

Some techniques for proving linearizability

Viktor Vafeiadis
Microsoft Research

Linearizability & linearization points

Linearizability *is* atomicity.

A module is linearizable iff all its operations are atomic.

Linearizability *is (essentially)* refinement.

Every operation has a point during its execution at which it appears to take place atomically.

For every terminating execution of an operation, there exists a *linearization point* (LP).

Basic proof technique:

Provide a witness to this existential.

Proving linearizability – The LP method

1. Specify the linearization point in the program.
 - There can be multiple points (each on a different exec. path).
 - These points need not be in the source code of the operation itself: they can be in the code of other concurrent operations.
2. Check that the specified point is indeed a linearization point:
 - Maintain two data structures: concurrent (***concrete***) and sequential (***abstract***).
 - Compare each concurrent execution to a specific abstract execution.
 - The annotated points occur ***at most once*** in every execution.
 - The annotated points occur ***exactly once*** in every ***terminating*** execution.
 - Each concrete operation returns ***the same result*** as its abstract counterpart.

Proving linearizability – The LP method

1. Specify the linearization point in the program.

Can we find heuristics to automate this?

2. Check that the specified point is indeed a linearization point:

Automatic:

Shape-value abstraction for verifying linearizability,
VMCAI 2009.

counterpart.

g

act

Example 1


Treiber's non-blocking stack [1986]

Treiber's non-blocking stack

```
push(v) {
```

```
1. Node t, x;  
2. x := new Node();  
3. x.val := v;  
4. do {  
5.   t := Top;  
6.   x.next := t;  
7. } while(¬CAS(&Top, t, x));  
}
```

```
pop() {
```

```
1. Node t, x;  
2. do {  
3.   t := Top;  If Top==NULL  
4.   if (t == NULL)  
5.     return EMPTY;  
6.   x := t.next;  
7. } while(¬CAS(&Top, t, x));  
8. return t.val;
```

If the CAS succeeds

Linearisation points:

- **push:** CAS, if it succeeds
- **pop:** CAS, if it succeeds; reading Top if it returns NULL.

Identifying candidate LPs (1/3)

→ Identify a LP by an atomic command.

- Normally, LPs are points in time when the effect occurs.
- Identify them by a point in the program which does the effect.

Heuristic 1:

For LPs, consider atomic commands appearing in the body of the method being verified.

Justification:

- In general, the LP can be part of another concurrent operation;
 - ... but this does not happen very often for operations with side-effects,
 - ... and such LPs are very difficult to find automatically.
- Therefore, only search for LPs appearing inside the code being verified.

Identifying candidate LPs (2/3)

Heuristic 2:

Consider only accesses to shared memory locations.

Justification:

- At the LP, the entire effect of the method “happens” atomically.
- Therefore, it must be a communication point.

Implementation trick:

- First do a memory safety proof to find out which accesses are to shared memory locations.

Treiber's non-blocking stack

```
push(v) {  
    Node t, x;  
    x := new Node();  
    x.val := v;  
    do {  
        t := Top;  
    } while( $\neg$ CAS(&Top, t, x));  
}
```

```
pop() {  
    Node t, x;  
    do {  
        t := Top;  
        if (t == NULL)  
            return EMPTY;  
        x := t.next;  
    } while( $\neg$ CAS(&Top, t, x));  
    return t.val;  
}
```

 Accesses to shared memory

Identifying candidate LPs (3/3)

Split executions to *effectful* and *effect-free*.

Heuristic 3 (for effectful executions):

Consider only writes to shared memory locations.

Justification:

- At the LP, the entire effect of the method “happens” atomically
- Therefore, it must be a communication point : usually a *write* to a shared memory location.
 - It can also be a shared read (very rare, so ignore).

Treiber's non-blocking stack

```
push(v) {  
  Node t, x;  
  x := new Node();  
  x.val := v;  
  do {  
    t := Top;  
    x.next := t;  
  } while( $\neg$ CAS(&Top, t, x));  
}
```

```
pop() {  
  Node t, x;  
  do {  
    t := Top;  
    if (t == NULL)  
      return EMPTY;  
    x := t.next;  
  } while( $\neg$ CAS(&Top, t, x));  
  return t.val;  
}
```

-  Reads from shared state
-  Writes to shared state


Example 2


A lazy concurrent list-based set algorithm
[Heller et al. '05]

Lazy concurrent list-based set [Heller et al '05]

- Set implemented as a sorted linked list with two sentinel nodes.
- Each node has a lock & a marked bit.
- Optimistic traversal:
 - Traverse the list without acquiring locks.
 - Validate the traversal was okay.
- Deletions in two steps:
 - Mark the node – *logical deletion*
 - Physically remove it from the list.
- Important properties:
 - The list is always well formed:
 - Nodes appear in sorted order
 - Head and tail nodes with values $-\text{INF}$ and $+\text{INF}$ respectively
 - If a node was once in the list and it is NOT marked, then it is still in the list.

What are the linearization points?

```
add(k) {  
1.  pred, curr := locate(k);  
2.  if curr.key != k then {  
3.      entry := new Entry();  
4.      entry.key := k;  
5.      entry.next := curr;  
6.  pred.next := entry;  
7.      res := true  
8.  } else {  
9.      res := false;  
10. }  
11. pred.unlock();  
12. curr.unlock();  
13. return res;  
}
```

```
remove (k) {  
1.  pred, curr := locate(k);  
2.  if curr.key == k then {  
3.  curr.marked := true;  
4.      entry := curr.next;  
5.      pred.next := entry;  
6.      res := true  
7.  } else {  
8.      res := false;  
9.  }  
10. pred.unlock();  
11. curr.unlock();  
12. return res;  
}
```

What are the linearization points?

```
contains(k) {  
1.  curr := Head;  
2.  while curr.key < k do {  
3.    curr := curr.next;  
4.  }  
5.  if (curr.key == k  
6.    && !curr.marked) then {  
7.    return true;  
8.  } else {  
9.    return false;  
10. }  
}
```

(Hint: Consider concurrent remove operations.)

Effect-free executions (1/2)

- LP is often in code of another thread:
 - Searching for candidate LPs is too difficult/expensive.
 - Give up!
- Happily, checking that a candidate LP is correct is easier.
 - *As before*, maintain two data structures: concurrent (**concrete**) and sequential (**abstract**).
 - Compare each concurrent execution to a specific abstract execution.
 - Since the abstract operation has no side-effects, it is safe to execute it multiple times.
 - For each concrete effect-free execution, there exists an invocation to its abstract counterpart during its execution that returns the same result.

A hacky verification method:

Accumulate the set of results that a linearizable effect-free execution is allowed to return.

- Start with the empty set.
- At each point during symbolic execution, enlarge this set with the return value of executing the abstract operation.
- At the end, check that the concrete return value is in that set.

(Work in progress)

Example 3

RDCSS [Harris et al. '05]

Restricted double-compare single-swap [Harris et al.]

- Address space partitioned to type 1 and type 2.
 - On type 1 addresses, normal reads & writes allowed.
 - On type 2 addresses, only RDCSS commands allowed.
- RDCSS takes a descriptor whose a1 field is a type 1 address and whose a2 field is a type 2 address and does the following:

```
RDCSS_spec (d) {  
    atomically {  
        local r := [d.a2];  
        if [d.a1]==d.o1 && [d.a2]==d.o2 then  
            [d.a2] := d.n2;  
        return r;  
    }  
}
```

Restricted double-compare single-swap

Complete(d) {

1. **local** r := [d.a1];
2. **if** r==d.o1 **then**
3. CAS1(d.a2, d, d.n2);
4. **else**
5. CAS1(d.a2, d, d.o2);
6. }

RDCSS_read (a2) {


1. **local** r := [a2];
2. **while** IsDesc(r) **do** {
3. Complete(r);
4. r := [a2];
5. }

RDCSS (d) {

1. **local** r;
2. r := CAS1(d.a2, d.o2, d);
3. **while** IsDesc(r) **do** {
4. Complete(r);
5. r := CAS1(d.a2, d.o2, d);
6. }
7. **if** r==d.o2 **then**
8. Complete(r);
9. **return** r;
10. }

Restricted double-compare single-swap

```
Complete(d) {  
1.  local r := [d.a1];  
2.  if r==d.o1 then  
3.    CAS1(d.a2, d, d.n2);  
4.  else  
5.    CAS1(d.a2, d, d.o2);  
}
```

LP is the  read, if and only if the future  CAS succeeds.

→ Use a *prophecy variable* to specify the LP.

→ Very difficult to automate the search for this kind of LPs.

→ Yet, it is quite a simple example:

- The descriptor acts as a lock for conflicting type 2 operations.
- The read `r:= [d.a1]` synchronizes with type 1 operations.

Summary & Conclusions

1. The main part of a linearizability proof is finding the LP. The rest is comparatively easy.
2. Introducing auxiliary state enables proofs ...
... but the proofs are ugly, long, & difficult to automate.
3. Some hacks possible for effect-free methods.

Practical conclusion:

Automation works very well; it can handle almost all practical cases.

Theoretical conclusion:

We need a better theory of atomicity/linearizability.

- Too much auxiliary state for contains & RDCSS.