6.826 Principles of Computer Systems	**Fall Semester, 1993**
Lecture 9: October 13, 1993	
Lecturer: Bill Weihl	*Scribe: Greg McLaren*

1 Administrivia

Handout	Title
26	Solutions to Problem Set #2
27	Problem Set #5

2 Lecture Overview

The lecture will cover concurrent modules and systems:

- Proving the Correctness of Concurrent Modules
- Examples—Dominant Threads
- Partial Correctness—Safety vs. Liveness Properties

3 Abstraction Functions on Concurrent Modules

3.1 Changes to the Abstraction Function

Recall that in a sequential implementation, to prove correctness we rely on showing the existence of an abstraction function, $AF_{atomic}(s)$, which maps initial states of the implementation to initial states of the specification, and each possible transition in the implementation to a legal transition of the specification. Invariants were often invoked on the state of the implementation, so that the abstraction function only had to be considered on states in the implementation satisfying the invariant.

When dealing with concurrent modules, however, we find that our old notions are no longer valid. There is no longer a one-to-one map of each state of the implementation to a single state in the specification. To see this, let `s` and `s'` be initial and final states of the implementation, respectively, and let `T` be the implied transition. Then in general, the image of the transition under the concurrent abstraction function, $AF_{con}(s)$, is a sequence α of transitions in the specification:

```
              alpha
  AF(s) -_-_-_-_-_-> AF(s')
  /|\                /|\
   |                  |
   |        T         |
   s       --->       s'        .
```

So in SPEC notation, we might write:

```
AF: Tr(Implementation) -> SEQ[Tr(Spec)].
```

Although true, it is usually the case that each transition in the implementation maps to a single or empty transition in the specification. A transition in the implementation is either an external transition, corresponding to a procedure invocation or return, or an internal transition, which corresponds to the execution of some atomic block in the implementation. External transitions always map to a single transition in the specification, but internal transitions can map to arbitrary length sequences of transitions of the specification (including the empty sequence).

3.2 Concurrent Implementation State

Unlike sequential abstraction functions, whose domain included only the global variables of the implementation, concurrent abstraction functions must be defined in terms of global variables of the implementation **and** the implementation's program counter(s)[1] **and** any local variables of the procedure containing an implementation program counter. The abstraction function takes a state of this type to a state consisting of our conventional (sequential) specification state (global spec variables), plus a specification program counter for each program counter active in the spec.

3.3 Concurrent Implementation Invariants

Invariants in concurrent modules differ from those in sequential modules in that they are often specific to the program counter of the various threads in the implementation. In other words, a different invariant may hold at each possible point of a program counter in the implementation. By examining the code in the implementation, statements that act atomically can be grouped together to reduce the number of labels in the implementation, and hence the number of points at which invariants must be considered.[2]

[1] If any `THREAD` definitions are given in the implementation, there'll be an active thread in one of the procedures plus one for each possible argument value of each `THREAD` definition.

[2] Remember that `S1 ; S2` is atomic iff no command in another thread may execute at the "`;`" which affects or is affected by the execution of `S1`.

4 Example: FIFO Buffer

How should we go about implementing a specification which is defined using large atomic actions (`<< ... >>`) when our platform only allows small atomic actions? Butler Lampson points out that, in general, proving concurrent modules correct is a very difficult job that can only be done successfully on small programs. However, he observes that in the special case in which mutexes are used, the proof can be carried through with relative ease for even large programs.

Let's look at the implementation of the FIFO buffer in Handout 20, page 9. A FIFO buffer is composed of two operations, `Produce`, which takes an argument and "pushes it" onto the queue by concatenating it onto the end of the sequence data structure, and `Consume`, which takes no argument but "pops" the first element of the queue (sequence), returning that element as its result. Clearly, without the atomicity brackets around the specification procedure bodies, they would not be correct. For example, the assignment in `Produce(t)` actually consists of an evaluation of `b`, and evaluation of `t`, a call to append a sequence element, and finally the assignment of the result to `b`. If at any point in this process the sequence were read or written to by another thread, errors might be generated.

Our implementation solves this problem with mutexes, but how can we formally prove that our solution works? The labeled specification and implementation of `Produce` are

```
PROC Produce(t) =      %Label P1
      << b := b++t>>   %Label P2

          and

PROC Produce (t) =        %Label p1
   Mutex.Acquire();       %Label p2
   ttemp := items++t;     %Label p3
   items := ttemp;        %Label p4
   Condition.Signal();    %Label p5
   Mutex.Release()        %Label p6
```

Well, suppose an external transition occurs in the implementation. Then we're done since

$$Thread_{impl}\neg \in \texttt{Produce} \text{ or } \texttt{Consume}$$

implies the implementation and specification transitions are the same.

If an internal transition occurs, say in `Produce`, then either

$$pc_{impl} \in \{p_1, p_2, p_3\} \text{ or is in } \texttt{Acquire()},$$

which implies $AF_{con}(pc_{impl}) = P1$, or else

$$AF_{con}(pc_{impl}) = P2. \tag{1}$$

What's the component of the abstraction function which that yields our specification state variable, `b`? (Having just seen the component that yields the specification program counter.) It is simply the implementation variable, **items**. This is true because the only transition of the implementation program counter in `Produce` that doesn't have a trivial image in the specification is the $p_3 \rightarrow p_4$ transition, which corresponds exactly to when **items** is updated. (To see this, note if `pc(t)` = p_3, then **ttemp** = **items** ++ `t.t`, while if `pc(t)` = p_4, then **items** has just been atomically updated to **ttemp** to reflect the appending of `t.t`.) Thus, the atomicity in the specification of `Produce` is preserved.

Alternatively, we could have defined the specification program counter to make its significant transition as soon as we enter the body of `Produce` protected by the mutex. If we were to do this, then we'd have to define the abstraction function component yielding `b` to be:

$$AF_{con}(pc_{impl}, items, t.t)|_b = items + +t.t$$

To write this AF we observe that we want "`b`" in the specification to change only on a transition of the program counter in the implementation which causes the specification program counter to jump from P1 to P2. (The $p_1 \rightarrow p_2$ implementation program counter transition is the significant one in this case.) So we define `b` so that it is constant throughout the mutex-protected body of `Produce()`, and changes only as we undergo the transition from p_0 to p_1. To make `b` constant inside the body, we note where the global variable of the implementation is changed—only when the actual assignment to **items** occurs, after the right side of the expression has been computed—and partition our definition of `b` into two cases on either side of this assignment.

The first solution is clearly simpler.

The formal proof is given in Handout 23, pages 2-5.

5 Dominant Threads

A thread `t` is **dominant** if `pc(t)` is in `Produce(t)` after the body of `Acquire()` and before that of `Release()`, or `pc(t)` is in `Consume()` between the bodies of `Acquire()` and `Release()` but not in `Wait()`. In general, a thread `t` is dominant if and only if Mutex.M = `t`, and

$\forall$ t,t' (t and t' dominant $\rightarrow$ t = t').

6 Example: Mutexes

Mutexes are specified in Handout 20, page 2 and implemented on pages 6-7, and a formal proof of the implementation's correctness is given in Handout 24, pages 5-9.

6.1 Lack of an Abstraction Function

When we apply the formal proof techniques used in the first example to the case of mutexes, we immediately find that no abstraction function can be written from the implementation state to the specification state because, while the specification keeps track of which thread holds the mutex, the implementation does not. To overcome this difficulty, we apply the technique of history variables introduced previously in the context of sequential implementation correctness. Once this new variable has been introduced, we define the abstraction function on the active implementation program counters, the global implementation variables, and any local variables, just as above when the FIFO buffer was dealt with.

6.2 Program Counter Specific Invariant

To prove the correctness of Mutexes, we again do a case analysis of the transitions. The possible implementation program counter positions include invocations and returns from external procedures, transitions inside the body of `Acquire`, and transitions inside the body of `Release`. The only tricky step significantly different from any used in the proof of the FIFO implementation is the introduction of several invariants which are program-counter specific, listed at the top of page 8 of Handout 24. The invariants are proved by induction, and in the case of a program-counter-specific invariant, part of the induction hypothesis requires that the state before the transition be a particular labeled implementation state. All that must be shown is that after a transition *starting at that state* the invariant holds.

7 Safety and Liveness

Correctness of an implementation can be split into safety, and liveness. A safety property, S, specifies the allowable transitions of the system. The useful information it provides is what a system **cannot** do. A typical S might say that responses to procedure invocations be mathematically and logically correct, and that the system be in an allowed abstract state. We can make assertions about state variables in S, and thus in general can require that deadlock not occur (since deadlock is a function of the current state) but cannot require termination unless we give a finite bound on the running time in S[3], since termination after arbitrarily many steps is not a computable function—the Halting Problem is intractable. (To prevent deadlock, we would use an S that disallows the system from entering deadlock-prone states.) Although termination after arbitrary many steps cannot be a safety property, we can have S assert that the worst-case behavior of a procedure involves returning in less than N_0 steps, for some fixed N_0. In general, if S is satisfied, we've achieved **partial correctness**.

To handle termination, we introduce *liveness properties*; they specify what a system must do, and are often used to assert that something good must eventually happen. Aside from termination, "livelock" is also a liveness property. Livelock is the condition which obtains when several contentious threads are running, but none is making any real progress at a high level. (E.g. two Ethernet nodes might be livelocked if they kept attempting to send a packet but always encountered a collision.) Proving liveness can be much more difficult than proving safety properties, because infinite sequences

[3] Using a counter for a state variable could permit this.

of implementation transitions must be considered in showing liveness, and a single invalid transition, occurring with an infinitesimally small probability, is enough to ruin the proof.

Due to the difficulty of proofs involving liveness and of stating such properties in SPEC, we'll largely concentrate of safety issues in the rest of the course, ignoring liveness concerns.