17. Formal Concurrency

In this handout we take a more formal view of concurrency than in handout 14 on practical 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, though our treatment is less formal than the one for atomic semantics in handout 9. Next we explain the general method for making large atomic actions out of small ones (easy concurrency) and prove its correctness. We continue with a number of 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 transitions 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 transitions. 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 transition the system non-deterministically chooses an action to execute from all the enabled actions.

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. 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 = a) => c`, so that `c` can only execute when `h`'s program counter equals `a`. Different actions have different values for `a`, 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 := ß`, thus 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 transition. The simple example, given in handout 14, 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 transitions in the two threads. Rather, a non-atomic command corresponds to a *sequence* of atomic transitions, 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.

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 transitions. However, threads are often written to loop indefinitely.

Spec does not have `COBEGIN` or `FORK` constructs, as many programming languages do, these are considerably more difficult to define precisely, since they are tangled up with the control structure of the program. Also, because one Spec thread starts up for every possible argument of the `THREAD` declaration, they tend to be more convenient for most of the specifications and code in this course. To keep the thread from doing anything until a certain point in the computation (or at all), use an initial guard for the entire body as in the `Sieve` example below.

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. So 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.

[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.
[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`.

Each atomic command defines a transition, just as in the sequential semantics. However, now a transition is enabled by the program counter value. That is, a transition can only occur if the program counter of some thread equals the label before the command, and the transition sets the program counter of that thread to the label after the command. If the command at the label in the program counter fails (for example, because it has a guard that tests for a buffer to be non-empty, and the buffer is empty in the current state), the thread is "stuck" and does not make any transitions. However, it may become unstuck later, because of the transitions of some other threads. 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 as we did for the atomic semantics in handout 9, since it involves a number of 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 transitions (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 transition (atomic procedures and functions are taken care of by the atomic semantics).
- The program counters enable transitions: a transition can only occur if the program counter for some thread equals the label before the command, and the transition 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 a and the one after the command is ß, and the transition function defined by `MC(c)` in handout 9 is `(\ s, o | rel)`. Then if `c` is in thread `h`, its transition function is

```
    (\ s, o | s(h+".$pc") = a /\ o(h+".$pc) = ß /\ rel')
```

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

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

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

Labels in Spec

What are the atomic transitions 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 model code in which expression evaluation is not atomic, 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 transition to evaluate the argument and set the `$a` variable, and one to send control to the start of the body. The `RET` command's transition sets `$a` and leaves control at the end of the body. The next transition leaves control after the invocation. So every procedure invocation has at least four transitions: 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 transitions that do not change any other state. These are the transitions that appear in the trace and therefore must be identical in code and a spec.

Minimizing atomicity means that an assignment is broken into separate transitions, 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 transition 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 transition of the compound command is enabled exactly when the initial transition of either of the subcommands is enabled (or if they both are). Thus, the choice is made based only on the first transition. 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 transition for `c1 [*] c2` can only be the initial transition of `c2` if the initial transition 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 transition; 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 transition of `c` is enabled when the program counter is the label on the `DO`, and the last transition 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.

As a consequence of this labeling, as we said before, a procedure invocation has
one transition to evaluate the argument expression,
one to set the program counter to the label immediately before the procedure body,
one for every transition 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 transition 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:

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

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 transitions do not affect the `$a` variable that communicates the argument and result values. That variable is set by the internal transitions 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 will do the work of the procedure body, and a second that retrieves the result. This style is used in I/O automata[4], but we will not use it.

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. Since the threads may not be synchronized, we must ensure that all the numbers = $n^{1/2}$ 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))
```

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

```
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 // p # 0) =>
           << primes \/ := {n} >>
       [*] SKIP
       FI;
       << done \/ := {n} >>                            % No more transitions

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". Differences from `Sieve1` are boxed.

```
CONST nThreads   := 10

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

THREAD Sieve2(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 \/ := {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.[5] It doesn't tell you where to write the brackets. If the environment in which the program has to run 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, and 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)
```

[5] 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 ....`

as well as an arbitrary collection of other commands. We denote the program counter value at the semi-colon by β. We are thinking of the program as

    A; h.$pc = β => B [] C_1 [] C_2 [] ...

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 guard for `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]

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

[6] See Leslie Lamport and Fred B. Schneider. Pretending atomicity. Research Report 44, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, May 1989. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-044.html

'lock conflicts'. See handout 14 on practical concurrency for more details on different kinds of locks.

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	C	Possible sequence?
`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]

Proofs

How can we make 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; [β] B                                          (1)
    << A; B >>                                        (2)
```

As usual, we call the complete program containing (2) the spec *S* and the complete program containing (1) the code *T*. We need an abstraction relation `AR` between the states of *T* and the states of *S* under which every transition of *T* simulates a (possibly empty) trace of *S*. Note that the state spaces of *T* 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 *T*, to avoid confusion with various other uses of *t*.

First we need a precise definition of "`C` is enabled at β and commutes with `A`". For any command `X`, we write `u X u'` for `MC(X)(u, u')`, that is, if `X` relates `u` to `u'`. The idea of 'commutes' is that `<<A; C>>` is the same as `<<C; A>>`, and the definition follows from the meaning of semicolon:

```
    (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`.

[7] Actually, this is not quite right. If you hold several locks, and touch data only when you hold its lock, you have atomicity until you release all the locks.

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.

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

`u("$pc") =` β

So we make `AR` the identity everywhere except at β, where it relates any state 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 *T*. (Since `A` may take many states to `s`, this can't just be an abstraction function.) We write `u ~ s` for "`AR` relates `u` to `s`". Precisely, we say that `u ~ s` 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 transition `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 transitions 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

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

  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'`.

Examples of concurrency

This section contains a number of example specs and codes 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 have to run. Some of the examples of code illustrate easy concurrency (that is, that use locks): `RWLockImpl` and `BufferImpl`. Others illustrate hard concurrency: `SpinLock`, `Mutex2Impl`, `ClockImpl`, `MutexImpl`, and `ConditionImpl`.

Incrementing a register

The first example involves incrementing a register 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 could use the following 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 spec

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

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 (exportable) 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; it is copied from handout 14 on practical concurrency. The state of a mutex is `nil` if the mutex is free, and otherwise is the thread that holds the mutex.

```
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 transition.

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 transition 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 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. 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 # held => RET [*] SKIP FI OD

PROC rel() = << fh := free >>

END SpinLock
```

Of course this code 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. Later, in `MutexImpl`, we present practical code that queues a waiting thread.

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.

Later we will show that `SpinLock` implements `ForgetfulMutex` and that `ForgetfulMutex` implements `Mutex`, from which it follows that `SpinLock` implements `Mutex`.

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 read lock if there are no current write locks.
     <<  SELF IN r \/ w => HAVOC [*] w      = {} => r \/ := {SELF} >>

APROC wAcq() =
% Acquires write lock if there are no current locks.
     <<  SELF IN r \/ w => HAVOC [*] r \/ w = {} => w    := {SELF} >>

APROC rRel() =
% Releases lock if the thread has it; otherwise anything can happen.
     << ~ (SELF IN r)    => HAVOC [*]                r  - := {SELF} >>

APROC wRel() =
     << ~ (SELF IN w)    => HAVOC [*]                w    := {}     >>

END RWLock
```

The following simple code is similar to `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 + := 1 >>
APROC wAcq() = << rw  = 0 => rw := -1 >>

APROC rRel() = << rw - := 1 >>
APROC wRel() = << rw := 0 >>

END ForgetfulRWL
```

We will see later how to code `ForgetfulRWL` using a mutex.

Condition variables

Mutexes are used to protect shared variables. Often a thread `h` cannot proceed until some condition is true of the shared variables, a condition produced by some other thread. Since the variables are protected by a lock, and can be changed only by the thread holding the lock, `h` has to release the lock. It is not efficient to repeatedly release the lock and then re-acquire it to check the condition. Instead, it's better for `h` to *wait* on a *condition variable*, as we saw in handout 14. Whenever any thread changes the shared variables in such a way that the condition might become true, it *signals* the threads waiting on that variable. Sometimes we say that the waiting threads 'wake up' when they are signaled. Depending on the application, a thread may signal one or several of the waiting threads.

Here is the spec for condition variables, copied from handout 14 on practical concurrency.

```
CLASS Condition EXPORT wait, signal, broadcast =

TYPE M = Mutex

VAR  c           :  SET Thread := {}
% Each condition variable is the set of waiting threads.

PROC wait(m) =
     << c \/ := {SELF}; m.rel >>;                        % m.rel=HAVOC unless SELF IN m
     << ~ (SELF IN c) => m.acq >>

APROC signal() = <<
% Remove at least one thread from c. In practice, usually just one.
     IF   VAR t: SET Thread | t <= c /\ t # {} => c - := t [*] SKIP FI >>

APROC broadcast() = << c := {} >>

END Condition
```

As we saw in handout 14, it's not necessary to have a single condition for each set of shared variables. We want enough condition variables so that we don't wake up too many threads whose conditions are not yet satisfied, but not so many that the cost of doing all the `signals` is excessive.

Coding read/write lock using condition variables

This example shows how to use easy concurrency to make more complex locks and scheduling out of basic mutexes and conditions. 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, each one that doesn't have `[*] SKIP`) 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           := m.new()
    c           := c.new()

% ABSTRACTION FUNCTION ForgetfulRWL.rw = rw

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

```
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
```

Coding `Mutex` with memory

The usual way to code `Mutex` is to use an atomic test-and-set operation; we saw this in the `MutexImpl` module above. If such an operation is not available, however, it's possible to code `Mutex` using only atomic read and write operations on memory. This requires an amount of storage linear in the number of threads, however. We give a fair algorithm due to Peterson[8] for two threads; if thread `h` is competing for the mutex, we write `h*` for its competitor.

```
CLASS Mutex2Impl EXPORT acq, rel =

VAR req         :  Thread -> Bool := {* -> false}
    lastReq     :  Int

PROC acq() =
    [$a_{11}$] req(SELF) := true;
    [$a_{12}$] lastReq := SELF;
    DO [$a_2$] (req(SELF*) /\ lastReq = SELF) => SKIP OD [$a_3$]

PROC rel() = req(SELF) := false

END Mutex2Impl
```

This is hard concurrency, and it's tricky to show that it works. To see the idea, consider first a simpler version of `acq` that ensures mutual exclusion but can deadlock:

```
PROC acq0() =
    [$a_1$] req(SELF) := true;
```

[8] G. Peterson, A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Programming Languages and Systems* **5**, 1 (Jan. 1983), pp 56-65.

```
    DO [$a_2$] req(SELF*) => SKIP OD [$a_3$]                 % busy wait
```

We get mutual exclusion because once `req(h)` is true, `h*` can't get from a_2 to a_3. Thus `req(h)` acts as a lock that keeps the predicate `h*.$pc = `a_2 true once it becomes true. Only one of the threads can get to a_3 and acquire the lock.

Of course, `acq0` is no good because it can deadlock—if both threads get to a_2 then neither can progress. `acq` avoids this problem by making it a little easier for a thread to progress: even if `req(h*)`, `h` can take (a_2, a_3) if `lastReq # h`. Intuitively this maintains mutual exclusion because:

> If both threads are at a_2, only the one ? `lastReq`, say `h`, can progress to a_3 and acquire the lock. Since `lastReq` won't change, `h*` will remain at a_2 until `h` releases the lock.
>
> Once `h` has acquired the lock with `h*` not at a_2, `h*` can only reach a_2 by setting `lastReq := h*`, and again `h*` will remain at a_2 until `h` releases the lock.

It ensures progress because the `DO` is the only place to get stuck, and whichever thread is not in `lastReq` will get past it. It ensures fairness because the first thread to get to a_2 is the one that will get the lock first.

There is lots more to say about coding `Mutex` efficiently, especially in the context of shared-memory multiprocessors.[9] On a uniprocessor you still need an implementation that can handle pre-emption; often the most efficient implementation gets the necessary atomicity by modifying the code for pre-emption to detect when a thread is pre-empted in the middle of the mutex code and either complete the operation or back up the state.

Multi-word clock

Often it's possible to get better performance by avoiding locking. Algorithms that do this are called 'wait-free'; we gave a brief discussion in handout 14. Here we present a wait-free algorithm due to Lamport[10] 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.

```
MODULE Clock EXPORT Read =

VAR t           :  Int := 0                             % the current time

THREAD Tick() = DO << t + := 1 >> OD                    % demon thread advances t
```

[9] J. Mellor-Crummey and M. Scott, Algorithms for scalable synchronization of shared-memory multiprocessors. *ACM Transactions on Computer Systems* **9**, 1 (Feb. 1991), pp 21-65. A. Karlin et al., Empirical studies of competitive spinning for a shared-memory multiprocessor. *ACM Operating Systems Review* **25**, 5 (Oct. 1991), pp 41-55.
[10] L. Lamport, Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems* **8**, 4 (Nov. 1990), pp 305-310.

```
PROC Read() -> Int = VAR t1: Int |
    << t1 := t >>; << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >>

END Clock
```

The code below is based on the idea of doing reads and writes of the same multi-word data in opposite orders. `Tick` writes `hi2`, then `lo`, then `hi1`. `Read` reads `hi1`, then `lo`, then `hi2`; if it sees different values in `hi1` and `hi2`, there must have been at least one carry during the read, so `t` must have taken on the value `hi2 * base`. The function `T` expresses this idea. The atomicity brackets in the code are the largest ones that are justified by big atomic actions.

```
MODULE ClockImpl EXPORT Read =

CONST base      := 2**32

TYPE Word       =  Int SUCHTHAT (\ i: Int | i IN base.seq)

VAR  lo         :  Word := 0
     hi1        :  Word := 0
     hi2        :  Word := 0

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 |
    << tH1 := h1 >>;
    << tLo := lo >>;
    << tH2 := h2; RET T(tLo, tH1, tH2) >>

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

END ClockImpl
```

Given this code for reading a two-word clock atomically starting with atomic reads of the low and high parts, it's obvious how to apply it recursively *n*–1 times to read an *n* word clock.

User and kernel mutexes and condition variables

This section presents code for mutexes and condition variables based on the Taos operating system from DEC SRC. Instead of spinning like `SpinLock`, it explicitly queues threads waiting for locks or conditions. The code for mutexes has a fast path that stays out of the kernel in `acq` when the mutex is free, and in `rel` when no other thread is waiting for the mutex. There is also a fast path for `signal`, for the common case that there's nobody waiting on the condition. There's no fast path for `wait`, since that always requires the kernel to run in order to reschedule the processor (unless a `signal` sneaks in before the kernel gets around to the rescheduling).

Notes on the code for mutexes:

1. `MutexImpl` maintains a queue of waiting threads, blocks a waiting thread using `Deschedule`, and uses `Schedule` to hand a ready thread over to the scheduler to run.

2. `SpinLock` and `ReleaseSpinLock` acquire and release a global lock used in the kernel to protect thread queues. This is OK because code running in the kernel can't be pre-empted.

3. The loop in `acq` serves much the same purpose as a loop that waits on a condition variable. If the mutex is already held, the loop calls `KernelQueue` to wait until it becomes free, and then tries again. `rel` calls `KernelRelease` if there's anyone waiting, and `KernelRelease` allows just one thread to run. That thread returns from its call of `KernelQueue`, and it will acquire the mutex unless another thread has called `acq` and slipped in since the mutex was released (roughly).

4. There is clumsy code in `KernelQueue` that puts the thread on the queue and then takes it off if the mutex turns out to be free. This is not a mistake; it avoids a race with `rel`, which calls `KernelRelease` to take a thread off the queue only if it sees that the queue is not empty. `KernelQueue` changes `q` and looks at `s`; `rel` uses the opposite order to change `s` and look at `q`.

This opposite-order access pattern often works in hard concurrency, that is, when there's not enough locking to do the job in a straightforward way. We saw another version of it in `Mutex2Impl`, which sets `req(h)` before reading `req(h*)`. In this case `req(h)` acts like a lock to keep `h*.$pc =` a_2 from changing from true to false. We also saw it in `ClockImpl`, where the reader and the writer of the clock touch its pieces in the opposite order.

The boxes show how the state, `acq`, and `rel` differ from the versions in `SpinLock`.

```
CLASS MutexImpl EXPORT acq, rel =                    % implements ForgetfulMutex

TYPE FH         =  Mutex.FH
VAR  fh         := free
     q          :  SEQ Thread := {}

PROC acq() = VAR t: FH |
     DO << t := fh; fh := held >>; IF t#held => RET [*] SKIP FI; KernelQueue() OD

PROC rel() = fh := free; IF q # {} => KernelRelease() [*] SKIP FI

% KernelQueue and KernelRelease run in the kernel so they can hold the spin lock and call the scheduler.

PROC KernelQueue() =
% This is just a delay until there's a chance to acquire the lock. When it returns acq will retry.
% Queuing SELF before testing fh ensures that the test in rel doesn't miss us.
% The spin lock keeps KernelRelease from getting ahead of us.
     SpinLock();                                     % indented code holds the lock
          q + := {SELF};
          IF   fh = free => q := q.reml              % undo previous line; will retry at acq
          [*]  Deschedule(SELF)                      % wait, then retry at acq
          FI;
     ReleaseSpinLock()

PROC KernelRelease() =
     SpinLock();                                     % indented code holds the lock
          IF q # {} => Schedule(q.head); q := q.tail [*] SKIP FI;
     ReleaseSpinLock()
     % The newly scheduled thread competes with others to acquire the mutex.

END MutexImpl
```

Now for conditions. Note that:

The 'event count' `ecSig` deals with the standard 'wakeup-waiting' race condition: the `signal` arrives after the `m.rel` but before the thread is queued. Note the use of the global spin lock as part of this. It looks as though `signal` always schedules exactly one thread if the queue is not empty, but other threads that are in `wait` but have not yet acquired the spin lock may keep running; in terms of the spec they are awakened by `signal` as well.

`signal` and `broadcast` test for any waiting threads without holding any locks, in order to avoid calling the kernel in this common case. The other event count `ecWait` ensures that this test doesn't miss a thread that is in `KernelWait` but hasn't yet blocked.

```
CLASS ConditionImpl EXPORT wait, signal, broadcast =    % implements Condition

TYPE M          =  Mutex

VAR  ecSig      :  Int := 0
     ecWait     :  Int := 0
     q          :  SEQ Thread := {}

PROC wait(m) = VAR i := ecSig | m.rel; KernelWait(i); m.acq

PROC signal() = VAR i := ecWait |
     ecSig + := 1; IF q # 0 \/ i # ecWait => KernelSig

PROC broadcast() = VAR i := ecWait |
     ecSig + := 1; IF q # 0 \/ i # ecWait => KernelBroadcast

PROC KernelWait(i: Int) =                               % internal kernel procedure
     SpinLock();                                        % indented code holds the lock
          ecWait + := 1;
          % if ecSig changed, there must have been a Signal, so return, else queue
          IF i = ecSig => q + := {SELF}; Deschedule(SELF) [*] SKIP FI;
     ReleaseSpinLock()

PROC KernelSig() =                                      % internal kernel procedure
     SpinLock();                                        % indented code holds the lock
          IF q # {} => Schedule(q.head); q := q.tail [*] SKIP FI;
     ReleaseSpinLock()

PROC KernelBroadcast() =
     SpinLock();                                        % indented code holds the lock
          DO q # {} => Schedule(q.head); q := q.tail OD;
     ReleaseSpinLock()

END ConditionImpl
```

The code for mutexes and conditions are quite similar; in fact, both are cases of a general semaphore.

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 transitions, 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, in order to reduce the number of transitions that your proofs need to consider. If you are programming with easy concurrency, that is, if your code uses a standard locking discipline, this will get rid of nearly all the work. If you are doing hard concurrency, there will still be lots of transitions, 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 transitions of the code to correspond one-for-one with the transitions 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 transition of the code simulates a sequence of transitions of the spec. An external transition must simulate a sequence that contains exactly one instance of the same external transition and no other external transitions; it can also contain any number of internal transitions. An internal transition must simulate a sequence that contains only internal transitions.

Here, once again, are the definitions:

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

> *Start*: If u is any initial state of T then $F(u)$ is an initial state of S.
>
> *Step*: If u and $F(u)$ are reachable states of T and S respectively, and (u, π, u') is a step of T, 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 T 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*(T) and *states*(S) such that:

> *Start*: If u is any initial state of T then there exists an initial state s of S such that $(u, s) \in AR$.
>
> *Step*: If u and s are reachable states of T and S respectively, $(u, s) \in AR$, and (u, π, u') is a step of T, then there is an execution fragment of S from s to some s' having the same trace, and such that $(u', s') \in AR$.

Theorem: If there exists an abstraction function or relation from T to S then T implements S; that is, every trace of T 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 won't work. Be warned.

Owicki-Gries proofs

Owicki and Gries invented a special case of this general method that is sometimes useful.[11] Their idea is to do an ordinary sequential proof of correctness for each thread `h`, annotating each

[11] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs. *Acta Informatica* **6**, 1976, pp 319-340.

atomic command in the usual style with an assertion that is true at that point if `h` is the only thread running. This proof shows that the code of `h` establishes each assertion. Then you show that each of these assertions remains true after any command that any other thread can execute while `h` is at that point. This condition is called 'non-interference'; meaning not that other threads don't interfere with *access* to shared variables, but rather that they don't interfere with the *proof*.

The Owicki-Gries method amounts to defining an invariant of the form

```
h.$pc = a ==> A_a /\ h.$pc = ß ==> A_ß /\ ...
```

and showing that it's an invariant in two steps: first, that every step of `h` maintains it, and then that every step of any other thread maintains it. The hope is that this decomposition will pay because the most complicated parts of the invariant have to do with private variables of `h` that aren't affected by other threads.

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 `BufferImpl` code for a FIFO buffer, the `SpinLock` code for a mutex (given in two versions), the `Mutex2Impl` code for a mutex used by two threads, and the `ClockImpl` code for a multi-word clock.

The amount of detail in these proofs is uneven. The proof of the FIFO buffer code and the second proof of the `Spinlock` code are the most detailed. The others give the abstraction functions and key invariants, but do not discuss 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 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 condition variable to prevent threads in `acq` from busy-waiting when they don't see the condition they require. It also uses a mutex to restrict accesses to `rw`, so that a series of accesses to `rw` can be done atomically.

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`. 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.

`BufferImpl` implements `Buffer`

The FIFO buffer is another example of easy concurrency, so again we don't need to do a transition-by-transition 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, in order to get some practice in a situation that is slightly more complicated but still straightforward, and in order 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 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:

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`

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` $\in \{c_1, \ldots, c_6\}$ 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 transition in the spec, some atomic transition 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 transition of the code simulates transitions of the spec as follows:

- If π is an external transition, 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 transitions π simulate the empty sequence.

This example illustrates a typical situation: we usually find that a transition in the code simulates a sequence of either zero or one transitions in the spec. Transitions that have no effect on the abstract state simulate the empty sequence, while transitions that change the abstract state simulate a single transition in the spec. The proof technique used here works fine if a transition simulates 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 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 transition are preserved by the abstraction function. Let's consider a couple of examples.

- Consider a transition 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 transition simulates the identical transition 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 transition.

- Consider a transition 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 transition sets `b` to `b + {t}`. To show that this transition 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.$pc` = p_3, then no other thread is dominant, so no other transition can change `b`. We also have to show that any transition that puts `h.$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 `b` 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 abstract effect, i.e., if they take r to r', then $AF(r) = AF(r')$. This is left as an exercise.

`SpinLock` implements `Mutex`, first version

The proof is done in two layers. First, we show that `ForgetfulMutex` implements `Mutex`. Second, we show that `SpinLock` implements `ForgetfulMutex`. For convenience, we repeat the definitions of the two modules.

```
CLASS Mutex EXPORT acq, rel =

VAR  m           :  (Thread + Null) := nil

PROC acq() = << m = nil  => m := SELF; RET >>
PROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

END Mutex
```

```
CLASS ForgetfulMutex EXPORT acq, rel =

TYPE M           =  ENUM[free, held]
VAR  m           := free

PROC acq() = << m = free => m := held; RET >>
PROC rel() = << m := free; RET >>

END ForgetfulMutex
```

Proof that `ForgetfulMutex` implements `Mutex`

These two modules have the same atomicity. The difference is that `ForgetfulMutex` forgets which thread owns the mutex, and so it can't check that the "right" thread releases it. We use an abstraction relation *AR*. It needs to be multi-valued in order to put back the information that is forgotten in the code. Instead of using a relation, we could use a function and history variables to keep track of the owner and havoc. The single-level proof given later on that `Spinlock` implements `Mutex` uses history variables.

The main interesting relationship that *AR* must express is:

s.`m` is non-`nil` if and only if *u*.`m` = `held`.

In addition, *AR* must include less interesting relationships. For example, it has to relate the `$pc` values for the various threads. In each module, each thread is either not there at all, before the body, or after the body. Thus, *AR* also includes the condition:

The `$pc` value for each thread is the same in both modules.

Finally, there is the technicality of the special `$havoc = true` state that occurs in `Mutex`. We handle this by allowing *AR* to relate all states of `ForgetfulMutex` to any state with `$havoc = true`.

Having defined *AR*, we just show that the two conditions of the abstraction relation definition are satisfied.

The start condition is obvious. In the unique start states of both modules, no thread is in the module. Also, if *u* is the state of `ForgetfulMutex` and *s* is the state of `Mutex`, then we have *u*.`m = free` and *s*.`m = nil`. It follows that $(u, s) \in AR$, as needed.

Now we turn to the step condition. Let *u* and *s* be reachable states of `ForgetfulMutex` and `Mutex`, respectively, and suppose that (u, π, u') is a step of `ForgetfulMutex` and that $(u, s) \in AR$. If *s*.`$havoc`, then it is easy to show the existence of a corresponding execution fragment of `Mutex`, because any transition is possible. So we suppose that *s*.`$havoc = false`. Invocation and response steps are straightforward; the interesting cases are the internal steps.

So suppose that π is an internal action of `ForgetfulMutex`. We argue that the given step corresponds to a single step of `Mutex`, with "the same" action. There are two cases:

1. π is the body of an `acq`, by some thread `h`. Since `acq` is enabled in `ForgetfulMutex`, we have *u*.`m = free`, and `h.$pc` is right before the `acq` body in *u*. Since $(u, s) \in AR$, we have *s*.`m = nil`, and also `h.$pc` is just before the `acq` body in *s*. Therefore, the `acq` body for thread `h` is also enabled in `Mutex`. Let *s*' be the resulting state of `Mutex`.

   By the code, *u*'.`m = held` and *s*'.`m = h`, which correspond correctly according to *AR*. Also, since the same thread `h` gets the mutex in both steps, the PC's are changed in the same way in both steps. So $(u', s') \in AR$.

2. π is the body of a `rel`, by some thread `h`. If *u*.`m = free` then `ForgetfulMutex` does something sensible, as indicated by its code. But since $(u, s) \in AR$, *s*.`m = nil` and `Mutex` does `HAVOC`. Since `$havoc` in `Mutex` is defined to correspond to everything in `ForgetfulMutex`, we have $(u', s') \in AR$ in this case.

   On the other hand, if *u*.`m = held` then `ForgetfulMutex` sets *u*'.`m := free`. Since $(u, s) \in AR$, we have *s*.`m` ≠ `nil`. Now there are two cases: If *s*.`m` = `h`, then corresponding changes occur in both modules, which allows us to conclude $(u', s') \in AR$. Otherwise, `Mutex` goes to `$havoc = true`. But as before, this is OK because `$havoc = true` corresponds to everything in `ForgetfulMutex`.

The conclusion is that every trace of `ForgetfulMutex` is also a trace of `Mutex`. Note that this proof does not imply anything about liveness, though in fact the two modules have the same liveness properties.

Proof that `SpinLock` implements `ForgetfulMutex`

We repeat the definition of `SpinLock`.

```
CLASS SpinLock EXPORT acq, rel =

TYPE M          =  ENUM[free, held]
VAR  m          := free

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

END SpinLock
```

These two modules use the same basic data. The difference is their atomicity. Because they use the same data, an abstraction function *AF* will work. Indeed, the point of introducing `ForgetfulMutex` was to take care of the need for history variables or abstraction relations there.

The key to defining *AF* is to identify the exact moment in an execution of `SpinLock` when we want to say the abstract `acq` body occurs. There are two logical choices: the moment when a thread converts *u*.`m` from `free` to `held`, or the later moment when the thread discovers that it has done this. Either will work, but to be definite we consider the earlier of these two possibilities.

Then *AF* is defined as follows. If *u* is any state of `SpinLock` then *AF*(*u*) is the unique state *s* of `ForgetfulMutex` such that:

- *s*.`m` = *u*.`m`, and
- The PC values of all threads "correspond".

We must define the sense in which the PC values correspond. The correspondence is straightforward for threads that aren't there, or are engaged in a `rel` operation. For a thread `h` engaged in an `acq` operation, we say that

- `h.$pc` in `ForgetfulMutex`, *s*.`h.$pc`, is just before the body of `acq` if and only if *u*.`h.$pc` is in `SpinLock` either (a) at the `DO`, and before the test-and-set ,or (b) after the test-and-set with `h`'s local variable `t` equal to `held`.
- `h.$pc` in `ForgetfulMutex`, *s*.`h.$pc`, is just after the body of `acq` if and only if *u*.`h.$pc` is either (a) after the test-and-set with `h`'s local variable `t` equal to `free` or (b) after the `t # held` test.

The proof that this is an abstraction function is interesting. The start condition is easy. For the step condition, the invocation and response cases are easy, so consider the internal steps. The `rel` body corresponds exactly in both modules, so the interesting steps to consider are those that are part of the `acq`. `acq` in `SpinLock` has two kinds of internal steps: the atomic test-and-set and the test for `t # held`. We consider these two cases separately:

1) The atomic test-and-set, (*u*, *test-and-set*, *u*'). Say this is done by thread `h`. In this case, the value of `m` *might* change. It depends on whether the step of `SpinLock` changes `m` from `free` to `held`. If it does, then we map the step to the `acq` body. If not, then we map it to the empty sequence of steps. We consider the two cases separately:

   a) The step changes `m`. Then in `SpinLock`, `h.$pc` moves after the test-and-set with `h`'s local `variable t = free`. We claim first that the `acq` body in `ForgetfulMutex` is enabled in state $AF(u)$. This is true because it requires only that s`.m = free`, and this follows from the abstraction function since u`.m = free`. Then we claim that the new states in the two modules are related by AF. To see this, note that `m = held` in both cases. And the new PC's correspond: in `ForgetfulMutex`, `h.$pc` moves to right after the `acq` body, which corresponds to the position of `h.$pc` in `SpinLock`, by the definition of the abstraction function.

b) The step does not change `m`. Then `h.$pc` in `SpinLock` moves to the test, with `t = held`. Thus, there is no change in the abstract value of `h.$pc`.

2) The test for `t # held`, (u, *test*, u'). Say this is done by thread `h`. We always map this to the empty sequence of steps in `ForgetfulMutex`. We must argue that this step does not change anything in the abstract state, i.e., that $AF(u') = AF(u)$. There are two cases:

a) If `t = held`, then the step of `SpinLock` moves `h.$pc` to after the `DO`. But this does not change the abstract value of `h.$pc`, according to the abstraction function, because both before and after the step, the abstract `h.$pc` value is before the body of `acq`.

b) On the other hand, if `t = free`, then the step of `SpinLock` moves `h.$pc` to after the `=>`. Again, this does not change the abstract value of `h.$pc` because both before and after the step, the abstract `h.$pc` value is after the body of `acq`.

`SpinLock` implements `Mutex`, second version

Now we show again that `SpinLock` implements `Mutex`, this time with a direct proof that combines the work done in both levels of the proof in the previous section. For contrast, we use history variables instead of an abstraction relation.

Abstraction function

As usual, we need to be precise about what constitutes a state of the code and what constitutes a state of the spec. A state of the spec 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 spec.

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

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

Now we have a problem: there is no way to define an abstraction function from a code state to a spec state. The problem here is that the code does not record which thread holds the mutex, yet the spec keeps track of this information. To solve this problem, we have to introduce a history variable or use an abstraction relation. We choose the history variable, and add it as follows:

- We augment the state of the code with two additional variables:

```
ms: (Thread + Null) := nil                          % m in the Spec
hs: Bool := false                                   % $havoc in the Spec
```

- We define the effect of each atomic action in the code on the history variable; written in Spec, this results in the following modified code:

```
PROC acq() = VAR t: FH |
   DO <<t  := m; m := held>>; IF t # held => <<ms := SELF>>; RET [*] SKIP FI OD;

PROC rel() = << m := free; hs := hs \/ (ms # SELF); ms := nil >>
```

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

This treatment of `ms` is the obvious way to keep track of the spec's `m`. Unfortunately, it turns out to require a rather complicated proof, which we now proceed to give. At the end of this section we will see a less obvious `ms` that allows a much simpler proof; skip to there if you get worn out.

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

A_1 — before the body of `acq`
A_2 — after the body of `acq`
R_1 — before the body of `rel`
R_2 — after the body of `rel`

For the code, they are:

- For a thread in `acq`:

a_1 — before the `VAR t`
a_2 — after the `VAR t` and before the `DO` loop
a_3 — before the test-and-set in the body of the `DO` loop
a_4 — after the test-and-set in the body of the `DO` loop
a_5 — before the assignment to `ms`
a_6 — after the assignment to `ms`

- For a thread in `rel`:

r_1 — before the body
r_2 — after the body

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

Here are the routines in `Mutex` annotated with the PC values:

APROC acq() = [A_1] << m = nil => m := SELF >> [A_2]

APROC rel() = [R_1] << m # SELF => HAVOC [*] m := nil >> [R_2]

Here are the routines in `SpinLock` annotated with the PC values:

```
PROC acq() = [$a_1$] VAR t := FH |
     [$a_2$] DO [$a_3$] << t := m; m := held >>;
     [$a_4$] IF t # held =>  [$a_5$] << ms := SELF >>; [$a_6$] RET [*] SKIP FI  OD;

PROC rel() =  [$r_1$] << m := free; hs := hs \/ (ms # SELF); ms := nil >> [$r_2$]
```

Now we can define the mappings on program counters:

- If a thread is not in `acq` or `rel` in the code, then it is not in either in the spec.
- $\{a_1, a_2, a_3, a_4, a_5\}$ 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 spec to have the value of `ms` in the code, and `$havoc` in the spec to have the value of `hs` in the code. As in handout 8, we just throw away all but the spec part of the state.

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

Once again, the abstraction function on program counters determines how transitions in the code simulate sequences of transitions in the spec:

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

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 code map to initial states of the spec. This is easy; all the thread states correspond, and the initial state of `ms` in the code is `nil`.

Next, we show that transitions in the code and the spec 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 `rel` is also straightforward. The hard cases are in the body of `acq`.

Consider all the transitions in `acq` before the one from a_5 to a_6. These all simulate the null transition, so they should leave the abstract state unchanged. And they do, because none of them changes `ms`.

The transition from a_5 to a_6 simulates the transition from A_1 to A_2. There are two cases: when `ms = nil`, and when `ms ? nil`.

1. 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.
2. Now consider the case when `ms ? nil`. We claim this case is possible only if a thread that didn't hold the mutex has done a `rel`. Then `hs = true`, the spec has done `HAVOC`, and anything can happen. In the absence of havoc, 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:

   Either

   If `m = free` then `ms = nil`, and

   If a thread is at a_5, or at a_4 with `t = free`, then `ms = nil`, `m = held`, there are no other threads at a_5, and for all other threads at a_4, `t = held`

   or `hs` is true.

Given this invariant, we are done: we have shown the appropriate correspondence for all the transitions in the code. So we must prove the invariant. We do this by induction. It's vacuously true in the initial state, since no thread could be at a_4 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 `rel` preserves the first conjunct of the invariant because afterwards both `m = free` and `ms = nil`. To prove that the transition in `rel` preserves the second conjunct of the invariant, there are two cases, depending on whether the spec allows `HAVOC`.

1. If it does, then the code sets `hs` true; this corresponds to the `HAVOC` transition in the spec, and thereafter anything can happen in the spec, so any transition of the code simulates the spec. The reason for explicitly simulating `HAVOC` is that the rest of the invariant may not hold after a rogue thread does `rel`. Because the rogue thread resets `m` to `free`, if there's a thread at a_5 or at a_4 with `t = free`, and `m = held`, then after the rogue `rel`, `m` is no longer `held` and hence the second conjunct is false This means that it's possible for several threads to get to a_5, or to a_4 with `t = free`. The invariant still holds, because `hs` is now true.
2. In the normal case `ms ? nil`, and since we're assuming the invariant is true before the transition, this implies that no thread is at a_4 with `t = free` or at a_5. After the transition to r_2 it's still the case that no thread is at a_4 with `t = free` or at a_5, so the invariant is still true.

Now we consider the transitions in `acq`. The transitions from a_1 to a_2 and from a_2 to a_3 obviously preserve the invariant. The transition from a_4 to a_5 puts a thread at a_5, but `t = free` in this case so the invariant is true after the transition by induction. The transition from a_4 to a_3 also clearly preserves the invariant.

The transition from a_3 to a_4 is the first interesting one. We need only consider the case `hs = false`, since otherwise the spec allows anything. This transition certainly preserves the first conjunct of the invariant, since it doesn't change `ms` and only changes `m` to `held`. Now we assume the second conjunct 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_4 with `t = free`. Then we have `m = held` by induction, so after the transition both `t = held` and `m = held`. This preserves the invariant.
2. Before the transition, there are no threads at a_5 or at a_4 with `t = free`. Then after the transition, there is still no thread at a_5, but there is a new thread at a_4. (Any others must have `t = held`.) Now, if this thread has `t = held`, the second part of the invariant is true vacuously; but if `t = free`, then we have:

    `ms = nil` (since when the thread was at a_3 `m` must have been `free`, hence the first part of the invariant applies);

    `m = held` (as a direct result of the transition);

    there are no threads at a_5 (by assumption); and

    there are no other threads at a_4 with `t = free` (by assumption).

    So the invariant is still true after the transition.

Finally, assume a thread `h` is at a_5, about to transition to a_6. If the invariant is true here, then `h` is the only thread at a_5, and all threads at a_4 have `t = 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_4 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.

This proof is a good example of how to use invariants and of the subtleties associated with preconditions. It's possible to give a considerably simpler proof, however, by handling the history variable `ms` in a less natural way. This version is closer to the two-stage proof we saw earlier. In particular, it uses the transition from a_3 to a_4 to simulate the body of `Mutex.acq`. We omit the `hs` history variable and augment the code as follows:

```
PROC acq() = [a1] VAR t := FH |
    [a2] DO [a3] << t := m; m := held; IF t # held => ms := SELF [*] SKIP FI  >>;
    [a4] IF t # held =>  [a6]  RET  [a7] [*] SKIP FI  OD;

PROC rel() =  [r1] << m := free; ms := nil >> [r2]
```

The abstraction function maps `ms` to `Mutex.m` as before, and it maps PC's a_1- a_3 to A_1 and a_6-a_7 to A_2. It maps a_4 to A_1 if `t = held`, and to A_2 if `t = free`; thus a_3 to a_4 simulates `Mutex.acq` only if `m` was `free`, as we should expect. There is no need for an invariant; we only used it at a_5 to a_6, which no longer exists.

The simulation argument is the same as before except for a_3 to a_4, which is the only place where we changed the code. If `m = held`, then `m` and `ms` don't change; hence `Mutex.m` doesn't change, and neither does the abstract PC; in this case the transition simulates the empty trace. If `m = free`, then `m` becomes `held`, `ms` becomes `SELF`, and the abstract PC becomes A_2; in this case the transition simulates A_1 to A_2, as promised.

The moral of this story is that it can make a big difference how you choose the abstraction function. The crucial decision is the choice of the 'critical transition' that models the body of `Mutex.acq`, that is, how to abstract the PC. It seems very natural to change `ms` in the code after the test of `t # held` that is already there, but this forces the critical transition to be after the test. Then there has to be an invariant to carry forward the relationship between the local variable `t` and the global variable `m`, which complicates things, and the `HAVOC` case in `rel` complicates them further by falsifying the natural statement of the invariant and requiring the additional `hs` variable to patch things up. The uglier code with a second test of `t # held` inside the atomic test-and-set command makes it possible to use that action, which does the real work, to simulate the body of `Mutex.acq`, and then everything falls out nicely.

More complicated code requires invariants even when we choose the best abstraction function, as we see in the next two examples.

`Mutex2Impl` implements `Mutex`

This is the rather subtle code that implements a mutex for two threads using only memory reads and writes. First we show that the simple, deadlocking version `acq0` maintains mutual exclusion. Recall that we write `h*` for the thread that is the partner of thread `h`. Here is the code for `acq0`.

```
PROC acq0() =
    [a1] req(SELF) := true;
    DO [a2] req(SELF*) => SKIP OD [a3]
```

Intuitively, we get mutual exclusion because once `req(h)` is true, `h*` can't get from a_2 to a_3. It's convenient to define

```
FUNC Holds0(h: Thread) = RET req(h) /\ h.$pc # a2
```

Abstractly, `h` has the mutex if `Holds0(h)`, and the transition from a_2 to a_3 simulates the body of `Mutex.acq`. Precisely, the abstraction function is

```
    Mutex.m = (Holds0.set = {} => nil [*] Holds0.set.choose)
```

Recall that if `P` is a predicate, `P.set` is the set of arguments for which it is true.

To make precise the idea that `req(h)` stops `h*` from getting to a_3, the invariant we need is

```
    Holds0.set.size <= 1 /\ (h.$pc = a2 ==> req(h))
```

The first conjunct is the mutual exclusion. It holds because, given the first conjunct, only (a_2, a_3) can increase the size of `Holds0.set`, and `h` can take that step only if `req(h*) = false`, so `Holds0.set` goes from `{}` to `{h}`. The second conjunct holds because it can never be `true ==> false`, since only the step (a_1, `req(h) := true`, a_2) can make the antecedent true, this step also makes the consequent true, and no step away from a_2 makes the consequent false.

This argument applies to `acq0` as written, but you might think that it's unrealistic to fetch the shared variable `req(SELF*)` and test it in a single atomic action; certainly this will take more than one machine instruction. We can appeal to big atomic actions, since the whole sequence from a_2 to a_3 has only one action that touches a shared variable (the fetch of `req(SELF*)`) and therefore is atomic.

This is the right thing to do in practice, but it's instructive to see how to do it by hand. We break the last line down into two atomic actions:

```
    VAR t | DO [$a_{2}$] << t := req(SELF*) >>; [$a_{21}$] << t => SKIP >> OD [$a_{3}$]
```

We examine several ways to show the correctness of this; they all have the same idea, but the details differ. The most obvious one is to add the conjunct `h.$pc #` a_{21} to `Holds0`, and extend the mutual exclusion conjunct of the invariant so that it covers a thread that has reached a_{21} with `t = false`:

```
    (Holds0.set \/ {h | h.$pc = $a_{21}$ /\ h.t = false}).size <= 1
```

Or we could get the same effect by saying that a thread acquires the lock by reaching a_{21} with `t = false`, so that it's the transition (a_{2}, a_{21}) with `t = false` that simulates the body of `Mutex.acq`, rather than the transition to a_3 as before. This means changing the definition of `Holds0` to

```
FUNC Holds0(h: Thread) =
    RET req(h) /\ h.$pc # $a_{2}$ /\ (h.$pc = $a_{21}$ ==> h.t = false)
```

Yet another approach is to make explicit in the invariant what `h` knows about the global state. One purpose of an invariant is to remember things about the global state that a thread has discovered in the past; the fact that it's an invariant means that those things stay true, even though other threads are taking steps. In this case, `t = false` in `h` means that either `req(h*) = false` or `h*` is at a_2 or a_{21}, in other words, `Holds(h*) = false`. We can put this into the invariant with the conjunct

```
    h.$pc = $a_{21}$ /\ h.t = false ==> Holds(h*) = false
```

and this is enough to ensure that the transition (a_{21}, a_3) maintains the invariant.

We return from this digression on proof methodology to study the non-deadlocking `acq`:

```
PROC acq() =
    [$a_{11}$] req(SELF) := true;
    [$a_{12}$] lastReq := self;
    DO [$a_{2}$] (req(SELF*) /\ lastReq = SELF) => SKIP OD [$a_{3}$]
```

We discussed liveness informally earlier, and we don't attempt to prove it. To prove mutual exclusion, we need to extend `Holds0` in the obvious way:

```
FUNC Holds(h: Thread) = req(h) /\ h.$pc # $a_{12}$ /\ h.$pc # $a_{2}$
```

and add `\/ h.$pc =` a_{12} to the antecedent of the invariant In order to have mutual exclusion, it must be true that `h` won't find `lastReq = h*` as long as `h*` holds the lock. We need to add a conjunct to the invariant to express this. This leaves us with:

```
      Holds0.set.size <= 1
   /\ (h.$pc = $a_{2}$ \/ h.$pc = $a_{12}$ ==> req(h))
   /\ (Holds(h*) /\ h.$pc = $a_{2}$   ==> lastReq = h)
```

The last conjunct holds because (a_{12}, a_2) makes it true, and the only way to make it false is for `h*` to do `lastReq := SELF`, which it can only do from a_{12}, so that `Holds(h*)` is false. With this invariant it's obvious that (a_2, a_3) maintains the invariant.

`ClockImpl` implements `Clock`

The spec says that a `Read` returns some value that the clock had between the beginning and the end of the `Read`. Here it is, with labels added.

```
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 |
    [$R_{1}$] << t1 := t >>; [$R_{2}$] << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >> [$R_{3}$]
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 (\ i: Int | i 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  $R_1 \leftrightarrow r_1$, $R_2 \leftrightarrow r_2$, $r_3$, $R_3 \leftrightarrow r_4$
% Tick and T are unchanged and omitted.

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word, t1Hist: Int |
    [$r_{1}$] << tH1 := hi1; t1Hist := T(lo, hi1, hi2) >>;
    [$r_{2}$] % I2: T(lo , tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
         << tLo := lo; >>
    [$r_{3}$] % I3: T(tLo, tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
         << tH2 := hi2; RET T(tLo, tH1, tH2) >>
    [$r_{4}$] % I4: $a IN t1Hist .. T(lo, hi1, hi2)

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.