Project
Overflowing Counters

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

17 May, 2022

This project is due on Sunday, June 5, at 23h59m. The exact details on how to turn in your solution will be made available at a later date. You will be expected to submit a zip file with your (annotated) Java files that should pass Verifast's analysis (with the arithmetic overflow check turned **off**). All methods must have pre and post conditions. Code that fails to meet these requirements will be considered as a failing project.

The project consists of verifying a sequence of bounded counters, which may be accessed concurrently, using the techniques discussed in class. The project is split into two tasks: Task 1 (Section 1) consists of the implementation and verification of a sequential counter sequence; Task 2 (Section 2) consists of the implementation and verification of a concurrent analogue of the sequence. You will have to install the spec `java.util.concurrent.locks.javaspec` (available online in CLIP) for the package `java.util.concurrent.locks` by copying it to the `bin/rt/` folder of your local verifast installation and append the `rt.jarspec` file with the name of the spec file.

Important Note: inconsistent predicates used as invariants or in preconditions of methods will let you **incorrectly** verify code. The symptom will be that you will be able to prove false, and that you will not be able to call those methods (since it is impossible to satisfy the precondition).

1 Sequential Counter Sequence (Task 1)

For this task we will not (yet) be concerned with concurrency. The goal here is to implement a fixed-capacity sequence of counters. The sequence will be backed by an array of `Counter` objects. The sequence must support the following API:

```
public class CounterSequence {
  ...

  public CounterSequence(int cap) { ... }

  public CounterSequence(int[] arr) { ... }

  public int length() { ... }
```

```
    public int capacity() { ... }

    public int getCounter(int i) { ... }

    public int addCounter(int limit) { ... }

    public void remCounter(int pos) { ... }

    public void remCounterPO(int pos) { ... }

    public void increment(int i, int val)  { ... }

    public void decrement(int i, int val) { ... }
}
```

Constructors. The first constructor takes as a parameter the maximum capacity of the sequence, allocating memory accordingly and creating a sequence that has no counters. The second constructor takes as input an array of integers, with the intent of creating a sequence that will have as many counters as there are integers in the array (i.e., the capacity of the sequence is the *length* of the array). Each integer in the array denotes the upper-limit of the corresponding counter in sequence (more on this below).

Selectors. The length and capacity methods return the current number of counters and the total capacity of the sequence, respectively. The `getCounter` method returns the value of the counter in position i of the sequence.

Modifiers. The `addCounter` appends a new counter to the end of the sequence with upper-limit given the parameter `limit`, assuming the sequence is not at maximum capacity. The method returns the index of the added counter. New counters always start with value 0. The two removal operations, `remCounter` and `remCounterPO`, both delete the counter at the given index of the sequence, assuming the index contains a counter. The `remCounter` operation is *not* order preserving, moving the last element of the sequence to the position of the removed counter. The `remCounterPO` operation must preserve the order of the elements of the sequence (i.e. moving all appropriate counters accordingly). The increment and decrement operations add and remove the given value to the counter in position i of the sequence. These operations assume the given value is positive and i is a valid index.

1.1 Individual Counters

We now focus on the individual counters that will be contained in the sequence. These objects should adhere to the following API:

```
public class Counter {
  private int val;
  private int limit;
  private boolean overflow;
```

```
    public Counter(int val, int limit)  { ... }

    public int getVal() { ... }
    public int getLimit() { ... }

    public void incr(int v) { ... }
    public void decr(int v) { ... }
  }
```

The fields of the counter represent its current value, its upper-limit and a boolean flag that becomes true if the counter has ever **over** or **underflowed** its limit. The limit is always a **positive** number. The get operations simply return the value of the counter and its limit.

The modifier operations increment and decrement the counter, respectively. The value of the counter will always be between 0 (inclusive) and its upper-limit (non-inclusive). The *increment* operation, if the increment results in an **overflow**, will update the boolean flag accordingly and set the counter value **modulo the limit**. For instance, a counter whose value is 5 and whose upper-limit is 10, given an increment of 5 will have value 0. A similar counter with an increment of 6 will have value 1, and so on. As is the case in the counter sequence, the increment and decrement values are assumed to be positive. The *decrement* operation aims to decrement the counter value, if the decrement **would** result in an **underflow** (i.e., a negative counter value), the operation updates the flag accordingly and sets the value to 0 instead. If no underflow occurs, the decrement decreases the value of the counter as expected.

1.2 Verification

Both classes must be accompanied with the appropriate predicates that characterize the memory footprint (and invariants) of their respective objects. All methods should have the **appropriate pre-conditions**, adhering to the informal but precise description above. In terms of post-conditions, the `Counter` operations should **precisely** describe the changes to the Counter's internal state (i.e., of its value and the flag), following the description of the modifier operations given above.

The `CounterSequence` operations, as a result of the predicate-based verification, need only visibly capture the number of elements of the sequence and its capacity. This means that the operations that add or remove counters from the sequence should have post-conditions that track this fact accordingly. The lookup operation need only additionally ensure that its result is non-negative (i.e., you need not verify that the result is within the upper-bound of the corresponding counter).

Important Note: The class invariant for the sequence should maintain the fact that all stored counter objects are **correct** (i.e., their values are between 0 and their upper-limits). This will require characterizing the array via the `array_slice_deep` predicate (up to the number of stored counters) and the `array_slice` predicate (for the `null` positions at the end of the sequence).

2 Concurrent Counter Sequence (Task 2)

The concurrent version of the sequence should be a *wrapper* for your sequential implementation from **Task 1**, adhering to the following structure:

```
public class CCSeq {
```

```
    CounterSequence seq;
    //Other relevant fields go here...

    public CCSeq(int cap) { ... }

    public int getCounter(int i)  { ... }

    public void incr(int i, int val) { ... }

    public void decr(int i, int val) { ... }

    public int addCounter(int limit) { ... }

    public void remCounter(int i) { ... }
}
```

The sequence's operations must be implemented (using monitors and conditions) such that they can be **safely** used by concurrent threads, as discussed in lecture. The constructor initializes a sequence with the given capacity. The `getCounter` operation returns the value of the counter at position i, or -1 **if that position is invalid**. Both `incr` and `decr` operations behave as before, except the index i may not necessarily be a valid index in the sequence. If i is not a valid index, the operations will return without modifying any counter in the sequence.

The `addCounter` operation will append a new counter (with the given limit) to the sequence, returning the index of the new counter. The `remCounter` operation will remove the counter at the given index, or have no effect if the index does not contain a counter. Note that these operations will require adequate concurrency control mechanisms to ensure that the insertion only takes place on a non-full sequence and that the removal only takes place on a non-empty sequence.

2.1 Client Code

You will also be required to implement a client that launches 100 threads to **(a)** add counters and perform increment/decrements to the added counter; **(b)** query a counter's value and remove it from the sequence, printing a log on the standard output. The threads that perform **(a)** should run concurrently with those that perform **(b)**.

2.2 Verification

You will need to use the verification technique for monitors and conditions discussed in class to verify concurrent usages of the sequence. It will be convenient to define **three predicate constructors**: one for the shared sequence state and two specialized variants that relate to the conditions necessary to verify the operations of the sequence. You will also need to define the **concurrent invariant**, which specifies the memory layout of the concurrent sequence and the logical representation of any monitors and conditions. This invariant must be preserved by all the operations of the sequence.