Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout ?? December 13, 1993

---

# Solutions to Problem Set #7

**Problem 1** (50 points)

*The abstraction function given in Handout 31 for the* `LogAndCache` *implementation of* `SequentialTransactions` *is wrong: the value given for* `vs` *doesn't work. The third invariant is similarly wrong. Your job in this problem is to fix them.*

First of all, I'm sure it will come as a great surprise to everyone that there were still bugs in the `LogAndCache` spec. after our last round of corrections. Just for the record:

1. In `Do`, the call to `DoOrUndoA` is not an expression, so it can't be returned. The result should be assigned to a local variable, and then the variable should be returned.
2. `DoOrUndoA` should be an `APROC`.
3. `DropW` needs to be an `APROC`. (Actually, it probably works as it stands, but the added actions aren't necessary).
4. `Truncate` needs to be an `APROC`—otherwise, `sl` could change (due to `Force` being called from `Commit`) between the guard and the action, and the result would be to lose stuff from `sl`.
5. `Force` needs to be an `APROC`—otherwise, two invocations of it can be running concurrently (one from `Commit` and one from `Flush`), and they can mess things up (*e.g.*, stuff can get moved twice to the stable log, which is a problem in the undo log). Notice that once we do this, the `forcing` flag can go away, and the second invariant is just the consequent of the implication.

Finally, the semantics of Spec seem to leave it unclear what happens to `THREAD`s on recovery from a crash. In this case, the background thread should be restarted after a crash but before recovery (so that `Flush` and `DropW` can happen during `Redo` and `Undo`). If this is done, then the body of the loop in `Redo` needs to be an atomic action. Probably the right thing to do is to say that the threads get restarted immediately after the first atomic action in `Crash` has been done.

With that said, here's the solution to what remains of the problem:

**(A)** (20 points)

*Give a correct abstraction function for* `LogAndCache`. *Fix the problem with the existing abstraction function for* `vs`, *and also give the abstraction function for* `ph`.

**(B)** (15 points)

*Fix the third invariant to explain correctly how* `sb++c` *relates to* `sb`, `sl`, `sul`, `vl` *and* `vul`.

**(C)** (15 points)

*Argue informally (but convincingly) that the abstraction function you gave in part (A) is correct. Identify the atomic actions in* `LogAndCache`, *and argue briefly for each how it is explained by an appropriate sequence of atomic actions in* `SequentialTransactions`. *(Note: this may seem like a big job, but there aren't actually that many atomic actions to be considered. Feel free to apply the techniques of Handout 21 to reduce the number of actions even further.)*

Starting with part (C):

## Atomic actions in `LogAndCache` (with explanations in terms of the spec)

Except for `Abort` and `Crash`, invocations and returns of the external operations are all explained by the identical actions in the spec.

| | | |
|---|---|---|
| 1 | `Begin` | the body of `Begin` in the spec. |
| 2 | `Do` | the body of `Do` in the spec. |
| 3 | Raising `crash` in `Commit` | the body of `Commit` in the spec (when it raises `crash`). |
| 4 | Modifying `sl`, `sul`, and `vph` in `Commit` | the body of `Commit` in the spec (when it sets `ss` to `vs`). |
| 5 | Each call of `Force` from `ForceAll` from `Commit` | the empty sequence. |
| 6 | An invocation of `Abort` (moving control to the beginning of the body of `Abort`) | the invocation of `Abort` followed by the body of `Abort` in the spec. |
| 7 | Each call of `DoOrUndoA` from `Undo` from `Abort` | the empty sequence. |
| 8 | Each call of `Force` from `ForceAll` from `Abort` | the empty sequence. |
| 9 | Setting `vph` to `idle` in `Abort` | the empty sequence. |
| 10 | Reading `c` in `Checkpoint` | the empty sequence. |
| 11 | Each evaluation of the guard in the loop in `Flush` called from `Checkpoint` | the empty sequence. |
| 12 | Each call of `Force` from `Flush` from `Checkpoint` | the empty sequence. |
| 13 | The assignment to `sb` at the end of `Flush` called from `Checkpoint` | the empty sequence. |
| 14 | Each call to `Truncate` from `Checkpoint` | the empty sequence. |

| | | |
|---|---|---|
| 15 | An invocation of `Crash` (moving control to the beginning of the body of `Crash`) | the invocation of `Crash` followed by the body of `Crash` in the spec. |
| 16 | Clearing `vl`, `vul`, and `c` in `Crash` | the empty sequence. |
| 17 | Reading `sl` in `Redo` | the empty sequence. |
| 18 | The body of the loop in `Redo` called from `Crash.` (We will assume that the background thread gets restarted after a crash once the volatile variables have been cleared and before `Redo` is called. This is important so that `Flush` and `DropW` can happen during crash recovery. But then we need to make the body of the loop in `Redo` atomic.) | the empty sequence. |
| 19 | Reading `sul` and `vul` in `Undo` | the empty sequence. |
| 20 | Each call of `DoOrUndoA` from `Undo` from `Crash` | the empty sequence. |
| 21 | Setting `vph` to `idle` in `Crash` | the empty sequence. |
| 22 | Each evaluation of the guard in the loop in `Flush` called from the background thread | the empty sequence. |
| 23 | Each call of `Force` from `Flush` from the background thread | the empty sequence. |
| 24 | The assignment to `sb` at the end of `Flush` called from the background thread | the empty sequence. |
| 25 | Each call of `DropW` from the background thread. | the empty sequence. |
| 26 | Each call of `Truncate` from the background thread. | the empty sequence. |

The abstraction function is then (using English in place of program counters):

| | | |
|---|---|---|
| `ss` | = | `sb + sl - sul` |
| `vs` | = | if no thread is in `Abort` or `Crash` then `sb + sl + vl` else `sb + (sl + vl) - (sul + vul)` |
| `ph` | = | if no thread is in `Abort` or `Crash` then `vph` else `idle` |

Invariant 3 becomes:

If no thread is in `Abort` or `Crash` then `sb ++ c = sb + sl + vl`

The correctness argument follows:

First, it's easy to see that the invocations and returns of the external operations can all be explained by the identical actions in the spec: Each of these actions modifies the pc of the invoking thread, and nothing else of interest. It's easy to see that their behavior is explained by the corresponding action in the spec.

Now:

| | |
|---|---|
| 1 | In the implementation, if `vph = idle` then `Begin` sets `vph` to `run`. Under the abstraction function, this corresponds to setting `ph` to `run`. The only other part of showing that this transition works properly is to show that `Begin` in the implementation has the abstract effect of setting `vs` to `ss`. This requires an invariant that says that:<br><br>`vph = idle => sb + sl - sul = sb + (sl+vl) - (sul+vul)`<br><br>If `vph = run` in the implementation, then `Begin` runs `Abort` and leaves `vph` alone. This means that `vph = run` after `Begin` is done, which is what is needed for `ph` in the spec. We then need to show that `Abort` has the abstract effect of setting `vs` to `ss`; this should be straightforward. |
| 2 | In the implementation, `Do` raises `crash` if `vph = idle`, which corresponds to the exception begin raised in the spec. If `vph = run`, `Do` calls `DoOrUndoA`; we need to show that this modifies `vs` appropriately and returns the correct value. But `DoOrUndoA` computes `vs` (which is correct by the third invariant), and then adds to `c` a set of updates that has the desired effect. |
| 3 | This happens when `vph = idle`, which corresponds to the same exception in the spec. |
| 4 | Adding the commit record to `sl` and setting `sul` to empty has the effect of changing `ss` (by the abstraction function) to include everything in the stable log. We need an invariant that tells us that `vl` is empty after the call to `ForceAll`, and then the abstraction function for `vs` tells us that `vs = sb + sl` at that point. Setting `sul` to empty then makes `ss` equal to `vs`, which is what is needed. |

**Problem 2** (50 points)

```
MODULE OptimisticConcurrentTransactions [
   V,
   C,                                  % State component.
   A WITH {meaning: C -> (C, V),       % Action:
           component: -> INT,          % a.component gives the
                                       %    component accessed;
           kind: -> Bool},             % a.kind returns true iff
                                       %    a is a read.
   T
   ]

TYPE
   SC = SEQ C
   S = SC WITH {s0:=Init}

VAR
   s: S := S$s0()                      % Committed state.
```

```
   rds: T -> SET INT := {* -> {}}   % Set of components read
                                    %     by a transaction.
   wrs: T -> SET INT := {* -> {}}   % Set of components written ...

   active: SET T := {}              % Currently active transactions.
   committed: SEQ T := {}           % Committed transactions
                                    %     (in commit order).
   start: T -> INT := {* -> }       % Number of commits before start.

   updates: T -> (INT -> C) := {* -> {* -> }}
                                    % New value written to a component
                                    %     by a transaction.

   used: SET T := {}                % Transaction ids used so far.

APROC Begin() -> T =
   << VAR t | ˜ t IN used =>
        start(t) := committed.size;
        active := active ++ t;
        used := used ++ t;
        RET t >>

APROC Do(t,a) -> V RAISES {badT} =
   << ˜ t IN active => RAISE badT
   [*] VAR i: INT := a.component, c |
       c := (updates(t)!i => updates(t)(i) [*] s(i));
       VAR c',v | (c',v) = a.meaning(c) |
       a.kind => % Read action
                 rds(t) := rds(t) ++ i;
                 RET v
       [*]       % Update action
                 wrs(t) := wrs(t) ++ i;
                 updates(t)(i) := c';
                 RET v
       >>

APROC Commit(t) RAISES {badT} =
   << t IN committed => SKIP
    [*] ˜ t IN active => RAISE badT
    [*] % Check to see if any transaction that committed since this
        % one started conflicted with it:
```

```
        NoConflicts(t) =>
          committed := committed ++ t;   % Add to committed sequence
          DO VAR i: INT |                % Install changes
             s(i) # updates(t)(i) => s(i) := updates(t)(i) OD;
          updates(t) := {* -> };
          active := active -- t
   >>

APROC Abort(t) RAISES {badT} =
   << t IN committed \/ ~ t IN used => RAISE badT
     [*] updates(t) := {* -> };
         active := active -- t >>
```

There are two ways of checking for conflicts: "backward validation" and "forward validation." Backward validation checks when a transaction tries to commit whether any other transaction that committed since it started conflicted with it. Forward validation checks when a transaction tries to commit whether it conflicts with any currently active transaction. Both options are shown below.

```
% Backward validation:
%    Return true iff no conflicts with already committed transactions.
FUNC NoConflicts(t) -> Bool =
   RET
      (ALL t' | t' IN committed.sub(start(t),committed.size-1) ==>
          (rds(t)+wrs(t)) * wrs(t') = {}  % t' didn't write anything
                                          % read or written by t
```

Note that we don't really need to check if `t` has written to anything read by a `t'`: if `t'` is already committed, then it's okay for `t` to write something after it. The serialization order is simply the commit order, and letting `t` update a component after an earlier committed transaction has read it won't violate serializability.

```
% Forward validation:
%      Return true iff no conflicts with other active transactions.
FUNC NoConflicts(t) -> Bool =
   RET
      (ALL t' | t' IN used /\ ~ t' IN committed ==>
         wrs(t) * (rds(t')+wrs(t')) = {}  % t' didn't read or write
                                          % anything written by t
```

Again, in this case it's not necessary to check if `t'` wrote to anything read by `t`, since if `t'` has already committed then it's ok for `t` to have read something that `t'` has update.

```
END OptimisticConcurrentTransactions
```