Noname manuscript No.
(will be inserted by the editor)

RGITL: A Temporal Logic Framework for Compositional Reasoning about Interleaved Programs

Gerhard Schellhorn · Bogdan Tofan · Gidon Ernst · Jörg Pfähler · Wolfgang Reif

the date of receipt and acceptance should be inserted later

Abstract This paper gives a self-contained presentation of the temporal logic Rely-Guarantee Interval Temporal Logic (RGITL). The logic is based on interval temporal logic (ITL) and higher-order logic. It extends ITL with explicit interleaved programs and recursive procedures. Deduction is based on the principles of symbolic execution and induction, known from the verification of sequential programs, which are transferred to a concurrent setting with temporal logic. We include an interleaving operator with compositional semantics. As a consequence, the calculus permits proving decomposition theorems which reduce reasoning about an interleaved program to reasoning about individual threads. A central instance of such theorems are rely-guarantee (RG) rules, which decompose global safety properties. We show how the correctness of such rules can be formally derived in the calculus. Decomposition theorems for other global properties are also derivable, as we show for the important progress property of lock-freedom. RGITL is implemented in the interactive verification environment KIV. It has been used to mechanize various proofs of concurrent algorithms, mainly in the area of linearizable and lock-free algorithms.

1 Introduction

The design and verification of concurrent programs is much more difficult than it is for sequential programs. Two reasons contribute to this: the more complex control flow caused by scheduling and the fact that reasoning about initial and final states only (preconditions and postconditions) is no longer sufficient but must be extended to intermediate states.

Numerous specialized automatic methods such as decision procedures or abstract interpretation have been developed to verify specific system classes and properties automatically. While these techniques mainly succeed in proving rather

Institute for Software and Systems Engineering
University of Augsburg
E-mail: {schellhorn,tofan,ernst,joerg.pfaehler,reif}@informatik.uni-augsburg.de

simple properties, they often have two significant disadvantages: First, they typically do not provide insight about why a property is correct. Second, there is usually not much feedback when they fail. If there is an output, it is often hard to understand, in particular when programs are encoded as first-order specifications of transition systems with program counters (i.e., instruction pointers).

The alternative to specific automated proof techniques is interactive theorem proving. A key advantage is that expressive specification languages can be used and (readable) feedback for failed proof attempts can be provided. Of course, the price is that a much higher expertise with the tool is required. This paper describes such an approach that implements a temporal logic in the interactive theorem prover KIV [46] with the following goals:

- High-level verification of system designs with abstract programs and abstract (algebraic) data types, as opposed to verification using a fixed set of data types and a specific programming language.
- Readable proof goals with explicit programs that are not encoded as transition systems.
- Verification of sequential programs should be no more complicated using the extended, concurrent logic than using the weakest-precondition (wp) calculus already implemented in KIV [46].
- A wide range of correctness and progress properties of programs should be expressible and provable.
- Compositional proofs for interleaved programs, in particular rely-guarantee reasoning, should be supported.

Since most interactive provers are based on a variant of higher-order logic (HOL), a standard approach to defining complex (program) logics is by using an embedding into higher-order logic. For standard ITL such an embedding has been done in PVS [11]. However, these embeddings typically suffer from inefficiency since elementary functions like type checking or renaming of variables must be specified as formulas in the logic (resulting in proof obligations) rather than being implemented in the programming language of the theorem prover. Therefore the implementation in KIV is done natively, i.e., formulas and rules of the logic are directly represented as data structures of KIV's programming language. This solution has the drawback that the basic rules of the temporal logic calculus are validated against the semantics on paper only, which for the complex logic discussed here is a significant source of flaws. Therefore one contribution of this work is an additional specification in KIV of the syntax and semantics of RGITL based on HOL that closely follows the definitions given in this paper. The specification is accessible online [28]. It contains many low-level details (e.g., the definition of free variables, a coincidence lemma, a substitution lemma for renaming of variables) that we omit here for readability. Over this specification now almost all the axioms and rules given in this paper could be justified (we give pointers at the relevant places). The effort to do this validation of the logic was quite substantial (several person months) and led to a number of minor corrections in our presentation here.

This paper is organized as follows: Section 2 describes the syntax and semantics of the logic. Section 3 describes the basic calculus, which uses symbolic execution that is encoded as sequent calculus rules. Section 4 describes how induction can be done in the calculus. Section 5 outlines how RG rules are specified and derived.

Section 6 deals with liveness rules for interleaving. Section 7 describes an interesting application of RGITL, the derivation of a decomposition theorem for the global progress property "lock-freedom" [33]. The decomposition proof applies the central features of the logic: compositionality, symbolic execution, RG and liveness rules for interleaving. Section 8 describes related work and Section 9 contains concluding remarks.

This work is based on two earlier papers [5, 48]. An earlier version of extending ITL with shared-variable interleaved programs is described in [5]. There, the focus is on porting the well-known principle of symbolic execution of sequential programs [26, 10] to parallel programs with first-order logic. Sections 2 and 3 revise this approach by using higher-order instead of first-order logic as the basis, by using local instead of global frame assumptions and by using a weaker normal form ("step form") for symbolic execution. Section 3 also gives the result that the step form always exists for a class of formulas.

Closer to this paper than [5] is its shorter version [48], which is extended here, giving a self-contained description of the full framework. This paper includes rules for symbolic execution, a new scheme for lazy induction (Section 4) and a new rule for liveness induction for (unfair) interleaving (Section 6). Section 7 is entirely new. It formally defines the three major nonblocking-progress properties – wait-freedom, lock-freedom and obstruction-freedom – and details a practical decomposition theorem for lock-freedom.

This paper does not give applications of the logic to verify specific concurrent algorithms, although the calculus has been successfully used to verify a number of case studies. The focus was on verification of linearizability [21] and lock-freedom [33], where we managed to verify some significant examples that had had no mechanized proof before. Our linearizability proofs were based on a decomposition theorem for linearizability as described in [48]. While the proofs of such decomposition theorems are often quite complex and have many cases, the proof complexity of verifying case studies has been quite manageable. The main difficulty of concurrency proofs is finding *appropriate* theorems with suitable invariants and rely conditions.

Proving linearizability of the well-known (MS) queue algorithm of Michael and Scott [35] with RGITL is described by us in [6]. The benefit of using RGITL is that backward reasoning (or prophecy variables) can be avoided, in contrast to the automata-based approach of [14], since future states can be explored a priori in a temporal setting. In [50], we focus on the application of a weaker version of Theorem 7 to prove lock-freedom of the MS queue, while we detail the correctness arguments for this theorem here. [52] gives the first formal (and mechanized) proof of the central safety and liveness properties of a challenging lock-free stack that performs the garbage collection technique "hazard pointers". The work considers non-atomic read and write access to shared locations, which are specified as abstract procedures (also cf. [48]). In [51] we verify a simpler memory reclamation technique using ownership annotations and separation logic. Finally, [43] describes the application of RGITL to specify and verify self-healing behavior of organic computing systems, in particular to compositionally prove functional correctness of self-(re)organization in resource-flow systems. Further case studies can also be found online [29].

2 RGITL: Syntax and Semantics

This section introduces the logic RGITL. The first subsection motivates our design decisions. Section 2.2 discusses the syntactic elements and Section 2.3 defines the semantics. Since the compositional semantics of interleaving is quite complex (it integrates termination and blocking), we explain it separately in Section 2.4. Programs are introduced in Section 2.5. They are not part of the core language but are added as abbreviations (i.e., as a flat embedding). Finally, Section 2.6 briefly discusses specifications that in addition to a signature and axioms may contain procedure declarations.

2.1 Motivation

RGITL integrates ideas from RG reasoning [25] and ITL [12] in a higher-order logic setting (i.e., simply typed lambda calculus). From ITL the idea of having programs as formulas is adopted, using the chop operator ";" as a primitive to encode sequential composition. We also use the idea to have finite and infinite intervals (sequences of states) in the semantics that model terminating and non-terminating program runs.

From RG reasoning we take over the idea that interleaving should be compositional, i.e., proving properties of an interleaved program should be reducible to proving properties of its components with simple rules. We generalize this idea from rely-guarantee properties to arbitrary properties by proving that the following basic rule is derivable in RGITL.

Theorem 1 (Compositionality of Interleaving) *The compositionality rule*

$$\frac{\varphi_1 \vdash \psi_1 \qquad \varphi_2 \vdash \psi_2 \qquad \psi_1 \| \psi_2 \ \vdash \psi}{\varphi_1 \parallel \varphi_2 \vdash \psi}\ \textit{compositionality} \tag{1}$$

is sound for both weak fair and unfair interleaving.

Theorem 1 states that the global formula ψ can be derived for the interleaved system $\varphi_1 \parallel \varphi_2$ if local abstractions ψ_1 and ψ_2 can be derived for its components φ_1 and φ_2 (premises 1 and 2), and ψ can be derived for the interleaving of ψ_1 and ψ_2 (premise 3).

A similar theorem is known to hold about the chop operator (sequential composition) and the star operator (iteration) of standard ITL. To prove the rule above correct, it is necessary to define a semantics for programs that allows environment steps between program steps. When programs are interleaved, environments steps of one program can then be identified with the program steps of the other. RGITL therefore modifies the semantics of standard ITL to alternate between program and environment steps. The semantics of interleaving can then be defined based on interleaving individual runs (i.e., intervals). With such a semantics, the proof for Theorem 1 is simple.

Theorem 1 (together with the compositionality rules for chop and star) is used as a basic rule of our calculus. It allows to derive various decomposition theorems, and we will prove two instances as applications of the theorem. The first application of the theorem will be to derive rely-guarantee rules for interleaved programs (cf.

Theorems 5 and 6 proved in Section 5). To demonstrate that the theorem is more powerful than just deriving decomposition theorems for safety properties, Section 7 will show that a decomposition theorem for the difficult liveness property of lock-freedom is derivable too.

2.2 Syntax

A higher-order signature $SIG = (S, OP, Proc)$ consists of three finite sets:

- The set S of sorts, with $bool \in S$, is used to define the set of types T as the least set that includes all sorts and all function types $\underline{t} \to t$, where $t \in T$ and $\underline{t} = t_1, \ldots, t_n$ with $t_1, \ldots, t_n \in T$.
- The set OP consists of typed operators $op : t$, including the usual boolean operators, e.g., $\mathrm{true}, \mathrm{false} : bool$ and $\vee : bool \times bool \to bool$ (written infix as usual).
- The set $Proc$ contains typed procedure names $proc : \underline{t_1}; \underline{t_2}$, where the vector $\underline{t_1}$ indicates the types of input (or call by value) parameters and $\underline{t_2}$ indicates the types of input/output (or call by reference) parameters.

Temporal logic expressions e are defined over a signature SIG, dynamic (flexible) variables $x, y, z \in X$ and static variables $u \in U$. Flexible variables are typically used as program variables, which change during the program's execution. Static variables do not change their value during execution and are typically used to capture the value of a flexible variable in some previous state. In concrete formulas, we follow the KIV convention to use uppercase names for flexible and lowercase names for static variables. An arbitrary variable is written $v \in X \cup U$.

$$
\begin{aligned}
e ::= {} & u \mid x \mid x' \mid x'' \mid op \mid e(\underline{e}) \mid e_1 = e_2 \mid \lambda\, \underline{u}.\ e \mid \forall\, \underline{v}.\ \varphi \mid \\
& \varphi_1 \ \mathbf{until}\ \varphi_2 \mid \varphi_1; \varphi_2 \mid \varphi^* \mid \mathbf{A}\,\varphi \mid \mathbf{last} \mid \circ\,\varphi \mid \mathbf{blocked} \mid \\
& proc(\underline{e}; \underline{x}) \mid \varphi_1 \| \varphi_2 \mid \varphi_1 \|_{\mathrm{nf}} \varphi_2 \ .
\end{aligned}
$$

As usual in higher-order logic, expressions e of type $bool$ are formulas, denoted as φ. Expressions must satisfy standard typing constraints, e.g., in $e(\underline{e})$ the type of e must be a function type with argument types equal to the types of the arguments $\underline{e}$. The parameters of lambda expressions and quantifiers must all be different variables. Using the constructs in the first line exclusively gives *higher-order expressions*, which do not involve temporal logic. In general expressions, however, the operators may be mixed freely, e.g., temporal operators may appear within the scope of quantifiers. Dynamic variables can be primed and double primed. Note that the result of priming a variable is an expression, not another variable. Therefore the set of free variables of x' is $\{x\}$, and renaming x to y in x' gives y'. Lambda expressions can bind static variables only, while quantifiers can be used with both static and dynamic variables.

As in ITL, the chop operator $\varphi_1; \varphi_2$ combines two formulas sequentially. The (chop-)star operator φ^* denotes finite (possibly zero) or infinite iteration of φ. Universal path quantification is denoted as $\mathbf{A}\,\varphi$. Formula **last** characterizes termination. The strong next operator $\circ\,\varphi$ asserts that there is a next state which satisfies φ. Formula **blocked** indicates blocked steps, as explained below. Formulas $\varphi_1 \| \varphi_2$ and $\varphi_1 \|_{\mathrm{nf}} \varphi_2$ will be explained in Section 2.4 and denote weak-fair and arbitrary (unfair, i.e., not necessarily weak-fair) interleaving of φ_1 and φ_2, respectively.

Procedure calls have the form $proc(\underline{e}; \underline{x})$ with $proc : \underline{t_1}; \underline{t_2} \in Proc$. The expressions $\underline{e}$ must be of types $\underline{t_1}$, and $\underline{x}$ must be pairwise distinct dynamic variables of types $\underline{t_2}$.

By convention, temporal operators bind more strongly than propositional connectives, which have precedence $\neg \succ \wedge \succ \vee \succ \rightarrow \succ \leftrightarrow$. Quantifiers bind as far to the right as possible. Free variables $\mathrm{free}(e)$ are defined as usual. Higher-order expressions $e(\underline{x})$ without primed and double primed variables are called *state expressions*, and *state formulas* are state expressions of type *bool*. A state expression without flexible variables (neither free nor bound) is called *static*. In the case of a boolean expression, it is a *static formula*.

2.3 Semantics

The semantics of a signature SIG is a SIG-algebra $\mathcal{A}$, which defines a nonempty carrier set A_s as the semantics of every sort s. The set A_{bool} is always $\{\mathbf{tt}, \mathbf{ff}\}$. The semantics of a function type is the set of all functions of that type. An operator symbol $op : t$ is interpreted as a total function $op^{\mathcal{A}} \in A_t$. The predefined boolean operators have standard semantics.

The semantics of expressions $[\![e]\!](I)$ is defined relative to a SIG-algebra $\mathcal{A}$ (left implicit as a parameter) and an *interval* I. An interval is a finite or infinite sequence of the form $I = (I(0), I(0)_b, I'(0), I(1), I(1)_b, I'(1), \ldots)$, where every $I(k)$ and $I'(k)$ is a state function that maps static and dynamic variables to values, and $I(k)_b$ is a boolean flag that denotes a blocked transition. Static variables do not change between states. Finite intervals with length $\#I = n$ have $2n + 1$ states (and n boolean flags) and end at the unprimed state $I(n)$. The empty interval with one state $I(0)$ has length zero. Infinite intervals have $\#I = \infty$. For an interval I and $m \leq n \leq \#I$, $I_{[m..n]}$ denotes the subinterval from $I(m)$ to $I(n)$ inclusive. $I_{[n..]}$ is the postfix starting with $I(n)$. Transitions from $I(k)$ to $I'(k)$ are called system (or program) steps. Transitions from a primed to the subsequent unprimed state are environment steps. Thus, the interval semantics alternates between system and environment transitions. The k-th system step is *blocked* whenever the boolean flag $I(k)_b$ is $\mathbf{tt}$. In this case, the system transition stutters, i.e., $I(k) = I'(k)$.

The semantics $[\![e]\!](I)$ of an expression e of type t w.r.t. an interval I is an element of A_t. In particular, a formula φ evaluates to $\mathbf{ff}$ or $\mathbf{tt}$. In the latter case we write $I \models \varphi$ (φ holds over I). A formula is valid, written $\models \varphi$, if it holds for all I.

Unprimed variables are evaluated over the first state, i.e., $[\![v]\!](I) = I(0)(v)$. Primed and double primed variables x' and x'' are evaluated over $I'(0)$ and $I(1)$ respectively, if the interval is nonempty. For an empty interval, both x' and x'' are evaluated over $I(0)$ by convention. Operators receive their semantics from the algebra, i.e., $[\![op]\!](I) = op^{\mathcal{A}}$.

The semantics of quantifiers is defined using *value sequences* $\sigma = (\sigma(0), \sigma'(0), \ldots)$ for a vector $\underline{v}$ of variables. Each $\sigma(i)$ and $\sigma'(i)$ is a tuple of values of the same types as $\underline{v}$. If some v_k is a static variable, then all values $\sigma(i)_k$ and $\sigma'(i)_k$ for that variable have to be identical. The value sequence for $\underline{x}$ in I is written $I(\underline{x})$, and the

modified interval $I[\underline{v} \leftarrow \sigma]$ maps $\underline{v}$ in each state to the corresponding values in σ, when $\#\sigma = \#I$. [1] Similarly, $I[\underline{u} \leftarrow \underline{a}]$ modifies static variables $\underline{u}$ to values $\underline{a}$.

The semantics $proc^{\mathcal{A}}$ of a procedure $proc : \underline{t_1}; \underline{t_2}$ is part of the algebra $\mathcal{A}$ and consists of a set of pairs $(\underline{a}, \sigma)$. Each pair describes a potential run of the procedure: $\underline{a}$ is a vector of initial values for the input parameters from the carrier sets of types $\underline{t_1}$. The value sequence σ exhibits how the reference parameters change in each step. Note that this semantics implies that procedures modify only variables which are on their parameter list, and that input parameters work like local variables. Changes to these parameters while the procedure is executing are not globally visible.

The semantics of a tuple of expressions $\underline{e}$ is the tuple of semantic values for the member expressions. With these prerequisites, the semantics of the remaining expressions, except interleaving, is defined as follows:

$$
\begin{aligned}
[\![e(\underline{e})]\!](I) &\equiv [\![e]\!](I)([\![\underline{e}]\!](I)) \\
[\![\lambda\ \underline{u}.\ e]\!](I) &\equiv \underline{a} \mapsto [\![e]\!](I[\underline{u} \leftarrow \underline{a}]) \\
I \models e_1 = e_2 &\text{ iff } [\![e_1]\!](I) = [\![e_2]\!](I) \\
I \models \forall\ \underline{v}.\ \varphi &\text{ iff for all } \sigma,\ \#\sigma = \#I : I[\underline{v} \leftarrow \sigma] \models \varphi \\
I \models \textbf{last} &\text{ iff } \#I = 0 \\
I \models \textbf{blocked} &\text{ iff } \#I > 0 \text{ and } I(0)_b \\
I \models \circ\ \varphi &\text{ iff } \#I > 0 \text{ and } I_{[1..]} \models \varphi \\
I \models \varphi_1\ \textbf{until}\ \varphi_2 &\text{ iff there is } n \leq \#I \text{ with } I_{[n..]} \models \varphi_2 \text{ and for all } m < n : I_{[m..]} \models \varphi_1 \\
I \models \mathbf{A}\,\varphi &\text{ iff for all intervals } J \text{ with } J(0) = I(0) : J \models \varphi \\
I \models \varphi_1; \varphi_2 &\text{ iff either } \#I = \infty \text{ and } I \models \varphi_1 \text{ or there is } n \leq \#I, n \neq \infty \text{ with} \\
&\quad I_{[0..n]} \models \varphi_1 \text{ and } I_{[n..]} \models \varphi_2 \\
I \models \varphi^* &\text{ iff either } \#I = 0 \text{ or there is a sequence } \nu = (n_0, n_1, \ldots), \text{ such that} \\
&\quad n_0 = 0 \text{ and for } i + 1 < \#\nu : n_i < n_{i+1} \leq \#I \text{ and } I_{[n_i..n_{i+1}]} \models \varphi. \\
&\quad \text{Additionally, when } \#\nu < \infty : I_{[n_{\#\nu-1}..]} \models \varphi \\
I \models proc(\underline{e}; \underline{x}) &\text{ iff } ([\![\underline{e}]\!](I), I(\underline{x})) \in proc^{\mathcal{A}}\ .
\end{aligned}
$$

The interval semantics of most expressions needs no further explanation. Note that the semantics of a state expression depends on the first state $I(0)$ only.

The semantics of the chop operator ";" corresponds to the sequential composition of intervals: finite intervals of φ_1 are combined with (possibly infinite) intervals of φ_2 in series, and infinite intervals of φ_1 are also acceptable. The star operator chops the interval into finitely or infinitely many non-empty parts $I_{[0..n_1]}, I_{[n_1..n_2]}, \ldots$, which satisfy φ respectively (the last part is infinite if the split is finite and the interval infinite).

The following operators are frequently used abbreviations:

$$
\begin{array}{lll}
\exists\ \underline{v}.\ \varphi \equiv \neg\ \forall\ \underline{v}.\ \neg\ \varphi & \mathbf{E}\ \varphi \equiv \neg\ \mathbf{A}\ \neg\ \varphi & \Diamond\ \varphi \equiv \text{true}\ \textbf{until}\ \varphi \\
\Box\ \varphi \equiv \neg\ \Diamond\ \neg\ \varphi & \textbf{step} \equiv \circ\ \textbf{last} & \bullet\ \varphi \equiv \neg\ \circ\ \neg\ \varphi \\
\textbf{inf} \equiv \Box\ \neg\ \textbf{last} & \textbf{fin} \equiv \neg\ \textbf{inf}\ . &
\end{array}
$$

An interval of length one is characterized by formula **step**, finite and infinite intervals are distinguished by **fin** and **inf**. The formula $\bullet\ \varphi$ (weak next) states that either the interval is empty or φ holds after the next environment step.

[1] Tacitly, we also assume that $\sigma(k) = \sigma'(k)$, when $I(k)_b$.

2.4 Compositional Interleaving

This section defines the semantics of interleaving two formulas. The semantics is based on interleaving two intervals I_1 and I_2, which is a rather complex operation. Therefore we illustrate it with a simple example.

The semantics of $\varphi_1 \parallel \varphi_2$ and of $\varphi_1 \parallel_{\text{nf}} \varphi_2$ simply interleaves the intervals that are in the semantics of φ_1 and φ_2. Therefore the soundness proof of Theorem 1, which we give at the end of this section, is quite simple.

To discern fair from unfair interleaving, we define the set of *scheduled interleavings* of $I_1 \oplus I_2$. Each element of this set is a pair (I, sc) of a resulting interval I and a schedule. A schedule $sc = (sc(0), sc(1), \ldots)$ is a finite or infinite sequence, where each $sc(i)$ is either 1 or 2, indicating which of the two interleaved components is scheduled for execution. We denote the postfix of sc starting with $sc(n)$ as $sc_{[n..]}$ and the length of the schedule with $\#sc$. A schedule is *fair* if it is either finite, or infinitely often changes the selected component.

$$I_1 \parallel I_2 \equiv \{I : \text{there is a fair schedule } sc \text{ such that } (I, sc) \in I_1 \oplus I_2\}$$
$$I_1 \parallel_{\text{nf}} I_2 \equiv \{I : \text{there is a schedule } sc \text{ such that } (I, sc) \in I_1 \oplus I_2\} .$$

Members of $I_1 \oplus I_2$, which are finite or infinite intervals, are defined stepwise by considering 6 cases. The following definition gives the first three, where a step of I_1 is attempted. The other three cases are symmetric.

Definition 1 (Left Interleaving of Intervals)

1. The first component terminates in the current state, i.e., I_1 is empty. If $I_1(0) = I_2(0)$, then $\{(I_2, ())\}$ with an empty schedule $sc = ()$ is returned. Otherwise, interleaving is not possible and the empty set is returned.
2. The first step of component 1 is not blocked, i.e., $I_1(0)_b$ is **ff**. Then its first program transition is executed, and the system continues with interleaving the rest of interval I_1 with I_2. The set of all pairs (I, sc) is returned, where $I(0) = I_1(0)$, $I'(0) = I_1'(0)$, $I(0)_b = \mathbf{ff}$, $sc(0) = 1$ and $(I_{[1..]}, sc_{[1..]}) \in I_{1\,[1..]} \oplus I_2$.
3. The first component is blocked in the current state. If I_2 has terminated, then the result would be a duplicate of the first case but with I_1 and I_2 exchanged. To avoid duplicate cases, no result is returned. Otherwise, $I_1(0) = I_2(0)$ must hold in order to have any results, and a transition of the second component is taken, even though the first is scheduled. The resulting pairs (I, sc) have $sc(0) = 1$, $I(0) = I_2(0)$, $I'(0) = I_2'(0)$, and $(I_{[1..]}, sc_{[1..]})$ must be in $I_{1\,[1..]} \oplus I_{2\,[1..]}$. Both transitions are consumed and the overall transition is blocked ($I(0)_b$ holds) iff the first transition of I_2 is blocked too.

The schedule in Definition 1 ends as soon as either I_1 or I_2 terminates and, therefore, it may be shorter than the resulting interleaved interval I.

Example As an example of interleaving two intervals, consider two unblocked one-step intervals

$$I_1 = (I_1(0), \mathbf{ff}, I_1'(0), I_1(1)) \text{ and } I_2 = (I_2(0), \mathbf{ff}, I_2'(0), I_2(1)) ,$$

with a schedule $sc = (1, 2)$. The interleaved result then is

$$I = (I_1(0), \mathbf{ff}, I_1'(0), I_2(0), \mathbf{ff}, I_2'(0), I_1(1))$$

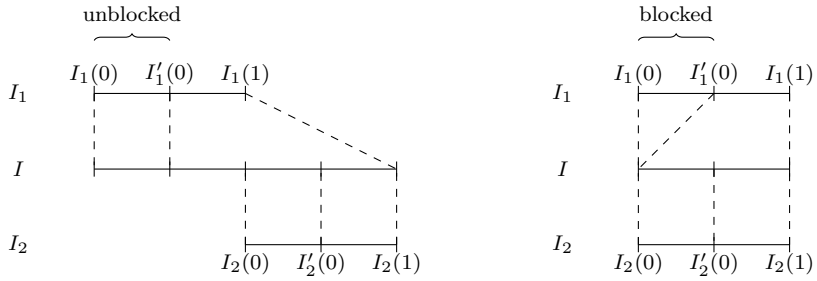

Fig. 1 Examples of interleaving two intervals

when $I_2(1) = I_1(1)$. Otherwise, interleaving is not possible. The local environment step from $I_1'(0)$ to $I_1(1)$ is mapped to the sequence $(I_1'(0), I_2(0), I_2'(0), I_1(1))$ in the result as shown in Fig. 1 (left), corresponding to the intuition that the environment steps of one component consist of alternating sequences of global environment steps and steps of the other component. Environment assumptions for one component must therefore be satisfied by such sequences.

As a variation, consider the case where the first step of I_1 is blocked, i.e., $I_1(0)_b = \mathbf{tt}$ and $I_1'(0) = I_1(0)$. Assume the schedule starts with the first interval. Since its first step is blocked, the steps of both programs will be executed, as shown in Fig. 1 (right). This is possible only if $I_2(0) = I_1(0)$. The first step of I will then be identical to the first step of I_2. The remaining intervals both have length zero. Interleaving is therefore possible only if $I_1(1) = I_2(1)$, according to case 1 above. Taken together, a scheduled interleaving that starts with $sc(0) = 1$ is possible only when $I_1(0) = I_2(0)$ and $I_1(1) = I_2(1)$. The result is then $sc = (1)$ and $I = I_2$.

Finally, the compositional semantics of interleaving two formulas φ_1 and φ_2 results from interleaving intervals as follows.

$$I \models \varphi_1 \parallel \varphi_2 \quad \text{iff there are } I_1, I_2 : \quad I_1 \models \varphi_1 \text{ and } I_2 \models \varphi_2 \text{ and } I \in I_1 \parallel I_2$$
$$I \models \varphi_1 \parallel_{\mathrm{nf}} \varphi_2 \quad \text{iff there are } I_1, I_2 : \quad I_1 \models \varphi_1 \text{ and } I_2 \models \varphi_2 \text{ and } I \in I_1 \parallel_{\mathrm{nf}} I_2 \,.$$

Based on this definition, the soundness proof of the compositionality rule (1) is simple.

Proof (of Theorem 1) For validity of the conclusion, we have to show that precondition $I \models \varphi_1 \parallel \varphi_2$ implies $I \models \psi$. Applying the definition above to the precondition gives intervals I_1, I_2 and a (possibly fair) schedule sc, such that $I_1 \models \varphi_1$, $I_2 \models \varphi_2$ and $(I, sc) \in I_1 \oplus I_2$. The first two premises therefore imply $I_1 \models \psi_1$, $I_2 \models \psi_2$, so $I \models \psi_1 \parallel \psi_2$ holds by the definition of the semantics. Finally, the last premise implies $I \models \psi$, as desired. □

2.5 Programs

The syntax of a program α is as follows:

$$\begin{aligned}\alpha ::= {} & \mathbf{skip} \mid \underline{z} := \underline{e} \mid \mathbf{if}^*\varphi \mathbf{\ then\ } \alpha_1 \mathbf{\ else\ } \alpha_2 \mid \mathbf{if\ } \varphi \mathbf{\ then\ } \alpha_1 \mathbf{\ else\ } \alpha_2 \mid \mathbf{while}^*\ \varphi \mathbf{\ do\ } \alpha \mid \\ & \mathbf{while\ } \varphi \mathbf{\ do\ } \alpha \mid \mathbf{await\ } \varphi \mid \mathbf{let\ } \underline{z} = \underline{e} \mathbf{\ in\ } \alpha \mid \mathbf{choose\ } \underline{z} \mathbf{\ with\ } \varphi \mathbf{\ in\ } \alpha_1 \mathbf{\ ifnone\ } \alpha_2 \mid \\ & \alpha_1; \alpha_2 \mid \alpha^* \mid \mathit{proc}(\underline{e}; \underline{x}) \mid \alpha_1 \parallel \alpha_2 \mid \alpha_1 \parallel_{\mathrm{nf}} \alpha_2 \mid \varphi \,.\end{aligned}$$

The program **skip** executes a stutter step. A (parallel) assignment has the form $\underline{z} := \underline{e}$. The difference between **if*** and **if** (likewise for **while*** and **while**) is that the latter executes its test in a separate system step. The former version can be used to model instructions such as compare-and-set (CAS), which executes both a test and an assignment in one atomic step. The operator **await** φ *blocks* the executing process as long as condition φ is not satisfied. It can be used for synchronization in parallel programs that communicate via shared variables. New local variables $\underline{z}$ are introduced with **let** and **choose**. For **let** the initial values of the variables are $\underline{e}$. The **choose**-statement is a very general nondeterministic construct (taken from ASMs [8]), which introduces infinite nondeterminism. It chooses some arbitrary initial values that satisfy φ and executes α_1. If there is no possible choice of values, e.g., if φ is false, then α_2 is executed. The **choose** construct can be used to define standard nondeterminism as

$$\alpha_1 \textbf{ or } \alpha_2 \equiv \textbf{choose } b \textbf{ with } \mathit{true} \textbf{ in } \{\textbf{if}^* \ b \textbf{ then } \alpha_1 \textbf{ else } \alpha_2\} \textbf{ ifnone skip} \ .$$

The compound operator $\alpha_1; \alpha_2$ denotes the sequential composition of the two subprograms. The star and interleaving operators are also overloaded for programs, and procedure calls $\mathit{proc}(\underline{e}; \underline{x})$ are also lifted to programs. Furthermore, any formula φ may be used as a program. (This is exploited for example in Section 5 to abstract a program by a RG formula.)

The semantics of programs is well-defined without any restrictions on the expressions and formulas used in programs. For the calculus described in the next section, however, we use *regular programs*, where all formulas φ and expressions e used are state expressions, and the last clause of the grammar that allows an arbitrary formula as a program is disallowed. Regular programs have a nonempty set of runs from any initial state, while general programs could be equivalent to false.

Unlike the assignment in conventional programming languages, assignments in standard ITL typically have no frame assumption, i.e., during an assignment, other variables may change arbitrarily. Instead, we use an explicit vector of distinct, flexible variables $\underline{x}$ as a local frame assumption for a program. We denote with $[\alpha]_{\underline{x}}$ the (shallow) embedding of the program α with frame assumption $\underline{x}$ into temporal logic: $[\alpha]_{\underline{x}}$ is not a new type of formula[2] but an abbreviation for a formula of the logic. An assignment $[\underline{z} := \underline{e}]_{\underline{x}}$ executes atomically in one unblocked step and leaves all variables unchanged that do not occur on the left hand side $\underline{z}$ but are in $\underline{x}$ (written $\underline{x} \setminus \underline{z}$).

$$[\underline{z} := \underline{e}]_{\underline{x}} \ \equiv \ \underline{z'} = \underline{e} \wedge \textbf{step} \wedge \neg \textbf{ blocked} \wedge \underline{y} = \underline{y'} \text{ where } \underline{y} = \underline{x} \setminus \underline{z} \ . \tag{2}$$

[2] However, the implementation in KIV supports programs, as well as temporal operators like $\Box$ and **E**, as syntactic constructs and provides rules for them directly in the calculus. This ensures that proof obligations remain readable.

Frame assumptions propagate over compound, star, interleaving, procedure calls and general formulas φ, reducing the semantics of these program constructs to the semantics of the corresponding operators on the formula-level:

$$\begin{array}{rcl@{\qquad}rcl}
[\alpha_1;\alpha_2]_{\underline{x}} & \equiv & [\alpha_1]_{\underline{x}};[\alpha_2]_{\underline{x}} & [\alpha^*]_{\underline{x}} & \equiv & ([\alpha]_{\underline{x}})^* \\
[\alpha_1 \parallel \alpha_2]_{\underline{x}} & \equiv & [\alpha_1]_{\underline{x}} \parallel [\alpha_2]_{\underline{x}} & [\alpha_1 \parallel_{\mathrm{nf}} \alpha_2]_{\underline{x}} & \equiv & [\alpha_1]_{\underline{x}} \parallel_{\mathrm{nf}} [\alpha_2]_{\underline{x}} \\
[proc(\underline{e};\underline{z})]_{\underline{x}} & \equiv & proc(\underline{e};\underline{z}) \wedge \Box\ \underline{y} = \underline{y'}, & [\varphi]_{\underline{x}} & \equiv & \varphi\ .
\end{array}$$

$$\text{where } \underline{y} = \underline{x} \setminus \underline{z}$$

The interval-based semantics of the remaining constructs for sequential programs can now be defined as follows:

$$[\mathbf{skip}]_{\underline{x}} \equiv \mathbf{step} \wedge \neg\, \mathbf{blocked} \wedge \underline{x'} = \underline{x} \tag{3}$$

$$[\mathbf{if}^*\varphi\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{x}} \equiv \varphi \wedge [\alpha_1]_{\underline{x}} \vee \neg\, \varphi \wedge [\alpha_2]_{\underline{x}} \tag{4}$$

$$[\mathbf{if}\ \varphi\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{x}} \equiv [\mathbf{if}^*\varphi\ \mathbf{then}\ (\mathbf{skip};\alpha_1)\ \mathbf{else}\ (\mathbf{skip};\alpha_2)]_{\underline{x}} \tag{5}$$

$$[\mathbf{while}^*\ \varphi\ \mathbf{do}\ \alpha]_{\underline{x}} \equiv (\varphi \wedge [\alpha]_{\underline{x}})^*;(\neg\, \varphi \wedge \mathbf{last}) \tag{6}$$

$$[\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha]_{\underline{x}} \equiv [(\mathbf{while}^*\ \varphi\ \mathbf{do}\ (\mathbf{skip};\alpha));\mathbf{skip}]_{\underline{x}} \tag{7}$$

$$[\mathbf{await}\ \varphi]_{\underline{x}} \equiv (\neg\, \varphi \wedge \mathbf{blocked} \wedge \mathbf{step})^*;(\varphi \wedge \mathbf{last}) \tag{8}$$

$$[\mathbf{let}\ \underline{z} = \underline{e}\ \mathbf{in}\ \alpha]_{\underline{x}} \equiv \exists\, \underline{y}.\, \underline{y} = \underline{e} \wedge [\alpha^{\underline{y}}_{\underline{z}}]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y''} = \underline{y'} \tag{9}$$

$$[\mathbf{choose}\ \underline{z}\ \mathbf{with}\ \varphi\ \mathbf{in}\ \alpha\ \mathbf{ifnone}\ \alpha_2]_{\underline{x}} \equiv \tag{10}$$
$$(\exists\, \underline{y}.\, \varphi^{\underline{y}}_{\underline{z}} \wedge [\alpha^{\underline{y}}_{\underline{z}}]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y''} = \underline{y'}) \vee (\neg\, \exists\, \underline{z}.\, \varphi) \wedge [\alpha_2]_{\underline{x}}\ .$$

A **skip** leaves all variables in the frame assumption unchanged. As long as φ is not satisfied, **await** φ repeatedly executes **blocked** steps. This gives the environment a chance to unblock the program. The definition of **let** expands local variables $\underline{z}$ to an existential quantifier over new variables $\underline{y}$. These must be disjoint from the variables used in $\underline{e}$, $\underline{x}$ and α. The variables in α are renamed to these new variables, written $\alpha^{\underline{y}}_{\underline{z}}$. The $\Box$-formula indicates that the variables $\underline{y}$ are local, i.e., they are not modified by environment steps. The semantics of **choose** is defined similarly.

2.6 Specifications

Basic specifications *SP = (SIG,X,U,Ax,Decl)* consist of a signature, static and flexible variables X and U, a set of axioms and a set of procedure declarations. The semantics of a specification is loose, i.e., it consists of all algebras that satisfy the axioms ($\mathcal{A} \models \varphi$ for all axioms $\varphi \in Ax$). Specifications may specify the semantics $proc^{\mathcal{A}}$ of some procedures *proc* by supplying procedure declarations as detailed below. The semantics of all other procedures is unspecified. Most specifications define abstract data types such as natural numbers, arrays or lists. In this case, the axioms are state formulas. In general, however, specifications may also contain temporal formulas as axioms, typically as properties (e.g., contracts) that restrict the semantics of any procedures without declarations.

Larger specifications are typically structured into a hierarchy of specifications using the usual algebraic operations (see, e.g., [46]) such as union or enrichment. Generic specifications that have sub-specifications as parameters are possible, an example being lists over an ordered parameter type of elements. Instantiation can be used to update this parameter specification using a signature morphism. In

the example, the elements could be instantiated with natural numbers, mapping the generic order predicate on elements to the usual order on natural numbers. Instantiation requires to prove the instantiated axioms of the parameter specification (here, the order axioms) as theorems over the actual specification. Instantiation is used in the context of temporal logic to generate proof obligations for procedures (this feature will be detailed in Section 5).

Procedure declarations are of the form $proc(\underline{x};\underline{y})\{\alpha\}$. They can be mutually recursive. The body α is subject to two restrictions that guarantee that the semantics is well defined. First, α may only assign to its parameters $\underline{x},\underline{y}$ and local variables introduced by **let** and **choose**. Second, α must be a *regular* program. Regular programs α are monotonic in their calls: for two procedures $proc_1$ and $proc_2$ with the same argument types if $proc_1^{\mathcal{A}} \subseteq proc_2^{\mathcal{A}}$, then $\{I : I \models \alpha\} \subseteq \{I : I \models \alpha'\}$, where α' is α with the calls to $proc_1$ replaced by calls to $proc_2$. Therefore, the semantics of recursive procedure declarations can be defined according to the standard Knaster-Tarski fix-point theorem. In particular, a declaration $proc(\underline{x};\underline{y}).\ \{\alpha\}$ yields the fix-point equation

$$proc^{\mathcal{A}} = \{(I(0)(\underline{x}), I(\underline{y})) : I \models [\textbf{let } \underline{z} = \underline{x} \textbf{ in } \alpha^{\underline{z}}_{\underline{x}}]_{\underline{y}}\}\ ,$$

which itself implies the *unfolding axiom*

$$proc(\underline{e};\underline{y}) \leftrightarrow \exists\ \underline{z}.\ \underline{z} = \underline{e} \wedge [\alpha^{\underline{z}}_{\underline{x}}]_{\underline{y},\underline{z}} \wedge \Box\ \underline{z'} = \underline{z''}\ . \tag{11}$$

by directly expanding the **let**. New local variables $\underline{z}$, which cannot be changed by the environment, are used for the value parameters $\underline{x}$ with initial values $\underline{e}$. Changes to the reference parameters are globally visible. Monotonicity and the unfolding axiom have been formally verified over the semantic embedding of RGITL into HOL [28].

3 Symbolic Execution

The basic deduction principle underlying the calculus is *symbolic execution*. We first motivate the principle and give an informal description of a symbolic execution step. Technically speaking, executing a step is done in two phases. The first phase computes "step form", which facilitates stepping forwards to the next state of an interval in the second phase by using suitable substitutions. The two phases are described in Subsections 3.2 and 3.3. Finally, Subsection 3.4 illustrates symbolic execution with a small example.

3.1 Motivation

The principle of symbolic execution is well-known for sequential programs. It replaces Hoare's assignment rule with the following rule on the right (a version of this rule is already in [16]).

$$\frac{}{\{\text{post}^t_x\}\ \ x := t\ \{\text{post}\}}$$

Hoare's simple assignment rule

$$\frac{\{\text{pre}^{x_0}_x \wedge x = t^{x_0}_x\}\ \ \alpha\ \ \{\text{post}\}}{\{\text{pre}\}\ \ x := t; \alpha\ \ \{\text{post}\}}$$

Rule for symbolic execution

The rule is given for a nonempty α. The premise uses a new variable x_0 to store the value of x before the assignment. For an empty α, the premise is just the implication $\mathrm{pre}_x^{x_0} \wedge x = t_x^{x_0} \rightarrow \mathrm{post}$. Although the rule looks more complicated, it is actually easier to use than Hoare's assignment rule. It allows one to step *forwards* through a program, computing postconditions from preconditions, while Hoare's rule computes backwards. We find reasoning forwards more natural since it corresponds to stepping through a program with a debugger. Instead of using the concrete values of a debugger, symbolic execution uses assertions describing a set of states. When assertions are used that describe a single initial state, symbolic execution directly mimics the behavior of a program interpreter.

We adapt this principle to temporal logic and reasoning about interleaved programs. For formal deduction, the sequent calculus is used, which writes assertions (goals) as *sequents* $\Gamma \vdash \Delta$, where the *antecedent* Γ and the *succedent* Δ are two (possibly empty) lists of formulas. A sequent $\Gamma \vdash \Delta$ is viewed as a convenient notation for the formula $(\bigwedge \Gamma) \rightarrow (\bigvee \Delta)$. It is therefore valid if the universal closure of this formula is valid. An empty antecedent corresponds to formula true and an empty succedent to false respectively. Rules in sequent calculus have the form

$$\frac{p_1 \ldots p_n}{c}$$

where sequent c is the conclusion, and sequents $p_1 \ldots p_n$ are the premises.

A rule with no premises is an axiom. A rule is sound if valid premises above the line imply a valid conclusion. Rules are applied bottom-up, reducing goals to simpler goals. Most used rules are *invertible*, i.e., a valid conclusion also implies valid premises.

A typical goal in temporal logic for a program α has the form

$$c \quad \equiv \quad \mathrm{pre}, \alpha, E \vdash \varphi \,, \tag{12}$$

where pre is a precondition (a state formula) describing the initial state, and E is an *environment assumption* about the steps of the environment. The *trivial environment assumption* requires that the environment never changes the variables $\underline{x}$ of the program: $\Box\, \underline{x'} = \underline{x''}$. Finally, formula φ is the property of the program to be proved.

A rough idea how to transfer symbolic execution to temporal logic is that computing the strongest postcondition of the first step of a program reduces goal c to a premise p

$$p \quad \equiv \quad \mathrm{pre}_1, \alpha_1, E_1 \vdash \varphi_1 \tag{13}$$

where the following hold:

- α_1 is α with its first step removed.
- pre_1 is an assertion about the state after the first step, the new precondition of α_1. If the first step of α is an assignment $x := t$, and if the environment assumption is the trivial one, then this is just the formula $\mathrm{pre}_x^{x_0} \wedge x = t_x^{x_0}$ used above. For a nontrivial environment assumption, this formula must be weakened after the first environment step. One more new variable x_1 is required to store the state after the assignment but before the environment step.
- E_1 is the assumption that follows from E for the rest of the program (often identical to E).

– φ_1 is the property that must be proved for the remaining program α_1.

Since the program has been reduced by one step, the intervals I that form possible runs of α must also be reduced to $I_{[1..]}$ by removing the leading step to get possible runs of α_1. Therefore, the soundness of the rule roughly requires proving $I \models c$ iff $I_{[1..]} \models p$.

This idea not only works for programs but also for all *regular formulas*. These contain only regular programs, and they use procedure calls as well as the chop and star operator just in such programs. The other temporal operators are used at the formula level only. Equations and lambda-expressions do not use temporal operators at all, and applications $e(\underline{e}')$ may use temporal operators only when e is a standard boolean operator.

In general, symbolic execution gives a main subgoal of the form (13) but will also result in side goals. If the proof obligation is, for example, a safety formula $\varphi = \Box\, p(x)$, then φ_1 will be $\Box\, p(x)$ too, and a side condition is necessary that requires to prove $p(x)$ for the current state. Proving a liveness condition $\Diamond\, p(x)$ requires to prove either $p(x)$ now or (13) with $\varphi_1 = \Diamond\, \varphi$.

To generate the correct premises, symbolic execution works in two phases, as defined below. The first phase splits formulas into *step-formulas* $\tau(\underline{x}, \underline{x}', \underline{x}'')$ and *next-formulas* of the form $\circ\, \varphi$ with arbitrary φ. The second phase removes the first step of the interval by replacing flexible variables $\underline{x}, \underline{x}', \underline{x}''$ with $\underline{x_0}, \underline{x_1}, \underline{x}$ in step-formulas, using new *static* variables $\underline{x}_0$ and $\underline{x}_1$, and by removing the leading next operator from next-formulas.

3.2 Phase 1 of a Symbolic Execution Step: Computing Step Form

The first phase of symbolic execution is based on the following definition:

Definition 2 (Step-formulas, Next-formulas and Step Form) A *step-formula* τ is a formula in which temporal logic operators are applied only to static subformulas. A *next-formula* is a formula of the form $\circ\, \varphi$. A formula χ is in *step form* if it is a predicate logic combination of step-formulas and next-formulas:

$$\chi ::= \quad \tau \quad | \quad \circ\, \varphi \quad | \quad \neg\, \chi \quad | \quad \chi_1 \wedge \chi_2 \quad | \quad \forall\, \underline{v}.\, \chi\ .$$

Semantically, next-formulas are independent of the first step (i.e., they depend on $I_{[1..]}$ only), while step-formulas satisfy

Proposition 1 *The validity of step-formulas τ depends on the first step of an interval I only:*[3]

$$I \models \tau \quad \textit{iff} \quad I_{[0..1]} \models \tau\ .$$

Phase 1 of symbolic execution transforms formulas to step form. It relies on the following theorem:

Theorem 2 (Transformation of Formulas into Step Form) *Any regular formula without procedure calls can be transformed into an equivalent formula in step form.*

[3] For an empty interval, we define $I_{[0..1]} := I$.

The proof of Theorem 2 is given in Appendix A. It relies on being able to transform programs to a certain normal form that is stronger than step form. The theorem leaves out procedure calls since not all procedures must have a declaration. Given a declaration, the unfolding axiom (11) is applied and the body of the procedure (which is a regular program) is transformed to normal form. When a procedure is specified with a property (e.g., if a contract is given), then the procedure call must be abstracted to this property first. Given that the property is a regular formula, then symbolic execution becomes possible with the abstraction.

The theorem above requires regular formulas. We emphasize that it is possible to transform a larger class of formulas to step form, and we occasionally use this fact. As we explain in Section 5, the proof of rule (19) abstracts a procedure call with a rely-guarantee property. The resulting non-regular program still can be symbolically executed by executing the (regular) RG formula instead of the original call.

The main problematic formulas that the calculus currently cannot handle are universal quantifiers over flexible variables within programs. An example is $(\forall\ x.\ \varphi);\ \psi$ since the universal quantifier over a flexible variable cannot be moved to the top: Even if $x \notin \mathrm{free}(\psi)$, the formula is *not* equivalent to $\forall\ x.\ (\varphi;\ \psi)$ (note that the equivalence is valid for a static variable as well as for existential quantification). Therefore, the crucial limitation of regular programs is that they only use *existential* quantifiers over flexible variables when being expanded to formulas (cf. **let** and **choose**).

3.3 Phase 2 of a Symbolic Execution Step: Applying Substitutions

For formulas χ in step form, two substitutions $\mathcal{N}(\chi)$ and $\mathcal{L}(\chi)$ are defined. Function $\mathcal{N}$ transforms relevant information about the first step (system and environment transition) of an interval to information about static variables. This allows one to consider the interval without the first step. For an empty interval, function $\mathcal{L}$ replaces all flexible variables with static variables. Applying these two substitutions constitutes phase 2 of a symbolic execution step.

More specifically, $\mathcal{N}(\chi)$ substitutes all free flexible variables $\underline{x}, \underline{x'}, \underline{x''}$ in step-formulas with $\underline{x}_0, \underline{x}_1, \underline{x}$ using fresh *static* variables $\underline{x}_0$ and $\underline{x}_1$ for the old values of $\underline{x}$ and $\underline{x'}$ respectively. It also replaces **blocked** with a fresh boolean variable bv, **last** with false and removes the leading next operator from next-formulas *without* substituting the body. The difficult case of applying the substitution on step-formulas is a quantifier over a flexible variable: $\mathcal{N}(\forall\ y.\ \varphi)$ is defined as $\forall\ y_0, y_1, y.\ \mathcal{N}(\varphi)$, where the $\mathcal{N}$ on the right-hand side replaces free variables and y, y', y'' in step-formulas with y_0, y_1, y, again using static variables y_0, y_1.

The second substitution $\mathcal{L}(\chi)$ is for the special case of empty intervals: It replaces all next-formulas with false, **last** with true, **blocked** with false and all flexible variables $\underline{x}$ (unprimed, primed and double primed) in step-formulas with fresh static variables $\underline{x}_0$. For quantifiers, $\mathcal{L}(\forall\ y.\ \varphi) \equiv \forall\ y_0.\ \mathcal{L}(\varphi)$, where the $\mathcal{L}$ on the right-hand side additionally replaces y, y' and y'' with the new static variable y_0. Both substitutions have no effect on static formulas within step-formulas.

Theorem 3 (Correctness of Substitutions $\mathcal{N}$ and $\mathcal{L}$) *For any formula χ in step form with* $\text{free}(\chi) \cap X = \underline{x}$*, fresh static variables $\underline{x}_0$,$\underline{x}_1$ and every interval I:*

$$\begin{aligned} &\textit{if } \#I \neq 0 \textit{ then:} && I \models \chi \;\; \textit{iff } I_{[1..]}[\underline{x}_0, bv, \underline{x}_1 \leftarrow I(0)(\underline{x}), I(0)_b, I(0)'(\underline{x})] \models \mathcal{N}(\chi) \\ &\textit{if } \#I = 0 \textit{ then:} && I \models \chi \;\; \textit{iff } I[\underline{x}_0 \leftarrow I(0)(\underline{x})] \models \mathcal{L}(\chi)\ . \end{aligned}$$

Proof This is by induction over the complexity of formulas in step form. The proof resembles standard proofs for the substitution lemma of predicate logic. □

The mechanized proof of Theorem 3 can be found online as part of the embedding of RGITL in HOL [28].

Theorem 4 (Step Rule) *If the formulas in Γ and Δ can be transformed to formulas Γ_1 and Δ_1 in step form, then the following rule is sound and invertible:*

$$\frac{\mathcal{N}(\Gamma_1) \vdash \mathcal{N}(\Delta_1) \qquad \mathcal{L}(\Gamma_1) \vdash \mathcal{L}(\Delta_1)}{\Gamma \vdash \Delta}\ \textit{step} \tag{14}$$

Proof Directly by applying Theorem 3 since the sequents $\Gamma \vdash \Delta$ and $\Gamma_1 \vdash \Delta_1$ are equivalent by Theorem 2. □

3.4 Example

To demonstrate how symbolic execution works in practice, we now consider the concrete goal

$$N = 1, \Box\ N'' \leq N', [\{\textbf{let } M = N + 1 \textbf{ in } N := M\}; N := 4]_N \vdash \Box\ N' \geq N\ ,$$

which states that starting with $N = 1$, both steps of the program will never decrease N if the environment never increases it. Recall the convention that in concrete formulas, uppercase letters denote flexible variables and lowercase letters static variables. The first phase of executing a step will transform formulas into step form. Since the top-level program is a sequential composition, the **let** must be transformed first. Applying the definition of **let** (9) and assignment (2), and unwinding the resulting $\Box$-formula (cf. Appendix A) gives

$$\exists\ M.\ M = N+1 \wedge N' = M \wedge \neg\ \textbf{blocked} \wedge \circ\textbf{last} \wedge M' = M \wedge M'' = M' \wedge \bullet(\Box\ M'' = M')\ ,$$

where the $\circ$ and the $\bullet$ formula can be contracted to $\circ\ (\textbf{last} \wedge \Box\ M'' = M')$. The step form shows that the program does not terminate immediately and has only one possible unblocked step. Substituting the result into the sequent, applying *chp-stp* (cf. Table 1 in Appendix A) and computing step form for the other formulas gives

$$\begin{aligned} &N = 1, N'' \leq N' \wedge (\textbf{last} \vee \circ\ \Box\ N'' \leq N'), \\ &\exists\ M. \quad M = N + 1 \wedge N' = M \wedge \neg\ \textbf{blocked} \wedge M' = M \wedge M'' = M' \\ &\qquad \wedge \circ\ (\textbf{last} \wedge \Box\ M'' = M'); [N := 4]_N \\ &\vdash N' \geq N \wedge (\textbf{last} \vee \circ\ \Box\ N' \geq N)\ . \end{aligned}$$

The sequent is now in step form[4]. Phase two of the symbolic execution step creates two goals. The goal where substitution $\mathcal{L}$ is used is trivial since the substitution gives false for $\circ(\mathbf{last} \wedge \Box\ M'' = M'); [N := 4]_N$. The other goal where $\mathcal{N}$ is applied replaces N, N', N'' with n_0, n_1 and N. Similarly, M, M', M'' become m_0, m_1 and M within the quantification. Formula **last** in the succedent is replaced with false, **blocked** with a boolean variable bv and the next operators are dropped. Therefore, the result of the symbolic execution step is

$$\begin{aligned} & n_0 = 1, N \leq n_1 \wedge (\text{false} \vee \Box\ N'' \leq N'), \\ & \exists\ m_0, m_1, M. \quad m_0 = n_0 + 1 \wedge n_1 = m_0 \wedge \neg\ bv \wedge m_1 = m_0 \wedge M = m_1 \\ & \qquad\qquad \wedge (\mathbf{last} \wedge \Box\ M'' = M'); [N := 4]_N \\ \vdash\ & n_1 \geq n_0 \wedge (\text{false} \vee \Box\ N' \geq N)\ . \end{aligned}$$

Predicate logic simplification now drops the existential quantifier and proves the first conjunct of the succedent $n_1 \geq n_0$. This shows that N was indeed not decreased in the first system step of the run. Simplifying the rest of the sequent and also dropping equations for static variables that are no longer needed ($u = t \vdash \varphi$ is equivalent to $\vdash \varphi$, when $u \notin \text{free}(\varphi)$) gives

$$N \leq 2, M = 2, \Box\ N'' \leq N', [N := 4]_N, \vdash \Box\ N' \geq N\ .$$

With another symbolic execution step one gets

$$n_0 \leq 2, m_0 = 2, n_1 = 4 \wedge \neg\ bv \wedge \mathbf{last} \vdash \Box\ N' \geq N\ .$$

For this goal, a last symbolic execution gives a trivial goal for $\mathcal{N}$ (since **last** is true now), and the final $\mathcal{L}$ goal requires proving $n_0 \geq n_0$.

4 Induction

Proofs of temporal formulas often rely on well-founded induction. This section first gives an induction rule for temporal logic (Section 4.1) and then shows how induction over safety formulas can be reduced to induction over counter variables, similar to N above (Section 4.2). In an interleaved context different execution paths often yield the same goal. We show how duplicate proofs can be avoided using proof lemmas (Section 4.3). Often it is necessary to use a very general induction hypothesis such that a goal for any repeated state can be easily closed. This is achieved by the lazy induction scheme from Section 4.4 that spares the user defining several induction hypothesis during a proof.

4.1 Well-founded Induction

In higher-order logic, proving a formula $\varphi(n)$ by induction over a well-founded order $\prec$ gives an induction hypothesis $\forall\ \underline{v}, m.\ m \prec n \rightarrow \varphi(m)$ with $\underline{v} = \text{free}(\varphi) \backslash \{n\}$. For temporal reasoning, adding this formula to the preconditions of a goal is

[4] The implementation in KIV simplifies $(\mathbf{last} \wedge \ldots); N := 4$ to $N := 4$ and also eliminates the existential quantifier. To explain the basic symbolic execution step we avoid such simplifications.

not sufficient as an induction hypothesis. The formula would hold for the *current* interval only, while de facto we need it for *all* intervals. Therefore, we use the following stronger rule for induction over a term e that quantifies over all possible intervals as well:

$$\frac{e = N, \mathbf{Ind\text{-}Hyp}(N) \vdash \varphi}{\vdash \varphi} \quad . \tag{15}$$

The rule uses a new flexible variable N that "starts" with the current value of e, the induction hypothesis is $\mathbf{Ind\text{-}Hyp}(N) \equiv \mathbf{A}\ \forall\ \underline{v}.\ (e \prec N \rightarrow \varphi)$, and $\underline{v} = \text{free}(\varphi) \cup \text{free}(e)$.

The validity of the induction hypothesis depends on the *initial* value of N only. If during execution the current value of N becomes less than its initial value, then the induction hypothesis becomes applicable by instantiating variables $\underline{v}$ with a substitution θ. Note that we make use of the "for all paths" operator $\mathbf{A}$ here. It allows using the induction hypothesis with any interval, where N is smaller than in the current interval. This is particularly relevant when the induction hypothesis is applied with $spawn_{n-1}$ in Figure 5 – its interval in the interleaved context typically consists of fragments of the original interval.

4.2 Counters and Prefix Induction

To reason about temporal formulas, most calculi use specialized induction rules in addition to well-founded induction. RGITL reduces such principles to standard well-founded induction whenever possible. In particular, the following equivalence can be used if a liveness property $\Diamond\ \varphi$ is known in order to introduce a term N for well-founded induction:

$$\Diamond\ \varphi \leftrightarrow \exists\ N.\ N = N'' + 1\ \mathbf{until}\ \varphi\ . \tag{16}$$

Counter N is a new flexible variable that is decremented in each step until a state is reached where φ holds. Note that $N = N''+1$ is equivalent to $N > 0 \wedge N'' = N–1$.

When *proving* a property $\Box\ \varphi$, equivalence (16) is used to get a proof by contradiction by *assuming* that there is a number of steps N after which φ is false. The proof is then by induction over the initial value of N. Formally, the derived rule is

$$\frac{N = N'' + 1\ \mathbf{until}\ \neg\ \varphi, \mathbf{Ind\text{-}Hyp}(N), \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \Box\ \varphi} \quad , \tag{17}$$

where

$$\mathbf{Ind\text{-}Hyp}(N)\ \equiv\ \mathbf{A}\ \forall\ \underline{v}, N_0.\ \left(N_0 < N \wedge (N_0 = N_0'' + 1\ \mathbf{until}\ \neg\ \varphi) \wedge \bigwedge \Gamma\right) \rightarrow \bigvee \Delta\ .$$

Variables $\underline{v}$ are the free variables of the rule's conclusion, and N_0 is a fresh counter variable.

Symbolically executing k steps of the program transforms $\mathbf{Ind\text{-}Hyp}(N)$ to $\mathbf{Ind\text{-}Hyp}(N+k)$, where $N+k$ is the *initial* value of N when rule (17) was applied. Hence the decreasing of N (cf. the $N_0 < N$ conjunct in $\mathbf{Ind\text{-}Hyp}(N)$) can be easily established by instantiating N_0 with the current value of N after at least one symbolic execution step.

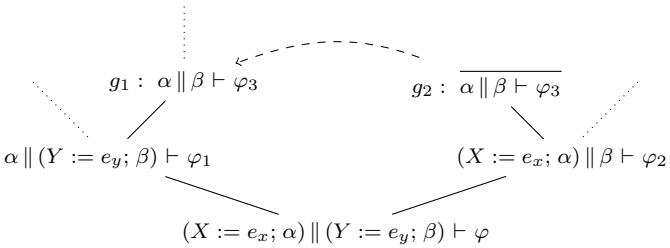

Fig. 2 Symmetric executions of interleaving and using proof lemmas

4.3 Proof Lemmas

The execution of interleaved parallel processes is nondeterministic. As a consequence, a large number of different paths of execution have to be considered. Very often, however, the order of execution has no effect on the computation; following different paths leads to the same state. A simple example is the program (with arbitrary programs α, β after the assignments)

$$(X := e_x; \alpha) \parallel (Y := e_y; \beta) \ .$$

Symbolic execution leads to the proof tree shown in Figure 2. The branches to the left/right denote steps of the left/right program, respectively. The same proof obligation $\alpha \parallel \beta \vdash \varphi_3$ arises both in goal g_1 and g_2 from two different symmetric schedules when the assignments commute. Naturally one wishes to prove it only once.

The calculus permits to introduce one goal of the proof as an ad-hoc lemma in another branch, as indicated by the dashed arrow: g_2 is closed by g_1. This is sound because the proof of g_1 can be copied for g_2. Alternatively, an explicit lemma can be introduced.

In general, when a goal $\Gamma \vdash \Delta$ is used as a lemma, $\theta(\bigwedge \Gamma \rightarrow \bigvee \Delta)$ is added to the assumptions of the current goal. A substitution θ for the free variables can be chosen at will and reflects that sequents are implicitly universally quantified. Proof lemmas can not be taken from proof goals in the same branch (below the current goal) as this would result in a cyclic argument.

4.4 Lazy Induction

Inductive proofs for loops and recursive procedures rely on an induction hypothesis. As described, induction rules add an induction hypothesis to the current goal. However, in the presence of interleaving, this is often not sufficient as the following example shows. Suppose that the initial goal g is

$$g : \ \ X := e_x; (\underbrace{\textbf{while}^* \text{ true } \textbf{do } \{Y := e_y; \alpha\}}_{w_1} \parallel \underbrace{\textbf{while}^* \text{ true } \textbf{do } \{Z := e_z; \beta\}}_{w_2}) \vdash \Box \ \varphi \ ,$$

where two while loops w_1 and w_2 are interleaved, prefixed with an initialization $X := e_x$, and $\Box \ \varphi$ should be proved.

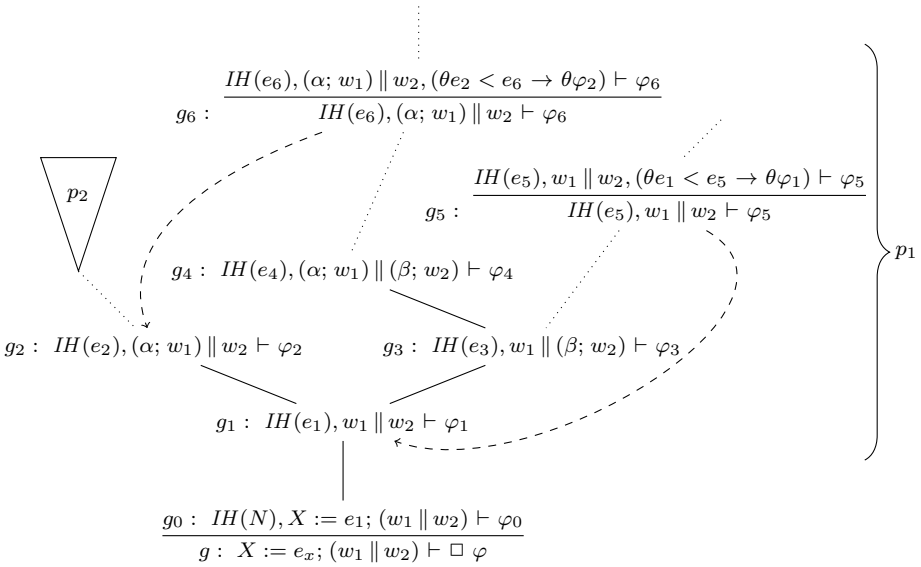

Fig. 3 Lazy induction example

The proof is by induction on $\Box\ \varphi$ using rule (17) and symbolic execution. The proof structure is shown in Figure 3. The solid lines represent symbolic execution steps, the dotted lines represent sequences of such steps, proof tree p_1 designates the whole part of the tree above g_1, and proof tree p_2 designates the sub-proof of g_2. Branches to the left/right denote steps where the left/right program is scheduled (some are omitted). The reader can disregard *IH* and g_0 for a moment.

First, the initial assignment is executed, leading to g_1 where the previous value of N is given by expression e_1 (which corresponds to $N + 1$ here, as explained before) and so on. Next, the different possible interleavings are considered.

The rightmost branch leading to goal g_5 executes w_2 once without scheduling the left program. At this point, induction needs to be applied. The matching goal (same program) can be found in g_1 (designated by the dashed arrow from g_5 to g_1). This suggests that the induction rule (17) should be applied in goal g_1 after executing the leading assignment. The hypothesis then yields $\theta\varphi_1$ for a chosen substitution θ, provided the counter has decreased $\theta e_1 < e_5$.

However, if the left program is scheduled in between, goal g_6 is eventually reached – but g_1 does not provide a helpful hypothesis (the programs do not match). In fact, the right hypothesis can be found in a different branch, namely at g_2 where only the first assignment of w_1 has been executed. This suggests that the induction hypothesis must be generalized to contain (at least) both g_2 and g_1.

In general, many such cases occur in proofs with interleaved programs. Collecting the different induction hypotheses lazily frees the programmer from anticipating them in the beginning. To do this, the lazy induction rule introduces a higher order variable *IH* as a placeholder for the generalized hypothesis instead of the concrete **Ind-Hyp** in rule (15).

The formula variable *IH* tracks the initial counter value and enables lazy induction application. As seen in the previous section, the argument of *IH increases*

with respect to counter N with every step. Intuitively, variable $IH(N+k)$ provides a hypothesis for *all* goals that evolve from executing more than k steps since applying lazy induction on N. (Induction can be even applied across different proof branches.) The benefit of this scheme is that IH never needs to be instantiated explicitly. In general, the induction application adds an instance of the hypothesis (the goal to the right) to the antecedent of the current goal (left) as follows:

$$\frac{IH(e_1), \theta e_2 \prec e_1 \rightarrow \theta\varphi_2 \vdash \varphi_1}{IH(e_1) \vdash \varphi_1} \qquad IH(e_2) \vdash \varphi_2$$

The lazy induction technique is sound since a traditional induction proof that uses only well-founded induction can always be constructed from one that uses lazy induction. (Of course this construction is not performed in practice.) The construction of the traditional proof consists of two parts that are illustrated for the example from Figure 3. First, the original proof is pruned at the lowest goals that were used as an induction hypothesis (here, at g_1). In this lower half of the proof, predicate variable IH is replaced with ($\lambda\ n.$ true) in all goals (so $IH(t)$ is equivalent to true). Since IH is implicitly universally quantified, the proof holds for all instances of IH, so the proof remains a valid proof for this instance. Then a lemma g_c is defined as the conjunction of all goals that were used as induction hypotheses to close the remaining premises (here, lemma g_c corresponds to $g_1 \wedge g_2$). The main proof of the example is then simply

$$\frac{\dfrac{}{g_1 :\ \text{true}, w_1 \parallel w_2 \vdash \varphi_1}\ \text{closed with lemma } g_c}{\vdots}$$
$$\frac{g_0 :\ \text{true}, X := e_1; (w_1 \parallel w_2) \vdash \varphi_0}{g :\ X := e_x; (w_1 \parallel w_2) \vdash \Box\, \varphi}$$

It remains to prove lemma g_c using well-founded induction only. This proof is illustrated in Figure 4. First, it adds a new induction variable N to g_c. (Technically, adding N is possible since $g_i \leftrightarrow \forall\ N.\ N = e_i \rightarrow g_i$.) Then it applies the rule for well-founded induction (15) on N. This gives an accumulated induction hypothesis **Ind-Hyp**(N) that implies all applications of induction in the lazy proof. Finally, the existing lazy proof is reused for each conjunct, instantiating IH with **Ind-Hyp**. Note that this may duplicate proofs: here p_2, which is a subtree of p_1, occurs twice.

KIV implements heuristics for both lazy induction and proof lemma application (based on identity of the program and the main temporal formula in the two goals). These can automatically verify simple mutual exclusion algorithms, e.g., mutual exclusion and starvation-freedom of Peterson's mutex algorithm [44] for two processes. However, such direct proofs for interleaved programs are typically rather large and fail for more complex examples. The next section describes a well-known compositional specification and verification technique, namely rely-guarantee reasoning, which gives compositional proofs and helps to reduce the proof effort.

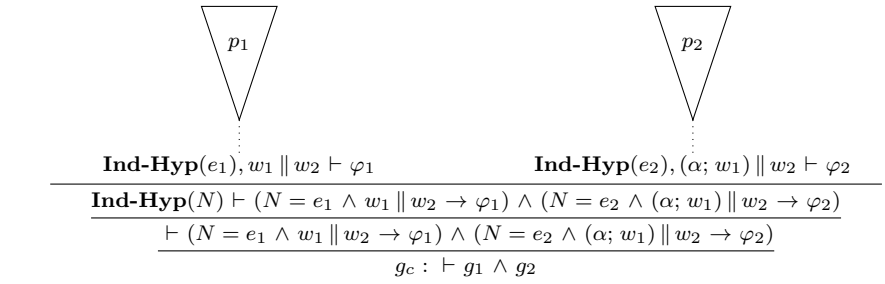

Fig. 4 Reuse of the proof with lazy induction

5 Rely-Guarantee Reasoning

This section describes the embedding of rely-guarantee (RG) reasoning in RGITL. It follows Jones' original approach [25] using binary predicates to specify rely and guarantee conditions (we currently do not use more general formulas as suggested by some other approaches, e.g., [40,17]). Parallel RG rules decompose global safety properties of a concurrent system to local proof obligations for the system's components. We show how the symbolic execution of the interleaving operator, and the compositionality properties of the chop, star and interleaving operators can be used to derive such rules.

Rely-guarantee abstractions are typically of the form

$$[proc(S)]_S \vdash R(S', S'') \xrightarrow{+} G(S, S') \; . \tag{18}$$

The formula uses a procedure *proc* with one reference parameter variable S that represents the *state* of some concurrent system (note that the abstract sort *state* is not related to semantic state functions of an interval), and two binary predicates of types $G, R : state \times state \rightarrow bool$.

This formula serves two purposes. On the one hand, it is an axiom for an abstract procedure *proc* that has no implementation in a (parameter) specification that has a signature with sort *state*, variable S and predicates R, G. In an enrichment of this specification, the conclusion of an RG rule for interleaving is proved as a theorem. The proof (see Appendix B) makes critical use of the compositionality of interleaving in Theorem (1).

On the other hand, formula (18) above becomes a proof obligation when the rely-guarantee rule is applied to verify a concrete algorithm. This is done by instantiating the parameter specification using a morphism that maps sort *state* and variable S to a vector of sorts $\underline{t}$ and variables $\underline{X} : \underline{t}$ from an instance specification. That specification contains a procedure declaration $proc(\underline{X}).\ \{\alpha\}$ that provides an algorithm to be verified. Predicates R and G are also mapped by the morphism to concrete predicates (of type $\underline{t} \times \underline{t} \rightarrow bool$). This gives an instance of the formula (18) above that (like all parameter axioms of an instantiation, see Section 2.6) has to be shown. When the proof is successful, the instance of the conclusion of the rely-guarantee rule becomes a valid theorem for the interleaved algorithm.

In the following we typically leave parameter S and the frame assumption implicit for brevity. The sustains operator $\xrightarrow{+}$ from (18) ensures that the guarantee

condition G is maintained by *proc*'s program transitions as long as *previous* environment transitions have preserved its rely condition R. In particular, the first program step must satisfy G. In the example interval below, property $R \xrightarrow{+} G$ enforces that the program step from $I(k)$ to $I'(k)$ must satisfy G (denoted as $\Rightarrow \in G$) because R holds previously.

$$\begin{array}{ccccccccccccc} I(0) & \text{——} & I'(0) & \cdots\cdots & I(1) & \text{——} & \cdots & \text{——} & I'(k-1) & \cdots\cdots & I(k) & \text{——} & I'(k) \quad \cdots \\ & \in G & & \in R & & \in G & \cdots & \in G & & & \in R & \Rightarrow \in G & \end{array}$$

Formally, the sustains operator is defined as[5]

$$R \xrightarrow{+} G \;\equiv \neg\, (R \,\mathbf{until} \neg\, G)\,.$$

The property "if all environment steps satisfy R, then all program steps satisfy G" is in general too weak to avoid circular dependencies between system components. It is easy to see that this property is strictly weaker than $\xrightarrow{+}$ (by induction over $\Box\, G$), i.e.,

$$(R \xrightarrow{+} G) \rightarrow (\Box\, R \rightarrow \Box\, G)\,,$$

but the reverse direction does not hold, as the following one-step interval shows. [6]

$$\begin{array}{ccccc} I(0) & \text{——} & I'(0) & \cdots\cdots & I(1) \\ & \notin G & & \notin R & \end{array}$$

The verification of a rely-guarantee property $R \xrightarrow{+} G$ of an interleaved system $\mathit{proc}_1 \parallel_{\mathrm{nf}} \mathit{proc}_2$ can be decomposed according to the following rely-guarantee rule. (For the weaker version, the rule would be wrong since it would require a cyclic argument.)

Theorem 5 (Rely-Guarantee Rule for Two Interleaved Operations) *The following rely-guarantee decomposition rule for two interleaved procedures is derivable in RGITL:*

$$\frac{\begin{array}{llll} \mathit{proc}_i \vdash R_i(S', S'') \xrightarrow{+} G_i(S, S') & & & G_i \vdash R_{3-i} \\ R \vdash R_i \qquad \vdash \mathit{trans}(R_i) & \vdash \mathit{refl}(G_i) & & G_i \vdash G \end{array}}{\mathit{proc}_1(S) \parallel_{nf} \mathit{proc}_2(S) \vdash R(S', S'') \xrightarrow{+} G(S, S')\,.} \tag{19}$$

The conclusion of the rule states that each transition of the interleaved system preserves a global guarantee G as long as the previous environment transitions satisfy R. (The weaker formula that $\Box\, G$ holds in a system environment $\Box\, R$ then simply follows, as explained above.) The main premise is the first. It decomposes the global property into two local rely-guarantee properties, one for each component (with premises for both $i = 1, 2$). All other premises are predicate logic side conditions that are derived from the definition of interleaving as we explain below.

Since a local environment transition of a component can consist of both transitions of the other component and transitions of the global environment, a component's rely R_i must be preserved by both the other component (premise 2) and

[5] Equivalent, but more complex definitions are also possible, e.g., using the weak until operator of TL (unless), as G **unless** $(G \wedge \neg\, R)$, or using chop and star, as $(G \wedge R \wedge \mathrm{step})^*$; $(G \wedge (\neg\, \mathrm{last} \rightarrow \neg\, R))$.

[6] The use of an operator similar to $\xrightarrow{+}$ can already be found in [36,25] and also in [2]. Note that a semantic definition as in [54] that essentially requires $(\Box\, R \rightarrow \Box\, G)$ to hold for *all prefixes* of the run is equivalent to $R \xrightarrow{+} G$.

each transitions of the global environment (premise 3). To group several subsequent transitions of other processes into one rely-transition, the rely condition must be transitive (premise 4). Premise 5 requires guarantee conditions to be reflexive, so that transitions which leave the state unchanged are admissible. To ensure that the global guarantee G is never violated, it must be implied by each local guarantee transition G_i (premise 6).

The proof of the theorem is given in Appendix B. It exploits two properties of a $R \xrightarrow{+} G$ formula. First, a $R \xrightarrow{+} G$ formula is a safety formula and therefore can be proved similarly to an always formula (cf. Section 4.2) by showing that there is *no* number N such that the rely holds for N transitions, and the guarantee is violated by the next transition. Formally,

$$(R \xrightarrow{+} G) \;\leftrightarrow\; \forall\, B.\, \Diamond\, B \rightarrow ((R \wedge \neg\, B) \xrightarrow{+} G) \tag{20}$$

holds for a boolean variable B, and equivalence (16) applied on $\Diamond\, B$ gives the required counter N for induction. Second, $R \xrightarrow{+} G$ can be symbolically executed using the following simple unwinding rule:

$$(R \xrightarrow{+} G) \;\leftrightarrow\; G \wedge (R \rightarrow \bullet\, (R \xrightarrow{+} G))\,. \tag{21}$$

Rely-guarantee rules for systems with an arbitrary finite number of interleaved components can also be derived. Figure 5 shows the specification of such an interleaved system. The concurrent system $spawn_n$[7] interleaves $n+1$ processes. Each spawned process $m \leq n$, executes a sequential procedure seq_m which infinitely often calls a generic procedure $proc_m$ that works on the overall system state S. Procedures $spawn_n$ and seq_m are implemented based on the parameter procedure $proc_m$ which is left unimplemented. The auxiliary function $Actf : \mathbb{N} \rightarrow bool$ is used to capture the termination of an individual $proc_m$. This function can be ignored for now; it will only become relevant in Section 7 for the specification and decomposition of the global progress property lock-freedom.

The call of seq_n in $spawn_n$ uses a reference parameter $Actf_n$ to indicate that seq_n modifies $Actf(n)$ but never modifies $Actf(m)$ for $m \neq n$. Procedure calls with reference parameters $f(\underline{e})$, where f is a function variable can be reduced to standard calls by the following rule:

$$[proc(\underline{e_0}; \underline{y}, f(\underline{e}))]_{\underline{y},f} \;\equiv\; \exists\, z.\, \Box\, (z = f(\underline{e}) \wedge f' = f(\underline{e}; z')) \wedge [proc(\underline{e_0}; \underline{y}, z)]_{\underline{y},z}\,. \tag{22}$$

A fresh flexible variable z records how $f(\underline{e})$ changes. The constraint $z = f(\underline{e})$ establishes that z equals $f(\underline{e})$ before each system step and after each environment step. Environment steps may modify f arbitrarily (no constraint given), while $f' = f(\underline{e}; z')$ ensures that system steps update f at $\underline{e}$ just as the procedure modifies z. Function modification $f(\underline{e}; z)$ is defined as

$$f(\underline{e}; z) \equiv \lambda\, \underline{u}.\ \text{if } \underline{u} = \underline{e} \text{ then } z \text{ else } f(\underline{u})\,,$$

with new variables $\underline{u}$. Expansion of the definition before using the unfolding axiom is unnecessary since one can directly substitute the formal parameter of the declaration with $f(\underline{e})$ in the body, using the convention that an assignment $f(\underline{e}) := t$ abbreviates $f := f(\underline{e}; t)$.

[7] For notational conciseness, the process number n is often written as an index rather than as an argument.

$spawn_n(S, Actf)$ {
if* $n = 0$ **then**
$seq_0(S, Actf_0)$
else
$seq_n(S, Actf_n)$
$\|_{\mathrm{nf}}$ $spawn_{n-1}(Actf, S)$ }

$seq_m(S, Act)$ {
while* true **do** {
Act := true;
$proc_m(S)$;
Act := false }}

Fig. 5 The concurrent system model

Similar to R in (19), the rely-guarantee rule for $spawn_n$ uses an overall system rely Rto_n which preserves each individual rely R_m of the spawned processes, whereas each system transition satisfies its local guarantee G_m that implies a global guarantee Gto_n, which corresponds to G in (19):

$$Rto_n(S', S'') \;\equiv\; \forall\, m \leq n.\; R_m(S', S'')$$
$$Gto_n(S, S') \;\equiv\; \exists\, m \leq n.\; G_m(S, S')\;.$$

Theorem 6 (Rely-Guarantee Reasoning for $spawn_n$)
If for all m, refl(G_m), trans(R_m) and $\forall\, m_0 \neq m.\; G_m \rightarrow R_{m_0}$ holds, and $proc_m(S) \vdash R_m(S', S'') \xrightarrow{+} G_m(S, S')$, then

$$spawn_n(Actf, S) \vdash Rto_n(S', S'') \xrightarrow{+} Gto_n(S, S')\;. \tag{23}$$

Proof First, we show that each seq_m sustains its guarantee G_m if its rely R_m holds in previous transitions:

$$seq_m \vdash R_m \xrightarrow{+} G_m\;. \tag{24}$$

This follows directly from $proc_m \vdash R_m \xrightarrow{+} G_m$, which is used as an abstraction for $proc_m$ in the procedure implementation of seq_m. Observe that we must also use the compositionality of the chop and star operators (for the loop in seq_m) here. Safety induction over the sustains formula using rules (20) and (16) reduces the verification of the infinite loop in seq_m to the verification of its body. Since G_m is reflexive, it holds trivially when the activity flag is set and thus each transition of the loop body preserves the RG property.

The main proof of (23) is by structural induction over n. It uses (24) and the induction hypothesis as abstractions for the interleaved components. The rest of the proof closely resembles the correctness proof of rule (19). Therefore, further details are omitted. □

For the practical verification of concrete algorithms, we have also derived parallel RG rules that include further predicates, e.g., for state-invariants or pre- and post-conditions. (These predicates are omitted here in order to better focus on the basic ideas.) Furthermore, for the frequent case of systems where components exhibit similar behavior, we have derived versions of rely-guarantee rules that permit specifications to consider a small number of representative components only, thus avoiding quantification over all local states, as in the original approach [25]. This reduction simplifies both specifications and proofs.

6 Fairness

Reasoning about an interleaved program $\alpha_1 \parallel \alpha_2$ by symbolic execution is indifferent to whether the interleaving is unfair or weak-fair. This is because symbolic execution only explores finite interval prefixes, while fairness is defined on infinite intervals only. Hence, symbolic execution alone is not sufficient to deal with (weak) fairness. We need another way to ensure that in a weak-fair interleaving each of the interleaved components *eventually* executes a step.

To formalize this "eventually" property, we define an extended interleaving operator $L_1\colon \alpha_1 \parallel L_2\colon \alpha_2$, where L_1 and L_2 are two formulas ("scheduling labels"), which enforce a step of α_1 and α_2, respectively. Informally, whenever label L_1 is true, the next step of the interleaving must be one of α_1. If this step is blocked, then a step of α_2 is executed as well. If α_1 is in its last state, then L_1 has no effect. Unlabeled interleaving $\alpha_1 \parallel \alpha_2$ is thus a special case of labeled interleaving where both labels are false. The corresponding definition from Section 2.4 is extended to

$$
\begin{aligned}
I \models L_1{:}\varphi \parallel L_2{:}\psi \quad & \text{iff there are } I_1, I_2, sc, l, r\colon \\
& I_1 \models \varphi \text{ and } I_2 \models \psi \text{ and } (I, sc, l, r) \in I_1 \oplus I_2 \text{ and } sc \text{ is fair} \\
& \text{and for all } n < \#sc\colon \qquad (l(n) = l(n+1) \rightarrow I_{[n..]} \not\models L_1) \\
& \qquad\qquad\qquad\qquad \text{and } (r(n) = r(n+1) \rightarrow I_{[n..]} \not\models L_2)\ .
\end{aligned}
$$

Similar scheduling labels are defined for $\parallel_{\mathrm{nf}}$. The rules for symbolic execution from Table 2 in Appendix A are adapted to propagate labels. The only other required adaption is the extra condition $\neg\ L_2$ in rule *ilvl-stp* in Table 2 from Appendix A. This ensures that when label L_2 is true, it is impossible that the next step is just a step of the left program.

The following fairness rule for α_1 (together with a symmetric one for α_2) introduces a scheduling label:

$$
L_1\colon \alpha_1 \parallel L_2\colon \alpha_2 \leftrightarrow \exists\ B.\ \Diamond\ B \wedge ((L_1 \vee B)\colon \alpha_1 \parallel L_2\colon \alpha_2)\ . \tag{25}
$$

Typically, labels L_1 and L_2 are both false, so the rule simplifies to

$$
\alpha_1 \parallel \alpha_2 \leftrightarrow \exists\ B.\ \Diamond\ B \wedge (B\colon \alpha_1 \parallel \alpha_2)\ . \tag{26}
$$

The formula asserts that there exists a number of steps after which the new boolean variable B becomes true, thus enforcing a step of α_1.

Example A simple example demonstrates the interplay between the given rules:

$$
\Box\ X' = X'',\ [X := 1 \parallel \mathbf{skip}^*]_X,\ X = 0 \vdash \Diamond\ X = 1
$$

can be derived by applying (26), then (16) on $\Diamond\ B$, which introduces the variable N. Induction (15) over N then yields

$$
\begin{aligned}
& N = N'' + 1\ \mathbf{until}\ B,\ \mathbf{Ind\text{-}Hyp}(N), \\
& \Box X' = X'',\ [B\colon X := 1 \parallel \mathbf{skip}^*]_X,\ X = 0 \vdash \Diamond\ X = 1\ .
\end{aligned}
$$

Executing a step now either makes B true, in which case the left process is executed and $X = 1$, or it lets B remain false. Then the resulting sequent is identical, except that N has decreased, so the induction hypothesis is applicable.

Interestingly, unfair interleaving satisfies almost the same rule. Either α_1 is executed after some steps, or the run consists of an infinite sequence of unblocked α_2 steps:

$$\begin{aligned} \alpha_1 \parallel_{\mathrm{nf}} \alpha_2 \quad \leftrightarrow \quad & (\exists\, B.\ \Diamond\, B \wedge (B\colon \alpha_1 \parallel_{\mathrm{nf}} \alpha_2)) \\ & \vee\ \alpha_2 \wedge \mathbf{inf} \wedge \Box\, \neg\, \mathbf{blocked} \wedge \mathbf{E}\, \exists\, \underline{x}.\ \alpha_1\ . \end{aligned} \tag{27}$$

Formula $\mathbf{E}\, \exists\, \underline{x}.\ \alpha_1$, where $\underline{\mathrm{x}} = \mathrm{free}(\alpha_1)$, ensures that α_1 is satisfied by at least one interval I_1 (i.e., is not equivalent to false) that can be used to derive $I_2 \in (I_1 \parallel_{\mathrm{nf}} I_2)$. For regular programs, this formula is trivially true. Compared to fair interleaving, rule (27) introduces only a simple additional case in which α_1 is de-scheduled and not resumed again (formula $\mathbf{E}\, \exists\, \underline{x}.\ \alpha_1$ becomes a precondition that can be ignored).

A second liveness rule is often required when an unfairly interleaved component satisfies a local eventually property $\Diamond\, \varphi$. Intuitively, the rule permits counting globally how often a component i is scheduled until it locally satisfies φ. The rule introduces a fresh counter variable N which is decremented whenever i is scheduled. The other component as well as the system's environment leave N constant:

$$\begin{aligned} (\Diamond\, \varphi \wedge \alpha_1) \parallel_{\mathrm{nf}} \alpha_2 \ \leftrightarrow\ \exists\, N. \quad & \Box\, N' = N'' \\ & \wedge\ ((N = N' + 1\, \mathbf{until}\, \varphi \wedge \alpha_1) \parallel_{\mathrm{nf}} (\alpha_2 \wedge \Box\, N = N'))\ . \end{aligned} \tag{28}$$

Counter N reaches zero when component 1 reaches the state where φ holds. It never reaches zero if the unfair scheduler does not consider component 1 often enough. The rule is valid when α_1 never blocks (in blocked steps, the counter could not be decremented).

The following symmetric versions of (27) and (28) also hold (with $\underline{x} = \mathrm{free}(\alpha_2)$):

$$\begin{aligned} \alpha_1 \parallel_{\mathrm{nf}} \alpha_2 \quad \leftrightarrow \quad & (\exists\, B.\ \Diamond\, B \wedge (\alpha_1 \parallel_{\mathrm{nf}} B\colon \alpha_2)) \\ & \vee\ \alpha_1 \wedge \mathbf{inf} \wedge \Box\, \neg\, \mathbf{blocked} \wedge \mathbf{E}\, \exists\, \underline{x}.\ \alpha_2 \\ \alpha_1 \parallel_{\mathrm{nf}} (\Diamond\, \varphi \wedge \alpha_2) \ \leftrightarrow\ \exists\, N. \quad & \Box\, N' = N'' \\ & \wedge\ ((\alpha_1 \wedge \Box\, N = N') \parallel_{\mathrm{nf}} (N = N' + 1\, \mathbf{until}\, \varphi \wedge \alpha_2))\ . \end{aligned}$$

The liveness rules (27) and (28) for unfair interleaving (and their symmetric versions) are used in the verification of the decomposition theorem for lock-freedom from the next section (cf. Appendix C). Rules (25), (27) and (28) have been mechanically verified with the specification of RGITL in HOL [28].

7 The Decomposition of Lock-Freedom

Standard RG reasoning (where relies and guarantees are simply binary relations over states and do not involve temporal logic) can be used to prove global safety properties of concurrent systems in a process-local way. Verifying liveness properties is one of the benefits of having a temporal logic setting (as [4] also note). We have combined both techniques to derive practical, process-local proof obligations for the global progress property of lock-freedom [33]. Our approach uses binary rely-guarantee predicates to capture safety properties, and an additional binary

predicate U for interference specific to liveness. (Other approaches such as [42,17] extend rely-guarantee conditions to more general formulas to capture liveness.)

This section defines three important nonblocking progress properties – wait-freedom, lock-freedom and obstruction-freedom – and illustrates them on a simple example. Subsequently, it details the decomposition theorem for lock-freedom. Its soundness proof is given in Appendix C and the KIV proofs are also available online [29]. We assume $spawn_n$ from Figure 5 as the underlying concurrent system model in the following.

Nonblocking progress is relevant in various application domains. For instance, high-availability systems are expected to run for a long time, even with the possibility that individual components fail and should not cause a deadlock of the entire system. Another example is the construction of operations that are async-signal safe to ensure that process interrupts do not lead to deadlock. Nonblocking progress is also of benefit in real-time systems, where prioritized threads must progress after a finite number of steps and nonblocking algorithms reduce the chance of long unpredictable delays.

Since nonblocking algorithms exclude the use of locks, we assume that the parameter operation $proc_m$ never blocks in the following:

$$proc_m(S) \vdash \Box \neg \textbf{blocked} \,. \tag{29}$$

Consequently, both seq_m and $spawn_n$ are also never blocked, i.e., the concurrent system never runs into a deadlock. Moreover, individual operations are executed in a safe environment which never violates safety properties R_m that are established by RG reasoning. Thus, we can define the following non-blocking progress notions.

An operation $proc_m$ is *wait-free* if it terminates in a safe environment:

$$proc_m(S), \Box\, R_m(S', S'') \vdash \Diamond\ \textbf{last} \,. \tag{30}$$

An operation is *obstruction-free* if it terminates when it eventually runs without interference from its (safe) environment:

$$proc_m(S), \Box\, R_m(S', S'') \vdash (\Diamond \Box\, S' = S'') \rightarrow \Diamond\ \textbf{last} \,. \tag{31}$$

The concurrent system $spawn_n$ is *lock-free* if infinitely often one of its interleaved operations progresses:

$$\begin{aligned} &spawn_n(S, Actf),\ \Box\ (S' = S'' \wedge Actf' = Actf'') \vdash \Box \Diamond\ Pto_n(Actf, Actf') \\ &\text{where } Pto_n(Actf, Actf') \equiv \exists\ m \leq n.\ P_m(Actf, Actf') \\ &\text{and } P_m(Act, Actf') \equiv Actf_m \wedge \neg\ Actf'_m \,. \end{aligned} \tag{32}$$

In formula (32), the global environment never alters the system state. The activity function $Actf$ is used to specify progress: global progress Pto_n requires that some spawned operation seq_m progresses P_m, i.e., it sets its activity flag from true to false (right after $proc_m$ terminates).

In contrast to wait-freedom and obstruction-freedom, which are progress properties of individual operations, lock-freedom is a global progress property that takes all interleaved operations into account. Hence, it is typically harder to prove directly and a practical decomposition theorem is desirable.

Example The following implementation of parameter operation $proc_m$ illustrates all three nonblocking properties. It is a simple implementation of a shared counter

N that offers a concurrent increment operation. (In this example, the abstract variable S is instantiated with variable N and sort *state* is natural numbers):

$$
\begin{array}{l}
proc_m(N)\{ \\
\textbf{let } Done = \text{false},\ M = 0 \textbf{ in} \\
\quad \textbf{while } \neg\ Done \textbf{ do } \{ \\
\quad\quad M := N; \\
\quad\quad \textbf{if}^* M = N \textbf{ then } \{N := M + 1,\ Done := \text{true}\} \textbf{ else skip } \}\} \ .
\end{array}
$$

The algorithm assumes that the counter cannot be atomically updated but must be loaded into a local snapshot M first. The second instruction checks whether other processes have updated N in the environment transition following the assignment $M := N$. If this is not the case, the counter is updated with the incremented value. Otherwise the local flag *Done* indicates that the attempt to increment was not successful and must be repeated. The use of **if*** makes the compare-and-swap an atomic step (atomic CAS instructions are supported by many processors).

Informally, the operation is not wait-free (30) since the repeated concurrent increment of N can lead to its starvation. It is obstruction-free (31) though since running the program in isolation $\Box\ N' = N''$ ensures its termination after at most one retry of the loop. Concurrently executing such operations is lock-free since if an individual loop must be retried, then some other operation has succeeded to increment N and to progress. However, a formal argument why the interleaving of such operations is really lock-free (32), is not immediately obvious.[8]

In the following, two simple local termination conditions for each individual $proc_m$ are defined which are sufficient to ensure lock-freedom of $spawn_n$. The first termination condition requires that $proc_m$ terminates whenever it does not suffer from critical interference from its environment, which is specified using an additional binary predicate U ("unchanged"). (In the example above, critical interference is the concurrent increment of N):

$$proc_m(S), \Box\ R_m(S', S'') \vdash \Diamond\ \Box\ U(S', S'') \rightarrow \Diamond\ \textbf{last} \ . \tag{33}$$

The reflexive and transitive binary relation $U : state \times state$ captures this critical interference formally. Predicate U specifies under which conditions termination of $proc_m$ can be guaranteed. These conditions are specific for termination and cannot be covered by rely conditions. While rely conditions R_m are safety properties that are never violated, the progress conditions U can be repeatedly violated during the execution of an operation.

The second termination condition, however, enforces that each $proc_m$ causes only a bounded amount of interference, i.e., it violates U only a finite number of times (in the example above, N is changed at most once). Formally, executions where U is violated by transitions of $proc_m$ infinitely often are disallowed:

$$proc_m(S), \Box\ R_m(S', S'') \vdash \Box\ \Diamond\ \neg\ U(S, S') \rightarrow \Diamond\ \textbf{last} \ . \tag{34}$$

Theorem 7 (Decomposition of Lock-Freedom)
Given the preconditions of Theorem 6 and the local termination conditions (33) and (34), it follows that the concurrent system $spawn_n$ is lock-free (32).

[8] Interestingly, [37] describes a compositional proof of the slightly stronger global property $\Box\ \Diamond\ N' > N$ for this example with ACL2.

The proof of Theorem 7 is given in Appendix C. Applying the decomposition theorem to prove lock-freedom for the example program above is simple. The termination conditions (33) and (34) can be easily shown with U defined as the identity relation over N and using no RG (safety) restrictions. (Locality of *Done* is implicit, as it is introduced using **let**).

The definition of lock-freedom considers unfair interleaving of operations, where individual operations might never be resumed again. In this case the remaining operations must still ensure progress. All intervals of the concurrent system model $spawn_n$ are nonblocking and infinite. Thus, the unfair scheduler can preempt each seq_m at any time during its execution and never reschedule it again (cf. the case of liveness rule (27) where component 1 gets discarded). This simulates the crash of a process. However, we do not explicitly consider crashes. In particular, we do not consider the case in which the entire system is never scheduled again (environment transitions always terminate), which would correspond to a system crash. Of course, proving progress for a crashed system is impossible. As a nice side effect, the local proofs of termination of a concrete lock-free algorithm do not deal with possible crashes (nor with non-terminating environment transitions). These are simulated in the generic decomposition proof only by the unfair scheduling.

The local proof obligation for lock-freedom given in the short version [48] of this paper implies (33) and (34). In particular, (34) is more general than the corresponding proof obligation there and can be instantiated for a wider range of concrete lock-free algorithms. While the proof obligation from [48] requires that each $proc_m$ violates U at most *once* (which is true for several interesting algorithms), termination condition (34) is more liberal. It demands that only a *finite* number of such violations may occur. We have instantiated these conditions to verify lock-freedom for several challenging examples from the literature (cf. Section 2.1).

8 Related Work

Relation to other Temporal Logics RGITL is based on ITL [38, 39, 41], adopting the concept of having finite and infinite intervals to distinguish between terminating and non-terminating runs of programs. The concept of having explicit programs as formulas also comes from ITL, and we have combined it with the idea of using frame assumptions, as defined in TLA [30].

Compared to TLA, the logic RGITL does not have built-in stuttering. This has some positive consequences, but also a drawback for refinement. On the positive side, no special measures (e.g., adding a "tick" event or a "final" label) are necessary to distinguish termination from infinite stuttering. Full TL can be used instead of excluding all formulas that are not stutter-invariant. On the negative side, refinement of an abstract program α to a concrete program β (trace inclusion, expressed as $\beta \rightarrow \alpha$) must often explicitly add or remove stuttering.

Programs are not encoded as transition systems, neither for specification nor for verification purposes. Therefore we follow the *exogenous* approach that has explicit programs in the calculus, rather than the *endogenous* approach (c.f. [45] and Section 4.5 of [20] for arguments in favor and against both approaches).

Using programs for specification is common; the simple programming language given in [31] or the input languages of some model checkers (e.g., Promela used

in SPIN [22]) offer programs for specification (although recursive procedures are often missing). All other tools that offer support for interleaved programs which we are aware of, however, encode programs as transition systems for verification purposes, e.g., STeP [7].

For fully automated model checkers, where the restriction on finiteness of the model can be exploited to generate counterexamples, this is the right choice. For interactive verification, where proof goals must be read and analyzed to find out how to continue or to find the reason for a failed proof, we (like [18], p. 135) find such encodings rather problematic.

RGITL is based on higher-order rather than first-order logic. Apart from the general advantages of being more expressive (e.g., well-foundedness can be directly defined as a formula), the use of higher-order notation as a basis makes it possible to define parametrized systems (e.g., the spawn procedure of Figure 5) with local variables without extending the language (as done, e.g., in [31]). Local variables of one process p become $lv(p)$ using a function variable lv.

Symbolic Execution The principle of proving sequential programs with symbolic execution is to our knowledge originally from [26, 10]. KIV's weakest-precondition calculus for sequential programs [46] is based on forward symbolic execution, and many other theorem provers also use this principle (e.g., [23]).

Interleaving In general, there are two ways of defining parallel composition: Either parallel composition is defined as a primitive operator in the syntax and is given an interleaving semantics [49, 54, 9, 53], similar to our approach, or it is defined (roughly) as the conjunction of the behaviors of the subcomponents [2, 40, 55, 19, 15].

We follow the first approach; our extension of the semantics that distinguishes between system and environment steps is similar to reactive sequences in [47]. However, these may begin and end with an environment step, while the intervals defined here start with a system step. The semantics is also somewhat similar to the semantics of modules in the simple programming language given in [31]. However, a program and its environment alternate steps such that individual environment steps are not visible (in contrast to Aczel-traces [47]). Sequences of explicit steps of other processes (even infinite ones if the interleaving is unfair) are visible in RGITL only when programs are interleaved. However, in the *local* view of one process, they appear again as just one environment step.

Our approach has the advantage of moving much of the complexity into the semantics. However, the complexity of the interleaving operator makes it more difficult to get a completeness proof, and it fixes a particular interpretation of concurrency. Different concepts of concurrency (e.g., asynchronous parallelism) need a different semantics foundation as well as the derivation of new rules.

We have not followed the approach of defining concurrency as conjunction since it has the following disadvantages:

- Conjunction can be used easily only when the variables modified by each component are disjoint (as in TLA [2]).
- Stuttering must be added.
- Fairness constraints must be used as separate, additional constraints.
- Auxiliary variables are needed to express termination.

When interleaved programs use shared variables, a definition of interleaving as conjunction is incompatible with a simple definition of assignment as $(N := t) \equiv N' = t \wedge \mathbf{step}$ (ignoring the issue of frame assumptions in this discussion). Otherwise, $N := 3 \parallel N := 4$ is equivalent to false, and $N := N + 1 \parallel N := N + 1$ might increment N once instead of twice.

The literature solves this problem by adding stuttering to the definition of assignment, and by adding an auxiliary variable P ("process identifier") [55] (or equivalently, boolean variables [19]) that determines which process executes which step. Programs are then indexed by the process executing them (indicated here as a subscript $P = i$). The definition of an atomic assignment then is

$$(N := t)_{P=i} \equiv ((\mathbf{step} \wedge P \neq i)^* \wedge \mathbf{fin}); (P = i \wedge N' = t \wedge \mathbf{step}) \tag{35}$$

(the **fin** constraint ensures termination). A definition of interleaving via conjunction must also avoid the inconsistency that results from interleaving a terminating program α with a diverging program β (an interval cannot be finite as well as infinite). This is usually done by adding final stuttering. However, termination then can no longer be expressed compositionally as $\Diamond$ **last**. Instead, two more boolean variables T_1 and T_2 ("tick" events) must be added that express termination of each individual program (so overall termination becomes $\Diamond\ (T_1 \wedge T_2)$). Top level interleaving of two programs then can be defined roughly (a frame assumption is necessary for T_1 and T_2) as:

$$\begin{aligned} \alpha_1 \parallel \alpha_2 \equiv \quad & (T_1 = \text{false} \wedge (\alpha_1; T_1 := \text{true})_{P=1}; \mathbf{step}^*) \\ & \wedge\ (T_2 = \text{false} \wedge (\alpha_2; T_2 := \text{true})_{P=2}; \mathbf{step}^*)\ . \end{aligned}$$

Interleaving defined in that way is typically restricted to concurrency on the top level (otherwise many process identifiers would be necessary). This kind of interleaving is still unfair and cannot easily handle fairness in conjunction with blocking. It seems that another auxiliary boolean variable Blk_i would be necessary that is managed by each process to express whether a program is blocked (or equivalently: not enabled). Only with such a variable does weak fairness become expressible using a global constraint $(\Diamond\Box\neg\ Blk_i) \rightarrow \Box\Diamond P = i$ for each i.

In summary, auxiliary variables and global constraints will be necessary and have to be managed by the prover when interleaving is defined as a derived operator based on conjunction. The approach is not compositional in the strong sense of our compositionality rule (1). In contrast, our calculus uses no such global constraints, and introduces an auxiliary variable only when a proof requires fairness.

A definition of assignment as (35) is also difficult to handle for symbolic execution. In liveness proofs, our calculus would require reasoning about induction over the number of stutter steps before the effect of the assignment takes place. Note that a proof of a safety property $\Box\varphi$ with rule (17) is still easy: The induction hypothesis is applicable after one stuttering since the remaining program is unchanged. Even the **fin** constraint in (35) is irrelevant for safety proofs.

Atomicity Somewhat related to the above discussion is the question of atomicity assumptions. Our calculus currently supports atomic assignments and other atomic primitives like locking, or CAS using **await** and **if***. This assumption is not realistic for assignments like $X := Y + Z$ when all three variables are shared: they would be executed as two read instructions and a write instruction on real

processors. We therefore typically require assignments and the tests of conditionals in shared-variable programs to reference one shared variable only. (This is the LCR requirement from [31]). The above non-atomic assignment would therefore have to be translated to $LY := Y; LZ = Z; X := LY + LZ$ in our logic using intermediate local variables LY and LZ. An extension that does this translation on the fly would be possible.

The translation above still assumes that elementary reads and writes are atomic, and assumes a sequentially consistent memory model. RGITL does not support weak memory models [3], which make reordering of assignments possible. We have, however, considered non-atomic reading and writing (where reading interleaved with writing gives an unspecified result) in one case study: Hazard pointers [34] can be read and written non-atomically without making the approach incorrect. Our solution uses two abstract procedures for reading/writing the value of the relevant variable. The procedures have no implementation; they are specified to a) terminate, and to b) read/write a meaningful value only if the environment never changes the value while they run. ([48] gives details and the full proofs are online [29].)

Induction Lazy induction is closely connected to verification diagrams [32]. The nodes of our proof graphs correspond to assertions given in the nodes of verification diagrams; the step rule corresponds to executing a transition, where the disjuncts from the symbolic execution step of the program correspond to the different edges in the diagram. The rank functions in [32] correspond to our well-founded orders. Verification diagrams for safety properties implicitly use the well-founded order that we make explicit by counting symbolic execution steps.

An important difference is that our step rule computes the *strongest* postcondition, while the proof obligation associated with an edge of a verification diagram proves that the assertion of the target node is *any* postcondition. To mimic this, we would have to use the weakening rule of sequent calculus. We found that such weakening steps are typically necessary only for the states before loops (and recursion), where the precondition has to be generalized to an invariant. Because of this, our proof graphs can *dynamically* generate many parts of verification diagrams automatically instead of having to predefine all intermediate assertions manually.

Compositionality Compositionality results for concurrent programs are of central importance for the handling of big case studies. There are many compositionality theorems for specific properties or specific systems (e.g., the theorem for TLA [1] assumes programs which assign to disjoint variables, i.e., have disjoint frame assumptions). Rely-Guarantee reasoning for partial correctness goes back to [24, 25], and to [36] for programs with send/receive primitives. RG reasoning has also been studied in [13] for non-atomic assignments.

Many variations of rely-guarantee rules exist, several of which are summarized and systematically presented in Xu et al. [54]. RG assertions there are written as

$$proc \ \underline{sat} \ (pre, rely, guar, post) \ .$$

A translation of these to RGITL is:

$$pre, proc \vdash rely^* \xrightarrow{+} (\textbf{if last then } post \textbf{ else } guar) \ . \tag{36}$$

Xu et al.'s rely predicates are not required to be transitive since the semantics is based on the fine-grained Aczel-trace semantics, which has several environment transitions in between system transitions. The translation has to use the transitive closure $rely^*$ to adapt to our setting which summarizes such steps as one environment step. Xu et al. also give stronger rules that ensure total correctness and absence of deadlock respectively. We can express total correctness of programs w.r.t. a rely-guarantee specification as[9]

$$pre, proc \vdash (rely^* \xrightarrow{+} guar) \wedge (\Box\ rely \rightarrow \Diamond\ (\mathbf{last} \wedge post)) \tag{37}$$

To modularly ensure absence of deadlock (which is a safety property) , Xu et al. use an additional *runs*-predicate for each process which characterizes unblocked program states. A similar encoding can be used in our setting by introducing such a predicate and adding $runs \rightarrow \neg\ \mathbf{blocked}$ to the guarantee.

$$pre, proc \vdash rely^* \xrightarrow{+} (guar \wedge (runs \rightarrow \neg\ \mathbf{blocked}))$$

We can then *derive* in RGITL that (under some simple side conditions for *runs*) an interleaved system in which processes do not block in their runs-states indeed never deadlocks.

Fairness The treatment of weak-fairness in RGITL seems to be unique, as other theorem provers typically use extra temporal logic constraints in addition to transition systems. This is more flexible (strong fairness and other notions of fairness may be defined, and fairness constraints may be given for individual transitions), but it is not an option if one does not want to encode programs as transition systems. On the negative side it generates many extra formulas. The use of counters in the logic is inspired by the alternative of using auxiliary variables to encode fairness, as described in [4]. However, we do not use such counters a priori but instead introduce them on the fly with liveness rules.

Lock-Freedom An automated decomposition technique for lock-freedom somewhat similar to ours [50] is given in [17]. The main idea there is that lock-freedom (32) can be reduced to the proof of termination of a reduced system $spawn_k^{red}$ that runs in isolation and where the infinite iterations of each process are removed, i.e., each seq_m calls $proc_m$ just once. It is argued informally that this reduction is correct, while we derive the correctness of our decomposition. Moreover, a calculus is defined to reduce the termination of $spawn_k^{red}$ to local termination conditions for $proc_m$ based on RG rules with *temporal* RG conditions, whereas our conditions are instead just binary predicates.

9 Conclusion

This paper gives a self-contained description of a general framework RGITL for interactive verification of interleaved programs. RGITL extends the well-known

[9] The implementation in KIV offers a special syntax for the important class of RG assertions (36) and (37) written as $pre \vdash [rely, guar, inv, proc]\ post$ and $pre \vdash \langle rely, guar, inv, proc\rangle\ post$, respectively. The additional predicate *inv* is used to encode single-state invariants.

principles for verifying sequential programs based on symbolic execution and induction to a setting with interleaved programs, recursive procedures and full temporal logic based on higher-order logic. One of the key features of the framework is the compositional semantics of interleaving that makes it possible to prove compositionality theorems. The framework is implemented in the interactive theorem prover KIV, which is available for downloading at [27].

There are still some open problems. Finding good proof rules for an elegant verification of refinement "modulo stuttering" (as in TLA but for arbitrary programs, not just transition systems) is still an open issue that is of great practical relevance. Moreover, from the theoretical point of view, completeness of the logic is still an open issue, although the calculus includes two complete fragments: Moszkowski's ITL axioms [41] and the RG calculi from Xu et al. [54].

Acknowledgment

We would like to thank Cliff Jones and Ben Moszkowski for thoroughly reading a preliminary version of this paper, and in addition Dimitar Guelev, Alexander Knapp and the anonymous referees for fruitful discussions that helped to improve this work.

References

1. Abadi, M., Lamport, L.: Composing specifications. In: J.W. de Bakker, W.P. de Roever, G. Rozenberg (eds.) Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness, vol. 430, pp. 1–41. Springer LNCS, Berlin, Germany (1989)
2. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems, pp. 507–534 (1995)
3. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29**, 66–76 (1995)
4. Apt, K.R., de Boer, F., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd edn. Springer (2009)
5. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications **23**(2,3), 285–307 (2010)
6. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. FAC J **23**(1), 91–112 (2011)
7. Bjørner, N., Manna, Z., Sipma, H., Uribe, T.: Deductive verification of real-time systems using STeP. Theoretical Computer Science **253**(1) (2001)
8. Börger, E., Stärk, R.F.: Abstract State Machines — A Method for High-Level System Design and Analysis. Springer (2003)
9. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1-3), 227–270 (2007)
10. Burstall, R.M.: Program proving as hand simulation with a little induction. Information Processing 74 pp. 309–312 (1974)
11. Cau, A., Moszkowski, B.: Using PVS for Interval Temporal Logic proofs. Part 1: The syntactic and semantic encoding. Tech. rep., De Montfort University (1996)
12. Cau, A., Moszkowski, B.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, De Montfort University, UK (2013). `www.tech.dmu.ac.uk/STRL/ITL` (Accessed 1 July 2013)
13. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Logic and Computation **17**, 807–841 (2007)
14. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE 2004, *LNCS*, vol. 3235, pp. 97–114 (2004)
15. Dongol, B., Derrick, J., Hayes, I.J.: Fractional permissions and non-deterministic evaluators in interval temporal logic. ECEASST **53** (2012)

16. Floyd, R.W.: Assigning meanings to programs. In: J.T. Schwartz (ed.) Proceedings of the Symposium on Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
17. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that nonblocking algorithms don't block. In: POPL, pp. 16–28. ACM (2009)
18. Groves, L.: Verifying michael and scott's lock-free queue algorithm using trace reduction. In: Proc. CATS '08, CATS '08, pp. 133–142. Australian Computer Society, Inc. (2008)
19. Guelev, D.P., Dang Van Hung: Prefix and projection onto state in Duration Calculus. Electr. Notes Theor. Comput. Sci. **65**(6), 101–119 (2002)
20. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
21. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Prog. Languages and Systems **12**(3), 463–492 (1990)
22. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison Wesley (2003)
23. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, KU Leuven (2008)
24. Jones, C.B.: Development Methods for Computer Programs Including a Notion of Interference. Ph.D. thesis, Oxford University (1981). Available as Programming Research Group Technical Monograph 25
25. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, pp. 321–332. North-Holland (1983)
26. King, J.C.: A Program Verifier. Ph.D. thesis, Carnegie Mellon University (1970)
27. KIV Download.
URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv` (2012). Accessed 1 September 2013
28. KIV: Presentation of a higher-order specifications of RGITL.
URL: `http://www.informatik.uni-augsburg.de/swt/projects/RGITL.html` (2012). Accessed 1 September 2013
29. KIV: Presentation of proofs for concurrent algorithms in RGITL.
URL: `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html` (2013). Accessed 1 September 2013
30. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994). DOI http://doi.acm.org/10.1145/177492.177726
31. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems – Safety. Springer (1995)
32. Manna, Z., Pnuelli, A.: Temporal verification diagrams. In: M. Hagiya, J. Mitchell (eds.) International Symposium on Theoretical Aspects of Computer Software, vol. 789, pp. 726–765. Springer Verlag (1994)
33. Massalin, H., Pu, C.: A Lock-Free Multiprocessor OS Kernel. Tech. Rep. CUCS-005-91, Columbia University (1991)
34. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
35. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)
36. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE, Trans. on Software Eng. **7**, 417–426 (1981)
37. Moore, J.S.: A mechanically checked proof of a multiprocessor result via a uniprocessor view. Form. Methods Syst. Des. **14**, 213–228 (1999)
38. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. IEEE Computer **18**(2), 10–19 (1985)
39. Moszkowski, B.: Executing Temporal Logic Programs. Cambr. Univ. Press (1986)
40. Moszkowski, B.: Compositional reasoning about projected and infinite time. In: Proceedings of the 1st ICECCS, pp. 238–245. IEEE Computer Society (1995)
41. Moszkowski, B.: An automata-theoretic completeness proof for Interval Temporal Logic. In: ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming, pp. 223–234. Springer-Verlag, London, UK (2000)
42. Moszkowski, B.: Interconnections between classes of sequentially compositional temporal formulas. Information Processing Letters **113**(9), 350–353 (2013)
43. Nafz, F., Seebach, H., Steghöfer, J.P., Bäumler, S., Reif, W.: A formal framework for compositional verification of organic computing systems. In: Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC 2010), pp. 17–31. Springer, LNCS (2010)

44. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters **12**(3), 115–116 (1981)
45. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Ann. IEEE Symp. on the Foundation of Computer Science (FOCS), pp. 46–57. IEEE Computer Society Press (1977)
46. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: W. Bibel, P. Schmitt (eds.) Automated Deduction—A Basis for Applications, vol. II, pp. 13–39. Kluwer, Dordrecht (1998)
47. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. No. 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
48. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of the 18th Int. Symposium on Temporal Representation and Reasoning (TIME), IEEE Computer Society Press, pp. 99–106 (2011)
49. Stølen, K.: A method for the development of totally correct shared-state parallel programs. In: In CONCUR'91, vol. 527, pp. 510–525. Springer LNCS (1991)
50. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: Proc. of MPC 2010, Springer LNCS 6120, pp. 377–396 (2010)
51. Tofan, B., Schellhorn, G., Ernst, G., Pfähler, J., Reif, W.: Compositional Verification of a Lock-Free Stack with RGITL. In: Proc. of International Workshop on Automated Verification of Critical Systems (to appear in ECEASST) (2013)
52. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Proc. ICTAC, pp. 239–255. Springer LNCS 6916 (2011)
53. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR, *Springer LNCS*, vol. 4703, pp. 256–271 (2007)
54. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. FAC J **9**(2), 149–174 (1997)
55. Xu, Q., Swarup, M.: Compositional reasoning using the assumption-commitment paradigm. Lecture Notes in Computer Science **1536**, 565–583 (1998)

A Proof of Transformation to Normal Form

Proof (of Theorem 2) By induction over the structure of the formula. In all cases where the top-level operator is not temporal or is **last** or **blocked**, the formula is already a step-formula by the definition of regular formulas. The cases of LTL operators are easy. They rely on simple unwinding properties:

$$\text{until: } \varphi \textbf{ until } \psi \leftrightarrow \psi \vee (\varphi \wedge \circ (\varphi \textbf{ until } \psi)) .$$

Although not strictly necessary for the proof, we also provide unwinding rules for the derived LTL operators:

$$\begin{aligned} \text{wnext: } & \bullet \varphi \leftrightarrow \textbf{last} \vee \circ \varphi \\ \text{always: } & \Box \varphi \leftrightarrow \varphi \wedge \bullet \Box \varphi \\ \text{eventually: } & \Diamond \varphi \leftrightarrow \varphi \vee \circ \Diamond \varphi . \end{aligned}$$

Assuming that φ and ψ are in step form already, then the right-hand side is also in step form. Path quantifiers require new static variables $\underline{u}$ for the free flexible variables $\underline{x}$ in φ (if there are some):

$$\text{allpath: } \mathbf{A}\ \varphi \leftrightarrow (\forall\ \underline{u}.\ \underline{x} = \underline{u} \rightarrow \mathbf{A}\ (\forall\ \underline{x}.\ \underline{x} = \underline{u} \rightarrow \varphi)) .$$

The **A**-formula on the right-hand side is a static formula and therefore a step-formula. Recall that a regular formula uses the chop and star operator and procedure calls in regular programs only. Therefore, the only remaining case is where the formula is actually a program. This case is more difficult and follows from the next theorem since any formula in "normal form" is also in step form. □

Theorem 8 (Transformation of Programs into Normal Form) *Any regular program can be transformed into an equivalent formula of the following normal form*

$$[\alpha]_{\underline{x}} \equiv (\tau_0 \wedge \mathbf{last}) \vee \bigvee_{i=1}^{n} (\exists\, \underline{v}_i.\ \tau_i \wedge b_i \wedge \circ\, \varphi_i)\ , \tag{38}$$

where all τ_i are step-formulas ($i = 0..n$) and $b_i \in \{\mathbf{blocked}, \neg\ \mathbf{blocked}\}$ for $i = 1..n$. The first disjunct with τ_0 may be missing and $n = 0$ is possible.

The first disjunct is needed for programs that may immediately terminate, e.g., unwinding $\mathbf{while}^*\varphi\ \mathbf{do}\ldots$ gives such a case with $\tau_0 \equiv \neg\ \varphi$. All the other disjuncts describe possible first steps τ_i of the program that are followed by others, where φ_i is either the remaining program or **last**. The steps may be blocked or unblocked, according to b_i. An assignment generates one unblocked step (τ_0 is missing, $n = 1$, $b_1 = \neg\ \mathbf{blocked}$), and a conditional gives two disjunctive clauses for its two branches. The existentially quantified variables v_i are needed to handle existential quantifiers over local variables introduced by **let** and **choose**.

Proof (of Theorem 8) By induction over program structure. The definitions of assignment (2) and **skip** (3) are already in normal form when **step** is expanded. The cases of conditionals (4, 5) are easily handled by recalling that the test is a state formula and therefore also a step-formula. An **await** statement (8) can be directly transformed into normal form using the unwinding rule

$$\text{await:}\ \ [\mathbf{await}\ \varphi]_{\underline{x}} \leftrightarrow (\varphi \wedge \mathbf{last}) \vee (\neg\ \varphi \wedge \mathbf{blocked} \wedge \circ\, [\mathbf{await}\ \varphi]_{\underline{x}})\ .$$

Formulas **let** (9) and **choose** (10) are simple too since they just require moving the existential quantifier into the disjunction of the normal form of α, and unwinding $\Box\ \underline{y'} = \underline{y''}$. While loops reduce to the star operator and a compound by formulas (6) and (7). The star operator itself also reduces to a compound by unwinding:

$$\text{star:}\ \ [\alpha^*]_{\underline{x}} \leftrightarrow \mathbf{last} \vee [(\alpha \wedge \neg\ \mathbf{last}); \alpha^*]_{\underline{x}}\ .$$

This leaves compounds $\alpha; \beta$ and interleavings as the only difficult cases. A compound is transformed into normal form using the rules from Table 1. They assume that α is already in normal form (38). First, the compound is distributed over the disjunction and the existential quantifiers using rules *comp-dis* and *comp-ex* (a bounded renaming of v to v_0 is necessary when v occurs free in ψ):

$$[\alpha; \beta]_{\underline{x}} \equiv (\tau_0 \wedge \mathbf{last}); [\beta]_{\underline{x}} \vee \bigvee_{i=1}^{n} \exists\, \underline{v}_i.\ (\tau_i \wedge \circ\, \varphi_i); [\beta]_{\underline{x}}\ .$$

When the first disjunct exists, α may terminate immediately. In this case, β must also be transformed into normal form since its first step is executed. Technically, this is done after applying rule *chp-lst*. The right-hand side of the rule removes primes and double primes from flexible variables since τ holds for a last state. The substitution also replaces possible occurrences of **last** and **blocked** with true and false, respectively. For the other disjuncts ($i = 1..n$), applying rule *chp-stp* immediately gives the normal form.

The rules for transforming weak fair and unfair interleaving are exactly the same since the fairness of the schedule is irrelevant when considering a *finite*

chp-dis:	$(\varphi_1 \vee \varphi_2); \psi$	$\leftrightarrow$	$(\varphi_1; \psi) \vee (\varphi_2; \psi)$
chp-ex:	$(\exists\, v.\ \varphi); \psi$	$\leftrightarrow$	$\exists\, v_0.\ \varphi_v^{v_0}; \psi$, where $v_0 \notin (\mathrm{free}(\varphi) \setminus \{v\}) \cup \mathrm{free}(\psi)$
chp-lst:	$(\tau \wedge \mathbf{last}); \psi$	$\leftrightarrow$	$\tau^{\underline{x},\underline{x}}_{\underline{x}',\underline{x}''} \wedge \psi$, where $\underline{x} = \mathrm{free}(\tau) \cap X$
chp-stp:	$(\tau \wedge \circ\, \varphi); \psi$	$\leftrightarrow$	$\tau \wedge \circ\, (\varphi; \psi)$

Table 1 Rules for computing the normal form of a compound

number of initial steps (additional rules for fairness are given in Section 6). The rules imitate the semantic clauses from Definition 1 using an auxiliary operator $\varphi_1 \parallel^< \varphi_2$, which gives precedence to executing φ_1. Three others are defined as abbreviations:

$I \models \varphi \parallel^< \psi$ iff there are I_1, I_2, sc :
$I_1 \models \varphi$ and $I_2 \models \psi$ and $(I, sc) \in I_1 \oplus I_2$ and sc is fair
and: if $sc \neq ()$ then $sc(0) = 1$ else $\#I_1 = 0$

$\varphi \parallel_b^\leq \psi \equiv (\mathbf{blocked} \wedge \circ\, \varphi) \parallel^< \psi$

$\varphi \parallel^> \psi \equiv \psi \parallel^< \varphi$

$\varphi \parallel_b^\geq \psi \equiv \psi \parallel_b^\leq \varphi$.

Operator $\varphi \parallel_b^\leq \psi$ handles the last of the three cases of Definition 1, where I_1 is blocked. It is assumed that φ describes the rest of I_1 with the blocked step already removed. The first step of I is the first step of (nonempty) I_2.

The rules needed for symbolic execution are summarized in Table 2. Transformation into normal form starts with rule *ilv* that gives two symmetric cases for left and right interleaving.

By the induction hypothesis we may assume that program (or formula) φ in $\varphi \parallel^< \psi$ can be transformed into normal form:

$$\varphi \;\equiv\; (\tau_0 \wedge \mathbf{last}) \vee \bigvee_{i=1}^{n} (\exists\, \underline{u}_i.\ \tau_i \wedge b_i \wedge \circ\, \varphi_i)\ .$$

The compositionality of interleaving makes it possible to replace φ with the right-hand side of the equivalence in the formula $\varphi \parallel^< \psi$. The resulting disjunction and existential quantifiers can be pulled out of the interleaving with rules *ilvl-dis* and *ilvl-ex*. This gives a first disjunct $(\tau_0 \wedge \mathbf{last}) \parallel^< \psi$ that is suitable for applying rule *ilvl-lst*. This case corresponds to case 1 of Definition 1 (I_1 is empty since **last** holds). In the result, formula ψ can be reduced to normal form by applying the induction hypothesis.

For the other disjuncts, depending on whether b_i is unblocked, one of the rules *ilvl-stp* or *ilvl-blk* is applicable. Note that a key issue for the correctness of the two rules is that they both introduce new static variables $\underline{u}$ that store the values $I_1(1)(\underline{x})$ which might be referenced as $\underline{x''}$ in τ. While the first system step is done by the left program, so variables $\underline{x}$ and $\underline{x'}$ agree between the local run I_1 and the global I, this is not true for $\underline{x''}$. The local computation continues with these values but the global state $I(1)$ may be different from $I_1(1)$.

Rule *ilvl-stp* executes an unblocked step of φ and corresponds to case 2 of Definition 1. It already gives a disjunct of the final normal form. In rule *ilvl-blk*

ilv:	$\varphi \parallel \psi$	$\leftrightarrow (\varphi \parallel^{<} \psi) \vee (\varphi \parallel^{>} \psi)$
ilvl-dis:	$(\varphi_1 \vee \varphi_2) \parallel^{<} \psi$	$\leftrightarrow (\varphi_1 \parallel^{<} \psi) \vee (\varphi_2 \parallel^{<} \psi)$
ilvl-ex:	$(\exists\, v.\ \varphi) \parallel^{<} \psi$	$\leftrightarrow \exists\, v_0.\ (\varphi_v^{v_0} \parallel^{<} \psi)$ where $v_0 \notin (\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi)$
ilvl-lst:	$(\tau \wedge \textbf{last}) \parallel^{<} \psi$	$\leftrightarrow \tau^{\underline{x},\underline{x}}_{\underline{x}',\underline{x}''} \wedge \psi$ where $\underline{x} = \text{free}(\tau) \cap X$
ilvl-stp:	$(\tau \wedge \neg\, \textbf{blocked} \wedge \circ\, \varphi) \parallel^{<} \psi$	$\leftrightarrow \exists\, \underline{u}.\ (\tau^{\underline{u}}_{\underline{x}''} \wedge \neg\, \textbf{blocked} \wedge \circ\, ((\underline{x} = \underline{u} \wedge \varphi) \parallel \psi))$ where $\underline{x} = \text{free}(\tau) \cap X, \underline{u}$ are new static variables.
ilvl-blk:	$(\tau \wedge \textbf{blocked} \wedge \circ\, \varphi) \parallel^{<} \psi$	$\leftrightarrow \exists\, \underline{u}.\ (\tau^{\underline{x},\underline{u}}_{\underline{x}',\underline{x}''}) \wedge ((\underline{x} = \underline{u} \wedge \varphi) \parallel^{<}_{b} \psi)$
ilvlb-dis:	$\varphi \parallel^{<}_{b} (\psi_1 \vee \psi_2)$	$\leftrightarrow \varphi \parallel^{<}_{b} \psi_1 \vee \varphi \parallel^{<}_{b} \psi_2$
ilvlb-ex:	$\varphi \parallel^{<}_{b} (\exists\, v.\ \psi)$	$\leftrightarrow \exists\, v_0.\ (\varphi \parallel^{<}_{b} \psi_v^{v_0})$ where $v_0 \notin (\text{free}(\psi) \setminus \{v\}) \cup \text{free}(\varphi)$
ilvlb-lst:	$\varphi \parallel^{<}_{b} (\tau \wedge \textbf{last})$	$\leftrightarrow$ false
ilvlb-stp:	$\varphi \parallel^{<}_{b} (\tau \wedge \neg\, \textbf{blocked} \wedge \circ\, \psi)$	$\leftrightarrow \exists\, \underline{u}.\ \tau^{\underline{u}}_{\underline{x}''} \wedge \neg\, \textbf{blocked} \wedge \circ\, (\varphi \parallel (\underline{x} = \underline{u} \wedge \psi))$
ilvlb-blk:	$\varphi \parallel^{<}_{b} (\tau \wedge \textbf{blocked} \wedge \circ\, \psi)$	$\leftrightarrow \exists\, \underline{u}.\ \tau^{\underline{x},\underline{u}}_{\underline{x}',\underline{x}''} \wedge \textbf{blocked} \wedge \circ\, (\varphi \parallel (\underline{x} = \underline{u} \wedge \psi))$

Table 2 Rules for symbolic execution of (both weak-fair and unfair) interleaving

the transition of the first component is blocked. Hence, unprimed and primed variables are equal and the first top level transition is determined by ψ according to case 3 of Definition 1. By the induction hypothesis, ψ can be transformed into normal form. Again, disjunctions and existential quantifiers distribute over the blocked interleaving according to rules *ilvlb-dis* and *ilvlb-ex*, respectively. Finally, rules *ilvlb-lst*, *ilvlb-stp* and *ilvlb-blk* give normal form in this case. The first rule gives false when I_2 is empty since this case is already considered when executing the second component. Again, rule *ilvlb-stp* and *ilvlb-blk* need static variables $\underline{u}$, as explained above.

□

Overall, the computations done by the first phase of symbolic execution are quite complex. Correctness of the first phase, however, depends only on the correctness of rewrite rules. Apart from simple ones for propositional logic and quantifiers, the interesting ones are those defining program constructs (formulas (2) to (10)) and those from Tables 1 and 2 for compounds and interleaving. To ensure that these are sound with respect to the semantics, we have mechanized their proofs in the embedding of RGITL in HOL [28].

B Proof of the Rely-Guarantee Theorem

To prove the correctness of rule (19) in Theorem 5 we first show that

$$[R_1 \xrightarrow{+} G_1] \parallel_{\text{nf}} [R_2 \xrightarrow{+} G_2] \vdash R \xrightarrow{+} G \tag{39}$$

can be derived from premises 2-6 of the theorem. Correctness of the rule then follows by an application of the compositionality rule (1), which makes it possible to substitute the local rely-guarantee properties for each procedure. Formally, the goal above is the last premise of the compositionality rule; the other two are obtained from premise 1 of rule (19) (with $i = 1$ and $i = 2$).

The proof of (39) is by induction over $R \xrightarrow{+} G$, using (20) and (16) to retrieve a counter from property $\Diamond\ B$. The resulting additional formulas for lazy induction are omitted in the following for brevity. Symbolic execution of one step executes the interleaving and applies (21) on all three $\xrightarrow{+}$ formulas. Symbolic execution of the succedent formula in (39) requires proving G for the first system step. This is simple since the first step of the system either satisfies G_1 or G_2, depending on which process executes a step. Then the leading operator $\bullet$ is removed, giving the original succedent again (together with the additional assumption that the first global environment step satisfies R, which is typically moved into the antecedent). Symbolic execution of the interleaving in the antecedent of (39) gives 6 cases, depending on whether the scheduled component terminates or executes an unblocked or a blocked step. We consider w.l.o.g. the first three, where the first component is scheduled.

Case 1: component 1 terminates. In this case the goal reduces to

$$R_2 \xrightarrow{+} G_2 \vdash R \xrightarrow{+} G\ ,$$

which is simple to prove (using premise 3 and 6 for $i = 2$).

Case 2: Component 1 executes an unblocked step. This gives two sub-cases. In the first one, the first local environment transition of component 1 satisfies R_1, then the resulting goal is again of the form (39) and the induction hypothesis closes it. Observe that property $R_1 \xrightarrow{+} G_1$ of component 1 ensures G_1 for its program step, which becomes globally visible, and premise $G_1 \rightarrow G$ of (19) ensures that the global guarantee G holds for this step, too.

The difficult second sub-case, when the first local environment transition violates R_1, must be impossible. A symbolic execution step in this case leads to

$$[u = S] \parallel_{\mathrm{nf}} [R_2 \xrightarrow{+} G_2],\ G_1(s_0, s_1),\ \neg\ R_1(s_1, u),\ R_1(s_1, S) \vdash R \xrightarrow{+} G\ , \tag{40}$$

where s_0 and s_1 is the state before and after the system step of the first component respectively. Formula $R_1(s_1, S)$ is implied by the global rely R from unwinding $R \xrightarrow{+} G$, and formula $\neg\ R_1(s_1, u)$ characterizes the first local environment transition of component 1. Hence, state u (introduced by rule *ilvl-stp* from Table 2) is the state after the first *local* environment step of component 1. Only if the next step is again by component 1, does state u become the next state of the global run ($u = S$), immediately closing the proof by the contradiction between $\neg\ R_1(s_1, u)$ and $R_1(s_1, S)$. In this case the first local environment step of component 1 is also the first global environment step.

Otherwise, when the next step is by component 2, further symbolic execution is necessary. The step introduces two static variables s_2, s_3 for S and S', which satisfy $G_2(s_2, s_3)$. Assumption $R_1(s_1, S)$ becomes $R_1(s_1, s_2)$. Also, $R(s_3, S)$ holds again from unwinding the sustains in the succedent. Together this implies $R_1(s_1, S)$

using $R \to R_1$, $G_2 \to R_1$, and transitivity of R_1. This is the core argument of the proof: a sequence of a global R-step, a G_2-step of the other component, and another global R-step together form a valid R_1-step.

The second symbolic execution step also introduces a new variable v for the state of component 2 after its local environment step. Again, it must be ensured that $R_2(s_3, v)$ holds. If it does, goal (40) is repeated, and the proof is closed by applying the induction hypothesis.

Otherwise symbolic execution gives

$$[u = S] \parallel_{\mathrm{nf}} [v = S], \neg\, R_1(s_1, u), R_1(s_1, S), G_2(s_2, s_3), \neg\, R_2(s_3, v), R_2(s_3, S) \qquad (41)$$
$$\vdash R \xrightarrow{+} G\,.$$

Now a further symbolic execution step closes the goal regardless whether component 1 or 2 is scheduled: both $u = S$ and $v = S$ give a contradiction in the antecedent.

Case 3: component 1 is scheduled but blocked. In this case component 2 executes a step, too. The proof then applies similar arguments but with both components switching roles. (When both relies are false immediately, the resulting goal is the special case of (41) with $s_3 = s_1$ and $v = u$.)

C Proof of the Decomposition Theorem for Lock-Freedom

This section details the proof of Theorem 7, which applies Theorem 6 (for RG reasoning), the compositionality of the chop, star and interleaving operator (cf. rule (1)), the liveness rules (27), (28) for induction, resp. their symmetric versions, and symbolic execution of the (unfair) interleaving operator. (Here only the case of unblocked steps is relevant since the interleaved operations seq_m neither block nor terminate.) First, some simple preliminary definitions and lemmas are given, mainly to enable an inductive liveness proof for the interleaving of one process seq_{m+1} with the other processes $spawn_m$. Furthermore, since each process only guarantees liveness under certain rely conditions R_m, liveness and RG behavior of the processes must be merged.

We start by strengthening the safety assumptions R_m and the system rely Rto_n from Theorem 6 to include that activity entries $Actf_m$ are local:

$$R_m^{act}(S', Actf', S'', Actf'') \equiv R_m(S', S'') \wedge Actf'_m = Actf''_m$$
$$Rto_n^{act}(S', Actf', S'', Actf'') \equiv \forall\, m \le n.\ R_m^{act}(S', Actf', S'', Actf'')\,.$$

It is straightforward to derive that each process m never changes any activity flag other than $Actf_m$, by applying rule (22) on seq_m. This property is denoted as $\lceil Actf_{=m} \rceil$ in the following. It is also simple to derive that $spawn_n$ never changes activity flags $Actf_k$, where k is greater than n, which we denote as $\lceil Actf_{\le n} \rceil$.

The two termination conditions (33) and (34) for $proc_m$, are lifted to one progress condition for seq_m in the next lemma.

Lemma 1 (Progress seq_m)
If the termination conditions (33) and (34) hold, then each seq_m satisfies the following progress condition:

$$seq_m(S, Actf_m) \vdash \Box\ R_m^{act} \to \Box\ (\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \to \Diamond\ P_m)\,.$$

Proof The proof uses induction on the always formula from the right-hand side of the implication in the succedent according to rule (17). This reduces the verification of the infinite loop in seq_m to the verification of its body. From (33) and (34), it follows that $proc_m$ satisfies the following termination property and (due to the implicit frame assumption) never changes the activity function:

$$proc_m \vdash \Box\ R_m \rightarrow \Box\ ((\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last}\,) \wedge Actf = Actf')\,. \quad (42)$$

Property (42) is used to abstract the procedure call $proc_m$ in seq_m. Observe that we must also use here the compositionality of the chop and star operator (for the loop). Its precondition $\Box\ R_m$ follows directly from the premise $\Box\ R_m^{act}$. After symbolic execution of the first assignment in seq_m, the activity flag $Actf_m$ is true since it is local (cf. R_m^{act}). Symbolic execution of (42) is guaranteed to terminate whenever the environment never interferes or process m always eventually interferes. Until termination, the activity flag $Actf_m$ remains true. Executing the last assignment sets $Actf_m$ to false, which establishes P_m. □

The next lemma merges the progress property of seq_m from Lemma 1 with RG property (24), the safety property $\lceil Actf_{=m} \rceil$ and the nonblocking behavior $\Box \neg\ \mathbf{blocked}$.

Lemma 2 (seq_m Progress and RG)
If the preconditions of Theorem 6, termination conditions (33) and (34) hold, then each seq_m satisfies the following properties:

$$\begin{aligned} seq_m \vdash \quad & (\Box\ R_m^{act} \rightarrow \Box\ (\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ P_m)) \\ & \wedge\ (R_m \xrightarrow{+} G_m) \wedge \lceil Actf_{=m} \rceil \wedge \Box \neg\ \mathbf{blocked}\,. \end{aligned}$$

Proof This follows from Lemma 1 and formulas (24), resp. (29). □

With these prerequisites, the decomposition theorem for lock-freedom can be proved by induction over the number of interleaved processes. The main idea is to apply the compositionality rule (1) using suitable abstractions for each component in the inductive step. For this purpose, (32) must be strengthened similarly to Lemma 2:

$$\begin{aligned} spawn_n \vdash \quad & (\Box\ Rto_n^{act} \rightarrow \Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ Pto_n)) \\ & \wedge\ (Rto_n \xrightarrow{+} Gto_n) \wedge \lceil Actf_{\leq n} \rceil \wedge \Box \neg\ \mathbf{blocked}\,. \end{aligned} \quad (43)$$

It is simple to see that the correctness of (43) implies Theorem 7.

Proof (of (43)) By structural induction over n. The base case for $n = 0$ follows from Lemma 2. In the inductive case $n = m + 1$, the concurrent system $spawn_{m+1}$ is $seq_{m+1} \parallel_{\mathrm{nf}} spawn_m$ and the induction hypothesis permits to assume (43) for $spawn_m$ on arbitrary paths. Application of the compositionality rule (1) together with Lemma 2 as abstraction for the first component seq_{m+1}, and the induction

hypothesis for the second component $spawn_m$ leads to the following proof obligation:

$$\begin{aligned}
&[\quad (\Box\ R^{act}_{m+1} \rightarrow \Box\ (\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ P_{m+1}))\\
&\quad \wedge (R_{m+1} \xrightarrow{+} G_{m+1}) \wedge \lceil Actf_{=m+1} \rceil \wedge \Box \neg\ \mathbf{blocked}\ ]\\
\|_{\mathrm{nf}}\ &[\quad (\Box\ Rto^{act}_{m} \rightarrow \Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ Pto_m))\\
&\quad \wedge (Rto_m \xrightarrow{+} Gto_m) \wedge \lceil Actf_{\leq m} \rceil \wedge \Box \neg\ \mathbf{blocked}\ ],\\
\Box\ Rto^{act}_{m+1}\ &\vdash\ \Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ Pto_{m+1})\ .
\end{aligned}$$

A local case distinction for both interleaved components depending on whether $\Box\ R^{act}_{m+1}$, resp. $\Box\ Rto^{act}_{m}$, holds gives four proof obligations ($\vee$ distributes over interleaving). These correspond to the remaining lemmas of this subsection. Lemma 3 covers the main case, where $\Box\ R^{act}_{m+1}$ and $\Box\ Rto^{act}_{m}$ holds in component 1 and 2, respectively. Lemma 4 deals with the case that $\Box\ R^{act}_{m+1}$ holds in component 1 but Rto^{act}_{m} is violated eventually in component 2; the lemma for the symmetric case is similar and omitted here. Finally, Lemma 5 handles the case where the rely conditions of both components are violated. □

The next lemma contains the core argument why the two interleaved components seq_{m+1} (component 1) and $spawn_m$ (component 2) ensure global progress. In the lemma, both components are already abstracted with their corresponding progress condition from Lemma 1 and (43), respectively. The intuitive correctness argument is as follows: If seq_{m+1} would always eventually interfere through $\Box \Diamond \neg\ U(S, S')$ with $spawn_m$, then its progress condition would imply that it can eventually progress $\Diamond\ P_{m+1}$. The contrary case, in which seq_{m+1} eventually never interferes, i.e., $\Diamond \Box\ U(S, S')$, implies that one of the other processes (in $spawn_m$) can eventually progress with $\Diamond\ Pto_m$. If one component is preempted and never resumes its execution, then the remaining component progresses, and the progress of either of the two components also ensures global progress Pto_{m+1}.

Lemma 3 (Progress in Components 1 and 2)

$$\begin{aligned}
&[\ \Box\ (\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ P_{m+1}) \wedge \Box \neg\ \mathbf{blocked}\ ]\\
\|_{nf}\ &[\ \Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ Pto_m) \wedge \Box \neg\ \mathbf{blocked}\ ]\\
\vdash\ &\Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ Pto_{m+1})\ .
\end{aligned}$$

Proof Application of induction rule (17) on the always formula in the succedent gives a suitable counter for induction and a proof by contradiction: It assumes that property $\Box\ \ U(S', S'')\ \rightarrow\ \Diamond\ Pto_{m+1}$ is globally violated after a finite number of steps. If a symbolic execution step does not reach this global violation point, induction can be applied to discard the goal. Otherwise, proving that the system does eventually progress $\Diamond\ Pto_{m+1}$ in an environment that never interferes (i.e., $\Box\ U(S', S'')$) gives the required contradiction.

As already explained, there are two possible cases for the local behavior of component 1. The first case, in which component 1 always eventually interferes by means of $\Box \Diamond \neg\ U(S, S')$, follows from the next sublemma:

$$\begin{aligned}
&[\ \Diamond\ P_{m+1} \wedge \Box \neg\ \mathbf{blocked}\ ] \qquad (44)\\
\|_{\mathrm{nf}}\ &[\ \Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ Pto_m) \wedge \Box \neg\ \mathbf{blocked}\ ],\ \Box\ U(S', S'')\ \vdash\ \Diamond\ Pto_{m+1}\ .
\end{aligned}$$

The opposite case, where component 1 eventually no longer interferes, i.e., where $\Diamond\,\Box\ U(S,S')$ holds, follows from the following sublemma, respectively:

$$\begin{array}{c}[\ \Diamond\,\Box\ U(S,S') \wedge \Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ P_{m+1}) \wedge \Box\,\neg\ \textbf{blocked}\ ] \\ \|_{\mathrm{nf}}\ [\ \Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ Pto_m) \wedge \Box\,\neg\ \textbf{blocked}\ ],\ \Box\ U(S',S'')\ \vdash\ \Diamond\ Pto_{m+1}\ .\end{array} \tag{45}$$

Intuitively, (44) is correct since either component 1 is scheduled sufficiently often and reaches the point where it progresses, or if it is preempted and never scheduled again before it reaches this point, then the remaining component 2 runs in an environment that never interferes (i.e., $\Box\ U(S',S'')$) and thus eventually progresses. Formally, liveness rules (28) and (27) are used to get suitable induction hypotheses. Rule (27) discerns whether component 1 is preempted and not resumed again. The positive case is simple since liveness property $\Diamond\ Pto_m$ of component 2 implies $\Diamond\ Pto_{m+1}$. In the negative case, a symbolic execution step of the interleaved formulas discerns which component is scheduled. If component 1 is scheduled and progresses now, then this step discards the goal since progress of component 1 also ensures global progress. All other cases are discarded by applying induction.

The proof of (45) is similar. The main difference is that once component 1 reaches the state from which its steps no longer interfere through $\Box\ U(S,S')$, one must establish that the local environment of component 2 never violates U. The positive case, in which $\Box\ U(S',S'')$ holds for component 2, follows from the following sublemma:

$$\begin{array}{c}[\ \Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ P_{m+1}) \wedge \Box\,\neg\ \textbf{blocked}\ ] \\ \|_{\mathrm{nf}}\ [\ \Diamond\ Pto_m \wedge \Box\,\neg\ \textbf{blocked}\ ],\ \Box\ U(S',S'')\ \vdash\ \Diamond\ Pto_{m+1}\ .\end{array}$$

Its proof is symmetric to the proof of (44) and therefore omitted here. The case in which the environment of component 2 eventually interferes through $\Diamond\,\neg\ U(S',S'')$ is discarded by the last sublemma:

$$\begin{array}{c}[\ \Box\ U(S,S') \wedge \Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ P_{m+1}) \wedge \Box\,\neg\ \textbf{blocked}\ ] \\ \|_{\mathrm{nf}}\ [\ \Diamond\,\neg\ U(S',S'') \wedge \Box\,\neg\ \textbf{blocked}\ ],\ \Box\ U(S',S'')\ \vdash\ \Diamond\ Pto_{m+1}\ .\end{array}$$

The proof uses similar arguments to those of case 2 in the RG proof of (39) and is thus not given in detail. The main difference is that induction on a safety formula is not applicable. Instead, the symmetric versions of liveness rules (27) and (28) are used for induction. □

The next lemma considers the case in which the rely conditions $\Box\ R^{act}_{m+1}$ of component 1 hold; thus it ensures progress and RG locally, according to Lemma 2, while the rely conditions of component 2 are eventually violated locally as $\Diamond\,\neg\ Rto^{act}_m$. The intuitive correctness argument for the lemma is as follows: If component 2 is scheduled sufficiently often, then the violation of Rto^{act}_m occurs globally and leads to a contradiction since the environment of component 2 never violates it. Otherwise, the remaining component 1 ensures global progress.

Lemma 4 (Progress and RG in Component 1)
Assuming the preconditions of Theorem 6, the following holds:

$$\begin{aligned}
&[\quad \Box\,(\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ P_{m+1}) \\
&\quad \wedge\ (R_{m+1} \xrightarrow{+} G_{m+1}) \wedge \lceil Actf_{=m+1} \rceil \wedge \Box \neg \textbf{ blocked }] \\
&\|_{nf}\ [\ \Diamond \neg\ Rto_m^{act} \wedge (Rto_m^{act} \xrightarrow{+} Gto_m) \wedge \Box \neg \textbf{ blocked }], \\
&\Box\ Rto_{m+1}^{act}\ \vdash\ \Box\,(\Box\ U(S', S'') \rightarrow \Diamond\ Pto_{m+1})\ .
\end{aligned}$$

Proof The symmetric versions of liveness rules (28) and (27) are used to get suitable induction hypotheses. The application of the latter liveness rule yields two cases. If component 2 is never scheduled again, then the remaining component 1 ensures global progress whenever $\Box\ U(S', S'')$ holds since P_{m+1} implies Pto_{m+1}. Otherwise, a symbolic execution step of the interleaved components yields four types of proof obligations, depending on which component is scheduled and whether R_{m+1} respectively Rto_m^{act} holds or not. If a component is scheduled and its environment transition does not violate R_{m+1}, resp. Rto_m^{act}, then the case follows with induction. If a component is scheduled and its environment transition violates R_{m+1}, resp. Rto_m^{act}, then further symbolic execution is necessary to discard the goal, similarly to case 2 in the proof of (39). □

The final lemma considers the case where the rely conditions of both interleaved components are eventually violated locally. Such a violation, however, never occurs globally because each component (and the global environment) sustains the other component's rely. Since no safety formula exists for induction, liveness rules (27) and (28) for unfair interleaving are used instead to derive a contradiction.

Lemma 5 (Rely Eventually Violated in Components 1 and 2)
Given the preconditions of Theorem 6, the following holds:

$$\begin{aligned}
&[\ \Diamond \neg\ R_{m+1}^{act} \wedge (R_{m+1}^{act} \xrightarrow{+} G_{m+1}) \wedge \lceil Actf_{=m+1} \rceil \wedge \Box \neg \textbf{ blocked }] \\
&\|_{nf} [\ \Diamond \neg\ Rto_m^{act} \wedge (Rto_m^{act} \xrightarrow{+} Gto_m) \wedge \lceil Actf_{\leq m} \rceil \wedge \Box \neg \textbf{ blocked }], \\
&\Box\ Rto_{m+1}^{act} \vdash\ .
\end{aligned}$$

Proof Liveness rule (28) and its symmetric version are used to get two counters that count the steps until $\neg\ R_{m+1}^{act}$ and $\neg\ Rto_m^{act}$ hold, respectively. The induction term is the sum of both introduced counters. Symbolically executing the interleaved formulas gives four types of proof obligations, similarly to the proof of Lemma 4, which are discarded by applying induction or further symbolic execution. □