Problem Set 5.5 Solutions

October 10, 2000

You do not have to turn in the solutions to this Problem Set. It is designed to help you study for the first quiz.

———————————————————————————————

Reading: Lecture notes 8-10.

Problem 1 One way to implement a dynamic set, *DSet* is to represent the set by a finite sequence of elements. The **add** operation could append the new element to the end of the sequence *whether or not it already appears in the sequence.* The **lookup** operation returns the answer to whether or not any copy of the element is in the sequence. The **delete** operation deletes the element from the sequence at every place that it occurs.

(a) Define a state machine *Seq* corresponding to this implementation. In implementing the operations, you should not try to represent individual search steps that try to locate an element in the sequence. Instead, let each operation perform its main computation in one large-granularity step, which involves the entire sequence. Thus, this description will still be rather abstract.

Solution:

- Q: The states consist of the components:
  - *seq*, a sequence (ordered) of elements
  - $req \in \{\text{"}add\text{"}, \text{"}delete\text{"}, \text{"}lookup\text{"}\} \cup \{null\}$
  - $resp \in \{\text{"}OK\text{"}, null\} \cup Boolean$
- Q_0: Initially, *seq* has no elements, and all the other components are *null*.
- L: identical to abstract case.
- δ: Input, output are identical to abstract case.

  $perform(add(x))$
  Can occur if: $req = add(x)$ and $resp = null$
  $seq := seq$ with x appended on the end
  $resp :=$ "OK"

  $perform(delete(x))$
  Can occur if $req = delete(x)$ and $resp = null$
  $seq := seq$ compressed to not include any elements equal to x
  $resp :=$ "OK"

perform($lookup(x)$)
Can occur if $req = lookup(x)$ and $resp = null$
if $x \in seq$ then
  $resp := true$
else
  $resp := false$

(b) Prove that your *Seq* state machine implements the *DSet* labeled state machine; that is, prove that any trace of *Seq* is a trace of *DSet*. Remember that a trace is a listing of only the input and output labels during an execution – the externally visible behavior. You may use the Abstraction Theorem.

Solution:

We want to show that any trace of Seq is a trace of $DSet$. In other words, any execution α of Seq has a corresponding execution β in $DSet$ that produces the same externally visible (input and output) labels.

$$trace(\alpha) = trace(\beta)$$

We show this using the Abstraction Theorem.

We need to come up with a suitable relation and prove that it indeed is an Abstraction relation.

Define $R = \{(q, q')|q \in Seq, q' \in DSet, q.req = q'.req, q.resp = q'.resp, unique(q.seq) = q'.set\}$

where $unique(s) = \{setofuniqueelementsins\}$

Now we need to show that R is indeed an abstraction relation. The possible transitions of Seq are:

$request(add(x))$
Also do a $request(add(x))$. Nothing changes besides the identical alterations, and the labels generated are the same. (By the abstraction relation, we know that the req in one execution will be $null$ if and only if it is $null$ in the other).

$request(delete(x))$
Also do a $request(delete(x))$. Nothing changes besides the identical alterations, and the labels generated are the same.

$request(lookup(x))$
Also do a $request(lookup(x))$. Nothing changes besides the identical alterations, and the labels generated are the same.

$perform(add(x))$
Do a $perform(add(x))$. While seq may now have several copies of x, this is allowed by our abstraction relation. Otherwise, the conditions are maintained.

$perform(delete(x))$
Do a $perform(add(x))$. While seq may have had several copies of x, all are deleted by the Seq transition.

$perform(lookup(x))$
Do a $perform(lookup(x))$. Will find the same result in both seq and set, because the abstraction relation says that they have the same elements.

$respond(\text{"}OK\text{"})$
Also do a $return(\text{"}OK\text{"})$. Nothing changes besides the identical alterations, and the labels generated are the same.

$respond(b)$, $b \in Boolean$
Also do a $return(b)$. Nothing changes besides the identical alterations, and the labels generated are the same.

(c) Low-level implementation:

Give an implementation called *SeqLow* that relies on lower-level operations for performing the searches and removals (operate at the element-level).

Solution:

- Q: The states consist of the components:
  - seq, a sequence (ordered) of integers seq_0 through seq_n
  - $index \in \mathbb{N}$
  - $offset \in \mathbb{N}$
  - $maxindex \in \mathbb{N}$
  - $req \in \{\text{"}add\text{"}, \text{"}delete\text{"}, \text{"}lookup\text{"}\} \cup \{null\}$
  - $resp \in \{\text{"}OK\text{"}, null\} \cup Boolean$
- Q_0: Initially, seq has no elements, $index = 0$, $maxindex = 0$, $offset = 0$, and all the other components are $null$.
- L: Input, output identical to abstract case.

  Internal labels:

  $perform(add(x)), \quad x \in \mathbb{Z}$
  $perform(delete(x)), \quad x \in \mathbb{Z}$
  $perform(lookup(x)), \quad x \in \mathbb{Z}$
- δ: Input, output are identical to abstract case.

  $perform(add(x))$
  Can occur if $req = \text{"}add\text{"}$ and $resp = null$
  $seq_{maxindex} := x$
  $maxindex := maxindex + 1$
  $resp := \text{"}OK\text{"}$

  $perform(delete(x))$
  Can occur if $req = \text{"}delete\text{"}$ and $resp = null$
  if $index = maxindex$ then
  $index := 0$
  $offset := 0$
  $maxindex := maxindex - offset$
  $resp := \text{"}OK\text{"}$

else if $seq_{index} = x$ then
  $index := index + 1$
  $\mathit{offset} := offset + 1$
else
  $seq_{index\text{-}\mathit{offset}} := seq_{index}$
  $index := index + 1$

$perform(lookup(x))$
Can occur if $req =$ "$lookup$" and $resp = null$
if $index = maxindex$ then
  $index := 0$
  $resp := false$
else if $seq_{index} = x$ then
  $index := 0$
  $resp := true$
else
  $index := index + 1$

(d) Give an informal argument that *SeqLow* implements *Seq*.

Solution:

We must show that every sequence of *SeqLow* is a sequence of *Seq*. Informally, we can argue that for any execution A of *SeqLow* that has the same trace as an execution B of *Seq*, every transition from A can be matched by some sequence of transitions from B.

Again, the abstraction relation is similar to what we did earlier. Since the exact sequence keeps changing in *SeqLow* when we do a delete operation, we cannot just say that two states are related if they have the same sequence. But we can use the same idea as before and say that two states are related if their sequences have the same unique elements (of course, their *req* and *resp* values must be the same too).

Input and output transitions are exactly matched.

$perform(add(x))$
Also do a $perform(add(x))$. Will add the element to the end in both cases.

$perform(delete(x))$
if $seq_i ndex$ was the last occurence of x or $index = maxindex$ then
Also do a $perform(delete(x))$. In both cases, all elements that match x are gone from the sequence, and the sequences of the resulting states are the same,
else Do nothing. No observable changes; just stepping through details.

$perform(lookup(x))$
if $index = maxindex$ then
Also do a $perform(lookup(x))$. Will return the same value, since we now know that x is not equal to any seq_i where $i < maxindex$.
else if $seq_{index} = x$ then
Also do a $perform(lookup(x))$. Will return the same value, since we know that x is equal to some seq_i (namely, $i = index$).

else
Do nothing. No observable changes; just stepping through details.

Problem 2 Consider an "abstract object" that maintains a dynamic polynomial *poly*. This polynomial has variable x, and integer coefficients. Initially, the polynomial is just the 0 polynomial (no terms). The polynomial supports operations:

- **add-term**(c, e), which adds the term cx^e to the maintained polynomial.
- **coeff**(e), which asks for the coefficient of the x^e term.

(a) Model this object by a labeled state machine *ADP*. The state should keep track of the polynomial (a value that is simply a mathematical polynomial – you don't have to think about how to represent it using lower-level mathematical objects like numbers, matrices, sequences, etc.) The state will also need some bookkeeping information to keep track of operations in progress. The labels should include inputs reflecting the requests and outputs reflecting the responses. Your machine should ignore requests that arrive while an operation is active.

Solution:

- Q:
  $p \in \{polynomials\}$
  $req \in \{\text{“}add\text{-}term\text{”}, \text{“}find\text{-}coeff\text{”}, null\} \cup \{\text{“}OK\text{”}\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$
- Q_0:
  A set with only one element:
  $\{(0, null, null, null)\}$
  or, equivalently,
  $p = 0$, $req = null$, $power = null$, $mult = null$
- L:

  Input labels:
    $req\text{-}add\text{-}term(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
    $req\text{-}coeff(e), \quad e \in \mathbb{N}$

  Internal labels:
    $comp\text{-}sum(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
    $comp\text{-}coeff(e), \quad e \in \mathbb{N}$

  Output labels:
    $return(\text{“}OK\text{”})$
    $return(c), \quad c \in \mathbb{Z}$

- δ:
  Each is associated with (generates) the label of its namesake.

  1. $req\text{-}add\text{-}term(c, e)$:
     Can occur anytime.
     if $req = null$ then
     $req :=$ "add"
     $power := e$
     $mult := c$
  2. $req\text{-}coeff(e)$:
     Can occur anytime.
     if $req = null$ then
     $req :=$ "$find\text{-}coeff$"
     $power := e$
     $mult := null$
  3. $comp\text{-}sum(c, e)$:
     Can occur if $req =$ "$add\text{-}term$", $mult = c$, and $power = e$.
     $p := p + c \cdot x^e$
     $req :=$ "OK"
  4. $comp\text{-}coeff(e)$:
     Can occur if $req =$ "$find\text{-}coeff$" and $power = e$.
     $req :=$ coefficient of x^e in p
  5. $return($"OK"$)$:
     Can occur if $req =$ "OK".
     $req, power, mult := null$
  6. $return(c)$:
     Can occur if $req = c$.
     $req, power, mult := null$

(b) List some (4–5) traces that are exhibited by your state machine ADP. Include at least one that involves a request arriving when it is not supposed to.

Solution:

Start with $(0, null, null, null)$:

$$
\begin{array}{l}
req\text{-}add\text{-}term(1,1) \\
return(\text{“}OK\text{”}) \\
req\text{-}add\text{-}term(2,2) \\
return(\text{“}OK\text{”}) \\
req\text{-}coeff(2) \\
return(2) \\
req\text{-}add\text{-}term(2,1) \\
return(\text{“}OK\text{”}) \\
req\text{-}coeff(1) \\
return(3)
\end{array}
$$

Start with $(0, null, null, null)$:

$$\begin{array}{l} req\text{-}add\text{-}term(1,1) \\ return(\text{“}OK\text{”}) \\ req\text{-}add\text{-}term(2,2) \\ req\text{-}coeff(2) \\ return(\text{“}OK\text{”}) \\ req\text{-}coeff(2) \\ req\text{-}add\text{-}term(3,2) \\ return(2) \\ req\text{-}coeff(2) \\ return(2) \end{array}$$

Start with $(0, null, null, null)$:

$$\begin{array}{l} req\text{-}add\text{-}term(4,2) \\ req\text{-}add\text{-}term(7,1) \\ return(\text{“}OK\text{”}) \\ req\text{-}coeff(3) \\ req\text{-}coeff(2) \\ return(0) \\ req\text{-}coeff(1) \\ return(0) \\ req\text{-}coeff(2) \\ return(4) \end{array}$$

Of course, there are many more traces. The important thing about this system is that requests will be dropped if there is still a pending request that has not generated a *return* – the value of *req* will not be *null* when they come in.

(c) Describe in words an implementation of your state machine *ADP*. The state of this implementation should involve lower-level mathematical objects like numbers, matrices, sequences, etc.

Solution:

Using Arrays: Keep an array *poly* such that the ith entry holds the coefficient of x^i in p. The array size will have to grow as necessary, or there will be a limit on the terms of the polynomial. $comp\text{-}add\text{-}term(c, e)$ will simply be $poly[e] := poly[e] + c$. $comp\text{-}coeff(e)$ will be $req := poly[e]$.

Using Ordered Pairs: Keep a set *terms* of ordered pairs (i, c) mapping the ith entry to the coefficient of x^i in p. $comp\text{-}add\text{-}term(c, e)$ will be more complex, unless we just add the pair and sum all pairs with like first components on lookup. We will have to search through for an existing pair with a matching first component. This will require more internal state. $comp\text{-}coeff(e)$ will be a similar search for the right pair, returning 0 if none is found.

(d) Formalize your implementation as a labeled state machine *Imp*. It should have the same external labels as your labeled state machine *ADP*.

Solution:

Using Arrays: Assume that we ignore the bounds of the array.

We need the following modifications to the abstract machine:

- Q:
  For all (many) $i \in \mathbb{N}$,
  $poly[i] \in \mathbb{Z}$
  $req \in \{\text{"}add\text{-}term\text{"}, \text{"}find\text{-}coeff\text{"}, null\} \cup \{\text{"}OK\text{"}\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$
- Q_0:
  A set with only one element:
  $poly[i] = 0$for all i, $req = null$, $power = null$, $mult = null$
- L: Same as before.
- δ: Same as before for input and output.
  - $comp\text{-}sum(c, e)$:
    Can occur if $req =$ "$add\text{-}term$", $mult = c$, and $power = e$.
    $poly[e] := poly[e] + c$
    $req :=$ "OK"
  - $comp\text{-}coeff(e)$:
    Can occur if $req =$ "$find\text{-}coeff$" and $power = e$.
    $req := poly[e]$

Using Unique Ordered Pairs: We need the following modifications to the abstract machine:

- Q:
  A linked list $terms$ of the form $(i, m, next)$ with $i \in \mathbb{N}$, $m \in \mathbb{Z}$, $next \in terms \cup \{null\}$.
  $firstterm \in terms$
  $curterm \in terms \cup \{null\}$
  $req \in \{\text{"}add\text{-}term\text{"}, \text{"}find\text{-}coeff\text{"}, null\} \cup \{\text{"}OK\text{"}\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$
- Q_0:
  A set with only one element:
  $firstterm = (0, 0, null)$, $curterm = firstterm$, $terms = \{firstterm\}$, $req = null$, $power = null$, $mult = null$
- L: Same as before for input and output labels.

  Internal labels:

  $comp\text{-}sum(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
  $comp\text{-}sum\text{-}next(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$

$comp\text{-}sum\text{-}default(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
$comp\text{-}coeff(e), \quad e \in \mathbb{N}$
$comp\text{-}coeff\text{-}next(e), \quad e \in \mathbb{N}$
$comp\text{-}coeff\text{-}default(e), \quad e \in \mathbb{N}$

- δ: Input, output transitions same as before.
  - $comp\text{-}sum(c, e)$:
    Can occur if $req =$ "*add-term*", $mult = c$, $power = e$, $curterm \neq null$, $curterm.i = e$
    $curterm.m := curterm.m + c$
    $req :=$ "OK"
    $curterm := firstterm$
  - $comp\text{-}sum\text{-}next(c, e)$:
    Can occur if $req =$ "*add-term*", $mult = c$, $power = e$, $curterm \neq null$, $curterm.i \neq e$
    $curterm := curterm.next$
  - $comp\text{-}sum\text{-}default(c, e)$:
    Can occur if $req =$ "*add-term*", $mult = c$, $power = e$, $curterm = null$
    $firstterm := (e, c, firstterm)$
    $req :=$ "OK"
    $curterm := firstterm$
  - $comp\text{-}coeff(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm \neq null$, $curterm.i = e$
    $req := curterm.m$
    $curterm := firstterm$
  - $comp\text{-}coeff\text{-}next(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm \neq null$, $curterm.i \neq e$
    $curterm := curterm.next$
  - $comp\text{-}coeff\text{-}default(e)$:
    Can occur if $req =$ "$find\text{-}coeff$", $power = e$, $curterm = null$
    $req := 0$
    $curterm := firstterm$

Note that we implicitly added a new term to *terms* every time we expanded the linked list.

Using Multiple Ordered Pairs: We need the following modifications to the abstract machine:

- Q:
  A linked list *terms* of the form $(i, m, next)$ with $i \in \mathbb{N}$, $m \in \mathbb{Z}$, $next \in terms \cup \{null\}$.
  $firstterm \in terms$
  $curterm \in terms \cup \{null\}$
  $total \in \mathbb{Z}$
  $req \in \{$"*add-term*", "$find\text{-}coeff$", $null\} \cup \{$"OK"$\} \cup \mathbb{Z}$
  $power \in \mathbb{N} \cup \{null\}$
  $mult \in \mathbb{Z} \cup \{null\}$

- Q_0:
  A set with only one element:
  $firstterm = (0, 0, null)$, $curterm = firstterm$, $terms = \{firstterm\}$, $total = 0$, $req = null$, $power = null$, $mult = null$
- L: Same as before for input and output labels.

  Internal labels:

  $comp\text{-}sum(c, e), \quad c \in \mathbb{Z}, e \in \mathbb{N}$
  $comp\text{-}coeff(e), \quad e \in \mathbb{N}$
  $comp\text{-}coeff\text{-}next(e), \quad e \in \mathbb{N}$
  $comp\text{-}coeff\text{-}end(e), \quad e \in \mathbb{N}$
- δ: Input, output transitions same as before.
  - $comp\text{-}sum(c, e)$:
    Can occur if $req =$ "*add-term*", $mult = c$, and $power = e$
    $firstterm := (e, c, curterm)$
    $req :=$ "OK"
  - $comp\text{-}coeff(e)$:
    Can occur if $req =$ "*find-coeff*", $power = e$, $curterm \neq null$, and $curterm.i = e$
    $total := total + curterm.m$
    $curterm := curterm.next$
  - $comp\text{-}coeff\text{-}next(e)$:
    Can occur if $req =$ "*find-coeff*", $power = e$, $curterm \neq null$, and $curterm.i \neq e$
    $curterm := curterm.next$
  - $comp\text{-}coeff\text{-}end(e)$:
    Can occur if $req =$ "*find-coeff*", $power = e$, and $curterm = null$
    $req := total$
    $curterm := firstterm$
    $total := 0$

(e) Argue that your implementation *Imp* is correct, in the sense that every trace of *Imp* is also a trace of *ADP*.

Solution:

Using Arrays: This is the easiest.

The abstraction relation is obvious: All corresponding elements have the same values and for every term cx^i in p, $poly[i] = c$. Only the differing transitions are nontrivial to examine. (Otherwise, the executions are identical and the abstraction relation is clearly preserved). However, even *comp-sum* and *comp-coeff* are easily accounted for: if an execution of *Imp* is extended by either, extend an execution of *ADP* with the same trace using the same transition. *comp-sum* preserves the abstraction relation by performing matching arithmetic. *comp-coeff* preserves the abstraction relation because p and *poly* are in correspondence and will set *req* to the same value.

Using Unique Pairs: Somewhat tricky to do formally.

The abstraction relation is: elements of the same name have the same values; every non-zero term in p has exactly one corresponding triplet in $terms$, and every zero-coefficient term in p either has no corresponding triplet or has a triplet with a second component of 0.

We need other elements of an invariant: In Imp, $firstterm$ begins a linked list that reaches all terms in $terms$; there are no terms with $i = mult$ 'between' $firstterm$ and $curterm$; no two terms have the same i component; if req is not "$find\text{-}coeff$" or "$add\text{-}term$" then $curterm = firstterm$.

Input and output transitions have a simple identity correspondence. Roughly, if the last transition of the partial execution of Imp is:

comp-sum$(\mathbf{c}, \mathbf{e})$
perform a $comp\text{-}sum(c, e)$ transition. Both set req to "OK"; the assignments keep the polynomial structures aligned. (We use the fact that the IH tells us that no other element of $terms$ represents the same exponent here). Although req is changed, $curterm$ is set to $firstterm$. Nothing else is changed.

comp-sum-next$(\mathbf{c}, \mathbf{e})$
do nothing. No data is changed, and the invariant holds over the change in $curterm$ because $i \neq mult$ for the old $curterm$.

comp-sum-default$(\mathbf{c}, \mathbf{e})$
perform a $comp\text{-}sum(c, e)$ transition. If there are no terms after $curterm$, and no terms with $i = mult$ between $firstterm$ and $curterm$, then there must be no terms at all with $i = mult$. So adding in the new term did not violate uniqueness. The new $firstterm$ is connected to all of the elements of $terms$ because the old $firstterm$ was.

comp-coeff$(\mathbf{e})$
perform a $comp\text{-}coeff(e)$ transition. Must be the correct coefficient because p and $terms$ are in correspondence and no two terms have the same i component.

comp-coeff-next$(\mathbf{e})$
do nothing. No data is changed, and the invariant holds over the change in $curterm$ because $i \neq mult$ for the old $curterm$.

comp-coeff-default$(\mathbf{e})$
perform a $comp\text{-}coeff(e)$ transition. If there are no terms after $curterm$, and no terms with $i = mult$ between $firstterm$ and $curterm$, then there must be no terms at all with $i = mult$. By the correspondence between p and $terms$, the coefficient must be 0 in both ADP and Imp.

Although the steps are not filled in, the above is a basic reasoning that traces of Imp can be matched by traces of ADP.

Using Multiple Pairs: Slightly trickier to do formally.

The abstraction relation is: elements of the same name have the same values; every term in p has a coefficient equal to the sum of m in all triplets $(i, m, next)$ in $terms$ with i equal to that coefficient.

We need other elements of an invariant: In Imp, $firstterm$ begins a linked list that reaches all terms in $terms$; $total$ is the sum of m in all triplets $(i, m, next)$ between $firstterm$ and $curterm$; if req is not "$find\text{-}coeff$" then $curterm = firstterm$.

Input and output transitions have a simple identity correspondence. Roughly, if the last transition of the partial execution of Imp is:

comp-sum(c, e)
perform a *comp-sum*(c, e) transition. Both set req to "OK". The assignments keep the polynomial structures aligned by adding in the new coefficient. The new $firstterm$ is connected to all of the elements of $terms$ because the old $firstterm$ was. Nothing else is changed.

comp-coeff(e)
do nothing. $total$ is appropriately increased, since $curterm$ is advanced beyond the term that is contributing.

comp-coeff-next(e)
do nothing. No data is changed, and the invariant holds over the change in $curterm$ because $i \neq mult$ for the old $curterm$.

comp-coeff-end(e)
perform a *comp-*$coeff(e)$ transition. If there are no terms after $curterm$, then all terms with $i = mult$ are between $firstterm$ and $curterm$. Thus, $total$ must be the proper coefficient of term e. Since p and $terms$ are in correspondence, Imp and ADP will update req to the same value.

Problem 3 Prove the Abstraction Theorem:
A implements B **iff** there is an abstraction relation from A to B

Solution:

($\Leftarrow$) (By induction)

Proof. Assume R is an abstraction relation from A to B.

Given an arbitrary execution a of A, construct a corresponding execution b for B inductively as follows:

For B's start state, pick a start state that's related to A's start state, ie, pick q_0^b such that $(q_0^a, q_0^b) \in$ R.

For each successive transition in A's execution, $q_a \rightarrow q_a{}'$, replace it by a (possibly empty) sequence of transitions $q_b q_b{}'$ such that $(q_a{}', q_b{}') \in$ R. This is guaranteed since R is an abstraction relation.

Now it's straightforward to prove that the traces of the two executions are the same.

Let P(n): The traces of the n-length prefix of A's execution and the *corresponding* part of B's execution are the same.

P(0) is trivially true.

P(n-1) and the fact that the sequence of B's execution corresponding to the n^{th} transition of A has the same trace implies P(n).

Since this is true for any execution of A it follows that A implements B. ∎

($\Rightarrow$) (By contradiction)

Proof. Assume there exists **no** abstraction relation from A to B.

In particular, R = A x B is also not an abstraction relation.

This implies that $\exists q_a, q_b, l$ such that $q_a \xrightarrow{l} q_a{}'$ and
$\forall s(q_b \, {}^s \, q_b{}' \Rightarrow \text{Trace(s)} \neq \text{Trace(l)})$.

Now we intend to show an execution of A whose trace is different from any execution of B's.

Consider an execution of A that ends in q_a such that the only execution of B that matches its trace ends in q_b (If this is not possible, we have two cases. Either no execution of B can match A's trace, in which case we are done. Or B's executions always end in states other than q_b in which case there's no need for q_a and q_b to be related to each other and we can start the arguement all over again with R - (q_a,q_b)).

Now we extend A's execution with the transition l. From our choice of q_a and q_b it's obvious that B cannot match the trace of this transition.

Thus, we now have an execution of A whose trace is different from any execution of B.

Hence, A does not implement B. ∎

Problem 4 Seeing how annoyingly slow the Student Center elevator is, the MIT Administration has decided to give you, the students in 6.042, the job of designing the elevator system for the new LCS building. Your first task is to specify the behavior of the elevator at an abstract level.

(a) Assume you have n floors.
The inputs to the elevator controller are of three types:

- *up*(i) (Person on i^{th} floor wants to go up)
- *down*(i) (Person on i^{th} floor wants to go down)
- *press*(i) (Person inside the elevator wants to go to the i^{th} floor)

The controller has two outputs:

- *open* (The elevator door opens at current floor)
- *close* (The elevator door closes at current floor)

The controller also has two internal transitions:

- *goUp* (Causes the elevator to go up one floor)
- *goDown* (Causes the elevator to go down one floor)

The internal state consists of:

- *floor* (The floor that the elevator is currently on)
- *requests* (The set of all pending floor numbers)
- *isClosed* (Yes, if door is closed, No otherwise)

Specify the basic behavior that you'd expect from this elevator controller (there is no unique solution; come up with something that you feel is reasonable).

Solution:

- Q: The states consist of the components *floor*, *requests* and *isClosed*
- Q_0: *floor* $= 1$ and *requests* $= \emptyset$ and *isClosed* $= Yes$
- L:

  Input labels:

  $up(i), down(i), press(i)$

  Internal labels:

  $goUp, goDown$

  Output labels:

  $open, close$

- δ:
  Each transition is associated with the label of its namesake.

  1. $up(i)$, $down(i)$, $press(i)$:
     Can occur anytime.
     if *floor* $\neq i$ or *isClosed* $= Yes$ then
     requests := *requests* $\cup$ {i}
  2. *goUp*:
     Can occur if *isClosed* = Yes and $\exists i \in$ *requests* $(i >$ *floor*$)$.
     floor := *floor* + 1;
  3. *goDown*:
     Can occur if *isClosed* = Yes and $\exists i \in$ *requests* $(i <$ *floor*$)$.
     floor := *floor* - 1;
  4. *open*:
     Can occur if *isClosed* = Yes and $\exists i \in$ *requests* $(i =$ *floor*$)$.
     isClosed := No;
     requests := *requests* - {i};
  5. *close*:
     Can occur if *isClosed* = No
     isClosed := Yes;

Note that this description is non-deterministic.

(b) You must have noticed that the behavior (transitions) of the elevator depends on the service policy. Specify the behavior when the elevator follows a greedy policy (ie, the elevator services the request closest to its current position)

Solution:

Only the *goUp* and *goDown* transitions are modified.

1. *goUp*:
   Can occur if *isClosed* = Yes and $\exists i \in requests\,(i > floor \wedge |i - floor|\ is\ minimum)$.
   floor := *floor* + 1 ;

2. *goDown*:
   Can occur if *isClosed* = Yes and $\exists i \in requests\,(i < floor \wedge |floor - i|\ is\ minimum)$.
   floor := *floor* - 1;

(c) This policy might be lead to *starvation* of certain requests (as is common at the Student Center). Show a trace where a request might be denied service for an arbitrary long period of time.

Solution:

Here's the trace of a scenario where a person on the n^{th} floor wants to come down but is frustrated by a guy who keeps hopping between the 1^{st} and 2^{nd} floors.

down(n), up(1), open, close, press(2), open, close, press(1), open, close, press(2) ...

(d) Let's modify the service policy to take care of this problem. The elevator will keep going in one direction till it has serviced all requests for that direction. Then it reverses direction and keeps going the other way. Specify the behavior of the elevator under this policy. Argue its correctness.

(Hint: In order to implement this policy, you'll need to add additional variables to the state. In particular, you'll have to keep track of the current direction of the elevator. It might also be helpful to split the set of requests into an *up* set and a *down* set)

Solution:

- Q: The states now consist of the components *floor*, *upRequests*, *downRequests*,*isClosed* and *dir*
- Q_0: *floor* = 1, *upRequests*, *downRequests* = $\emptyset$, *isClosed* = Yes, *dir* = Up
- L:
  Same as before with the following additional internal transition:

  reverseDir (for switching directions of the elevator)

- δ:

  1. *up(i)*:
     Can occur anytime.
     if *floor* $\neq i$ or *isClosed* = Yes then
     upRequests := *upRequests* $\cup$ {i}
  2. *down(i)*:
     Can occur anytime.
     if *floor* $\neq i$ or *isClosed* = Yes then
     downRequests := *downRequests* $\cup$ {i}
  3. *press(i)*:
     Can occur anytime.
     if *floor* $\neq i$ or *isClosed* = Yes then

if i $\geq$ *floor* then
upRequests := *upRequests* $\cup$ {i}
else
downRequests := *downRequests* $\cup$ {i}

4. *goUp*:
   Can occur if *isClosed* = Yes and *dir* = Up and *upRequests* $\neq \emptyset$ *floor* := *floor* + 1;
5. *goDown*:
   Can occur if *isClosed* = Yes and *dir* = Down and *downRequests* $\neq \emptyset$ *floor* := *floor* - 1;
6. *reverseDir*:
   Can occur if (*dir* = Up and *upRequests* = $\emptyset$) or (*dir* = Down and *downRequests* = $\emptyset$)
   dir := *opposite*(*dir*);
7. *open*:
   Can occur if *isClosed* = Yes and (*condition1* or *condition2*)
   where *condition1* = (*dir* = Up and $\exists i \in$ *upRequests* (i = *floor*))
   and *condition2* = (*dir* = Down and $\exists i \in$ *downRequests* (i = *floor*))
   isClosed := No;
   if *condition1* then
   upRequests := *upRequests* - {i};
   else
   downRequests := *downRequests* - {i};

8. *close*:
   Can occur if *isClosed* = No
   isClosed := Yes;

Incidentally, this strategy is commonly used in different applications such as disk scheduling where it goes under the not so surprising name of the *elevator algorithm* !