Science of Computer Programming 96 (2014) 297–314

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Two approaches for proving linearizability of multiset

Bogdan Tofan [a,*], Oleg Travkin [b], Gerhard Schellhorn [a,*], Heike Wehrheim [b]

[a] Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
[b] Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany

HIGHLIGHTS

- Two approaches for verifying linearizability.
- Mechanized verification of local proof obligations.
- Challenging case study of a multiset implementation with fine-grained locking.
- Rely–Guarantee reasoning for a Temporal Logic with Programs.

ARTICLE INFO

Article history:
Received 15 March 2013
Received in revised form 27 March 2014
Accepted 7 April 2014
Available online 19 April 2014

Keywords:
Linearizability
Concurrent data types
Interactive verification

ABSTRACT

Linearizability is a key correctness criterion for concurrent software. In our previous work, we have introduced local proof obligations, which, by showing a refinement between an abstract specification and its implementation, imply linearizability of the implementation. The refinement is shown via a process local simulation. We have incorporated the approach of verifying linearizability based on refinement in two rather different proof systems: a predicate logic based approach performing a simulation for two processes and second, an approach based on temporal logic that shows a refinement for an individual process using rely–guarantee reasoning and symbolic execution. To compare both proof techniques, we use an implementation of a multiset as running example. Moreover, we show how ownership annotations have helped us to reduce the proof effort. All proofs are mechanized in the theorem prover KIV.

1. Introduction

With an increasing number of cores per CPU, data structures like lists or sets can be used more and more concurrently. To avoid bottlenecks in a system, such data structures must be designed for maximizing throughput. This is done by applying fine-grained synchronization schemes or avoiding the use of locks at all and using hardware primitives for synchronization instead. Due to the fine granularity of synchronization and interleaving of instructions, verifying such data structures to be correct is hard.

Reasoning about correctness of concurrent data structures usually means reasoning about their linearizability [1]. Linearizability is a safety property. Data structure implementations are linearizable, if for each concurrent execution there exists a corresponding sequential execution, such that the order of non-concurrent executions of operations is preserved. Operations of linearizable data structures "seem to take effect instantaneously at some point in time between their invocation and response". This point is referred to as the linearization point (LP) or sometimes also as the commit point. The LP does

* Corresponding authors.
E-mail addresses: tofan@informatik.uni-augsburg.de (B. Tofan), oleg82@upb.de (O. Travkin), schellhorn@informatik.uni-augsburg.de (G. Schellhorn), wehrheim@upb.de (H. Wehrheim).

http://dx.doi.org/10.1016/j.scico.2014.04.001
0167-6423/

not necessarily have to be a fixed location in the code. Sometimes, the LP is even outside of the operation, i.e., an operation is linearized by another concurrently running operation.

Proving a data structure to be linearizable typically involves showing a refinement relation between an abstract specification of a data structure and its actual implementation. An abstract specification contains atomic operations only. To show that an implementation is linearizable w.r.t. its abstract specification, one has to prove that the implemented operations non-atomically refine the operations from the abstract specification.

Several different approaches exist that are used to prove linearizability, such as shape abstraction [2], rely–guarantee reasoning and separation logic combined [3] and our own two simulation-based approaches [4,5]. We formally defined a general theory relating refinement theory and linearizability following the notion of Herlihy and Wing [1]. From our general theory, we could derive proof obligations that are process local and, once proved for a concrete data structure implementation, imply linearizability of the implementation. While our first approach [4] is based on predicate logic (PL) and performs a simulation for two processes, our second approach [5] is based on temporal logic (TL) and rely–guarantee reasoning, and shows refinement via symbolic execution of a single process. Both approaches perform a local proof for two (resp. one) processes and make it possible to conclude proof results for arbitrary numbers of processes.

In our ongoing work, we aim to determine strengths and weaknesses of both approaches. As a case study, we use a multiset implementation proposed by Elmas et al. which they verified linearizable [6]. It is an interesting case study that poses several problems:

(1) The operation *InsertPair* adds two elements to the multiset using several non-atomic instructions. Thus, finding an abstraction function for a refinement proof can be challenging for such an operation as Elmas et al. already pointed out. It seems to be counter-intuitive to have exactly one location for the LP, since element insertion is implemented by two instructions. (2) It includes a *LookUp* operation with a LP that is not statically fixed. In such cases usually a backward simulation over the full program state is required. We could avoid this by using our theory for potential linearization points [7]. Moreover, we were able to apply a local temporal logic refinement proof as well, which does not use backward simulation at all. (3) The *Delete* operation has potential linearization points. In contrast to *LookUp*, a *Delete* manipulates the state of the data structure and while the manipulation might be visible to some processes it might be hidden to others. This interplay of processes grows heavily in complexity and it becomes challenging to prove linearizability with a *Delete* operation. In fact, the linearizability proof by Elmas et al. and our own proofs do not consider the *Delete* operation, because all of us initially assumed it would not be linearizable. However, neither Elmas et al. nor we were able to provide an adequate counterexample.

This paper extends our previous work [8]. In particular, it gives new solutions for problems (1) and (2) for the multiset. Our previous verification faithfully sticks to the original code of [6]. When we applied the temporal logic approach, we were inspired by their proof scripts, which use ownership annotations, and also added such annotations to the code that we verified. This had a positive effect on both the specification and the proofs for the multiset. As we explain later on, it helped us to significantly simplify the rely–guarantee proof obligation, the invariants and the abstraction function, as well as to reduce the verification effort. We used the interactive theorem prover KIV [9] to formalize the implementation, the abstract specification as well as to mechanize all our linearizability proofs.

First, we present a PL-based linearizability proof for the multiset, which was initially published in [8]. The multiset, the encoding of its abstract specification, and its implementation are introduced in Section 2. The abstraction function that we used for the PL-based refinement proofs is presented in Section 3. We give a brief overview of our local proof obligations in Section 4, followed by an explanation of the invariants used throughout the proofs in Section 5. Second, we present our TL-based proof, where Section 6 outlines the underlying temporal logic. The proof obligation for linearizability based on rely–guarantee reasoning is introduced in Section 7, along with a high-level description of the proofs. The third part, compares our approaches and discusses related work in Section 8. Finally, we conclude in Section 9.

2. The multiset case study

As a case study for our local theory we have chosen an implementation of an integer multiset from [6]. The parts of the implementation we considered in our proof are the three operations presented in Fig. 1 as Elmas et al. published it. We elaborate on the multiset challenges in Section 2.3 in more detail.

The multiset is implemented as an array M containing elements of type *Slot*. A slot encodes its value by the integer attribute *elt*. A second attribute *stt* of type *Status* encodes the state of insertion. A slot can be *empty*, *reserved* or *full*. An integer i is considered to be in the multiset if there is a slot with element i and status *full* in the array M. The *LookUp* operation tests whether a particular value is contained in the set. By traversal of the array, it tests (at location $L3$) each element for equality to the parameter value and being in *full* state. A slot is locked before the test is performed and released afterwards to prevent other processes from modifying the slot while it is tested. Elements are inserted pairwise in the *InsertPair* operation. *InsertPair* calls the *FindSlot* operation to reserve empty slots. Actually, *FindSlot* is implemented similarly to *LookUp* by means of an array traversal, but we use the algorithm from [6] here for brevity and assume it to be atomic. Atomicity of *FindSlot* ensures that two concurrent *FindSlot* calls always reserve different slots. If *FindSlot* cannot find