Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 24 October 6, 1993

Proving Concurrent Modules Correct

This handout contains notes on proving the correctness of concurrent program modules.

1 General Technique

We use the same basic technique as for sequential modules. However, it must be elaborated in several ways. The biggest change is that the transitions of the implementation no longer correspond one-to-one with the transitions of the specification. (We are assuming here that the only interaction between the module and the rest of the program is through calls to the external routines provided by the module.) Thus, we need to show for each transition of the implementation that there exists some reasonable way of explaining it in terms of transitions of the specification.

In general, a transition of the implementation is explained by a sequence of transitions of the specification. The key constraint is that an external transition must be explained by a sequence that contains exactly one instance of the same external transition and no other external transitions; similarly, an internal transition must be explained by a sequence that contains no external transitions. We say that such sequences are *externally the same* as the transitions they correspond to.

Given an abstraction function AF on the state and a rep invariant on the state of the implementation, we must show:

- If r is an initial state of the implementation, then AF(r) is an initial state of the specification.
- For all transitions *tr* and states r and r' of the implementation such that r satisfies the rep invariant and *tr* can take r to r', there must exist a sequence of transitions of the specification that is externally the same as *tr* and that can take AF(r) to AF(r').

These conditions are essentially the same as the ones we used for sequential modules, except that for sequential modules we fixed the correspondence between transitions of the implementation and transitions of the specification to be the identity.

The remainder of this handout contains example proofs of correctness for two examples: the FIFO buffer from Handout 20, and the spinlock implementation of mutexes from Handout 20.

2 FIFO Buffer

First, we give the abstraction function; then we use it to show that the implementation simulates the specification.

2.1 An Example Abstraction Function

The abstraction function on the state must explain how to interpret a state of the implementation as a state of the specification. To prove a concurrent program correct, we need to consider the entire state of a module, including the program counters and local variables of threads. For sequential programs, we managed to avoid this by treating each external operation as a single atomic action.

To describe the abstraction function, we thus need to explain how to construct a state of the specification from a state of the implementation. So what is a state of the Buffer module in Handout 20? It consists of:

- A sequence of items (the buffer itself);
- for each thread that is active in the module, a program counter; and
- for each thread that is active in the module, values for local variables.

A state of the implementation is similar, except that it includes the states of the Mutex and Condition modules.

To define the mapping, we need to enumerate the possible program counters. For the specification, they are:

- P_1 — before the body of Produce
- P_2 — after the body of Produce
- C_1 — before the body of Consume
- C_2 — after the body of Consume

For the implementation, they are:

- For a thread in Produce:

  p_1 – before Mutex.Acquire
  in Mutex.Acquire – either before or after action
  p_2 – before `temp := items++t`
  p_3 – before `items := temp`
  p_4 – before Condition.Signal
  in Condition.Signal – either before or after action
  p_5 – before Mutex.Release
  in Mutex.Release –either before or after action
  p_6 – after Mutex.Release

- For a thread in Consume:

  c_1 – before Mutex.Acquire
  in Mutex.Acquire –either before or after action
  c_2 – before the test `items.size=0`
  c_3 – before Condition.Wait
  in Condition.Wait – at beginning, in middle, or at end
  c_4 – before `VAR t := items.head`
  c_5 – before `temp := items.tail`
  c_6 – before `items := temp`
  c_7 – before Mutex.Release
  in Mutex.Release – either before or after action
  c_8 – before `RET t`
  c_9 – after `RET t`

Notice that we have broken the assignment statements into their constituent atomic actions, introducing a temporary variable to hold the result of evaluating the right-hand-side. Also, the pc's in the Mutex and Condition operations are taken from the specifications of those modules.

Now we can define the mapping on program counters:

- If a thread t is not in Produce or Consume in the implementation, then it is not in either procedure in the specification.
- If a thread t is in Produce in the implementation, then:
  - If pc(t) is in $\{p_1, p_2, p_3\}$ or is in Mutex.Acquire, then in the spec pc(t) = P_1.
  - If pc(t) is in $\{p_4, p_5, p_6\}$ or is in Mutex.Release or Condition.Signal then in the specification pc(t) = P_2.
- If a thread t is in Consume in the implementation, then:
  - If pc(t) is in $\{c_1, ..., c_6\}$ or is in Mutex.Acquire or Condition.Wait then in the specification pc(t) = C_1.
  - If pc(t) is in $\{c_7, c_8, c_9\}$ or is in Mutex.Release then in the specification pc(t) = C_2.

The general strategy here is to pick, for each atomic transition in the specification, some atomic transition in the implementation at which the transition in the specification appears to occur. Here, I have chosen the modification of `items` in the implementation as the point at which the body of the corresponding operation in the specification appears to occur. Thus, program counters before that point in the implementation map to program counters before the body in the specification, and similarly for program counters after that point in the implementation.

This choice of the abstraction function for program counters determines how each transition of the implementation is explained in terms of transitions of the specification as follows:

- If *tr* is an external transition, *tr* is explained by the singleton sequence containing just *tr*.
- If *tr* takes a thread from a pc of p_3 to a pc of p_4, *tr* is explained by the singleton sequence containing just the body of Produce.
- If *tr* takes a thread from a pc of c_6 to a pc of c_7, *tr* is explained by the singleton sequence containing just the body of Consume.
- For all other transitions *tr*, AF(*tr*) is the empty sequence.

This example illustrates a typical situation: we usually find that a transition in the implementation is explained by a sequence of either zero or one transitions in the specification. Transitions that have no effect on the abstract state are explained by the empty sequence, while transitions that change the abstract state are explained by a single transition in the specification. The proof technique used here works fine if a transition is explained by a sequence with more than one transition in it, but this doesn't show up in most examples.

In addition to defining the abstract program counters for threads that are active in the module, we also need to define the values of their local variables. For this example, the only local variable is the item `t`. For threads active in either Produce or Consume, the abstraction function defines the value of `t` in a state of the specification to be the value of the identically named variable in the corresponding operation of the implementation.

Finally, we need to describe how to construct the state of the buffer from the state of the implementation. Given the choices above, this is simple: the abstraction function defines the value of `b` in a state of the specification to be the value of `items` in the implementation.

2.2 Proof Sketch

To prove the implementation correct, we need to prove some invariants on the state. Here are some obvious ones; the others we need will become clear as we work through the rest of the proof.

First, define a thread t to be *dominant* if pc(t) is in Produce and pc(t) is in $\{p_2, p_3, p_4, p_5\}$ or is at the end of Mutex.Acquire, in Condition.Signal, or at the beginning of Mutex.Release, or if pc(t) is in Consume and pc(t) is in $\{c_2, c_3, c_4, c_5, c_6, c_7\}$ or is at the end of Mutex.Acquire, at the beginning or end of Condition.Wait (but not in the middle), or at the beginning of Mutex.Release.

Now, I claim that the following property is invariant: a thread t is dominant if and only if `Mutex.m` = t. This simply says that t holds the mutex if and only if its pc is at an appropriate point.

Given this property, we can easily show that operations are mutually exclusive: for all threads t, t' such that t $\neq$ t', if t is dominant then t' is not dominant. In other words, at most one thread can be in the middle of one of the operations in the implementation at any time.

Now let's consider what needs to be shown to prove the implementation correct. First, we need to show that the claimed invariants actually are invariants. We do this using the standard inductive proof technique: show that each initial state of the implementation satisfies the invariants, and then show that each atomic action in the implementation preserves the invariants. This is left as an exercise.

Next, we need to show that the abstraction function defines a "simulation" of the specification by the implementation. Again, this is an inductive proof. The first step is to show that an initial state of the implementation is mapped by the abstraction function to an initial state of the specification. This should be straightforward, and is left as an exercise. The second step is to show that the effects of each transition are preserved by the abstraction function. Let's consider a couple of examples.

- Consider a transition *tr* from r to r' in which an invocation of an operation occurs for thread t. Then in state r, t was not active in the module, and in r', its pc is at the beginning of the operation. This transition is explained by the identical transition in the specification, which has the effect of moving the pc of this thread to the beginning of the operation. So AF(r) is taken to AF(r') by the transition.

- Consider a transition in which a thread moves from pc=p_3 to pc=p_4, setting `items` to the value stored in temp. The corresponding abstract transition sets `b` to `b++t`. To show that this transition does the right thing, we need an additional invariant:

  – if pc(t) = p_3, then temp = `items++t`.

  To prove this, we use the fact that if pc(t)=p_3, then no other thread is dominant, so no other transition can change `items`. We also have to show that any transition that puts t's pc at this point establishes the consequent of the implication — but there is only one transition that does this (the one that assigns to temp), and it clearly establishes the desired property.

The transition in Consume that assigns to `items` relies on a similar invariant. The rest of the transitions involve straightforward case analyses. For the external transitions, it is clear that they correspond directly. For the other internal transitions, we must show that they have no apparent abstract effect; i.e., if they take r to r', then AF(r) = AF(r'). This is left as an exercise.

3 Mutex Implementation

Now we consider the spinlock implementation of mutexes given in Handout 20. Again, we start by defining the abstraction function, and then sketch the correctness proof.

3.1 Abstraction Function

As in the case of the FIFO buffer, we need to be precise about what constitutes a state of the implementation and what constitutes a state of the specification. A state of the specification consists of:

- A value for `m` (either a thread or `nil`); and

- for each thread that is active in the module, a program counter.

There are no local variables for threads in the specification.

A state of the implementation is similar; it consists of:

- A value for `m` (either `available` or `held`);
- for each thread that is active in the module, a program counter; and
- for each thread that is active in Acquire, a value for the local variable `b`.

Now we have a problem: there is no way to define an abstraction function from an implementation state to a specification state. The problem here is that the implementation does not record which thread holds the mutex, yet the specification keeps track of this information. To solve this problem, we have to introduce a history variable. We do this as follows:

- We augment the state of the implementation with an additional variable, `ms`, which has type `(Thread + Null)`; the initial state of `ms` is `nil`.
- We define the effect of each atomic action in the implementation on the history variable; written in Spec, this results in the following modified code:

```
PROC Acquire () =
   VAR b: M := held |
   DO b=held => << b:=m; m:=held >> OD;
   << ms := SELF >>

PROC Release () = << m := available; ms := nil >>
```

  You can easily check that these additions to the code satisfy the constraints required for adding history variables.

Now we can proceed to define the abstraction function. First, we enumerate the program counters. For the specification, they are:

- A_1 — before the body of Acquire
- A_2 — after the body of Acquire
- R_1 — before the body of Release
- R_2 — after the body of Release

For the implementation, they are:

- For a thread in Acquire:

  a_1 – before the assignment to `b`
  a_2 – after the assignment to `b` and before the `DO` loop
  a_3 – before the test of the guard `b = held`
  a_4 – before the atomic action in the body of the `DO` loop
  a_5 – after the `DO` loop and before the assignment to `ms`
  a_6 – after the assignment to `ms`

- For a thread in Release:

  r_1 – before the body
  r_2 – after the body

The transitions in Acquire may be a little confusing: there's a transition from a_4 to a_3, as well as transitions from a_3 to both a_4 and a_5.

Now we can define the mappings on program counters:

- If a thread is not in Acquire or Release in the implementation, then it is not in either in the specification.
- $\{a_1, a_2, a_3, a_4, a_5\}$ maps to A_1, a_6 maps to A_2
- r_1 maps to R_1, r_2 maps to R_2

The part of the abstraction function dealing with the global variables of the module simply defines `m` in the specification to have the value of `ms` in the implementation. As in Handout 12, we just throw away all but the specification part of the state.

Since there are no local variables in the specification, the mapping on program counters and the mapping on the global variables are enough to define how to construct a state of the specification from a state of the implementation.

Once again, the abstraction function on program counters determines how transitions in the implementation are explained by sequences of transitions in the specification:

- If tr is an external transition, tr is explained by the singleton sequence containing just tr.
- If tr takes a thread from a_5 to a_6, tr is explained by the singleton sequence containing just the transition from A_1 to A_2.
- If tr takes a thread from r_1 to r_2, tr is explained by the singleton sequence containing just the transition from R_1 to R_2.
- All other transitions are explained by the empty sequence.

3.2 Proof Sketch

As in the previous example, we will need some invariants to do the proof. Rather than trying to write them down first, we will see what we need as we do the proof.

First, we show that initial states of the implementation map to initial states of the specification. This is easy; all the thread states correspond, and the initial state of `ms` in the implementation is `nil`.

Next, we show that transitions in the implementation and the specification correspond.

All transitions from outside the module to just before a routine's body are straightforward, as are transitions from the end a routine's body to outside the module (i.e., when a routine returns).

The transition in the body of Release is also straightforward. The hard cases are in the body of Acquire.

Consider all the transitions in Acquire before the one from a_5 to a_6. These are all explained by the null transition, so they should leave the abstract state unchanged. And they do, because none of them change `ms`.

The transition from a_5 to a_6 is explained by the transition from A_1 to A_2. There are two cases: when `ms = nil`, and when `ms # nil`. In the first case, the transition from A_1 to A_2 is enabled and, when taken, changes the state so that `m = SELF`. This is just what the transition from a_5 to a_6 does.

Now consider the case when `ms # nil`. We claim this case is impossible—that if a thread is at a_5, then `ms = nil`. But even though this invariant is what we want, it's too weak to prove itself inductively; for that, we need the following, stronger invariant:

> (1) If `m = available` then `ms = nil`; and (2) If a thread is at a_5, or at a_3 with `b = available`, then `ms = nil`, `m = held`, there are no other threads at a_5, and for all other threads at a_3, `b = held`.

Given this invariant, we are done: we have shown the appropriate correspondence for all the transitions in the implementation. So we must prove the invariant.

We prove the invariant by induction. It's vacuously true in the initial state, since no thread could be at a_3 or a_5 in the initial state. Now, for each transition, we assume that the invariant is true before the transition and prove that it still holds afterwards.

The external transitions preserve the invariant, since they change nothing relevant to it.

The transition in Release preserves the first part of the invariant because afterwards both `m = available` and `ms = nil`. To prove that the transition in Release preserves the second part of the invariant, we assume the precondition for Release in the specification is true (otherwise, anything is allowed). If the precondition is true, `ms # nil`, and since we're assuming the invariant is true before the transition, this implies that no thread is at a_3 with `b = available` or at a_5. After the transition to r_2 it's still the case that no thread is at a_3 with `b = available` or at a_5, so the invariant is still true.

Now we consider the transitions in Acquire. The transition from a_1 to a_2 obviously preserves the invariant. The transition from a_2 to a_3 puts a thread at a_3, but `b = held` in this case so the invariant is true after the transition. The transition from a_3 to a_4 also clearly preserves the invariant.

The transition from a_4 to a_3 is the first interesting one. It certainly preserves the first part of the invariant, since it doesn't change `ms` and only changes `m` to `held`. Now we assume the second part of the invariant true before the transition. There are two cases: (1) Before the transition, there is a thread at a_5 or at a_3 with `b = available`. Then we have `m = held`, so after the transition both `b = held` for this thread and `m = held`. This preserves the invariant. (2) Before the transition, there are no threads at a_5 or at a_3 with `b = available`. Then after the transition, there is still no thread at a_5, but there is a new thread at a_3. (Any others must have `b=held`.) Now, if this thread has `b = held`, the second part of the invariant is true vacuously; but if `b = available`, then we have (a) `ms = nil` (since when the thread was at a_4 `m` must have been `available`, hence the first part of the invariant applies); (b) `m = held` (as a direct result of the transition); (c) there are no threads at a_5 (by assumption); and (d) there are no other threads at a_3 with `b = available` (by assumption). So the invariant is still true after the transition.

Now consider the transition from a_3 to a_5. The first part of the invariant is obviously preserved. For this transition to happen, we must have `b = available`, so the antecedent of the second part of the invariant is satisfied. Thus we know that there are no threads at a_5, and no other threads at a_3 with `b = available`. After the transition, there is only one thread at a_5, and no threads at a_3 with `b = available`, so the second part of the invariant is preserved.

Finally, assume a thread t is at a_5, about to transition to $a6$. If the invariant is true here, then t is the only thread at a_5, and all threads at a_3 have `b = held`. So after it makes the transition, the invariant is vacuously true, because there is no other thread at a_5 and the threads at a_3 haven't changed their state.

We have proved the invariant. The invariant implies that if a thread is at a_5, `ms = nil`, which is what we wanted to show.