# Proving linearizability with temporal logic

Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, Wolfgang Reif

Under consideration for publication in Formal Aspects of Computing

# Proving Linearizability with Temporal Logic

Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen
Universität Augsburg
D-86135, Augsburg, Germany
email:{baeumler,schellhorn,tofan,reif}@informatik.uni-augsburg.de

**Abstract.** Linearizability is a global correctness criterion for concurrent systems. One technique to prove linearizability is applying a composition theorem which reduces the proof of a property of the overall system to sufficient rely-guarantee conditions for single processes. In this paper, we describe how the temporal logic framework implemented in the KIV interactive theorem prover can be used to model concurrent systems and to prove such a composition theorem. Finally, we show how this generic theorem can be instantiated to prove linearizability of two classic lock-free implementations: a Treiber-like stack and a slightly improved version of Michael and Scott's queue.

**Keywords:** Lock-Free, Linearizability, Verification, Temporal Logic, Compositional Reasoning, Rely-Guarantee

## 1. Introduction

As multi-core processor architectures have become standard, the study of concurrent algorithms is a current and important research topic. Among these, lock-free algorithms are a class of algorithms for concurrent access to data structures, which typically do not use locks and mutual exclusive access, but instead rely on atomic *compare-and-swap*-operations. Such operations are now implemented by all common processor types, language support is provided e.g. by Java and C#.

Lock-free algorithms are less vulnerable to common problems such as deadlocks or priority inversion. On the other hand these algorithms are more complex, since they do not use the universal principle of mutual exclusion. This makes it harder to assure their correctness by intuitive arguments. Therefore, a number of researchers have started to develop formal verification techniques that guarantee their correctness.

*Correspondence and offprint requests to*: Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, Wolfgang Reif
Lehrstuhl für Softwaretechnik und Programmiersprachen
Universität Augsburg
D-86135, Augsburg, Germany
email:{baeumler,schellhorn,tofan,reif}@informatik.uni-augsburg.de

Correctness of lock-free algorithms is typically based on verifying linearizability as defined by Herlihy & Wing [HW90]. The underlying idea of linearizability is to view the concurrent operations on a data object as though they occur in sequential order, similar to serializability for database transactions. Informally, linearization states that every concurrent execution of a set of operations is equivalent to a sequential execution of the same set. Verification of this property is usually done by showing that each operation has a linearization point [HW90]. A linearization point is an atomic step between the call and return of an operation, where the externally visible effect occurs.

Usually, reasoning over a concurrent system is hard and tedious work as all possible interleavings have to be considered. A common method which avoids reasoning over the entire system is compositional reasoning. It was first formulated in [Dij65] by Dijkstra. The basic idea is to split a system into several subcomponents and to prove an overall system property using adequate properties of the subcomponents only.

A common compositional proof technique is the rely-guarantee paradigm, which was introduced by Jones [Jon83] and by Misra & Chandy [MC81] (under the name assumption-commitment). The basic idea of this paradigm is that each component can make specific assumptions about its environment, in order to guarantee a specific behavior. Usually, rely-guarantee techniques provide a theorem that specifies in a number of proof obligations how assumptions and guarantees have to be connected in order to show a property for the overall system. Ideally, these proof obligations only deal with single subcomponents and properties of these subcomponents, but not the complete system itself. This results in several proofs of feasible size. Several rely-guarantee methods are expressed in temporal logic. Examples of rely-guarantee reasoning can be found e.g. in [dRdBH$^+$01, CGP00, AL95, CC96].

The approach presented in this paper combines rely-guarantee reasoning and refinement to prove linearizability. It breaks down the linearizability condition into rely-guarantee proof obligations for single processes using a composition theorem. This theorem states that for any interleaved run of an arbitrary number of concrete processes, there is an equivalent run of abstract processes, in which abstract operations have a single atomic step, marking the linearization point. Similar reasoning about linearizability can also be found in [GC07, VHHS06].

This paper presents a formal approach which *fully* mechanizes all three steps[1]: the formal specification and verification of a generic rely-guarantee theorem for refinement, the instantiation of this theorem with two lock-free algorithms to prove linearizability, and the verification of the resulting proof obligations. All three steps are carried out within the temporal logic framework [BBRS08, Bal05] of KIV [RSSB98], which supports the intuitive deduction principle of symbolic execution [Bur74]. A simple to read programming language is used both for specification *and* verification purposes. Application of the approach is shown with a simple lock-free stack algorithm [Tre86] and a more sophisticated lock-free queue algorithm [MS96]. With regard to the queue algorithm which has a non-trivial linearization point, the approach benefits from the ability to refine concretes steps by several abstract steps: no further techniques are required for the refinement proof of the dequeue operation.

In the following, we assume that the reader has at least some basic knowledge of the sequent calculus and temporal logic. The paper is subdivided as follows: To get an idea about the subject matter, Section 2 introduces a simple lock-free stack algorithm and gives some intuition for its correctness. In Section 3 we shortly describe the temporal logic framework of KIV and the basics of its rely-guarantee technique. Section 4 presents the generic refinement theorem which is applied in the next two sections on the stack (Section 5) and a lock-free queue algorithm (Section 6). The paper concludes with sections about related work (Section 7) and a short summary (Section 8).

## 2. Lock-Free Stack Algorithm

The basic principle of Treiber's lock-free stack algorithm [Tre86] consists of three phases. In the first phase, the required data is prepared, e.g. storage is allocated or local variables are initialized. In the second phase, a local snapshot of the global data structure is taken. The third phase deals with changing the global data in one atomic step, based upon the local snapshot. This phase typically uses a *compare-and-swap* (CAS) command. If this atomic step fails because the snapshot has become obsolete in the meantime, the main data structure remains unchanged and the algorithm repeats phase two.

[1] Proofs are online at [KIV].

```
1   CPush(V; Top, Hp, UNew, USuccess) {
2       choose RefNew with RefNew ≠ Null ∧ ¬RefNew ∈ Hp in {
3           Hp := Hp ∪ {RefNew}, UNew := RefNew, USuccess := False;
4           Hp[UNew].val := V;
5           let UTop = Null in {
6               while ¬USuccess do {
7                   UTop := Top;
8                   Hp[UNew].nxt := UTop;
9                   CAS(UTop, UNew; Top, USuccess)
10              }
11          }
12      }
13  }
```

**Fig. 1.** Declaration of the push process in KIV

The informal idea of the CAS command is that a local pointer $L_1$ is compared to a global pointer $G$. If both pointers are identical, $G$ is set to another local variable $L_2$ and the CAS command succeeds. If they are different, $G$ is left unchanged and the CAS command fails. The following KIV specification is used for the CAS-operation:

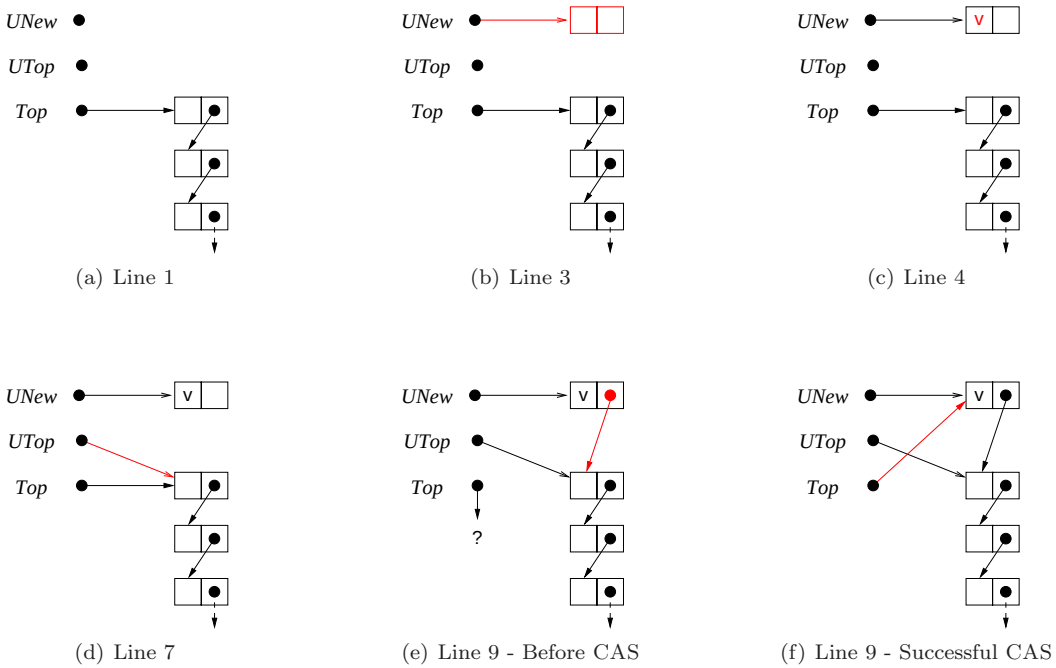
```
CAS(L1, L2; G, Success) {
    if G = L1 then {
        G := L2, Success := True;
    } else {
        Success := False;
    }
}
```

The parameters are separated by a semicolon in *input* parameters, which are read only and *transient* parameters which can be read and modified. The entire **if-then-else** block of CAS takes place atomically. The **then** case describes a successful CAS, which sets $G$ to the value of $L_2$ and the success flag *Success* to true (the comma denotes an atomic assignment). The **else** case leaves $G$ unchanged and sets *Success* to false.

In KIV the stack is represented by a linked list which is stored in a heap *Hp*. Each heap cell has a field for the data value (accessible by the function .val) and a field for a reference (accessible via .nxt), which can also be the Null reference, denoting the end of the stack. The top of the stack is represented by a variable *Top*. The stack is empty if *Top* is Null.

The push algorithm is depicted in Figure 1. The line numbers are given for explanatory purposes only. They are not used in KIV. Parameters of CPush are the value $V$, which should be inserted into the stack and the variables *Top* and *Hp* for the stack representation. Variables *UNew* and *USuccess* are local variables originally. Since it is necessary to observe their intermediate values in correctness assertions, they have been converted to transient parameters. Figure 2 visualizes the configuration of the heap and important variables at several points of the execution. Figure 2(a) depicts the situation when the push operation is called. The algorithm starts by allocating a new cell on the heap and storing the pointer in variable *UNew* (line 2 and 3). This is done by atomically choosing a new unallocated reference (line 2), adding it to the heap and setting the pointer *UNew* to the newly allocated cell (both line 3). The result is depicted in Figure 2(b). Also, the variable *USuccess* is initialized in line 3. The data value is stored in this newly allocated cell (line 4, Figure 2(c)). In line 5, the local variable *UTop* is initialized with a null pointer. After that, the algorithm loops as long as the insertion of the new cell in the stack data structure fails (line 6 to 10). Inside the loop, the pointer of the current top cell is stored in the local variable *UTop* (line 7, Figure 2(d)) and the .nxt-pointer of the previously allocated cell is set to *UTop* (line 8). Finally, the algorithm tries to add the new data value to the stack via CAS (line 9). If the current *Top* is still the same as it was in line 7, the previously allocated cell *UNew* contains the correct .nxt-pointer and *Top* can be set to *UNew*. If the top of stack was changed by another push or pop process in the meantime, the CAS operation fails and the while loop is reiterated. Figure 2(e) depicts the situation just before the CAS operation and Figure 2(f) visualizes the configuration of the stack directly after a successful CAS operation.

The principle for the pop algorithm is the same as for the push algorithm. The specification of the pop




(a) Line 1 (b) Line 3 (c) Line 4

(d) Line 7 (e) Line 9 - Before CAS (f) Line 9 - Successful CAS

**Fig. 2.** Visualization of the Stack During a Push Operation

```
1   CPop(; Top, Hp, OTop, OSuccess, O) {
2     let Lo = Empty, ONxt = Null in {
3       OSuccess := False;
4       while ¬OSuccess do {
5         OTop := Top;
6         if OTop = Null then {
7           Lo := Empty;
8           OSuccess := True;
9         } else {
10          ONxt := Hp[OTop].nxt;
11          Lo := Hp[OTop].val;
12          CAS(OTop, ONxt; Top, OSuccess);
13        }
14      }
15      O := Lo
16    }
17  }
```

**Fig. 3.** Declaration of the pop algorithm in KIV

algorithm is shown in Figure 3. Here, parameter $O$ is used as return value of CPop and the local variable $Lo$ to temporarily store the return value. The pointer $ONxt$ is used to point to the second cell of the stack, which becomes the new top cell if the pop operation succeeds.

Inside the loop, the first operation is to locally store the current top-pointer in $OTop$, to detect changes of the stack afterwards (line 5). If the stack is currently empty, a special value Empty is returned and the process terminates (line 6-8). Otherwise, the data value and the .nxt-pointer of $OTop$ are retrieved (line 10 and 11). If $OTop$ is still accurate in line 12, the CAS-operation correctly changes the top of stack pointer to the second cell, else the loop body is executed again. When the loop is exited, the local value of $Lo$ is assigned to the return parameter $O$ (line 15).

Since both algorithms use pointers to detect whether the global stack has changed by comparing references, the so called ABA-problem should be mentioned. It can occur when a cell is deallocated (by freeing

*OTop* at the end of CPop) and re-allocated using the *same* reference (in CPush), while there is a pointer to that cell (by *OTop* of another process $i$). In this case, $i$ might perform a successful CAS although the stack has changed, which could result in incorrect behavior. There are several techniques to avoid this problem. One is to simply assume garbage collection, as we do in this paper (see the end of Section 4.3 for a formal justification). An alternative is to use modification counters that allow detection of cell changes. These can be introduced in a separate data refinement (see e.g. [GC09]).

## 3. Temporal Logic Framework

In this section we shortly present the temporal logic calculus integrated into the interactive theorem prover KIV. The logic is a variant of interval temporal logic (ITL) [Mos86, CMZ02]. A detailed description can be found in [BBRS08, Bal05].

### 3.1. Interval Temporal Logic

Compared to standard ITL, the formalism used here is extended by explicitly including the behavior of the environment into each step. The basis for ITL are algebras (to interpret the signature) and finite or infinite sequences $I = [I(0), I'(0), I(1), I'(1), \ldots]$ of valuations (mappings of variables to values in the algebra). $|I| \in \mathbb{N} \cup \{\infty\}$ denotes the length of I. Finite intervals of length $n$ end with state $I(n)$. Variables are partitioned into *static* variables (written lower case), which never change their value ($I(0)(v) = I'(0)(v) = I(1)(v) = \ldots$) and *flexible* variables $V$ (starting with an uppercase letter). Valuations in $I$ are called *states*. All expressions are evaluated over an algebra and an interval $I$. The first state $I(0)$ is used to give meaning to predicate logic expressions in the usual way. Intervals alternate *system transitions* and *environment transitions*. The first system transition leads to state $I'(0)$. The next transition to $I(1)$ is an environment transition. To access the values of flexible variables in $I'(0)$ and $I(1)$, *primed variables* $V'$ and *double primed variables* $V''$ are used. By convention $V = V' = V''$ for the last state of the interval. A selection of temporal operators supported by KIV is:

| | |
|---|---|
| $\Box\, \varphi$ | $\varphi$ holds *always* from now on in every state |
| $\Diamond\, \varphi$ | there exists a state where $\varphi$ holds |
| $\bullet\ \varphi$ | $\varphi$ holds in the next state, if there is one |
| $\circ\ \varphi$ | $\varphi$ holds in the next state (which must exist) |
| **last** | the current state is the last |
| $\varphi$ **until** $\psi$ | $\psi$ holds in some state and $\varphi$ holds in every state before |
| $\varphi$ **unless** $\psi$ | $\varphi$ holds as long as no state with $\psi$ is reached |
| $\alpha \parallel \beta$ | interleaving |
| $\mathbf{E}\, \varphi$ | there exists a path, such that $\varphi$ holds |
| $X := t_1, Y := t_2$ | parallel assignment (leaves other variables unchanged) |
| $\alpha$ **or** $\beta$ | nondeterministic choice |
| **choose** $X$ **with** $\varphi$ **in** $\alpha$ | bind local variable X to some value that satisfies $\varphi$ and execute $\alpha$ |
| $\alpha; \beta$ | sequential composition |
| **if** $\psi$ **then** $\alpha_1$ **else** $\alpha_2$ | case distinction |
| **let** $X = t$ **in** $\alpha$ | local variable declaration |
| **while** $\psi$ **do** $\alpha$ | loop |
| **await** $\varphi$ | blocking until $\varphi$ becomes true |
| $\alpha^*$ | iterate $\alpha$ any (finite or infinite) number of times |
| $p(\bar{x}; \bar{y})$ | procedure call (no global variables are allowed in procedure declarations; only variables in $\bar{y}$ are modified) |

Our ITL variant supports classic temporal logic operators as well as program operators. Although we usually write $\alpha, \beta$ to indicate a program and $\varphi, \psi$ to indicate a formula, there is no syntactic distinction between them. Both evaluate to true or false over an interval $I$. In particular, a program evaluates to true ($I \models \alpha$) if $I$ is a possible run of the program. This allows us to mix programs with temporal logic formulas, which is useful for abstraction. Semantically, programs initiate system transitions only, with arbitrary environment

steps interleaved (this idea is used in the semantics described in [dRdBH$^+$01] as well as in the semantics of ASMs [Gur95],[BS03]).

Interleaving is axiomatized to be weakly fair (each continuously enabled transition eventually occurs). Assignments must be placed within a program frame $[.]_{V1,\ldots,Vn}$ to avoid expressions with infinitely many free variables (this is similar to TLA [Lam94], but our logic does not have built-in stuttering).[2] As an example the semantics of the program

$$[X := 3]_{X,Y,Z}$$

consists of all intervals $[I(0), I'(0), I(1)]$ with one program transition from $I(0)$ to $I'(0)$ that changes $X'$ to 3 and leaves $Y$ and $Z$ unchanged. All other variables may change arbitrarily (and are therefore not free in the formula). The environment transition from $I'(0)$ to $I(1)$ is not constrained by the program.

Exemplarily the semantics of the **until** operator, the sequential composition operator and the existential path quantifier is given:

$I \models \varphi$ **until** $\psi$ iff there exists $n \in \mathbb{N}, n \leq |I|$ with $(I(n), I'(n), \ldots) \models \psi$
and $(I(m), I'(m), \ldots) \models \varphi$ for all $0 \leq m < n$

$I \models \alpha; \beta$ iff there exists $n \in \mathbb{N}, n \leq |I|$ with $(I(0), I'(0), \ldots, I(n)) \models \alpha$
and $(I(n), I'(n), \ldots) \models \beta$
or $|I| = \infty$ and $I \models \alpha$

$I \models \mathbf{E}\ \varphi$ iff there exists $J$ with $J(0) = I(0)$ such that $J \models \varphi$

Other temporal logic operators, such as $\Box$, $\Diamond$ or **unless** can be derived, e.g.

$$\Diamond\ \varphi :\equiv \textsc{True}\ \textbf{until}\ \varphi$$
$$\Box\ \varphi :\equiv \neg \Diamond \neg\ \varphi$$
$$\varphi\ \textbf{unless}\ \psi :\equiv \Box\ \varphi \vee \varphi\ \textbf{until}\ \psi$$

### 3.2. Symbolic Execution and Induction

A typical sequent in proofs about interleaved programs has the form $P, E, I \vdash \varphi$. An interleaved program $P$ executes the system steps, $E$ contains a temporal formula that describes the behavior of the program's environment and $I$ is a predicate logic formula that describes the current state. $\varphi$ is the property which has to be shown.

To verify that $\varphi$ holds, symbolic execution is used. For example, a sequent of the form mentioned above might look like this:

$$[M := M + 1; \alpha]_M,\ \Box\ M' = M'', M = 2\ \vdash\ \Box\ M > 0$$

The program executed is $M := M + 1; \alpha$ ($\alpha$ can be an arbitrary program) and the environment is assumed not to change $M$ (formula $\Box\ M' = M''$). The current state $M = 2$ does not violate $\Box\ M > 0$. A symbolic execution step is used to show that the next program transition does not violate that formula too. The intuitive idea of a symbolic execution step is to execute the first program statement, i.e. applying the changes on the current state and to discard the first statement. In the example above, a symbolic execution step leads to a predicate logic goal for the initial state

$$M = 2 \vdash M > 0$$

and a sequent that describes the remaining interval from the second state on

$$\alpha,\ \Box\ M' = M'', M = 3\ \vdash\ \Box\ M > 0$$

Of course, the environment assumption has to be considered too, but it simply leaves $M$ unchanged in this example. More complex formulas in the succedent might change too during the step (e.g. if the formula in the succedent is a program too, it has to be symbolically executed like the example program in the antecedent).

[2] This replaces the global frame assumptions in [BBRS08, Bal05], which say that *all* other variables are unchanged

The principle of symbolic execution is also applied to temporal logic formulas. The basic principle of splitting a formula into a predicate logic part that describes the first step and a part that describes the remaining interval from the second state on remains the same. E.g. the $\Box$ or the **unless** operator can be executed by using the following equivalences

$$\Box\ \varphi \leftrightarrow \varphi \wedge \bullet\ \Box\ \varphi$$
$$\varphi\ \mathbf{unless}\ \psi \leftrightarrow \psi \vee (\varphi \wedge \bullet\ \varphi\ \mathbf{unless}\ \psi)$$

The basic idea to prove safety properties is to advance in the interval until a valuation that was considered earlier in the interval is reached. In this case a loop was executed, and the proof can be finished with an inductive argument.

The technical implementation of this idea uses theorems that introduce a counter $N$ and then apply well-founded induction on $N$. For a proof of $\Box\ \varphi$ this is simply done by rewriting $\Box\ \varphi$ to $\neg \Diamond \neg\ \varphi$ and by exploiting the equivalence

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

The equivalence states, that $\varphi$ is eventually true, if and only if a variable N storing a natural number can be decremented (note that $N = N'' + 1$ is equivalent to $N > 0 \wedge N'' = N - 1$) until $\varphi$ becomes true. The "if" direction of this equivalence is easy to show, for the "only if" direction, the initial value of $N$ must be chosen to be at least the number of (system + environment) steps needed to reach the state where $\varphi$ holds. Proving an unless formula (as needed later in rely-guarantee proofs) can be reduced to the case of an eventually formula using the equivalence

$$\varphi\ \mathbf{unless}\ \psi \leftrightarrow \forall\ B.\ (\Diamond\ B) \rightarrow (\varphi\ \mathbf{unless}\ (\psi \vee B))$$

($\varphi$ **unless** $\psi$ is true, if it is true on every prefix of the trace, that is terminated by the first time when boolean variable B becomes true). KIV uses a rule "VD incuction" that integrates applying the above rules and well-founded induction in one proof step (the rule essentially enables to create *V*erification *D*iagrams of safety formulas dynamically as sequent calculus proofs, therefore the name).

General safety formulas satisfy the principle, that they are true on an infinite interval $I$, already if every finite prefix $I|_n := (I(0), I'(0), \ldots, I(n))$ of $I$ can be extended with some interval $J$ that starts with $J(0) = I(n)$ such that $\varphi$ is true on this concatenation (see e.g. [AS87]). In addition to always and unless formulas, this class also contains all sequential programs without local variables and all predicate logic formulas (including those that mention primed and double primed variables). Safety formulas are closed against conjunction and disjunction (but not negation). To express the general principle of induction over the length of a prefix seems unfortunately not expressible with the usual ITL operators. A special operator $\mathbf{prefix}(\varphi, \psi)$ is needed. Again the idea is to use the first time, when a boolean variable (used as $\psi$) becomes true as the length of the prefix. The definition of its semantics $I \models \mathbf{prefix}(\varphi, \psi)$ has three cases:

1. If $I \models \neg \Diamond\ \psi$ then $I \models \varphi$ is required.
2. If $I$ is finite, $I \models \Diamond\ \psi$ and $\psi$ becomes true in the last state only, then $I \models \varphi$ is required too.
3. Otherwise, if n is minimal such that $I|_n \models \psi$ then for some interval $J$ with $J(0) = I(n)$

   $$(I(0), I'(0),\ \ldots, I(n), J'(0), J(1), \ldots) \models \varphi$$

   must hold.

The first disjunct of this definition could be defined arbitrarily. It is chosen to give simple symbolic execution rules. The second clause is necessary to cope with finite intervals, which may not be extended (an interval I of length one models $X = X'$. The only prefix of I is I itself, which could be extended to model $X \neq X'$ making the theorem below false). The third clause is the usual definition as used in [AS87].

Safety formulas $\varphi$ satisfy

$$\varphi \leftrightarrow (\forall\ B.\ \Diamond\ B \rightarrow \mathbf{prefix}(\varphi, B))$$

which gives the desired induction principle (again using the equivalence (1) on $\Diamond\ B$). Symbolic execution of $\mathbf{prefix}(\varphi, B)$ always gives two cases, depending on the current value of B. If B is false then executing a step of $\mathbf{prefix}(\varphi, B)$ gives the same results as executing $\varphi$ and applying the prefix operator after the step (since

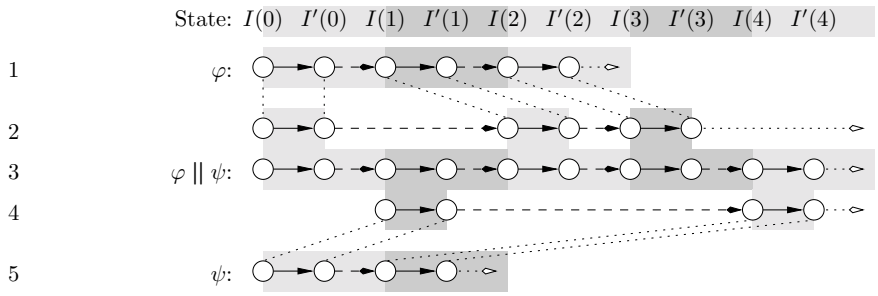



**Fig. 4.** Interleaving of two formulas

the first step is just a step of the considered interval). If B is true, then $\mathbf{prefix}(\varphi, B)$ is equivalent to $\mathbf{E}\,\varphi$ since we may now choose an arbitrary interval $J$ starting with the current state. $\mathbf{E}\,\alpha$ is typically true for ordinary sequential programs $\alpha$, but note that the condition is necessary since it is possible to mix programs and formulas: $\mathbf{E}\,\alpha$; **false** is false.

Formally assuming $\tau$ is a predicate logic formula describing the first system and environment step (it may refer to primed and double primed variables) the axioms

$$\begin{aligned}
\neg\, B &\;\rightarrow\; (\mathbf{prefix}(\tau \wedge \mathrm{last}, \psi) \leftrightarrow \tau \wedge \mathrm{last}) \\
\neg\, B &\;\rightarrow\; (\mathbf{prefix}(\tau \wedge \circ\, \varphi) \leftrightarrow \tau \wedge \circ\, \mathbf{prefix}(\varphi, B)) \\
\mathbf{prefix}(\varphi \vee \psi, B) &\;\leftrightarrow\; \mathbf{prefix}(\varphi, B) \vee \mathbf{prefix}(\psi, B) \\
B &\;\rightarrow\; (\mathbf{prefix}(\varphi, B) \leftrightarrow \mathbf{E}\,\varphi)
\end{aligned}$$

are used to symbolically execute formulas with the **prefix** operator.

### 3.3. Executing Interleaved Programs

Two interleaved programs are executed by executing the first transition from one or the other. After this, the proof continues with interleaving the remaining formulas. For example, if there are two interleaved programs in the antecedent

$$[M := 1; \alpha_1 \parallel N := 2; \alpha_2]_{M,N},\ \Gamma \vdash \Delta$$

this formula can be transformed into the following two cases:

$$[M := 1; (\alpha_1 \parallel N := 2; \alpha_2)]_{M,N,},\ \Gamma \vdash \Delta$$

$$[N := 2; (M := 1; \alpha_1 \parallel \alpha_2)]_{M,N,},\ \Gamma \vdash \Delta$$

which can be symbolically executed by the mechanism described above.

As our framework does not distinguish between programs and formulas, it is necessary to define interleaving of two temporal formulas $\varphi$ and $\psi$. This definition is based on interleaving two intervals $I_1$ and $I_2$ with $I_1 \models \varphi$ and $I_2 \models \psi$. A formal definition (that includes blocking steps) is given in [Bal05], here we show in Figure 4, how the steps of the interleaved system are composed from steps of the components. Line 1 and line 5 represent the two traces $I_1$ and $I_2$ of the components $\varphi$ and $\psi$, where a full arrow depicts a step of the component and a dashed arrow stands for a step of the component's environment (the dotted arrows at the end of all intervals symbolizes that the intervalls may continue after the last depicted state). Line 3 depicts a trace of the interleaving $\varphi \parallel \psi$. Here a full arrow depicts a step of the interleaved system, i.e. a step of either $\varphi$ or $\psi$, and a dashed arrow depicts a step of the system's environment. The relation of local steps/states to the global steps/states is depicted with dotted lines. From the point of view of a component, an environment step (dashed arrow in lines 2 and 4) is a sequence of steps of either another component within the same interleaved system (full arrow in line 3) or a step which is also an environment step of the entire system (dashed arrow in line 3).

If one of the intervals $I_1$ or $I_2$ is finite, once the last state of that interval has been reached the continuation of the interleaved intervall $I$ is equal to the remaining interval $I_1$ or $I_2$. The end of the terminating interval

(i.e. the last state) has to match one of the states of the remaining interval $I$ (not necessarily the first one). If both intervals $I_1$ and $I_2$ are finite, $I$ is also finite.

The interleaving operator is associative and commutative. Therefore it is possible to interleave several programs or formulas by using nested interleaving operators.

### 3.4. Compositionality of Interleaving

One of the key features of our logic, which is due to the interleaving of system and environment steps, is that the interleaving operator is compositional. It allows us to replace interleaved components with abstractions. To apply compositionality, rule *comp* is used:

$$\frac{\alpha \vdash \varphi \qquad \beta \vdash \psi \qquad \varphi \parallel \psi, \Gamma \vdash \chi}{\alpha \parallel \beta, \Gamma \vdash \chi}\ comp$$

To prove that an interleaved program $\alpha \parallel \beta$ satisfies a property $\chi$ in some context $\Gamma$ (conclusion), it is sufficient to show that programs $\alpha$ and $\beta$ individually satisfy properties $\varphi$ and $\psi$ respectively (first and second premise; usually $\varphi$ and $\psi$ are e.g. rely-guarantee properties – see next section). Then each program can be replaced by its property in the premise. This rule is essential to prove the rely guarantee theorem presented in the next section. A similar rule holds for sequential composition $\alpha; \beta$ in place of $\alpha \parallel \beta$. For full details about symbolic execution, interleaving and induction in KIV see [Bal05].

## 4. Rely-Guarantee Reasoning

In [BNBR08] we presented a generic composition theorem for proving safety formulas similar to standard composition theorems such as [dRdBH$^+$01, CC96]. To show linearizability we need a suitable variant of this theorem that can prove refinement between atomic abstract operations and interleaved concrete operations. We first describe the idea of rely-guarantee proofs and their encoding in our temporal logic. After that we present the theorem used here.

### 4.1. Rely-Guarantee Properties

The rely-guarantee proof method usually uses a two state rely predicate $R_i : cstate \times cstate$ over arbitrary states of type *cstate* for every process (component) $i : nat$ of a concurrent system. $R_i$ defines restrictions on environment steps that are necessary for $i$ to work correctly. In return, the impact of each process on the environment is described using a binary predicate $G_i : cstate \times cstate$ which preserves the rely conditions of all other processes. This informal argument involves circular reasoning. To break the circularity, a special "while-plus" operator $\overset{+}{\rightarrow}$ (as defined in [AL95]) is used and it is proved that each process $i$ satisfies $R_i \overset{+}{\rightarrow} G_i$. This means informally that if $R_i$ holds up to step $k$, then $G_i$ must hold up to step $k+1$. With this operator it is possible to express that a component violates its guarantee $G_i$ *only after* its rely condition $R_i$ has been violated.

The explicit separation between program and environment transitions in our logic allows us to specify guarantees as predicates $G_i(CS, CS')$ with unprimed and primed variables $CS : cstate$ that describe steps of process $i$. Rely predicates $R_i(CS', CS'')$ restrict steps of $i$'s environment and use primed and doubly primed variables. The formal definition of $\overset{+}{\rightarrow}$ is then based on the **unless** operator:

$$R_i \overset{+}{\rightarrow} G_i := G_i(CS, CS')\ \mathbf{unless}\ (G_i(CS, CS') \wedge \neg R_i(CS', CS''))$$

### 4.2. Proof Obligations for Components

Processes execute a generic operation $\mathrm{COP}(i, In; CS, Out)$. Its input parameters include the process identifier and an input array $In : nat \rightarrow input$. Once $\mathrm{COP}(i, \dots)$ is invoked, $i$ works on a local copy of the current value $In(i)$. The transient parameters consist of a state variable $CS$ and an additional output parameter

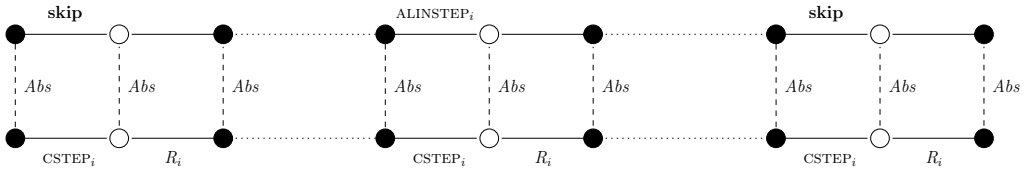



**Fig. 5.** Diagram relating COP$(i, \dots)$ steps and AOP$(i, \dots)$ steps

$Out : nat \to output$, which enables the operation to return an output value $Out(i)$ . This generic operation is instantiated in the verification of the stack case study (see section 5) with an operation that randomly executes a single CPush or CPop operation.

Our formal definition of the rely-guarantee theorem for a process $i$ is slightly more complex than stated in the former subsection, since it additionally uses an invariant single state predicate $Inv : cstate$.

$$\mathrm{COP}(i, In; CS, Out), \Box\ R_i(CS', CS''), Inv(CS) \vdash \Box\ (G_i(CS, CS') \wedge Inv(CS) \wedge Inv(CS')) \tag{2}$$

The sequent states that executing the operation COP$(i, \dots)$ from an initial state that satisfies $Inv$ in an environment that never violates $R_i$, will produce runs that guarantee $G_i$ in every step and will also preserve $Inv$. Suitable instantiations for $Inv$ and $R_i$ will be presented in section 5.

To prove (2), the main proof obligation to be shown is that the guarantee is never violated before the rely condition:

$$\mathrm{COP}(i, In; CS, Out), Inv(CS) \vdash R_i(CS', CS'') \overset{+}{\to} G_i(CS, CS') \tag{3}$$

As stated above, the guarantee of each process $i$ must imply the rely conditions of all other processes.

$$\forall\ i, j \neq i.\ G_i(cs_1, cs_2) \to R_j(cs_1, cs_2) \tag{4}$$

This proof obligation is purely first order logic. It is described using static variables $cs_1, cs_2 : cstate$ and ensures that no step of $i$ will violate the rely condition of another process $j$. Since an entire environment step of process $i$ may consist of several steps of other processes, the rely predicate has to be transitive to make sure that a full environment step also satisfies $R_i$.

$$\forall\ i.\ R_i(cs_1, cs_2) \wedge R_i(cs_2, cs_3) \to R_i(cs_1, cs_3) \tag{5}$$

These two premises are enough to deduce that no component violates the rely condition of another component. It finally remains to enforce that the invariant is preserved in all steps. In (2), the environment of process $i$ always satisfies $R_i$ by assumption. It is sufficient to require that the invariant predicate is stable over each local rely condition.

$$\forall\ i.\ Inv(cs_1) \wedge R_i(cs_1, cs_2) \to Inv(cs_2) \tag{6}$$

With (4) it then follows that for all $i$ the invariant is also stable over $G_i$. Altogether we get: if all processes satisfy conditions (3) to (6), then each process also satisfies (2).

### 4.3. Rely-Guarantee Theorem for Refinement

Usually, rely-guarantee techniques are used to show that an interleaved system preserves a global guarantee. In this work, rely-guarantee is used to show a refinement property, i.e. that any run of an interleaved system of concrete operations has an equivalent run of a system with abstract operations. As programs are formulas in our framework, trace inclusion for a concrete component COP and an abstract component AOP can be simply expressed by a formula $\mathrm{COP}(CS) \vdash \mathrm{AOP}(AS)$, where $AS$ is an abstract state variable $AS : astate$. This formula does not consider data refinement, nor the possible influence of the environment. A refinement formula that includes both is

$$\begin{aligned}\mathrm{COP}(i, In; CS, Out), \Box\ R_i(CS', CS''), Inv(CS) \vdash \exists\ AS.\ (\quad & \Box\ (Abs(AS, CS) \wedge Abs(AS', CS')) \\ & \wedge\ \mathrm{AOP}(i, In; AS, Out))\end{aligned} \tag{7}$$

The left hand side of the sequence defines concrete runs (semantically intervals) of COP$(i, \dots)$, which start in a state satisfying $Inv$, in an environment that always respects $R_i$. For every such run the right hand side

proves the existence of a suitable abstract run, working on the same input and returning the same output as $\mathrm{COP}(i, \dots)$. Such an abstract run is suitable if it has the same length as the concrete run and refinement relation $Abs : astate \times cstate$ holds before and after each step (unprimed and primed variables).

In order to prove linearizability, the generic operation $\mathrm{AOP}(i, \dots)$ will be instantiated as depicted in Figure 5. $\mathrm{AOP}(i, \dots)$ must take place in one atomic step ($\textsc{alinstep}_i$) whereas other steps must leave the abstract data structure unchanged, i.e. are basically *skip* steps. The lower interval depicts a concrete run of $\mathrm{COP}(i, \dots)$, where $\textsc{cstep}_i$ denotes a step of $\mathrm{COP}(i, \dots)$. Finding an abstract run as previously described for every concrete run, intuitively corresponds to the idea of linearizability [HW90]. The effect of the concrete operation on the abstract data structure happens atomically at the linearization point, which is somewhere between invocation and return.

It remains to formally define the full scenario of any number of processes calling any number of concrete operations. A process calls $\mathrm{COP}(i, \dots)$ arbitrarily often. It may finish at any time or never. Inbetween these calls the process will do some other irrelevant computations, i.e. steps that are visible from outside as **skip** steps. Formally, this sequence of $\mathrm{COP}(i, \dots)$-operations is specified by the procedure CSeq

$$
\begin{array}{l}
\textsc{CSeq}(i; In, CS, Out)\ \{ \\
\quad (\ \mathbf{skip}\ \vee\ \mathrm{COP}(i, In; CS, Out))^* \\
\}
\end{array}
$$

where $\alpha^*$ iterates $\alpha$ finitly or infinitly often. As a side-effect, **skip** steps must satisfy the local guarantee, i.e. the local guarantee must be reflexive:

$$\forall\ i.\ G_i(cs, cs) \tag{8}$$

To specify an arbitrary number of parallel processes, a recursively defined spawn procedure is used.

$$
\begin{array}{l}
\textsc{CSpawn}(n; In, CS, Out)\ \{ \\
\quad \mathbf{if}\ n = 0\ \mathbf{then} \\
\qquad \textsc{CSeq}(0; In, CS, Out) \\
\quad \mathbf{else} \\
\qquad \textsc{CSeq}(n; In, CS, Out)\ \|\ \textsc{CSpawn}(n-1; In, CS, Out) \\
\}
\end{array}
$$

The spawn procedure generates $n+1$ processes. For every number $n$, the spawning process $\textsc{CSpawn}(n; \dots)$ can be expanded to $\textsc{CSeq}(n; \dots)\ \|\ \dots\ \|\ \textsc{CSeq}(0; \dots)$. Processes are discerned by a process identifier ranging from 0 to n. The abstract level has a similar spawning procedure $\textsc{ASpawn}(n; In, AS, Out)$ that calls $\textsc{ASeq}(i; In, AS, Out)$ for $i \leq n$.

Since our generic setting assumes that even the interleaving of all processes still has a global environment which can do steps, we also need a global binary predicate $R : cstate \times cstate$. It is used to restrict the possible behaviour of the system's environment in a way that preserves each local rely $R_i$.

$$\forall\ i.\ R(cs_1, cs_2) \rightarrow R_i(cs_1, cs_2) \tag{9}$$

With these definitions it is possible to prove the following composition theorem.

**Theorem 1.** If for all $0 \leq i, j \leq n$, $i \neq j$ formulas (3) to (9) hold, then:

$$
\begin{array}{l}
\textsc{CSpawn}(n; In, CS, Out), \Box\ R(CS', CS''), Inv(CS) \vdash \exists\ AS.\ (\quad \Box\ (Abs(AS, CS) \wedge Abs(AS', CS')) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \textsc{ASpawn}(n; In, AS, Out))
\end{array} \tag{10}
$$

The theorem states that for every interleaved run of an arbitrary number of processes, which all execute $\mathrm{COP}(i, \dots)$ as often as they like, a corresponding abstract run of operations $\mathrm{AOP}(i, \dots)$ exists. Moreover, concrete and abstract operations execute on the same input and yield the same output.

Unlike standard rely-guarantee theorems (e.g. [dRdBH$^+$01, CC96]), this theorem does not reason about one interleaved system but about two interleaved systems connected by refinement. The theorem was formally proved in KIV and is available online [KIV]. The proof is similar to the one outlined in [BNBR08].

Finally, note that CSpawn still runs in an environment with a global rely condition $R$ that is the conjunction of the local rely conditions $R_i$. All local rely conditions defined for the stack and the queue example (Sections 5 and 6) will require only that local pointers are not modified and that the cells of the current stack or queue representation are unchanged. Therefore, it is possible to collect all other cells by a garbage collection algorithm that runs in the environment of CSpawn.

```
APush(v; Stack) {
    skip*;
    Stack := push(v, Stack);
    skip*
}
```

```
APop(; Stack, O) {
    let Lo = Empty in {
        skip*;
        if Stack ≠ Empty then {
            Lo := top(Stack),
            Stack := pop(Stack);
        }
        skip*;
        O := Lo
    }
}
```

**Fig. 6.** Formal definition of the abstract stack operations

## 5. Correctness of the Stack Algorithms

In this section we describe how the generic theory has been instantiated in the stack case study to show linearizability. The first subsection presents the instantiation of the generic operations COP$(i, \dots)$ resp. AOP$(i, \dots)$ and of the refinement relation *Abs*. The other subsections present the invariant properties and rely-guarantee conditions used in the verification.

### 5.1. Concrete and Abstract Stack Operations

The generic state variable *CS* becomes a tuple consisting of a shared state *Top*, *Hp* and local states *UNew*$(i)$, *USuccess*$(i)$, *OTop*$(i)$, *OSuccess*$(i)$ for every process $i$ . The concrete definition of COP$(i, \dots)$ then is

```
COP(i, In; Top, Hp, UNew, USuccess, OTop, OSuccess, Out) {
    CPush(In(i); Top, Hp, UNew(i), USuccess(i))
     ∨
    CPop(; Top, Hp, OTop(i), OSuccess(i), Out(i))
}
```

It randomly calls a CPush or CPop operation. New input is chosen for the push process as the input *In* is randomly reassigned in every environment step. AOP$(i, \dots)$ is defined similarly. It works on an algebraic Stack which corresponds to the abstract state *AS* and calls APush or APop as defined in Figure 6. These operations modify the stack in one atomic assignment (*push* and *pop* are atomic operations defined on the algebraic Stack). Additional skip steps ensure that the abstract run corresponding to the concrete run of COP$(i, \dots)$ will have the same length. Note that the output of APop is kept in a local variable *Lo* to ensure that the visible output value *O* is changed in the same (last) step as in the concrete operation CPop. This guarantees that the outputs are set simultaneously in corresponding abstract and concrete runs.

To instantiate (10) in Theorem 1, we need a rely condition $R(CS', CS'')$ for the global environment and an abstraction relation *Abs*. The global rely condition is simply the conjunction of all local rely conditions, since no other process from outside of the system should interfere with any of the local processes. The abstraction relation is recursively defined as

$$Abs(\text{Empty}, top, hp) \leftrightarrow top = \text{Null} \tag{11}$$

$$\begin{aligned} Abs(push(v, st), top, hp) \leftrightarrow \quad & top \neq \text{Null} \wedge top \in hp \wedge hp[top].\text{val} = v \\ & \wedge\ Abs(st, hp[top].\text{nxt}, hp) \end{aligned} \tag{12}$$

*st* is the abstract stack and *v* is a data value. The empty stack is represented by the Null pointer (formula (11)). For the recursive case in formula (12), *top* has to be allocated in *hp*, the data value of the cell referenced by *top* has to be the data value on top of the stack and the rest of the stack has to be represented by the next pointer of the cell referenced by *top*.

To prove the verification conditions of Theorem 1 we furthermore need local rely conditions $R_i$ for each process $i$ and an invariant *Inv*. Once this is done, the proof itself can be done by simply stepping through

the program. Except for the while loops which often need a manual generalization of the precondition at the beginning to get an invariant, these proofs require only little interaction.

### 5.2. Invariant

Although some parts of the invariant and rely conditions are schematic, finding suitable conditions is the main creative task in verifying a lock-free algorithm. The invariant *Inv* consists of three parts. The first part is schematic. It states that the data structure represented by the heap and the top variable has to be *valid*, i.e. it really represents a stack and is neither cyclic nor has invalid references.

$$\mathit{valid}(\mathit{top}, \mathit{hp}) \leftrightarrow \exists\ \mathit{st}.\ \mathit{Abs}(\mathit{st}, \mathit{top}, \mathit{hp})$$

The other two parts of the invariant assert properties of the local variables used in CPush and CPop. A standard assumption is that local variables contain either a Null pointer, or must point to an allocated heap address. This prevents illegal dereferencing. For the pop operation the only relevant variable is *OTop*, and this is all that is needed:

$$\mathit{Inv}_{\mathit{pop}}(\mathit{otop}, \mathit{hp}) \leftrightarrow \forall\ i.\ \mathit{otop}(i) = \textsc{Null} \vee \mathit{otop}(i) \in \mathit{hp}$$

For the push operation the same proposition is needed for *UNew*. Additionally, the cell allocated by a push process at the start of the operation must not be part of the stack before a successful CAS of the push operation integrates it into the stack. Without this proposition, modifying the cell could accidentally change the stack. A simple way to characterize "before a successful CAS" is to check that *USuccess* is false. We get the following invariant for push:

$$\begin{aligned}\mathit{Inv}_{\mathit{push}}(\mathit{unew}, \mathit{usuccess}, \mathit{top}, \mathit{hp}) \leftrightarrow \forall\ i.\quad & (\mathit{unew}(i) = \textsc{Null} \vee \mathit{unew}(i) \in \mathit{hp}) \\ & \wedge\ (\neg\ \mathit{usuccess}(i) \rightarrow \neg\ \mathit{reachable}(\mathit{top}, \mathit{unew}(i), \mathit{hp}))\end{aligned}$$

Predicate $\mathit{reachable}(\mathit{top}, r, \mathit{hp})$ is true iff $r$ is a reference of the stack representation which is reachable from *top*. Reachability is defined using lists of references $p = [r_1, \ldots r_n]$ which are paths in the heap chained by .nxt:

$$\mathit{reachable}(\mathit{top}, r, \mathit{hp}) \leftrightarrow \exists\ p.\ \mathit{path}(\mathit{top} + p + r, \mathit{hp})$$

where predicate *path* is recursively defined as:

$$\neg\ \mathit{path}([\,], \mathit{hp})$$

$$\mathit{path}([r + [\,], \mathit{hp}) \leftrightarrow r \in \mathit{hp} \wedge r \neq \textsc{Null}$$

$$\mathit{path}(r_1 + r_2 + p, \mathit{hp}) \leftrightarrow r_1 \in \mathit{hp} \wedge r_1 \neq \textsc{Null} \wedge \mathit{hp}[r_1].\mathrm{nxt} = r_2 \wedge \mathit{path}(r_2 + p, \mathit{hp})$$

The '+'–operator is overloaded to concatenate lists as well as elements. Finally, a proposition is needed that the references of functions *UNew* and *OTop* are disjoint. This ensures that the changes a push operation makes on its newly allocated cell, do not interfere with other push operations or pop operations retrieving the data values out of *OTop* cells. Note that the fact that *UNew* cells are not reachable, while *OTop* cells are, when the assignment *OTop* := *Top* is executed in CPop, is not sufficient to ensure disjointness, since an *OTop* cell is not guaranteed to stay within the stack after the assignment has happened. Formally the predicate *disj* is used.

$$\begin{aligned}\mathit{disj}(\mathit{unew}, \mathit{usuccess}, \mathit{otop}) \quad \leftrightarrow \forall\ i, j.\ i \neq j \rightarrow \quad & (\mathit{unew}(i) = \textsc{Null} \vee \mathit{unew}(i) \neq \mathit{unew}(j)) \\ & \wedge\ (\neg\ \mathit{usuccess}(i) \wedge \mathit{otop}(j) \neq \textsc{Null} \rightarrow \mathit{unew}(i) \neq \mathit{otop}(j))\end{aligned}$$

Altogether the complete invariant is

$$\mathit{Inv}(\mathit{cs}) \quad \leftrightarrow \quad \mathit{valid}(\mathit{top}, \mathit{hp}) \wedge \mathit{Inv}_{\mathit{push}}(\mathit{unew}, \mathit{usuccess}, \mathit{top}, \mathit{hp}) \wedge \mathit{disj}(\mathit{unew}, \mathit{usuccess}, \mathit{otop})$$

### 5.3. Rely and Guarantee Properties

The local rely condition $R_i$ consists of three parts. The first part is schematic: it states that the invariant *Inv* is preserved, as required by condition (6). The second and third part are again assertions for push and

pop.

$$R_i(cs_1, cs_2) \;\leftrightarrow\; (Inv(cs_1) \rightarrow Inv(cs_2)) \wedge R_{i,push}(unew_1, usuccess_1, hp_1, unew_2, usuccess_2, hp_2) \wedge R_{i,pop}(otop_1, osuccess_1, hp_1, otop_2, osuccess_2, hp_2)$$

Both rely conditions for push and pop consist of a trivial part that asserts that the local variables are not changed by the environment. In addition, the content of the allocated cell should not be changed as long as the push operation is trying to include it into the stack. This leads to the following rely for push operations:

$$R_{i,push}(unew_1, usuccess_1, hp_1, unew_2, usuccess_2, hp_2) \;\leftrightarrow\; usuccess_1(i) = usuccess_2(i) \wedge unew_1(i) = unew_2(i) \wedge (unew_1(i) \neq \textsc{Null} \wedge \neg\, usuccess_1(i) \rightarrow hp_1[unew_1(i)] = hp_2[unew_2(i)])$$

State variables with index one denote a state before an environment step and index two denotes the state afterwards. The rely for pop operations is similar. The content of the cell *OTop* is not changed as long as the pop operation is active. This condition is necessary to prevent unwanted changes of the stack, as they could occur with the ABA problem.

$$R_{i,pop}(otop_1, osuccess_1, hp_1, otop_2, osuccess_2, hp_2) \;\leftrightarrow\; osuccess_1(i) = osuccess_2(i) \wedge otop_1(i) = otop_2(i) \wedge (otop_1(i) \neq \textsc{Null} \wedge \neg\, osuccess_1(i) \rightarrow hp_1[otop_1(i)] = hp_2[otop_2(i)])$$

It finally remains to define the guarantee of each process. As the guarantee only has to be strong enough to imply the rely properties for all other processes (condition (4)), the guarantee can be defined as their conjunction.

$$\forall\, i.\; G_i(cs_1, cs_2) \leftrightarrow \forall\, j.\; j \neq i \rightarrow R_j(cs_1, cs_2)$$

The proofs for all proof obligations are available online at [KIV]. The predicate logic proofs for conditions (4) to (6), (8) and (9) are simple. The only complex proofs are the local rely-guarantee condition (3) and the refinement condition (7). These proofs split into one that symbolically executes push and one that executes pop. The proofs for (3) use induction over the unless formula as described in Section 3.2. Finding an induction principle to prove (7) is more difficult, since the formula contains existential quantifiers: one is the explicit $\exists\, AS$. The other is implicit in the let for the variable *Lo* in the APop algorithm. Since in general safety formulas are not closed against existential quantification (a counter example can be found e.g. in [JT96]) induction is not admissible. In general, image-finiteness of the existential quantifier seems to be required for $\exists\, AS.\; \varphi$ to be a safety formula when $\varphi$ is. For the case study the solution is easier, since *Abs* is a partial function that admits only one choice anyway (and therefore is trivially image-finite). Therefore we define the skolem function *absf* of the *Abs* predicate by

$$Abs(\textsc{Stack}, Top, Hp) \rightarrow Abs(absf(Top, Hp), Top, Hp)$$

and instantiate the quantifier immediately with $absf(Top, Hp)$. Note that *absf* is always defined since the invariant contains $valid(top, hp)$. The existential variable *Lo* from the let of the APop algorithm can be instantiated to **if** *OSuccess* **then** *Lo* **else** Empty using the variable of the same name from the let of the CPop algorithm. After that, the resulting formula is a conjunction of a sequential program and an always formula, which is a safety formula. Prefix induction as explained in Section 3.2 is used to prove it. The appropriate linearization point in the refinement proof is chosen interactively. As an interesting aside, there is no need to generalize the precondition of the while loops to an (Hoare style) invariant that holds whenever the algorithm is at the start of the loop. This seems to be a particularity of lock-free algorithms, where restarting the loop discards the results of previous iterations.

## 6. Lock-free Queue Algorithm

This section describes the application of the introduced verification technique on a more complicated lock-free algorithm. It is a slightly optimized implementation of a concurrent, lock-free queue presented by Doherty

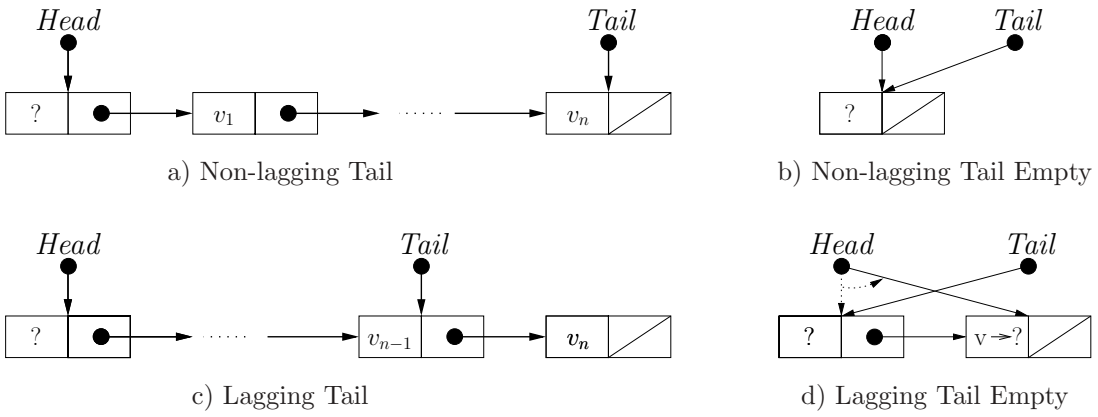



**Fig. 7.** Queue Representation

et. al. in [DGLM04], based on the original algorithm of Michael and Scott [MS96]. Subsection 6.1 presents the queue algorithm focusing mainly on the dequeue operation, as the determination of the linearization point for this operation poses an additional problem which the stack example does not have. In Subsection 6.2 the instantiation of the generic operations is explained and two of the rely conditions are described. Subsection 6.3 outlines the proof for the critical case of the dequeue operation. Again full details of all proofs are available on the Web [KIV].

### 6.1. Queue Specification

On the concrete level the queue is represented by means of a singly linked list of nodes, a global pointer *Head* indicating the front of the queue and a global pointer *Tail* which marks the end of the queue as illustrated in Figure 7 a) and b). It is convenient to use a *dummy node* to avoid a special case for an empty queue. The value of this node is irrelevant and is represented by a question mark. At all times, *Head* points to the dummy node, which is the entry point of the representation list.

Two operations are implemented: the enqueue operation *CEnq* adds a node at the end of the queue whereas *CDeq* removes the first node from the queue, returning its value if the queue is not empty. Otherwise, *Head*'s next reference is NULL and special return value *empty* indicates that dequeue has been executed on an empty queue.

When *CEnq* tries to attach a new node *Newe* at the end of the queue, two global changes must be made. The last node's next field must be set to the new node *and* the tail pointer itself must be shifted to point to *Newe*. Unfortunately CAS allows only one atomic write access. CAS is therefore utilized twice to cope with this problem. The first successful CAS-transition sets *Tail*'s next field to point to *Newe* (the linearization point), leaving *Tail* lagging one node behind the last node of the queue, as shown in Figure 7 c). The second employment of CAS shifts the lagging tail pointer to its successive node *Newe*. These two CAS-transitions do *not* both take effect atomically. Some other process $j$ which executes *CEnq* might encounter a state inbetween, when *Tail* does not mark the end of the queue. Whenever $j$ encounters a lagging tail pointer, it knows that another process $i$ has already successfully attached a new node but has not yet shifted *Tail*. Process $j$ helps $i$ by applying CAS to try to shift *Tail* to the next node of the linked list. Hence, *Tail* will never lag more than one node behind. All other aspects of *CEnq* are similar to the push operation in the stack example. In particular, no additional difficulties arise when trying to determine the linearization point.

The dequeue operation (see Figure 8 a)) in contrast has a non-trivial linearization point and is therefore presented in more detail. A process $i$ executing *CDeq* takes a snapshot *Hdd* of the global head pointer in line C5. Then *Hdd*'s next reference is stored in *Nxtd*. If the following check in line C7 fails, $i$ must reiterate the loop-body, as the snapshot *Hdd* has become obsolete. Otherwise, if *Nxtd* is NULL then the queue was empty when $i$ executed line C6. *CDeq* completes and returns *empty*. If in contrast *Nxtd* is not null, process $i$ locally stores *Nxtd*'s value and tries to shift *Head*, making *Nxtd* the new dummy node, line C13. If this CAS-transition fails, $i$ executes the loop-body again. Otherwise *Head* is shifted and $i$ then checks for a special configuration which emerges from shifting *Head* when the queue contains exactly *one* value $v$ and the tail

```
C1  CDeq(; Hp, Head, Tail, Hdd, Nxtd, SuccD, O) {
C2      let Lo = EMPTY, Tld = NULL in {
C3          SuccD := FALSE;
C4          while ¬ SuccD do {
C5              Hdd := Head;
C6              Nxtd := Hp[Hdd].nxt;
C7              if Hdd = Head then {
C8                  if Nxtd = NULL then {
C9                      Lo := EMPTY;
C10                     SuccD := TRUE;
C11                 } else {
C12                     Lo := Hp[Nxtd].val;
C13                     CAS(Hdd, Nxtd; Head, SuccD);
C14                     if SuccD then {
C15                         Tld := Tail;
C16                         if Tld = Hdd then {
C17                             CAS(Tld, Nxtd; Tail);
C18                         }
C19                     }
C20                 }
C21             }
C22         }
C23         O := Lo;
C24     }
C25 }
```

a) Concrete Dequeue

```
A1  ADeq(; Queue, O) {
A2      let Lo = EMPTY in {
A3          skip*;
A4          if Queue ≠ EMPTY then {
A5              Lo := head(Queue),
A6              Queue := deq(Queue);
A7          }
A8          skip*;
A9          O := Lo
A10     }
A11 }
```

b) Abstract Dequeue

**Fig. 8.** Formal Specification of Dequeue

pointer is lagging behind. This configuration is depicted in Figure 7 d). *Head* gets shifted ahead of *Tail*. In this situation only, $i$ can help out the enqueueing-process which has enqueued $v$, to shift the lagging tail pointer. The remaining lines of code, C14 - C17, deal with this situation. Note that *CDeq* reads the global tail pointer at most once. The original implementation of Michael and Scott reads the shared tail pointer whenever the loop-body is executed. The current implementation reduces shared memory access, if the dequeue loop-body has to be executed several times.

Finding the possible linearization point when process $i$ runs *CDeq* is more complicated. Two cases are discerned depending on what is read in line C6. If *Nxtd* is set to a non-null reference and *Head* remains unchanged after $i$ took the snapshot, the linearization point simply coincides with the successful CAS-transition in line C13. This transition removes the oldest value from the queue whereas all other transitions leave the queue untouched. The critical case in which *Nxtd* is set to NULL is depicted in Figure 9 (see also Figure 8 b) for the abstract dequeue operation). Only in this situation can *CDeq* assure that it was executed on an empty queue. However, if *Nxtd* is NULL at C6 this does not guarantee that *CDeq* will complete and return empty. This is because if the snapshot *Hdd* becomes obsolete between execution of line C6 and C7, $i$ has to retry. In this case, execution of line C6 has not been a linearization point, i.e. concrete trace (1) corresponds to abstract trace (4). If the snapshot is still accurate when $i$ executes line C7, *CDeq* completes returning *empty*. Hence, concrete trace (2) corresponds to abstract trace (3) and executing line C6 has been a linearization point corresponding to abstractly running dequeue on an empty queue.

The queue algorithm belongs to a different class of lock-free algorithms than the stack algorithm. Whether a transition (C6) is a linearization point depends on future behaviour which cannot be determined at the point of execution. It is this place, where verification approaches that refine single steps of a concrete algorithm individually, run into problems and require additional techniques since an abstract V-shaped diagram is refined to a Y-shaped diagram (moving nondeterminism backward). The automata based approach presented by Doherty et. al. in [DGLM04] introduces an intermediate automaton and applies backward simulation in order to complete the formal verification. Vafeiadis [Vaf07] proposes to use a prophecy variable in his PhD to

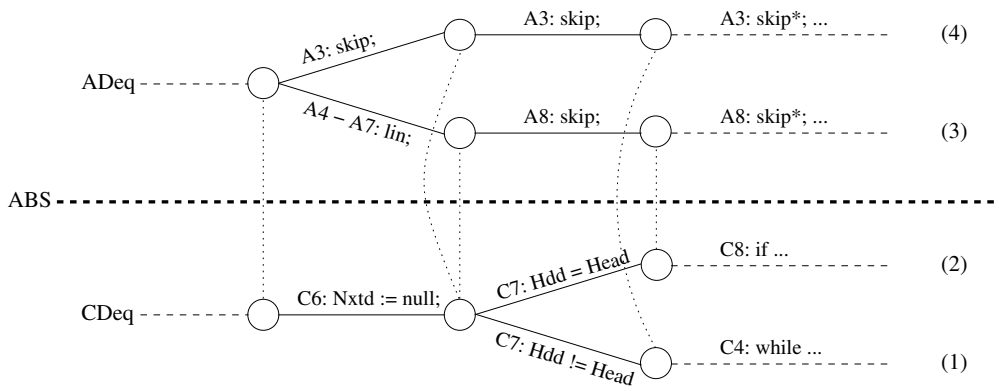



**Fig. 9.** Dequeue Empty Trace Relations

handle the problem (which is basically equivalent). The solution of our approach is easier, since we directly verify trace inclusion.

## 6.2. Instantiation of Generic Operations and Rely-Guarantee Conditions

As in the stack example, generic operation *COP* is instantiated by the nondeterministic disjunction of the concrete queue operations *CEnq* and *CDeq*. The generic state variable *CS* becomes a tuple consisting of a shared state for the heap and the head and tail pointers and local states according to *CEnq* and *CDeq* for every process $i$.

From an abstract point of view, a queue is a finite sequence of elements offering an atomic operation that adds an element at the end of the sequence and an atomic operation that removes the first element from the sequence. These atomic operations are analogously extended as already presented in Subsection 5.1, Figure 6 for the stack. The extended abstract dequeue operation is explicitely given in Figure 8 b). The abstract state variable *AS* becomes a dynamic variable *Queue* of type list and generic operation *AOP* is the nondeterministic choice between the two extended abstract operations. In order to prove the refinement relation between *COP* and *AOP*, generic predicate *ABS* is defined in conformance with Figure 7.

The required local rely-guarantee conditions $R_i : cstate \times cstate$ for the refinement proof are similar to the stack case study. Each process requires other processes to maintain a valid representation according to *ABS* and to respect locality assumptions. As an example of the differences, predicate $R_{i,deq}$, the analogon of subproperty $R_{i,pop}$ from the stack, looks as follows.

$$\begin{array}{ll} & R_{i,deq}(cs_1, cs_2) \\ \leftrightarrow & \quad succdf_1(i) = succdf_2(i) \wedge hddf_1(i) = hddf_2(i) \wedge nxtdf_1(i) = nxtdf_2(i) \\ & \wedge (\quad hddf_1(i) \neq null \wedge hp_1[hddf_1(i)].nxt \neq null \\ & \quad \rightarrow \quad hp_1[hddf_1(i)].nxt = hp_2[hddf_2(i)].nxt \\ & \quad \wedge hp_1[hp_1[hddf_1(i)].nxt].val = hp_2[hp_2[hddf_2(i)].nxt].val) \end{array} \tag{13}$$

The environment of $i$ is not allowed to change local variables $succdf_1(i)$, $hddf_1(i)$ and $nxtdf_1(i)$. If snapshot $hddf_1(i)$'s next reference is not *null*, other processes leave this reference unchanged as well as the value of $hddf_1(i)$'s direct successor cell. This ensures that the successful CAS in line C13 shifts the head pointer correctly and it ensures also that abstract and concrete output values do not differ.

A property that is part of the invariant condition $Inv : cstate$ and which has no counterpart in the stack case study is the following.

$$\begin{array}{ll} & hddf(i) \neq \textsc{Null} \\ \rightarrow & \quad (hp[hddf(i)].nxt = \textsc{Null} \vee hp[hddf(i)].nxt \in hp) \\ & \wedge (hp[hddf(i)].nxt = \textsc{Null} \rightarrow hddf(i) = hdq) \end{array} \tag{14}$$

It states that $i$'s local snapshot $hddf(i)$ has a next pointer which is either null or pointing to a location in

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\mathbf{C8}, RI, s_2 \vdash \dots}}{\dots \vdash \mathbf{A8} \wedge A}\ (7)}{\dots, s_1, Hddf(i) = Head \vdash \dots}\ (5) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\mathbf{C4}, RI, s_3 \vdash \dots}}{\dots \vdash \mathbf{A3} \wedge A}\ (8)}{\dots, s_1, Hddf(i) \neq Head \vdash \dots}\ (6)}{\mathbf{C7},\ RI,\ s_1 \vdash \mathbf{A3} \wedge A,\ \mathbf{A8} \wedge A}\ (4)}{\mathbf{C6},\ RI,\ s_0, Hp[Hddf(i)].\mathrm{nxt} = \textsc{Null} \vdash skip; \mathbf{A3} \wedge A,\ \mathbf{A4} \wedge A}\ (3)}{\mathbf{C6},\ RI,\ s_0, Hp[Hddf(i)].\mathrm{nxt} = \textsc{Null} \vdash \mathbf{A3} \wedge A}\ (2) \quad \dots \neq \textsc{Null} \vdash \dots}{\mathbf{C6},\ RI,\ s_0 \vdash \mathbf{A3} \wedge A}\ (1)$$

$$\begin{aligned}
RI &:\equiv \Box\ (R_i(CS', CS'') \wedge Inv(CS) \wedge Inv(CS')) \\
A &:\equiv \Box\ abs(Queue, CS) \wedge abs(Queue', CS')) \\
s_0 &:\equiv \neg\ Succdf(i) \wedge Hddf(i) \neq \textsc{Null} \wedge abs(queue_0, CS) \\
s_1 &:\equiv \neg\ Succdf(i) \wedge Hddf(i) \neq \textsc{Null} \wedge Nxtdf(i) = \textsc{Null} \wedge abs(queue_1, CS) \\
s_2 &:\equiv \neg\ Succdf(i) \wedge Hddf(i) \neq \textsc{Null} \wedge Nxtdf(i) = \textsc{Null} \wedge abs(queue_2, CS) \\
s_3 &:\equiv \neg\ Succdf(i) \wedge Hddf(i) \neq \textsc{Null} \wedge Nxtdf(i) = \textsc{Null} \wedge abs(queue_3, CS)
\end{aligned}$$

**Fig. 10.** Proof Outline Dequeue Empty

the application heap. If this pointer is null, the snapshot is still accurate, i.e. it coincides with the global head pointer *hdq*.

### 6.3. Proof Outline

We outline the proof for the critical transition of *CDeq* executed by an arbitrary process $i$ in Figure 10. There **Ck** denotes the remaining program of *CDeq* starting from line Ck, e.g. **C6** denotes the sequence of instructions C6 to C10, followed by the while loop. Analogously **Ak** stands for the remaining code of *ADeq* from line Ak. Further abbreviations are introduced in the figure.

The conclusion of the proof tree shows the proof goal after symbolically executing lines C1 - C5 of *CDeq*, the program to execute is **C6**. Process $i$ has reached a state which satisfies predicate logic formula $s_0$. Assuming the rely guarantee and invariance conditions $RI$, it has to be shown that the sequence of values $absf(Head, Tail, Hp)$ is an execution of **A3** and preserves the abstraction $A$.[3]

The first step (1) of the proof does a case split on whether $Hddf(i)$'s next pointer is Null, to get the critical case in the first premise. The current queue ($queue_0$) is then indeed empty, since according to invariant property (14) from the previous subsection, the snapshot $Hddf(i)$ is still the head of the queue.

The next step (2) expands the star operator using the equivalence

$$[skip^*; \gamma] \equiv [skip;\ skip^*;\ \gamma] \vee [\gamma]$$

for an arbitrary program $\gamma$ (either another skip is executed or $\gamma$ starts immediately). This gives the two succedent formulas of the premise (a comma between formulas in the antecedent/succedent of a sequent stands for the conjunction/disjunction of these formulas).

The next step (3) symbolically executes the instruction at C6 which sets $Nxtdf(i)$ to Null and the skip transition of the first formula in the succedent is executed. The second formula in the succedent also executes a skip step, as the test of the if-statement at A4 is false. The remaining program in this case is just $skip^*; O := Lo$ (written **A8** in the goal). This gives a proof obligation according to the premise of (3) and a side goal which demands to prove that the last step has indeed preserved $A$. The proof of this side goal is trivial, as $s_0$ fulfills *abs* and none of the three formulas has changed the concrete resp. abstract state. Therefore, both abstract transitions can be pursued in constructing an abstract trace. This is not the case with linearization points that change the data representation, where only one choice of executing an abstract skip or executing the abstract operation will give a suitable abstract state. For these, one of the two resulting

[3] In the following we abstract from the substitution of the abstract state and output variables as described in Subsection 5.3

formulas will simplify to false and disappear. Delaying the decision, whether the linearization point has been executed, seems to be possible only when the abstract data structure is not modified. The case is common for operations that observe the data structure (e.g. the test for an element being in a list in the algorithm studied in [VHHS06]). Our proof approach can handle this case without any extra work, since it does not require to fix a unique value of a program counter for the new abstract state.

The proof continues with the new state $s_1$. This is the state after execution of C6 and after the environment has executed its rely step. In this state the case split whether the last step has been a linearization point can be made (step (4) of the proof). If the global head pointer $Head$ has not changed since taking the snapshot $Hddf(i)$, executing the transition in line C6 has been a linearization point. Formula $\mathbf{A3} \wedge A$ can be weakened from the succedent in step (5) by applying the weakening rule of the sequent calculus which permits to strengthen a sequent by dropping formulas from its succedent (or antecedent). Both sequents of this inference step are given in more detail.

$$\frac{N = N'' + 1 \,\mathbf{until}\, B, \mathit{Indhyp}(N), \mathbf{C7}, RI, s_1, \mathit{Hddf}(i) = \mathit{Head} \vdash \mathbf{prefix}(\mathbf{A8} \wedge A, B)}{N = N'' + 1 \,\mathbf{until}\, B, \mathit{Indhyp}(N), \mathbf{C7}, RI, s_1, \mathit{Hddf}(i) = \mathit{Head} \vdash \mathbf{prefix}(\mathbf{A3} \wedge A, B),\ \mathbf{prefix}(\mathbf{A8} \wedge A, B)}$$

According to Subsection 3.2 a counter $N$ and a boolean variable $B$ are used for induction. The induction hypothesis $\mathit{Indhyp}(N)$ is the universal closure (except $N$) of the following formula.

$$\forall\, N_0 < N.\ N_0 = N_0'' + 1 \,\mathbf{until}\, B \wedge \mathbf{C4} \wedge RI \wedge \neg\, \mathit{Succdf}(i) \rightarrow \mathbf{prefix}(\mathbf{A3} \wedge A, B) \tag{15}$$

During symbolic execution the induction hypothesis is defined just before the loop-body is entered (**C4**) and since $N$ is decremented in each step, $\mathit{Indhyp}(N)$ can be applied if after at least one execution step, a configuration is reached which satisfies the preconditions of the implication in (15). This is the case if executing the loop-body leads back to line C4 (see below). Since in the current state formulas $\mathit{Hddf}(i) = \mathit{Head}$ and $\mathit{Nxtdf}(i) = \textsc{Null}$ hold, the following symbolic execution steps (7), ... lead to eventually exiting the loop-body and returning output value *empty*. In this case of linearization the induction hypothesis is not required.

If however the snapshot is deprecated, the executed concrete transition has not been a linearization point and formula $\mathbf{A8} \wedge A$ can be weakened in step (6). In this case of non-linearization the next symbolic execution step (8) jumps back to the start of the loop body **C4**:

$$\frac{N = N'' + 1 \,\mathbf{until}\, B, \mathit{Indhyp}(N), \mathbf{C4}, RI, s_3 \vdash \mathbf{prefix}(\mathbf{A3} \wedge A, B)}{N = N'' + 1 \,\mathbf{until}\, B, \mathit{Indhyp}(N), \mathbf{C7}, RI, s_1, \mathit{Hddf}(i) \neq \mathit{Head} \vdash \mathbf{prefix}(\mathbf{A3} \wedge A, B)}$$

The new state fulfills $s_3$ which implies $\neg\, \mathit{Succdf}(i)$ and the resulting sequent can be closed with the induction hypothesis (15) in step (9).

## 7. Related Work

Our approach is based on an expressive temporal logic, which is built into the theorem prover. In this sense it is similar to the STEP prover [BBO+99] or to PlusCal [Lam06]. All these approaches require to encode programs to a normal form of transition systems for deduction. Although this is not a drawback for automated proofs using model checking, we found it very unintuitive to reason about program counters in interactive verification. Therefore we avoid such an encoding.

An alternative to using temporal logic and rely-guarantee reasoning directly, is to use an encoding into higher-order logic (e.g. [PA03], [Mer95], [Kal95] or [Pre03]). This has the advantage that soundness of the logic can be reduced to the soundness of higher-order logic, but it requires to encode a lot of the semantics (we are not aware of encodings that support all of local variables, recursion, blocking and interleaving).

Verification of lock-free algorithms is currently an active research topic. Various algorithms have been proved correct, e.g. algorithms working on a global queue [DGLM04, AC05], a lazy caching algorithm [Hes06] or a concurrent garbage collection [GGH07].

The stack and queue algorithms considered here were taken from Groves et. al. [CDG05, DGLM04], who have given a correctness proof using IO automata and the theorem prover PVS. Their work adds modification counts to avoid the ABA problem. In contrast to this formal proof our proof is not monolithic, and does not require to encode programs as automata using program counters. Another difference is, that their work needs a intermediate specification and backward simulation for verification of the queue algorithm.

Recent work by the same authors [GC07, GC09] discusses incremental development of the algorithm

using refinement calculus and programs very similar to ours. For the stack algorithm we have only studied the core algorithm, while their work discusses extending the algorithm with elimination arrays from [HSY04]. The resulting steps are rather intuitive for explaining the ideas and possible variations. Again this work also discusses various extensions and variations of the algorithm. The refinement calculus used is quite close to parts of the logic used in KIV [RSSB98] (in particular to Dynamic Logic [Har84] for sequential programs). Therefore, we tried to imitate some of the steps, but we found that this is not really possible. The basic idea underlying the paper of commuting statements that assign to disjoint variables is almost never applicable, since most assignments work on one variable: the global heap. Such assignments commute only if it can be proved that the locations they access are disjoint. Indeed most of the complexity of our assumptions is to answer the question, why processes cannot modify or access certain locations. Answers to these questions are only given informally in [GC07, GC09].

In [VHHS06], Vafeiadis et. al. describe a rely-guarantee approach that is applied informally on an implementation of sets using fine-grained locking. [CPV07, VP07, Vaf09] mechanizes the approach by providing tool support that checks the proof obligations (correctness of the proof obligations is justified using a rely-guarantee theorem that is proved on the semantic level). In contrast to our approach, the approach mixes the abstract and concrete layer, by calling the abstract operation at the linearization point within the concrete code. We have not used this idea, since it is incompatible with the idea of developing concrete programs incrementally from abstract specifications. Nevertheless the technique allows to use a standard rely-guarantee theorem for a single program and is beneficial for automation. Verification of proof obligations is fully automatic on a number of examples. An impressive range of specialized automation techniques is used: separation logic [Rey02] is used to reason over pointer structures, the distinction between local (i.e. modifiable only by one process) and global references (our *disj* predicate) is encoded as boxed and unboxed formulas. A variant of the abstraction and invariant generation technique proposed by [DOY06] is used to deal with loops. More analysis is necessary, how these automation techniques could be integrated with our generic temporal logic approach that uses arbitrary abstract data types.

Fully automatic approaches based on static analysis are given by Amit et. al. in [ARR$^+$07] and by Berdine et. al. in [BLAM$^+$08]. These work with a simplified problem: first, CSeq calls in CSpawn are replaced bt using COP (arguing, that two sequential calls of COP by one process could equivalently be done by two different processes, when the second process starts after the first has finished). Second, they argue that a standard data refinement could be used to refine AOP by an atomically executed COP (which is outside their interest). They then formally verify only the problem of refining a number of atomically executed COP operations by an interleaved execution of the same operations. This is done by comparing the shapes of the pointer structures during the executions and computing finite approximations of the shape differences. Their approach also abstracts the arbitrary number of processes into classes which currently execute the same instruction (since the number of instructions is finite this abstraction is finite too). The approach is specialized to pointer structures and their shapes of the resulting graphs, while our approach gives a generic refinement theorem. Nevertheless most lockfree algorithms use pointer structures, and recognizing the shape of data structures automatically could lead to siginificant improvements in the automation of our approach.

The second author of this paper has also contributed to [DSW07] and [DSW08], where a data refinement theory for lock-free algorithms is developed. While the goal, to modularize the linearizability proof is similar, the technique used is rather different: control structure of the operations is encoded using CSP in [DSW07] and using program counters in [DSW08]. Single steps of the algorithm are given as Z operations. Interleaving of processes is done explicitly using promotion, while we use the interleaving operator of temporal logic. The stack algorithm is also used as a running example. The resulting proof obligations resemble the Owicki-Gries technique [OG76] for proving program correctness. They have the advantage that they can be verified using predicate logic only. Also, they exploit some more of the symmetry between processes. On the other hand many intermediate assertions must be defined that the approach given here computes automatically by symbolic execution. [DSW08] explicitly proves that the example satisfies the original criterion of linearizability that was defined in Herlihy and Wing's original paper [HW90], while linearizability is only implicitly implied by this and all other related work we are aware of.

In current work we try to combine the advantages of both techniques by integrating the formal definition of linearizability and the ideas for exploiting symmetry with the rely-guarantee conditions.

## 8. Conclusion

In this paper we have developed a proof technique based on rely-guarantee reasoning for verifying refinements of abstract data types to interleaved algorithms.

Compared to other approaches, the main contribution of this work is, that we are able to not only prove the proof obligation of a rely-guarantee technique, but also to construct and prove the compositional theorem itself in the same temporal logic framework. The standard example of Treiber's stack and a more complicated, practical implementation of a lock-free queue were successfully verified with this technique. Both, data refinement and decomposition using rely-guarantee reasoning were expressed using the temporal logic available in the theorem prover KIV.

Tool support delivers proofs of higher quality compared to the verification of programs with pen and paper. The interactive verifier KIV allows us to directly verify parallel programs in a rich programming language using the intuitive proof principle of symbolic execution. An additional translation to a special normal form (as e.g. in TLA [Lam94]) using explicit program counters is not necessary.

For further work we have started to consider more complex examples of lock-free algorithms and to verify liveness in addition to linearizability. Another direction of research is the integration of techniques for improving automation, without giving up the incremental development principle based on refinement.

*Acknowledgements*

We would like to thank the anonymous reviewers for their critical and constructive remarks during the writing of this paper.

## References

[AC05] J.-R. Abrial and D. Cansell. Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). *Journal of Universal Computer Science*, 11(5):744–770, 2005.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 1995.

[ARR+07] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3), 1987.

[Bal05] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction*. PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[BBO+99] Nikolaj S. Bjørner, Anca Browne, Michael A. Col On, Bernd Finkbeiner, Henny B. Sipma, and Tomás Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. In *Formal Methods in System Design*, volume 16, page 2000, 1999.

[BBRS08] M. Balser, S. Bäumler, W. Reif, and G. Schellhorn. Interactive Verification of Concurrent Systems using Symbolic Execution. In *Proceedings of 7th International Workshop of Implementation of Logics (IWIL 08)*, 2008.

[BLAM+08] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV'08*. Springer, 2008.

[BNBR08] S. Bäumler, F. Nafz, M. Balser, and W. Reif. Compositional proofs with symbolic execution. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *Ceur Workshop Proceedings*, 2008.

[BS03] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[Bur74] R. M. Burstall. Program proving as hand simulation with a little induction. *Information processing 74*, pages 309–312, 1974.

[CC96] A. Cau and P. Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33(2):153–176, 1996.

[CDG05] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *ENTCS*, 137:93–110, 2005.

[CGP00] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[CMZ02] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic*. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. `www.cms.dmu.ac.uk/~cau/itlhomepage`.

[CPV07] C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, pages 233–248, 2007.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114, 2004.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[DOY06] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920, pages 287–302. Springer, 2006.

[dRdBH+01] 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*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[DSW07] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In *IFM*, pages 195–214, 2007.

[DSW08] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanising a correctness proof for a lock-free concurrent stack. In *Prooceedings of FMOODS 2008, Oslo*, volume 5051 of *LNCS*, pages 78–95, 2008.

[GC07] L. Groves and R. Colvin. Derivation of a scalable lock-free stack algorithm. *Electron. Notes Theor. Comput. Sci.*, 187:55–74, 2007.

[GC09] L. Groves and R. Colvin. Trace-based derivation of a scalable lock-free stack algorithm. *Formal Aspects of Computing (FAC)*, 21(1–2):187–223, 2009.

[GGH07] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. *Sci. Comput. Program.*, 64(3):341–374, 2007.

[Gur95] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford Univ. Press, 1995.

[Har84] David Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[Hes06] W. H. Hesselink. Refinement verification of the lazy caching algorithm. *Acta Inf.*, 43(3):195–222, 2006.

[HSY04] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[JT96] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theor. Comput. Sci.*, 167(1-2):47–72, 1996.

[Kal95] S. Kalvala. A formulation of TLA in Isabelle. http://www.research.digital.com/SRC/personal/ lamport/ tla/tla.html, June 1995.

[KIV] Web presentation of the composition theorem and the lock-free stack and queue case study in KIV. URL: `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam06] L. Lamport. The +CAL algorithm language. Technical report, Microsoft, 2006.

[MC81] Jayadev Misra and K. Mani Chandi. Proofs of networks of processes. *IEEE Transactions of Software Engineering*, 1981.

[Mer95] S. Merz. Mechanizing TLA in Isabelle. In Robert Rodošek, editor, *Workshop on Verification in New Orientations*, pages 54–74, Maribor, July 1995. Univ. of Maribor.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, 1986.

[MS96] 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*, pages 267–275, 1996.

[OG76] S. S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.*, 6:319–340, 1976.

[PA03] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL verification system. In *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.

[Pre03] L. Prensa Nieto. The rely-guarantee method in Isabelle /HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

[Tre86] R. K. Treiber. System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[Vaf09] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proceedings VMCAI 2009*, volume 5403 of *LNCS*. Springer, 2009.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[VP07] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

*Received 10 March 2009*
*Revised 30 June 2009 and 9 September 2009*
*Accepted 14 September 2009 by Egon Börger and Michael Butler*