MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
6.826 *Principles of Computer Systems*

February 15, 2000 **Due: Thursday, February 24, 2000**

Problem Set 3

Instructions: There are **five** *problems in this problem set;* **please turn in each problem on a separate sheet of paper with your name at the top, or you will run the risk of your solutions being lost**. *Also give the amount of time you spend on each problem.*

Problem 1

The following is a specification for a database that provides the first and second order statistics on a totally-ordered set.

```
MODULE OrderStatDB [ V WITH {``<='': (V,V) -> V} ] % comparison operator
  EXPORT {add, first, second} =

  VAR db : SEQ V := {}

  APROC add(v) = << db + := {v}; RET >>
  APROC first() -> V = << VAR v | (ALL i | db!i ==> db(i) <= v) >>
  APROC second() -> V = << VAR v | (EXISTS j | v <= db(j) /\
    (ALL i | db!i /\ i # j ==> db(i) <= v)) >>
End OrderStatDB
```

(a) Write an optimized implementation of `OrderStatDB`, `OrderStatDBImpl`, that uses the minimum amount of state, and performs the minimum amount of comparison operations for each of the three methods.

(b) Augment your implementation with a sufficient amount of history state (call the new implementation `OrderStatDBImplH` so that you can write an abstraction function from `OrderStatDBImplH` to `OrderStatDB`.

(c) Using the abstraction function you just defined, prove that `OrderStatDBImplH` implements `OrderStatDB`.

(d) Now write an abstraction *relation* from the original `OrderStatDBImpl` to `OrderStatDB`. You need not prove the implements property again. Comment on the relative difficulty of the two approaches for this particular problem.

Problem 2

```
MODULE RestrictedSeq [V]

  VAR arr : SEQ V := {}
```

```
  INT lower := 0;

  APROC add(v) = << arr + := {v}; RET >>
  APROC get(i) -> V = << i >= lower => RET v(i) >>
  APROC restrict() = << lower + := 1 >>
END RestrictedSeq
```

(a) Write an optimized implementation of `RestrictedSeq`, `RestrictedSeqImplH`, that uses the minimum amount of state possible.

(b) Augment your implementation with a sufficient amount of history state (call the new implementation `RestrictedSeqImplH` so that you can write an abstraction function from `RestrictedSeqImplH` to `RestrictedSeq`.

(c) Using the abstraction function you just defined, prove that `RestrictedSeqImplH` implements `RestrictedSeq`.

(d) Now write an abstraction *relation* from the original `RestrictedSeqImpl` to `RestrictedSeq`. You need not prove the implements property again. Comment on the relative difficulty of the two approaches for this particular problem.

Problem 3

Tired of Course VI, Louis Reasoner decides to seek a UROP in Course VIII investigating quantum mechanics. For his first task, the professor gives Ben the following specification for a quantum-mechanical experiment to measure the spin direction of an electron injected into a magnetic field:

```
MODULE QE EXPORTS {inject, observe}
  TYPE O = ENUM [up, down]
       Q = SEQ O

  VAR q

  APROC inject() = << q + := {up} [] q + := {down} >>
  APROC observe() -> O = << VAR o | o = q.head => q := q.tail; RET o >>
END QE
```

Louis blithely assumes that collapse of the wave function only occurs after an observation is made. Following this assumption, he decides to implement his quantum-mechanical experiment simulation as follows:

```
MODULE QEI EXPORTS {inject, observe}
  TYPE O = ENUM [up, down]
  VAR count: INT := 0

  APROC inject() = << count + := 1 >>
  APROC observe() = << count > 0 => count - := 1; RET up [] RET down >>
END LouisQE
```

(a) Write a new module called QEP. QEP should augment the state of QEI to include the prophecy variables necessary to prove that QEI implements QEP.

(b) Give appropriate abstraction functions from QEP to QE, and from QEP to QEI.

(c) Using one abstraction function you gave, prove that QEP implements QE.

(d) Finally, prove that QEI implements QEP. You should prove a lemma about the state space and transitions of QEP.

Problem 4

Consider the following two specifications for reliable FIFO channels:

```
MODULE Channel [M, A] EXPORTS {put, get} = % M = message, A = address
  TYPE Q = SEQ M
       SR = [s: A, r: A]

  VAR q := (SR -> Q){* -> {}}

  APROC put(sr, m) = << q(sr) := q(sr) + {m} >>

  APROC get(sr) -> M = << VAR m | m = q(sr).head =>
    q(sr) := q(sr).tail; RET m >>
END Channel

MODULE BigChannelImpl [M, A] EXPORTS {put, get} = % M = message, A = address
  TYPE Q = SEQ M
       SR = [s: A, r: A]

  VAR q1 := (SR -> Q) {* -> {}},
      q2 := (SR -> Q) {* -> {}}

  APROC put(sr, m) = << q2(sr) := q2(sr) + {m} >>

  APROC get(sr, m) = << VAR m | m = q1(sr).head =>
    q1(sr) := q1(sr).tail; RET m >>

  THREAD xfer() = << DO VAR m | m = q2(sr).head =>
    q2(sr) := q2(sr).tail; q1(sr) := q1(sr) + {m} OD >>
END BigChannelImpl
```

Give a rigorous proof that BigChannelImpl implements Channel, using history and/or prophecy variables and abstraction functions as appropriate.

Problem 5

Given the following module:

```
MODULE DoNothing
END DoNothing
```

(a) Argue that this module implements any specification.