

UNIVERSITÄT AUGSBURG

**Proving Linearizability with Temporal
Logic**

S. Bäuml, M. Balser, W. Reif, G. Schellhorn

Report 2008-19

December 2008

INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG



Copyright © S. Bäumlér, M. Balsér, W. Reif, G. Schellhorn
Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
<http://www.Informatik.Uni-Augsburg.DE>
— all rights reserved —

Abstract

Linearizability is a correctness criterion for concurrent systems. In this report, we describe how temporal logic can be used to prove linearizability of a concurrent lock-free stack implementation. The logic used is an extended variant of Interval Temporal Logic, which is integrated in the KIV interactive theorem prover. To reduce the proof to single components only a compositional reasoning technique is used.

1 Introduction

Verification of concurrent algorithms is a current and important research topic. Usually, reasoning over a concurrent system is hard and tedious work as all possible interleavings have to be considered. To avoid reasoning over the complete concurrent system, a common technique is compositional reasoning. The idea of compositional reasoning was first formulated in [1] by Dijkstra. The basic idea of this technique is, to split a system into several subcomponents. Then, the overall property is proved with corresponding properties of the subcomponents only.

A common compositional proof technique is the assumption-guarantee paradigm, which was introduced by Jones [2] and by Misra & Chandy [3]. 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 an assumption-guarantee technique provides a theorem, that specifies in a number of proof obligations how the various assumptions and guarantees have to be connected in order to show the property for the overall system. Ideally, these proof obligations contain only single subcomponents and properties of these subcomponents, but not the complete system itself. This results in several proofs of feasible size. Most assumption-guarantee techniques use temporal logics as underlying logic. Examples for this technique can be found e.g. in [4, 5, 6].

Herlihy & Wing [7] define linearizability as a correctness criterion for processes with concurrent access on a shared data structure. 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.

Lock-free algorithms are a class of algorithms for concurrent access to data structures. Unlike the classic mutex based algorithms lock-free algorithms require no locking. Therefore they are less vulnerable to common problems such as deadlocks, livelocks and priority inversion. On the other hand, a disadvantage of lock-free algorithms compared to mutex algorithms is an increased complexity which makes it hard to assure their correctness intuitively.

In [8] a lock-free stack algorithm is presented that accesses the stack only via atomic *compare-and-swap*-operations. Using this example this paper presents an approach that combines refinement and compositional assumption-guarantee

reasoning to prove linearizability. The temporal logic we use [9] has explicit imperative interleaved programs and is a variant of ITL [10]. It is integrated into the interactive theorem prover KIV [11].

In the following, we assume that the reader has at least basic knowledge about the sequent calculus and temporal logic. This paper is subdivided as follows: Section 2 introduces the lock-free stack algorithm we use and presents an informal overview over our proof strategy. In Section 3 we give an informal overview of the temporal logic framework of KIV and the assumption-guarantee technique we use. The main part of the paper, Section 4, contains the details of the linearization proof of the lock-free stack algorithm. The paper concludes with sections about related work (Section 5) and a short summary (Section 6).

2 Lock-Free Stack Algorithm in KIV

The basic principle of the lock-free stack algorithm consists of two phases. In the first phase all the needed data are prepared without changing the main data structure, e.g. storage is allocated or data are read. Then, in a second phase, the algorithm attempts to make the data consistent in a single atomic step by using a *compare-and-swap* (CAS) command. If this atomic step fails (e.g. because the data structure has changed), the main data structure is not changed and the algorithm repeats phase one.

The informal idea of the CAS command is, that a local pointer l_1 is compared to a global pointer G .¹ 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:

$$\begin{array}{l} \text{CAS}(l_1, l_2; G, \text{Success}) \\ 1 \quad (G = l_1 \wedge (G := l_2, \text{Success} := \text{TRUE})) \vee \\ 2 \quad (G \neq l_1 \wedge \text{Success} := \text{FALSE}) \end{array}$$

Line 1 covers the case where the CAS succeeds (the comma denotes an atomic assignment). In line 2 the failure of the CAS operation is handled.

The stack algorithm is represented in KIV by a linked list which is stored in a heap H . Each heap cell has a field for the data value (accessible by the function *.val*) and a field for a pointer (accessible via *.next*), which can also contain a NULL value that denotes 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

¹Note, that variables beginning with a small letter are considered as constants in our framework, while variables starting with a capital letter can change dynamically

```
CPUSH( $v$ ;  $Top$ ,  $H$ )
1  let  $Ss = \text{NULL}$ ,  $Nl = \text{NULL}$ ,  $Success = \text{FALSE}$ 
2  in { $\text{ALLOC}(H, Nl)$ ;
3      $H[Nl].data := v$ ;
4     while  $\neg Success$ 
5     do { $Ss := Top$ ;
6          $H[Nl].next := Ss$ ;
7          $\text{CAS}(Ss, Nl; Top, Success)$ 
8     }
9 }
```

Figure 1: Formalization of the push algorithm in KIV

are the value v , which should be inserted into the stack and the variables Top and H for the stack representation. In line 1 some local variables are defined. Ss and Nl are pointers while $Success$ is a boolean variable. The pointer Ss is used to detect, if the stack has changed while Nl contains the new data value, that should be included into the stack. The algorithm starts by allocating a new cell on the heap and storing the pointer in the variable Nl (line 2) and storing the data value in this new allocated cell (line 3). After that the algorithm loops, as long as the insertion of the new cell in the stack fails (line 4 to 8). Inside the loop the pointer of the current top cell is stored in variable Ss (line 5) and the *.next*-pointer of the previously allocated cell is set to the current top cell (line 6). Finally, the new data value is added to the top of stack data structure by a CAS operation (line 7). That means, if the current top-of-stack pointer Top is still the same as it was in line 5, the previously allocated cell Nl contains the correct *.next*-pointer and Top can be set to Nl . If the top-of-stack pointer was changed by another push or pop process in the meantime, the CAS operation fails and the while-loop is reiterated.

The principle for the pop algorithm is the same as for the push algorithm. The specification of the pop algorithm is shown in Figure 2. Here, parameter V is used as return value of CPOP and the local variable $V0$ to temporarily store the return value. Inside the loop, the first operation is to store the current top-pointer inside Ss to detect changes of the stack afterwards (line 3). If the stack is currently empty a special value `EMPTY` is returned and the process terminates (line 4-6). In the other case the pointer to the second cell in the stack and the data of the top-cell is retrieved (line 7 and 8). In line 9, if the stack is still unchanged the CAS-operation changes the Top -pointer to the second cell, else it repeats the loop again. After execution of the loop the final result is assigned to the return parameter V (line 12).

As both algorithms use pointers to detect whether the stack has changed the so called ABA-problem is common for the class of lock-free algorithms. It can occur, if cells are deallocated and newly allocated afterwards while there is still a pointer referencing to it. In this case, both algorithms can perform a successful

```

CPOP( $V, Top, H$ )
1  let  $V0 = \text{EMPTY}, Ss = \text{NULL}, NewTop = \text{NULL}, Success = \text{FALSE}$ 
2  in {while  $\neg Success$ 
3      do { $Ss := Top$ ;
4          if  $Ss.isnull$ 
5              then {  $V0 := \text{EMPTY};$ 
6                       $Success := \text{TRUE}$  }
7          else {  $NewTop := H[Ss].next$ ;
8                   $V0 := H[Ss].val$ ;
9                   $\text{CAS}(Ss, NewTop, Top, Success)$ 
10                 };
11         }
12      $V := V0$ ;
13 }

```

Figure 2: Formalization of the pop algorithm in KIV

CAS although the stack has changed, which may result in unpredictable results of the algorithms. There are several techniques to avoid this problem. One is to use a garbage collection, which deallocates cells only if there is no pointer referencing to it. In this paper we do not deallocate cells explicitly, assuming there is a separate garbage collection deallocating all unreferenced cells. An alternative is to use separate modification counters, that allow detection of cell changes (see e.g. [12, 13]). These can be introduced in a separate refinement. [14]

3 Temporal Logic Framework

In this section we give an informal overview over the temporal logic calculus we use, which is integrated into the interactive theorem prover KIV. The logical formalism is described in detail in [15, 9]. The temporal logic framework is a variant of ITL [10] that is extended by explicitly including the behavior of the environment into each step. The basis for ITL are finite or infinite sequences π of valuations, which are called *intervals*. Valuations in π are called *states*. Each state is described by a higher order predicate logic formula over dynamic variables V , which also can be *primed* V' or *double primed* V'' . V and V' describe a *system transition*, whereas the step between V' and V'' describes an *environment transition*. The value of V'' in a state must be equal to the value of V in the next successive state. Thereby system and environment transitions alternate. Constants are written in small letters. A selection of temporal operators supported by KIV are:

$\Box\varphi$	φ holds <i>always</i> from now on in every state
φ unless ψ	either φ holds always from now on or ψ holds in some state and φ holds in every state before
$\varphi_1 \parallel \varphi_2$	interleaving
$\exists V.\varphi$	temporal exists-quantifier
$X := t$	assignment
$X := t_1, Y := t_2$	atomic assignment of multiple variables
$\varphi_1; \varphi_2$	sequential composition
if ψ then φ_1 else φ_2	case distinction
let $X = t$ in φ	local variable declaration
while ψ do φ	loop

As shown, our ITL variant supports classic temporal logic operators as well as program operators. This allows us to mix programs with temporal logic formulas. Program operators describe only system steps that alternate with arbitrary environment steps.

3.1 Symbolic Execution

A typical sequent in proofs about interleaved programs has the form $P, A, \Gamma \vdash \Delta$. P is the interleaved program that executes the system steps, A contains a temporal formula that describes behavior of the environment and Γ is a predicate logic formula for the current variable assignment, while Δ contains the property which has to be shown.² To verify that Δ holds it must be proved that the current state Γ does not violate Δ and that the rest of the program run of P does not violate Δ either. To show the latter, symbolic execution is used.

For example, a sequent of the form mentioned above might look like this:

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

The program executed is $M := M + 1; \alpha$ (α can be an arbitrary program) and the environment is assumed not to change M (formula $\Box M' = M''$). As the current state $M = 2$ does not violate $\Box M > 0$, a symbolic execution step is used to show that the rest of the program 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. So for the example above, a symbolic execution step would lead to the following formula:

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

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

² P, A or Γ can be left unspecified by simply omitting them. E.g. if A is missing in a formula an arbitrary behaviour of the environment is assumed.

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).

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. If we can prove that the property is true at beginning and during the loop it is invariant the proof can be finished with an inductive argument.

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

$$M := 1; \alpha_1 \parallel N := 2; \alpha_2, \Gamma \vdash \Delta$$

this formula can be transformed into the following two cases:

$$\begin{aligned} M := 1; (\alpha_1 \parallel N := 2; \alpha_2), \Gamma \vdash \Delta \\ N := 2; (m := 1; \alpha_1 \parallel \alpha_2), \Gamma \vdash \Delta \end{aligned}$$

which can be symbolically executed by the mechanism described above. For more details about symbolic execution, interleaving and induction in KIV see [9].

3.2 Rule for Compositional Reasoning

In [16] we presented a compositional theorem for proof safety formulas similar to standard compositional theorems such as [4, 6]. However, to show linearizability we use a modification of this theorem, which is presented in the following.

Most assumption-guarantee based compositional proof techniques use a special operator similar to the "while-plus" operator $\overset{+}{\rightarrow}$ presented in [5]. Informally, the term $A \overset{+}{\rightarrow} G$ means, that if A holds up to step i , then G must hold up to step $i + 1$. With this operator it is possible to express that a component violates its guarantee G only after its assumption A is violated. This operator is needed to break the circularity of the compositional rule used.

Assumptions and guarantees can be formulated with propositional predicates over unprimed and primed variables (e.g. [6]). We use the same approach in this work, but for the assumptions we use predicates over primed and doubly primed variables. In this way it is possible to formalize which steps are allowed for the components and which steps are allowed for the environment. This also allows to use a standard TL operator **unless** as $\overset{+}{\rightarrow}$ operator, i.e.:

$$A \overset{+}{\rightarrow} G := G \text{ unless } (G \wedge \neg A)$$

With these preliminaries it is possible to construct the compositionality theorem depicted in Figure 3. Unlike standard assumption-guarantee theorems (e.g. [4,

If for all $1 \leq i, j \leq n, i \neq j$:

1. $M_i(V), \square A_i(V', V''), I(V) \vdash AM_i(V)$
2. $M_i(V), I(V) \vdash A_i(V', V'') \xrightarrow{+}_p G_i(V, V')$
3. $G_i(v_1, v_2) \vdash A_j(v_1, v_2)$
4. $I(v_1) \wedge G_i(v_1, v_2) \vee A(v_1, v_2) \vdash I(v_2)$
5. $A(v_1, v_2) \vdash A_i(v_1, v_2)$
6. $A_i(v_1, v_2), A_i(v_2, v_3) \vdash A_i(v_1, v_3)$

then: $(\parallel_i M_i(V)), \square A(V', V''), I(V) \vdash (\parallel_i AM_i(V))$

Figure 3: AG-Theorem

6]), this theorem does not reason about one interleaved system but about two interleaved systems connected by refinement.

Here, V is the set of variables that describes the system state. M_i is a parallel component, A_i is its assumption to the environment and G_i is its guarantee to the environment. AM_i is an abstract component and A describes the behavior of the overall environment. I contains the initial variable assignment.

The conclusion of the theorem states, that each trace described by interleaving of all parallel components M_i (with assumption of the environment behaviour A and initial state I) has an equivalent trace described by the interleaving of the abstract components AM_i . Premise 1. and 2. are temporal logic formulae while premise 3. - 6. are predicate logic formulas. The proof obligations have the following informal meaning:

1. The concrete component M_i and its assumption to the environment A_i imply the abstract component AM_i .
2. All components M_i must sustain their guarantee G_i as long as their assumption A_i holds.
3. The guarantee of each component does not violate the assumptions of all other components.
4. I is preserved by all guarantees and the global assumption.
5. All component assumptions hold, if the global assumption holds. Therefore, no component assumption is violated in an environment step.
6. The assumptions of all components are transitive. With this property, the components assumption is preserved even if other components make several steps.

This theorem was formally proven in KIV. The proof is similar to the one outlined in [16].

Assumptions and guarantees are usually formulated as conjunction of propositions. This fact is used to split proofs of the predicate logic premises up into several small proofs, i.e. instead of showing that $(G1_i \wedge G2_i) \vdash (A1_j \wedge A2_j)$ holds we can separately show that e.g. $G1_i \vdash A1_j$ and $G2_i \vdash A2_j$ hold. This allows to use the correctness management of KIV to keep track, which assumptions are connected with which guarantees and vice versa.

The theorem can be extended to show also data refinement. I.e. the abstract components AM_i have an abstract data type as parameter, which is described with the variable set V_A . Therefore, an abstraction relation R is needed. The new conclusion of the extended theorem is

$$(\parallel_i M_i(V)), \square A(V', V''), I(V) \vdash \exists V_A. (\square (R(V, V_A) \wedge R(V', V'_A))) \wedge \exists V. (\parallel_i AM_i(V_A)) \quad (1)$$

This conclusion is similar to the original theorem with addition of formula $\exists V_A. \square (R(V, V_A) \wedge R(V', V'_A))$. This formula states that there exist suitable values of V_A generated by runs of all $AM_i(V_A)$, such that a representation invariant R holds before and after system steps. The second quantifier $\exists V$ hides the changes of the concrete system from the abstract system, which is necessary for technical reasons. To show this new conclusion, premise 1. of the theorem in Figure 3 has to be exchanged with

$$(M_i(V), \square A_i(V', V''), I(V) \vdash \exists V_A. (\square (R(V, V_A) \wedge R(V', V'_A))) \wedge \exists C. AM_i(V_A)) \quad (2)$$

while premises 2.-6. are the same.

4 Linearization of the Stack Algorithms

First, a short overview over the case study is given before some of the interesting technical issues are explained in the following subsections. The KIV formalization of the used stack algorithm is presented in Section 2. In the following CSTACK is used as short form for either a push or a pop process (i.e. $CSTACK(C) \leftrightarrow CPUSH(C) \vee CPOP(C)$). The global data used by these algorithms is abbreviated with C . Similarly, ASTACK(A) abbreviates the atomic push and pop (see Fig. 4) on an abstract stack A .

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 [7]. A linearization point is an atomic step that is between the call and return of the operation which contains all effects of the abstract data structure. To show the linearization property for the lock-free stack algorithm in KIV, our

goal is to show that for any run of an arbitrary number n of parallel CSTACK processes there is an equivalent run of n parallel ASTACK processes. The atomic push (resp. pop) marks then the linearization point of that operation. This approach is similar to other approaches showing linearization, e.g. [12, 17]. So the overall property which is proved is the following:

$$\begin{aligned} & (\parallel_{i \in \{1, \dots, n\}} \text{CSTACK}_i(C), \Box C' = C'', \text{Init}(C)) \\ & \vdash \exists A. \Box (R(C, A) \wedge R(C', A')) \wedge \exists C. (\parallel_{i \in \{1, \dots, n\}} \text{ASTACK}_i(A)) \quad (3) \end{aligned}$$

This formula is an instantiation of formula (1). The main proposition of this formula is that for every trace of the concrete system $\parallel_{i \in \{1, \dots, n\}} \text{CSTACK}_i(C)$ an equivalent trace of abstract processes $\parallel_{i \in \{1, \dots, n\}} \text{ASTACK}_i(A)$ exists. This proposition reflects the linearization property. The global assumption $\Box C' = C''$ specifies a closed system, i.e. no external process except CPUSH or CPOP changes the heap. The predicate $\text{Init}(C)$ specifies the initial configuration of the heap.

To avoid reasoning over the whole system, the AG-theorem presented in Section 3 is used to reduce the proof to single components. For process i this gives the proof obligation (as instantiation of formula (2))

$$\begin{aligned} & \text{CSTACK}_i(C) \wedge \Box A_i(C', C'') \wedge \text{Init}(C) \\ & \vdash \exists A. \Box (R(C, A) \wedge R(C', A')) \wedge \exists C. \text{ASTACK}_i(A) \quad (4) \end{aligned}$$

This formula replaces the whole system by a single component. The environment assumption $\Box A_i(C', C'')$ is needed to describe the behavior of other push and pop processes, which form the environment for the process $\text{CSTACK}_i(C)$. Of course the A_i s must be chosen in a way that the premises 4. and 5. of the AG-Theorem hold.

Finally, to apply the AG-Theorem and prove the overall property (formula (3)) we have to find suitable guarantees for CPUSH and CPOP that imply the assumptions A_i (premise 3. of the AG-Theorem) and holds for the respective component (premise 2.).

In Subsection 4.1 the formalization of the abstract stack and the predicates R and Init is described. The proof of formula (4) and the assumptions A_i are described in Subsection 4.2. Finally, Subsection 4.3 describes the application of the AG-theorem.

4.1 Specification of Important Predicates

For verification, the following adaptations of the algorithms presented in Section 2 were made to simplify the verification in KIV: For the push algorithm in Figure 1 a global array $Nl[1] \dots Nl[n]$ is used instead of the local variables Nl to

<pre> APUSH(<i>v</i>; STACK) 1 skip*; 2 STACK := push(<i>v</i>, STACK); 3 skip* </pre>	<pre> APOP(<i>V</i>, STACK) 1 let V0 = NIL 2 in {skip*; 3 V0 := top(STACK), 4 STACK := pop(STACK); 5 skip*; 6 V := V0} </pre>
---	---

Figure 4: Formalization of the abstract stack algorithm in KIV

store the new allocated cells of the push algorithms in the array-cell $Nl[i]$. So every occurrence of Nl in $CPUSH_i$ is replaced with $Nl[i]$ and the process identifier i is added as additional parameter. This enables the formalization of the environment assumption, as shown in Subsection 4.2. For the same reason cells $Ss[i]$ of a global array are used for storing the top-of-stack pointer in $CPOP_i$ (Figure 2 instead of the local variable Ss). The correctness of both adaptations can be shown by an additional indirection step. Apart from these changes, the same programs as shown in Section 2 are used directly in KIV.

The abstract push and pop algorithms are depicted in Figure 4. For the abstract stack a standard algebraic definition for stacks is used. Both, the APUSH and the APOP algorithm access the stack only in a single atomic operation. Before and after this operation, a *skip** operation is used to include an undefined number of stuttering steps. As last step, APOP assigns the result to the global return variable V .

For the refinement relation the predicate $represents(St, Top, H)$ is defined recursively over the length of the stack by the following two axioms:

$$represents(EMPTY, Top = null, H) \tag{5}$$

$$represents(push(d, St), Top, H) \leftrightarrow Top \in H \wedge H[Top].val = d \wedge represents(St, H[Top].next, H) \tag{6}$$

St is a stack, d is a data value, Top is the pointer to the top of the stack and H is a heap. In formula (5) the empty stack is represented when Top is the nullpointer. For the recursive case in formula (6), Top has to be allocated in H , 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 pointer in the cell referenced by Top . As refinement relation $R(C, A)$ we use the formula $represents(St, Top, H)$.

To show the correctness of the concrete stack algorithms an assumption about the initial state of the stack is needed. Initially, the data structure represented by the heap and the top variable has to be *valid*, i.e. that it really represents a stack and is not cyclic.

$$valid(Top, H) \leftrightarrow \exists St.represents(St, Top, H) \tag{7}$$

PushA1 $valid(Top', H') \rightarrow valid(Top'', H'')$

PushA2 $\neg reachable(Nl'[i], Top', H') \rightarrow$
 $\neg reachable(Nl''[i], Top'', H'')$

PushA3 $Nl'[i] = Nl''[i]$

PushA4 $H'[Nl'[i]] = H''[Nl''[i]]$

PushA5 $Nl'[i] \in H' \rightarrow Nl''[i] \in H''$
 (a) Assumptions for the Push Process

PopA1 $valid(Top', H') \rightarrow valid(Top'', H'')$

PopA2 $Ss'[i] = Ss''[i]$

PopA3 $Ss'[i] \in H' \rightarrow Ss''[i] \in H''$

PopA4 $Ss'[i] \in H' \rightarrow (H'[Ss'[i]] = H''[Ss''[i]])$
 (b) Assumptions for the Pop Process

Figure 5: Formalization of Assumptions

The formula $valid(Top, H)$ is used as initial condition $Init(C)$ for the system.

4.2 Abstraction Proof

The proof of formula (4) relies on assumptions A_i about the behaviour of other concrete CPUSH and CPOP executions. Finding the right assumptions is the main proof effort for the complete verification.

For a process CPUSH_{*i*} we used a conjunction of PushA1-PushA4 (shown in Figure 5(a)) as assumption A_i . Informally, they have the following informal meaning

PushA1 If the data structure on the heap is valid, it is valid after the environment step.

PushA2 If the allocated cell $Nl[i]$ is not yet integrated into the stack, it will not be in the stack after the environment step.³

PushA3 The environment will not change the variable $Nl[i]$.

PushA4 The environment will not change the content of the cell $Nl[i]$.

PushA5 The environment step will not deallocate the cell $Nl[i]$ points to.

³The predicate $reachable(p, Top, H)$ is true, iff p is a reference of the stack representation reachable from Top

4.3 Modularization 4 LINEARIZATION OF THE STACK ALGORITHMS

Only assumption PushA1 and PushA2 are specific for the both stack algorithms, while the assumptions PushA3 - PushA5 are generic for all algorithms that allocate variables (PushA4 may need generalization in some cases).

As assumption A_i for a process CPOP _{i} the conjunction of the formulas PopA1-PopA4 (depicted in Figure 5(b)) are used. They have the following meaning

PopA1 The same as PushA1.

PopA2 The variable $Ss[i]$ is not changed by the environment.

PopA3 Analogous to PushA5.

PopA4 The content of the cell where $Ss[i]$ points to will not be changed by the environment.

The assumptions PopA2 and PopA3 are generic for processes with local variables. The assumption PopA4 formalizes the fact, that cells are not changed as long as they are part of the stack data structure. This fact is vital for the pop algorithm.

With all these assumptions the push abstraction proof proceeds straight forward by using symbolic execution and induction as described in Section 3. The only interesting step is the execution of the CAS-step, where a case distinction is necessary. In the first case CAS is succesful where the program terminates after 2 additional steps. In the other case, if CAS failed, the program loops and induction can be applied to close this case.

The abstraction proof for the pop algorithm is very similar, even though it is a bit longer as an additional case distinction has to be considered, which discerns whether the stack is empty or not (line 4 in the pop algorithm).

4.3 Modularization

To apply the assumption-guarantee technique described in Section 3, it remains to find guarantees for CPUSH and CPOP, so that premises 2. and 3. of the AG-Theorem hold, i.e. guarantees must hold for their respective component and must imply the assumptions of the other components. For most of the assumptions this task is very straight forward. E.g. the assumption PushA1 and PopA1 can be formulated directly as guarantee (i.e. $valid(Top, H) \rightarrow valid(Top', H')$). For the following two assumptions this task is more difficult:

The first is the statement, that the newly allocated cell $Nl[i]$ of the push algorithm is not changed by another process (assumption PushA4). The guarantees for the pop process are very simple, as pop changes neither H nor any $Nl[i]$. It is easy to show, that CPUSH($v, i; Nl, H$) does not change any other cell than the cell where $Nl[i]$ points to. Therefore, the invariant $\forall i0 \neq i1. Nl[i0] \neq Nl[i1]$ is used, which states that all $Nl[i]$ -cells are always disjoint. With this invariant

a guarantee can be easily formalized, that implies assumption PushA4. Also, guarantees that preserve this invariant can be easily formulated.

Assumption PopA4 is more difficult, because $Ss[i]$ can point to any cell that is or was in the stack representation. We must guarantee that no CPUSH process modifies this cell (in the assignment in line 3 of Fig. 1) to ensure that CPOP extracts the right return value (line 8 of Fig. 2). Therefore, a history variable is used to monitor which cells are or were part of the stack. Formally, we exploit the fact that we have an explicit environment which can record the history in a variable L . A change of the program is unnecessary, we just add

$$\square L'' = L' \cup \text{reachableSet}(Top', H') \quad (8)$$

to the global environment assumptions. $\text{reachableSet}(Top, H)$ is the set of all cells that are reachable from Top . With this new environment assumption, the following guarantee is shown for the CPUSH process.

$$\forall a.a \in L \rightarrow H[a] = H'[a] \quad (9)$$

CPUSH never changes a cell that is or was part of the stack. Therefore, CPUSH never violates the assumption PopA4.

In total, 7 assumptions and 10 guarantees are specified for the push algorithm and 5 assumptions and 7 guarantees are used for the pop process. Most of the additional assumptions and guarantees are needed to formalize properties of the history described above. Compared to the effort to find and specify all assumptions and guarantees, the proof effort for the assumption-guarantee proofs is relative low, as many proofs require only little interaction.

5 Related Work

Verification of lock-free algorithms is currently an active research topic. Various algorithms have been proven correct, e.g. algorithms working on a global queue [18], [19], a lazy caching algorithm [20] or a concurrent garbage collection [21].

The algorithm considered here was taken from Colvin and Groves [18, 22], who have given a correctness proof using IO automata and the theorem prover PVS. We have only studied the core algorithm, their work also discusses extending the algorithm with elimination arrays, and 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.

Recent work by the same authors [12, 13] discusses incremental development of the algorithm using refinement calculus and programs very similar to ours. 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 [11] (in particular to Dynamic Logic [23] 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 (e.g. PushA4, PopA4) is to answer the question, why processes cannot modify or access certain locations. Answers to these questions are only given informally in [12].

In [17], Vafeiadis et. al. describe a rely-guarantee approach, that is similar to ours. The approach is applied informally on an implementation of sets using fine-grained locking. [24, 25] extends the approach by using a specialized separation logic and by providing tool support. The approach is specialised to reasoning over pointer structures and therefore has stronger automation than ours.

Fully automatic approaches based on static analysis are given by Amit et. al. in [26] and by Berdine et. al. in [27]. They are also specialised on reasoning over pointer structures.

The second author of this paper has also contributed to [28] and [29], 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 [28] and using program counters in [29]. 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 used as a running example, but the complexity of proofs is much lower than the ones given here, since the concrete level does not use a global heap. [29] explicitly proves that the example satisfies the original criterion of linearizability that was defined in Herlihy and Wing's original paper [7]. Linearizability is only implicitly implied by this and all other related work. In future work we plan to connect the proofs done here to the explicit formal definition of linearizability. The explicit history of events needed to do this should be definable exploiting the explicit environment as we did for the *reachableSet* predicate in Section 4.3.

6 Conclusion

In this paper we have developed a proof technique for verifying refinements of abstract data types to interleaved algorithms. The standard example of Treiber's stack could be verified with this technique. Both, data refinement and decomposition using assumption-guarantee reasoning were expressed using the temporal logic available in KIV. A suitable modularization theorem was proved for this purpose.

We see the following advantages in our approach: first, the tool support gives 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. An additional translation to a special normal form (as e.g. in TLA [30]) using explicit program counters is not necessary. The proof strategy of symbolic execution in KIV is very intuitive and can be automated to a large extent. In this paper we have also shown how to decompose proofs; instead of a single large proof it was sufficient to construct several small and comprehensible proofs for single processes.

As future work this approach opens several interesting possibilities. For the compositional verification of the lock-free algorithm, we have verified a number of interesting properties of more general nature, e.g., properties concerning memory allocation. Can these properties be reused for the verification of other algorithms? Another direction we want to investigate is to examine other properties for lock-free algorithms, e.g. liveness properties. In KIV, liveness properties can also be verified with symbolic execution and induction. It remains to incorporate a suitable modularization theorem into the calculus of KIV.

References

- [1] Dijkstra, E.W.: Solution of a problem in concurrent programming control. *Commun. ACM* **8**(9) (1965) 569
- [2] Jones, C.B.: Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* **5**(4) (1983) 596–619
- [3] Misra, J., Chandi, K.: Proofs of networks of processes. *IEEE Transactions of Software Engineering* (1981)
- [4] de Roever, W.P., et al.: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press (2001)
- [5] Abadi, M., Lamport, L.: Conjoining specifications. *ACM Transactions on Programming Languages and Systems* (1995)
- [6] Cau, A., Collette, P.: Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Inf.* **33**(2) (1996) 153–176
- [7] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* **12**(3) (1990) 463–492
- [8] Treiber, R.K.: *System programming: Coping with parallelism*. Technical Report RJ 5118, IBM Almaden Research Center (1986)
- [9] Balsler, M.: *Verifying Concurrent System with Symbolic Execution*. Shaker Verlag, Germany (2006)

-
- [10] Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
- [11] Reif, W., Schellhorn, G., Stenzel, K., Balsler, 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
- [12] Groves, L., Colvin, R.: Derivation of a scalable lock-free stack algorithm. *Electron. Notes Theor. Comput. Sci.* **187** (2007) 55–74
- [13] Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. *Formal Aspects of Computing (FAC)* (2008) (to appear).
- [14] Balsler, M.: Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction. PhD thesis, University of Augsburg, Augsburg, Germany (2005)
- [15] Balsler, M., Reif, W.: Interactive verification of concurrent systems using symbolic execution. Technical Report 2008-12, Universität Augsburg (2008)
- [16] Bäumlner, S., Nafz, F., Balsler, M., Reif, W.: Compositional proofs with symbolic execution. In Beckert, B., Klein, G., eds.: Proceedings of the 5th International Verification Workshop. Volume 372 of *Ceur Workshop Proceedings*. (2008)
- [17] Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPOPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2006) 129–136
- [18] 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
- [19] Abrial, J.R., Cansell, D.: Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). *Journal of Universal Computer Science* **11**(5) (2005) 744–770
- [20] Hesselink, W.H.: Refinement verification of the lazy caching algorithm. *Acta Inf.* **43**(3) (2006) 195–222
- [21] Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free parallel and concurrent garbage collection by mark&sweep. *Sci. Comput. Program.* **64**(3) (2007) 341–374
- [22] Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. *ENTCS* **137** (2005) 93–110

-
- [23] Harel, D.: Dynamic logic. In Gabbay, D., Guenther, F., eds.: Handbook of Philosophical Logic. Volume 2. Reidel (1984) 496–604
- [24] Calcagno, C., Parkinson, M.J., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In Nielson, H.R., Filé, G., eds.: SAS. Volume 4634 of Lecture Notes in Computer Science., Springer (2007) 233–248
- [25] Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In Caires, L., Vasconcelos, V.T., eds.: CONCUR. Volume 4703 of Lecture Notes in Computer Science., Springer (2007) 256–271
- [26] Amit, D., Rinetzkly, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007) 477–490
- [27] Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: CAV’08: 20th International Conference on Computer Aided Verification, Springer (2008)
- [28] Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizability via non-atomic refinement. In Davies, J., Gibbons, J., eds.: IFM. Volume 4591 of Lecture Notes in Computer Science., Springer (2007) 195–214
- [29] Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanising a correctness proof for a lock-free concurrent stack. In: Proceedings of FMOODS 2008, Oslo. LNCS (2008)
- [30] Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16**(3) (1994) 872–923