PROBLEM SET 3

Assigned: February 22, 2006 Due: March 1, 2006

Problem 1: Disk remapping (30 points)

In class, we've alluded to the fact that modern disks remap defective sectors to good sectors. This is usually implemented as follows. When the disk fails to read a sector, it will automatically retry the read operation several times. If a sector consistently fails to be readable, then it is marked bad, and remapped to a spare sector. To make your job easier, we will break the `Disk` abstraction to implement remapping; assume that `Read` can raise `Error` for transient errors, and that the function `Disk.LastError() -> DA` returns the `DA` for the most recently encountered error. Your implementation should mark a `DA` as bad after B retries. After a `DA` has been marked bad, you should attempt to read it N times when attempting a remap of that sector; if the sector cannot be read after N retries, an error should be raised. As in `BufferedDisk`, stack your `RemappingDisk` on top of a plain `Disk`.

a) (10 points) Provide an implementation of `Disk` that remaps bad sectors.

b) (10 points) Write down the abstraction function from the implementation to the specification.

c) (10 points) Prove that your code implements the spec of `Disk`.

Problem 2: Modeling a Hard Disk Head (20 points)

Here is a specification for a hard disk head.

```
MODULE Head EXPORT Move, Position

  TYPE Place = [cyl: Int, sec: Int]
       Path = SEQ Place

  VAR p : Path := {}
```

```
  APROC Move(dCyl: Int, dSec : Int) =
    << p := p + { Place(cyl:=dCyl, sec:=dSec)} >>

  FUNC SumCyl() = + : (p * (\ place | place.cyl))
  FUNC SumSec() = + : (p * (\ place | place.sec))
  FUNC Position() -> Place = Place(cyl:=sumCyl(), y:=sumSec())
END Head
```

The hard disk head, of course, doesn't care about where it has been in the past.

a) (10 points) Write an implementation that implements the `Head` specification and avoids storing the entire path of the head.

b) (10 points) Prove that your `HeadImpl` implements the `Head` specification.

The initial position of the disk head is at $(0, 0)$.

Problem 3: Weighted Sums (30 points)

The following specification maintains a sequence of pairs, from which it can derive the running total of weighted values and the largest value seen so far.

```
MODULE WeightedSumCalculator EXPORT Sum, LargestVal, Add
  VAR pairs : SEQ (Int, Int) := {}
  FUNC Car (p : (Int, Int)) -> Int = p.seq.(0)
  FUNC WeightedVal (p : (Int, Int)) -> Int = p.seq(0) * p.seq(1)

  APROC Add(val:Int, count:Int) = << pairs + := (val, count); >>
  FUNC LargestVal() -> Int RAISES {Empty} =
                IF pairs = {} => RAISE Empty
                              [*] (pairs * Car).max
  FUNC Sum() -> Int RAISES {Empty} =
                IF pairs = {} => RAISE Empty
                              [*] + : (pairs * WeightedVal)
End WeightedSumCalculator
```

a) (5 points) Write an implementation `WeightedSumCalculatorImpl` which only maintains the running weighted total and maximum.

b) (5 points) Augment this implementation with history variables to produce a new implementation, `WeightedSumCalculatorHImpl`. The augmented implementation should admit an abstraction function from `WeightedSumCalculatorHImpl` to `WeightedSumCalculator`.

c) (10 points) Write the abstraction function from the history-enabled implementation to the specification, and prove that the implementation indeed implements its spec.

d) (10 points) Write (without proof) an abstraction relation from `WeightedSumCalculatorImpl` to `WeightedSumCalculator`. How do these approaches compare? Would choosing a different, still-complete representation of the input data have yielded an easier proof?

Problem 4: Mind-Reading Vending Machine (20 points)

Here is a spec for a vending machine that reads the user's mind to decide what item to dispense.

```
MODULE MindReadingDispenser EXPORT insertCoin, dispense =
    VAR coinInserted : Bool, selection : String

    APROC InsertCoin() = << coinInserted := true;
                            selection := "soda" [] selection := "pop"
                         >>
    APROC Dispense() -> String =
                  << coinInserted =>
                       coinInserted := false; RET selection >>
END MindReadingDispenser
```

The implementation, on the other hand, decides what to dispense based on the current phase of the moon when `dispense` is called; we model this with a nondeterministic choice in `dispense`.

```
MODULE DispenserImpl EXPORT insertCoin, dispense =
    VAR coinInserted : Bool, selection : String

    APROC InsertCoin() = << coinInserted := true; >>
    APROC Dispense() -> String =
        << coinInserted => coinInserted := false;
            selection := "soda" [] selection := "pop";
            RET selection; >>
END DispenserImpl
```

In this problem, you will prove that this implementation implements the spec.

a) (5 points) Write a new module called `DispenserProph` that augments `DispenserImpl` with a prophecy variable.

b) (5 points) Give appropriate abstraction functions/relations from `DispenserImpl` to `DispenserProph` and from `DispenserProph` to `MindReadingDispenser`.

c) (5 points) Prove that `DispenserImpl` implements `DispenserProph`.

d) (5 points) Prove that `DispenserProph` implements `MindReadingDispenser`. (Why does that give the result we're looking for?)