Homework 2: Queue

Program verification

Tanja de jong, Niek Haarman, Tinus Pool

July 2, 2015

1 Problem with permissions at two sides:

While working at the sequential exercises we did want to hold the permission stick on both ends. The problem however is that you can only release one end of the stick at a time, while you have to release both ends to make the stick float in the air. Only when the stick floats in the air you can adjust it.

This is a problem we encountered in our first try with the queue. We had a node last and first and the resource state required the next state of the node. This way we created a chain from the first till the last. So if you wanted to edit the last you where not allowed because last permission was also locked by the chain created by implied by the first state.

This was easily solved by just removing the last, and following the chain till the end to change the last element of the queue. An other possible option was creating all the nodes in a factory and watch the permissions in this factory. However because this concept had some troubles implementing at first and the other implementation was faster implemented, we decided to not use a factory but remove the last node. We still could argue what solution would be better because they both have there nice things and ugly things. Another reason why the factory was not added in the end is because Vercors can be a bit fragile in times. We did not want to burn our fingers when there is already something workable.

2 Concurrent implementation

The problem we encountered in the concurrent implementation was kind like this: How to build in two locks in queue that was standing for two different things in the queue? This problem is created because you can only implement one lock_invariant resource in queue. We solved this problem at first with a kind of symbolic_locks. We where not actually locking anything, but because the implementation would be consistent in the usage and a kind of barrier of violating what we would actually wanting to lock, it was a good start and in our opinion valid for the moment. One symbolic locks protects that we will not take elements that are not there, the other protect the queue for not having more elements then it's capacity.

In this way we where able to write all the implementations for the methods. However for inserting and removing objects we used the same method as in the sequential implementation. The sequential implementation required to have the full permission over the queue and it's node. So that implementation is like one stick or chain that is only allowed to be held by one hand. In concurrent programs however you have to be able to hold the same side of the stick/chain and to be able to hold it with multiple hands.

Luckily for us the lock_invariant is something magical that spawns invisible hand and holds up parts of the chain. So in state of creating the chain by saying that you also want to ensure the next state, you can now only know the next, and the lock_invariant of that node will know it's own state. The lock_invariant will preserve the state, so you don't have to save that state by letting remember the next node's state. This enabled us to introduce the head and last node again. The queue became a thing where we only want to ensure that it has a value. We don't need write permission of the queue itself to change elements. The symbolic locks will be changed to elements that know the queue instance. The one will lock the first, the other will lock the last. By locking the first, you can not remove elements from the queue. By locking the last we will protect adding new elements.

3 Other problems encountered:

- Cannot test with java, because java doesn't recognize seq¡..¿ as data type.
- Need a lockAndUnlock action for some methods, to create a kind of atomic get. In our atomic integer we where not allowed to use our get in verifying our code, because a normal method might change code.
- Didn't entirely grasp why you sometime have to assert that a value not is null after you you did Value(obj).
- Have to catch return value's. Making this to a habit makes people doubt you're developer skills.
- Error messages for not catching return value's and using reserved variables can be confusing and hard to discover if you did write lots of code in a flow.
- assigning null to a return or variable was troubling.

It is possible that along the way some of the problems mentioned above where solved.

4 Things we still could do to verify concurrent code?

We could create local data storage objects and store snapshots of situations in this local data storage. Based on what is inside those snapshots you could argue what was happening inside the concurrent code and based on those things what

the end result could be. However because the border between what is used to verify code in PVL and the actual code is already fragile, we didn't want to push the developer by giving an extra object to a method what only would be used to verify the code instead of helping to get the actual result. Maybe it would be nice to create a subtiele border between what is used to verify code and the actual functionality of the code. In stead of a history, you could give a local box that would help reason about what happend inside the code. History would probably be easier to use and less complex to evaluate what would be the end result.