Interleaved Programs and Rely-Guarantee Reasoning with ITL

Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst and Wolfgang Reif
Institute for Software and Systems Engineering
University of Augsburg
Augsburg, Germany
{*schellhorn,tofan,ernst,reif*}*@informatik.uni-augsburg.de*

***Abstract*—This paper presents a logic that extends basic ITL with explicit, interleaved programs. The calculus is based on symbolic execution, as previously described. We extend this former work here, by integrating the logic with higher-order logic, adding recursive procedures and rules to reason about fairness. Further, we show how rules for rely-guarantee reasoning can be derived and outline the application of some features to verify concurrent programs in practice. The logic is implemented in the interactive verification environment KIV.**

***Keywords*-Interval Temporal Logic; Compositional Reasoning; Concurrency; Rely-Guarantee Reasoning**

I. Introduction

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

Numerous specialized automatic methods have been developed to verify decidable system classes, e.g., model checking, decision procedures and abstract interpretation techniques. While these are often successful on correct programs, they have two significant disadvantages: first, they typically do not provide insight, 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.

The alternative to specific automated proof techniques is interactive theorem proving. A main 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. Most interactive provers are based on variants of higher-order logic (HOL). However, embedding concurrency into HOL requires a big effort to encode the semantics of programs and of (temporal) assertions. Therefore, the expressive temporal logic and the native programming language described here has been directly implemented in the theorem prover KIV [1] 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 not become more complicated using the extended logic than using the wp-calculus already implemented in KIV.
- Arbitrary correctness and progress properties of programs should be expressible.
- Compositional proofs for parallel programs, in particular rely-guarantee reasoning.

The basic approach to extend ITL [2] by shared-variable interleaved programs has already been described in [3]. It focuses on porting the well-known principle of symbolic execution [4] of sequential programs to parallel programs. For lack of space, we do not repeat the rules used to implement this principle here.

Instead, this paper focuses on several extensions of the logic. Section II describes the embedding of the basic logic into higher-order logic, instead of first-order logic. Global frame assumptions are replaced with local ones. Section III describes compositional interleaving. Section IV adds recursive procedures. Section V outlines how forms of rely-guarantee reasoning can be derived, by abstracting each program with a rely-guarantee formula. Section VI describes well-founded induction, and shows how weak fairness of interleaving is encoded. Section VII outlines a few applications, highlighting the use of some features of the logic. Finally, Section VIII concludes.

II. The Basic Logic

Our definition of ITL is similar to [5], but instead of first-order logic we use higher-order logic, i.e., simply typed lambda calculus as the base logic and we extend the semantics to interleaved programs.

A. Signatures and Algebras

A higher-order signature $SIG = (S, OP)$ consists of two finite sets. A set S of sorts, with $bool \in S$, which 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 $\underline{t} = t_1, \ldots, t_n$. The set OP contains typed operators $op : t$,

including the usual boolean operators, e.g., $\mathrm{true}, \mathrm{false} : \mathit{bool}$ and $. \vee . : \mathit{bool} \times \mathit{bool} \to \mathit{bool}$.

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

B. Expressions and Temporal Formulas

Temporal logic expressions are defined over a signature SIG, dynamic (flexible) variables $x, y, z \in X$ and static variables $u \in U$. 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 $v \in X \cup U$. As usual in higher-order logic, expressions e of type bool are formulas, denoted by φ.

$$e ::= u \mid x \mid x' \mid x'' \mid op \mid e(\underline{e}) \mid e_1 = e_2 \mid \lambda \underline{u}.\ e \mid \forall\ \underline{v}.\ \varphi \mid$$
$$\varphi_1\ \textbf{until}\ \varphi_2 \mid \varphi_1; \varphi_2 \mid \varphi^* \mid \mathbf{A}\,\varphi \mid \textbf{step} \mid \varphi_1 \| \varphi_2 \mid \varphi_1 \|_{\mathrm{nf}} \varphi_2$$

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. The first line defines higher-order expressions that do not involve temporal logic. Dynamic variables can be primed and double primed. Lambda expressions allow for static variables only, while quantifiers allow both static and dynamic variables. The chop operator $\varphi_1; \varphi_2$ is used as sequential composition of programs. The star operator φ^* is similar to a loop. Universal path quantification is denoted as $\mathbf{A}\,\varphi$, and **step** characterizes atomic steps of a program. $\varphi_1 \| \varphi_2$ and $\varphi_1 \|_{\mathrm{nf}} \varphi_2$ denote weak-fair and arbitrary (non-fair) interleaving of φ_1 and φ_2. By convention, temporal operators bind stronger than junctors, and quantifiers bind as far to the right as possible. Free variables $\mathrm{free}(e)$ are defined as usual.

C. Semantics

Standard semantics of ITL defines an interval $I = (I(0), I(1), \ldots)$ to be a finite or infinite sequence of states, where a state maps variables to values. Static variables are disallowed to change between states. To have a compositional semantics for interleaving (as explained in Section III), our semantics alternates between system and environment transitions by adding intermediate primed states: $I = (I(0), I'(0), I(1), I'(1), \ldots)$. The transitions from $I(0)$ to $I'(0)$, $I(1)$ to $I'(1)$ etc. are system steps, while the steps from $I'(0)$ to $I(1)$, $I'(1)$ to $I(2)$ etc. are environment steps. The idea is similar to reactive sequences in [6].

Finite intervals with length $\#I = n$ have $2n + 1$ states and end in the unprimed state $I(n)$. Infinite intervals have $\#I = \infty$. For an interval I and $m \le n \le \#I$, $I_{[m..n]}$ denotes the subinterval from $I(m)$ to $I(n)$ inclusive. $I_{[n..]}$ is the postfix starting with $I(n)$.

The semantics $[\![e]\!](I)$ of an expression e of type t w.r.t. an interval I (and an algebra $\mathcal{A}$, which we leave implicit) is an element of A_t. In particular, a formula φ evaluates to **ff** or **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 are evaluated over I(0) by convention. Operators get 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 $\#\sigma = \#I$. Similarly, $I[\underline{u} \leftarrow \underline{a}]$ modifies static variables $\underline{u}$ to values $\underline{a}$. The semantics of a tuple of expressions $\underline{e}$ is the tuple of semantic values for each e_k. With these prerequisites, the semantics of the rest of the expressions, except interleaving, is defined as follows:

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

The semantics of higher-order formulas $\varphi(\underline{x})$ without primed variables or temporal operators depends on $I(0)$ only. These formulas are called *state formulas* in the following. Higher-order formulas $\varphi(\underline{x}, \underline{x'})$ describe properties of the first system step, while formulas $\varphi(\underline{x'}, \underline{x''})$ describe the first environment step respectively.

The semantics of the chop operator "; " agrees with the semantics of a compound. Either the first part (φ_1) does

not terminate and the full interval is a run of φ_1, or the interval can be split into two parts: a first, finite part where φ_1 runs and a second, possibly infinite, part where φ_2 runs. Similarly, the star operator corresponds to a loop which runs φ for a nondeterministic, maybe infinite number of times. The iteration splits the interval into finitely or infinitely many parts $I_{[0..n_1]}, I_{[n_1..n_2]}, \ldots$, each of which is required to satisfy φ (the last part is infinite, if the split is finite, but the interval infinite). For an empty interval φ^* trivially holds using zero iterations.

In the following, we use the following operators defined as abbreviations:

$$\exists\ \underline{v}.\ \varphi \equiv \neg \forall\ \underline{v}.\ \neg \varphi \qquad \mathbf{E}\,\varphi \equiv \neg\, \mathbf{A} \neg \varphi$$
$$\Diamond\ \varphi \equiv \mathrm{true}\ \textbf{until}\ \varphi \qquad \Box\ \varphi \equiv \neg \Diamond \neg \varphi$$
$$\circ\ \varphi \equiv \textbf{step}; \varphi \qquad \bullet\ \varphi \equiv \neg \circ \neg \varphi$$
$$\textbf{last} \equiv \neg\ (\textbf{step}; \mathrm{true}) \qquad \textbf{inf} \equiv \Box \neg\ \textbf{last}$$

The empty interval consisting of just $I(0)$ is characterized by the formula **last**, infinite intervals by **inf**.

D. Programs

Programs are introduced as specific formulas, which influence system steps only. A program is valid over an interval I if I is a possible run of the program. Finite intervals correspond to terminating programs.

Deviating from [3], assignments $x := e$ are not required to leave *all* variables except x unchanged (expressed there as a formula $\lceil x \rceil$, the global frame assumption). This requirement turned out not to be practical, as then all variables are free in assignments, which prevents elimination of quantifiers by new variables, as used in the rules of sequent calculus.

Therefore, like in TLA [7], we now use an explicit vector of disjoint, flexible variables $\underline{x}$ as a local frame assumption around a program α. Assignments in $[\alpha]_{\underline{x}}$ leave all variables unchanged that do not occur on the left hand side, but are in $\underline{x}$. For (parallel) assignments we therefore have:

$$[\underline{z} := \underline{e}]_{\underline{x}} \ \equiv\ \underline{z'} = \underline{e} \wedge \textbf{step} \wedge \underline{y} = \underline{y'} \quad \text{, where } \underline{y} = \underline{x} \setminus \underline{z}$$

Any formula φ may be used as a program. Frame assumptions propagate over chop and star to assignments

$$[\varphi_1; \varphi_2]_{\underline{x}} \ \equiv\ [\varphi_1]_{\underline{x}}; [\varphi_2]_{\underline{x}} \quad \text{and} \quad [\varphi^*]_{\underline{x}} \equiv\ ([\varphi]_{\underline{x}})^*$$

and similarly over interleaving. Frame assumptions around other types of formulas are simply dropped. All the usual constructs for sequential programs can now be defined:

$$\begin{aligned}
[\textbf{skip}]_{\underline{x}} &\equiv \textbf{step} \wedge \underline{x'} = \underline{x} \\
[\textbf{if}^* \varphi\ \textbf{then}\ \alpha_1\ \textbf{else}\ \alpha_2]_{\underline{x}} &\equiv \varphi \wedge [\alpha_1]_{\underline{x}} \vee \neg\, \varphi \wedge [\alpha_2]_{\underline{x}} \\
[\textbf{if}\ \varphi\ \textbf{then}\ \alpha_1\ \textbf{else}\ \alpha_2]_{\underline{x}} &\equiv [\textbf{if}^* \varphi\ \textbf{then}\ (\textbf{skip}; \alpha_1) \\
&\qquad\qquad \textbf{else}\ (\textbf{skip}; \alpha_2)]_{\underline{x}} \\
[\textbf{while}^*\ \varphi\ \textbf{do}\ \alpha]_{\underline{x}} &\equiv (\varphi \wedge [\alpha]_{\underline{x}})^*; (\neg\, \varphi \wedge \textbf{last}) \\
[\textbf{while}\ \varphi\ \textbf{do}\ \alpha]_{\underline{x}} &\equiv [\textbf{while}^*\ \varphi\ \textbf{do}\ (\textbf{skip}; \alpha)]_{\underline{x}}
\end{aligned}$$

$$\begin{aligned}
[\textbf{let}\ \underline{z} = \underline{e}\ \textbf{in}\ \alpha]_{\underline{x}} &\equiv \exists\ \underline{y}.\, \underline{y} = \underline{e} \wedge [\alpha^{\underline{y}}_{\underline{z}}]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y''} = \underline{y'} \\
[\textbf{choose}\ \underline{z}\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha_1\ \textbf{ifnone}\ \alpha_2]_{\underline{x}} &\equiv \begin{array}{l}(\exists\ \underline{y}.\, \varphi^{\underline{y}}_{\underline{z}} \wedge [\alpha_1{}^{\underline{y}}_{\underline{z}}]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y''} = \underline{y'}) \\ \vee\ (\neg\, \exists\ \underline{z}.\, \varphi) \wedge [\alpha_2]_{\underline{x}}\end{array}
\end{aligned}$$

skip is a stutter step, which leaves all variables in $\underline{x}$ unchanged. A normal **if** evaluates the test in an extra step (indicated by leading **skip**s in the then and else part). **if*** is used to model instructions such as compare-and-set (CAS), which execute a test and an assignment atomically.

The definition of **let** introduces new flexible variables $\underline{y}$ as local variables for $\underline{z}$. 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 local variables $\underline{y}$ are not modified by environment steps. The **choose** is a nondeterministic let (taken from ASMs [8]). It chooses some values that satisfy φ, binds them to $\underline{y}$ and executes α_1 with $\underline{z}$ renamed to $\underline{y}$. If there is no possible choice of values, e.g., if φ is false, then α_2 is executed. Note that the semantics of programs is well-defined without any restrictions on the expressions and formulas used in programs. In practice, however, tests and assignments are state expressions (such programs are guaranteed to have a nonempty set of runs from any initial state, while others could be equivalent to false).

III. Compositional Interleaving

One of the crucial design criteria for our logic was that interleaving (and operators such as chop and star) must be *compositional*, i.e., the following rule of sequent calculus is sound[1].

$$\frac{[\alpha_1]_{\underline{x}} \vdash \varphi_1 \qquad [\alpha_2]_{\underline{x}} \vdash \varphi_2 \qquad (\varphi_1 \| \varphi_2) \vdash \psi}{[\alpha_1 \parallel \alpha_2]_{\underline{x}} \vdash \psi} \tag{1}$$

Proving that an interleaved program $[\alpha_1 \parallel \alpha_2]_{\underline{x}}$ satisfies a property ψ then can be done by proving that the two individual programs satisfy properties φ_1 and φ_2 and then abstracting the programs to their properties to prove ψ. Section V shows how this feature can be exploited, by setting φ_1 and φ_2 to suitable rely-guarantee properties. Compositionality holds, when the semantics of interleaving is definable by interleaving individual intervals:

$$\begin{aligned}
I \models \varphi_1 \| \varphi_2 \quad &\text{iff there are } I_1, I_2 : \\
&I_1 \models \varphi_1 \text{ and } I_2 \models \varphi_2 \text{ and } I \in I_1 \parallel I_2
\end{aligned}$$

This definition of interleaving is possible only for a semantics of programs that takes its environment into account. Therefore, we have chosen a semantics with alternating system and environment steps.

In our setting, parallel programs communicate via shared variables. For synchronization we use an operator **await** φ

[1] A sequent $\varphi_1, \varphi_2, \ldots \vdash \psi_1, \psi_2, \ldots$ abbreviates the formula $(\varphi_1 \wedge \varphi_2 \wedge \ldots) \rightarrow (\psi_1 \vee \psi_2 \vee \ldots)$. A rule is sound, if valid premises above the line imply a valid conclusion. Rules are applied bottom-up reducing goals to simpler goals.

that *blocks* the executing process as long as condition φ is not satisfied. A blocked process repeatedly executes stutter steps that additionally fulfill the formula **blocked**. It is defined in terms of a special boolean variable *Blk*, which is implicitly contained in all frame assumptions and therefore not changed by assignments and **skip**. In contrast, a blocked step is *specified* to toggle *Blk*.

$$\begin{aligned} \textbf{blocked} &\equiv \mathit{Blk}' \neq \mathit{Blk} \\ [\textbf{await } \varphi]_{\underline{x}} &\equiv [\textbf{while}^* \neg \varphi \textbf{ do } \mathit{Blk} := \neg \mathit{Blk}]_{\underline{x}} \end{aligned}$$

We now define weak fair $I_1 \parallel I_2$ and non-fair $I_1 \parallel_{\text{nf}} I_2$ interleaving of two intervals. In comparison to [3] (that is based on SOS rules) we prefer a slightly different approach here that fits better to the axioms from Section VI.

To characterize fairness, we introduce explicitly *scheduled interleavings* $I_1 \oplus I_2$. They are sets of pairs (I, s) of the resulting intervals I and schedules $s = (s(0), s(1), \ldots)$. Each $s(i)$ is either 1 or 2, indicating which interval was scheduled for execution. We denote the postfix of s starting with $s(n)$ as $s_{[n..]}$. A schedule is fair if it is either finite, or infinitely often changes the selected process. Thus, we have:

$$\begin{aligned} I_1 \parallel I_2 &\equiv \{I : \text{there is a fair } s \text{ with } (I, s) \in I_1 \oplus I_2\} \\ I_1 \parallel_{\text{nf}} I_2 &\equiv \{I : \text{there is } s \text{ with } (I, s) \in I_1 \oplus I_2\} \end{aligned}$$

The set $I_1 \oplus I_2$ is defined recursively as the union of 6 cases. We describe the first three, where I_1 is scheduled, the other three cases are symmetric.

1) The first process 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 is returned. Otherwise interleaving is not possible and the empty set is returned.
2) The first step of process 1 is not blocked ($I_1 \models \neg$ **blocked**). Then its first system transition is executed, and the system continues with interleaving the remaining process with the second. The set of all pairs (I, s) is returned, where $I(0) = I_1(0)$, $I'(0) = I_1'(0)$, $s(0) = 1$ and $(I_{[1..]}, s_{[1..]}) \in I_{1\,[1..]} \oplus I_2$.
3) The first process is blocked in the current state. If I_2 has terminated, then the result is as in the first case, but with I_1 and I_2 exchanged. Otherwise, $I_1(0) = I_2(0)$ must hold, to have any results, and a transition of the second process is taken, even though the first is scheduled. The resulting pairs (I, s) have $s(0) = 1$, $I(0) = I_2(0)$, $I'(0) = I_2'(0)$, and $(I_{[1..]}, s_{[1..]})$ must be in $I_{1\,[1..]} \oplus I_{2\,[1..]}$. Both transitions are consumed and the overall transition is blocked iff the first transition of I_2 is blocked too.

Note that the schedule ends as soon as one interval is finished, it may be shorter than the resulting interleaved interval. As an example for interleaving consider two one-step intervals $I_1 = (I_1(0), I_1'(0), I_1(1))$ and $I_2 = (I_2(0), I_2'(0), I_2(1))$ with unblocked steps and a schedule $s = (1, 2)$. The interleaved result then is $I = (I_1(0), I_1'(0), I_2(0), I_2'(0), I_1(1))$, 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, corresponding to the intuition that the environment steps of one process consist of alternating sequences of global environment steps and steps of the other process. Environment assumptions for one process (cf. rely conditions in Section V) must therefore be satisfied by such sequences.

IV. Procedures

To be practically useful, a programming language should have recursive procedures. Many semantic encodings of programming languages either do not consider procedures at all, or they study procedures without parameters only. A common way of introducing procedures when defining the semantics of programming languages is to have local procedure declarations and environments which store them. Our logic instead prefers *globally defined* procedures. This has the advantage that it becomes possible to *specify* procedures using axioms in addition to procedure implementations. We will exploit the possibility to specify procedures in Section VII. We do not use procedures that compute on global variables, since these do not allow to determine the modified variables. Instead, all variables a procedure computes on must be given explicitly as parameters.

Technically, the signature $SIG = (S, OP, Proc)$ is extended with typed procedure names $p : \underline{t_1}; \underline{t_2} \in Proc$. The first vector of types indicates the types of input (or call by value) parameters, the second vector indicates the types of input/output (or call by reference) parameters. The set of formulas (boolean expressions) is extended to contain procedure calls $p(\underline{e}; \underline{x})$, where $\underline{e}$ are expressions of types $\underline{t_1}$ and $\underline{x}$ are pairwise disjoint variables of types $\underline{t_2}$.

The semantics $p^{\mathcal{A}}$ of a procedure $p : \underline{t_1}; \underline{t_2}$ 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 $\underline{t_1}$. σ is a value sequence that exhibits, how the reference parameters change in each step. Note that this semantics implies that input parameters work like local variables. Changes to these parameters while the procedure is executing are not globally visible. The semantics of a procedure call is

$$I \models p(\underline{e}; \underline{x}) \text{ iff } (\llbracket \underline{e} \rrbracket(I), I(\underline{x})) \in p^{\mathcal{A}}$$

A procedure call within a frame assumption abbreviates

$$[p(\underline{e}; \underline{z})]_{\underline{x}} \equiv p(\underline{e}; \underline{z}) \wedge \Box\ \underline{y} = \underline{y'} \quad \text{, where } \underline{y} = \underline{x} \setminus \underline{z}$$

Specifications may now contain axioms for procedures. A typical contract for a procedure with pre- and postcondition φ, ψ is:

$$\varphi \wedge p(\underline{x}; \underline{y}) \wedge (\Box\ \underline{y''} = \underline{y'}) \rightarrow \Diamond\ (\textbf{last} \wedge \psi)$$

It states that starting the procedure p in a state where φ holds and assuming that the environment never changes the reference parameters $\underline{y}$, the procedure will always reach a final state, where ψ holds. Procedures can also be used as placeholders for arbitrary formulas φ by specifying $p(;\underline{x}) \leftrightarrow \varphi$ using $\underline{x} = \text{free}(\varphi)$ as reference parameters.

The implementation of procedures is specified by (possibly mutually recursive) procedure declarations of the form $p(\underline{x};\underline{y})\ .\ \alpha$. Two requirements for the body α guarantee that the semantics is correct. 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: such a program uses state formulas only in its expressions (tests, parameters of procedures, right hand sides of assignments).

Regular programs α can be proved to be monotonic in their calls: for two procedures p and q with the same argument types if $p^{\mathcal{A}} \subseteq q^{\mathcal{A}}$, then $\{I : I \models \alpha\} \subseteq \{I : I \models \alpha'\}$ where α' replaces all calls to p in α with calls to q. Therefore, the semantics of recursive procedure declarations can be defined according to Knaster-Tarski's standard fixpoint theorem. In particular, a declaration $p(\underline{x};\underline{y}).\alpha$ yields the fixpoint equation

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

For deduction, the unfolding axiom

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

is implied, which directly expands the **let**. New local variables $\underline{z}$, which can not be changed by the environment, are used for the value parameters $\underline{x}$ (initialized with $\underline{e}$). Changes to the reference parameters are globally visible.

V. Rely-Guarantee Proofs

Rely-guarantee rules [9] define suitable abstractions for individual system components to avoid reasoning about their interleaved execution. In our setting, these abstractions typically are

$$p(\underline{y}) \vdash R(\underline{y}', \underline{y}'') \xrightarrow{+} G(\underline{y}, \underline{y}')$$

where a procedure p is abstracted by a temporal formula $R \xrightarrow{+} G$. The state variables $\underline{y}$ and the frame assumption are usually omitted. The sustains operator $\xrightarrow{+}$ ensures that the guarantee conditions G are maintained by p's steps, as long as *previous* environment transitions have preserved its rely conditions R. It is defined as[2]

$$R \xrightarrow{+} G \ \equiv \neg\ (R\, \textbf{until} \neg\ G)$$

[2]In previous work, we used the equivalent, but more complex formula G **unless** $(G \wedge \neg\ R)$. Formula $G^*;(G \wedge \neg\ R)$ is equivalent too.

For instance, the verification of $p_1 \parallel_{\text{nf}} p_2$ can be decomposed with the following rely-guarantee rule, for $i = 1, 2$.

$$\frac{\begin{array}{lll} \vdash \text{reflexive}(G_i) & \vdash \text{transitive}(R_i) & G_i \vdash R_{3-i} \\ p_i \vdash R_i \xrightarrow{+} G_i & R \vdash R_i & G_i \vdash G \end{array}}{p_1 \parallel_{\text{nf}} p_2,\ \Box\ R \vdash \Box\ G} \quad (2)$$

The conclusion of this rule states that each step of an interleaved system execution preserves a guarantee G at all times in an environment R. This is because a component's rely R_i is preserved by both the system's environment and each step of the other component. Hence, the first premise ensures that each system step of p_i preserves its guarantee G_i, thus G, at all times. Note that according to the definition of interleaving in Section III, an environment step of a procedure can consist of both steps of the global environment and steps of the other program. Therefore, guarantee conditions must imply the other rely.

The calculus permits to formally derive rule (2) as follows: first, components p_1 and p_2 are abstracted by the corresponding sustains formula using (1). Then the proof derives a contradiction by induction over the number of steps until G is violated and symbolic execution. Abstraction is used, since an arbitrary component procedure p_i can not be executed. The sustains operator, however, can be executed according to the following unwinding rule.

$$(R \xrightarrow{+} G) \ \leftrightarrow\ G \wedge (R \rightarrow \bullet\ (R \xrightarrow{+} G))$$

A symbolic execution step of $\xrightarrow{+}$ proves that G is maintained by the first program transition and by the rest of the interval if the previous environment transition satisfies R.

We note that rely-guarantee rules for systems with an unbounded number of interleaved components can be derived as well. In practice, these rules typically include further predicates, e.g., for invariants or pre- postconditions. We have also derived local rely-guarantee rules, where specifications consider a small number of representative components only, instead of using an arbitrary number of local states, as in the original approach [9]. Such reductions are useful when verifying concurrent data structures, where processes exhibit similar behaviors. Rely-guarantee reasoning also serves as a base for the decomposition of global correctness and progress properties of concurrent systems (cf. Section VII).

Furthermore, we have encoded the complete rely-guarantee proof system from Xu et al. [10]. Informally, their rely-guarantee specifications p sat $(pre,\ rely,\ guar,\ post)$ ensure that starting from a state that satisfies precondition pre in an environment that always fulfills $rely$, program p satisfies the guarantee $guar$ in each step and establishes postcondition $post$ upon termination. A translation of these specifications in our logic is:

$$p, pre \vdash rely^* \xrightarrow{+} (guar \wedge (\ \textbf{last}\ \rightarrow post))$$

By using the transitive closure $rely^*$, we can summarize consecutive rely (environment) transitions. This is necessary to weaken our requirement of transitive relies, since environment steps abstract from the number of transitions of other processes in our setting. In contrast, executions in [10] record environment transitions at the level of atomic actions and therefore do not have to be transitive.

To prove that no deadlocks occur, Xu uses an additional run-predicate, which characterizes unblocked program states. A similar encoding can be defined in our setting, by introducing an additional predicate run and adding $run \rightarrow \neg$ **blocked** to the guarantee conditions. Moreover, we can express total correctness of programs w.r.t. a rely-guarantee specification simply as

$$p, pre \vdash (rely^* \xrightarrow{+} guar) \wedge \Diamond\ (\mathbf{last} \wedge post)$$

VI. Induction and Weak Fairness

In higher-order logic, proving a formula $\varphi(N)$ by induction over a well-founded order $\prec$ gives an induction hypothesis $\forall\ M.\ M \prec N \rightarrow \varphi(M)$, which must be shown to imply $\varphi(N)$. For temporal reasoning this is not sufficient, as this induction hypothesis would hold only for the *current* interval, while de facto it holds for *all* intervals.

In [3] we have put a $\Box$ in front of the induction hypothesis, which gives an induction hypothesis for all *suffixes* of the current interval. However, when recursive procedures are used, it is sometimes necessary to have the induction hypothesis for an *infix* of the current interval, which is determined by a recursive call. Thus, the following stronger rule is used for induction over a term e:

$$\frac{e = n, \textbf{Ind-Hyp}(n)\ \vdash \varphi}{\vdash \varphi} \tag{3}$$

where $\textbf{Ind-Hyp}(n) \equiv \mathbf{A}\forall\ \underline{v}.(e \prec n \rightarrow \varphi)$, n is a new static variable, and $\underline{v} = \text{free}(\varphi) \cup \text{free}(e)$.

The validity of the induction hypothesis depends on the static variable n only, so it is preserved unchanged when stepping through an interval by symbolic execution[3]. The induction hypothesis is applied like a global lemma $e \prec n \rightarrow \varphi$.

To reason about temporal formulas, in addition to well-founded induction most calculi use additional induction rules to reason about the passing of time (here: the length of intervals). Our calculus prefers to reduce such principles to standard well-founded induction whenever possible. In particular, the following equivalence is used:

$$\Diamond\ \varphi \leftrightarrow \exists\ N.\ N = N'' + 1\ \textbf{until}\ \varphi \tag{4}$$

[3]It would be sufficient to use the weaker $\boxdot$ operator instead of $\mathbf{A}$, where "for all subintervals" is defined as $\boxdot\varphi \equiv\ true; \varphi; true$. However, $\boxdot$ has other uses (see [11]) where it should be symbolically executed, while the induction hypothesis should not.

N is a new flexible variable for natural numbers, that is decremented 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 $\Box\ \varphi$, this equivalence 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. Proving that a program satisfies a rely-guarantee property $R \xrightarrow{+} G$ first introduces a new boolean variable B, and then applies (4) on $\Diamond\ B$.

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

The resulting counter N counts the number of steps for which the guarantee must be upheld, provided the rely is true until then.

Both induction principles are special cases of induction over the length of a prefix of the current interval (called prefix induction). Such an induction is possible for safety formulas φ, that are valid over a full interval I, when every prefix of I can be extended to an interval where φ holds. All higher-order formulas, always-, until- formulas and all regular sequential programs without local variables and procedure calls fall into the class of safety formulas. More details on prefix induction and the semantics of the necessary **prefix** operator is given in [12].

Reasoning about an interleaved program $\alpha_1 \parallel \alpha_2$ by symbolic execution is indifferent to whether the interleaving is weak-fair or nonfair. Either the first step of α_1 is executed, leaving a restprogram $\alpha_1' \parallel \alpha_2$, or the first step of α_2 is executed, leaving $\alpha_1 \parallel \alpha_2'$.

However, symbolic execution alone is not sufficient to deal with weak fairness. We need a way to ensure, that in a fair interleaving, *eventually* each of the programs will execute a step. To make this "eventually" explicit, 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 ("labels"), which enforce scheduling. 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. $\alpha_1 \parallel \alpha_2$ is thus considered as an abbreviation for both labels being false. The definition of Section III is adapted to remove (I, s) from $I_1 \oplus I_2$ if for some $n < \#s\ I_{[n\ldots]} \models L_1$, but $s(n) = 2$, or if $I_{[n\ldots]} \models L_2$ and $s(n) = 1$. No interleaving is possible when both L_1 and L_2 are true in the same state.

Using scheduling labels is inspired by the auxiliary variables used in [13] to encode fairness. However, our calculus does not pre-encode fairness (by immediately transforming the program), but introduces them on the fly by the rule

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

and a symmetric rule for α_2. Typically, L_1 and L_2 are both

false, so the rule simplifies to:

$$\alpha_1 \parallel \alpha_2 \leftrightarrow \exists\ B.\ \Diamond\ B \wedge (B\colon \alpha_1 \parallel \alpha_2) \tag{5}$$

Informally, 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.

A simple example demonstrates the interplay between the given rules.

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

would be proved by applying (5), then (4) on $\Diamond\ B$, which introduces the variable N. Induction (3) over N then yields

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

Executing a step now either makes B true, then the left process is scheduled and $X = 1$ now, or if B remains false, then the resulting sequent is almost identical, but N has been decremented ($n = N+1$) and induction can be applied.

Interestingly, nonfair interleaving satisfies almost the same rule. Either α_1 will be scheduled after some steps, or the run consists of an infinite sequence of unblocked α_2 steps:

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

The requirement $\mathbf{E}\ \exists\ \underline{x}.\ \alpha_1$ where $\underline{\mathrm{x}}$ = free(α_1) ensures that α_1 is satisfied by at least one interval I_1 that can be used to derive $I_2 \in (I_1 \parallel_{\text{nf}} I_2)$. Compared to fair interleaving, this rule introduces only a simple additional case in proofs. It has been used in the verification of lock-freedom, where unfair scheduling of processes must be considered.

VII. Applications

Based on rely-guarantee reasoning, we have derived decomposition theorems for linearizability [14] and lock-freedom [15]. This section outlines them and their application on some case studies from the literature. Further details are available online [16].

Decomposition of Linearizability and Lock-Freedom Linearizable procedures appear to take effect instantly at one step (the linearization point) between invocation and response. We prove linearizability by locating the linearization point of a procedure cp during its execution in a refinement proof, using an abstraction predicate $Abs(cs, as)$, which relates valid concrete states cs to corresponding abstract states as. Refinement between a concrete and abstract procedure cp resp. ap can then be expressed as

$$cp(cs) \vdash \exists\ as.\ ap(as) \wedge \Box\ (Abs(cs, as) \wedge Abs(cs', as'))$$

To prove linearizability, ap is instantiated with skip steps $\mathbf{skip}^*$ that model concrete non-linearization steps, and an atomic step $alin(as)$ for the linearization point. Moreover, rely conditions R that were established by rely-guarantee reasoning may be assumed (cf. [12] for details). When Abs is a partial function $Absf$, the existential quantifier for as can be dropped, resulting in the proof obligation

$$cp(cs), \Box\ (R \wedge Absf(cs) = as \wedge Absf(cs') = as') \tag{6}$$
$$\vdash \mathbf{skip}^*;\ alin(as);\ \mathbf{skip}^*$$

The right hand side becomes a safety formula and prefix induction can be applied. Finding an induction principle that also covers existentially quantified (safety) formulas is an open issue.

Lock-free implementations avoid major problems associated with locks, such as convoying, deadlocks, livelocks or priority inversion. Lock-freedom guarantees termination of some operation in a finite number of steps, even when individual operations are arbitrarily delayed or fail. However, individual operations might starve under interference.

We use an additional, reflexive and transitive relation U to describe interference freedom ("unchanged"). To prove lock-freedom, each system procedure must terminate without U-interference and also after violating predicate U in a step (cf. [17] for details):

$$cp, \Box\ R \vdash \Box\ (\Box\ U(cs', cs'') \vee \neg\ U(cs, cs') \rightarrow \Diamond\ \mathbf{last}\,)$$

The temporal framework permits to derive that this local proof obligation implies lock-freedom of an interleaved system. In contrast, [18] defines a new logic to reason about lock-freedom, outlining their decomposition on paper only.

Case Studies Proof obligation (6) suffices to verify linearizability of algorithms that have an internal linearization point (within the code of the executing process), even when its location depends on subsequent system behavior. This is possible, since future states of an interval can be easily analyzed in temporal logic. One example of such a linearization point can be found in Michael and Scott's lock-free queue algorithm [19]. In case of a dequeue when the queue is empty, the reading of the shared head-of-queue pointer is a linearization point if the read copy equals the shared head-of-queue in a future state. While other verification approaches, e.g., [20], require additional techniques in such cases, one can decide whether to linearize in the current state of an execution, using the temporal next operator.

Our verification of a lock-free stack with hazard pointers applies abstraction on sequential programs. In programming environments without support for garbage collection, hazard pointers [21] enable safe memory reclamation of objects that are removed from a lock-free data structure. Each process is associated with a fixed number of shared pointers (so called hazard pointers), to signal contending processes not to deallocate a location.

Originally, atomic access on hazard pointers was assumed. Our work confirms that non-atomic access to hazard pointers suffices too, even though a process might then read corrupted hazard pointer entries. To generically specify non-atomic read and write operations, we exploit that we can

specify procedures. The non-atomic read operation na_read is specified as follows.

$$
\begin{aligned}
&na\_read(Lv, Sv) \vdash \\
&\quad \circ \Diamond\ \mathbf{last} \wedge \Box\ (\neg\ \mathbf{blocked} \wedge Sv = Sv') \\
&\wedge\ (\quad \Box\ (\circ \neg\ \mathbf{last} \rightarrow Sv' = Sv'') \\
&\quad \rightarrow \Box\ (\circ\ \mathbf{last} \rightarrow Lv' = Sv'))
\end{aligned}
$$

Procedure na_read terminates after at least one step, it does not block and never changes the shared value Sv to be read; if Sv is never changed by the environment, the local copy Lv finally equals Sv. Using such generic specifications, we can abstract from implementation details of non-atomic access to shared variables. The proofs apply abstraction to replace each generic procedure call with its specification, thus enabling symbolic execution.

VIII. Conclusion

This paper contributes some new and improved concepts and their semantic foundation – embedding into higher-order logic, procedures, rules for fairness and induction – to the basic approach based on symbolic execution of programs and formulas we have defined earlier. The calculus has been successfully used to verify a number of case studies. The current focus was on verification of linearizability and lock-freedom of lock-free algorithms, where we managed to verify some significant examples that had no mechanized proof before. Proof complexity has been quite manageable. The main difficulty of concurrency proofs is finding *correct* theorems with *correct* invariants and rely conditions.

There are still some open problems. Finding good proof rules that allow 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. From the theoretical point of view, our calculus contains two complete fragments: Moszkowski's ITL axioms [22] and the rely-guarantee calculus from [10]. Completeness in general, however, is still an open issue.

References

[1] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser, "Structured specifications and interactive proofs with KIV," in *Automated Deduction—A Basis for Appl.*, W. Bibel and P. Schmitt, Eds. Dordrecht: Kluwer, vol. II, pp. 13 – 39, 1998.

[2] B. Moszkowski, "A temporal logic for multilevel reasoning about hardware," *IEEE*, vol. 18, no. 2, pp. 10–19, 1985.

[3] S. Bäumler, M. Balser, F. Nafz, W. Reif, and G. Schellhorn, "Interactive verification of concurrent systems using symbolic execution," *AI Comm.*, vol. 23, no. (2,3), pp. 285–307, 2010.

[4] R. M. Burstall, "Program proving as hand simulation with a little induction," *Information Processing*, pp. 309–312, 1974.

[5] A. Cau, B. Moszkowski, and H. Zedan, *ITL – Interval Temporal Logic*, Software Techn. Research Laboratory, SER-Centre, De Montfort University, The Gateway, Leicester, 2002, `www.cms.dmu.ac.uk/ cau/itlhomepage`.

[6] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001, no. 54.

[7] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994.

[8] E. Börger and R. F. Stärk, *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[9] C. B. Jones, "Specification and design of (parallel) programs," in *Proc. of IFIP'83*. North-Holland, pp. 321–332, 1983.

[10] Q. Xu, W. de Roever, and J. He, "The rely-guarantee method for verifying shared variable concurrent programs," *FACJ*, vol. 9, no. 2, pp. 149–174, 1997.

[11] F. Ortmeier and G. Schellhorn, "Formal fault tree analysis - practical experiences," in *Proceedings of AVoCS 2006*, 2006.

[12] S. Bäumler, G. Schellhorn, B. Tofan, and W. Reif, "Proving linearizability with temporal logic," *Formal Aspects of Computing (FAC)*, 2009, appeared online first, http://www.springerlink.com/content/7507m59834066h04/.

[13] K. Apt and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[14] M. Herlihy and J. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. on Prog. Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[15] H. Massalin and C. Pu, "A lock-free multiprocessor os kernel," Columbia University, Tech. Rep. CUCS-005-91, 1991.

[16] "Presentation of KIV-proofs for concurrent algorithms," 2011, `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`.

[17] B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif, "Temporal logic verification of lock-freedom," in *In Proc. of MPC 2010*, ser. Springer LNCS 6120, 2010, pp. 377–396.

[18] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "Proving that nonblocking algorithms don't block," in *POPL*. ACM, 2009, pp. 16–28.

[19] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th ACM Symp. on Principles of Distributed Computing*, 1996, pp. 267–275.

[20] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm," in *FORTE 2004*, ser. LNCS, vol. 3235, 2004, pp. 97–114.

[21] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[22] B. C. Moszkowski, "An automata-theoretic completeness proof for interval temporal logic," in Proc. of *ICALP*. London, UK: Springer-Verlag, pp. 223–234, 2000.