17. Formal Concurrency

In this handout we take a more formal view of concurrency. Our goal is to be able to prove that a general concurrent program implements a spec.

We begin with a fairly precise account of the non-atomic semantics of Spec. Next we explain the general method for making large atomic actions out of small ones (easy concurrency) and prove its correctness. We continue with several examples of concurrency, both easy and hard: mutexes, condition variables, read-write locks, buffers, and non-atomic clocks. Finally, we give fairly careful proofs of correctness for some of the examples.

Non-atomic semantics of Spec

We have already seen that a Spec module is a way of defining an automaton, or state machine, with steps corresponding to the invocations of external atomic procedures. This view is sufficient if we only have functions and atomic procedures, but when we consider concurrency we need to extend it to include internal steps. To properly model crashes, we introduced the idea of atomic commands that may not be interrupted. We did this informally, however, and since a crash "kills" any active procedure, we did not have to describe the possible behaviors when two or more procedures are invoked and running concurrently. This section describes the concurrent semantics of Spec.

The most general way to describe a concurrent system is as a collection of independent atomic actions that share a collection of variables. If the actions are `A1, ..., An` then the entire system is just the 'or' of all these actions: `A1 [] ... [] An`. In general only some of the actions will be enabled, but for each step the system non-deterministically chooses an action to execute from all the enabled actions. Thus non-determinism encompasses concurrency. In the language of Lamport's TLA, each action is a predicate relating a pre-state and a post-state, and `[]` is logical "or".

Usually, however, we find it convenient to carry over into the concurrent world as much of the framework of sequential computing as possible. To this end, we model the computation as a set of *threads* (also called 'tasks' or 'processes'), each of which executes a sequence of atomic actions; we denote threads by variables `h`, `h'`, etc.; since `t` has other useful meanings. To define its sequence, each thread has a state variable called its 'program counter' `$pc`, and each of its actions has the form `(h.$pc = α) => c`, (this is Spec for **if** $h.\$pc = \alpha$) **then** c, and `[*]` is Spec for **else**) so that `c` can only execute when `h`'s program counter equals `α`. Different actions have different values for `α`, so that at most one action of a thread is enabled at a time. Each action advances the program counter with an assignment of the form `h.$pc := β`, enabling the thread's next action.

It's important to understand there is nothing truly fundamental about threads, that is, about organizing the state transitions into sets such that at most one action is enabled in each set. We do so because we can then carry forward much of what we know about sequential computing into the concurrent world. In fact, we want to achieve our performance goals with as little concurrency as possible, since concurrency is confusing and error-prone.

We now explain how to use this view to understand the non-atomic semantics of Spec.

Non-atomic commands and threads

Unlike an atomic command, a non-atomic command cannot be described simply as a relation between states and outcomes, that is, an atomic step. The simple example of a non-atomic assignment `x := x + 1` executed by two threads should make this clear: the outcome can increase `x` by `1` or `2`, depending on the interleaving of the steps in the two threads. Rather, a non-atomic command corresponds to a *sequence* of atomic steps, which may be interleaved with the sequences of other commands executing concurrently. To describe this interleaving, we use *labels* and *program counters*. We also need to distinguish the various *threads* of concurrent computation.

Intuitively, threads represent sequential processes. Roughly speaking, each point in the program between two atomic commands is assigned a label. Each thread's program counter `$pc` takes a label as its value,[1], indicating where that thread is in the program, that is, what command it is going to execute next. The variables declared by a program are not allowed to have labels as their values, hence there is no `Label` type.

Spec threads are created by top level `THREAD` declarations in a module. They make all possible concurrency explicit at the top level of each module. A thread is syntactically much like a procedure, but instead of being invoked by a client or by another procedure, it is automatically invoked in parallel initially, for every possible value of its arguments.[2] When it executes a `RET` (or reaches the end of its body), a thread simply makes no more steps. However, threads are often written to loop indefinitely.

A thread is named by the name in the declaration and the argument values. Thus, the threads declared by `THREAD Foo(bool) = ...`, for example, would be named `Foo(true)` and `Foo(false)` The names of local variables are qualified by both the name of the thread that is the root of the call stack, and by the name of the procedure invoked.[3] In other words, each procedure in each thread has its own set of local variables. For example, the local variable `p` in the `Sieve` example appears in the state as `Sieve(0).p, Sieve(1).p, ...`. If there were a `PROC Foo` called by `Sieve` with a local variable `baz`, the state might be defined at `Sieve(0).Foo.baz, Sieve(1).Foo.baz, ...`. The pseudo-names `$a`, `$x`, and `$pc` are qualified only by the thread.

Each atomic command defines a step, just as in the sequential semantics. However, now a step is enabled by the program counter value. That is, a step can only occur if the program counter of some thread equals the label before the command, and the step sets the program counter of that thread to the label after the command. If the command at the label in the program counter fails (can't make any steps), the thread is "stuck" and does not make any steps. For example, this might happen because it has a guard that tests for a buffer to be non-empty, and the buffer is empty in the current state. However, it may become unstuck later, because of the steps of some other threads; for example, another

[1] The variables declared by a program are not allowed to have labels as their values, hence there is no `Label` type.

[2] This differs from the threads in Java, in Modula 3, or in many C implementations. These languages start a computation with one thread and allow it to create and destroy threads dynamically using `fork` and `join` operations. The reason for Spec's scheme is that it makes all the concurrency explicit at a glance.

[3] This works for non-recursive procedures. To accommodate recursive procedures, the state must involve something equivalent to a stack. Probably the simplest solution is to augment the state space by tacking on the nesting depth of the procedure to all the names and program counter values defined above. For example, `h + ".P.v"` becomes `h + ".P.v" + d.enc`, for every positive integer `d`. An invocation transition at depth `d` goes to depth `d+1`. You can see that the interaction between threads and control flow can get pretty messy.

thread might put something into the buffer. Thus, a command failing does not necessarily (or even usually) mean that the thread fails.

We won't give the non-atomic semantics precisely here, since it involves many fussy details that don't add much insight. Also, it's somewhat arbitrary. You can always get exactly the atomicity you want by adding local variables and semicolons to increase the number of atomic steps (see the examples below), or `<<...>>` brackets to decrease it.

It's important, however, to understand the basic ideas.

- Each atomic command in a thread or procedure defines a step (atomic procedures and functions are taken care of by the atomic semantics).
- The program counters enable steps: a step can only occur if the program counter for some thread equals the label before the command, and the step sets that program counter to the label after the command.

Thus the state of the system is the global state plus the state of all the threads. The state of a thread is its `$pc`, `$a`, and `$x` values, the local variables of the thread, and the local variables of each procedure that has been called by the thread and has not yet returned.

Suppose the label before the command `c` is α and the one after the command is `β`, and the step function defined by the atomic semantics is `(λ s, o | rel)`.[4] Then if `c` is in thread `h`, its step function is

```
     (λ s, o | s(h+".$pc") = α /\ o(h+".$pc) = β /\ rel')
```

Here `rel'` is `rel` with each reference to a local variable `v` changed to `h + ".v"` or `h + ".P.v"`.

If `c` is in procedure `P`, that is, `c` can execute for any thread whose program counter has reached α, its step function is

```
     (λ  s, o | (EXISTS h: Thread |
          s(h+".P.$pc") = α /\ o(h+".P.$pc) = β /\ rel'))
```

Here are some examples of a non-atomic program translated into the non-deterministic form. The first one is very simple:

```
[α1] C1 ; [α2] C2 [α3]              pc= α1 => << C1;                pc:= α2 >>
                                 [] pc= α2 => << C2;                pc:= α3 >>
                                 [] pc= α3 => ...
```

As you can see, `[α] C; [β]` translates to `pc=α => << C; pc:=β >>`. If there's a non-deterministic choice, that shows up as a choice in one of the actions of the translation. For example, because `;` binds more tightly that `[]`,

```
[α1] IF C1 ; [α2] C2 [] C3 FI [α3]     pc= α1 => IF << C1;            pc:= α2 >>
                                              [] << C3;            pc:=α3 >>
                                              FI
                                    [] pc= α2 => << C2;            pc:= α3 >>
                                    [] pc= α3 => ...
```

You might find it helpful to write out the state transition diagram for this program.

[4] This function gives the relation between the initial state `s` and the final outcome (state or exception) `o`.

The second example is a simple real Spec program. The next section explains the detailed rules for where the labels go in Spec; note that an assignment is two atomic actions, one to evaluate the expression on the right and the other to change the variable on the left. The extra local variable `t` is a place to store the value between these two actions.

```
                                        VAR t |
[α1] DO i < n =>                           << pc=α1 /\ i < n => pc:=α2 [*] pc:=α6 >>
    [α2] sum := [α3] sum + x(i);        []<< pc=α2 => t := sum + x(i);     pc:=α3 >>
                                        []<< pc=α3 => sum := t;            pc:=α4 >>
    [α4] i := [α5] i + 1                []<< pc=α4 => t := i + 1;          pc:=α5 >>
                                        []<< pc=α5 => i := 1;              pc:=α1 >>
OD [α6]                                 []<< pc=α6 => ...
```

Labels in Spec

What are the atomic steps in a Spec program? In other words, where do we put the labels? The basic idea is to build in as little atomicity as possible, since you can always put in what you need with `<<...>>`. However, expression evaluation must be atomic, or reasoning about expressions would be a mess. To use Spec to model code in which expression evaluation is not atomic (C code, for example), you must add temporary variables. Thus `x := a + b + c` becomes

```
    VAR t | << t := a >>; << t := t + b >>; << x := t + c >>
```

For a real-life example of this, see `MutexImpl.acq` below.

The simple commands, `SKIP`, `HAVOC`, `RET`, and `RAISE`, are atomic, as is any command in atomicity brackets `<<...>>`.

For an invocation, there is a step to evaluate the argument and set the `$a` variable, and one to send control to the start of the body. The `RET` command's step sets `$a` and leaves control at the end of the body. The next step leaves control after the invocation. So every procedure invocation has at least four steps: evaluate the argument and set `$a`, send control to the body, do the `RET` and set `$a`, and return control from the body. The reason for these fussy details is to ensure that the invocation of an external procedure has start and end steps that do not change any other state. These are the steps that appear in the trace and therefore must be identical in both the spec and the code that implements it.

Minimizing atomicity means that an assignment is broken into separate steps, one to evaluate the right hand side and one to change the left hand variable. This also has the advantage of consistency with the case where the right hand side is a non-atomic procedure invocation. Each step happens atomically, even if the variable is "big". Thus `x := exp` is

```
    VAR t | << t := exp >> ; << x := t >>
```

and `x := p(y)` is

```
    p(y); << x := $a >>
```

Since there are no additional labels for the `c1 [] c2` command, the initial step of the compound command is enabled exactly when the initial step of either of the subcommands is enabled (or if they both are). Thus, the choice is made based only on the first step. After that, the thread may get stuck in one branch (though, of course, some other thread might unstick it later). The same is true for `[*]`, except that the initial step for `c1 [*] c2` can only be the initial step of `c2` if the initial step of `c1` is not enabled. And the same is also true for `VAR`. The value chosen for `id` in `VAR id | c` must allow `c` to make at least one step; after that the thread might get stuck.

`DO` has a label, but `OD` does not introduce any additional labels. The starting and ending program counter value for `c` in `DO c OD` is the label on the `DO`. Thus, the initial step of `c` is enabled when the program counter is the label on the `DO`, and the last step sets the program counter back to that label. When `c` fails, the program counter is set to the label following the `OD`.

To sum up, there's a label on each `:=`, `=>`, '`;`', `EXCEPT`, and `DO` outside of `<<...>>`. There is never any label inside atomicity brackets. It's convenient to write the labels in brackets after these symbols.

There's also a label at the start of a procedure, which we write on the `=` of the declaration, and a label at the end. There is one label for a procedure invocation, after the argument is evaluated; we write it just before the closing ')'. After the invocation is complete, the PC goes to the next label after the invocation, which is the one on the `:=` if the invocation is in an assignment.

Because of this labeling, as we said before, a procedure invocation has
    one step to evaluate the argument expression,
    one to set the program counter to the label immediately before the procedure body,
    one for each step of the procedure body (using the labels of the body),
    one for the `RET` command in the body, which sets the program counter after the body,
    and a final step that sets it to the label immediately following the invocation.

Here is a meaningless sequential example, just to show where the labels go. They are numbered in the order they are reached during execution.

```
PROC P() = [$P_1$] VAR x, y |
    IF x > 5 => [$P_2$] x := [$P_4$] Q(x + 1, 2 [$P_3$]); [$P_5$] y := [$P_6$] 3
    [] << y := 4 >>
    FI; [$P_7$]
    VAR z | DO [$P_8$] << P() >> OD [$P_9$]
```

External actions

In sequential Spec a module has only external actions; each invocation of a function or atomic procedure is an external action. In concurrent Spec there are two differences:

1. There are internal actions. These can be actions of an externally invoked `PROC` or actions of a thread declared and executing in the module.

2. There are two external actions in the external invocation of a (non-atomic) procedure: the call, which sends control from after evaluation of the argument to the entry point of the procedure, and the return, which sends control from after the `RET` command in the procedure to just after the invocation in the caller. These external steps do not affect the `$a` variable that communicates the argument and result values. That variable is set by the internal steps that compute the argument and do the `RET` command.

There's another style of defining external interfaces in which every external action is an `APROC`. If you want to get the effect of a non-atomic procedure, you have to break it into two `APROC`'s, one that delivers the arguments and sets the internal state so that internal actions do the work of the procedure body, and a second that gets the result. I/O automata[5] use this style, but we don't.

[5] Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996, Chapter 8.

Examples

Here are two Spec programs that search for prime numbers and collect the result in a set `primes`; both omit the even numbers, initializing `primes` to `{2}`. Both are based on the *sieve of Eratosthenes*, testing each prime less than $n^{1/2}$ to see whether it divides n (of course this is not an efficient way of testing for primality). Since the threads may not be synchronized, we must ensure that all the numbers $\leq n^{1/2}$ have been checked before we check n.

The first example is more like a spec, using an infinite number of threads, one for every odd number.

```
CONST Odds       =  {i: Nat | i // 2 = 1 /\ i > 1 }

VAR  primes      :  SET Nat := {2}
     done        :  SET Nat := {}                 % numbers checked

INVARIANT (ALL n: Nat |    n IN done /\ IsPrime(n) ==> n IN primes
                       /\ n IN primes ==> IsPrime(n))

THREAD Sieve1(n :IN Odds) =

       {i :IN Odds | i <= Sqrt(n)} <= done =>     % Wait for possible factors
           IF   (ALL p :IN primes | p <= Sqrt(n) ==> n mod p # 0) =>
                << primes := primes ∪ {n} >>
           [*]  SKIP
           FI;
           << done = done ∪ {n} >>                % No more steps

FUNC Sqrt(n: Nat) -> Int = RET { i: Nat | i*i <= n }.max
```

The second example, on the other hand, is closer to code, running ten parallel searches. Although there is one thread for every integer, only threads `Sieve(0)`, `Sieve(1)`, …, `Sieve(9)` are "active", because of the initial guard, Differences from `Sieve1` are boxed.

```
CONST nThreads   := 10

VAR  primes      :  SET Int := {2}
     next        := nThreads.seq

THREAD Sieve(i: Int) = next!i =>
     next(i) := 2*i + 3;
     DO VAR n: Int  := next(i) |
          (ALL j :IN next.rng | j >= Sqrt(n)) =>
                IF   (ALL p :IN primes | p <= Sqrt(n) ==> n // p # 0) =>
                     << primes := primes ∪ {n} >>
                [*]  SKIP
                FI;
                next(i) := n + 2*nThreads

     OD
```

Big atomic actions

As we saw earlier, we need atomic actions for practical, easy concurrency. Spec lets you specify any grain of atomicity in the program just by writing << ... >> brackets. (As we have seen, Spec does treat expression evaluation as atomic. Recall that if you are dealing with an environment in which an expression like `x(i) + f(y)` can't be evaluated atomically, you should model this by writing `VAR t1, t2 | t1 := x(i); t2 := f(y); ... t1 + t2 ....)`.) It doesn't tell you where to write the brackets. If the environment in which the program runs doesn't impose any constraints, it's usually best to make the atomic actions as big as possible, because big atomic actions are easier to reason about. But big atomic actions are often too hard or too expensive to code, or the reduction in concurrency hurts performance too much, so that we have to make do with small ones. For example, in a shared-memory multiprocessor typically only the individual instructions are atomic, and we can only write one disk block atomically. So we are faced with the problem of showing that code with small atomic actions satisfies a spec with bigger ones.

The idea

The standard way of doing this is by some kind of 'non-interference'. The idea is based on the following observation. Suppose we have a program with a thread `h` that contains the sequence

```
    A; B                                                    (1)
```

as well as an arbitrary collection of other commands C_i. We denote the program counter value before `A` by α and at the semi-colon by β: `[α] A; [β] B`. So the transition relation for the program is

```
    h.$pc = α => A [] h.$pc = β => B [] C1 [] C2 [] ...
```

where each command has an appropriate guard that enables it only when the program counter for its thread has the right value. We have written the guards for `A` and `B` explicitly.

Suppose `B` denotes an arbitrary atomic command, and `A` denotes an atomic command that commutes with every command in the program (other than `B`) that is enabled when `h` is at the semicolon, that is, when `h.$pc = β`. (We give a precise meaning for 'commutes' below.) In addition, both `A` and `B` have only internal actions. Then it's intuitively clear that the program with (1) simulates a program with the same commands except that instead of (1) it has

```
    << A; B >>                                              (2)
```

Informally this is true because any `C`'s that happen between `A` and `B` have the same effect on the state that they would have if they happened before `A`, since they all commute with `A`. Note that the `C`'s don't have to commute with `B`; commuting with `A` is enough to let us 'push' `C` before `A`. A symmetric argument works if all the `C`'s commute with `B`, even if they don't commute with `A`.

Thus we have achieved the goal of making a bigger atomic command `<< A; B >>` out of two small ones `A` and `B`. We can call the big command `D` and repeat the process on `E; D` to get a still bigger command `<< E; A; B >>`.

How do we ensure that only a command `C` that commutes with `A` can execute while `h.$pc = β`? The simplest way is to ensure that the variables that `A` touches (reads or writes) are disjoint from the variables that `C` writes, and vice versa; then they will surely commute. Two such commands are called 'non-interfering'. There are two easy ways to show that commands are non-interfering. One is that `A` touches only local variables of `h`. Only actions of `h` touch local variables of `h`, and the only action of `h` that is enabled when `h.$pc = β` is `B`. So any sequence of commands that touch only local variables is atomic, and if it is preceded or followed by a single arbitrary atomic command the whole thing is still atomic.[6]

[6] See Lamport and Schneider. Pretending atomicity. Research Report 44, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, May 1989. http://lamport.azurewebsites.net/pubs/pretending.pdf .

The other easy case is a critical section protected by a mutex. Recall that a critical section for `v` is a command with the property that if some thread's `PC` is in the command, then no other thread's `PC` can be in any critical section for `v`. If the only commands that touch `v` are in critical sections for `v`, then we know that only `B` and commands that don't touch `v` can execute while `h.$pc = β`. So if every command in any critical section for `v` only touches `v` (and perhaps local variables), then the program simulates another program in which every critical section is an atomic command. A critical section is usually coded by acquiring a *lock* or *mutex* and holding it for the duration of the section. The property of a lock is that it's not possible to acquire it when it is already held, and this ensures the mutual exclusion property for critical sections.

It's not necessary to have exclusive locks; reader/writer locks are sufficient for non-interference, because read operations all commute with each other. Indeed, any locking scheme will work in which *non-commuting operations hold mutually exclusive locks*; this is the basis of rules for 'lock conflicts'.

Another important case is mutex acquire and release operations. These operations only touch the mutex, so they commute with everything else. What about these operations on the same mutex in different threads? If both can execute, they certainly don't yield the same result in either order; that is, they don't commute. When can both operations execute? We have the following cases (writing the executing thread as an explicit argument of each operation):

`A [β]`	`C`	Possible sequence (`C` is enabled at β)?
`m.acq(h)`	`m.acq(h')`	No: `C` is blocked by `h` holding `m`
`m.acq(h)`	`m.rel(h')`	No: `C` won't be reached because `h'` doesn't hold `m`
`m.rel(h)`	`m.acq(h')`	OK
`m.rel(h)`	`m.rel(h')`	No: one thread doesn't hold `m`, hence won't do `rel`

So `m.acq` commutes with everything that's enabled at β, since neither mutex operation is enabled at β in a program that avoids havoc. But `m.rel(h)` doesn't commute with `m.acq(h')`. The reason is that the `A; C` sequence can happen, but the `C; A` sequence `m.acq(h'); m.rel(h)` cannot, because in this case `h` doesn't hold `m` and therefore can't be doing a `rel`. Hence it's not possible to flip every `C` in front of `m.rel(h)` in order to make `A; B` atomic.

What does this mean? You can acquire more locks and still keep things atomic, but as soon as you release one, you no longer have atomicity.[7]

A third important case of commuting operations, producer-consumer, is like the mutex case. A producer and a consumer thread share no state except a buffer. The operations on the buffer are `put` and `get`, and these operations commute with each other. The interesting case is when the buffer is empty. In this case `get` is blocked until a `put` occurs, just as in the mutex example when `h` holds the lock `m.acq(h')` is blocked until `m.rel(h)` occurs. This is why programming with buffers, or dataflow programming, is so easy.

[7] Actually, this is not quite right. If you hold several locks, don't acquire new ones after you release one, and touch data only when you hold its lock, you have atomicity until you release all the locks. This is called "two-phase locking".

Proofs

How can we make all this precise and *prove* that a program containing (1) implements the same program with (2) replacing (1), using our trusty method of abstraction relations? For easy reference, we repeat (1) and (2).

```
    << A; B >>                                                   (2: S)
    A; [β] B                                                     (1: U)
```

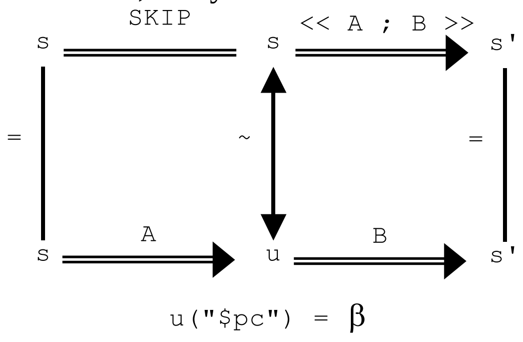
As usual, we call the complete program containing (2) the spec *S* and the complete program containing (1) the code *U*. We need an abstraction relation `AR` between the states of *U* and the states of *S* under which every step of *U* simulates a (possibly empty) trace of *S*. Note that the state spaces of *U* and *S* are the same, except that `h.$pc` can never be β in *S*. We use *s* and *u* for states of *S* and *U*.

First we need a precise definition of "`C` is enabled at β and commutes with `A`". For any atomic command `X`, we write `u X u'` if `X` relates `u` to `u'`. The idea of 'commutes' is that `<<A; C>>`, as a relation between states, is a *subset* of the relation `<<C; A>>`. To make this precise all we need is to plug in the meaning of semicolon: `C` commutes with `A` iff

```
    (ALL u1, u2 |   (EXISTS u  | u1 A u  /\ u  C u2 /\ u("h.$pc") = β)
                ==> (EXISTS u' | u1 C u' /\ u′ A u2) )
```

This says that any result that you could get by doing `A; C` you could also get by doing `C; A`. Note that it's OK for `C; A` to have more steps, since we want to show that `A; C; B` implements `C;`
`<< A; B >>`, not that they are equivalent. This is not just nit-picking; if `C` tries to acquire a lock that `A` holds, there is no step from `A` to `C` in the first case.

It seems reasonable to do the proof by making `A` simulate the empty trace and `B` simulate `<<A; B>>`, since we know more about `A` than about `B`; every other command simulates itself.


$$\begin{tikzcd}
s \arrow[r, equal, "\texttt{SKIP}"] \arrow[d, dash, "="'] & s \arrow[r, Rightarrow, "\texttt{<< A ; B >>}"] \arrow[d, leftrightarrow, "\sim"'] & s' \arrow[d, dash, "="] \\
s \arrow[r, Rightarrow, "\texttt{A}"] & u \arrow[r, Rightarrow, "\texttt{B}"] & s'
\end{tikzcd}$$

`u("$pc") = β`

So we make `AR` the identity everywhere except at β, where it relates any state `u` that can be reached from `s` by `A` to `s`. This expresses the intention that at β we haven't yet done `A` in *S*, but we have done `A` in *U*. (Since `A` may take many states to `u`, this can't just be an abstraction function.) We write `s ~ u` for "`AR` relates `s` to `u`". Precisely, we say that `s ~ u` if

```
       u("h.$pc") ≠ β /\ s = u
    \/ u("h.$pc") = β /\ s A u.
```

Why is this an abstraction relation? It certainly relates an initial state to an initial state, and it certainly works for any step `u -> u'` that stays away from β, that is, in which `u("h.$pc") ≠ β` and `u'("h.$pc") ≠ β`, since the abstract and concrete states are the same. What about steps that do involve β?

- If `h.$pc` changes to β then we must have executed `A`. The picture is

$$\begin{tikzcd}
s \arrow[r, equal, "="] \arrow[d, dash, "="'] & s' \arrow[d, "A"] \\
u \arrow[r, Rightarrow, "A"] & u'
\end{tikzcd}$$

  The abstract trace is empty, so the abstract state doesn't change: `s = s'`. Also, `s = u` because only equal states are related when `h.$pc # β`. But we executed `A`, so `u A u'`, so `s' ~ u'` because of the equalities.

- If `h.$pc` starts at β then the command must be either `B` or some `C` that commutes with `A`. If the command is `B`, then the picture is

$$\begin{tikzcd}
s \arrow[r, Rightarrow, "<<A;\ B>>"] \arrow[d, "A"'] & s' \arrow[d, "="] \\
u \arrow[r, Rightarrow, "B"] & u'
\end{tikzcd}$$

  To show the top relation, we have to show that there exists an `s0` such that `s A s0` and `s0 B s'`, by the meaning of semicolon. But `u` has exactly this property, since `s' = u'`.

- If the command is `C`, then the picture is

$$\begin{tikzcd}
\mathbf{u1} & s \arrow[r, Rightarrow, "C"] \arrow[d, "A"'] & s' \arrow[d, "A"] & \mathbf{u'} \\
 & u \arrow[r, Rightarrow, "C"] & u' & \mathbf{u2}
\end{tikzcd}$$

  But this follows from the definition of 'commutes': we are given `s`, `u`, and `u'` related as shown, and we need `s'` related as shown, which is just what the definition gives us, with `u1 = s`, `u2 = u'`, and `u' = s'`. Less formally, the diagram commutes because `<< C ; A >> = << A ; C >>`.

Examples of concurrency

This section contains several examples of specs and code that illustrate various aspects of concurrency. The specs have large atomic actions that keep them simple. The codes have smaller atomic actions that reflect the realities of the machines on which they run. Some of the examples of code illustrate easy concurrency (using locks): `RWLockImpl` and `BufferImpl`. Others illustrate hard concurrency: `SpinLock` and `ClockImpl`.

Incrementing a register

The first example involves incrementing a register (or a single memory location) that has `Read` and `Write` operations. Here is the unsurprising spec of the register, which makes both operations atomic:

```
MODULE Register EXPORT Read, Write =
VAR  x           :  Int := 0
APROC Read() -> Int = << RET x  >>
APROC Write(i: Int) = << x := i >>
END Register
```

To increment the register, we might be tempted to use the following (bad) procedure:

```
PROC Increment() = VAR t: Int |
     t := Register.Read(); t := t + 1; Register.Write(t)
```

Suppose that, starting from the initial state where $x = 0$, n threads execute `Increment` in parallel. Then, depending on the interleaving of the low-level steps, the final value of the register could be anything from 1 to n. This is unlikely to be what was intended. Certainly this code doesn't implement the obvious spec

```
PROC Increment() = << x := x + 1 >>
```

Exercise: Suppose that we weaken our atomicity assumptions to say that the value of a register is represented as a sequence of bits, and that the only atomic operations are reading and writing individual bits. Now what are the possible final states if n threads execute `Increment` in parallel?

Alternatively, consider a new module `RWInc` that explicitly supports `Increment` operations in addition to `Read` and `Write`. This might add the following (exported) procedure to the `Register` module:

```
PROC Increment() = x := x+1
```

Or, more explicitly:

```
PROC Increment() =  VAR t: Int | << t := x >>; << x := t+1 >>
```

Because of the fine grain of atomicity, it is still true that if n threads execute `Increment` in parallel then, depending on the interleaving of the low-level steps, the final value of the register could be anything from 1 to n. Putting the procedure inside the `Register` module doesn't help. Of course, making `Increment` an `APROC` would certainly do the trick.

Mutexes

Here is a spec of a simple `Mutex` module, which can be used to ensure mutually exclusive execution of critical sections. The state of a mutex is the thread that holds the mutex, or `nil` if it's free.

```
CLASS Mutex EXPORT acq, rel =
VAR  m           :  (Thread + Null) := nil
% Each mutex is either nil or the thread holding the mutex.
% The variable SELF is defined to be the thread currently making a step.
APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>
END Mutex
```

If a thread invokes `acq` when `m ≠ nil`, then the body fails. This means that there's no possible step for that thread, and the thread is blocked, waiting at this point until the guard becomes true. If many threads are blocked at this point, then when `m` is set to `nil`, one is scheduled first and it sets `m` to itself atomically; the other threads are still blocked.

The spec says that if a thread that doesn't hold `m` does `m.rel`, the result is `HAVOC`. As usual, this means that the code is free to do anything when this happens. As we shall see in the `SpinLock` code below, one possible 'anything' is to free the mutex anyway.

Here is a simple use of a mutex `m` to make the `Increment` procedure atomic:

```
PROC Increment() = VAR t: Int |
     m.acq; t := Register.Read(); t := t + 1; Register.Write(t); m.rel
```

This keeps concurrent calls of `Increment` from interfering with each other. If there are other write accesses to the register, they must also use the mutex to avoid interfering with threads executing `Increment`.

Spin locks

A simple way to code a mutex is to use a *spin lock*. The name is derived from the behavior of a thread waiting to acquire the lock—it "spins", repeatedly attempting to acquire the lock until it is finally successful.

Here is incorrect code:

```
CLASS BadSpinLock EXPORT acq, rel =
TYPE FH          =  ENUM[free, held]
VAR  fh          := free
PROC acq() =
     DO << fh = held => SKIP >> OD;                    % wait for fh = free
     << fh := held >>                                  % and acquire it
PROC rel() = << fh := free >>
END BadSpinLock
```

This is wrong because two concurrent invocations of `acq` could both find `fh = free` and subsequently both set `fh := held` and return.

Here is correct code. It uses a more complex atomic command in the `acq` procedure (highlighted in bold). This command corresponds to the atomic "test-and-set" instruction provided by many real machines to code locks. It records the initial value of the lock, and then sets it to `held`. Then it tests the initial value; if it was `free`, then this thread was successful in atomically changing the state of the lock from `free` to `held`. Otherwise some other thread must hold the lock, so we "spin", repeatedly trying to acquire it until we succeed. The important difference in `SpinLock` is that the guard now involves only the local variable `t`, instead of the global variable `fh` in `BadSpinLock`. A thread acquires the lock when it is the one that changes it from `free` to `held`, which it checks by testing the value returned by the test-and-set.

```
CLASS SpinLock EXPORT acq, rel =
TYPE FH          =  ENUM[free, held]
VAR  fh          := free
```

```
PROC acq() = VAR t: FH |
     DO << t := fh; fh := held >>; IF t = free => RET [*] SKIP FI OD
PROC rel() = << fh := free >>
END SpinLock
```

Of course code that spins is not practical in general unless each thread has its own processor; it is used, however, in the kernels of most operating systems for computers with several processors.

The `SpinLock` code differs from the `Mutex` spec in another important way. It "forgets" which thread owns the mutex. The following `ForgetfulMutex` module is useful in understanding the `SpinLock` code—in `ForgetfulMutex`, the threads get forgotten, but the atomicity is the same as in `Mutex`.

```
CLASS ForgetfulMutex EXPORT acq, rel =
TYPE FH          =  ENUM[free, held]
VAR  fh          := free
PROC acq() = << fh = free => fh := held >>
PROC rel() = << fh := free >>
END ForgetfulMutex
```

Note that `ForgetfulMutex` releases a mutex regardless of which thread acquired it, and it does a `SKIP` if the mutex is already free. This is one of the behaviors permitted by the `Mutex` spec, which allows anything under these conditions.

It's not hard to show that `SpinLock` implements `ForgetfulMutex` and that `ForgetfulMutex` implements `Mutex`, from which it follows that `SpinLock` implements `Mutex`. We don't give the abstraction functions here because they involve the details of program counters.

Wait-free primitives

It's also possible to implement spin locks with a *wait-free* primitive rather than with test-and-set, although this rather misses the point of wait-free synchronization.

The simplest wait-free primitive is compare-and-swap (CAS), which is illustrated in the following code for `acq`. If the current contents of `fh` is `free` it stores `new` into `fh` (which is an address parameter in real life) and returns true; otherwise it is `SKIP`. Now `acq` has no atomicity brackets.

```
VAR  x : Any
PROC acq() = DO VAR t := CAS(free, held); IF t => RET [*] SKIP FI OD
APROC CAS(old: Any, new: Any)-> Bool =
     << IF x = old => x := new; RET true [*] RET false >>
```

A more general form of compare-and-swap allows you to do an arbitrary computation on the old contents of a variable. It is called load-locked/store-conditional. The idea is that if anyone writes the variable in between the load-locked and the store-conditional, the store fails.

```
VAR  lock : Bool := false                      % a global variable; could be per variable
APROC LL() -> Any = << lock := true; RET x >>
APROC SC(new: Any) -> BOOL = << IF lock => x := new; RET true [*] RET false >>
```

Now we can write `acq`:

```
PROC acq() = VAR fh', OK: Bool |
     DO   fh' := LL();
```

```
          IF fh' = free => OK := SC(held); IF OK => RET [*] SKIP FI
          [*] SKIP FI
     OD
```

Of course we can program `CAS` using `LL/SC`:

```
PROC CAS(old: Any, new: Any)-> Bool = VAR fh', OK: Bool |
     fh' := LL(); IF fh' = old => OK := SC(new); RET OK [*] RET false FI
```

We can also program operations such as incrementing a variable, either with `CAS`:

```
     VAR OK: Bool := false | DO ~OK => i := x; OK := CAS(i, i+1) OD
```

or with `LL/SC`:

```
     VAR OK: Bool := false | DO ~OK => i := LL(); OK := SC(i+1) OD
```

More generally, you can update an arbitrary data structure with an arbitrary function `f` by replacing `i+1` in the `CAS` implementation with `f(i)`. The way to think about this is that `f` computes a new version of `i`, and you install it if the version hasn't changed since you started. This is a form of optimistic concurrency control. Like optimistic concurrency control in general, the approach runs the danger of doing a lot of work that you then have to discard, or of starving some of the threads because they never get done before other threads sneak in and change the version out from under them. There are clever tricks for minimizing this danger; the basic idea is to queue your `f` for some other thread to execute along with its own.

Read/write locks

Here is a spec of a module that provides locks with two modes, read and write, rather than the single mode of a mutex. Several threads can hold a lock in read mode, but only one thread can hold a lock in write mode, and no thread can hold a lock in read mode if some thread holds it in write mode. In other words, read locks can be shared, but write locks are exclusive; hence the locks are also known as 'shared' and 'exclusive'.

```
CLASS RWLock EXPORT rAcq, rRel, wAcq, wRel =
TYPE ST          =  SET Thread
VAR  r           :  ST := {}
     w           :  ST := {}
APROC rAcq() =                                          % Acquires r if no current write locks
     <<    SELF IN (r ∪ w) => HAVOC [*] w       = {} =>  r := r ∪ {SELF} >>
APROC wAcq() =                                          % Acquires w if no current locks
     <<    SELF IN (r ∪ w) => HAVOC [*] (r ∪ w) = {} =>  w :=     {SELF} >>
APROC rRel() =                                          % Releases r if the thread has it
     << ~ (SELF IN r)       => HAVOC [*]                  r := r - {SELF} >>
APROC wRel() =
     << ~ (SELF IN w)       => HAVOC [*]                  w := {}         >>
END RWLock
```

The following simple code is like `ForgetfulMutex`. It has the same atomicity as `RWLock`, but uses a different data structure to represent possession of the lock. Specifically, it uses a single integer variable `rw` to keep track of the number of readers (positive) or the existence of a writer (`-1`).

```
CLASS ForgetfulRWL EXPORT rAcq, rRel, wAcq, wRel =
VAR  rw          := 0
% >0 gives number of readers, 0 means free, -1 means one writer
```

```
APROC rAcq() = << rw >= 0 => rw := rw + 1 >>
APROC wAcq() = << rw  = 0 => rw := - 1 >>
APROC rRel() = << rw := rw - 1 >>
APROC wRel() = << rw := 0 >>
END ForgetfulRWL
```

Coding read/write lock using a mutex and condition variable

This example shows how to use easy concurrency to make more complex locks and scheduling out of basic mutexes and conditions[8]. We use a single mutex and condition for all the read-write locks here, but we could have separate ones for each read-write lock, or we could partition the locks into groups that share a mutex and condition. The choice depends on the amount of contention for the mutex.

Compare the code with `ForgetfulRWL`; the differences are highlighted with boxes. The `<<...>>` in `ForgetfulRWL` have become `m.acq ... m.rel`; this provides atomicity because shared variables are only touched while the lock is held. The other change is that each guard that could block (in this example, all of them) is replaced by a loop that tests the guard and does `c.wait` if it doesn't hold. The release operations do the corresponding signal or broadcast operations.

```
CLASS RWLockImpl EXPORT rAcq, rRel, wAcq, wRel =   % implements ForgetfulRWL

VAR  rw          :  Int := 0
     m: Mutex    := m.new()
     c: Condition := c.new()

% ABSTRACTION FUNCTION ForgetfulRWL.rw = rw

PROC rAcq(l) = m.acq; DO ~ rw >= 0 => c.wait(m) OD; rw := rw + 1; m.rel
PROC wAcq(l) = m.acq; DO ~ rw  = 0 => c.wait(m) OD; rw := -1;     m.rel

PROC rRel(l) = m.acq; rw := rw - 1; IF rw = 0 => c.signal [*] SKIP FI; m.rel
PROC wRel(l) = m.acq; rw := 0;                   c.broadcast;         m.rel

END RWLockImpl
```

This is the prototypical example for scheduling resources. There are mutexes (just `m` in this case) to protect the scheduling data structures, conditions (just `c` in this case) on which to delay threads that are waiting for a resource, and logic that figures out when it's all right to allocate a resource (the read or write lock in this case) to a thread.

Note that this code may starve a writer: if readers come and go but there's always at least one of them, a waiting writer will never acquire the lock. How could you fix this?

An unbounded FIFO buffer

In this section, we give a spec and code for a simple unbounded buffer that could be used as a communication channel between two threads. This is the prototypical example of a *producer-consumer* relation between threads. Other popular names for `Produce` and `Consume` are `Put` and `Get`.

[8] A condition variable has `wait(m)` and `signal` operations. `c.wait(m)` released `m` and blocks the thread on `c`, and `c.signal` unblocks a thread waiting on `c`, and `c.broadcast` unblocks all the waiting threads.

```
MODULE Buffer[T] EXPORT Produce, Consume =

VAR  b           :  SEQ T := {}

APROC Produce(t) = << b + := {t} >>
APROC Consume() -> T = VAR t | << b # {} => t := b.head; b := b.tail; RET t >>

END Buffer
```

The code is another example of easy concurrency.

```
MODULE BufferImpl[T] EXPORT Produce, Consume =

VAR  b           :  SEQ T := {}
     m           := m.new()
     c           := c.new()

% ABSTRACTION FUNCTION Buffer.b = b

PROC Produce(t) =  m.acq; IF b = {} => c.signal [*] SKIP FI; b + := {t}; m.rel

PROC Consume() -> T = VAR t |
     m.acq; DO b = {} => c.wait(m) OD; t := b.head; b := b.tail; m.rel; RET t

END BufferImpl
```

… The remainder of the section on examples is omitted …

Proving concurrent modules correct

This section explains how to prove the correctness of concurrent program modules. It reviews the simulation method that we have already studied, which works just as well for concurrent as for sequential modules. Then several examples illustrate how the method works in practice. Things are more complicated in the concurrent case because there are many more atomic steps, and because the program counters of the threads are part of the state.

Before using this method in its full generality, you should first apply the theorem on big atomic actions as much as possible, to reduce the number of steps that your proofs need to consider. If you are programming with easy concurrency, that is, if your code uses a standard locking discipline and locks every shared variable before touching it, this will get rid of nearly all the work. If you are doing hard concurrency, there will still be lots of steps, and in doing the proof you will probably find bugs in your program.

The formal method

We use the same simulation technique that we used for sequential modules, as described in handouts 6 and 8 on abstraction functions. In particular, we use the most general version of this method, presented near the end of handout 8. This version does not require the steps of the code to correspond one-for-one with the steps of the spec. Only the external behavior (invocations and responses) must be the same—there can be any number of internal steps. The method proves that every trace (external behavior sequence) produced by the code can also be produced by the spec.

Of course, the utility of this method depends on an assumption that the external behavior of a module is all that is of interest to callers of the module. In other words, we are assuming here, as everywhere in this course, that the only interaction between the module and the rest of the program is through calls to the external routines provided by the module.

We need to show that each step of the code simulates a sequence of steps of the spec. An external step must simulate a sequence that contains exactly one instance of the same external step and no other external steps; it can also contain any number of internal steps. An internal step must simulate a sequence that contains only internal steps.

Here, once again, are the definitions:

Suppose *U* and *S* are modules with same external interface. An abstraction function *F* is a function from *states*(*U*) to *states*(*S*) such that:

> *Start*: If *u* is any initial state of *U* then *F*(*u*) is an initial state of *S*.
>
> *Step*: If *u* and *F*(*u*) are reachable states of *U* and *S* respectively, and (*u*, π, *u*') is a step of *U*, then there is an execution fragment of *S* from *F*(*u*) to *F*(*u*'), having the same trace.

Thus, if π is an invocation or response, the fragment consists of a single π step, with any number of internal steps before and/or after. If π is internal, the fragment consists of any number (possibly 0) of internal steps.

As we saw in handout 8, we may have to add history variables to *U* in order to find an abstraction function to *S* (and perhaps prophecy variables too). The values of history variables are calculated in terms of the actual variables, but they are not allowed to affect the real steps.

An alternative to adding history variables is to define an abstraction relation instead of an abstraction function. An abstraction relation *AR* is a relation between *states*(*U*) and *states*(*S*) such that:

> *Start*: If *u* is any initial state of *U* then there exists an initial state *s* of *S* such that (*u*, *s*) ∈ *AR*.
>
> *Step*: If *u* and *s* are reachable states of *U* and *S* respectively, (*u*, *s*) ∈ *AR*, and (*u*, π, *u*') is a step of *U*, then there is an execution fragment of *S* from *s* to some *s*' having the same trace, and such that (*u*', *s*') ∈ *AR*.

Theorem: If there exists an abstraction function or relation from *U* to *S* then *U* implements *S*; that is, every trace of *U* is a trace of *S*.

Proof: By induction.

The strategy

The formal method suggests the following strategy for doing hard concurrency proofs.

1. Start with a spec, which has an abstract state.

2. Choose a concrete state for the code.

3. Choose an abstraction function, perhaps with history variables, or an abstraction relation.

4. Write code, identifying the critical actions that change the abstract state.

5. While (checking the simulation fails) do

Add an invariant, checking that all actions of the code preserve it, or

Change the abstraction function (step 3), the code (step 4), the invariant (step 5), or more than one, or

Change the spec (step 1).

This approach always works. The first four steps require creativity; step 5 is quite mechanical except when you find an error. It is somewhat laborious, but experience shows that if you are doing hard concurrency and you omit any of these steps, your program will have bugs. Be warned.

Prospectus for proofs

The remainder of this handout contains example proofs of correctness for several of the examples above: the `RWLockImpl` code for a read/write lock, the `ClockImpl` code for a multi-word clock, and the `BufferImpl` code for a FIFO buffer.

The proofs give the abstraction functions and key invariants, but do not check each simulation step.

Read/write locks

We sketch how to prove directly that the module `RWLockImpl` implements `ForgetfulRWL`. This could be done by big atomic actions, since the code uses easy concurrency, but as an easy introduction we discuss how to do it directly. The two modules are based on the same data, the variable `rw`. The difference is that `RWLockImpl` uses a mutex to restrict accesses to `rw`, so that a series of accesses to `rw` can be done atomically. It also uses a condition variable to prevent threads in `acq` from busy-waiting when they don't see the condition they require.

An abstraction function maps `RWLockImpl` to `ForgetfulRWL`. The interesting part of the state of `ForgetfulRWL` is the `rw` variable. We define that by the identity mapping from `RWLockImpl`.

The mapping for steps is mostly determined by the `rw` identity mapping: the steps that assign to `rw` in `RWLockImpl` are the ones that correspond to the procedure bodies in `ForgetfulRWL` Then the checking of the state and step correspondences is pretty routine.

There is one subtlety. It would be bad if a series of `rw` steps done atomically in `ForgetfulRWL` were interleaved in `RWLockImpl`. Of course, we know they aren't, because they are always done by a thread holding the mutex. But how does this fact show up in the proof?

The answer is that we need some invariants for `RWLockImpl`, that do for this specific case what easy concurrency with mutexes does in general. The first, a "dominant thread invariant", says that only a thread whose name is in `m` (a 'dominant thread') can be in certain portions of its code (those guarded by the mutex). The dominant thread invariant is in turn used to prove other invariants called "data protection invariants".

For example, one data protection invariant says that if a thread (in `RWLockImpl`) is in middle of the assignment statement `rw + := 1`, then in fact `rw` ≥ 0 (that is, the test is still true). We need this data protection invariant to show that the corresponding abstract step (the body of `rAcq` in `ForgetfulRWLock`) is enabled.

The proof of `BufferImpl` below illustrates these ideas in more detail.

`ClockImpl` implements `Clock`

Often it's possible to get better performance by avoiding locking. Algorithms that do this are called 'wait-free'. Here we present a wait-free algorithm due to Lamport[9] for reading and incrementing a clock, even if clock values do not fit into a single memory location that can be read and written atomically.

We begin with the spec. It says that a `Read` returns some value that the clock had between the beginning and the end of the `Read`. As we saw in handout 8 on generalized abstraction functions, where this spec is called `LateClock`, it takes a prophecy variable to show that this spec is equivalent to the simpler spec that just reads the clock value. Here it is, with labels.

```
MODULE Clock EXPORT Read =
VAR  t            :  Int := 0                              % the current time
THREAD Tick() = DO << t + := 1 >> OD                       % demon thread advances t
PROC Read() -> Int = VAR t1: Int |
     [R1] << t1 := t >>; [R2] << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >> [R3]
END Clock
```

To show that `ClockImpl` implements this we introduce a history variable `t1Hist` in `Read` that corresponds to `t1` in the spec, recording the time at the beginning of `Read`'s execution. The invariant that is needed is based on the idea that `Read` might complete before the next `Tick`, and therefore the value `Read` would return by reading the rest of the shared variables must be between `t1Hist` and `Clock.t`. We can write this most clearly by annotating the labels in `Read` with assertions that are true when the PC is there.

```
MODULE ClockImpl EXPORT Read =
CONST base       := 2**32
TYPE Word        =  Int SUCHTHAT word IN base.seq)
VAR  lo          :  Word := 0
     hi1         :  Word := 0
     hi2         :  Word := 0
% ABSTRACTION FUNCTION Clock.t = T(lo, hi1, hi2), Clock.Read.t1 = Read.t1Hist,
  Clock.Read.t2 = T(Read.tLo, Read.tH1, read.tH2)
% The PC correspondence is R1 <-> r1, R2 <-> r2, r3, R3 <-> r4
THREAD Tick() = DO VAR newLo: Word, newHi: Word |
     << newLo := lo + 1 // base; newHi := hi1 + 1 >>;
     IF   << newLo # 0  => lo := newLo >>
     [*]  << hi2 := newHi >>; << lo := newLo >>; << hi1 := newHi >>
     FI OD
PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word, [t1Hist: Int] |
     [r1] << tH1 := hi1; [t1Hist := T(lo, hi1, hi2)] >>;
     [r2] % I2: T(lo , tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
          << tLo := lo; >>
     [r3] % I3: T(tLo, tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
          << tH2 := hi2; RET T(tLo, tH1, tH2) >>
     [r4] % I4: $a IN t1Hist .. T(lo, hi1, hi2)
```

[9] L. Lamport, Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems* **8**, 4 (Nov. 1990), pp 305-310.

```
FUNC T(l: Int, h1: Int, h2: Int) -> Int = h2 * base + (h1 = h2 => l [*] 0)
END ClockImpl
```

The whole invariant is thus

h.$pc = r_2 ==> I2 /\ h.$pc = r_3 ==> I3 /\ h.$pc = r_4 ==> I4

The steps of `Read` clearly maintain this invariant, since they don't change the value before `IN`. The steps of `Tick` maintain it by case analysis.

`BufferImpl` implements `Buffer`

The FIFO buffer is another example of easy concurrency, so again we don't need to do a step-by-step proof for it. Instead, it suffices to show that a thread holds the lock `m` whenever it touches the shared variable `b`. Then we can treat the whole critical section during which the lock is held as a big atomic action, and the proof is easy. We will work out the important details of a low-level proof, however, to get some practice in a situation that is slightly more complicated but still straightforward, and to convince you that the theorem about big atomic actions can save you a lot of work.

First, we give the abstraction function; then we use it to show that the code simulates the spec. We use a slightly simplified version of `Produce` that always signals, and we introduce a local variable `temp` to make explicit the non-atomicity of assignment to the shared variable `b`.

Abstraction function

The abstraction function on the state must explain how to interpret a state of the code as a state of the spec. Remember that 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 can 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 spec from a state of the code. So what is a state of the `Buffer` module above? It consists of:

- A sequence of items `b` (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 code is similar, except that it includes the state of the `Mutex` and `Condition` modules.

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

$P1$ — before the body of `Produce`
$P2$ — after the body of `Produce`
$C1$ — before the body of `Consume`
$C2$ — after the body of `Consume`

or as annotations to the code:

```
PROC Produce(t) = [P_1] << b + := {t} >> [P_2]

PROC Consume() -> T =
      [C_1] << b # {} => VAR t := b.head | b := b.tail; RET t >> [C_2]
```

For the code, they are:

- For a thread in `Produce`:

  *p*1 — before `m.acq`
  in `m.acq`—either before or after the action
  *p*2 — before `temp := b + {t}`
  *p*3 — before `b := temp`
  *p*4 — before `c.signal`
  in `c.signal`—either before or after the action
  *p*5 — before `m.rel`
  in `m.rel`—either before or after the action
  *p*6 — after `m.rel`

- For a thread in `Consume`:

  *c*1 — before `m.acq`
  in `m.acq`—either before or after action
  *c*2 — before the test `b # {}`
  *c*3 — before `c.wait`
  in `c.wait`—at beginning, in middle, or at end
  *c*4 — before `t := b.head`
  *c*5 — before `temp := b.tail`
  *c*6 — before `b := temp`
  *c*7 — before `m.rel`
  in `m.rel`—either before or after action
  *c*8 — before `RET t`
  *c*9 — after `RET t`

or as annotations to the code:

```
PROC Produce(t) = VAR temp |
     [p_1] m.acq;
     [p_2] temp = b + {t};
     [p_3] b := temp;
     [p_4] c.signal;
     [p_5] m.rel  [p_6]
PROC Consume() -> T = VAR t, temp |
     [c_1] m.acq;
     DO [c_2] b # {} => [c_3] c.wait OD;
     [c_4] t := b.head;
     [c_5] temp := b.tail; [c_6] b := temp;
     [c_7] m.rel;
     [c_8] RET t [c_9]
```

Notice that we have broken the assignment statements into their constituent atomic actions, introducing a temporary variable `temp` to hold the result of evaluating the right hand side. Also, the PC's in the `Mutex` and `Condition` operations are taken from the specs of those modules (*not* the code; we prove their correctness separately). Here for reference is the relevant code.

```
APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>
APROC signal() = << VAR hs: SET Thread |
     IF  hs <= c /\ hs # {} => c - := hs [*] SKIP FI >>
```

Now we can define the mapping on program counters:

- If a thread `h` is not in `Produce` or `Consume` in the code, then it is not in either procedure in the spec.
- If a thread `h` is in `Produce` in the code, then:

  If `h.$pc` is in {*p*1, *p*2, *p*3} or is in `m.acq`, then in the spec `h.$pc` = *P*1.

  If `h.$pc` is in {*p*4, *p*5, *p*6} or is in `m.rel` or `c.signal` then in the spec `h.$pc` = *P*2.
- If a thread `h` is in `Consume` in the code, then:

  If `h.$pc` ∈ {*c*1, …, *c*6} or is in `m.acq` or `c.wait` then in the spec `h.$pc` = *C*1.

  If `h.$pc` is in {*c*7, *c*8, *c*9} or is in `m.rel` then in the spec `h.$pc` = *C*2.

The general strategy here is to pick, for each atomic step in the spec, some atomic step in the code to simulate it. Here, we have chosen the modification of `b` in the code to simulate the corresponding operation in the spec. Thus, program counters before that point in the code map to program counters before the body in the spec, and similarly for program counters after that point in the code.

This choice of the abstraction function for program counters determines how each step of the code simulates steps of the spec as follows:

- If π is an external step, π simulates the singleton sequence containing just π.
- If π takes a thread from a PC of *p*3 to a PC of *p*4, π simulates the singleton sequence containing just the body of `Produce`.
- If π takes a thread from a PC of *c*6 to a PC of *c*7, π simulates the singleton sequence containing just the body of `Consume`.
- All other steps π simulate the empty sequence.

This example illustrates a typical situation: we usually find that a step in the code simulates a sequence of either zero or one steps in the spec. Steps that have no effect on the abstract state simulate the empty sequence, while steps that change the abstract state simulate a single step in the spec. The proof technique used here works fine if a step simulates a sequence with more than one step 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 variables are `temp` and the item `t`. For threads active in either `Produce` or `Consume`, the abstraction function on `temp` and `t` is the identity; that is, it defines the values of `temp` and `t` in a state of the spec to be the value of the identically named variable in the corresponding operation of the code.

Finally, we need to describe how to construct the state of the buffer `b` from the state of the code. Given the choices above, this is simple: the abstraction function is the identity on `b`.

Proof sketch

To prove the code 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 `h` to be *dominant* if `h.$pc` is in `Produce` and `h.$pc` is in {*p*2, *p*3, *p*4, *p*5} or is at the end of `m.acq`, in `c.signal`, or at the beginning of `m.rel`, or if `h.$pc` is in `Consume` and `h.$pc` is in {*c*2, *c*3, *c*4, *c*5, *c*6, *c*7} or is at the end of `m.acq`, at the beginning or end of `c.wait` (but not in the middle), or at the beginning of `m.rel`.

Now, we claim that the following property is invariant: a thread `h` is dominant if and only if `Mutex.m` = `h`. This simply says that `h` holds the mutex if and only if its PC is at an appropriate point. This is the basic mutual exclusion property. Amazingly enough, given this property we can easily show that operations are mutually exclusive: for all threads `h`, `h'` such that `h` ≠ `h'`, if `h` is dominant then `h'` is not dominant. In other words, at most one thread can be in the middle of one of the operations in the code at any time.

Now let's consider what needs to be shown to prove the code 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 code satisfies the invariants, and then show that each atomic action in the code preserves the invariants. This is left as an exercise.

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

- Consider a step π from *r* to *r*' in which an invocation of an operation occurs for thread `h`. Then in state *r*, `h` was not active in the module, and in *r*', its PC is at the beginning of the operation. This step simulates the identical step in the spec, 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 step.
- Consider a step in which a thread `h` moves from `h.$pc` = *p*3 to `h.$pc` = *p*4, setting `b` to the value stored in temp. The corresponding abstract step sets `b` to `b + {t}`. To show that this step does the right thing, we need an additional invariant:
    If `h.$pc` = *p*3, then `temp = b + {t}`.
  To prove this, we use the fact that if `h.$pc` = *p*3, then no other thread is dominant, so no other step can change `b`. We also have to show that any step that puts `h.$pc` at this point establishes the consequent of the implication — but there is only one step that does this (the one that assigns to `temp`), and it clearly establishes the desired property.

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