Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 26 October 13, 1993

Solutions to Problem Set #2

The maximum score for this problem set was 70 points.[1]

Problem 1 (25 points)

Consider the following specification of a multiset (also sometimes known as a "bag") module:

```
MODULE Multiset [T] =

TYPE S = T->Int

VAR s := S{* -> 0}

APROC Insert (t) =  << s(t) := s(t)+1 >>

APROC Delete (t) RAISES {NotFound} =
          << s(t) = 0 => RAISE NotFound
                      [*] s(t) := s(t)-1 >>

FUNC Member(t)->Bool = RET( s(t)>0 )

END Multiset
```

(A) (5 points)

*Describe the state machine denoted by the specification above (*i.e.*, describe briefly the state domain, the initial states, and the state transitions.)*

The states are the functions from `T->Int`. The initial state is the constant `0` function (`S{*->0}`). The transitions are, from every state `s`, for every `t`:

- a transition `(Insert(t),RET)` to `s{t->s(t)+1}`;

[1]No, these points are not weighted the same as those for PS #1.

- if `s(t)#0`, a transition `(Delete(t),RET)` to `s{t->s(t)-1}`;
- if `s(t)=0`, a transition `(Delete(t),NotFound)` to `s`;
- if `s(t)>0`, a transition `(Member(t),true)` to `s`;
- if `s(t)<=0`, a transition `(Member(t),false)` to `s`;

Comments: Note that there is no "exception state" as such; every transition (and note that every transition is *labelled*) leaves the state machine in some ordinary state.

(B) (5 points)

Write an implementation of the Multiset module using a linear sequence with linear search to implement the multiset.

```
MODULE MultisetImpl [T] =

TYPE R = [d:T,c:Int]
     SS = SEQ R

VAR ss := SS{}

APROC Search (t)->Int RAISES {NotFound} =
   << VAR i := 0 |
         DO i < ss.size /\ ss(i).d # t => i:=i+1 OD;
         i = ss.size => RAISE NotFound
                    [*] RET i >>

APROC Insert (t) =
   << VAR i |
         << i := Search(t);
            ss(i).c := ss(i).c + 1 >>
         EXCEPT {NotFound} => ss := ss ++ R{d:=t,c:=1} END >>

APROC Delete (t) RAISES {NotFound} =
   << VAR i |
         i := Search(t);
         ss(i).c = 0 => RAISE NotFound
                    [*] ss(i).c := ss(i).c - 1 >>

APROC Member(t)->Bool =
   << VAR i |
         << i := Search(t); RET (ss(i) > 0) >>
         EXCEPT {NotFound} => false END >>
```

```
END MultisetImpl
```

(C) (5 points)

Describe the state machine denoted by the implementation you wrote in part (B).

The states are the sequences of type `SS`. The initial state is the empty sequence `SS{}`. The transitions are, from every state `ss`, for every `t`:

- if `∃i.ss(i).d=t`, a transition `(Insert(t),RET)` to `ss{i->R{d:=t,c:=ss(i).c+1}}`;
- otherwise, a transition `(Insert(t),RET)` to `ss++R{d:=t,c:=1}`;
- if `∃i.ss(i)=R{d:=t,c:=x}∧x#0`, a transition `(Delete(t),RET)` to `ss{i->R{d:=t,c:=x-1}`;
- otherwise, a transition `(Delete(t),NotFound)` to `ss`;
- if `∃i.ss(i)=R{d:=t,c:=x}∧x>0`, a transition `(Member(t),true)` to `ss`;
- otherwise, a transition `(Member(t),false)` to `ss`;

(D) (5 points)

Give an abstraction function and rep invariant for your implementation from part (B).

We want two things out of our rep. invariant: the sequences must represent functions, and the values of the functions must always be natural numbers:

```
(ALL i:Int | ss!i ==> ss(i).c >= 0
                   /\ (ALL j:Int | ss!j  /\ i#j  ==>  ss(i).d # ss(j).d)
```

The abstraction function can be declared in Spec as:

```
FUNC AF(ss) -> S =
  RET (LAMBDA (t) -> Int =
        (VAR int | ss(int).d = t  => RET ss(int).c [*] RET 0))
```

(E) (5 points)

Sketch a proof of correctness for your implementation from part (B), using your abstraction function and rep invariant from part (D). (List the cases that need to be considered, and for each case give a brief informal argument that the appropriate conditions are satisfied.)

The answer that follows will list the cases that need to be proven, and prove a couple of representative cases in a fair amount of detail. The level of detail shown here was not, in fact, required of your proof sketches.

First, since we're relying on the rep. invariant, we have to prove that it is in fact a rep. invariant; thus we have to show

1. The invariant holds of the initial state. (It does, trivially, since there are no indices for which `ss` is defined.)

2. Every transition listed in (C) beginning in a state satisfying the invariant leads to a state satisfying the invariant. Taking the most trivial case, note that `(Member(t),bool)` starting in a state satisfying the invariant, leads to the same state, and thus leads to a state satisfying the invariant.

Next, we need to show that the abstraction function is a proper abstraction function. We first note that the initial state of `MultisetImpl` maps to the initial state of `Multiset`:

```
AF(SS{}) = (LAMBDA (t) -> Int =
                (VAR int | SS{}(int).d = t => RET SS{}(int).c [*] RET 0))
         = (LAMBDA (t) -> Int = RET 0)
         = S{*->0}
```

Next we must prove that, for every transition T from `ss` to `ss'` in the implementation, there must be a transition T from `AF(ss)` to `AF(ss')` in the specification. There are four types of transitions to be considered: `(Insert(t),RET)`, `(Delete(t),RET)`, `(Delete(t),NotFound)`, and `(Member(t),bool)`. We will show only the first in detail:

(Insert(t),RET) Given a transition `(Insert(t),RET)` from `ss` to `ss'`, we know that one of two cases hold:

- The first case is that $\exists$`i` such that `ss(i).d=t`. By the rep. invariant we know that there is only one such `i`, and we know that `ss' = ss{i->R{d:=t, c:=ss(i).c+1}}`. We can show that

  ```
  AF(ss')(t) = AF(ss{i->R{d:=t,c:=ss(i).c+1}})
             = AF(ss){t->ss(i).c+1}
             = AF(ss){t->AF(ss)(t)+1}
  ```

  and thus there must be a transition `(Insert(t),RET)` from `AF(ss)` to `AF(ss')`.

- The alternative is that $\nexists$`i` such that `ss(i).d=t`, in which case `ss' = ss++R{d:=t,c:=1}`. We know that

  ```
  AF(ss')(t) = AF(ss++R{d:=t,c:=1})
             = AF(ss){t->1}
  ```

  But, since `AF(ss)(t)=0`, this again gives us

  ```
  AF(ss')(t) = AF(ss){t->AF(ss)(t)+1}
  ```

  so there is a transition `(Insert(t),RET)` from `AF(ss)` to `AF(ss')`.

The cases for the other types of transitions follow by a similar analysis.

Problem 2 (10 points)

This problem was also used last year. Do not look at last year's solution; work it out on your own.

Prove the following theorem:

Theorem: *Let I be an implementation, S be a specification, and AF be an abstraction function from I to S that satisfies the two conditions given in Handout 8. Then every trace of I is a trace of S.*

Proof: (by induction on the length of the trace) Recall that a trace is a sequence of transitions, where a transition is denoted by an invocation/response pair. We prove a stronger hypothesis by induction on the length of the trace:

> If T is a trace of I and s is a possible final state for T, then T is also a trace of S with $AF(s)$ as a possible final state.

A trace of length 0 leaves the state machine in one of the initial states. By Condition 1, if s is an initial state of I, then $AF(s)$ is an initial state of S. Therefore the hypothesis holds.

Suppose the hypothesis is true for traces of length k. Let T be a trace of I of length $k+1$ leading to state s_2. Decompose T into a trace T' of length k leading to state s_1 and a final transition p from s_1 to s_2. By induction, T' is also a trace of S with $AF(s_1)$ a possible final state. By Condition 2, there must exist a transition p in S from state $AF(s_1)$ to $AF(s_2)$. Therefore, T is a trace of S and $AF(s_2)$ is a possible final state.

Problem 3 (35 points)

The most popular way to get a highly available disk is to use two physical disks for each logical one, and to write an identical copy of each block on both disks (this is called "mirroring" or "shadowing"). This approach protects both against the corruption of individual blocks and the failure, temporary or permanent, of an entire disk. We would like to maintain the invariant that for each block, both copies are the same or one of the copies gives an error when read, but this isn't possible, since the only atomic operation is to write a single disk block to one disk.

(A) (10 points)

Modify the Dsk spec given in Handout 9 to reflect the possibility that a disk block can have either permanent or transient read errors. Explain your model for how disks can fail and why your spec is consistent with your model.

Here is one model of disk failure:

1. The unit of failure is a disk block.
2. A disk block can suffer either transient or permanent failure. In a transient failure, the disk block still contains usable data, which might be returned by a future read. In a permanent

failure, the data on the disk block cannot be retrieved forever due to a bad disk sector, a damaged disk head, *etc*.

3. However, a permanent disk block failure cannot be distinguished from a transient one.

4. A disk block error goes unnoticed until an attempt to read the block is made. If there is an error in reading a block, an error signal is raised.

With this model, only the `PROC BlocksToData` and `PROC ReadBlocks` of the `Dsk` spec in Handout 9 are modified. When there is a permanent or transient disk block error in the sequence of disk blocks passed to the `PROC BlocksToData`, the procedure raises `ReadError`, and subsequently the `PROC ReadBlocks` propagates this error signal up.

```
% raise ReadError when there is a permanent or transient disk block error
PROC BlocksToData(disk) -> Data RAISES {ReadError} = ...

% propagate a possible ReadError of BlocksToData
PROC ReadBlocks(e) -> Data RAISES {notThere, ReadError} =
         e.at + e.size >= disk.size => RAISE notThere
     [*] RET BlocksToData(disk.sub(e.at, e.at + e.size))
```

Comments: It is perfectly acceptable to state clearly a simple and reasonable model and to make only a few necessary modifications on the spec. to fit the model stated. Again, there were many alternative models that we received from the students, each was acceptable as long as it was clear and reasonable and the corresponding spec was well-commented and consistent with the model.

(B) (5 points)

Give a precise statement of the invariant mentioned above as a predicate in Spec. Let the two component disks be `dsk` and `dsk'`. Then we get:

```
(ALL e | e.size = 1 /\ dsk.ReadBlocks¡e /\ dsk'.ReadBlocks¡e
           ==> dsk.ReadBlocks(e) = dsk'.ReadBlocks(e))
```

In other words, for every single-block extent, if the extent can be successfully read from both disks, then the values read must be the same.

(C) (20 points)

Give an implementation of your Dsk spec using two copies of itself; in other words; give an implementation of a mirrored disk. Your implementation should maintain the invariant except while a Write is in progress. The tricky part of this is dealing with a failure during a write, either of a disk or of the process implementing the mirroring.

You should assume that if the process fails it stops executing after some atomic action, and the the next thing that happens is an invocation of a PROC called Crash, which must re-establish the invariant.

Let's parse the question into several key points:

1. Assume that if the process fails it stops executing after some atomic action.
2. Invariant: for each block, both copies are the same or one of the copies gives an error when read.
3. Give an implementation of a mirrored disk with two copies of the modified Dsk.
4. Implementation maintains the invariant except while a write is in progress.
5. Implementation deals with a failure during a write of a disk.
6. Implementation deals with a failure during a write of of the process implementing the mirroring.
7. `PROC Crash` is called to re-establish the invariant.

In the following implementation, `MirrorDisk`, takes into consideration all the key points listed above. For clarity, let's call the modified specification of `Dsk` given in Part (A) `FDsk`; `MirrorDisk` makes use of the `ReadBlocks` and `WriteBlocks` operations of the `FDsk`.

```
MODULE MirrorDisk

TYPE
  DA   = FDsk.DA
  DB   = FDsk.DB
  Disk = FDsk.Disk
  E    = FDsk.E
  Data = FDsk.Data

VAR
  DBSize         : Int := 1024        % assumed underlying DBSize of FDsk
  numRetries     : Int := 5           % configurable global variable
  safeNumRetries : Int := 20          % configurable global variable:
                                      % trying hard to make sure of
                                      % the nature of disk block error
  fdsk                                % primary disk
  fdsk'                               % mirrored disk
                                      % run PROC Crash at startup to
                                      % keep the representation invariant
                                      % between the two disks
```

`PROC ReadBlocks` first reads data from `fdsk` and, if it fails, it reads from `fdsk'`. It tries to read from `fdsk` and `fdsk'` alternately until one disk on one try does not return `ReadError` with `FDsk.ReadBlocks`, then `PROC ReadBlocks` returns the data read. If the number of unsuccessful reads made to each disk exceeds “`numRetries`” (a configurable number), then `PROC ReadBlocks` gives up and raises `ReadError` (suggesting disk blocks error on both disks).

```
PROC ReadBlocks(e) -> Data RAISES {notThere, ReadError} =
    VAR tryNum  : Int  := 0 ,
        error   : Bool := true ,
        data    : Data |
        DO tryNum < numRetries /\ error =>
            tryNum := tryNum + 1 ;          % one more try ...
            error := false ;                % hoping no error ...
            % ReadBlocks from fdsk
            data := fdsk.ReadBlocks(e)
                    EXCEPT {notThere} => RAISE notThere,
                           {ReadError} => error := true END ;
            error =>
                % unsuccessfully ReadBlocks from fdsk,
                % try from fdsk'
                data := fdsk'.ReadBlocks(e)
                        EXCEPT {notThere} => RAISE notThere,
```

```
                              {ReadError} => error := true END
        OD ;
        % The first successful read ReadBlocks from one of the
        % disks or too many failed reads terminates the loop above.
        ~error => RET data [*] RAISE ReadError
```

`PROC WriteBlocks` writes to each disk block by block using `FDsk.Writeblocks` which writes disk block atomically. It writes each block to `fdsk` then to `fdsk'` so as to minimize the discrepancies between the two disks.

```
PROC WriteBlocks(at: DA, data) RAISES {notThere} =
    VAR size : Int := data.size/DBSize |
        DO size > 0 =>
            fdsk.WriteBlocks(at,data.sub(0,DBSize-1))
                EXCEPT {notThere} => RAISE notThere END ;
            fdsk'.WriteBlocks(at,data.sub(0,DBSize-1))
                EXCEPT {notThere} => RAISE notThere END ;
            at := at + 1 ;
            size := size - 1 ;
            data := data.sub(DBSize, data.size)
        OD
```

`Crash` re-establishes the invariant. The invariant may be violated after a failure occurs during a write of a disk or of the process implementing the mirroring procedure. In either case, the atomic block write actions carried out by `WriteBlocks` will be executed to completion. According to the `WriteBlocks` implementation above, an atomic disk block write action is done on `fdsk` (the primary disk) before it is done on `fdsk'` (the mirror disk). As a result, if the two readable blocks of the same address of the two disks are different in content, the data in the `fdsk` block should be taken as the desired data and copied to `fdsk'` to restore the invariant. This implementation of `Crash` takes a long time to examine the whole `fdsk` and `fdsk'` and to compare their content block by block.

The following table shows the cases to worry about for the disks after a crash that occurred while writing block X, where an "err" is permanent error (identified after failing to read for a total of "`safeNumRetries`" times). For block X on:

`fdsk`	`fdsk'`	
old	old	data didn't get written, *OK*
old	new	can't happen since `fdsk` written before `fdsk'`
old	err	data didn't get written, *OK*
new	old	*Invariant Violated: Need to Fix in* `Crash`
new	new	data did get written, *OK*
new	err	data did get written, *OK*
err	old	data didn't get written, *OK*
err	new	data did get written, *OK*
err	err	data didn't get written, *OK*

```
PROC Crash () =
  VAR at : DA := 0 |
    DO at < fdsk.size =>                    % scan the whole disk
      VAR tryNum1 : Int  := 0 ,
          error1  : Bool := true ,
          data1   : Data |
        DO tryNum1 < safeNumRetries /\ error1 =>
          tryNum1 := tryNum1 + 1 ;          % one more try ...
          error1 := false ;                 % hoping no error ...
          % read a block of data from fdsk
          data1 := fdsk.ReadBlocks(E{at:=at,size:=1})
                   EXCEPT {ReadError} => error1 := true END ;
        OD ;
        % The first successful read ReadBlocks from fdsk or too
        % many (``safeNumRetries" times) failed reads terminates the
        % loop above. If fdsk does not return some readable data, the
        % invariant is not violated in this block; otherwise, the
        % disk block content from fdsk' must be checked.
        ~error1 =>
          VAR tryNum2 : Int  := 0 ,
              error2  : Bool := true ,
              data2   : Data |
            DO tryNum2 < safeNumRetries /\ error2 =>
              tryNum2 := tryNum2 + 1 ;      % one more try ...
              error2 := false ;             % hoping no error ...
              % read a block of data from fdsk'
              data2 := fdsk'.ReadBlocks(E{at:=at,size:=1})
                       EXCEPT {ReadError} => error2 := true END ;
            OD ;
            % If fdsk' does not return some readable data, the
            % invariant is not violated in this block; otherwise,
            % data1 must be compared with data2.
            ~error2 =>
              data1 # data2 =>              % when invariant is
                fdsk'.WriteBlocks(at,data1); % violated, fdsk' block
                                            % is overwritten
    OD;
```

Comments: Again, we received many alternative implementations from students; each was acceptable as long as it handled the key issues listed above and was clear, reasonable, well-commented, and consistent with the spec. and model. It did not need to be very complicated.