

Concurrent Programming Lecture 4

5th September 2002

Comments on homework 2

Do not assume that because there are no marks on a problem that it is correct! Many solutions were so *incorrect* that I couldn't begin to mark what needed fixing, especially in 2.22 and 2.30. If a solution was correct I tried to mark it *ok*.

A large fraction of the class needs much more practice in translating from English descriptions of properties to the corresponding first-order predicates. In addition to problems getting the logic correct, some people are writing formulas that aren't even well-formed. If you haven't reviewed Rick Sheldon's tutorial yet, please do. And if your formulas aren't correct, review it again.

DeMorgan's rules: $\neg(A \wedge B) \equiv \neg A \vee \neg B$, $\neg(A \vee B) \equiv \neg A \wedge \neg B$

DeMorgan's rules for quantifiers: $\neg \forall x \bullet P(x) \equiv \exists x \bullet \neg P(x)$, $\neg \exists x \bullet P(x) \equiv \forall x \bullet \neg P(x)$

Type checking: what does an inference rule look like? Did you write an inference rule or something else for 2.22? Does $i < \{I\}$ make sense? Does $i \wedge j \wedge k$ make sense if i, j , and k are all integer variables? Informally, what are the types of x and e in the assignment axiom? What about I and B in the WHILE rule?

Magic (also known as pulling rabbits out of hats): in 2.30 on what evidence do you assume that you can limit looking at $f()$, $g()$, and $h()$ to parameter values less than some arbitrary n or even separate limits m , n , p for the different functions? {In my solution below I seemingly pull \min out of a hat – but I justify its existence from the givens of the problems and I use it only in the proof and not as a program variable.}

Changing the rules: example: if the question is about the at-most-once property it means “according to the definition”, not “according to some changed version of the definition that you just made up”. If you don't know how to solve a problem as given, it *may* be permissible to alter the problem definition to one you can solve (researchers do this all the time, of course, in order to find tractable subproblems of the problems they would really like to solve.) However, if you change the problem definition you are obligated to state the original problem

definition, state why you believe it is intractable as given, explain how you modified it and explain why you made the particular modifications that you did.

Suggestion: make it a habit on homeworks, exams, projects in this class and others to check your solutions for type-checking failures, use of magic, and places where you've changed the rules. If any of these occur, your solution is not a solution and needs to be changed. I find that type-checking is often a good algorithm to run mentally while listening to technical talks. By tracking "what kind of thing" each symbol refers to I find that I understand better what is being said and sometimes gain insights that even the presenter has missed.

Non-interference (continued)

Global invariants

I is a global invariant if I is true when the processes are created and I is preserved by every assignment action in every process.

If a critical assertion, C in process P , can be written as $I \wedge L$ where I is a global invariant and L references only local variables of P and global variables assigned to only by P , then C is not interfered with by any assignment action.

Synchronization

Since assertions within await statements are not critical assertions we do not have to ensure local non-interference for them in order to ensure global non-interference. Making a collection of assignment statements atomic may make our programs and proofs simpler.

Example

The example consists of a pair of processes that copy an array item by item through a buffer. The example is less artificial than it may seem: it is an example of the use of a *bounded buffer* to communicate between two processes. The techniques used are similar to what is required to implement Unix pipes or any stream-oriented communication between local processes in an operating system.

In this example the size of the buffer is one element. When we look at other synchronization methods we will examine approaches to generalizing to any fixed size buffer.

Simplification compared to Fig. 2.4: assume that $a[]$ is a constant array so I don't have to show that a is unchanged by my program.

We'll use global variables c and p to step through the arrays in the consumer and producer processes.

Choose a global invariant:

$$GI: c \leq p \leq c+1 \wedge ((p==c+1) \Rightarrow (buf == a[p-1]))$$

The consumer and producer are either at the same position or the producer is one slot ahead. When the producer is one slot ahead, the buffer contents are valid, otherwise not.

Our goal is to establish at the end of the consumer

$$b[0..n-1] == a[0..n-1]$$

By now the iteration pattern for visiting all the elements of an array should be becoming familiar. The pattern will be used in both the producer and consumer.

```

int buf, p=0, c=0;
{GI}
process Producer
  const int a[n] -- assume initialized
  {PI: GI  $\wedge$  p<=n} -- producer invariant
  while (p<n)
    {PI  $\wedge$  p<n}
    <await p==c>
(7) {PI  $\wedge$  p<n  $\wedge$  p==c}
    buf = a[p]
(1) {PI  $\wedge$  p<n  $\wedge$  p==c  $\wedge$  buf==a[p]}
(2) p = p+1
(3) {PI}
    {PI  $\wedge$  p==n}

process Consumer
  int b[n]
  {CI: GI  $\wedge$  c<=n  $\wedge$  b[0..c-1]==a[0..c-1]}
  while (c<n)
    {CI  $\wedge$  c<n}
    <await p==c+1>
(8) {CI  $\wedge$  c<n  $\wedge$  p==c+1}
    b[c] = buf
(4) {CI  $\wedge$  c<n  $\wedge$  p==c+1  $\wedge$  b[c]==a[c]}
(5) c = c+1
(6) {CI}
    {CI  $\wedge$  c==n}
    b[0..n-1]==a[0..n-1]

```

Verify that lines labelled 1, 2, and 3 above constitute a theorem:

- (1) $\{PI \wedge p < n \wedge p == c \wedge buf == a[p]\}$
 $\{c \leq p+1 \leq c+1 \wedge p < n \wedge p == c \wedge buf == a[p]\}$
 $\{c \leq p+1 \leq c+1 \wedge ((p == c) \Rightarrow (buf == a[p])) \wedge p < n\}$
- (1') $\{c \leq p+1 \leq c+1 \wedge ((p+1 == c+1) \Rightarrow (buf == a[p])) \wedge p+1 < n\}$
- (2) $p = p+1$
- (3) $\{PI: c \leq p \leq c+1 \wedge ((p == c+1) \Rightarrow (buf == a[p-1])) \wedge p < n\}$

The unnumbered lines following line 1 follow from line 1 by logical consequence, and lines 1', 2 and 3 are a theorem by the assignment axiom, so 1, 2, and 3 are a theorem. Notice how we didn't use anything from PI in line 1. Proving that 4, 5, and 6 constitute a theorem is left for an exercise.

Non-interference: except for lines 1, 7, 4, and 8 all of the critical assertions are of form GIAL, so non-interference for them is automatic.

Let's consider line 1. Call the assertion there C. The only assignment action that could interfere with it is on line 5. Proceeding mechanically we have to show that $\{pre(c=c+1) \wedge C\} c=c+1 \{C\}$ is a theorem (local non-interference). Observe that $\{pre(c=c+1) \wedge C\}$ is *false* because it contains $(p == c) \wedge (p == c+1)$. So by the consequence rule, $\{pre(c=c+1) \wedge C\} c=c+1 \{C\}$ is a theorem.

Exactly the same thing happens in the other 3 required proofs.

Global safety properties

Something BAD does not happen

Examples of BAD things: deadlock – several processes each waiting for another to do something; letting two processes use a resource simultaneously. For sequential programs, absence of BAD behavior is captured in the assertions attached to program locations, but for concurrent programs the bad behavior whose absence must be established may involve multiple processes.

Steps: express the bad behavior as a conjunction of assertions appearing in a proof outline for the concurrent program. Show that the resulting assertion is *false*. Since each of the individual assertions has been established by proof to be true whenever execution is at that point in the program, if the conjunction is false, the processes can't all be at that set of points at the same time.

As an example, in producer-consumer problem that we just finished looking at, in the course of showing non-interference we established that the two processes would not be at their $p=p+1$ and $c=c+1$ statements at the same time (mutual exclusion).

We would also want to show freedom from deadlock. In this program deadlock occurs if both processes block at their await statements at the same time.

For both of them to be blocked we must have $p \neq c \wedge p \neq c+1$, but the global invariant which is a precondition of both await statements says that at all times $p = c \vee p = c+1$. The conjunction of all of these is *false*, hence there is no deadlock.

Liveness properties

Examples: the program reaches its final state (terminates); whenever the program reads a keystroke it echoes it to the screen;

Liveness depends on scheduling:

Programs can have different liveness behavior depending on precisely how their instructions are interleaved. How instructions are interleaved is a result of a *scheduling policy*. Some interesting properties of scheduling policies are:

Unconditional Fairness: A scheduling policy is unconditionally fair if every unconditional atomic action that is eligible is executed eventually.

Weak Fairness: A scheduling policy is weakly fair if it is unconditionally fair and every eligible conditional atomic action whose condition becomes and remains true is executed eventually.

Strong Fairness: A scheduling policy is strongly fair if it is unconditionally fair and every eligible conditional atomic action whose condition becomes true infinitely often (infinitely many times) is executed eventually. (Terminology: infinitely often means infinitely many times – it has nothing to do with frequency in the sense that EEs understand it).

Exercises for next time

Verify that lines 4, 5, and 6 constitute a theorem in the Consumer process above. The proof should look much like the proof for lines 1, 2, and 3 covered in class today.

Solutions to selected exercises from Homework 2

2.12

In part a, since the two statements are atomic, the only possible executions are one first and then the other. So the results are either $(x=5, y=15)$ or $(x=8, y=6)$. If the statements are not atomic, both expressions can be evaluated before either variable is assigned so a third possibility arises $(x=5, y=6)$.

2.14 and 2.20

Discussed in Lecture 3

2.22

The for loop written as an annotated while loop looks like:

```

{I}
i=1
{I}
while (i≤n)
  {I∧i≤n}
  S;
  i = i+1
  {I}
{I∧i>n}

```

So one way to proceed is to make a so-called *side-condition* on our new rule that i not appear free in I , in which case the above requires that $\{I \wedge i \leq n\} S \{I\}$ be a theorem and our FOR rule would be:

FOR: if $\{I \wedge i \leq n\} S \{I\}$ is a theorem and i does not appear free in I , then $\{I\}$ for $[i=1 \text{ to } n] \{I \wedge i > n\}$ is a theorem.

We can come up with a more appealing rule, however. Let's distinguish the effect of S alone from the effect of $S; i=i+1$. Then our for loop as a while loop looks like this:

```

{I}
i=1
{I∧1≤i≤n+1}
while (i≤n)
  {I∧1≤i≤n}
  S;
  {I∧1≤i≤n}
  i = i+1
  {I∧1≤i≤n+1}
{I∧i==n+1}

```

because $\{I \wedge 1 \leq i \leq n+1\}$ is an invariant of the while loop. So our new rule for the for statement is

FOR': if $\{I \wedge 1 \leq i \leq n\} S \{I \wedge 1 \leq i \leq n\}$ is a theorem, $\{I\}_{i=1}\{I\}$, and $\{I \wedge 1 \leq i \leq n\} i = i+1 \{I\}$ are theorems then $\{I\}$ for $[i=1 \text{ to } n] S \{I \wedge i == n+1\}$ is a theorem.

See how I've replaced the side condition " i is not free in I " with two additional hypotheses concerning the effect of assignment to i . The side condition implies the two hypotheses but not the other way around so this rule would work in situations where the former one would not.

Also, see how S is not constrained to avoid assignment to i . As long as it only assigns values in the range $1..n$ proofs will still go through and the conclusion is correct.

2.30

In this problem we're asked to find the least values of i, j, k such that $f(i) = g(j) = h(k)$ given that each function is a strictly increasing function of non-negative integers, and that there exists such values. The problem is to show that we have computed the smallest such i, j, k . In order to do this we'll define a *logical variable* min to be the required value of f (and g and h). We're told it exists as part of the problem; we don't know what its value is but we know enough of its properties to make use of it in the proof.

The problem stated in the book requires that we perform the tests concurrently.

```

(2)  i=0; j=0; k=0;
(3)  {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
(1)  while f(i)≠g(j) ∨ g(j)≠h(k)
      co
(6)      {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
(5)      if f(i)<g(j) ∨ f(i)<h(k) then
(7)          {f(i)<min ∧ g(j)≤min ∧ h(k)≤min}
(5)          i=i+1
          {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
      //
(6)      {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
(5)      if g(j)<f(i) ∨ g(j)<h(k) then
(7)          {f(i)≤min ∧ g(j)<min ∧ h(k)≤min}
(5)          j=j+1
          {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
      //
(6)      {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
(5)      if h(k)<f(i) ∨ h(k)<g(j) then
(7)          {f(i)≤min ∧ g(j)≤min ∧ h(k)<min}
(5)          k=k+1
          {I: f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
      oc
(3)  {f(i)≤min ∧ g(j)≤min ∧ h(k)≤min}
(4)  f(i)==g(j) ∧ g(j)==h(k) ∧ I

```

You should check that each of the required sequential proofs is present in this proof outline.

Non-interference:

The only assignment actions are in the lines labelled (5). The invariant is a critical assertion; let's exam in the case for non-interference of I by the assignment $i=i+1$ in the first branch. The theorem we need is

$$\{f(i) \leq \text{min} \wedge g(j) \leq \text{min} \wedge h(k) \leq \text{min} \wedge f(i) < \text{min} \wedge g(j) \leq \text{min} \wedge h(k) \leq \text{min}\} i=i+1 \{I\}$$

But this is just the same theorem as already surrounds $i=i+1$ which we already established. The proofs of non-interference with the critical assertions on lines labelled (7) are similar.

Instead of a co for each iteration of a loop, we could write a co containing 3 loops:

```
i=0; j=0; k=0;
co
  while f(i)≠g(j)∨g(j)≠h(k)
    if f(i)<g(j) ∨ f(i)<h(k) then
      i=i+1
  //
  while f(i)≠g(j)∨g(j)≠h(k)
    ...
  //
  while f(i)≠g(j)∨g(j)≠h(k)
    ...
oc
```

The proof is almost the same, but the loops do not necessarily make progress toward a solution on each iteration. This program can waste a lot of computer cycles.