Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.826 Principles of Computer Systems

PROBLEM SET 3 SOLUTIONS

Problem 1. Turtle Robot

a)

```
MODULE TurtleImpl EXPORT Move, Position

TYPE Coord = [x: Int, y: Int]
     Path = SEQ Coord

VAR pos: Coord := Coord(x:=0, y:=0)

APROC Move(dx: Int, dy: Int) =
  << pos.x := pos.x + dx;
     pos.y := pos.y + dy; >>

FUNC Position() -> Cord = RET pos;
```

b) We prove that `TurtleImpl` implements `Turtle` using an abstraction *relation* `AR`. The relation relates each concrete state to the set of abstract states.

```
FUNC sumdxAR(p) = + : (p * (\ coord | coord.x))
FUNC sumdyAR(p) = + : (p * (\ coord | coord.y))

FUNC AR(pos, p) -> Bool =
  (pos = Coord(x:=sumdxAR(p), y:=sumdyAR(p)))
```

Base Case. For the initial state `Coord(0,0)` of `TurtleImpl` there exists an initial state of `Turtle` such that `AR(Coord(0,0),{ })` because the sum of empty sequence of elements is zero. (There are other sequences related to `Coord(0,0)` but all we need to show that at least one of them is the initial state of `Turtle`.)

Induction Step. Let `pos=Coord(x,y)` be *any* state in the implementation and let `p` be *any* sequence in the specification such that `AR(pos,p)`. This means that

```
Coord(x,y) = Coord(sumdxAR(p), sumdyAR(p))
```

A step in the implementation can be either `Move(dx,dy)` or `Position()->Coord(x,y)`.

Consider `Move(dx,dy)`. The resulting concrete state is `pos' = Coord(x+dx, y+dy)`. The same transition `Move(dx,dy)` in the specification leads to the state `p' = p + Coord(dx,dy)` , so we need to show that `AR(pos', p')`, i.e.

```
Coord(x+dx, y+dy) = Coord(sumdxAR(p + {Coord(dx,dy)}),
                          sumdyAR(p + {Coord(dx,dy)}))
```

The condition reduces to:

```
x+dx = sumdxAR(p + { Coord(dx,dy) })
```

and

```
y+dy = sumdyAR(p + { Coord(dx,dy) })
```

Consider the first condition (the second one is analogous). Unfolding the definition of `sumdxAR` yields:

```
x+dx = + : ((p + { Coord(dx,dy) }) * (\ coord | coord.x))
```

Next we use the property of sequences:

$$(s_1 + s_2) * f = (s_1 * f) + (s_2 * f)$$

to reduce our problem to:

```
+ : ((p * (\ coord | coord.x) + { dx }) )
```

which by folding of `sumdxAR` is

```
x+dx = sumdxAR(p) + dx
```

and is true by the hypothesis.

Next consider the step `Position()->Coord(x,y)`. The step in the specification is

```
Position()->Coord(sumdx(),sumdy())
```

`sumdx()` is `sumdxAR(p)`, and `sumdxAR(p)=x` by the abstraction relation. For the analogous reason we have `sumdyAR(p)=y`. Hence the two `Position` actions return the same values. The state is unchanged in both cases so the implementation and specification state still remain related via `AR`.

This completes the proof.

Problem 2. Lossy Memory

The basic idea of the solution is the following. To avoid expanding the set of traces, the implementation must write back a dirty location on the first read to that location. This way we avoid traces like this one:

```
 action:            c(a1)     um(a1)

Write(a1,v1)         v1         *
[write-back(a1)]     v1         v1
Write(a1,v2)         v2         v1
Read(a1)->v2         v2         v1
Read(a2)->...
  [evict(a1)]        -          v1
Read(a2)->v3
Read(a1)->...
  [load(a1)]         v1         v1
Read(a1)->v1
```

Such trace is not allowed by the specification (because it has two successive reads that return different results without any writes between them). However, this trace could be generated if we implemented the usual write-back cache on top of `LMemory`.

a)

```
MODULE LWBCache [A, V] EXPORT Read, Write =

TYPE C = A -> V      % cache type
     D = A -> Bool   % dirty bit type
     M = A -> V      % memory type
```

```
CONST Csize: Int := 1024

VAR c := InitC()        % cache
    d := D{* -> true}   % dirty bit
    um:= InitM()        % underlying memory

APROC InitC() -> C = << VAR c' | c'.dom.size = Csize => RET c' >>
APROC InitM() -> M = << VAR m' | (ALL a | m'! a) => RET m' >>

APROC Read(a) -> V = <<
  IF c!a =>
     IF d(a) =>
        % Write and read it back
        MemWrite(a, c(a));
        c(a) := um(a);
     [*] SKIP
     FI
  [*]
     FlushOne();
     c(a) := um(a);
     d(a) := false
  FI;
  RET c(a); >>

APROC Write(a, v) = <<
  IF ~c!a => FlushOne() [*] SKIP FI;
  c(a) := v;
  d(a) := true; >>

APROC FlushOne() = << VAR a | c!a => Evict(a) >>

APROC Evict(a) = <<
  IF d(a) => MemWrite(a,c(a))
  [*] SKIP
  FI;
  d(a) := false;
  c(a) := undefined >>

APROC MemWrite(a,v) = <<
  BEGIN um(a) := v [] SKIP END;
  d(a) := false;  % the rest of the code will enforce this is ok
>>

END LWBCache
```

This implementation is in principle a bit better than write-through because successive writes without intervening reads need not go to memory.

b) This solution consists of three parts. The first part proves directly the trace set inclusion using an ad-hoc technique. The second part uses the result of the first part to illustrate the notion of a canonical automaton that shows that using first one forward and then one backward simulation relation is always enough to show trace set inclusion. The third part is a completely independent solution to the problem; it uses history variables to show trace inclusion with first backward simulation relation and then forward simulation relation.

Proof 1. Direct Proof. This proof proceeds by direct reasoning about the traces generated by the specification and traces generated by the implementation. For the specification `LMemory` we define the function `nextAction(t)` which computes the set of actions of `LMemory` that can extend the trace `t`:

```
nextAction(t) = { a | (t + {a}) IN traces(LMemory) }
```

Next we describe how to compute `nextAction`. For every `t`, the set `nextAction(t)` consists of all possible write operations and only some read operations. To see which read operations can extend a given trace `t`, consider the sequence of all reads and writes to address `a` in `t`. Let the values written and read in this sequence be:

$$\ldots\mathtt{r\text{->}v1}, \mathtt{w(v2)}, \ldots, \mathtt{r\text{->}vk}, \mathtt{w(vk_1)}, \ldots, \mathtt{w(wk_{\mathit{n}})}$$

In other words, let the last read operation of location `a` read the value `vk` and let all operations after that read be writes to `a` writing values $\mathtt{vk_1}$,...,$\mathtt{vk}_n$. Then the result of a read operation is any element of the set

$$\{\mathtt{vk}, \mathtt{vk_1}, \ldots, \mathtt{vk}_n\}$$

This is because the value of location at the last read was `vk`, and it can change only due to writes to that location. Conversely, for every value $\mathtt{vk}_i$ there exists an execution where write of $\mathtt{vk}_i$ succeeds and all subsequent writes in this sequence fail.

We have in fact used the characterization of traces to rule out some inappropriate caching policies for a) which would generate a larger set of traces. In this part we use the characterization to show that the implementation in in a) implements the `LMemory` specification. We show that all traces generated by `LWBCache` satisfy the characterization. This means that we need to show that a read of the address `a` returns either the result of the last read of `a` or the value written to `a` by some write to `a` that follows the last read. This invariant is affected by writes to the cache and memory, because they change the state of `LWBCache`, and it is also affected by the completion of read and write actions, because this changes the trace of reads and writes performed.

We need to show the following three invariants:

1. The value `c(a)` is either undefined or it is the result of the last read of `a` or the value written to `a` by some write to `a` that follows the last read (or any value if there are no previous accesses to that address)

2. The value `m(a)` is also either the result of last read of `a` or the value written to `a` by some write to `a` that follows the last read (or any value if there are no previous accesses to that address)

3. If valid an entry in the cache has the dirty flag false, then it's value is equal to the value of memory. Formally, if `m` is the value of underlying memory,

   ```
   c!a /\ d(a) ==> d(a)=m(a)
   ```

We show that the conjunction of these three invariants is satisfied using induction on the trace. The conjunction is clearly satisfied in the initial state: all three invariants trivially hold. We show that the invariants are preserved by both read and write actions.

Consider a write action. `Write` might execute `FlushOne`. `FlushOne` does not violate the first two invariants, because the requirement for both `c` and `m` values is the same. It also does not violate the third invariant because `c!a` becomes false. Statement `c(a):=v` does not violate the invariant because after the `Write` action is performed `v` will be one of the permissible values to be stored in both cache and memory. Setting `d(a)` to `true` clearly does not violate the third invariant.

Now consider a read action. Again `FlushOne` will not violate any invariant. Next, because of `RET c(a)` the invariant for `c(a)` will be satisfied by definition. We just need to show that after the procedure executed the content of the memory is equal to the content of the cache. In the branch executed after `d(a)` this is clear because the value of cache is set to the result of the read of memory. In the case when `d(a)` does not hold, the equality of cache and memory follows from the third invariant.

Now that we have shown the invariants, the condition that `Read` returns one of the values $\{\mathtt{vk}, \mathtt{vk_1}, \ldots, \mathtt{vk}_n\}$ follows from the first invariant and the fact that `Read` always the value `c(a)` from the cache.

Note. The proof does not rely on the code

```
IF d(a) => MemWrite(a,c(a))
[*] SKIP
FI;
```

in `FlushOne`. Indeed, we could omit this part and retain the same set of traces, but the implementation would be less useful because it would be even more lossy than the original memory (the set of traces would be the same, but the statistical distribution of traces would be worse). Similarly, one could write back more frequently, but that is likely to degrade the performance of the system.

Proof 2. Simulation Relations via Canonical Automaton. Observe that `nextAction` completely characterizes the set of traces of `LMemory`. We could use `nextAction` to build a state machine `LMemoryCanonical` whose state is the trace of previous execution, and which makes all choices by invoking `nextAction` on its state and picking one of the transitions that are permitted by `nextAction`.

```
MODULE LWBCache [A, V] EXPORT Read, Write =

TYPE ReadA  = [a: A, res:V]
     WriteA = [a: A, v: V]
     Action = ReadA \/ WriteA
     Trace  = SEQ Action

VAR  t: Trace;

FUNC nextAction(t: Trace) -> SET Action = % see the text

APROC Read(a) -> V = <<
  VAR act: ReadA, v: V | (act = ReadA(a,v) /\
                          act IN nextAction(t)) =>
    t := t + { act };
    RET v >>

APROC Write(a, v) = <<
  WriteA(a,v) IN nextAction(t) =>
    t := t + { WriteA(a,v) }
>>

END LWBCache
```

The `LMemoryCanonical` has several nice properties:

1. Its set of traces is exactly the set of traces of the original `LMemory`, this is simply a part of the definition of `nextAction`.

2. It is deterministic, because given a trace `t` and an action `a` it has transition only to the state `t + a`.

3. Its state is just the history i.e. the trace so far. This means that every possible implementation `Impl` of `LMemoryCanonical` can be shown to implement `LMemoryCanonical` by a forward simulation relation which assigns to every state of `Impl` the set of all traces that can lead to `Impl`. Intuitive explanation why we can always use forward simulation relation is that `LMemoryCanonical` never makes choices before showing these choices to the environment.

4. Again because the state of the state of `LMemoryCanonical` is just the trace, the condition for backward simulation relation can be used to show that `LMemoryCanonical` implements `LMemory`, and this will work in fairly general case.

It is clear that the construction just described is by no means specific to `LMemory`. In principle, we can construct a canonical machine for every specification. `LMemory` also shows that the description of such canonical machine may be more complicated than the original specification. We can say that `LMemory` uses the current values of its locations as a prophecy for the results of future read operations, and we see that making a prophecy can result in a simpler description.

The reason why computing `LMemoryCanonical` makes sense in our example is that we were able to find relatively succinct description of `nextAction` that refers only to some actions in the trace. In the worst case, the `LMemoryCanonical` machine might have nothing better to do than to run the original specification machine on the entire trace. Given the definition of a generic forward simulation relation to the canonical machine, this would be no better than proving that traces of `Impl` are included in the traces of `LMemory` by induction on the trace length.

As a completeness result we argued that both forward and backward simulation relations are needed in general. In our case the backward simulation relation is hidden as part of reasoning when computing `nextAction`. When computing `nextAction` we need to show that all actions in `nextAction(t)` can be generated by the `LMemory`. This cannot be done using forward simulation. So when reasoning about read actions of `LMemory` we had to argue that for every choice of the values returned in a read action of `nextAction`, it was possible to find an execution of `LMemory` that has the same read action, and in fact we had to refer to the choices that `LMemory` had to make in earlier steps. The crucial property that prevents the use of forward simulation relation is that for different read actions of `LMemoryCanonical` from the same state, we must have different states of `LMemory` that generate those actions.

From theses remarks it should follow how to make the proof using a forward simulation relation from `LWBCache` to `LMemoryCanonical` and a backward simulation relation from `LMemoryCanonical` to `LMemory`.

Proof 3. Prophecy Variables. In previous part we argued that the proof can be performed using forward and then backward simulation relation with the canonical automaton as the intermediate automaton and we relied on characterization of sets of traces of the specification and direct reasoning about the traces. Such characterizations might be hard to obtain in general. In this part we show a proof that uses prophecy variables to construct the intermediate automaton.

We extend `LWBCache` with a prophecy variable `p` which is used to determine if the write back to the memory should succeed or not.

```
MODULE LWBCacheP [A, V] EXPORT Read, Write =

TYPE C = A -> V     % cache type
     D = A -> Bool  % dirty bit type
     M = A -> V     % memory type
     P = A -> Bool  % PROPHECY type

CONST Csize: Int := 1024

VAR c := InitC()        % cache
    d := D{* -> true}   % dirty bit
    um:= InitM()        % underlying memory
    p := InitP()        % PROPHECY variable

APROC InitC() -> C = << VAR c' | c'.dom.size = Csize => RET c' >>
APROC InitM() -> M = << VAR m' | (ALL a | m'! a) => RET m' >>
APROC InitP() -> P = << VAR p' | (ALL a | p'! a) => RET p' >>

APROC Read(a) -> V = <<
  IF c!a =>
     IF d(a) =>
        % Write and read it back
        MemWrite(a, c(a));
        c(a) := um(a);
     [*] SKIP
     FI
```

```
  [*]
     FlushOne();
     c(a) := um(a);
     d(a) := false;
     p(a) := false;
  FI;
  RET c(a); >>

APROC Write(a, v) = <<
  IF ~c!a => FlushOne() [*] SKIP FI;
  IF ~p(a) =>  % fails if condition false
    c(a) := v;
    d(a) := true;
    BEGIN p(a) := true [] p(a) := false END; % set PROPHECY
  FI
>>

APROC FlushOne() = << VAR a | c!a => Evict(a) >>

APROC Evict(a) = <<
  IF d(a) => MemWrite(a,c(a))
  [*] SKIP
  FI;
  d(a) := false;
  p(a) := false;
  c(a) := undefined >>

APROC MemWrite(a,v) = <<
  BEGIN p(a) => um(a) := v [*] SKIP END;
  d(a) := false;  % the rest of the code will enforce this is ok
  p(a) := false;  % destroy prophecy info
>>

END LWBCacheP
```

We immediately observe that, like `LWBCache`, module `LWBCacheP` correctly maintains the dirty bit. Formally, it preserves the invariant:

```
INVARIANT DirtyOK(c,um,d) -> Bool =
  ALL a | ~d(a) /\ c!a ==> c(a)=um(a)
```

The invariant `DirtyOK` holds in the initial states because all dirty bits are initially true. To see that it is preserved in every step it is sufficient to check cases when `d(a)` is changed to `false` and cases when `c(a)` and `um(a)` are changed to they might become different. The first change occurs in the body of `Read`, but is always accompanied by setting cache to the value of memory. It also occurs when calling `Evict(a)` but there it is accompanied by making `c!a` false. The second change occurs only in `Write` and is accompanied by setting `d(a)` to false. This shows that invariant is maintained in every step.

Next we show that `traces(LWBCache)` is a subset of `traces(LWBCacheP)` using a backward simulation relation from `LWBCache` to `LWBCacheP`. We use a special kind of backward simulation relation that is suitable for use with history variables. It assigns a state t of `LWBCache` with a state (t, p) of `LWBCacheP`.

We define the abstraction relation by stating the condition between $t = (c, um, d)$ and p such that t is related to (t, p) iff $\mathtt{BR}(t, p)$ holds.

```
BACKWARD RELATION BR(c,um,d,p) -> Bool =
  ALL a | ~d(a) => ~p(a)
```

The abstraction relation assigns `p(a)=false` to a clean location `a` and assigns both `true` and `false` to a dirty location.

The first condition for a backward relation is that for each state of `LWBCache` there exists at least one related state of `LWBCacheP`. This holds because to every state we can at least assign state which has the prophecy for all locations set to `false`.

The next condition is that for every initial state of `LWBCache`, states related to it are initial states of `LWBCacheP`. This holds because the function `InitP` initializes prophecy variables with all possible values.

Finally we need to show the inductive step: for every state t of `LWBCache`, for every step $t' \; \pi \; t$ of `LWBCache`, and for every state (t,p) such that `BR`(t,p), there exists a state s' and a step $s' \; \pi \; s$. We need to show this for steps generated by `Read` and `Write` actions.

There are three cases for the `Read` action depending on the preconditions satisfied. If `c!a /\ ~d(a)`, the state is unchanged and these steps are clearly present in both `LWBCache` and `LWBCacheP` so we need not worry about them.

Next consider `Read` steps taking branch with condition `c!a` and then `d(a)`. Let the transition in `LWBCache` be $t' \; \pi \; t$. Then t has `d(a)=false` so it is mapped by `BR` to the states (t,p) which have `p(a)=false`. Take any such state (t,p). If the transition was generated with `MemWrite` succeeding in writing the value, then in `LWBCacheP` there exists a corresponding transition with `p(a)=false` initially. If the transition was generated with `MemWrite` failing, the corresponding transition in `LWBCacheP` has `p(a)=true` initially. Both states (t',p') that are the source of these transitions are related to t' because `d(a)=true` in t', so `BR` relates t' to both of them.

As the last possibility for `Read` consider the steps $t' \; \pi \; t$ where `~c!a` in t'. These steps act the same on `a` location in both automata, but invoke `Evict(a')` on some location `a'` different from `a`. `Evict(a')` in turn invokes `MemWrite`. Similarly to the previous case, we observe that `d(a')=false` in t, so `p(a')=false` in (t,p). Two possible branches in `MemWrite` correspond to two possible states, one with `p(a')=true` and one with `p(a')=false`. Again both of these are related to t' because `d(a)=true` whenever `MemWrite` is executed.

It remains to consider the `Write` actions $t' \; \pi \; t$. Each `Write` invokes `FlushOne()`. For this part of the transition the proof is the same as for the last case of `Read`, because the precondition and postcondition for location evicted are exactly the same as in the `Read` case. It remains to handle the fact that `Write(a,v)` in `LWBCacheP` executes only when `p(a)=false`. Because `d(a)=true` in t, there are two states (t,p_1) and (t,p_2) related to t, with $p_1(a) = \texttt{true}$ and $p_2(a) = \texttt{false}$. But we can always pick a corresponding state (t,p') where $p'(a) = \texttt{false}$ which will be related to t' regardless of the value of `d(a)` in t'. From this state the write step is enabled and goes into both (t,p_1) and (t,p_2) so both (t,p_1) and (t,p_2) have a write in `LWBCacheP` corresponding to the write in `LWBCache`.

This completes the proof that `BR` is a backward simulation relation from `LWBCache` to `LWBCacheP`, which implies that traces of `LWBCache` are a subset of traces of `LWBCache`.

In the last phase we prove that there exists an abstraction function from `LWBCacheP` to `LMemory`. We define the abstraction function as follows.

```
ABSTRACTION FUNCTION AF(c,um,d,p) -> M =
  RET (\ a | c!a /\ p(a) => RET c(a)
             [*] RET um(a))
```

We call the result of the abstraction function “abstract memory”, this is just the memory in the `LMemory` module. We use the term underlying memory to refer to `um` used in the implementation of `LWBCacheP`.

An image of every initial state of `LWBCacheP` is an initial state of `LMemory` because all states are initial in `LMemory`.

Observe first that `Evict(a)` corresponds to no change of the abstract state under the abstraction function. Clearly locations other than `a` are unchanged by `Evict(a)`. The abstract memory after `Evict(a)` is equal to `um(a)` because `c(a)=undefined`, so we need to show that `AF` would return `um(a)` before `Evict(a)` as well. If `d(a)` is false, then `c(a)=um(a)` before `Evict` so regardless of the case that applies abstract memory has value equal to `um(a)`. If `p(a)` was false initially, then abstraction function initially had the value of `um(a)` as well. If `p(a)` was true initially, then since `Evict(a)` is called only when `c!a`, the first case applied, so abstract memory was equal to the cache in the state before. But if `p(a)` then the content of the cache was copied to the memory, so `um(a)` is equal to the content of the cache before `Evict(a)`.

Next we show that `Read` and `Write` in `LWBCacheP` correspond to `Read` and `Write` in `LMemory`.

There are two cases in the first branch of Read (when `c!a`). If `p(a)` was true, then at the end the underlying memory gets the value of the cache, and the abstract memory was initially defined in terms of cache, so abstract memory before and after are equal. If `p(a)` was false, then write back failed, so both cache and the underlying memory get the value of the underlying memory. But in this case the abstract memory before `Read` was defined in terms of the underlying memory, so it is still unchanged. In both cases `Read` returns the result in the cache, and in both cases the result corresponds to the value of abstract memory. In the second branch of `Read` when `~c!a`, abstract memory is defined in terms of the underlying memory, the value in cache is copied from the underlying memory and the value of underlying memory is thus returned as a result. `FlushOne` is called but, as argued before, it does not change the abstract memory. We conclude that in all cases steps generated by `LWBCacheP.Read` correspond to steps generated by `LMemory.Read`.

Next consider `Write`. Again, calling `FlushOne` does not change the abstract memory. Due to the `~p(a)` guard, whenever `Write` completes we know that the abstract memory was defined in terms of the underlying memory before `Write`. Given the definition of `LMemory.Write`, the resulting value of abstract memory must be either previous value of underlying memory or `v`. Because the value of cache is set to `v`, this is indeed the case.

From this it follows that traces of `LWBCacheP` are subset of the traces of `LMemory`. Previously we have shown that traces of `LWBCache` are a subset of traces of traces of `LWBCacheP` so `LWBCache` implements `LMemory`.