Program Verification
Homework Assignment 2
192114300

Specification and Verification of Data Structures

Deadline: Friday July 3, 10:00 CET, 2015
Report to be handed in by mail to Marieke Huisman
(M.Huisman@utwente.nl)
Presentation to be given at the deadline, room to be announced.

Introduction

Some practical information concerning the assignment:

- The assignments are to be done in groups.
- The deadline for the homework is **Friday July 3, 10:00 CET, 2015**.
- You should hand in a report describing your solution, approach taken, and distribution of work between the group members. Also submit a zip-file with all your annotated programs.
- The report and the zip-file should be submitted by mail to Marieke Huisman, M.Huisman@utwente.nl.
- **July 3, 10:00 am** the groups will give a **short presentation** of their solution.

If you have questions or encounter problems using the VerCors tool set, you can can use the Blackboard forum or contact Stefan Blom, email address: s.c.c.blom@utwente.nl, office: Zilvering 3082.

Goal

The goal of this exercise is to specify a standard data structure and to verify several Java implementations, including a concurrent one.

Before presenting the details of the exercise, we first give some background information.

```
class OwickiGries {
  int x;

  resource lock_invariant ()=Perm(x,1);

  OwickiGries(){
    x=0;
  }

  void main(){
    Worker w1=new Worker(this);
    Worker w2=new Worker(this);
    fork w1; fork w2;
    join w1; join w2;
  }
}
```

Listing 1: Owicki-Gries main code.

Using locks in PVL

Simple non-reentrant locks can be used in PVL. To be able to lock objects belonging to a certain class:

1. The class must define the predicate `lock_invariant ()`.
2. The class must have a constructors, and the lock invariant must hold at the end of the constructor. (If it does not, the tool will complain that it cannot fold because there is an implicit fold at the end of the constructor.)
3. The syntax for locking and unlocking an object `o` is `lock o;` and `unlock o;`, respectively.

As an example, we show the classical Owicki-Gries example: two threads are spawned and each thread increments a global variable by one. The main program code can be found in Lst. 1 and the code for the worker in Lst. 2.

The PVL language also has `wait o;` and a `notify o;` statements. These statements behave as the Java `wait` and `notifyAll ()` statements. Moreover, they require that the lock is held: `held(o)`. (See files in `examples/waitnotify`.)

Compiling and Running PVL Programs

An experimental compiler for PVL programs exists. The compiler translates PVL to Java, which allows PVL code to be called from Java. For example, the code in Lst. 3 instantiates and calls the Owicki-Gries main method. To compile the program, run the (unix only) `pvlc` script:

```
class Worker {
  OwickiGries store;

  ensures Value(store) ** store==og;
  Worker(OwickiGries og){
    store=og;
  }

  requires Value(store);
  ensures  Value(store);
  void run(){
    lock  store;
    store.x=store.x+1;
    unlock  store;
  }
}
```

Listing 2: Owicki-Gries worker code.

```
class TestOwickiGries {
  public static void main(String args[]){
    OwickiGries og=new OwickiGries();
    System.err.printf("og.x == %d%n",og.x);
    og.main();
    System.err.printf("og.x == %d%n",og.x);
  }
}
```

Listing 3: Owicki-Gries test code.

```
pvlc OwickiGries.pvl TestOwickiGries.java Worker.pvl
```

This will translate the PVL files into Java and then it will compile both the generated and hand-written Java files. To run a program use the `pvl` script:

```
pvl TestOwickiGries
```

Exercises

The goal of this exercise is to implement a specified and verified data structure, whose behavior is identical to the behavior of some standard Java interface. For Group 1 this will be the Map, and for Group 2 this will be the Queue. In both cases the java.util.concurrent package extends the interface with methods that are useful for concurrent programs (ConcurrentMap and BlockingQueue, respectively).

	Group 1	Group 2
sequential API	Map<K,V>	Queue<E>
methods	put get	offer poll peek
concurrent API	ConcurrentMap<K,V>	BlockingQueue<E>
methods	putIfAbsent replace(K,expect, val) remove(K,val)	put take
implementation	ConcurrentHashMap<K,V>	LinkedBlockingQueue<E>

Table 1: Overview of relevant methods.

Both types use generics, which PVL does not have, so we use integers for the key and value types of the map, and the element type of the queue. Table 1 lists the most important methods of each interface, as well as examples of concurrent implementations. You can reuse the standard API implementation of these methods, and simplify them whenever necessary and/or appropriate.

When developing the specification for a concurrent data structure, it is important to know how the various operations transfer permissions between threads and locked objects. Sometimes internal objects/operations play critical roles in the whole bookkeeping process. For example, the AtomicInteger count in LinkedBlockingQueue.

1. **(30 points)** Create a specified and verified implementation of the sequential API. Consider both data race freedom and functional properties.
2. **(15 points)** Create at least three different specified and verified test cases for the sequential API that show how the exact values represented by the data structure are preserved.
3. **(30 points)** Create a specified and verified implementation of the concurrent API. Consider both data race freedom only for the specification.
4. **(15 points)** Create at least three different specified and verified test cases for the concurrent API that show how multiple threads can interact with the data structure at the same time.
5. **(10 points)** Discuss the possibilities adding functional properties (max. 500 words).

Presentation

Discuss your specifications, problems encountered (and their solutions), both for the sequential and the concurrent case.