Verifying Optimistic Algorithms Should be Easy
(Position Paper)

Noam Rinetzky
Queen Mary University of London

Martin T. Vechev Eran Yahav Greta Yorsh
IBM T.J. Watson Research Center

Abstract

In this paper, we call to bridge the gap between what makes highly-concurrent optimistic algorithms work and current approaches for proving their correctness.

The Problem: Verification of Optimistic Concurrent Algorithms

Highly-concurrent optimistic algorithms are notoriously hard to verify. In particular, verifying that an optimistic algorithm is linearizable [3] is quite challenging. (See, e.g., [9]). Given a highly-concurrent algorithm, our goal is to find a proof that captures its designer's intuition as to why the algorithm works. We believe that simple and intuitive proofs can, and should, be obtained by embracing the *spirit* in which these algorithms are written.

In this paper, we show that the intuition behind many optimistic concurrent algorithm can be naturally captured using global invariants,[1] à la Lamprot [4], of a *particular* class: In this class, observations regarding the local state of a thread are completely separated from observations regarding either the local states of other threads or the global state.

What Makes Highly-Concurrent Optimistic Algorithms Work? A distinguishing feature of optimistic algorithms is that every thread makes very little assumptions on the environment in which it operates. A thread can rely on a structural invariant of the global state, but it cannot rely on local properties of other threads. A thread operates by checking a *local property* to establish the validity of an update before it takes place. The local property concerns only its local variables and a small fraction of the global shared memory. When the local property does not hold, indicating that the desired update might lead to a violation of the structural invariant, the thread has the ability to "rollback" its actions, and restart the operation. This approach allows the thread to maintain safety under any environment (possibly by sacrificing progress).

A Motivating Example. Fig. 1 shows an optimistic set algorithm. The algorithm is one of the concurrent set algorithms derived in [10]. The code is instrumented with operations that manipulated the set's abstract value. (The instrumentation, written in italics, is explained in Example 3.)

The set is implemented as a sorted singly-linked linked list with designated sentinel *Head* and *Tail* nodes. The *Head* node holds the smallest possible key, denoted $-\infty$, and the *Tail* node holds the largest possible key, denoted ∞. For simplicity, we illustrate our approach using only two set operations: `add` and `remove`, with their standard meaning. (We note that, although omitted, we can handle the `contains` operation). The `key` argument to these operations, supplied by the client, must be strictly larger than $-\infty$ and strictly smaller than ∞.

Both `add` and `remove` use the macro `LOCATE` to traverse the list and locate an item based on the value of its key. The list traversal performed by `LOCATE` is optimistic and is done without any form of synchronization.

[1] In this paper, we use the term "global invariant" as "global within a the context of the algorithm", i.e., an invariant concerning the shared resources used to implement the verified data structure, and not as an invariant concerning the whole state.

```
typedef struct E{
  int key;
  struct E *next;
  boolean marked;
} Entry;

LOCATE(pred, curr, key){
  pred := Head
  curr := Head->next
  while(curr->key < key){
    pred := curr
    curr := curr->next
  }
}
```

```
boolean remove(int key) {
  Entry *pred,*curr,*r
restart_remove:
  LOCATE(pred; curr; key)
  atomic{
    if (curr->key == key){
      curr->marked := true
      r := curr->next
      b := ! pred->marked
      if ( (pred->next==curr) && b){
        pred->next := r
        res := (k ∈ Abs)
        Abs := Abs \ {k}
        assert (res == true)
        return true
      } else { goto restart_remove}
    } else { return false }
  }
}
```

```
boolean add(int key){
  Entry *pred,*curr,*entry
restart_add:
  LOCATE(pred; curr; key)
  atomic{
    if(curr->key != key){
      entry := new Entry(key)
      entry->next := curr
      b := !pred ->marked
      if ( (pred->next==curr) && b){
        res := (k ∉ Abs)
        Abs := Abs ∪ {k}
        pred->next := entry
        assert (res == true)
        return true
      } else { goto restart_add}
    } else { return false }
  }
}
```

Figure 1: The (instrumented) code of an optimistic set implemented using atomic sections.

Once `LOCATE` finds the position for the desired update, the operation attempts to apply this update atomically. For simplicity, the update is implemented using an atomic section. Note that as a result, interference can occur during, and after, the list traversal but not during the update.

Verification Challenges

Interference. One of the main challenges in verifying concurrent programs is the need to reason about interference. Interference cannot be ignored due to the manipulation of shared resources by different threads. The challenge is to find a way to reason about interference in a simple, yet useful, way. More specifically, the challenge is to find a way which allows us to make as coarse assumptions about the environment as possible, while still being able to successfully prove the desired properties.

Most existing approaches make in their reasoning distinctions about the state that cannot be observed by the executing threads. We find that as a result of this practice, the proofs become less intuitive and needlessly complicated. We believe that it is both possible and desirable to avoid making such distinctions in the proofs.

Thread-Local Linearization Points. Current approaches focus on finding a linearization point for every operation, that is, the point in which the operation "seems to take effect instantaneously". This point occurs between the time the operation is invoked and the time the operation terminates. This approach yields quite natural and simple proofs when the linearization points of an operation are *thread-local*, i.e., can be determined by the local state of the thread performing the operation. (Thread-local linearization points are often referred to as *fixed* linearization points). Usually, linearization points of this kind are easy to identify and typically correspond to a statement in the code of the operation which performs a global update.

Example 1 The linearization points of *successful* `add` and `remove` operations, i.e., operations which return `true`, are *thread-local*: They can be associated with the destructive update of `pred->next`.

Thread-Global Linearization Points. The proof of linearizability becomes much more demanding when the linearization points are *thread-global*, i.e., can be determined only be examining the local state of several threads or the shared state. (Thread-global linearization points are often referred to as *non-fixed* linearization points). This type of linearization points is very common in optimistic concurrent algorithms, especially in operation which are read only.

Example 2 The linearization points of *unsuccessful* `add` and `remove` operations are *thread-global*.

The need to consider non thread-local properties in order to determine the linearization point of an operation is particularly frustrating when we consider the rather simple *atomic* observations and permutations that an operation can make. Intuitively, in many cases, every atomic mutation done by an operation can be simulated by an `MCAS` operation. Thus, although the algorithm manages to fulfill its task correctly using very local and limited observations, the assertions used in a formal proof requires much stronger observational power. Indeed, and perhaps unsurprisingly, all existing approaches for *automatic verification* of linearizability, e.g., [1, 2, 5, 8], are limited to verifying algorithms with fixed linearization points.

Our Goal: Simple Verification of Optimistic Concurrent Algorithms

We believe that it is possible to simplify the verification of optimistic concurrent algorithms by limiting the observation in the assertions to a stylized form which separates the thread-local observation from a global invariant regarding the shared state. (These invariants are a special case of the invariants suggested by Lamport [4]). We suggest to overcome the challenge of verifying operations with thread-global linearization points by using non-constructive proofs. Specifically, *instead of showing the linearization points of every operation which occur in the concurrent trace, we suggest to only show their existence*.

We believe that proofs in the spirit of our approach would be simpler and more intuitive than existing proofs and easier to automate. We also believe that they will provide better insight into the way that the algorithm works, thus allowing us to use the proofs as a way to understand the intentions of the algorithm designer.

Stable-by-Construction Global Invariants. We ensure that our assertions are stable under interference by restricting them to have two *separate* parts: a *thread-local* invariant and a *global* invariant. The thread-local invariant records correlation between local variables of single threads. Specifically, the it is not allowed to relate the local variables of several threads or to observe any mutable part of the heap. The global invariant correlates *only* relationships in the global heap. Specifically, it is not allowed to observe any mutable part of the local state of threads.

More technically, we require that all assertions pertaining to statements occurring *outside* of an atomic section be of the form $I_G \wedge I_L$. The *global assertion* I_G is a formula over global variables and fields of objects. The local assertion I_L is a formula over the local variables of a thread, its program counter, and *immutable* global variables.

The key observation is that assertions that separate global state from thread-local states are stable under interference. That is, the (determined) effect of executing the operations concurrently is the same as executing each one by itself.

Example 3 The following properties are expressible in the restricted form that we define and hold for the running example. We use Abs to denote the set's abstract value. The latter is represented by the set of keys in the nodes which are reachable from the head of the list. (I.e., Abs can be determined by a function from the current global state. See property φ_6).

$\varphi_1 \stackrel{\text{def}}{=} \forall v.n^*(v,T)$	every node reaches tail
$\varphi_2 \stackrel{\text{def}}{=} \forall v.\neg n^+(v,v)$	acyclic
$\varphi_3 \stackrel{\text{def}}{=} \forall k \in Keys.H.k < k < T.k$	head and tail keys
$\varphi_4 \stackrel{\text{def}}{=} \forall v.\neg v.m \Rightarrow n^*(H,v)$	fumble
$\varphi_5 \stackrel{\text{def}}{=} \forall v_1,v_2.(v_1.n = v_2) \Rightarrow v_1.k < v_2.k$	sorted
$\varphi_6 \stackrel{\text{def}}{=} \forall k.(k \in Abs) \iff (\exists v.n^*(H,v) \wedge v.k = k)$	representation

Discussion. Verification using our restricted form of invariants can be related to the works Lamport [4], Owicki-Gries method [6], Reynholds [7]: Following Lamport, we advocate using global invariants. However, our approach can also be viewed as a special (and simple) case of the Owicki-Gries method [6] in which interference-freedom is guaranteed by the special form of the invariants: The global part is (by construction) stable under interference. The local part is also stable under interference, as one thread cannot modify (or even observe) the local variables of another thread. (Specifically,the soundness proof of our approach uses Owicki-Gries method [6] to show the absence of interference.) In addition, our restrictions on the form of the assertions can be viewed as a way to enforce a syntactic control of interference [7].

The Road Ahead

We believe that our approach can be used to verify an entire family of optimistic concurrent algorithms, using no more than the small set of invariants shown in Example 3, with a few minor adaptations. In the near future, we plan to put this belief to the test. We note that if our belief is found to be correct, then our approach would have an additional benefit by revealing hidden commonalities between different algorithms. We have a *proof sketch* for the correctness of our approach, and we plan to complete it to produce a formal proof.

Our longer term goals are (i) extend the approach to handle progress properties and (ii) use our approach as a basis for an automatic verification tool.

Acknowledgements. We are grateful for the encouragement and insights we gained from fruitful discussions with Peter O'Hearn.

References

[1] AMIT, D., RINETZKY, N., REPS, T. W., SAGIV, M., AND YAHAV, E. Comparison under abstraction for verifying linearizability. In *Computer Aided Verification* (2007), pp. 477–490.

[2] BERDINE, J., LEV-AMI, T., MANEVICH, R., RAMALINGAM, G., AND SAGIV, S. Thread quantification for concurrent shape analysis. In *Computer Aided Verification* (2008), pp. 399–413.

[3] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *Transactions on Programming Languages and Systems 12*, 3 (1990).

[4] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (1977), 125–143.

[5] MANEVICH, R., LEV-AMI, T., SAGIV, M., RAMALINGAM, G., AND BERDINE, J. Heap decomposition for concurrent shape analysis. In *Static Analysis Symposium* (2008), pp. 363–377.

[6] OWICKI, S., AND GRIES, D. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM 19*, 5 (1976), 279–285.

[7] REYNOLDS, J. C. Syntactic control of interference. In *ACM Principles of Programming Languages* (1978), pp. 39–46.

[8] VAFEIADIS, V. Shape-value abstraction for verifying linearizability. In *Verification, Model Checking, and Abstract Interpretation* (2009), pp. 335–348.

[9] VAFEIADIS, V., HERLIHY, M., HOARE, T., AND SHAPIRO, M. Proving correctness of highly-concurrent linearisable objects. In *Principles and Practice of Parallel Programming* (2006).

[10] VECHEV, M. T., AND YAHAV, E. Deriving fine-grained concurrent linearizable objects. In *ACM Programming Languages Design and Implementation* (2008).