Concurrent Programming Lecture 4

5th September 2003

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≤p≤c+1 ∧ ((p==c+1)⇒(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 ∧ p<=n} -- producer invariant
   while (p<n)
      {PI ∧ p<n}
      <await p==c>
(7)   {PI ∧ p<n ∧ p==c}
      buf = a[p]
(1)   {PI ∧ p<n ∧ p==c ∧ buf==a[p]}
(2)   p = p+1
(3)   {PI}
   {PI ∧ p==n}

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

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

```
(1) {PI ∧ p<n ∧ p==c ∧ buf==a[p]}
```

```
    {c≤p+1≤c+1 ∧ p<n ∧ p==c ∧ buf==a[p]}
    {c≤p+1≤c+1 ∧ ((p==c)⇒(buf==a[p]))∧ p<n}
(1′){c≤p+1≤c+1 ∧ ((p+1==c+1)⇒(buf==a[p])) ∧ p+1<=n}}
(2) p=p+1
(3) {PI: c≤p≤c+1 ∧ ((p==c+1)⇒(buf==a[p-1])) ∧ 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 GI∧L, 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) ∧ C} c=c+1 {C} is a theorem (local non-interference). Observe that {pre(c=c+1) ∧ C} is *false* because it contains (p==c)∧(p==c+1). So by the consequence rule, {pre(c=c+1) ∧ 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 proceses 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≠c∧p≠c+1, but the global invariant which is a precondition of both await statements says that at all times p=c∨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 infintely many times – it has nothing to do with frequency in the sense that EEs understand it).

Exercises for next time

Problem 2.12, 2.22, 2.30

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.