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 \( A_1, \ldots, A_n \) then the entire system is just the ‘or’ of all these actions: \( A_1 \ldots A_n \). 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. Thus non-determinism encompasses concurrency.

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’ \( \text{pc} \), and each of its actions has the form \( (h.\text{pc} = a) \Rightarrow c \), 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 \( h.\text{pc} := \bar{p} \), 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 \( \text{pc} \) 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 \texttt{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 \texttt{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 \texttt{COBEGIN} or \texttt{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 \texttt{THREAD} declaration, they tend to be more convenient for most of the specs 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 \texttt{Sieve} example below.

A thread is named by the name in the declaration and the argument values. Thus, the threads declared by \texttt{THREAD Foo(bool) = ...} for example, would be named \texttt{Foo(true)} and \texttt{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 \texttt{Sieve} example appears in the state as \texttt{Sieve(0).p, Sieve(1).p, ...}. If there were a \texttt{PROC Foo} called by \texttt{Sieve} with a local variable \( \texttt{baz} \), the state might be defined at \texttt{Sieve(0).Foo.baz, Sieve(1).Foo.baz, ...}. The pseudo-names \( \$a, \$x, \text{and } \$pc \) 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 \texttt{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 \texttt{fork} and \texttt{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, \( h + \ \texttt{.P.v} \) becomes \( h + \ \texttt{.P.v} + d.\text{enc} \), 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 un stuck 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 \[\cdots\] 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 $\text{\$pc}$, $\alpha$, and $\beta$ 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 $\alpha$ and the one after the command is $\beta$, and the transition function defined by $\text{MC}(c)$ in handout 9 is ($\\text{\$s}$, $\alpha$ | rel). Then if $c$ is in thread $\text{\$h}$, its transition function is

\[
(\\text{\$s}, \alpha | \text{rel}) = (\\text{\$s}, \beta | \text{rel'})
\]

If $c$ is in procedure $\text{\$P}$, that is, $c$ can execute for any thread whose program counter has reached $\alpha$, its transition function is

\[
(\\text{\$s}, \alpha | \text{rel}) = (\\text{\$s}, \beta | \text{rel'})
\]

Here $\text{rel'}$ is $\text{rel}$ with each reference to a local variable $v$ changed to $\text{\$h + \text{\$P.v}}$ or $\text{\$h + \text{\$P.v}}$.

**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 \[\cdots\]). However, expression evaluation must be atomic, or reasoning about expressions would be a mess. To use Spec to model code in which expression evaluation is not atomic (C code, for example), you must add temporary variables. Thus $\text{\$x := a + b + c}$ becomes

\[
\\text{\$var t | \langle t := a \rangle ; \langle t := t + b \rangle ; \langle x := t + c \rangle}
\]

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

The simple commands, \text{\$SKIP}, \text{\$HAVOC}, \text{\$RET}, and \text{\$RAISE}, are atomic, as is any command in atomicity brackets \[\cdots\].

For an invocation, there is a transition to evaluate the argument and set the $\alpha$ variable, and one to send control to the start of the body. The \text{\$RET} command’s transition sets $\alpha$ 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 $\alpha$, send control to the body, do the \text{\$RET} and set $\alpha$, 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 both the spec and the code that implements it.

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 $\text{\$x := \text{\$exp}}$ is

\[
\\text{\$var t | \langle t := \text{\$exp} \rangle ; \langle x := t \rangle}
\]

and $\text{\$p(y)}$ is

\[
\\text{\$p(y) ; \langle x := \text{\$a} \rangle}
\]

Since there are no additional labels for the $c_1 \[\cdots\] c_2$ 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 $\text{\$\text{\$v}}$, except that the initial transition for $c_1 \{\text{\$v}\} c_2$ can only be the initial transition of $c_2$ if the initial transition of $c_1$ is not enabled. And the same is also true for $\text{\$VAR}$. The value chosen for $\text{id}$ in $\text{\$VAR id | c}$ must allow $c$ to make at least one transition; after that the thread might get stuck.

\text{\$DO}$ has a label, but \text{\$\text{\$DO}$} does not introduce any additional labels. The starting and ending program counter value for $c$ in \text{\$DO $c$ \text{\$DO}$} is the label on the \text{\$DO$. Thus, the initial transition of $c$ is enabled when the program counter is the label on the \text{\$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 \text{\$DO.}$

To sum up, there’s a label on each \text{\$\text{\$v}$}, \text{\$\text{\$v}$}, \text{\$\text{\$\text{\$v}$}, and \text{\$DO$ outside of $\langle\cdots\rangle$. 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 $\text{\$v}$ 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 $\text{\$v}$ 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 $\text{\$RET}$ 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() = \[P_1\] \[P_2\] \[P_3\] \[P_4\] \[P_5\] \[P_6\] \[P_7\] \[P_8\] \[P_9\]
IF x > 5 => \[P_2\] x := \[P_4\] \(q(x + 1, 2)\); \[P_5\] y := \[P_6\] 3
\[P_7\] \[P_8\] \[P_9\]
VAR x, y \[P_1\] DO \[P_3\] \[P_4\] OD \[P_5\]

**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 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, 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 \(\sqrt{n}\) to see whether it divides \(n\). Since the threads may not be synchronized, we must ensure that all the numbers \(\leq n^{1/2}\) have been checked before we check \(n\).

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

\[
\text{CONST Odds} = \{i: \text{Nat} \mid i \div 2 = 1 \wedge i > 1\}
\]

\[
\text{VAR primes} : \text{SET Nat} := \{2\}
\]

\[
\text{VAR done} : \text{SET Nat} := \{
\}
\]

\[
\text{INVARIANT (ALL n: \text{Nat} \mid n \text{ IN done} \wedge \text{IsPrime}(n) \Rightarrow n \text{ IN primes} \wedge n \text{ IN primes} \Rightarrow \text{IsPrime}(n))}
\]

\[
\text{THREAD Sieve1(n :IN Odds) =}
\]
\[
\text{IF} \ (\text{ALL p :IN \text{primes} \mid p \leq \sqrt{n} \Rightarrow n \div p \neq 0)}\]
\[
\Rightarrow \{\}
\]

\[
\text{IF} \ \forall \ (n \in \text{primes} \mid n \leq \sqrt{n}) \Rightarrow \Rightarrow \}
\]

\[
\text{FI;}
\]

\[
\text{FIJ;}
\]

\[
\text{VAR x, y \[P_1\] DO \[P_3\] \[P_4\] OD \[P_5\]}
\]

\[
\text{FUNC Sqrt(n: Nat) -> Int =} \text{RET \{ i: Nat \mid i \ast i \leq n \}.max}
\]

**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. 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, or the reduction in concurrency hurts performance too much, so that we have to make do with small ones. For example, in a shared-memory multiprocessor typically only the individual instructions are atomic, and we can only write one disk block atomically. So we are faced with the problem of showing that code with small atomic actions satisfies a spec with bigger ones.

**The idea**

The standard way of doing this is by some kind of ‘non-interference’. The idea is based on the following observation. Suppose we have a program with a thread \(t\) that contains the sequence

\[
A_1; B
\]

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 \(\text{VAR} \ t_1, t_2 \mid t_1 := x(i); t_2 := f(y); \ldots t_1 + t_2 \ldots\).
as well as an arbitrary collection of other commands. We denote the program counter value before \( \alpha \) by \( \alpha \) and at the semi-colon by \( \beta \). We are thinking of the program as

\[
h \cdot \text{spc} = \alpha \Rightarrow A \] \[h \cdot \text{spc} = \beta \Rightarrow B \] ... \[C_1 \] \[C_2 \] ... \]

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 guards for \( \alpha \) and \( \beta \) explicitly.

Suppose \( \beta \) denotes an arbitrary atomic command, and \( \lambda \) denotes an atomic command that commutes with every command in the program (other than \( \beta \)) that is enabled when \( h \) is at the semicolon, that is, when \( h \cdot \text{spc} = \beta \). (We give a precise meaning for `commutes' below.) In addition, both \( \alpha \) and \( \beta \) 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

\[
<< \alpha \; \beta >>
\]

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

Thus we have achieved the goal of making a bigger atomic command \( << \alpha \; \beta >> \) out of two small ones \( \alpha \) and \( \beta \). We can call the big command \( \beta \) and repeat the process on \( \beta \; \\beta \) to get a still bigger command \( << \alpha \; \beta \; \beta >> \).

How do we ensure that only a command \( \beta \) that commutes with \( \alpha \) can execute while \( h \cdot \text{spc} = \beta \)?

The simplest way is to ensure that the variables that \( \gamma \) touches (reads or writes) are disjoint from the variables that \( \alpha \) 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 \( \alpha \) 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 \( h \cdot \text{spc} = \beta \) is \( \beta \). 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.

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 \( \text{pc} \) is in the command, then no other thread’s \( \text{pc} \) 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 \( h \cdot \text{spc} = \beta \). So if every command in any critical section for \( v \) only touches \( v \) (and perhaps local variables), then the program simulates another program in which every critical section is an atomic command. A critical section is usually coded by acquiring a lock or mutex and holding it for the duration of the section. The property of a lock is that it’s not possible to acquire it when it is already held, and this ensures the mutual exclusion property for critical sections.

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

`lock conflicts'. 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):

\[
\begin{array}{ccc}
\alpha & \beta & \text{Possible sequence?} \\
m \cdot \text{acq}(h) & m \cdot \text{acq}(h') & \text{No: } c \text{ is blocked by } h \text{ holding } m \\
m \cdot \text{rel}(h) & m \cdot \text{rel}(h') & \text{No: } c \text{ won’t be reached because } h' \text{ doesn’t hold } m \\
m \cdot \text{rel}(h) & m \cdot \text{acq}(h') & \text{OK} \\
m \cdot \text{acq}(h) & m \cdot \text{rel}(h') & \text{No: one thread doesn’t hold } m, \text{ hence won’t do } \text{rel} \\
\end{array}
\]

So \( m \cdot \text{acq} \) commutes with everything that’s enabled at \( \beta \) since neither mutex operation is enabled at \( \beta \) in a program that avoids havoc. But \( m \cdot \text{rel}(h) \) doesn’t commute with \( m \cdot \text{acq}(h') \). The reason is that the \( \alpha \) sequence can happen, but the \( \beta \) sequence \( m \cdot \text{acq}(h') \); \( m \cdot \text{rel}(h) \) cannot, because in this case \( h \) doesn’t hold \( m \) and therefore can’t be doing \( \text{rel} \). Hence it’s not possible to flip every \( c \) in front of \( m \cdot \text{rel}(h) \) in order to make \( \alpha \) 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.

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

**Proofs**

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

\[
<< \alpha \; \beta >>
\]

\[
<< \alpha \; \beta >>
\]

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 \( \lambda \) 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 \( h \cdot \text{spc} \) can never be \( \beta \) in \( S \). We use \( s \) and \( u \) for states of \( S \) and \( T \), to avoid confusion with various other uses of \( t \).

---


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.
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:

\( (\text{ALL } u_1, u_2 \mid (\text{EXISTS } u \mid u_1 A u \land u C u_2 \land u("h.$pc") = \beta) \implies (\text{EXISTS } u' \mid u_1 C u' \land u' A u_2)) \)

This says that any result that you could get by doing \( A; C \), you could also get by doing \( C; A \). 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.

\[ s \xrightarrow{\text{SKIP}} s' \]
\[ A \]
\[ u("h.$pc") = \beta \]
\[ B \]
\[ s \]
\[ s' \]

So we make \( AR \) the identity everywhere except at \( \beta \), where it relates any state \( u \) that can be reached from \( s \) by \( A \) to \( s \). This expresses the intention that at \( \beta \) we haven’t yet done \( A \) in \( s \), but we have done \( A \) in \( T \). (Since \( A \) may take many states to \( \beta \), 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

\[ u("h.$pc") \neq \beta \land s = u \]
\[ \lor u("h.$pc") = \beta \land s A u \]

Why is this an abstraction relation? It certainly relates an initial state to an initial state, and it certainly works for any transition \( u \rightarrow u' \) that stays away from \( \beta \), that is, in which \( u("h.$pc") \neq \beta \) and \( u'("h.$pc") \neq \beta \), since the abstract and concrete states are the same. What about transitions that do involve \( \beta \)?

- If \( h.$pc \) changes to \( \beta \) then we must have executed \( A \). The picture is

\[ s \]
\[ A \]
\[ u("h.$pc") = \beta \]
\[ B \]
\[ s' \]

The abstract trace is empty, so the abstract state doesn’t change: \( s = s' \). Also, \( s' = u \) because only equal states are related when \( h.$pc \neq \beta \). But we executed \( A \), so \( u A u' \), so \( s' = u' \) because of the equalities.

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

\[ s \]
\[ B \]
\[ s' \]

This shows that \( s \) is related to \( s' \) by \( B \). 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 \( u_1 = s, u_2 = u', \) 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:

```plaintext
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() = \(< < x := x + 1 >>\)

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

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

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

Or, more explicitly:

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

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

** Mutexes**

Here is a spec of a simple `Mutex` module, which can be used to ensure mutually exclusive execution of critical sections; 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.

```plaintext
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.

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

END Mutex
```

If a thread invokes `acq` when \( m \neq \text{nil} \), 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:

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

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

** Spin locks**

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

Here is incorrect code:

```plaintext
CLASS BadSpinLock
    EXPORT acq, rel =

    TYPE FH = ENUM[free, held]
    VAR fh := free

    PROC acq() = \(< << fh = held => SKIP >> OD; \>
        % wait for fh = free
    PROC rel() = \(< << fh := free >>\)

END BadSpinLock
```

This is wrong because two concurrent invocations of `acq` could both find `fh = free` 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.

```plaintext
CLASS SpinLock
    EXPORT acq, rel =

    TYPE FH = ENUM[free, held]
    VAR fh := free

    PROC acq() = VAR t: FH |
        DO \(< << t := fh; fh := held >>; \>
            % and acquire it
        IF t = free => 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 give 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. We don’t give the abstraction function here because it involves the details of program counters.

### 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 := {}  

    PROC rAcq() = << r := {} >>
    PROC wAcq() = << w := {} >>
    PROC rRel() = << r := {} >>
    PROC wRel() = << 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

    PROC rAcq() = << rw + := 1 >>
    PROC wAcq() = << rw := -1 >>
    PROC rRel() = << rw - := 1 >>
    PROC 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 := {}

    PROC wait(m) =
        c := {SELF}; m.rel >>
    
    PROC signal() =
        IF c # {} THEN c - := t; c := c | t >>
    
    PROC 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 on the mutex.

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

**CLASS RWLockImpl**

```plaintext
EXPORT rAcq, rRel, wAcq, wRel =
% implements ForgetfulRWL
VAR 
trw : 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 rRel(l) =
   m.acq; rw - := 1;
   [if rw = 0 => c.signal [*] SKIP FI; m.rel]
PROC wAcq(l) =
   m.acq; DO
   ~ rw = 0 => c.wait(m) OD;
   rw := -1;
   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]**

```plaintext
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]**

```plaintext
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**

```plaintext
EXPORT acq, rel =
VAR 
req : Thread -> Bool := {* -> false}
lastReq : Int
PROC acq() =
   [a] req(SELF) := true;
   [a1] lastReq := SELF;
   DO [a1] (req(SELF*) /\ lastReq = SELF) => SKIP OD [a3]
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:

```plaintext
PROC acq0() =
   [a] req(SELF) := true;
   [a1] lastReq := SELF;
   DO [a1] (req(SELF*) /\ lastReq = SELF) => SKIP OD [a3]
PROC rel0() = req(SELF) := false
END Mutex2Impl
```

We get mutual exclusion because once \( \text{req}(h) \) is true, \( h^* \) can’t get from \( a_3 \) to \( a_6 \). Thus \( \text{req}(h) \) acts as a lock that keeps the predicate \( h^*, \text{spc} = a_5 \) true once it becomes true. Only one of the threads can get to \( a_5 \) and acquire the lock. We might call the algorithm ‘polite’ because each thread defers to the other one at \( a_2 \).

Of course, \( \text{acq0} \) is no good because it can deadlock—if both threads get to \( a_2 \) then neither can progress. \( \text{acq} \) avoids this problem by making it a little easier for a thread to progress: even if \( \text{req}(h^*) \), \( h \) can take \( a_5, a_2 \) if last\( \text{req} \neq h \). Intuitively this maintains mutual exclusion because:

If both threads are at \( a_2 \), only the one \( h \neq \text{last}\( \text{req} \), say \( h \), can progress to \( a_5 \) and acquire the lock. Since last\( \text{req} \) won’t change, \( h^* \) will remain at \( a_5 \) until \( h \) releases the lock.

Once \( h \) has acquired the lock with \( h^* \) not at \( a_2 \), \( h^* \) can only reach \( a_5 \) by setting last\( \text{req} := h^* \), and again \( h^* \) will remain at \( a_5 \) until \( h \) releases the lock.

It ensures progress because the \( \text{DO} \) is the only place to get stuck, and whichever thread is not in last\( \text{req} \) 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.

Abstractly, \( h \) has the mutex if \( \text{req}(h) \land h^*, \text{spc} \neq a_2 \), and the transition from \( a_2 \) to \( a_5 \) simulates the body of \( \text{Mutex}.\text{acq} \). Precisely, the abstraction function is

\[
\text{Mutex}.m = \{ \text{Holds0}.set = \{ \} \Rightarrow \text{nil} \} \text{Holds0}.set.\text{choose}
\]

We sketch the proof that \( \text{Mutex2Impl} \) implements \( \text{Mutex} \) later.

There is lots more to say about coding \( \text{Mutex} \) efficiently, especially in the context of shared-memory multiprocessors.\(^9\) Even 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 \( \text{Read} \) returns some value that the clock had between the beginning and the end of the \( \text{Read} \). As we saw in handout 8 on generalized abstraction functions,

\[\text{FUNC} T(l: \text{Int}, h1: \text{Int}, h2: \text{Int}) \rightarrow \text{Int} = h2 \times \text{base} + (h1 = h2 \Rightarrow l [\ast] 0)\]

where this spec is called \( \text{LateClock} \), it takes a prophecy variable to show that this spec is equivalent to the simpler spec that just reads the clock value.

**MODULE Clock**

\[
\text{EXPORT Read} = \\
\text{VAR} t : \text{Int} := 0 \ % \text{the current time} \\
\text{THREAD Tick() = DO} \langle t + := 1 \rangle \ OD \ % \text{demon thread advances} t \\
\text{PROC Read() -> Int} \langle \text{VAR t1: Int} \mid \\
\langle t1 := t \rangle; \langle \text{VAR t2: Int} \mid t1 := t2 \wedge t2 := t \Rightarrow \text{RET t2} \rangle \rangle
\]

**END Clock**

The code below is based on the idea of doing reads and writes of the same multi-word data in opposite orders. \( \text{Tick} \) writes \( hi2 \), then \( lo \), then \( hi1 \). \( \text{Read} \) 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 \( hi2 \times \text{base} \). The function \( T \) expresses this idea. The atomicity and fairness in the code are the largest ones that are justified by big atomic actions.

**MODULE ClockImpl**

\[
\text{EXPORT Read} = \\
\text{CONST base} := 2^{\ast}32 \\
\text{TYPE Word} = \text{Int SUCHTHAT} (\ i: \text{Int} \mid i \text{ IN base}.\text{seq}) \\
\text{VAR lo : Word := 0} \\
\text{hi1 : Word := 0} \\
\text{hi2 : Word := 0} \\
\% \text{ABSTRACTION FUNCTION} \ \\
\text{Clock.t} \rightarrow T(lo, hi1, hi2), \text{Clock.Read.t1} \rightarrow \text{Read.t1Hist}, \\
\text{Clock.Read.t2} \rightarrow T(\text{Read.tLo, Read.tH1, read.th2}) \\
\text{THREAD Tick() = DO} \text{VAR newLo: Word, newHi: Word} \mid \\
\langle <\text{newLo} := lo + 1 \text{ // base}; \text{newHi} := hi1 + 1 \rangle; \\
\text{IF} \langle <\text{newLo} \neq 0 \Rightarrow lo := \text{newLo} \rangle; \langle <\text{hi2} := \text{newHi} \rangle; \langle <\text{lo} := \text{newLo} \rangle; \langle <\text{hi1} := \text{newHi} \rangle \rangle \ \text{FI \ OD} \\
\text{PROC Read() -> Int} \text{VAR tLo: Word, tH1: Word, tH2: Word} \mid \\
\langle <\text{tH1} := \text{hi1} \rangle; \langle <\text{tLo} := \text{lo} \rangle; \\
\langle <\text{th2} := \text{hi2}; \text{RET} (\text{tLo, tH1, th2}) \rangle \\
\text{FUNC} T(l: \text{Int}, h1: \text{Int}, h2: \text{Int}) \rightarrow \text{Int} = h2 \times \text{base} + (h1 = h2 \Rightarrow l [\ast] 0)
\]

**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 \( n \text{–}1 \) times to read an \( n \text{ word} \) 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 \( \text{acq} \)


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 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 mutex 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 `reg(h)` before reading `reg(h*)`. In this case `reg(h)` acts like a lock to keep `h*`, `Sp` = `a2` 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 |

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();

PROC KernelRelease() =
% undo previous line; will retry at acq
[*] Deschedule(SELF)
FI;
ReleaseSpinLock()
```

```
PROC KernelWait(i: Int) =
% 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 `reg(h)` before reading `reg(h*)`. In this case `reg(h)` acts like a lock to keep `h*`, `Sp` = `a2` 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 |

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();

PROC KernelRelease() =
% undo previous line; will retry at acq
[*] Deschedule(SELF)
FI;
ReleaseSpinLock()
```

HANDOUT 17. Formal Concurrency
DO q # () => Schedule(q.head); q := q.tail OD;
ReleaseSpinlock()

Step: If $u$ and $F(u)$ are reachable states of $T$ and $S$ respectively, and $(u, \pi, u')$ 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 $\pi$ is an invocation or response, the fragment consists of a single $\pi$ step, with any number of internal steps before and/or after. If $\pi$ 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 $(u, \pi, u')$ 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 well known and sometimes useful. Their idea is to do an ordinary sequential proof of correctness for each thread $h$, annotating each 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

$$h.$pc $= \alpha \implies A_\alpha /\ \ h.$pc $= \beta \implies A_\beta /\ ...$$

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 as an easy introduction 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 RWLockImpl were interleaved in ForgetfulRWL. 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 $rw \geq 0$ (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:

For the code, they are:

- For a thread in Produce:
  - Before m.acq:
    - Temp := b + {t} 
  - Before b := temp
  - Before c.signal:
    - Temp := b.tail
    - b := temp
  - After m.rel:
    - RET t

- For a thread in Consume:
  - Before m.acq:
    - Temp := b + {t}
  - Before b := temp
  - Before c.signal:
    - Temp := b.tail
  - After m.rel:
    - RET t

or as annotations to the code:

```
PROC Produce(t) = VAR temp |
  m.acq;
  temp = b + {t};
  b := temp;
  c.signal;
  m.rel |
PROC Consume() -> T = VAR t, temp |
```

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 | RET t >>
```

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.$pc is in {p1, p2, p3} or is in m.acq, then in the spec h.$pc = P1.
  - If h.$pc is in {p4, P5, P6} or is in m.rel or c.signal then in the spec h.$pc = P2.
- If a thread h is in Consume in the code, then:
  - If h.$pc in {c1, c2, c3} or is in m.acq or c.wait then in the spec h.$pc = C1.
  - If h.$pc in {c4, c5, c6} or is in m.rel then in the spec h.$pc = C2.

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 h 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 p3 to a PC of p6, π simulates the singleton sequence containing just the body of Produce.
- If π takes a thread from a PC of c4 to a PC of c7, 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.pdc is in Produce and h.pdc is in \{\text{p}1, \text{p}3, \text{p}4, \text{p}5\} or is at the end of m.acq, \text{c.signal}, or \text{c.wait}, or if h.pdc is in Consume and h.pdc is in \{\text{p}2, \text{p}3, \text{p}4, \text{p}5\}\} or is at the end of m.acq, at the beginning or end of \text{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 \( \pi \) 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 \( h.pdc = p3 \) to \( h.pdc = p4 \), 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 \( h.pdc = p4 \), then \( temp = b + \{t\} \).

To prove this, we use the fact that if \( h.pdc = p4 \), then no other thread is dominant, so no other transition can change b. We also have to show that any transition that puts \( h.pdc \) 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.

```plaintext
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:
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() = << m := free >>
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 $m$ 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:

1. $s.m = u.m$, and
2. 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

- $h.p$ in ForgetfulMutex, $x.h.p$ is just before the body of acq if and only if $u.h.p$ 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.
- $h.p$ in ForgetfulMutex, $x.h.p$ is just after the body of acq if and only if $u.h.p$ is either (a) after the test-and-set with $h$'s local variable $t$ equal to free or (b) after the test with $t$ 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, \((u, \text{test-and-set, } u')\). Say this is done by thread \(u\). In this case, the value of \(m\) might change. It depends on whether the step of \(\text{SpinLock}\) changes \(m\) from \(\text{free}\) to \(\text{held}\). If it does, then we map the step to the \(\text{acq}\) 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 \(\text{SpinLock}\), \(h.\text{spec}\) moves after the test-and-set with \(h\)'s local variable \(t = \text{free}\). We claim first that the \(\text{acq}\) body in \(\text{ForgetfulMutex}\) is enabled in state \(AF(u)\). This is true because it requires only that \(x.m = \text{free}\), and this follows from the abstraction function since \(u.m = \text{free}\). Then we claim that the new states in the two modules are related by \(AF\). To see this, note that \(m = \text{held}\) in both cases. And the new PC's correspond: in \(\text{ForgetfulMutex}\), \(h.\text{spec}\) moves to right after the \(\text{acq}\) body, which corresponds to the position of \(h.\text{spec}\) in \(\text{SpinLock}\), by the definition of the abstraction function.

b) The step does not change \(m\). Then \(h.\text{spec}\) in \(\text{SpinLock}\) moves to the test, with \(t = \text{held}\). Thus, there is no change in the abstract value of \(h.\text{spec}\).

2) The test for \(t \neq \text{held}\, (u, \text{test-and-set, } u')\). Say this is done by thread \(u\). We always map this to the empty sequence of steps in \(\text{ForgetfulMutex}\). 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 \(t = \text{held}\), then the step of \(\text{SpinLock}\) moves \(h.\text{spec}\) to after the \(\text{DO}\). But this does not change the abstract value of \(h.\text{spec}\), according to the abstraction function, because both before and after the step, the abstract \(h.\text{spec}\) value is before the body of \(\text{acq}\).

b) On the other hand, if \(t = \text{free}\), then the step of \(\text{SpinLock}\) moves \(h.\text{spec}\) to after the \(\Rightarrow\). Again, this does not change the abstract value of \(h.\text{spec}\) because both before and after the step, the abstract \(h.\text{spec}\) value is after the body of \(\text{acq}\).

**SpinLock implements Mutex, second version**

Now we show again that \(\text{SpinLock}\) implements \(\text{Mutex}\), 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 \(\text{nil}\)); 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 \(\text{free}\) or \(\text{held}\));
- for each thread that is active in the module, a program counter; and
- for each thread that is active in \(\text{acq}\), 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:
  
  \[
  \begin{align*}
  \text{ms} & : (\text{Thread} + \text{Null}) := \text{nil} & \% \text{ m in the Spec} \\
  \text{hs} & : \text{Bool} := \text{false} & \% \text{ havoc in the Spec}
  \end{align*}
  \]

  - 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:

    \[
    \begin{align*}
    \text{PROC acq()} & = \text{VAR } t : \text{FH} | \\
    & \text{ DO } <t := m; m := \text{held}>>; \text{ IF } t \neq \text{held} \Rightarrow <\text{ms} := \text{SELF}>> \text{ RET } [*] \text{ SKIP PI OD} \\
    \text{PROC rel()} & = << m := \text{free}; \text{ hs := hs} \lor (\text{ms} \neq \text{SELF}); \text{ ms := nil} >>
    \end{align*}
    \]

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

This treatment of \(\text{ms}\) 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 \(\text{ms}\) 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 \(\text{acq}\)
- \(A_2\) — after the body of \(\text{acq}\)
- \(R_1\) — before the body of \(\text{rel}\)
- \(R_2\) — after the body of \(\text{rel}\)

For the code, they are:

- For a thread in \(\text{acq}\):
  
  \[
  \begin{align*}
  a_1 & — \text{ before the } \text{VAR } t \\
  a_2 & — \text{ after the } \text{VAR } t \text{ and before the } \text{DO} \text{ loop} \\
  a_3 & — \text{ before the test-and-set in the body of the } \text{DO} \text{ loop} \\
  a_4 & — \text{ after the test-and-set in the body of the } \text{DO} \text{ loop} \\
  a_5 & — \text{ before the assignment to } \text{ms} \\
  a_6 & — \text{ after the assignment to } \text{ms}
  \end{align*}
  \]

- For a thread in \(\text{rel}\):
  
  \[
  \begin{align*}
  r_1 & — \text{ before the body} \\
  r_2 & — \text{ after the body}
  \end{align*}
  \]
The transitions in \texttt{acq} may be a little confusing: there's a transition from \texttt{a}_2 to \texttt{a}_4, as well as transitions from \texttt{a}_3 to \texttt{a}_5.

Here are the routines in \texttt{Mutex} annotated with the PC values:

\begin{verbatim}
APROC acq() = \{A_1\} << m = nil => m := SELF >> \{A_2\}
APROC rel() = \{R_1\} \{m \# SELF \Rightarrow \texttt{HAVOC} \} m := nil >> \{R_2\}
\end{verbatim}

Here are the routines in \texttt{SpinLock} annotated with the PC values:

\begin{verbatim}
APROC acq() = \{a_1\} \{m = \texttt{FH} \mid \{a_2\} \{t := \texttt{FH} \mid \{a_3\} \{t := m \mid \{a_1\} \}\mid \{a_4\} \}\mid \{a_5\} \}\mid \{a_6\} \}
\{a_7\} \{t := \texttt{ms} \Rightarrow \texttt{HAVOC} \} \{a_8\} \}\mid \{a_9\} \}\mid \{a_{10}\} \}
\end{verbatim}

\begin{verbatim}
APROC rel() = \{r_1\} \{m := \texttt{free} \mid \{r_2\} \{ms := \texttt{ms} \Rightarrow \texttt{HAVOC} \} \{r_3\} \}\mid \{r_4\} \}\mid \{r_5\} \}
\end{verbatim}

Now we can define the mappings on program counters:

- If a thread is not in \texttt{acq} or \texttt{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 \texttt{A}_1, \texttt{a}_6 maps to \texttt{A}_2
- \texttt{r}_1 maps to \texttt{R}_1, \texttt{r}_2 maps to \texttt{R}_2

The part of the abstraction function dealing with the global variables of the module simply defines \texttt{ms} in the spec to have the value of \texttt{ms} in the code, and \texttt{HAVOC} in the spec to have the value of \texttt{ms} 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 \( \pi \) is an external transition, \( \pi \) simulates the singleton sequence containing just \( \pi \).
- If \( \pi \) takes a thread from \texttt{a}_2 to \texttt{a}_4, \( \pi \) simulates the singleton sequence containing just the transition from \texttt{A}_1 to \texttt{A}_2.
- If \( \pi \) takes a thread from \texttt{r}_1 to \texttt{r}_2, \( \pi \) simulates the singleton sequence containing just the transition from \texttt{R}_1 to \texttt{R}_2.
- All other transitions simulate the empty sequence.

\textbf{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 \texttt{ms} in the code is \texttt{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 \texttt{rel} is also straightforward. The hard cases are in the body of \texttt{acq}.

Consider all the transitions in \texttt{acq} before the one from \texttt{a}_2 to \texttt{a}_4. These all simulate the null transition, so they should leave the abstract state unchanged. And they do, because none of them changes \texttt{ms}.

The transition from \texttt{a}_2 to \texttt{a}_5 simulates the transition from \texttt{A}_1 to \texttt{A}_2. There are two cases: when \texttt{ms} = \texttt{nil}, and when \texttt{ms} \neq \texttt{nil}.

1. In the first case, the transition from \texttt{A}_1 to \texttt{A}_2 is enabled and, when taken, changes the state so that \texttt{ms} = \texttt{SELF}. This is just what the transition from \texttt{a}_2 to \texttt{a}_5 does.

2. Now consider the case when \texttt{ms} \neq \texttt{nil}. We claim this case is possible only if a thread that didn't hold the mutex has done a \texttt{rel}. Then \texttt{hs} = \texttt{true}, the spec has done \texttt{HAVOC}, and anything can happen. In the absence of \texttt{HAVOC}, if a thread is at \texttt{a}_5, then \texttt{ms} = \texttt{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:

\begin{itemize}
  \item \texttt{ms} is true.
\end{itemize}

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 \texttt{a}_5 or \texttt{a}_2 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 \texttt{rel} preserves the first conjunct of the invariant because afterwards both \texttt{m} = \texttt{free} and \texttt{ms} = \texttt{nil}. To prove that the transition in \texttt{rel} preserves the second conjunct of the invariant, there are two cases, depending on whether the spec allows \texttt{HAVOC}.

1. If it does, then the code sets \texttt{hs} = true; this corresponds to the \texttt{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 \texttt{HAVOC} is that the rest of the invariant may not hold after a rogue thread does \texttt{rel}. Because the rogue thread resets \texttt{ms} to \texttt{free}, if there's a thread at \texttt{a}_6 or at \texttt{a}_7 with \texttt{t} = \texttt{free}, and \texttt{m} = \texttt{held}, then after the rogue \texttt{rel}, \texttt{ms} is no longer held and hence the second conjunct is false. This means that it's possible for several threads to get to \texttt{a}_5, or to \texttt{a}_4 with \texttt{t} = \texttt{free}. The invariant still holds, because \texttt{hs} is now true.
2. In the normal case \(ms \neq \text{nil}\), and since we’re assuming the invariant is true before the transition, this implies that no thread is at \(a_t\) with \(t = \text{free}\) or at \(a_t\). After the transition to \(r_2\) it’s still the case that no thread is at \(a_t\) with \(t = \text{free}\) or at \(a_t\), so the invariant is still true.

Now we consider the transitions in \(acq\). The transitions from \(a_0\) to \(a_1\) and from \(a_2\) to \(a_3\) obviously preserve the invariant. The transition from \(a_1\) to \(a_2\) puts a thread at \(a_0\), but \(t = \text{free}\) in this case so the invariant is true after the transition by induction. The transition from \(a_3\) to \(a_4\) also clearly preserves the invariant.

The transition from \(a_0\) to \(a_1\) is the first interesting one. We need only consider the case \(hs = \text{false}\). 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 \(\text{held}\). 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 \(a_0\), or at \(a_4\) with \(t = \text{free}\). Then we have \(m = \text{held}\) by induction, so after the transition both \(t = \text{held}\) and \(m = \text{held}\). This preserves the invariant.

2. Before the transition, there are no threads at \(a_0\) or at \(a_4\) with \(t = \text{free}\). Then after the transition, there is still no thread at \(a_0\), but there is a new thread at \(a_t\). (Any others must have \(t = \text{held}\).) Now, if this thread has \(t = \text{held}\), the second part of the invariant is true vacuously; but if \(t = \text{free}\), then we have:

\[
\begin{align*}
ms &= \text{nil} \quad \text{(since when the thread was at } a_t \text{ it must have been } \text{free}, \text{ hence the first part of the invariant applies)}; \\
m &= \text{held} \quad \text{(as a direct result of the transition)}; \\
\text{there are no threads at } a_t \quad \text{(by assumption)}; \quad \text{and} \\
\text{there are no other threads at } a_0 \quad \text{(by assumption)}.
\end{align*}
\]

So the invariant is still true after the transition.

Finally, assume a thread \(h\) is at \(a_4\) about to transition to \(a_5\). If the invariant is true here, then \(h\) is the only thread at \(a_t\), and all threads at \(a_4\) have \(t = \text{held}\). 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 \(a_5\), \(ms = \text{nil}\), which is what we wanted to show.

Simplifying the proof

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 \(a_0\) to \(a_2\) to simulate the body of \(\text{Mutex}.acq\). We omit the \(hs\) history variable and augment the code as follows:

```plaintext
PROC acq() = [\{a_0\}] VAR t := FH |
[\{a_1\}] DO [\{a_1\}] << t := m; m := held; [\{t \# held \} => ms := SELF [*] SKIP FI] >> |
[\{a_2\}] IF t \# held => [\{a_2\}] RET [\{a_2\}] [*] SKIP FI OD;
PROC rel() = [\{r_1\}] << m := free; ms := nil >> [\{r_1\}]
```

The abstraction function maps \(ms\) to \(\text{Mutex}.m\) as before, and it maps PC’s \(a_2\) to \(A_1\) and \(a_4\) to \(A_2\). It maps \(a_0\) to \(A_1\) if \(t = \text{held}\), and to \(A_2\) if \(t = \text{free}\); thus \(a_0\) simulates \(\text{Mutex}.acq\) only if \(m\) was \text{free}, as we should expect. There is no need for an invariant; we only used it at \(a_0\) to \(a_0\), which no longer exists.

The simulation argument is the same as before except for \(a_0\) to \(a_2\), which is the only place where we changed the code. If \(m = \text{held}\), then \(m = \text{held}\), and \(ms\) don’t change; hence \(\text{Mutex}.m\) doesn’t change, and neither does the abstract PC; in this case the transition simulates the empty trace. If \(m = \text{free}\), then \(m = \text{free}\), \(ms\) becomes \(\text{SELF}\), 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 \(\text{Mutex}.acq\), that is, how to abstract the PC. It seems very natural to change \(ms\) and neither does the abstract PC; in this case the transition simulates the empty trace. If \(m = \text{free}\), then \(m = \text{free}\), \(ms\) becomes \(\text{SELF}\), and the abstract PC becomes \(A_2\); in this case the transition simulates \(A_1\) to \(A_2\), as promised.

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. We begin with a proof in the style of the last few, and then give an entirely different proof based on model checking.

First we show that the simple, deadlocking version \(\text{acq}\) maintains mutual exclusion. Recall that we write \(h^+\) for the thread that is the partner of thread \(h\). Here are the spec and code again:

```plaintext
CLASS Mutex EXPORT acq, rel =
VAR m : (Thread + Null) := nil
PROC acq() = [\{m \neq \text{nil} \} => m := SELF; RET >>
PROC rel() = [\{m = \text{SELF} \} => m := nil; RET [*] HAVOC >>
END Mutex
```
CLASS Mutex2Impl0 EXPORT acq, rel =

VAR
req : Thread -> Bool := (* -> false)
lastReq : Int
PROC acq0() =
  [a] req(SELF) := true;
  DO [a] req(SELF*) => SKIP OD [a]
PROC rel() = req(SELF) := false
END Mutex2Impl0

Intuitively, we get mutual exclusion because once req(h) is true, h* can’t get from a2 to a3. It’s convenient to define

FUNC Holds0(h: Thread) = req(h) /
  lastReq := SELF

Abstractly, h has the mutex if Holds0(h), and the transition from a2 to a3 simulates the body of Mutex.acq. Precisely, the abstraction function is

Mutex.m = (Holds0.set = {h | lastReq = h} 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 a3, 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 (a2, a3) can increase the size of Holds0.set, and h can take that step only if req(h*) = false, so Holds0.set goes from (h) to (h). The second conjunct holds because it can never be true => false, since only the step (a2, req(h) = true, a2) can make the antecedent true, and no step away from a2 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 a2 to a3 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] << t := req(SELF*) >>; [a] << t => SKIP >> OD [a]

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 = a3 to Holds0, and extend the mutual exclusion conjunct of the invariant so that it covers a thread that has reached a21 with t = false:

(Holds0.set /
  h.$pc = a2 /
  h.t = false).size <= 1

Or we could get the same effect by saying that a thread acquires the lock by reaching a21 with t = false, so that it’s the transition (a2, a21) with t = false that simulates the body of Mutex.acq, rather than the transition to a3 as before. This means changing the definition of Holds0 to

FUNC Holds0(h: Thread) =
  req(h) /
  h.$pc = a2 /
  (h.$pc = a2 => h.t = false)

Yet another approach is to make explicit 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 a2 or a21, in other words, Holds(h*) = false. We can put this into the invariant with the conjunct

h.$pc = a21 /
  h.t = false => Holds(h*) = false

and this is enough to ensure that the transition (a21, a3) maintains the invariant.

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

PROC acq() =
  [a] req(SELF) := true;
  [a] lastReq := self;
  DO [a] (req(SELF*) /
  lastReq = SELF) => SKIP OD [a]

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.$pc = a2 /
  (h.$pc = a2 => req(h))

We also discussed liveness formally 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.$pc = a2 /
  (h.$pc = a2 => lastReq = SELF)

and add /
  h.$pc = a3 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 /
  (Holds(h*) /
  h.$pc = a2 /
  (h.$pc = a2 => lastReq = h*)

The last conjunct holds because (a1, a2) makes it true, and the only way to make it false is for h* to do lastReq := SELF, which it can only do from a3, so that Holds(h*) is false. With this invariant it’s obvious that (a2, a3) maintains the invariant.

Proof by model checking

We have been doing all our proofs by establishing invariants; these are called assertional proofs. An alternative method is to explore the state space exhaustively; this is called model checking. It only works when the state space not too big. In this case, if the two threads are a and b, the state space is just:

a.$pc IN (a2, a1, a2, a3)
b.$pc IN (a2, a1, a2, a3)
req(a) IN (false, true)
req(b) IN {false, true}
lastReq IN {a, b}

We can write down a state concisely with one digit to represent each PC, a \texttt{t} or \texttt{f} for each req, and an a or b for lastReq. Thus \texttt{00ffa} is $a.pc = a$, $b.pc = b$, req(a) = false, req(b) = false, lastReq = a. When the value of a component is unimportant we write $x$ for it.

Now we can display the complete state machine.

Note the extensive symmetries. Nominally there are 128 states, but many are not reachable:

1. The value of req follows from the PC's, which cuts the number of reachable states to 32.
2. $33xxx$ is not reachable. This is the \textit{mutual exclusion invariant}, which is that both PC's cannot be in the critical section at the end of \texttt{acq}. This removes 2 states.
3. At the top of the picture the value of lastReq is not important, so we have shown it as $x$. This removes 4 states.
4. We can't have $20xxb$ or $21xxb$ or $30xxb$ or $31xxb$ or $32xxa$, or the 5 symmetric states, because of the way lastReq is set. This removes 10 states.

In the end there are only 16 reachable states, and 7 of them are obtained from the others simply by exchanging the two threads a and b.

Since there is no non-determinism in this algorithm and a thread is never blocked from making a transition, there are two transitions from each state, one for each thread. If there were no transitions from a state, the system would deadlock in that state. It's easy to see that the algorithm is live if both threads are scheduled fairly, since there are no non-trivial cycles that don't reach the end of \texttt{acq}. It is fair because the transitions from $00ffa$ and $11ttx$ are fair.

The appeal of model-checking should be clear from the example: we don't have to think, but can just search the state space mechanically. The drawback is that the space may be too large. This small example illustrates that symmetries can cut the size of the search dramatically, but the symmetries are often not obvious.

**ClockImpl implements Clock**

We conclude with the proof of the clock implementation. The spec says that a \texttt{Read} returns some value that the clock had between the beginning and the end of the \texttt{Read}. Here it is, with labels.

**MODULE Clock**

```
MODULE Clock
EXPORT Read =
VAR t : Int := 0 % the current time
THREAD Tick() = DO << t + := 1 >> OD % demon thread advances t
PROC Read() -> Int = VAR t1: Int | [
R1] << t1 := t >>; [
R2] % I2: T(lo, tH1, hi2) IN t1Hist .. T(lo, tH1, hi2) << VAR tLo, tH1, tH2, t1Hist: Int | [
R3] % I3: T(tLo, tH1, tH2) IN t1Hist .. T(lo, tH1, hi2) << tLo := lo; >>[
R4] % I4: $a IN t1Hist .. T(lo, tH1, hi2) << tH2 := hi2; RET T(tLo, tH1, tH2) >>[
R5] ] END Clock
```

To show that \texttt{ClockImpl} implements this we introduce a history variable \texttt{t1Hist} in \texttt{Read} that corresponds to \texttt{t} in the spec, recording the time at the beginning of \texttt{Read}'s execution. The invariant that is needed is based on the idea that \texttt{Read} might complete before the next \texttt{Tick}, and therefore the value \texttt{Read} would return by reading the rest of the shared variables must be between \texttt{t1Hist} and \texttt{Clock.t}.

**MODULE ClockImpl**

```
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 $R_1 \leftrightarrow r_1$, $R_2 \leftrightarrow r_2$, $R_3 \leftrightarrow r_3$
THREAD Tick() = DO VAR newLo: Word, newHi: Word | << newLo := lo + 1 // base; newHi := hi1 + 1 >>; IF << newLo # 0 => lo := newLo >> [*] << hi2 := newHi >>; FI OD
PROC Read() -> Int = VAR tLo, tH1, tH2, t1Hist: Int | [
R1] << tH1 := hi1; t1Hist := T(lo, tH1, hi2) >>; [
R2] % I2: T(lo, tH1, hi2) IN t1Hist .. T(lo, tH1, hi2) << tLo := lo; >> [
R3] % I3: T(tLo, tH1, hi2) IN t1Hist .. T(lo, tH1, hi2) << tH2 := hi2; RET T(tLo, tH1, tH2) >> [
R4] % I4: $a IN t1Hist .. T(lo, tH1, hi2) END Clock
```

The appeal of model-checking should be clear from the example: we don't have to think, but can just search the state space mechanically. The drawback is that the space may be too large. This small example illustrates that symmetries can cut the size of the search dramatically, but the symmetries are often not obvious.
The whole invariant is thus

\[ h.pc = r_2 \implies I_2 \land h.pc = r_3 \implies I_3 \land h.pc = r_4 \implies I_4 \]

The steps of \texttt{Read} clearly maintain this invariant, since they don’t change the value before \texttt{IN}. The steps of \texttt{Tick} maintain it by case analysis.