Local Rely-Guarantee Conditions for Linearizability and Lock-Freedom

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg
{tofan,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. Rely-guarantee reasoning specifications typically consider all components of a concurrent system. For the important case where components operate on a shared data object, we derive a local instance of rely-guarantee reasoning, which permits specifications to examine a single pair of representative components only. Based on this instance, we define local proof obligations for linearizability and lock-freedom, which we then apply to a non-blocking concurrent stack with explicit memory reuse. Both the derivation of this local instance and its application are mechanized in the KIV interactive theorem prover.

Keywords: Verification, Temporal Logic, Rely-Guarantee, ABA-problem, Linearizability, Lock-Freedom

1 Introduction

The rely-guarantee method [1] deals with the challenges that arise when reasoning about concurrent systems with shared resources. It provides a compositional treatment of interference between system components, i.e., to analyze properties of the overall system, each component can be examined separately based on its specification of expected environment behavior.

Rely-guarantee reasoning proves to be a valuable technique to verify non-blocking (here lock-free) implementations of concurrent data structures, such as stacks, queues or sets. Such algorithms play an important role in multi-core systems and are also contained in concurrency packages of modern, high-level object-oriented programming languages (e.g., java.util.concurrent). They try to better utilize the capacity of multi-cores by avoiding locking and thus increasing the potential of operations to execute in parallel. Their main correctness property *linearizability* [2] ensures that each interleaved execution of concrete data structure operations corresponds to an abstract, sequential execution that preserves the (real-time) order of non-interleaved concrete calls. Their main progress property *lock-freedom* [3] guarantees that in each interleaved execution, always eventually one of the running operations terminates. Lock-free implementations avoid deadlocks, livelocks, convoying and priority inversion. Thus, they are also heavily used in real-time systems (e.g., for real-time garbage collection).

The verification framework is based on interval temporal logic [4] and symbolic execution [5] and is implemented in the interactive theorem prover KIV [6]. It permits to verify the soundness of a typical form of rely-guarantee reasoning, as well as to (mechanically) derive more specific instances of it and to prove decomposition theorems for system-wide correctness or progress properties, such as linearizability or lock-freedom. This paper describes such an instance and illustrates its use.

Our earlier embedding of rely-guarantee reasoning in interval temporal logic described in [7] follows the global approach of [1]: specifications consider the overall program state, consisting of the local states of *all* components and the shared state. Here we reduce this former embedding to allow for specifying properties of concurrent systems – that are unbound in their number of components – in terms of two representative components. This valuable reduction leads to both simpler specifications and proofs, e.g., in the frequent case of verifying concurrent data structures where all processes have similar behaviors. We define local proof obligations for linearizability and lock-freedom based on this instance and show its application by verifying the major safety and liveness aspects of a lock-free stack that recycles memory from a shared pool of reusable memory locations.

To the best of our knowledge, this is the first *mechanized derivation* of a local instance of rely-guarantee reasoning. The instance permits local proofs of both linearizability and lock-freedom. Furthermore, we describe the first mechanized verification of the main aspects of a well-known lock-free stack [8] with explicit memory reuse. Although different versions of the stack have been verified before, these have mainly focused on linearizability and all except one informal proof [9] implicitly assume garbage collection, which significantly simplifies the proofs. A complete presentation of the verification of the theory and its application to several lock-free data structures, is available online [10].

The rest of this paper is structured as follows: Section 2 introduces the stack case study. Section 3 gives a short introduction to the temporal logic framework. In Section 4 we briefly describe the embedding of global rely-guarantee reasoning and derive its local instance and local proof obligations for linearizability and lock-freedom. Section 5 shows the application of this instance to the case study. Section 6 presents related work and a comparison. Finally, Section 7 concludes with a short summary and an overview of our current and future work.

2 The Lock-Free Stack

Lock-free algorithms typically apply atomic synchronization primitives such as CAS (Compare-And-Swap) instead of locks.

CAS($Old, New; SV, Succ$) {
  if* $SV = Old$ **then** $\{SV := New,\ Succ := true\}$ **else** $Succ := false\}$

CAS compares a shared value SV with an older local copy of it Old (called snapshot). If these values are equal SV is updated to a new value New and true

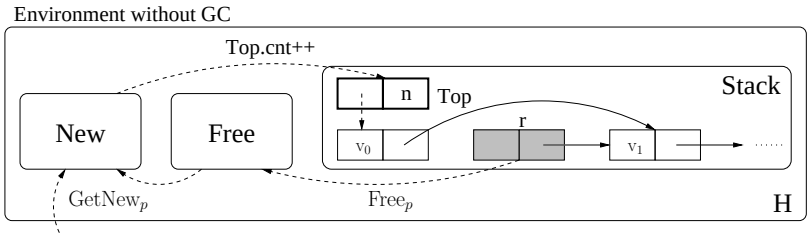

Fig. 1. Data-stack that reuses memory locations from a shared pool (Free).

is returned; otherwise false is returned. Throughout this work we use formal KIV specifications to describe programs. In the specification of CAS, the semicolon separates input from in-output parameters; the comma indicates parallel assignments and in **if*** evaluating the if-condition requires no extra step.

Explicit Memory Reuse Lock-free data structures are often used in programming environments without implicit garbage collection (GC). There, memory locations that are removed from the data structure should be reclaimed in some way to avoid memory leaks. However, removed locations can not be simply deallocated (e.g., using a *free* library call in C/C++), since they are typically still used by concurrent processes. To solve the problem an explicit garbage collection scheme is added to the main algorithm. This paper considers introducing a shared pool of reusable locations as originally proposed by Treiber [8]. (A more advanced solution are hazard pointers [11], see Section 6.) Treiber notes that the possible concurrent reuse of a location can lead to data structure corruption at runtime, when a location is concurrently reinserted in the data structure with a modified content and these intermediate modifications are not detected by CAS. This is a well-known and fundamental problem of CAS-based algorithms, called the ABA-problem. Treiber's solution attaches a modification counter to ABA-prone shared resources so that CAS detects their possible concurrent access.

This work considers a lock-free data-stack that recycles memory from a shared pool of reusable locations (Free), as shown in Figure 1. The stack stores arbitrary data values v_i in a singly linked list of cells (pairs of values and locations having .val and .nxt selector functions) which resides in the application's memory heap H[1]. A shared variable Top marks the top cell of the stack; it is a pair of a reference (location) and a (natural number) modification counter with selector functions .ref and .cnt respectively. A process p which executes a push tries to reuse locations from Free and allocates new ones only if Free is empty (GetNew$_p$). Whenever p pops a location r from the stack, it subsequently adds r to Free (Free$_p$). To detect the concurrent reuse of a location, the modification counter Top.cnt is always incremented atomically with the insertion of a new cell in the stack (Top.cnt++).

[1] H is a partial function from references $r : ref$ (with $null \in ref$) to cells with standard operations, e.g., $r \in H$ tests if r is allocated, $H[r]$ is lookup and $H + r$ is allocation.

```
U1 Push(In; UNew, USucc, Top, Free, H) {
U2   GetNew(In; UNew, USucc, Free, H);
U3   let UTop = ? in {
U4     while ¬ USucc do {
U5       UTop := Top;
U6       H[UNew].nxt := UTop.ref;
U7       CAS(UTop, (UNew × UTop.cnt + 1); Top, USucc)}}}

GetNew(In; UNew, USucc, Free, H) {
  choose r with (r ≠ null ∧ (if* Free = ∅ then r ∉ H else r ∈ Free)) in {
     Free := Free \ {r}, H := H + r,
     H[r].val := In, UNew := r, USucc := false}}

O1 Pop(; OTop, OSucc, Top, Free, H, Out) {
O2   let Lo = empty, ONxt = ? in {
       while ¬ OSucc do {
O3       OTop := Top;
O4       if OTop.ref = null then {
O5         OSucc := true;
         } else {
O6         ONxt := H[OTop.ref].nxt;
O7         CAS(OTop, (ONxt × OTop.cnt); Top, OSucc)}}
O8     if OTop.ref ≠ null then {
O9       Lo := H[OTop.ref].val′;
O10      Free := Free ∪ {OTop.ref}, OSucc := false}
O11    Out := Lo, OSucc := false}}
```

Fig. 2. Implementation of push and pop.

The Implementation Figure 2 shows the implementation of the stack which is taken from [9] and attributed to [8]. Variables *UNew*, *USucc*, *OTop* and *OSucc* are local variables of "pUsh" resp. "pOp". They are defined as in-output parameters (instead of using **let**) to allow us to reason about them. Whenever a process executes a push, it first allocates and initializes a new cell *UNew* in one step (*GetNew*). Then it repeatedly tries to CAS the shared top pointer to this new cell. (A "?" denotes an arbitrary value.) A pop process reads the shared top in line O3 (if the snapshot's pointer is null, the special value *empty* is returned) and locally stores its next reference which becomes the target of the subsequent CAS. If it succeeds, the top cell is removed from the stack and then added to the shared reference set *Free* (freed). Variable *OSucc* (initially *false*) is used in the verification to characterize removed locations, i.e., locations that have been removed from the stack but not yet freed.

Without a modification counter, an ABA-problem could occur as follows: suppose that a pop-process p takes a snapshot of the top pointer when the stack consists of exactly one cell at location A and the free-list is empty. Process p is delayed after setting its local next reference *ONxt* to null in line O6 for another process. This other process removes A from the stack without yet freeing it.

Subsequently, a further process q executes a successful push, thereby allocating a new location B. Then A is freed and q executes a successful push of A. If now p is rescheduled, its CAS operation in line O7 would erroneously succeed, violating the semantics of pop by deleting the entire stack.

Note that the (double-word) CAS in line O7 atomically compares both a location and a counter. It fails if the snapshot location $OTop$.ref has been concurrently removed from the stack and not reinserted, since it is then not equal to Top.ref. CAS also fails if $OTop$.ref is concurrently reinserted in the stack, since the snapshot's modification counter $OTop$.cnt then does not coincide with Top.cnt ($OTop$.cnt $<$ Top.cnt). Thus, the ABA-problem is avoided. (In Section 5 we intuitively formalize and verify this non-trivial synchronization scheme.)

3 The Verification Framework

This section gives a brief overview of the underlying verification framework. We refer the interested reader to [12, 13] for further details on the syntax and semantics of the logic.

Interval Temporal Logic ITL [4] in KIV is based on algebras, to define the semantics of the signature, [2] and intervals (executions), which are finite or infinite sequences of states. A state maps variables to values. Different from standard ITL, the logic explicitly includes the behavior of a program's environment in each step (similar to reactive sequences in [14]): in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first transition from state $I(0)$ to the primed state $I'(0)$ is a program transition, whereas the next transition from state $I'(0)$ to $I(1)$ is a transition of a program's environment. In this manner program and environment transitions alternate. A variable V is evaluated over $I(0)$, whereas its primed resp. double primed version V' resp. V'' is evaluated over $I'(0)$ and $I(1)$ respectively. (For an empty interval [I(0)], both are evaluated over I(0).) E.g., formula $V \neq V'$ denotes that variable V is changed in the first program transition, whereas $V' = V''$ states that V is not changed in the first environment transition. The last state of an interval is characterized by formula **last**.

The logic provides standard temporal operators to describe interval properties: the (weak) next operator $\bullet\, \varphi$ holds in an interval I iff I is either empty, or φ holds in I's postfix interval $[I(1), \ldots]$. $\varphi\, \mathbf{until}\, \psi$ holds in I iff ψ holds in some $[I(n), \ldots]$ and φ holds in $[I(m), \ldots]$ for each $m < n$. Further standard operators are introduced as abbreviations, e.g., $\Diamond\, \varphi \equiv \mathit{true}\, \mathbf{until}\, \varphi$, or $\Box\, \varphi \equiv \neg \Diamond \neg\, \varphi$. In our embedding of rely-guarantee reasoning, temporal formulas of the form $R(V', V'') \xrightarrow{+} G(V, V')$ are of particular interest (cf. Section 4). Predicates G and R specify guarantee resp. rely conditions and the sustains operator $\xrightarrow{+}$ ensures that G is maintained by a program transition if previous environment transitions have preserved R, as shown below. Thus, $R(V', V'') \xrightarrow{+} G(V, V') \equiv \neg\, (R(V', V'')\, \mathbf{until} \neg\, G(V, V'))$, as shown below. Note that G must always hold for the first program transition.

[2] We use higher-order signatures and algebras, see [13]. An example of a higher-order variable is the local state function LSf from Section 4.2.

$$[I(0) \xrightarrow[\subseteq \mathrm{G}]{} I'(0) \xrightarrow[\subseteq \mathrm{R}]{} I(1) \longrightarrow \cdots \longrightarrow I'(n-1) \xrightarrow[\subseteq \mathrm{R}]{} I(n) \xrightarrow[\Rightarrow\ \subseteq \mathrm{G}]{} I'(n) \ldots]$$

The formal programming language in KIV provides the common sequential constructs. Moreover, it provides a construct for weak-fair interleaving ($\|$) and one for non-fair interleaving ($\|_{\mathrm{nf}}$). Programs α and formulas can be mixed, since they both evaluate to true or false in an interval. In particular, α evaluates to true in I iff I is an execution of α with arbitrary environment steps.

The verification framework is based on the sequent calculus. A sequent is an assertion of the form $\Gamma \vdash \Delta$ (where Γ and Δ are lists of formulas), which states that the conjunction of all formulas in antecedent Γ implies the disjunction of all formulas in succedent Δ. Sequents are implicitly universally closed. A sequent (proof obligation) about concurrent programs α typically has the form $\alpha, E, F \vdash \varphi$ where α executes in an environment constrained by temporal formula E, predicate logic formula F describes the current state and φ is the property of interest. As a simple example, consider the following sequent.

$$(M := M + 1;\ \beta),\ M = 1 \ \vdash\ M' = M'' \xrightarrow{+} M' > M \tag{1}$$

The executed program is $(M := M+1;\ \beta)$ where β is a program and environment behavior is unrestricted ($E = \mathit{true}$ omitted). The current state maps counter M to 1 and it has to be shown that the program always increases M if previous environment transitions have not changed it ($M' = M'' \xrightarrow{+} M' > M$).

Symbolic Execution Sequents that contain temporal assertions are verified by symbolically stepping forward to the next states of an interval, calculating strongest postconditions for each program transition, which are then possibly weakened according to assumptions for the following environment transition. Restricting environment transitions to never change any program variables yields the sequential setting. Thus, the calculus is rather similar to classic symbolic execution of sequential programs [5], but in a concurrent setting.

A step is executed in two implicit phases which concern programs as well as formulas. In the first phase, information about the first program and environment transition is separated from information about the rest of an interval by applying unwinding rules. A program is unwound by calculating the effect of its first statement; $\xrightarrow{+}$ is unwound according to the following rule:

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

Applying this rule to the succedent of (1) yields $M' > M \wedge (M' = M'' \rightarrow \bullet\ (M' = M'' \xrightarrow{+} M' > M))$. That is, we must prove that M is increased in the first program transition ($M' > M$) as a first subgoal. If the following environment transition leaves M unchanged ($M' = M''$), then the sustains formula must further hold in the rest of the interval ($\bullet\ (M' = M'' \xrightarrow{+} M' > M)$). The second phase of a symbolic execution step "moves" to the rest of an interval by eliminating leading next operators. This leads to the following further subgoal when proving (1):

$$\beta,\ M = 2 \ \vdash\ M' = M'' \xrightarrow{+} M' > M$$

Induction Well-founded induction is used to deal with loops. For infinite intervals, a term for well-founded induction can be derived from a known liveness property $\diamond\ \varphi$ as the number N of steps until φ holds.

$$\diamond\ \varphi\ \leftrightarrow\ \exists\ N.\ (N = N'' + 1)\ \textbf{until}\ \varphi$$

The fresh variable N is decremented in each step until φ becomes true. (Note that $N = N'' + 1$ is equivalent to $N'' = N - 1 \wedge N > 0$.)

The proof of a sustains formula on an infinite interval I can be carried out by induction over the length of an arbitrary finite I-prefix.

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

The fresh boolean B characterizes the length of the prefix, which ends as soon as B becomes true for the first time. Again, the number of steps until B holds is used for well-founded induction.

4 Deriving Local Rely-Guarantee Conditions for Linearizability and Lock-Freedom

Rely-guarantee reasoning basically defines proof obligations for individual components of a concurrent system instead of reasoning about their interleaved execution. This section briefly describes our concurrent system model and outlines our embedding of global rely-guarantee reasoning (cf. [7]). Then it derives its local instance – which is simpler to use when verifying concrete systems with similar components – in detail and briefly defines local proof obligations for linearizability and lock-freedom.

4.1 The Concurrent System Model and Global Rely-Guarantee Reasoning

The Concurrent System Model As shown in Figure 3, our generic concurrent system SPAWN recursively spawns $n+1$ components ($n : \mathbb{N}$) to execute in parallel. Operation SEQ defines the possible sequential behaviors of each component $p : \mathbb{N}$. Either p instantly terminates or it executes finitely or infinitely often – as denoted by the star operator ***** – a generic interface procedure COP or *skip* which models steps that are unrelated to COP. [3] The unspecified procedure COP models arbitrary operations that p can execute on the overall concurrent system state $CS : cstate$. Functions $Inf : \mathbb{N} \rightarrow input$ and $Outf : \mathbb{N} \rightarrow output$ are used to insert or return values $Inf(p)$ and $Outf(p)$ respectively.

Global Rely-Guarantee Reasoning To avoid tedious reasoning about interleaved executions of SPAWN, we have embedded rely-guarantee reasoning in

[3] The auxiliary function $Actf : \mathbb{N} \rightarrow bool$ distinguishes whether a component executes COP (i.e., is active) or not, since the logic does not use program counters. This is mainly relevant for the decomposition proof of lock-freedom.

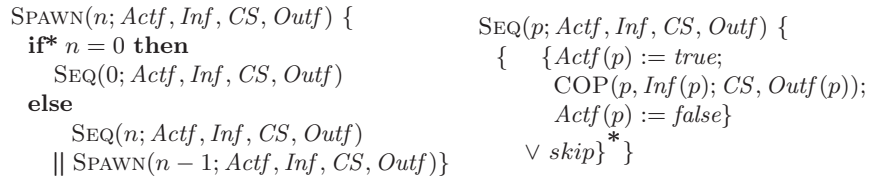

Fig. 3. The concurrent system model.

the logical framework. In this embedding specifications use the overall concurrent system state CS and thus we call it "global". The basic idea is to abstract away from environment behavior using rely conditions $R_{ext} \subseteq \mathbb{N} \times cstate \times cstate$ for each component p and to guarantee a certain behavior towards p's environment according to guarantee conditions $G_{ext} \subseteq \mathbb{N} \times cstate \times cstate$. Our central rely-guarantee proof obligation for an individual component p then claims that in p's execution of $\mathrm{COP}(p, \dots)$, each program transition satisfies $G_{ext}(p, \dots)$ if the preceding environment transitions have preserved $R_{ext}(p, \dots)$.

$$\mathrm{COP}(p, Inf(p); CS, Outf(p)) \ \vdash R_{ext}(p, CS', CS'') \xrightarrow{+} G_{ext}(p, CS, CS') \tag{2}$$

We introduce further subpredicates to structure G_{ext} and R_{ext} into three categories: step invariant guarantee and rely conditions $G, R \subseteq \mathbb{N} \times cstate \times cstate$, state invariant conditions $Inv \subseteq cstate$ and local idle state conditions $Idle \subseteq \mathbb{N} \times cstate$ which hold before and after each finite execution of COP. The full version of (2) which takes into account these structural predicates simply is:

$$\begin{aligned} &\mathrm{COP}(p, Inf(p); CS, Outf(p)),\ Inv(CS),\ Idle(p, CS) \vdash R_{ext} \xrightarrow{+} G_{ext} \\ &\text{where } G_{ext} :\leftrightarrow \quad G(p, CS, CS') \wedge (Inv(CS) \rightarrow Inv(CS')) \\ &\qquad\qquad \wedge (\mathbf{last} \rightarrow Idle(p, CS)) \wedge \forall\, q \neq p.\ Idle(q, CS) \leftrightarrow Idle(q, CS') \\ &\text{and } R_{ext} :\leftrightarrow \quad R(p, CS', CS'') \wedge (Inv(CS') \rightarrow Inv(CS'')) \\ &\qquad\qquad \wedge (Idle(p, CS') \leftrightarrow Idle(p, CS'')) \end{aligned} \tag{3}$$

Program steps in $\mathrm{COP}(p, \dots)$ executions maintain $G(p, \dots)$, Inv and establish $Idle(p, \dots)$ in their last state (and do not change the idle state assumptions of other components q), as long as environment transitions maintain $R(p, \dots)$, Inv and $Idle(p, \dots)$ respectively. This embedding makes two improvements over our previous embedding [7]. First, the invariant is now decoupled from R and G to avoid unnecessarily strong rely resp. guarantee conditions. Second, we have introduced predicate $Idle$ to express local, idle state conditions.

Proving that these predicates hold indeed in every execution of SPAWN can be decomposed to basically showing (3) for an arbitrary component, according to the following theorem (cf. [7, 10] for technical details).

Theorem 1 (Global Rely-Guarantee Reasoning).
If (3) holds for an arbitrary overall system state CS and some transitive R, reflexive

G with $G(p, CS, CS') \rightarrow R(q, CS, CS')$, $\forall\ q \neq p$, *and predicates* $Inv, Idle$, *then:*

$$\textsc{Spawn},\ \Box\ R_{\textsc{Spawn}},\ Init_{\textsc{Spawn}} \vdash \Box\ ((\exists\ p.\ G(p, \dots)) \wedge \varphi_{Inv} \wedge \varphi_{Idle})$$

Spawn starts in an initial state satisfying $Init_{\textsc{Spawn}}$, which must imply *Inv* and *Idle* for all components (these are initially inactive). The *system*'s environment behavior is restricted by rely $R_{\textsc{Spawn}}$, which is the identity relation over the in-output parameters of Spawn. Then each system step of a component p always satisfies $G(p, \dots)$, invariant *Inv* holds in each state according to φ_{Inv} and any component is idle before and after it executes COP, according to φ_{Idle}.

4.2 Deriving Process-Local Rely-Guarantee Reasoning

Theorem 1 can be applied in scenarios where each component exhibits a different behavior (e.g., the producer-channel-consumer described in [12] where $n = 2$ and $\mathrm{COP}(0, \dots)$ is the producer, $\mathrm{COP}(1, \dots)$ the channel and $\mathrm{COP}(2, \dots)$ the consumer) since specifications account for the whole system state *CS*, including all local states. However, this expressiveness is often not required when components have similar behaviors, in particular when all components execute the operations of a concurrent data type. As an example, consider the global specification of the following simple invariant of the stack from Section 2 where components are concurrent processes that execute push or pop: pointers to new cells – which are not yet inserted in the stack – are disjoint during concurrent push operations.

$$\forall\ p \neq q.\ \neg\ USuccf(p) \wedge \neg\ USuccf(q) \rightarrow UNewf(p) \neq UNewf(q)$$

Global specifications require variable functions (e.g., $UNewf : \mathbb{N} \rightarrow ref$ for variable *UNew* in push) and quantification over all identifiers p, q. Thus they are less succinct and harder to read. Moreover, proofs that use such specifications are harder to automate, since finding right quantifier instantiations often fails.

However, the frequent case of concurrent data type implementations permits local specifications that consider say two representative components p resp. q with local states $LS : lstate$ resp. $LSQ : lstate$. The encoding of the aforementioned invariant then simply is:

$$\neg\ USucc \wedge \neg\ USuccq \rightarrow UNew \neq UNewq \tag{4}$$

From Global to Local Rely-Guarantee Specifications The reduction to local specifications is based on splitting *CS* into its local and shared parts $LSf \times S$ where $LSf : \mathbb{N} \rightarrow lstate$ maps each component to its local state and $S : sstate$ is the shared state. Each component p now executes the same procedure $\mathrm{LCOP}(Inf(p); LSf(p), S, Outf(p))$. In the stack case study, LCOP is the nondeterministic choice between one of the operations that each process can execute.

$$\mathrm{LCOP}(In; LS, S, Out)\ \{Push(In; LS, S) \vee Pop(;\ LS, S, Out)\}$$

The shared state S of the stack consists of the shared variables *Top*, *Free*, *H* for the top-of-stack pointer, the free-set and the application's heap, whereas the local state *LS* is the tuple of the local variables *UNew*, *USucc*, *OTop* and *OSucc*.

Furthermore, our local rely-guarantee embedding introduces a new invariant predicate *LDisj* to encode disjointness properties, such as (4), between the two local states. [4] The local counterpart of (3) now is:

$$\begin{aligned}
&\mathrm{LCOP}(In; LS, S, Out),\ LIID(LS, LSQ, S),\ LIdle(LS) \vdash LR_{ext} \xrightarrow{+} LG_{ext} \\
&\text{where } LG_{ext} :\leftrightarrow \quad LG(LS, LSQ, S, LS', S') \wedge (\mathbf{last} \rightarrow LIdle(LS)) \\
&\qquad\qquad \wedge (LIID(LS, LSQ, S) \rightarrow LIID(LS', LSQ', S')) \\
&\text{and } LR_{ext} :\leftrightarrow \quad LS' = LS'' \wedge LR(LS', S', S'') \\
&\qquad\qquad \wedge (LIID(LS', LSQ', S') \rightarrow LIID(LS'', LSQ'', S'')) \\
&\text{and } LIID(LS, LSQ, S) :\leftrightarrow LInv(LS, S) \wedge LInv(LSQ, S) \wedge LDisj(LS, LSQ)
\end{aligned} \tag{5}$$

Similar to (3), LCOP-steps must maintain the local guarantee conditions *LG* and the local state invariants *LIID*, plus, they must establish the local idle state *LIdle*, as long as environment transitions do not modify *LS* and they maintain the local rely *LR* and *LIID* respectively. A more detailed description of the local structural predicates is given in the following; their instantiation in the stack case study is shown in detail in Section 5.

The Local Structural Predicates The first three parameters of $LG \subseteq lstate \times lstate \times sstate \times lstate \times sstate$ denote the local states of the two components and the shared state before a program transition; the last two parameters stand for the executing component's local state and the shared state after this transition. Predicate $LIdle \subseteq lstate$ encodes local, idle state conditions that hold between finite executions of LCOP. In the case study for example, idle states satisfy the following local restrictions: $LIdle(LS) :\leftrightarrow USucc \wedge \neg\, OSucc$.

The first parameter of $LR \subseteq lstate \times sstate \times sstate$ corresponds to a component's local state before an environment transition. The second resp. third parameter is the shared state before resp. after this transition. In the case study, *LR* ensures for instance that the content of a new cell in push is not changed by the environment if this cell is not yet inserted in the stack.

$$\neg\, USucc' \rightarrow H''[UNew'] = H'[UNew'] \tag{6}$$

Together we can prove the following local decomposition theorem for Spawn where *CS* is replaced by $LSf \times S$ and COP by LCOP respectively:

Theorem 2 (Local Rely-Guarantee Reasoning). *If (5) holds for two arbitrary disjoint local states LS, LSQ, the shared state S and some transitive rely LR, reflexive predicate LG with $LG(LS, LSQ, S, LS', S') \rightarrow LR(LSQ, S, S')$, symmetric predicate LDisj and predicates LInv, LIdle, then:*

$$\textsc{Spawn}, \Box\ R_{\textsc{Spawn}}, Init_{\textsc{Spawn}} \vdash \Box\ ((\exists\ p.\ \varphi_{LG}(p)) \wedge \varphi_{LI} \wedge \varphi_{LIdle})$$

where $\varphi_{LG}(p)$ states that the system step of a component p does not modify the local states of other components q and it satisfies LG,

$$\varphi_{LG}(p) :\leftrightarrow \forall\ q \neq p.\ LSf(q) = LSf'(q) \wedge LG(LSf(p), LSf(q), S, LSf'(p), S')$$

[4] These are part of *Inv* in the global rely-guarantee theory.

the invariant conditions hold for all components at all times,

$$\varphi_{LI} :\leftrightarrow \forall\ p \neq q.\quad LInv(LSf(p), S) \wedge LInv(LSf'(p), S') \\ \wedge\ LDisj(LSf(p), LSf(q)) \wedge LDisj(LSf'(p), LSf'(q))$$

and each p is idle before and after LCOP*:*

$$\varphi_{LIdle} :\leftrightarrow \forall\ p.\ \neg\ Actf(p) \rightarrow LIdle(LSf(p))$$

Proof. By instantiating CS with $LSf \times S$, COP with LCOP, predicates Inv, $Idle$, G and R with predicates $Inv_\natural$, $Idle_\natural$, $G_\natural$ and $R_\natural$ as given below and verifying that the preconditions of Theorem 1 follow from those of Theorem 2.

$$Inv_\natural(LSf, S) :\leftrightarrow \forall\ p \neq q.\ LInv(LSf(p), S) \wedge LDisj(LSf(p), LSf(q))$$
$$Idle_\natural(p, LSf, S) :\leftrightarrow LIdle(LSf(p)); \quad G_\natural(p, LSf, S, LSf', S') :\leftrightarrow \varphi_{LG}(p)$$
$$R_\natural(p, LSf', S', LSf'', S'') :\leftrightarrow LSf'(p) = LSf''(p) \wedge LR(LSf'(p), S', S'')$$

4.3 Local Proof Obligations for Linearizability and Lock-Freedom

Linearizability [2] and lock-freedom [3] are major correctness resp. progress properties of concurrent systems. In this section we define local proof obligations for LCOP which imply linearizability and lock-freedom of Spawn. They are based on invariant properties $LISR$ with one local state LS that each component may always assume during its execution of LCOP$(In; LS, S, Out)$, according to Theorem 2.

$$LISR :\leftrightarrow LInv(LS, S) \wedge LInv(LS', S') \wedge LS' = LS'' \wedge LR(LS', S', S'')$$

Every component can assume $LInv$ at all times according to φ_{LI}. Since $\varphi_{\mathrm{LG}}(p)$ implies that each component p does not modify other local states and satisfies its guarantee, each component can in return also assume that its local state is never concurrently changed and that its rely holds at all times (recall $LG \rightarrow LR$).

Linearizability Based on these assumptions established by rely-guarantee reasoning, we prove linearizability by locating the linearization point (i.e., the step where a call appears to take effect) of each operation in LCOP. [5] Conceptually, the linearization point is determined in a refinement proof using an abstraction function $Abs \subseteq sstate \times astate$ (a partial function on shared states that satisfy $LInv$, which returns a corresponding abstract state). In the stack example, Abs maps the stack in memory to a finite algebraic list St of its data values.

$$Abs(Top.\mathrm{ref}, H, [\,]) :\leftrightarrow Top.\mathrm{ref} = null$$
$$Abs(Top.\mathrm{ref}, H, v + St) :\leftrightarrow \quad Top.\mathrm{ref} \neq null \wedge Top.\mathrm{ref} \in H \\ \wedge\ H[Top.\mathrm{ref}].\mathrm{val} = v \wedge Abs(H[Top.\mathrm{ref}].\mathrm{nxt}, H, St)$$

[5] Our current approach suffices when a linearization point is within the code of the executing component, even when its location depends on future behavior. This is possible, since analyzing future states of an interval is possible in ITL (cf. [7] for a detailed description of such an example.)

$$
\begin{array}{l}
APush(In; St)\ \{ \\
\quad skip^*; \\
\quad St := push(In, St); \\
\quad skip^*\}
\end{array}
\qquad
\begin{array}{l}
APop(; St, Out)\ \{ \\
\quad \textbf{let}\ Lo = empty\ \textbf{in}\ \{ \\
\quad\quad skip^*; \\
\quad\quad \textbf{if*}\ St \neq [\,]\ \textbf{then}\ \{ \\
\quad\quad\quad Lo := top(St), St := pop(St)\}; \\
\quad\quad skip^*; Out := Lo\}\}
\end{array}
$$

Fig. 4. Abstract stack operations.

To prove linearizability, one has to show that each concrete operation from LCOP non-atomically refines a corresponding abstract operation, which is defined in a further generic procedure AOP. In the case study, AOP is the nondeterministic choice between an abstract *APush* or *APop*, which are shown in Figure 4. They use atomic operations *push* resp. *pop* to add resp. remove an element from *St* at concrete linearization points and additional skip steps at non-linearization points.

Refinement (i.e., interval inclusion) between LCOP and AOP is simply expressed as LCOP $\vdash$ AOP in the logical framework. Hence, the local refinement proof obligation for linearizability is:

$$
\begin{array}{l}
\mathrm{LCOP}(In; LS, S, Out),\ \Box\ (LISR \wedge Abs(S, AS) \wedge Abs(S', AS')),\ LIdle(LS) \vdash \\
\mathrm{AOP}(In; AS, Out)
\end{array}
\tag{7}
$$

Lock-Freedom A concurrent system is lock-free if some of its running operations always terminates in a finite number of steps, even if individual components are arbitrarily delayed or fail. In our concurrent system model (Fig. 3), this is modeled by requiring that some active operation (expressed using activity function *Actf*) always eventually becomes inactive. This is true, even if the scheduling is non-fair $\|_{\mathrm{nf}}$ (to model failure), as discussed in [15], p. 393 and following. (Also see [10] for full proofs.) However, individual components of a lock-free system might starve. In the stack example, single push and pop operations can be forced to always retry their loop if another process modifies the shared top pointer. Yet, if such an interference always occurs, it is an interfering process which terminates its current execution and without interference, the current process eventually terminates. We formalize this intuitive argument using an additional reflexive and transitive relation $U \subseteq sstate \times sstate$ ("unchanged") which describes interference freedom. Note that U represents an unbounded amount of interference which a process might "suffer" from *or* perform. For the stack, we determine the "unchanged" relation as identity of the shared variable *Top*.

$$U(S_0, S_1) :\leftrightarrow Top_0 = Top_1$$

To prove lock-freedom (based on rely-guarantee conditions *LISR*), two local termination proofs for each operation in LCOP are sufficient: termination without interference from the *environment* ($\Box\ U(S', S'') \rightarrow \Diamond$ **last**) and termination after violation of U by the *program* ($\neg\ U(S, S') \rightarrow \Diamond$ **last**):

$$
\begin{array}{l}
\mathrm{LCOP}(In; LS, S, Out),\ \Box\ LISR,\ LIdle(LS) \vdash \\
\Box\ ((\Box\ U(S', S'')) \vee \neg\ U(S, S') \rightarrow \Diamond\ \textbf{last})
\end{array}
\tag{8}
$$

5 Local Verification of the Stack

This section describes the application of the local decomposition theory to verify memory-safety, ABA-prevention, linearizability and lock-freedom of a concurrent stack application $\textsc{Spawn}(n;\dots)$ where all $n+1$ processes execute the push and pop operations from Figure 2. The specifications and proofs consider at most two representative processes. The explicit reuse of memory locations makes the verification notably more challenging than proving the stack under the assumption of GC, which implicitly avoids the reuse of a memory location that is referenced in some operation.

5.1 Instantiating the Local Predicates

LInv Predicate *LInv* encodes several state invariant properties of the stack $\mathit{LInv} :\leftrightarrow \varphi_{st} \wedge \varphi_n \wedge \varphi_{\mathrm{free}} \wedge \varphi_t$. According to φ_{st}, the implementation represents some finite list, i.e., $\mathit{Abs}(\mathit{Top}.\mathrm{ref}, H, \mathit{St})$ always holds for some St.

$$\varphi_{st} :\leftrightarrow \exists\ \mathit{St}.\ \mathit{Abs}(\mathit{Top}.\mathrm{ref}, H, \mathit{St})$$

To maintain this property new cells that are to be pushed on the stack must be allocated and disjoint from the stack according to φ_n. (A standard reachability predicate $\mathit{reach}(\mathit{Top}.\mathrm{ref}, r, H)$ checks whether a location r is in the stack.)

$$\varphi_n :\leftrightarrow \neg\ \mathit{USucc} \rightarrow \mathit{UNew} \neq \mathit{null} \wedge \mathit{UNew} \in H \wedge \neg\ \mathit{reach}(\mathit{Top}.\mathrm{ref}, \mathit{UNew}, H)$$

Invariant φ_{free} ensures major safety aspects of the memory reclamation scheme: freed locations $r \in \mathit{Free}$ are allocated and disjoint from the stack, since otherwise the reuse of r would cause an access error or corrupt the stack; r is also disjoint from new cells and from removed locations (i.e., removed from the stack but not yet freed $\mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null}$) and thus the memory pool is duplicate-free.

$$\begin{aligned}\varphi_{\mathrm{free}} :\leftrightarrow \forall\ r \in \mathit{Free}.\quad & r \neq \mathit{null} \wedge r \in H \wedge \neg\ \mathit{reach}(\mathit{Top}.\mathrm{ref}, r, H)\\ & \wedge (\neg\ \mathit{USucc} \rightarrow r \neq \mathit{UNew})\\ & \wedge (\mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \rightarrow r \neq \mathit{OTop}.\mathrm{ref})\end{aligned}$$

We must also know that locations $\mathit{OTop}.\mathrm{ref} \neq \mathit{null}$ are allocated and that removed locations are disjoint from the stack (φ_t).

$$\begin{aligned}\varphi_t :\leftrightarrow\quad & (\mathit{OTop}.\mathrm{ref} \neq \mathit{null} \rightarrow \mathit{OTop}.\mathrm{ref} \in H)\\ & \wedge (\mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \rightarrow \neg\ \mathit{reach}(\mathit{Top}.\mathrm{ref}, \mathit{OTop}.\mathrm{ref}, H))\end{aligned}$$

LDisj Three disjointness properties between local pointers of the two processes are used. To ensure symmetry we define $\mathit{LDisj}(LS, LSQ) :\leftrightarrow \mathit{disj}(LS, LSQ) \wedge \mathit{disj}(LSQ, LS)$ where $\mathit{disj}(LS, LSQ) :\leftrightarrow (4) \wedge \delta_{rm} \wedge \delta_{tn}$. Property δ_{rm} states that concurrently removed locations are disjoint, whereas δ_{tn} ensures that removed locations are disjoint from concurrent new cells.

$$\begin{aligned}\delta_{rm} :\leftrightarrow\quad & \mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \wedge \mathit{OSuccq} \wedge \mathit{OTopq}.\mathrm{ref} \neq \mathit{null}\\ & \rightarrow \mathit{OTop}.\mathrm{ref} \neq \mathit{OTopq}.\mathrm{ref}\\ \delta_{tn} :\leftrightarrow\ & \mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \wedge \neg\ \mathit{USuccq} \rightarrow \mathit{OTop}.\mathrm{ref} \neq \mathit{UNewq}\end{aligned}$$

LR We define several rely conditions $LR :\leftrightarrow \rho_{st} \wedge \rho_{nst} \wedge \rho_{ge} \wedge \rho_{rm} \wedge (6)$ to intuitively formalize the non-trivial synchronization mechanism that avoids the ABA-problem. In particular, rely conditions ρ_{st} and ρ_{nst} make sure that during a pop, the ABA-prone location $OTop$.ref either stays in the stack and its contents are unchanged or if it is concurrently removed, then $OTop$.ref is not reinserted in the stack or the modification counter is increased.

$$\begin{aligned}\rho_{st} :\leftrightarrow \quad & \neg\, OSucc' \wedge OTop'.\text{ref} \neq null \wedge Top' = OTop' \\ \rightarrow \quad & Top'' = Top' \wedge H''[OTop'.\text{ref}] = H'[OTop'.\text{ref}] \\ & \vee \neg\, reach(Top''.\text{ref}, OTop'.\text{ref}, H'') \vee Top''.\text{cnt} > Top'.\text{cnt}\end{aligned}$$

$$\begin{aligned}\rho_{nst} :\leftrightarrow \quad & \neg\, OSucc' \wedge OTop'.\text{ref} \neq null \wedge \neg\, reach(Top'.\text{ref}, OTop'.\text{ref}, H') \\ \rightarrow \; & \neg\, reach(Top''.\text{ref}, OTop'.\text{ref}, H'') \vee Top''.\text{cnt} > Top'.\text{cnt}\end{aligned}$$

The remaining simple relies ensure that the modification counter never decreases (ρ_{ge}) and that the content of removed locations is unchanged (ρ_{rm}).

$$\rho_{ge} :\leftrightarrow Top'.\text{cnt} \leq Top''.\text{cnt}$$
$$\rho_{rm} :\leftrightarrow OSucc' \wedge OTop'.\text{ref} \neq null \rightarrow H'[OTop'.\text{ref}] = H''[OTop'.\text{ref}]$$

LG The reclamation scheme avoids memory leaks, i.e., all heap locations r are either in the stack or in the free-set or owned by a process at all times in each execution of Spawn, where every process owns its new and removed locations.

$$\begin{aligned}&owns(r, LS){:} \leftrightarrow \\ &(\neg\, USucc \wedge UNew = r) \vee (OSucc \wedge OTop.\text{ref} \neq null \wedge OTop.\text{ref} = r)\end{aligned}$$

We decompose the absence of memory leaks to a local guarantee *noleaks*, which ensures that process steps do not create leaks.

$$\begin{aligned}&noleaks(LS, S, LS', S') :\leftrightarrow \\ &\forall\, r. \quad r \notin H \vee reach(Top, r, H) \vee r \in Free \vee owns(r, LS) \\ &\qquad \rightarrow r \notin H' \vee reach(Top', r, H') \vee r \in Free' \vee owns(r, LS')\end{aligned}$$

Predicate LG is then defined to maintain *noleaks* and the rely conditions of the other process $LG(LS, LSQ, \dots) :\leftrightarrow noleaks(\dots) \wedge LR(LSQ, \dots)$.

5.2 The Main Proofs

The main effort of the case study is to prove (5) –sustainment of the verification conditions LG, $LInv$ and $LDisj$ for the steps of each operation if the environment has previously maintained LR. We proceed by case analysis over operation $Op \in \{Push, Pop\}$. The proof resembles a Hoare-style proof of a sequential program. In particular, before executing a loop we generalize the current state assumptions to a Hoare-style invariant (and use $\xrightarrow{+}$ induction when the loop is reiterated). Each program statement in Op is consecutively, symbolically executed according to Section 3. Only some major arguments are outlined.

Sustainment of the Verification Conditions $Op \equiv Push$: The allocation step ($GetNew$) resets the content of a new cell. However, if the free-set is empty, this step does not affect allocated locations and otherwise invariant φ_{free} ensures that no rely conditions of the other process are violated.

$Op \equiv Pop$: After taking the snapshot in line O3 in case of a non-null top-of-stack pointer, the proof proceeds by case distinction: according to rely condition ρ_{st} there are three possible cases in the next state. First, when the shared top-of-stack pointer has not been concurrently modified, the content of the snapshot location is unchanged and the current iteration can still succeed correctly. In the second resp. third case, the snapshot location $OTop$.ref is either not in the stack anymore or it has been reinserted and thus the modification counter has been increased. In both latter cases rely conditions ρ_{nst} and ρ_{ge} ensure that the loop must be reiterated. Hence the CAS can not erroneously succeed and cause an ABA-problem.

Linearizability The proof of linearizability (proof obligation (7)) distinguishes between the two possible concrete operations. In case of a push operation, the linearization point is the successful CAS. Rely (6) ensures that the initial value of the new cell and its next reference are immutable. Hence, the successful CAS corresponds to an abstract push of the invoked value. The pop operation has one linearization point in line O3 if the stack is empty, or else in line O7 if the CAS succeeds. Relies ρ_{st} and ρ_{rm} ensure that the successful CAS corresponds to an abstract pop and that the correct value is returned.

Lock-Freedom According to (8), the proof of lock-freedom requires termination proofs for each data structure operation if environment behavior is restricted according to U and if a step violates U. The termination proofs for push and pop mainly automatically step through the code until an operation terminates or they apply induction whenever a loop is retried. The required term for induction is extracted from the always formula in the succedent of (8).

Verification Effort in KIV The soundness proofs for the improved global decomposition theory took about four man-weeks. In particular, the decomposition proof of lock-freedom is tedious as it must consider many possible interleavings. The derivation of the local instance took about two man-weeks. The main challenge was to find the right local proof obligations and the instantiation of the global predicates (see proof of Theorem 2). Using the local instead of the global theory to verify the stack under GC reduced the size of the verification conditions by around one third. The verification of the stack with explicit memory reuse took around two man-weeks and was about twice as complex as verifying the stack under GC. The main new challenge was to find the right heap invariants that ensure memory-safety (φ_{free}, *noleaks*) and the rely conditions for ABA-prevention (ρ_{st}, ρ_{nst}).

6 Related Work and Comparison

Compositional Verification Most approaches to compositional reasoning justify the rules they use on a semantic level (e.g., [14], [16]). A mechanized soundness and completeness proof for global rely-guarantee rules for interleaved programs with shared variables has been given by Nieto et al [17]. The verification is based on Isabelle's higher-order logic, and therefore in essence had to explicitly formalize intervals (using a small-step semantics for programs). Since our

proof is based on a strong temporal logic (instead of just HOL), where intervals are already part of the semantics, proving the soundness of rely-guarantee rules using ITL is much simpler in our setting.[6]

Local Rely-Guarantee Proof Obligations There are two approaches [18, 19] which combine rely-guarantee and separation logic for heap-modular reasoning. Our work borrows this idea to achieve process-local reasoning and complements their work by also considering liveness (lock-freedom). Separation logic's operator $*$ and the framing rule permits to "hide" heap disjointness invariants, which we encode explicitly. Fu et al. [20] define a temporal logic of the past to manually verify ABA-prevention for a lock-free stack with hazard pointers [11]. The local rely-guarantee instance presented here mechanizes such proofs, and allows to *additionally* mechanize linearizability and lock-freedom proofs of this challenging algorithm. This is demonstrated in [21] which briefly sketches the main ideas of the local rely-guarantee theory layed out in this paper, and then mainly focusses on the generic verification of lock-free algorithms with hazard pointers. In contrast, this work gives a detailed presentation of our process-local rely-guarantee reasoning approach and outlines its application using another well-known lock-free memory reclamation scheme.

In general, techniques that exploit the symmetry of identical system components have also been developed in model checking (cf. [22] for an overview). However, proving linearizability using (symmetric) model checking often fails (cf. [23] and [24] for recent work on model checking linearizability). Model checking is good at finding bugs in lock-free algorithms by showing counter examples. However, since it checks short executions of a few processes only, it does not give full proofs.

Verification of Linearizability Mechanized verification approaches for linearizability can be roughly classified into three categories: automated approaches based on shape analysis and separation logic, and interactive approaches. Automated approaches can verify the stack example assuming garbage collection, see [25] and [26], and the latter is able to solve many interesting examples automatically, including some cases where linearization points lie outside of the code of the executing thread. Our proof obligations for linearizability have to be generalized to handle some of these examples. However, verification under GC is much simpler than verification using modification counters (which is still simpler than with hazard pointers).

The work most closely related to ours is Doherty, Groves et al. [27], which verifies linearizability of a lock-free queue with modification counters in PVS. The approach is related to ours in also using refinement to prove linearizability. It is global however, and has to encode the algorithms as a concurrent IO-Automaton. Lock-freedom is not discussed. Later on, Groves et al. [9] gave a manual verification approach for the stack with modification counters, based on trace reduction and incremental refinement. Our impression is that mechanizing their arguments

[6] We have justified some of the more difficult rules of ITL using an embedding of the semantics into HOL. Proofs over this theory are rather complex too. Like many others, the embedding of ITL into HOL is not usable to verify case studies.

about commuting steps would be hard. Nevertheless our verification benefited from knowing many of their informal arguments.

Verification of Lock-Freedom Gotsman et al. [28] developed a new logic for proving liveness properties of non-blocking algorithms based on rely-guarantee reasoning and separation logic. Their approach can automatically discharge manually derived proof obligations for lock-freedom, using a combination of tools. In contrast, we mechanically verify both decomposition theorems for safety and liveness properties of concurrent programs and local proof obligations in one logical framework and tool.

7 Conclusion

We have described a mechanically derived local rely-guarantee instance. Such local instances are useful to avoid reasoning about the overall system state when verifying concurrent algorithms where components have similar behaviors, e.g., lock-free data type implementations. Moreover, we have defined local proof obligations for linearizability and lock-freedom based on this instance and have shown its application to verify the major safety and liveness aspects of a lock-free stack with explicit memory reuse.

In current work, we have successfully applied the approach described here to locally verify linearizability and lock-freedom of the Michael-Scott queue with hazard pointers [11] and of a refined version of the stack, where the abstract free-set is replaced by a further lock-free stack. These proofs are online too [10]. Our recent work also shows that a local verification of Michael's lock-free set algorithm [29] is possible too. Moreover, we currently generalize the decomposition of linearizability to treat more complex linearization points, adapting results from [30]. These improved techniques are applied to further challenging concurrent algorithms.

References

1. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, North-Holland (1983) 321–332
2. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Prog. Languages and Systems **12**(3) (1990) 463–492
3. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University (1991)
4. Moszkowski, B.: Executing Temporal Logic Programs. Cambr. Univ. Press (1986)
5. Burstall, R.M.: Program proving as hand simulation with a little induction. Information processing 74 (1974) 309–312
6. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In Bibel, W., Schmitt, P., eds.: Automated Deduction—A Basis for Applications. Volume II: Systems and Implementation Techniques. Kluwer Academic Publishers, Dordrecht (1998) 13 – 39
7. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. Formal Aspects of Computing (FAC) **23**(1) (2011) 91–112

8. Treiber, R.K.: System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
9. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) **21**(1–2) (2009) 187–223
10. KIV: Presentation of proofs for concurrent algorithms (2011) URL: `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`.
11. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6) (2004) 491–504
12. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications **23**((2,3)) (2010) 285–307
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of TIME, to appear. IEEE, CPS (2011)
14. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
15. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: In Proc. of MPC 2010. Springer LNCS 6120 (2010) 377–396
16. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Logic and Computation **17** (August 2007) 807–841
17. Prensa Nieto, L.: The rely-guarantee method in Isabelle /HOL. In Degano, P., ed.: ESOP'03. Volume 2618 of LNCS., Springer (2003) 348–362
18. Feng, X., Ferreira, R., Z.Shao: On the relationship between concurrent separation logic and ag reasoning. In: Proc. ESOP. Springer LNCS 4421 (2007) 173 – 188
19. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. Volume 4703 of Springer LNCS. (2007) 256–271
20. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: CONCUR. (2010) 388–402
21. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Proc. ICTAC, Springer LNCS 6916 (2011)
22. Miller, A., Donaldson, A., Calder, M.: Symmetry in temporal logic model checking. ACM Comput. Surv. **38** (September 2006)
23. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software, Springer-Verlag (2009) 261–278
24. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., R.Alur: Model checking of linearizability of concurrent list implementations. In: CAV. Volume 4144 of LNCS. (2010) 465–479
25. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: CAV'08, Springer (2008)
26. Vafeiadis, V.: Automatically proving linearisability. In: CAV. Volume LNCS 6174., Springer (2010) 450–464
27. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE 2004. Volume 3235 of LNCS. (2004) 97–114
28. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that nonblocking algorithms don't block. In: POPL, ACM (2009) 16–28
29. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: In Proc. of SPAA '02. SPAA '02, ACM (2002) 73–82
30. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisabilty with potential linearisation points. In: Proc. Formal Methods (FM), Springer LNCS 6664 (2011)