models [1, 2, 12, 17–19, 33, 34], which are centred around relations between instructions in multiple threads, thereby precluding the benefits of thread-local reasoning. Notable amongst these is the work by Abdulla et al. [1, 2] which aims at automated tool support via stateless model checking and is based on the axiomatic semantic model of [3]. Instead of thread-local reasoning the approaches deal with execution graphs which include not only the interleaving behaviour of concurrent threads but also "parallelisation" of sequential code resulting from weak memory behaviour. Techniques to combat the resulting state-space explosion and improve scalability include elaborate solutions to dynamic partial order reduction, context bounds for a bug-finding technique [1] and (for a sound approach) coarsening the semantic model of execution graphs through reads-from equivalences [2].

Approaches that propose a purely thread-local analysis for concurrent code under weak memory models include the work by Ridge [27] and Suzanne et al. [32]. Both capture the weak memory model of x86-TSO [29] by modelling the concept of store buffers. This limits their applicability to that of this relatively simple memory model and prohibits adaption to weaker memory models.

Closer to our approach are the proof systems for concurrent programs under the C11 memory model developed by Lahav et al. [19] and Dalvandi et al. [9]. These proof system are based on the notion of Owicki-Gries reasoning with interference assertions between each line of code to capture potential interleavings.

However, to achieve a thread-local approach the authors of [19] present their logic in a "rely/guarantee style" in which interference assertions are collected in "rely sets" whose stability needs to be guaranteed by the current thread. This leads to a fine-grained consideration of interference between threads whereas in standard rely/guarantee reasoning the interference is abstracted into a rely condition which summarises the effects of the environment. Moreover, similarly to [1, 2] the semantic model is based on (an abstraction of) the axiomatic model in [3] so that the interference between threads includes additionally weak memory effects thereby further complicating the analysis over each instruction. A somewhat-related approach to capture assertions on thread interference is presented in [18] which computes the reads-from relation between threads which is then taken into account by the thread-local static analyser.

In contrast, the work in [9] provides a more expressive view-based assertion language which allows for the use of the standard proof rules of Owicki-Gries reasoning for concurrent systems despite the effects of the weak memory model including non-multicopy atomicity. As a consequence, the complications of weak memory effects need to be taken into account when crafting the assertions to show interference freedom. In our approach the constraints that define the guarantee and rely conditions are specified as simple relational predicates over the shared variables. The intricacies of weak memory effects and non-multicopy atomicity are hidden in the proof technique given by our logic which is proven sound with respect to the weak memory model.

We begin the paper in Section 2 with a formalisation of a basic proof system for rely/guarantee reasoning introduced in [37]. In Section 3, we abstractly introduce reordering semantics for weak memory models and our notion of *reordering interference freedom* which suffices to account for the effects of the weak memory model under multicopy atomicity. We discuss the practical implications of the approach. To take the effects of non-multicopy atomicity into account Section 4 introduces the additional notion of *global* reordering interference freedom which is encoded into the proof system via a refined compatibility check. In Section 5 we present the instantiation of the approach with a simple language and demonstrate reasoning in Section 6 by means of an example. We conclude in Section 7.

2 PRELIMINARIES

The language for our framework is purposefully kept abstract so that it can be instantiated for different programming languages. It consists of individual instructions α, whose executions are atomic, and *commands* (or programs) c which are composed of instructions using sequential composition, nondeterministic choice, iteration, and parallel composition. Commands also include the empty program ϵ denoting termination.

$$c ::= \epsilon \mid \alpha \mid c_1 \,;\, c_2 \mid c_1 \sqcap c_2 \mid c^* \mid c_1 \parallel c_2$$

Note that conditional instructions (like if-then-else and loops) and their evaluation are modelled via silent steps making a nondeterministic choice during the execution of a program (see Section 5).

A *configuration* of a program is a pair (c, σ), consisting of a command c to be executed and state σ (a mapping from variables to values) in which it executes. The behaviour of a component, or thread, in a concurrent program can be described via steps the program, including its environment, can perform during execution, each modelled as a relation between the configurations before and after the step. A *program step*, denoted as $(c, \sigma) \xrightarrow{ps} (c', \sigma')$, describes a single step of the component itself and changes the command (i.e., the remainder of the program). A program step may be an *action step* $(c, \sigma) \xrightarrow{as} (c', \sigma')$ which performs an instruction that also changes the state, or a *silent step*, $(c, \sigma) \rightsquigarrow (c', \sigma)$ which does not execute an instruction but makes a choice and thus changes the command only. Hence $\xrightarrow{ps} = (\xrightarrow{as} \cup \rightsquigarrow)$. An *environment step*, $(c, \sigma) \xrightarrow{es} (c, \sigma')$, describes a step of the environment (performed by any of the other concurrent components); it may alter the state but not the remainder of the program (of the component).

Program *execution* is defined via a small-step semantics over the command.

$$\begin{array}{l} \alpha \mapsto_\alpha \epsilon \\ c_1 \,;\, c_2 \mapsto_\alpha c_1' \,;\, c_2 \text{ if } c_1 \mapsto_\alpha c_1' \\ c_1 \parallel c_2 \mapsto_\alpha c_1' \parallel c_2 \text{ if } c_1 \mapsto_\alpha c_1' \text{ or } c_1 \parallel c_2 \mapsto_\alpha c_1 \parallel c_2' \text{ if } c_2 \mapsto_\alpha c_2' \end{array} \tag{1}$$

The *semantics* of program steps is based on the evaluation of instructions. Each atomic instruction α has a relation over (pre- and post-) states $beh(\alpha)$, formalising its execution behaviour. A program step $(c, \sigma) \xrightarrow{as} (c', \sigma')$ requires an execution $c \mapsto_\alpha c'$ to occur such that the state is updated according to the executed instruction α, i.e.,

$$(c, \sigma) \xrightarrow{as} (c', \sigma') \iff \exists \alpha.\ c \mapsto_\alpha c' \wedge (\sigma, \sigma') \in beh(\alpha)\,. \tag{2}$$

2.1 Rely/guarantee reasoning

A proof system for rely/guarantee reasoning in a Hoare logic style has been defined in [37]. Our approach largely follows its definitions, but includes a customisable verification condition, *vc*, with each instruction. This verification condition serves to capture the state an instruction must execute under to enforce properties such as the component's guarantee and potentially more specialised

analyses. For example, in an information flow security analysis (cf. [36]), it can be used to check that the value assigned to a publicly accessible variable is not classified. We define a Hoare triple as follows. For simplicity of presentation, we treat predicates as sets of states or equivalently $P \subseteq vc(\alpha) \cap wp(beh(\alpha), Q)$, using the definition of weakest preconditions [10].

$$P\{\alpha\}Q \mathrel{\widehat{=}} P \subseteq vc(\alpha) \cap \{\sigma \mid \forall \sigma'.\, (\sigma, \sigma') \in beh(\alpha) \implies \sigma' \in Q\} \tag{3}$$

The rely and guarantee conditions of a thread, denoted $\mathcal{R}$ and $\mathcal{G}$ respectively, are relations over (pre- and post-) states. The rely condition captures allowable environments steps and the guarantee constrains all program steps. A rely/guarantee pair $(\mathcal{R}, \mathcal{G})$ is wellformed when the rely condition is reflexive and transitive, and the guarantee condition is reflexive.

Given that $\mathcal{R}$ is transitive, stability of a predicate P under rely condition $\mathcal{R}$ is defined such that $\mathcal{R}$ maintains P.

$$stable_{\mathcal{R}}(P) \mathrel{\widehat{=}} P \subseteq \{\sigma \mid \forall \sigma'.\, (\sigma, \sigma') \in \mathcal{R} \implies \sigma' \in P\} \tag{4}$$

The conditions under which an instruction satisfies $\mathcal{G}$ is defined as

$$sat(\alpha, \mathcal{G}) \mathrel{\widehat{=}} \{\sigma \mid \forall \sigma'.\, (\sigma, \sigma') \in beh(\alpha) \implies (\sigma, \sigma') \in \mathcal{G}\}. \tag{5}$$

These ingredients allow us to introduce a rely/guarantee judgement. We do this on three levels: the instruction level $\vdash_{\mathsf{a}}$, the component level $\vdash_{\mathsf{c}}$, and the global level $\vdash$. On the instruction level the judgement requires that the pre- and post-condition are stable under $\mathcal{R}$. This ensures that these conditions, and hence the Hoare triple, hold despite any environmental interference. Additionally, the judgement requires that the instruction satisfies the guarantee $\mathcal{G}$.

$$\mathcal{R}, \mathcal{G} \vdash_{\mathsf{a}} P\{\alpha\}Q \mathrel{\widehat{=}} stable_{\mathcal{R}}(P) \wedge stable_{\mathcal{R}}(Q) \wedge vc(\alpha) \subseteq sat(\alpha, \mathcal{G}) \wedge P\{\alpha\}Q \tag{6}$$

A rely/guarantee proof system on the component and global levels follows straightforwardly and is given in Figure 1. At the component level, note the necessity for the invariant of the [Iteration] rule to be stable (such that it continues to hold amid environmental interference). At the global level, the rule for parallel composition [Par] includes a compatibility check ensuring that the guarantee for each component implies the rely conditions of the other component. A standard [Conseq] rule over global satisfiability is supported by the proof system, but omitted in Figure 1.

Such rules are standard to rely/guarantee reasoning [37]. Our modification can be seen in can be seen in [Comp], in which global satisfiability is deduced from component satisfiability $\vdash_{\mathsf{c}}$ plus an additional check on *reordering interference freedom*, $rif(\mathcal{R}, \mathcal{G}, c)$, which we introduce in Section 3.2. As a consequence, component-based reasoning in this proof system is based on standard rely/guarantee reasoning which can be conducted independently from the interference check.

Moreover, the proof system supports a notion of *auxiliary* variables, common to rely/guarantee reasoning [31, 37]. These variables increase the expressiveness of the specification ($\mathcal{R}$, $\mathcal{G}$, P and Q) by representing properties of intermediate execution states. Auxiliary variables cannot influence program execution, as they are abstract, and their modification must be coupled with an instruction such that they are considered atomic.

3 MULTICOPY ATOMIC MEMORY MODELS

Weak memory models are commonly defined to maintain sequentially consistent behaviour given the absence of data races, thereby greatly simplifying reasoning for the majority of programs. However, as we are interested in the analysis of racy concurrent code, it is necessary to reason on a semantics that fully captures the behaviours these models may introduce.

Colvin and Smith [7] show that weak memory behaviour for multicopy atomic processors such as x86-TSO, ARMv8 and RISC-V can be captured in terms of instruction reordering. A memory model,

$$[\text{Atom}]\frac{\mathcal{R}, \mathcal{G} \vdash_{\text{a}} P\{\alpha\}Q}{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{\alpha\}Q} \qquad [\text{Seq}]\frac{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c_1\}M \quad \mathcal{R}, \mathcal{G} \vdash_{\text{c}} M\{c_2\}Q}{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c_1 \,;\, c_2\}Q}$$

$$[\text{Choice}]\frac{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c_1\}Q \quad \mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c_2\}Q}{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c_1 \sqcap c_2\}Q} \qquad [\text{Iteration}]\frac{\mathit{stable}_{\mathcal{R}}(P) \quad \mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c\}P}{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c^*\}P}$$

$$[\text{Conseq}]\frac{P' \subseteq P \quad \mathcal{R}' \subseteq \mathcal{R} \quad \mathcal{G} \subseteq \mathcal{G}' \quad Q \subseteq Q' \qquad \mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c\}Q}{\mathcal{R}', \mathcal{G}' \vdash_{\text{c}} P'\{c\}Q'}$$

$$[\text{Comp}]\frac{\mathcal{R}, \mathcal{G} \vdash_{\text{c}} P\{c\}Q \quad \mathit{rif}(\mathcal{R}, \mathcal{G}, c)}{\mathcal{R}, \mathcal{G} \vdash P\{c\}Q}$$

$$[\text{Par}]\frac{\mathcal{R}_1, \mathcal{G}_1 \vdash P_1\{c_1\}Q_1 \quad \mathcal{R}_2, \mathcal{G}_2 \vdash P_2\{c_2\}Q_2 \quad \mathcal{G}_2 \subseteq \mathcal{R}_1 \quad \mathcal{G}_1 \subseteq \mathcal{R}_2}{\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2 \vdash P_1 \cap P_2\{c_1 \parallel c_2\}Q_1 \cap Q_2}$$

Fig. 1. Proof rules for rely/guarantee reasoning

in these cases, is characterised by a reordering relation over pairs of instructions indicating whether the two instructions can execute out-of-order when they appear in a component's code. This complicates reasoning significantly. For example, one needs to determine whether an instruction α that is reordered to execute earlier in a program can invalidate verification conditions that are satisfiable under normal executions (following the program order without reordering). In that sense, we are facing not only interference between concurrent components (which can be visualised as *horizontal* interference) but also interference between the instructions within one component (which can be pictured as *vertical* interference).

3.1 Reordering semantics

The reordering relation, $\hookleftarrow$, of a component is syntactically derivable based on the rules of the specific memory model (see Section 3.3). In ARMv8, for example, two instructions which do not access (write or read) a common variable are deemed semantically independent and can change their execution order. Moreover, weak memory models support various memory barriers that prevent particular forms of reordering. For example, a full fence prevents all reordering, while a control fence prevents speculative execution (for a complete definition refer to [7]).

Matters are complicated by the concept of *forwarding*, where an instruction that reads from a variable written in an earlier instruction might replace the reading access with the written value, hence shedding the dependence to the variable in common. This allows it to execute earlier, anticipating the write before it happens. For example x := z; y := x can execute as y := z; x := z. We denote the instruction α with the value written in an earlier instruction β forwarded to it as $\alpha_{\langle\beta\rangle}$. Note that $\alpha_{\langle\beta\rangle} = \alpha$ whenever β does not write to a variable that is read by α.

Forwarding can span a series of instructions and can continue arbitrarily, with later instructions allowed to replace variables introduced by earlier forwarding modifications. The term $\gamma \prec c \prec \alpha$ denotes reordering of the instruction α prior to the command c, with the cumulative forwarding effects producing γ [6]. $\alpha_{\langle\langle c\rangle\rangle}$ denotes the cumulative forwarding effects of the instructions in

command c on α. We define both terms recursively over c.

$$\begin{array}{rcl} \alpha_{\langle\beta\rangle} \prec \beta \prec \alpha & \widehat{=} & \beta \hookleftarrow \alpha_{\langle\beta\rangle} \\ \alpha_{\langle\langle c_1\,;\,c_2\rangle\rangle} \prec c_1\,;\,c_2 \prec \alpha & \widehat{=} & \alpha_{\langle\langle c_1\,;\,c_2\rangle\rangle} \prec c_1 \prec \alpha_{\langle\langle c_2\rangle\rangle} \;\wedge\; \alpha_{\langle\langle c_2\rangle\rangle} \prec c_2 \prec \alpha \end{array} \tag{7}$$

To capture the effects of reordering, we extend the definition of executions (1) with an extra rule that captures out-of-order executions: A step can execute an instruction whose original form occurs later in the program if reordering and forwarding can bring it (in its new form γ) to the beginning of the program.

$$c_1\,;\,c_2 \mapsto_\gamma c_1\,;\,c_2' \text{ if } \gamma \prec c_1 \prec \alpha \wedge c_2 \mapsto_\alpha c_2' \tag{8}$$

3.2 Reordering interference freedom

Our aim is to eliminate the implications of this reordering behaviour and, therefore, enable standard rely/guarantee reasoning despite a weak memory context. To achieve this, we note that a valid reordering transformation will preserve the thread-local semantics and, hence, will only invalidate reasoning when observed by the environment. Such interactions are captured either as invalidation of the component's guarantee $\mathcal{G}$ or new environment behaviours, as allowed by its rely condition $\mathcal{R}$. Consequently, reorderings may be considered benign if the modified variables are not related by $\mathcal{G}$ or $\mathcal{R}$.

We capture such benign reorderings via reordering interference freedom. Two instructions are said to be *reordering interference free* (*rif*) if we can show that reasoning over the instructions in their original (program) order is sufficiently strong to also include reasoning over their reordered behaviour. Consider the program text $\beta\,;\,\alpha$, where α can be forwarded and executed before β, resulting in an execution equivalent to $\alpha_{\langle\beta\rangle}\,;\,\beta$. Reordering interference freedom between α and β under given rely/guarantee conditions is then formalised as follows.

$$\begin{array}{l} rif_{\mathsf{a}}(\mathcal{R},\mathcal{G},\beta,\alpha) \mathrel{\widehat{=}} \forall P, Q, M.\ \mathcal{R},\mathcal{G} \vdash_{\mathsf{a}} P\{\beta\}M \wedge \mathcal{R},\mathcal{G} \vdash_{\mathsf{a}} M\{\alpha\}Q \implies \\ \qquad \exists M'.\ \mathcal{R},\mathcal{G} \vdash_{\mathsf{a}} P\{\alpha_{\langle\beta\rangle}\}M' \wedge \mathcal{R},\mathcal{G} \vdash_{\mathsf{a}} M'\{\beta\}Q \end{array} \tag{9}$$

Importantly, rif_{a} is defined independently of the pre- and post-states of the given instructions, as can be seen by the universal quantification over P, M and Q in (9). This independence allows for the establishment of rif_{a} across a program via consideration of only pairs of reorderable instructions, rather than that of all execution traces under which they may be reordered. Such an approach dramatically reduces the complexity of reasoning in the presence of reordering, from one of $n!$ transformed programs for n reorderable instructions to $n(n-1)/2$ pairs.

The definition of rif_{a} extends inductively over commands c with which α can reorder. Command c is *reordering interference free* from α under $\mathcal{R}$ and $\mathcal{G}$, if the reordering of α over each instructions of c is interference free, including those variants of α produced by forwarding.

$$\begin{array}{rcl} rif_{\mathsf{c}}(\mathcal{R},\mathcal{G},\beta,\alpha) & = & rif_{\mathsf{a}}(\mathcal{R},\mathcal{G},\beta,\alpha) \\ rif_{\mathsf{c}}(\mathcal{R},\mathcal{G},c_1\,;\,c_2,\alpha) & = & rif_{\mathsf{c}}(\mathcal{R},\mathcal{G},c_1,\alpha_{\langle\langle c_2\rangle\rangle}) \wedge rif_{\mathsf{c}}(\mathcal{R},\mathcal{G},c_2,\alpha) \end{array} \tag{10}$$

From the definition of executions including reordering behaviour given in (8) we have $c \mapsto_{\alpha_{\langle r\rangle}} c' \implies r\,;\,\alpha \in prefix(c) \wedge \alpha_{\langle\langle r\rangle\rangle} \prec r \prec \alpha$, where $prefix(c)$ refers to the set of prefixes of c. Program c is *reordering interference free* if and only if all possible reorderings of its instructions over the respective prefixes are reordering interference free.

$$rif(\mathcal{R},\mathcal{G},c) \mathrel{\widehat{=}} \forall \alpha, r, c'.\ c \mapsto_{\alpha_{\langle r\rangle}} c' \implies rif_{\mathsf{c}}(\mathcal{R},\mathcal{G},r,\alpha) \wedge rif(\mathcal{R},\mathcal{G},c') \tag{11}$$

As can be seen from the definitions, checking $rif(\mathcal{R},\mathcal{G},c)$ amounts to checking $rif_{\mathsf{a}}(\mathcal{R},\mathcal{G},\beta,\alpha)$ for all pairs of instructions β and α that can reorder in c, including those pairs for which α is a new instruction generated through forwarding. Therefore one can reason about a component's code as follows.

(1) Compute all pairs of reorderable instructions, i.e., each pair of instructions (β, α) such that there exists an execution trace where α reorders before β according to the memory model under consideration.
(2) Demonstrate reordering interference freedom for as many of these pairs as possible (using $rif_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, \beta, \alpha)$).
(3) If rif_{a} cannot be shown for some pairs, introduce memory barriers to prevent their reordering or modify the verification problem such that their reordering can be considered benign.
(4) Verify the component in isolation, using standard rely/guarantee reasoning with an assumed sequentially consistent memory model.

We detail steps 1-3 in the following sections and assume the use of any standard rely/guarantee reasoning approach for step 4.

3.3 Computing all reorderable instructions

Pairs of potentially reorderable instructions can be identified via a dataflow analysis [16], similar to dependence analysis commonly used in compiler optimisation. However, rather than attempting to establish an absence of dependence, we are interested in demonstrating its presence, such that instruction reordering is not possible during execution. This notion of dependence is derived from the language's reordering relation, such that α is dependent on β iff $\beta \not\hookleftarrow \alpha$. All pairs of instructions for which a dependence cannot be established are assumed reorderable.

The approach is constructed as a backwards analysis over a component's program text, incrementally determining the instructions a particular instruction is dependent on and, inversely, those it can reorder before. Therefore, the analysis can be viewed as a series of separate analyses, one from the perspective of each instruction in the program text.

We describe one instance of this analysis for some instruction α. The analysis records a notion of α's cumulative dependencies, which simply begins as all instructions γ for which $\gamma \not\hookleftarrow \alpha$. The analysis commences at the instruction immediately prior to α in the program text and progresses backwards. For each instruction β we first determine if α depends on β by consulting α's cumulative dependencies. Given a dependence exists, α's cumulative dependencies are extended to include β's dependencies via a process we refer to as *strengthening*, such that the analysis may subsequently identify those instructions α is dependent on due to its dependence on β. If a dependence on β cannot be shown, the instructions are considered reorderable, subsequently requiring $rif_{\mathsf{a}}(\mathcal{R}, \mathcal{G}, \beta, \alpha)$ to be shown. Moreover, a process of *weakening* is necessary to remove α's cumulative dependencies that β may resolve due to forwarding.

To illustrate the evolving nature of cumulative dependencies, consider the sequence $\beta\,;\,\gamma\,;\,\alpha$ where $\gamma \not\hookleftarrow \alpha$ and $\beta \not\hookleftarrow \gamma$ but $\beta \hookleftarrow \alpha$. The analysis from the perspective of α starts at γ and identifies a dependence, due to $\gamma \not\hookleftarrow \alpha$. Therefore, α gains γ's dependencies via strengthening. The analysis progresses to the next instruction, β, for which a dependence can be established due to α's cumulative dependencies including $\beta \not\hookleftarrow \gamma$. Consequently, despite no direct dependency between α and β, the sequence does not produce reordering pairs for α. Repeating this process for γ and β ultimately finds no reordering pairs over the entire sequence, resulting in no rif_{a} checks.

A realistic implementation of this analysis is highly dependent on the language's reordering relation. In most examples, this relation only considers the variables accessed by the instructions and special case behaviours for memory barriers, as illustrated by the instantiation in Section 5. Consequently, cumulative dependencies can be efficiently represented as sets of such information, for example capturing the variables read by α and those instructions it depends on. This representation lends itself to efficient set-based manipulations for strengthening and weakening.

The analysis has been implemented for both a simple while language and an abstraction of ARMv8 assembly, with optimisations to improve precision in each context. In particular, precision can be improved through special handling of the forwarding case as the effects of forwarding typically result in trivial rif_a checks. The implementations have been encoded and verified in Isabelle/HOL, along with proofs of termination (following the approach suggested in [23]).

3.3.1 Address calculations. Dependence analysis is considerably more complex in the presence of address calculations. Under such conditions, it is not possible to syntactically identify whether two instructions access equivalent addresses, complicating an essential check to establishing dependence. Without sufficient aliasing information the analysis must over-approximate and consider the two addresses distinct, potentially introducing excess reordering pairs.

The precision of the analysis can be improved using an alias analysis to first identify equivalent address calculations, feeding such information into the dependency checks. Precision may also be improved by augmenting the interference check, rif_a, with any calculations that have been assumed to be distinct. For example, consider $[x] := e; [y] := f$, where $[v] := e$ represents a write to the memory address computed by the expression v. If an alias analysis cannot establish $x = y$, it is necessary to consider their interference. As they are assumed to reorder, a proof demonstrating $rif_a(\mathcal{R}, \mathcal{G}, [x] := e, [y] := f)$ can assume $x \neq y$. Such a property extends to any other comparisons with cumulative dependencies.

We have implemented such improvements in our analysis for ARMv8, relying on manual annotations to determine aliasing address calculations. These aliasing annotations are subsequently added to each instruction's verification condition to ensure they are sound.

3.4 Interference checking

Given the set of reordering pairs, it is necessary to demonstrate rif_a on each to demonstrate freedom of reordering interference. Many rif_a properties can be shown trivially. For example, if one instruction does not access shared memory, rif_a can be immediately shown to hold as no interference via $\mathcal{R}$ could take place. Additionally, if the two instructions access distinct variables and these variables are not related by $\mathcal{R}$, then no interference would be observed.

If these shortcuts do not hold, then it is necessary to consider rif_a directly. The property can be rephrased in terms of weakest precondition calculation [10], assisting automated verification.

3.5 Elimination of reordering interference

Step 3 of the process is intended to handle situations where rif_a cannot be shown for a particular pair of instructions. A variety of techniques can be applied in such conditions, depending on the overall verification goals. In some circumstances, a failure to establish rif_a indicates a problematic reordering such that the out-of-order execution of the instruction pair will violate any variation of the desired rely/guarantee reasoning. In such circumstances, it is necessary to prevent reordering through the introduction of a memory barrier.

As these barriers incur a performance penalty, this is not a suitable technique to correct all problematic pairs. Some reordering pairs can instead be resolved by demonstrating stronger properties during the standard rely/guarantee reasoning in step 4. We describe a series of techniques that can be employed to extract these stronger properties by modifying a program's verification conditions and/or abstracting over its behaviour. These techniques, while incomplete, are easily automated and cover the majority of cases.

3.5.1 Strengthening. Establishing rif_a may fail in cases where an instruction in a reordering pair modifies the other's verification condition. In such circumstances, it is possible to *strengthen* verification conditions such that the interference becomes benign by capturing both the in-order

and out-of-order execution behaviours. Given a reordering pair (β, α), this is achieved by first determining the weakest P that solves $P\{\alpha_{\langle\beta\rangle}; \beta\}(true)$, representing the implications of each instruction's verification conditions when executed out-of-order. This P is then used to strengthen β's verification condition, such that the stronger constraints are established during the standard rely/guarantee reasoning stage.

For example, consider the component $(y = 0)\{z := z + 1; x := y\}(true)$ where, due to a specialised analysis, the assignment to x has the verification condition $z = 1 \vee y = 0$ (and that for the assignment to z is $true$). Assume that $\mathcal{R}$ is the identity relation, i.e., no variables are changed by environment steps, and $\mathcal{G}$ is $true$. The rely/guarantee reasoning to establish this judgement is trivial, as Q is $true$ and $x := y$ will execute in a state where $y = 0$.

However, assuming the two assignments may be reordered, it is necessary to establish $rif_{\mathrm{a}}(\mathcal{R}, \mathcal{G}, z := z + 1, x := y)$. Unfortunately, such a property does not hold. For example, setting the pre-state of the program, P, to be $z = 0$ and the post-state, Q, to be $true$, we have $(z = 0)\{z := z + 1\}(z = 1) \wedge (z = 1)\{x := y\}(true)$ but not $\exists M'.\ (z = 0)\{x := y\}M' \wedge M'\{z := z + 1\}(true)$ since the verification condition of $x := y$ does not hold in the pre-state $z = 0$.

Applying the strengthening approach, we compute P for the out-of-order execution as $z = 1 \vee y = 0$. This predicate is then used as the verification condition for $z := z + 1$, which was originally $true$. With this strengthened verification condition, we have $rif_{\mathrm{a}}(\mathcal{R}, \mathcal{G}, z := z + 1, x := y)$ since $(z = 0)\{z := z + 1\}(z = 1) \wedge (z = 1)\{x := y\}(true)$ no longer holds.

With rif established, the standard rely/guarantee reasoning in step 4 must demonstrate $(y = 0)\{z := z + 1; x := y\}(true)$, with the strengthened verification condition for $z := z + 1$. This obviously holds given $y = 0$ initially.

3.5.2 Ignored reads. An additional issue when correcting for rif_{a} derives from the quantification of the pre- and post-states. This quantification reduces the proof burden, such that only pairs of reorderable instruction must be considered, but can introduce additional proof effort where the precise pre- and post-states are well known and limited reordering takes place. For instance, consider the simple component $(true)\{x := 1; z := y\}(x = 1)$ with a rely specification that will preserve the values of x and z always and the value of y given $x = 1$. The rely/guarantee reasoning to establish this judgement is trivial. However, the component will fail to demonstrate rif_{a} when considering the reordering of $x := 1$ and $z := y$, as their program order execution may establish the stronger $(true)\{x := 1; z := y\}(x = 1 \wedge z = y)$, whereas the reordered cannot.

We employ two techniques to amend such situations. The most trivial is a weakening of the component's $\mathcal{R}$ specification to remove the relationship between y and x, as it is unnecessary for the component's verification. Otherwise, if this is not possible, the component can be abstracted to $(true)\{x := 1; \mathsf{chaos}\ z\}(x = 1)$, where chaos v encodes a write of any value to the variable v. Consequently, the read of y is ignored. Both standard rely/guarantee reasoning and rif can be established for this modified component, subsequently enabling verification of the original via a refinement argument.

We propose the automatic detection of those reads that do not impact reasoning and, therefore, can be ignored when establishing rif. In general, such situations are rare as the analysis targets assembly code produced via compilation. Consequently, such unnecessary reads are eliminated via optimisation. Moreover, the $\mathcal{R}$ specification infrequently over-specifies constraints on the environment.

3.6 Soundness

Soundness of the proof system has been proven in Isabelle/HOL and is available in the accompanying theories at https://bitbucket.org/wmmif/wmm-rg. A proof sketch can be found in Appendix A.

4 NON-MULTICOPY ATOMIC WEAK MEMORY MODELS

Some modern hardware architectures, such as POWER and older versions of ARM, implement weaker memory models, referred to as non-multicopy atomic (NMCA), that cannot be fully characterised by a reordering relation. Under these architectures, a component's writes may become observable to other components at different points in time. Consequently, there is no shared state that all components agree on throughout execution, invalidating a core assumption of standard rely/guarantee reasoning. Moreover, such systems provide weak guarantees in terms of the cumulativity of writes [3]. For instance, a component may observe the effect of another component's instruction before writes that actually enabled the instruction's execution. This substantially complicates reasoning, as it results in behaviour that appears to execute out-of-order, invalidating a traditional notion of causality.

Building on the work of Colvin and Smith [7], we observe that the state of any pair of components can at most differ by writes from other components that the pair has inconsistently observed. Therefore, we propose a simple modification to the rules introduced in Section 2 to support reasoning under such memory models and comment on potential improvements to the approach's precision.

4.1 Write history semantics

Non-multicopy atomic behaviour can be modelled as an extension to the reordering semantics introduced in Section 3.1, as demonstrate by Colvin and Smith [7]. Under this extension, each component is associated with a unique identifier and the shared memory state is represented as a list of variable writes, i.e. $\langle w_1, w_2, w_3, \ldots \rangle$, with metadata to indicate which components have performed and observed particular writes. The order of events in this *write history* provides an overall order to the system's events, with those later in the list being the most recent. Each w_i is a write of the form $(x \mapsto v)^{wr}_{rds}$ assigning value v to variable x, with wr being the writer component's identifier and rds the set of component identifiers that have observed the write. We introduce the definitions $\mathsf{writer}((x \mapsto v)^{wr}_{rds}) = wr$, $\mathsf{readers}((x \mapsto v)^{wr}_{rds}) = rds$ and $\mathsf{var}((x \mapsto v)^{wr}_{rds}) = x$ to access metadata associated with a write.

To model the effects of instructions on the write history, it is necessary to associate each with the identifier for the component that executed them. Moreover, it is necessary to extract the instruction's effects in a form suitable for manipulation of the write history. To resolve these issues, we restrict the possible instructions that the language may execute over the global state to either store instructions of the form $(x := v)_i$, denoting a write to variable x of the constant value v from component i; load instructions of the form $[x = v]_i$, asserting the variable x must hold constant value v from the perspective of component i; memory barriers such as fence_i, corresponding to the execution of a fence by component i; or silent skip instructions, in which a component performs some internal step.

We refine the relation beh to model transitions over the write history for each of these instruction types. Modifications to the write history are constrained such that they may not invalidate variable coherence from the perspective of a component. For example, when component i executes the write instruction $x := v$, it must introduce a new write event for x with the written value of v and place it after all writes to x that i has observed and any writes that i has performed.

$$\begin{aligned} beh((x := v)_i) = \{(h \frown h', h \frown (x \mapsto v)^{i}_{\{i\}} \frown h') \mid \\ \forall w \in \mathsf{ran}(h') \cdot \mathsf{writer}(w) \neq i \wedge (\mathsf{var}(w) = x \implies i \notin \mathsf{readers}(w))\} \end{aligned} \tag{12}$$

Before such a write may be read by another component, the NMCA system must first *propagate* it from the writing component to the reading component. These transitions result in a component's

view of a variable x progressing to the next write to x that it has not yet observed. They are modelled as environment effects and can take place at any point during the execution. Moreover, they are only constrained to respect the order in which individual variables are modified, allowing components to observe writes to different variables in any arbitrary order. We define the set of possible propagations as follows.

$$\begin{aligned} prp \mathrel{\widehat{=}} \{(h \frown (x \mapsto v)^j_r \frown h', h \frown (x \mapsto v)^j_{r \cup \{i\}} \frown h') \mid \\ i \notin r \wedge \forall w \in \mathsf{ran}(h) \cdot \mathsf{var}(w) = x \implies i \in \mathsf{readers}(w)\} \end{aligned} \tag{13}$$

A component can access the value of a variable via the execution of a load instruction $[x = v]_i$. This read is constrained to the most recent write to x visible to component i, which must have written the value v. Additionally, memory barriers may constrain the write history, depending on the architecture. For instance, the fence_i instruction on ARM ensures that all components have observed the writes seen by component i. Finally, silent skip instructions are trivially defined as id over the write history.

$$\begin{aligned} beh([x = v]_i) = \{(h \frown (x \mapsto v)^j_r \frown h', h \frown (x \mapsto v)^j_r \frown h') \mid \\ \forall w \in \mathsf{ran}(h') \cdot \mathsf{var}(w) = x \implies i \notin \mathsf{readers}(w)\} \\ beh(\mathsf{fence}_i) = \{(h, h) \mid \forall w \in \mathsf{ran}(h) \cdot i \in \mathsf{readers}(w) \implies \forall y \cdot y \in \mathsf{readers}(w)\} \end{aligned} \tag{14}$$

Note that a component's writes can be perceived out-of-order even if they were considered ordered within the component's command, as the environment may decide to propagate writes to different variables arbitrarily. This can be perceived as a weakening of the reordering relation semantics, such that only instructions over the same variable are known to be ordered. Additionally, the propagation of a write from one component to another provides no constraint to relate the writes the destination and source components have both perceived, beyond the history of the written variable. Consequently, it is possible to propagate a write w_i from a source component to a destination component before the destination observes effects that enabled the execution of w_i to begin with.

To simplify specification and reasoning, we extend the language with a new constructor $comp(i, m, c)$, indicating a component with identifier i, local state m and command c. Moreover, we assume the specification of a local behaviour relation, $lbeh$, such that $(m, \alpha', m') \in lbeh(\alpha)$ denotes that the execution of α modifies the local state from m to m' and will result in the execution of α' in the shared state, where α' must be one of the shared memory instructions introduced above. Given these definitions, we can extract instructions over the shared state from transitions internal to a component and ensure appropriate annotation with the component identifier.

$$comp(i, m, c) \mapsto_{\alpha'_i} comp(i, m', c') \iff c \mapsto_\alpha c' \wedge (m, \alpha', m') \in lbeh(\alpha) \tag{15}$$

This structure is intended to capture the transition from local to global reasoning (as can be seen in Figure 3 in Section 4.3), with the constraint that systems are constructed as the parallel composition of a series of *comp* commands. Moreover, this structure enables trivial support for local state, such as hardware registers, and the partial evaluation of instructions via *lbeh*, such that they can be appropriately reduced to the shared memory instructions over which NMCA has been defined. For instance, an instruction $x := r_1 + r_2$, where r_1 and r_2 correspond to local state, could be partially evaluated to $x := v$ based on the values of r_1 and r_2 in m.

4.2 Reasoning under NMCA

We aim to quantify the implications of non-multicopy atomicity such that standard rely/guarantee reasoning may be preserved on these architectures. We first redefine the implications of a rely/guarantee judgement in the context of NMCA. To do so, we introduce the concept of a

view of the write history h for a set of components I. This view corresponds to the standard interpretation of shared memory, mapping variables to their current values. As there is no guarantee that all components in I will agree on the value a variable holds, we select the most recent write all components in I have observed, i.e., $\mathsf{view}_I(h, x)$ provides a value v for x such that

$$\mathsf{view}_I(h,x) = v \;\; \text{iff} \;\; h = h' \frown (x \mapsto v)^w_r \frown h'' \wedge I \subseteq r \wedge \forall w_i \in h'' \cdot \mathsf{var}(w_i) = x \implies I \not\subseteq \mathsf{readers}(w_i)$$

For brevity, we overload the case of a singleton set, such that $\mathsf{view}_i = \mathsf{view}_{\{i\}}$. Therefore, a judgement over a component i with command c of the form $\mathcal{R}, \mathcal{G} \vdash P\{c\}Q$ can be interpreted as constraints over the modifications to view_i throughout execution. Specifically, such a judgement encodes that for all executions of c, given the execution operates on the write history h such that $\mathsf{view}_i(h) \in P$ and all propagations to i modify view_i in accordance with $\mathcal{R}$, then i will modify view_i in accordance with $\mathcal{G}$ and, given termination, will end with a write history h' such that $\mathsf{view}_i(h') \in Q$.

This state mapping allows for rely/guarantee judgements over individual components to be trivially lifted from a standard memory model to their respective views of a write history. However, arguments for parallel composition are significantly more complex, as it is necessary to relate differing component views. Specifically, it is necessary to demonstrate that, given the execution of an instruction α by some component i satisfies its guarantee specification $\mathcal{G}_i$ in state h, formally $\mathsf{view}_i(h) \in sat(\alpha, \mathcal{G}_i)$, then the effects of propagating α's writes to some other component j will satisfy its rely specification $\mathcal{R}_j$ in its view, i.e., $\mathsf{view}_j(h) \in sat(\alpha, \mathcal{R}_j)$. Evidently, establishing such a notion of compatibility requires reasoning over the differences between the views of any arbitrary pair of components.

At a high level, we observe that it is possible to relate the views of two components by only considering the *difference* in their observed writes, i.e., the writes one component has observed but the other has not. When considering two components i and j, this difference manifests as two distinct sets of writes, those that i has observed but j has not and those that j has observed but i has not. Therefore, to successfully map $sat(\alpha, \mathcal{G}_i)$ from the view of component i to that of j, it is only necessary to consider the effects of these two sets of writes on $sat(\alpha, \mathcal{G}_i)$. Building on the ideas presented in Section 3.2, we frame the problem in terms of reordering by considering α's out-of-order execution with respect to these differing writes and establish a new notion of reordering interference freedom rif_{nmca}, such that $sat(\alpha, \mathcal{G}_i)$ must hold independent of any differing writes between i and j.

4.2.1 Relating a Pair of Views. We formally define the difference in observed writes between components given a write history h. To facilitate reasoning over these writes as a form of instruction reordering, the evaluated writes are converted back into instructions and composed via sequential composition. We define $\Delta_{i,j}(h)$ to perform such a conversion, returning a command consisting of all writes in h that i has observed but j has not. These writes are sequenced in the same order they appear in the write history h, therefore respecting any constraints such as variable coherence.

$$\Delta_{i,j}(h) \mathrel{\widehat{=}} \begin{cases} x := v;\ \Delta_{i,j}(h'') & \text{if} \;\; h = h' \frown (x \mapsto v)^w_r \frown h'' \;\wedge\; i \in r \;\wedge\; j \notin r \;\wedge \\ & \qquad \forall w_i \in h' \cdot i \in \mathsf{readers}(w_i) = j \in \mathsf{readers}(w_i) \\ \epsilon & \text{otherwise} \end{cases}$$

Note that $\Delta_{i,j}(h)$ consists only of instructions of the form $x := v$, where x is a shared variable and v is a constant value, as this reflects their representation in h. Moreover, $\Delta_{i,j}(h)$ cannot contain writes performed by component j, as it only contains writes j has not observed and j must have observed its own instructions.

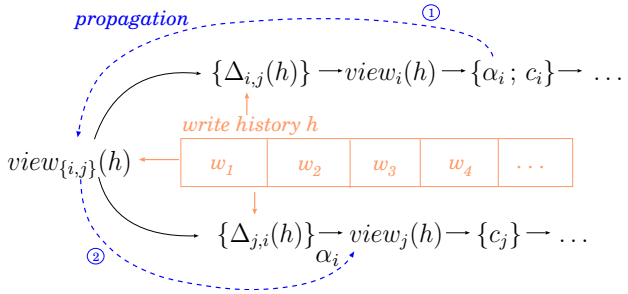

Fig. 2. Non-multicopy atomicity as reordering

We observe that the execution of the command $\Delta_{i,j}(h)$ with an initial state $\text{view}_{\{i,j\}}(h)$, i.e., the shared view of memory for components i and j, will terminate in the state $\text{view}_i(h)$. The final state must be $\text{view}_i(h)$, as this memory will only differ with $\text{view}_{\{i,j\}}(h)$ for some variable x if there is a write in h to x that i has observed but j has not. Therefore, this write must exist in $\Delta_{i,j}(h)$. A similar property holds from the perspective of j, such that the execution of the command $\Delta_{j,i}(h)$ with an initial state $\text{view}_{\{i,j\}}(h)$ will terminate in the state $\text{view}_j(h)$. Consequently, it is possible to relate the views of two components i and j via their respective Δs and their shared view of the write history, $\text{view}_{\{i,j\}}(h)$.

4.2.2 Reordering Before $\Delta_{i,j}$. Based on this relation between two component views, we aim to demonstrate rely/guarantee compatibility when propagating a write instruction α from component i to component j. Given component i must evaluate instruction α such that $\text{view}_i(h) \in sat(\alpha, \mathcal{G}_i)$, we first establish that α can be executed in the share view with j and it will still satisfy $\mathcal{G}_i$ in such a context, i.e., $\text{view}_{\{i,j\}}(h) \in sat(\alpha, \mathcal{G}_i)$. As these two views are related by the execution of the write sequence $\Delta_{i,j}(h)$, this property can be establish by considering the reordering of α before $\Delta_{i,j}(h)$ (see step ① in Figure 2).

The instruction α will be reorderable with all writes in the write sequence without changing the sequential semantics of their execution due to constraints imposed on propagation transitions (cf. definition (13)). Specifically, when propagating the effects of α from component i to j, component j must have already observed all prior writes to the variable α modifies. As j has observed these writes, they will not be present in $\Delta_{i,j}(h)$, resulting in α writing to a distinct variable with respect to the writes it must reorder with. Moreover, α must be of the form $(x := v)_i$ when propagation occurs, where v is a constant, and, therefore, it's behaviour must be independent of the writes in $\Delta_{i,j}(h)$.

We demonstrate $\text{view}_{\{i,j\}}(h) \in sat(\alpha, \mathcal{G}_i)$ via an induction over the write sequence $\Delta_{i,j}(h)$ in reverse, where $\text{view}_i(h) \in sat(\alpha, \mathcal{G}_i)$ represents the base case. Recall that the sequence cannot contain writes from j. Consequently it must consist of writes from i itself or components other than i and j. When considering a write from component i, the effects of propagating α earlier than this write are equivalent to the reordering behaviour introduced in Section 3.1 with a sufficiently relaxed reordering relation. We assume component i has been verified with a *rif* condition capturing such possible reorderings and exploit this condition to preserve $sat(\alpha, \mathcal{G}_i)$ across all instructions in the write sequence derived from i.

Next, we consider the effects of writes derived from components other than i and j. This case captures the main complication introduced by a NMCA system, such that i may have demonstrated $sat(\alpha, \mathcal{G}_i)$ based on writes that j has not yet observed. Therefore, the compatibility between i and j only holds if $sat(\alpha, \mathcal{G}_i)$ can be shown independently of these writes. We phrase this notion of

independence as rif_{nmca} and define it in terms of the weakest precondition of some relation $\mathcal{E}$ intended to capture the possible writes i may have observed ahead of j.

$$rif_{nmca}(\mathcal{E}, \alpha, \mathcal{G}_i) \mathrel{\widehat{=}} wp(\mathcal{E}, sat(\alpha, \mathcal{G}_i)) \subseteq sat(\alpha, \mathcal{G}_i)$$

This property captures that $sat(\alpha, \mathcal{G}_i)$ must hold prior to the execution of some transition $\mathcal{E}$ if it held after, preserving $sat(\alpha, \mathcal{G}_i)$ across those writes in $\Delta_{i,j}(h)$ from components k other than i and j given they satisfy $\mathcal{E}$. To derive a suitable $\mathcal{E}$, we observe that these writes must satisfy the specification $\mathcal{R}_i \cap \mathcal{R}_j$, given a similar overall compatibility argument between k and both i and j. Moreover, according to the constraints imposed by the propagation transition (as outlined above), these writes must not modify the variable written by α. We introduce the relation id_α denoting all state transitions in which the variable written by α does not change, capturing this constraint. Therefore, the property $rif_{nmca}(\mathcal{R}_i \cap \mathcal{R}_j \cap \mathrm{id}_\alpha, \alpha, \mathcal{G}_i)$ is sufficient to establish the induction proof and ultimately demonstrate $\mathrm{view}_{\{i,j\}}(h) \in sat(\alpha, \mathcal{G}_i)$.

4.2.3 Reordering After $\Delta_{j,i}$. With the execution of α established in the shared view such that it must satisfy $\mathcal{G}_i$, we consider its execution in view_j. Following the prior argument, these views are related by the command $\Delta_{j,i}(h)$, however, we now consider the preservation of a property after the execution of this command, modelled by reordering α after $\Delta_{j,i}(h)$.

When propagating α to component j (see step ② in Figure 2), it is possible that j may have observed a more recent write to the variable α modifies, where recent implies a later placement in the write history h. This can occur if component i placed α earlier in h than writes that j had already observed. As view_j maps each variable to its most recent write, j's view will not be modified by the propagation of α under such conditions, resulting in a trivial compatibility proof. Alternatively, if j has not observed a more recent write to the variable α modifies, then it must be trivially reorderable with the write sequence $\Delta_{j,i}(h)$ following the same argument as the prior section.

To preserve $sat(\alpha, \mathcal{G}_i)$ across $\Delta_{j,i}(h)$, we note that the write sequence must not contain writes derived from component i, as i must have observed its own writes. Therefore, all writes in $\Delta_{j,i}$ must satisfy $\mathcal{R}_i$, i.e., the constraint i imposes on writes derived from all other components. Moreover, the existing argument establishing $sat(\alpha, \mathcal{G}_i)$ must be stable under $\mathcal{R}_i$, as this is a requirement of standard rely/guarantee reasoning. Given the properties of stability, $sat(\alpha, \mathcal{G}_i)$ must therefore be preserved by the execution of $\Delta_{j,i}$, establishing $\mathrm{view}_j(h) \in sat(\alpha, \mathcal{G}_i)$. Finally, given compatibility between i and j such that $\mathcal{G}_i \subseteq \mathcal{R}_j$, the desired property $\mathrm{view}_j(h) \in sat(\alpha, \mathcal{R}_j)$ must hold via the monotonicity of sat.

Note that reordering α after $\Delta_{j,i}(h)$ reduces to existing proof obligations imposed by standard rely/guarantee reasoning. This can be attributed to its similarity with scheduling effects, as a scheduler may place an arbitrary number of instructions from other components between i's execution of α and j's subsequent observation of α's effects when considering a standard memory model. Consequently, rif_{nmca} is the only novel constraint imposed when considering a NMCA system.

4.3 NMCA Rules

We modify the rules [Comp] and [Par] introduced in Figure 1 to enforce NMCA compatibility conditions. To simplify modifications, we encode rif_{nmca} between components within the check of *compatibility*. First, we note that the standard rely/guarantee compatibility condition for two components i and j takes the form of $\mathcal{G}_i \subseteq \mathcal{R}_j$. This condition can be reinterpreted to consider the variable being modified as $\forall x, v \cdot sat(x := v, \mathcal{G}_i) \subseteq sat(x := v, \mathcal{R}_j)$, denoting that the conditions i guarantees will hold when executing $(x := v)_i$ imply the conditions j assumes to hold when it observes $(x := v)_i$.

Evidently, this reinterpretation of compatibility and rif_{nmca} can be combined based on the transitivity of $\subseteq$ to define our new notion of compatibility under NMCA, such that

$$\mathsf{compat}(\mathcal{G}_i, \mathcal{R}_i, \mathcal{R}_j) \quad \widehat{=} \quad \forall x, v \cdot wp(\mathcal{R}_i \cap \mathcal{R}_j \cap \mathsf{id}_x, sat(x := v, \mathcal{G}_i)) \subseteq sat(x := v, \mathcal{R}_j)$$

This notion of compatibility roughly denotes that i may have observed some additional writes from components other than j and its argument for compatibility with j must be independent of these writes. Note that $\mathcal{R}_i \cap \mathcal{R}_j \cap \mathsf{id}_x$ is reflexive and transitive, due to constraints on $\mathcal{R}$ specifications. Therefore, this property captures the execution in which i observes no additional writes, implying the original compatibility condition, as well as those with an arbitrary number of additional writes seen by i.

A modified rule for parallel composition limited to only two components would be updated to this new notion of compatibility as follows.

$$\frac{\mathcal{R}_1, \mathcal{G}_1 \vdash P_1\{c_1\}Q_1 \quad \mathcal{R}_2, \mathcal{G}_2 \vdash P_2\{c_2\}Q_2 \quad \mathsf{compat}(\mathcal{G}_1, \mathcal{R}_1, \mathcal{R}_2) \quad \mathsf{compat}(\mathcal{G}_2, \mathcal{R}_2, \mathcal{R}_1)}{\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2 \vdash P_1 \cap P_2\{c_1 \parallel c_2\}Q_1 \cap Q_2}$$

However, this approach is limited to two components, due to constraints in establishing compat. Observe that compat must be demonstrated over each pair-wise combination of components in a system, due to its dependence on their net environment specification ($\mathcal{R}_i \cap \mathcal{R}_j$). Consequently, it is necessary to know the rely/guarantee specification for each individual component within a judgement to successfully demonstrate compatibility with a new component. Unfortunately, the standard rule for parallel composition merges the individual component specifications in ($\mathcal{R}_i \cap \mathcal{R}_j$), allowing for more abstract reasoning but resulting in the loss of information necessary to establish the pair-wise compat (i.e., $\mathcal{R}_i$ and $\mathcal{R}_j$ are not accessible anymore).

We resolve this issue by retaining the necessary rely specification throughout reasoning. We modify $\mathcal{R}$ and $\mathcal{G}$ to partial maps, mapping identifiers of the sub-components to their original rely/guarantee specification (see Rule [Comp'] in Figure 3). The domain of the partial map corresponds to the sub-components the judgement operates over. We use the syntax $M(k)$ to represent accessing map M with key k and $[k \rightarrow v]$ to represent a new partial map, which returns v for key k. Moreover, we introduce operators over the partial map, such that $\mathsf{dom}(M)$, return the domain of the map M, corresponding to the identifiers it holds specifications for, $\mathsf{disjoint}(M, N)$ returns whether the maps M and N have disjoint domains (i.e., share any sub-components), and $M \uplus N$ combines two disjoint maps. The generalised rule [Par'] is shown in Figure 3. Note that we assert that the domains of the rely/guarantee specification for two parallel components must be disjoint, to enforce the uniqueness of identifiers.

The judgement $\mathcal{R}, \mathcal{G} \vdash P\{c\}Q$ can be interpreted such that for all executions of c commencing in a write history h, given for all i in $\mathsf{dom}(\mathcal{R})$, $\mathsf{view}_i(h) \in P$ and all propagations to i modify view_i in

$$[\text{Comp'}]\frac{\mathcal{R}, \mathcal{G} \vdash_c P\{c\}Q \quad rif(\mathcal{R}, \mathcal{G}, c)}{[i \rightarrow \mathcal{R}], [i \rightarrow \mathcal{G}] \vdash P\{comp(i, m, c)\}Q}$$

$$[\text{Par'}]\frac{\begin{array}{c}\mathcal{R}_1, \mathcal{G}_1 \vdash P_1\{c_1\}Q_1 \qquad \mathcal{R}_2, \mathcal{G}_2 \vdash P_2\{c_2\}Q_2 \qquad \mathsf{disjoint}(\mathcal{R}_1, \mathcal{R}_2) \\ \forall i \in \mathsf{dom}(\mathcal{R}_1) \cdot \forall j \in \mathsf{dom}(\mathcal{R}_2) \cdot \mathsf{compat}(\mathcal{G}_1(i), \mathcal{R}_1(i), \mathcal{R}_2(j)) \\ \forall i \in \mathsf{dom}(\mathcal{R}_2) \cdot \forall j \in \mathsf{dom}(\mathcal{R}_1) \cdot \mathsf{compat}(\mathcal{G}_2(i), \mathcal{R}_2(j), \mathcal{R}_1(j))\end{array}}{\mathcal{R}_1 \uplus \mathcal{R}_2, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash P_1 \cap P_2\{c_1 \parallel c_2\}Q_1 \cap Q_2}$$

Fig. 3. Proof rules for rely/guarantee reasoning under NMCA

accordance with $\mathcal{R}(i)$, then i will modify view_i in accordance with $\mathcal{G}(i)$ and, given termination, c will end with a write history h' such that $\text{view}_i(h') \in Q$.

4.4 Soundness

These rules have been encoded in Isabelle/HOL as an abstract theory, with a minimal instantiation for NMCA versions of the ARM architecture, and are available at https://bitbucket.org/wmmif/wmm-rg. Based on the work of Colvin and Smith [7], it should be possible to implement a similar instantiation for the POWER architecture.

4.5 Precision

This approach does not precisely capture the set of writes that may influence a component's execution without being fully propagated to all other components in the system. To reason about such a set, it is necessary to identify those variables that a component may read and their relative placement with fence instructions. For instance, writes may be propagated to a particular component ahead of others, however, if the component does not read these writes then they will not influence execution. Moreover, even if the component does read these propagated writes, it is possible to safely reason about the implications as long as any observable behaviour based on the read values occurs after a fence instruction. This is due to the fence's effect on the write history, ensuring that all other components observe the writes that the executing component has seen. We illustrate this issue via the following example.

Component 1:	**Component 2:**	**Component 3:**
$x := 1$	$r_1 := x$;	$r_2 := y$;
	$y := r_1$	fence;
		$r_3 := x$

Fig. 4. Non-multicopy atomicity example.

Component 1 writes 1 to x, while component 2 reads x and writes the resulting value to y. Finally, component 3 will read y followed by x. It is assumed that there is no reordering possible within these components, as component 3's instructions are ordered by a fence and component 2's instructions cannot be reordered without changing their behaviour. We assume that all variables hold 0 to begin with. Under a sequentially consistent memory model, it should be possible to establish that component 3 terminates in a state such that $r_2 = 1 \implies r_3 = 1$, as y will only hold the value 1 if 1 has been written to x earlier.

This reasoning is trivially preserved on a MCA system, as there are no reorderable instructions to consider, however, it fails to carry over to a NMCA system. On such a system, it is possible for the write $x := 1$ from component 1 to be propagated to component 2 before component 3. Component 2 is then able to read x and perform the write $y := 1$. If this write is then propagated to component 3 before the earlier write $x := 1$, component 3 can read a value of 1 for y and 0 for x, violating the desired post-condition.

It is possible to resolve this issue by introducing a fence instruction in component 2, between its read and subsequent write. Such a fence would ensure that component 3 must see the same or a later value for x relative to the value component 2 observed when executing $r_1 := x$. Therefore, if component 2 read a value of 1 for x, component 3 must also read a value of 1 after component 2 executes its added fence.

The approach introduced in Section 4.3 can capture the invalidation of reasoning, however, it is not sufficiently precise to detect that the introduced fence resolves the issue. Observe that, for rely/guarantee reasoning to establish the desired post-condition, component 3 must know that the write $y := 1$ will only occur in a state where $x = 1$. Therefore, $\mathcal{R}_3$ must be specified such that $sat(y := 1, \mathcal{R}_3) = (x = 1)$. As component 2 performs a write to y, it must guarantee a similar condition, such that $sat(y := 1, \mathcal{G}_2) = (x = 1)$. Moreover, the behaviour of component 1 cannot be more precise than $beh(x := 1)$, as this is its only instruction. Consequently, the compat condition must at least show $wp(beh(x := 1), x = 1) \subseteq (x = 1)$, when considering whether the guarantee of component 2 is compatible with the rely of component 3 in a system that includes effects from component 1. Evidently, this condition reduces to $True \subseteq x = 1$, which cannot be shown.

This same reasoning must be performed if a fence is inserted in to component 2, resulting in a failure to prove the desired post-condition $r_2 = 1 \implies r_3 = 1$. To correct this case, it would be necessary to identify the reads that can influence the behaviour of a write with no fence between them and then only consider the possible environment interference for effects on those reads. With such a technique, the case without a fence would still fail to show compatibility, while the case with a fence would be trivially shown, as there would be no reads that could influence $y := r_1$ without being propagated by the fence first.

We believe that such an extension to the technique is feasible, as a static analysis can identify the necessary reads via an approach similar to that suggested for reorderable instruction pairs in Section 3.3. We leave the implementation and verification of this approach to future work.

5 INSTANTIATING THE PROOF SYSTEM

In this section, we illustrate instantiating the proof system with a simple while language. The Isabelle/HOL theories accompanying this work also include an instantiation for ARMv8 assembly weakened to allow NMCA behaviour.

We distinguish three different types of state variables: global variables Glb and local variables Loc, which are program variables, and global auxiliary variables Aux. Local variables are unique to each component and cannot be modified by others.

Atomic instructions in our language comprise skips, assignments, guards, two kinds of fences, and coupling of an instruction with an auxiliary variable assignment and/or with a specific verification condition (similar to an assertion)

$$inst ::= \mathsf{nop} \mid v := e \mid \mathsf{guard}\ p \mid \mathsf{fence} \mid \mathsf{cfence} \mid \langle inst, a := e_a \rangle \mid \{\!|\, p_a \,|\!\}\, inst$$

where v is a program variable, e an expression over program variables, p a Boolean expression over program variables, a an auxiliary variable, e_a an expression over program and auxiliary variables, p_a a Boolean expression over program and auxiliary variables, and $\langle inst, a := e_a \rangle$ denotes the execution of $inst$ followed by the execution of $a := e_a$ atomically.

Commands are defined over atomic instructions and their combinations

$$cmd ::= inst \mid cmd\,;\, cmd \mid \mathsf{if}\ p\ \mathsf{then}\ cmd\ \mathsf{else}\ cmd \mid \mathsf{do}\ cmd\ \mathsf{while}(p, Inv)$$

where Inv denotes a loop invariant. Instructions instantiate individual instructions (i.e., α) in our abstract language. Sequential composition directly instantiates its abstract counterpart. Conditionals and loops are defined via the choice and iteration operator, i.e., if p then c_1 else c_2 is defined as $(\mathsf{guard}\ p)\,;\, c_1 \sqcap (\mathsf{guard}\ \neg p)\,;\, c_2$, and do c while(p, Inv) as $(c\,;\, (\mathsf{guard}\ p))^*\,;\, c\,;\, (\mathsf{guard}\ \neg p)$, where the invariant Inv holds at the start of c's execution.

A reordering relation $\overset{inst}{\hookleftarrow}$ (and its inverse $\overset{inst}{\not\hookleftarrow}$) is defined over atomic instructions based on syntactic independence of reorderable instruction [7]. For all instructions α and β

$$\begin{aligned}
&\text{fence} \overset{inst}{\not\hookleftarrow} \alpha, \qquad \alpha \overset{inst}{\not\hookleftarrow} \text{fence}, \qquad \text{guard } p \overset{inst}{\not\hookleftarrow} \text{cfence},\\
&\text{cfence} \overset{inst}{\not\hookleftarrow} \alpha \quad \text{if } rd(\alpha) \not\subseteq Loc,\\
&\text{guard } p \overset{inst}{\not\hookleftarrow} \alpha \quad \text{if } wr(\alpha) \in Glb \vee wr(\alpha) \in rd(\text{guard } p) \vee rd(\text{guard } p) \cap rd(\alpha) \not\subseteq Loc,
\end{aligned}$$

and for all other cases,

$$\beta \overset{inst}{\hookleftarrow} \alpha \quad \text{if } wr(\beta) \neq wr(\alpha) \wedge wr(\alpha) \notin rd(\beta) \wedge rd(\beta) \cap rd(\alpha) \subseteq Loc.$$

where $wr(\alpha)$ is the program variable written by α and $rd(\alpha)$ the program variables read by α. Note that a cfence is used to prevent speculative reads of global variables when placed prior to the reading instruction and after a guard [7].

Forwarding a value to an assignment instruction in our language is defined as $(v_\alpha := e_\alpha[v_\beta \backslash e_\beta]) \prec (v_\beta := e_\beta) \prec (v_\alpha := e_\alpha)$ and to a guard as $(\text{guard } p[v_\alpha \backslash e_\alpha]) \prec (v_\alpha := e_\alpha) \prec (\text{guard } p)$ where $e[v \backslash e']$ replaces every occurrence of v in e by e'. The instruction after forwarding carries the same verification condition as the original instruction, i.e., $vc(\alpha_{\langle\beta\rangle}) = vc(\alpha)$.

Note that auxiliary variables and verification conditions do not influence the reordering relation, as they will not constrain execution behaviour. Moreover, these annotations remain linked to their respective instructions during reordering and forwarding.

6 PETERSON'S MUTUAL EXCLUSION ALGORITHM

To demonstrate the workings of our technique for NMCA architectures requires a system with more than two components for this weaker memory model to have a possibly observable effect. We use the extension of Peterson's mutual exclusion algorithm which implements the behaviour of n components [25] each of which aims to get exclusive access to a critical section.

The proposed solution models $n-1$ *waiting rooms* through which the components have to advance before the critical section can be entered from the last. Each component p_i maintains its current waiting room in $level[p_i]$. Additionally, for each waiting room which component was the last to enter is monitored in variables $lastEnter[1], \ldots, lastEnter[n-1]$. This organises the components' advancement. These program variables are globally shared between the components and are accessible outside the critical section. A component can advance from one waiting room to the next if it is not the last to enter, or if it is the last to enter and no other component is in the same waiting room or a waiting room ahead (i.e, it is the first to advance to its current waiting room). The algorithm ensures that only two components can be present in the last waiting room and only the one that was not the last to enter can access the critical section which provides mutual exclusion.

The algorithm (shown in Figure 5) depicts one component p_1 instantiating this algorithm. The parameter e_1 is an auxiliary variable that does not affect the algorithm itself but is used during reasoning (see further details below). The critical section is represented by a placeholder in the figure, and fences have been added where required to guarantee mutual exclusion. The other components $p_2, \ldots p_n$ are encoded similarly.

In the algorithm the outer loop increments r over the $n-1$ waiting rooms the component has to pass through before it can enter the critical section. Within that loop it first records the room number the component is about to enter in $level[p_1]$. In a second step the appearance of the component in the room is notified by setting the component to be the last that has entered the room. Note that in the following we consider the component to have entered the room only after this second step. The auxiliary variable e_1 is updated twice to indicate when this entering phase is

```
Peterson(p_1, e_1, p_2, ... p_n) ≙ {
    var r = 1; reg_1 = 0; ... ; reg_n = 0;
    while (0 < r < n,                                        (* branch condition *)
           exitCond(p_1) ∧ level[p_1] = r − 1 ∧ ¬e_1 )       (* loop invariant *)
    {
      ⟨ level[p_1] := r,  e_1 := tt ⟩;
      fence;
      ⟨ lastEnter[r] := p_1,  e_1 := ff ⟩;
      fence;
       do
           reg_1 := lastEnter[r];
           reg_2 := level[p_2];
              ...
           reg_n := level[p_n];
       while( ⋁_{1<i≤n}(reg_i ≥ r) ∧ reg_1 = p_1,             (* branch condition *)
              room(p_1) = r ∧ ¬e_1 ∧ (lastEnter[r] = p_1 ∨ exitCond(p_1))   (* loop invariant*)
       r := r + 1;
    }
    {| exitCond(p_1) ∧ room_1 = n − 1 |} cfence;
    critical_section
}
```

Fig. 5. One component of Peterson's algorithm for n components with fences to guarantee correctness under weak memory

complete, i.e., in the first step *entering* is set to *true* and in the second step, when the component has fully entered, *entering* is completed and e_1 set to $false$.

The inner loop implements a busy wait in the current waiting room until the exit conditions for this room are met and the component can proceed to the next waiting room (i.e., no other component is ahead or one other component has entered after this one). As initial condition of the overall system we require that $\forall i \in C \cdot level[p_i] = 0 \wedge \neg e_i$, where C is the set of components.

In order to demonstrate our rely/guarantee reasoning, we define a rely condition for each component that is reflected by the other components' guarantee conditions, i.e., $\forall p_i \in C \cdot \mathcal{G}_i = \bigwedge_{j \cdot p_j \neq p_i} \mathcal{R}_j$. These conditions refer to the auxiliary variables e_i, for $0 < i \leq n$, which indicate for each component whether its current waiting room has been fully entered (i.e., p_i has set variable *lastEnter* for this room at some stage) . Furthermore, the following auxiliary functions are used:

- $room(p_i)$ determines the waiting room that component p_i has fully entered

$$room(p_i) \mathrel{\widehat{=}} \begin{cases} level[p_i] - 1 & \text{if } e_i \wedge level[p_i] > 0 \\ level[p_i] & \text{otherwise} \end{cases}$$

- $aheadOf(p_i)$ provides the number of components that component p_i is ahead of. Let $\#S$ denote the cardinality of set S.

$$aheadOf(p_i) \mathrel{\widehat{=}} \#\{\, p_j \mid room(p_i) > room(p_j) \vee (room(p_i) = room(p_j) \wedge p_i \neq p_j \wedge lastEnter[room(p_i)] = p_j \}$$

- $exitCond(p_i)$ formalises that the number of components p_i is ahead of is at least the same as the level of p_i's current waiting room. This constitutes (an abstraction of) the exit condition to each waiting room, e.g., in the last waiting room $n - 1$ component p_i needs to be ahead of

at least $n-1$ other components for its progress into the critical section to be enabled.

$$exitCond(p_i) \mathrel{\widehat{=}} aheadOf(p_i) \geq room(p_i)$$

The rely condition $\mathcal{R}_i$ for the component p_i can then be phrased as follows where $r \in \{1, \ldots, n-1\}$ ranges over the waiting rooms. Rely conditions for the other components are formalised and can be explained similarly.

$$\begin{aligned}
\mathcal{R}_i = \;& level[p_i] = level'[p_i] \;\wedge e_i = e_i' \wedge exitCond(p_i) = exitCond'(p_i) \wedge && \text{(i)}\\
& lastEnter'[room(p_i)] = p_i \implies lastEnter[room(p_i)] = p_i && \text{(ii)}\\
& lastEnter[room(p_i)] = p_i \wedge lastEnter'[room(p_i)] \neq p_i \implies exitCond'(p_i) && \text{(iii)}\\
& \forall p_j \in C \cdot room(p_j) < room(p_i) \wedge lastEnter'[room(p_i)] = p_i \implies \\
& \qquad\qquad room'(p_j) < room'(p_i) && \text{(iv)}
\end{aligned}$$

That is, $\mathcal{R}_i$ specifies that

(i) no other component modifies $level[p_i]$, e_i, or p_i's abstract exit condition, only component p_i itself can do so;
(ii) no other component can set variable $lastEnter$ of p_i's current waiting room to p_i;
(iii) if another component has entered p_i's current waiting room after p_i then the abstract exit condition needs to be maintained;
(iv) if p_i is ahead of component p_j (i.e., p_j is in a waiting room on a lower level) and p_i is the component that last entered its current waiting room after the environment step(s), then p_i will remain ahead of p_j.

Reasoning over the reordering interference freedom (*rif*) for p_1 showed where fence instructions were required to eliminate interferences caused by reorderings within the component (as indicated in Figure 5). For the body of the inner loop it is easy to see that executing the load instructions out of order does not have an effect on the exit condition of the loop, and can be considered benign. Contrary to that, an out-of-order execution of the two store instructions before the inner loop affects the coordination between p_1 and other components which ensures mutual exclusion. The condition $rif_{\mathsf{a}}(\mathcal{R}_1, \mathcal{G}_1, \beta, \alpha)$ for these two instructions could not be established, and a fence instruction was placed after each.

Additionally we have to show global reordering interference freedom (rif_{nmca}) by using the refined compatibility check between pairs of rely and guarantee conditions. Unfolding the definition given in Section 4.3 results in compatibility conditions for $i, j \in \{1, \ldots, n\}$ and $i \neq j$ that are of the form

$$\mathsf{compat}(\mathcal{G}_i, \mathcal{R}_i, \mathcal{R}_j) = wp(\mathcal{R}_i \cap \mathcal{R}_j \cap \mathsf{id}_x, sat(x := v, \mathcal{G}_i)) \subseteq sat(x := v, \mathcal{R}_j).$$

With $\mathcal{G}_i \implies \mathcal{R}_j$ and monotonicity of wp we deduce that it suffices to show that

$$wp(\mathcal{R}_i \cap \mathcal{R}_j \cap \mathsf{id}_x, sat(x := v, \mathcal{R}_j)) \subseteq sat(x := v, \mathcal{R}_j)$$

in order to prove $\mathsf{compat}(\mathcal{G}_i, \mathcal{R}_i, \mathcal{R}_j)$.

For example, let $x := v$ be $\langle lastEnter[n-1] := p_i; e_i := \mathsf{ff}\rangle$. That is, component p_i is entering the last waiting room $n-1$. In this case, to satisfy $\mathcal{R}_j$, in particular condition (iii), the exit condition of p_j, $exitCond(p_j)$, must be maintained if $room(p_j) = n-1$. Since no other component can modify $exitCond(p_j)$ this condition must have been satisfied before the step, i.e.,

$$sat(\langle lastEnter[n-1] := p_i; e_i := \mathsf{ff}\rangle, \mathcal{R}_j) = room(p_j) \;=\; n-1 \implies aheadOf(p_j) \geq room(p_j).$$

$\mathcal{R}_i \cap \mathcal{R}_j \cap \mathsf{id}_{lastEnter[n-1]}$ abstractly represents all behaviours of some component p_k (for $k \neq i$ and $k \neq j$) that do not modify $lastEnter[n-1]$. This reduces the steps of p_k to be considered to

only local steps and those that modify $level[p_k]$ or $lastEnter[m]$ for $m \neq n-1$. Although these steps might increase $room(p_k)$ it will remain lower than $room(p_j)$ and consequently $aheadOf(p_j)$ remains unaffected. Hence we have

$$wp(\mathcal{R}_i \cap \mathcal{R}_j \cap \mathrm{id}_{lastEnter[n-1]}, sat(\langle lastEnter[n-1] := p_i; e_i := \mathrm{ff}\rangle, \mathcal{R}_j)) \\ = sat(\langle lastEnter[n-1] := p_i; e_i := \mathrm{ff}\rangle, \mathcal{R}_j)$$

which proves compatibility of $\mathcal{G}_i$ and $\mathcal{R}_j$ for this instruction. We reason over all other instructions in a similar fashion.

7 CONCLUSION

This paper presents a truly thread-local approach to reasoning about concurrent code on a range of weak memory models. When considering multicopy atomic memory models, it employs standard rely/guarantee reasoning to handle interference between components, and a separate check of *reordering interference freedom* to handle interference within a component due to weak memory behaviour.

Reordering interference freedom provides evidence that the weak memory model under consideration will not invalidate properties shown via standard rely/guarantee reasoning. It is a novel concept that hinges on a thread-local reordering semantics which can be defined for any hardware architecture as it is based on the notion of instruction dependence, a core concept of processor pipelining.

Importantly, our approach reduces the check of reordering interference to only pairs of instructions, thereby significantly reducing its complexity. In situations where freedom of reordering interference cannot be shown, our approach includes methods to amend the program, to prohibit reordering behaviour, or modify its verification conditions, such that stronger arguments for reordering interference freedom may be shown.

When considering non-multicopy atomic memory models, the approach is extended via a simple modification to the rely/guarantee notion of component compatibility. This novel compatibility property identifies the conditions under which rely/guarantee reasoning between two components will not be invalidated by the inconsistent observation of writes from other components. Critically, this modification only alters the approach's rules when considering parallel composition, preserves the compositional nature of rely/guarantee reasoning and extends the approach to support all widely implemented hardware memory models.

The paper exemplifies an instantiation of the approach for a simple while language and NMCA memory model, and uses it to verify the mutual exclusion property of Peterson's algorithm extended to synchronise multiple components. These results, along with a soundness proof for our approach, have been encoded in Isabelle/HOL. In future work we intend to improve the precision of the techniques, addressing some of the concerns we raise, and improve tool support to ease verification.

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017. Context-Bounded Analysis for POWER. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017 (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10206. 56–74. https://doi.org/10.1007/978-3-662-54580-5_4

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 150:1–150:29. https://doi.org/10.1145/3360576

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, Thomas

Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[5] Hans-Juergen Boehm. 2012. Can seqlocks get along with programming language memory models?. In *Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '12*, Lixin Zhang and Onur Mutlu (Eds.). ACM, 12–20. https://doi.org/10.1145/2247684.2247688

[6] Robert J. Colvin. 2021. Parallelized Sequential Composition and Hardware Weak Memory Models. In *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings (Lecture Notes in Computer Science)*, Radu Calinescu and Corina S. Pasareanu (Eds.), Vol. 13085. Springer, 201–221.

[7] Robert J. Colvin and Graeme Smith. 2018. A Wide-Spectrum Language for Verification of Programs on Weak Memory Models. In *Formal Methods - 22nd International Symposium, FM 2018 (Lecture Notes in Computer Science)*, Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink (Eds.), Vol. 10951. Springer, 240–257. https://doi.org/10.1007/978-3-319-95582-7_14

[8] Nicholas Coughlin, Kirsten Winter, and Graeme Smith. 2021. Rely/guarantee reasoning for multicopy atomic weak memory models. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science)*, Vol. 13047. Springer, 292–310.

[9] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-Gries Reasoning for C11 RAR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:26.

[10] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, Heidelberg.

[11] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 608–621. https://doi.org/10.1145/2837614.2837615

[12] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Computer Aided Verification - 31st International Conference, CAV 2019 (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11561. Springer, 355–365. https://doi.org/10.1007/978-3-030-25540-4_19

[13] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[14] Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *IFIP Congress*. 321–332.

[15] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. http://dl.acm.org/citation.cfm?id=3009850

[16] G. A. Kildall. 1973. A unified approach to global program optimization. In *Proc. of POPL*. ACM, 194–206.

[17] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: persistency semantics for verification under ext4. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434324

[18] Markus Kusano and Chao Wang. 2017. Thread-modular static analysis for relaxed memory models. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 337–348. https://doi.org/10.1145/3106237.3106243

[19] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015 (Lecture Notes in Computer Science)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.), Vol. 9135. Springer, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

[20] N.M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. 2013. Correct and Efficient Work-stealing for Weak Memory Models. In *PPoPP '13*. ACM, 69–80. https://doi.org/10.1145/2442516.2442524

[21] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, James E. Burns and Yoram Moses (Eds.). ACM, 267–275. https://doi.org/10.1145/248052.248106

[22] Mark Moir and Nir Shavit. 2004. Concurrent Data Structures. In *Handbook of Data Structures and Applications.*, Dinesh P. Mehta and Sartaj Sahni (Eds.). Chapman and Hall/CRC. https://doi.org/10.1201/9781420035179.ch47

[23] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics - With Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

[25] Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12, 3 (1981), 115–116. https://doi.org/10.1016/0020-0190(81)90106-X

[26] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[27] Tom Ridge. 2010. A Rely-Guarantee Proof System for x86-TSO. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010 (Lecture Notes in Computer Science)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.), Vol. 6217. Springer, 55–70. https://doi.org/10.1007/978-3-642-15057-9_4

[28] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. https://doi.org/10.1145/1993498.1993520

[29] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

[30] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00346ED1V01Y201104CAC016

[31] Ketil Stølen. 1991. A Method for the Development of Totally Correct Shared-State Parallel Programs. In *CONCUR '91, 2nd International Conference on Concurrency Theory (Lecture Notes in Computer Science)*, Jos C. M. Baeten and Jan Friso Groote (Eds.), Vol. 527. Springer, 510–525.

[32] Thibault Suzanne and Antoine Miné. 2018. Relational Thread-Modular Abstract Interpretation Under Relaxed Memory Models. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018 (Lecture Notes in Computer Science)*, Sukyoung Ryu (Ed.), Vol. 11275. Springer, 109–128. https://doi.org/10.1007/978-3-030-02768-1_6

[33] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 691–707. https://doi.org/10.1145/2660193.2660243

[34] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 867–884. https://doi.org/10.1145/2509136.2509532

[35] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. 2016. *The RISC-V instruction set manual. Volume 1: User-level ISA, version 2.2.* Technical Report EECS-2016-118. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[36] Kirsten Winter, Nicholas Coughlin, and Graeme Smith. 2021. Backwards-directed information flow analysis for concurrent programs. In *IEEE Computer Security Foundations Symposium (CSF 2021)*. IEEE Computer Society.

[37] Qiwen Xu, Willem P. de Roever, and Jifeng He. 1997. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing* 9, 2 (1997), 149–174. https://doi.org/10.1007/BF01211617

A SOUNDNESS FOR THE PROOF SYSTEM UNDER MCA

We include a proof sketch establishing soundness of the system under a MCA memory model. This argument can be extended to NMCA memory models via the arguments presented in Section 4.2. To prove soundness of the proof system, we reason over a program's configuration traces. A configuration consists of a pair (c, σ), containing a command c to be executed and state σ (a mapping from variables to values) in which it executes. We denote the set of configurations by C. We define the computations of a program as sequences of configurations which are linked via program or environment steps, starting with the initial configuration.[1]

$$cp(c) \mathrel{\widehat{=}} \{t \in \mathrm{seq}(Configs) \mid \begin{array}{l} \exists \sigma_0.\ t_0 = (c, \sigma_0) \wedge \\ \forall i.\ 0 < i < len(t).\ (t_{i-1}, t_i) \in (\xrightarrow{ps} \cup \xrightarrow{es}) \end{array}\} \tag{16}$$

We define the properties of such computations below, where $t_i = (c_i, \sigma_i)$.

- t satisfies precondition P if the initial state of t satisfies P,
  $pre(t, P) \mathrel{\widehat{=}} \sigma_0 \in P$

[1]Note that $len(s)$ denotes the length of sequence s and s_i provides its i-th element.

- t adheres to rely condition $\mathcal{R}$ if all its environment steps adhere to $\mathcal{R}$,
  $rels(t, \mathcal{R}) \mathrel{\widehat{=}} (\forall i.0 < i < len(t).(t_{i-1}, t_i) \in \overset{es}{\longrightarrow} \implies (\sigma_{i-1}, \sigma_i) \in \mathcal{R})$.
- t adheres to guarantee $\mathcal{G}$ if all its program steps adhere to $\mathcal{G}$,
  $grs(t, G) \mathrel{\widehat{=}} (\forall i.0 < i < len(t).(t_{i-1}, t_i) \in \overset{ps}{\longrightarrow} \implies (\sigma_{i-1}, \sigma_i) \in G)$
- t satisfies post-condition Q if upon termination its final state satisfies Q,
  $post(t, Q) \mathrel{\widehat{=}} len(t) \in N \wedge c_{len(t)-1} = \epsilon \implies \sigma_{len(t)-1} \in Q$.

Validity of a judgement over a program, $\models c$ sats $[P, \mathcal{R}, \mathcal{G}, Q]$, states that for all computations of the program if the initial state satisfies P and every environment step adheres to $\mathcal{R}$ then the final state, if the computation terminates, satisfies Q and each program step adheres to $\mathcal{G}$.

$$\begin{aligned} \models c \text{ sats } [P, \mathcal{R}, \mathcal{G}, Q] \mathrel{\widehat{=}} \forall t \in cp(c).& \\ pre(t, P) \wedge rels(t, \mathcal{R}) \implies grs(t, \mathcal{G}) \wedge post(t, Q)& \end{aligned} \tag{17}$$

Soundness of the proof system requires that the rely/guarantee judgement implies its validity.

Theorem A.1. *Soundness*

$$\mathcal{R}, \mathcal{G} \vdash P\{c\}Q \implies \models c \text{ sats } [P, \mathcal{R}, \mathcal{G}, Q]$$

Proof sketch:
At the heart of the proof we have that if two sequentially composed instructions are deemed correct (via rely/guarantee judgements) and they can reorder and are reordering interference free, then the sequential composition of the reordered (and forwarded) instructions is correct. This follows straightforwardly from (9).

$$\begin{aligned} \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} P\{\beta\}M \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} M\{\alpha\}Q \wedge rif_{\mathrm{a}}(\mathcal{R}, \mathcal{G}, \beta, \alpha) \implies& \\ \exists M'.\, \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} P\{\alpha_{\langle\beta\rangle}\}M' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} M'\{\beta\}Q& \end{aligned} \tag{18}$$

This property extends to programs as follows. If an instruction, which can reorder to the start of a program is interference free in the program then correctness of the original program implies correctness of the executions in which the reordered instruction occurs first. This can be shown by structural induction using (18) and (10).

$$\begin{aligned} \mathcal{R}, \mathcal{G} \vdash_{\mathrm{c}} P\{r\}M \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} M\{\alpha\}Q \ \wedge\ rif_{\mathrm{c}}(\mathcal{R}, \mathcal{G}, r, \alpha) \implies& \\ \exists P', M'.\, P \subseteq P' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} P'\{\alpha_{\langle\langle r\rangle\rangle}\}M' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{c}} M'\{r\}Q& \end{aligned} \tag{19}$$

Note that we may weaken the precondition P of the original program to P' for the reordered program. This is to ensure all intermediate states in the reordered program (such as M') are stable in cases where conjuncts from P would lead to unstable conjuncts in intermediate states.

From (19) it follows that component level judgements over a program c are preserved across interference-free execution steps reordered to the start of c.

$$\begin{aligned} \mathcal{R}, \mathcal{G} \vdash_{\mathrm{c}} P\{c\}Q \wedge rif_{\mathrm{c}}(\mathcal{R}, \mathcal{G}, r, \alpha) \wedge c \mapsto_{\alpha_{\langle r\rangle}} c' \implies& \\ \exists P', M'.\, P \subseteq P' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} P'\{\alpha_{\langle\langle r\rangle\rangle}\}M' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{c}} M'\{c'\}Q& \end{aligned} \tag{20}$$

Property (20) straightforwardly extends to global rely/guarantee judgements. Note that reordering interference freedom is a component level condition and is hidden in the global rely/guarantee judgement $\vdash$. The global judgement can only be shown by lifting the component level judgement $\vdash_{\mathrm{c}}$ to the global level using proof rule [Comp] (see Figure 1) which includes $rif(\mathcal{R}, \mathcal{G}, c)$ as a precondition.

$$\begin{aligned} \mathcal{R}, \mathcal{G} \vdash P\{c\}Q \ \wedge\ c \mapsto_{\alpha_{\langle r\rangle}} c' \implies& \\ \exists P', M'.\, P \subseteq P' \wedge \mathcal{R}, \mathcal{G} \vdash_{\mathrm{a}} P'\{\alpha_{\langle\langle r\rangle\rangle}\}M' \wedge \mathcal{R}, \mathcal{G} \vdash M'\{c'\}Q& \end{aligned} \tag{21}$$

With (21) and the component and global proof rules (Figure 1) we can prove via induction over environment and program steps of each computation the validity of the global judgement. If the program is correct and for each computation of the program it holds that the precondition is

satisfied by the initial state and each environment step adheres to the rely condition, then the final state satisfies the post-condition (in the case when the computation terminates) and all program steps adhere to the guarantee condition.

$$t \in cp(c) \wedge \mathcal{R}, \mathcal{G} \vdash P\{c\}Q \wedge pre(t, P) \wedge rels(t, \mathcal{R}) \implies post(t, Q) \wedge grs(t, \mathcal{G})$$

Using (17) this proves the theorem.