17. Formal Concurrency

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 name and the argument values. 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 \texttt{threads} (also called ‘tasks’ or ‘processes’), each of which executes a sequence of atomic actions; we denote threads by variables \( h, h', \ldots \). To define its sequence, each thread has a state \( \alpha \) and each of its actions has the form \( (h.\text{pc} = \alpha) \rightarrow c \), so that \( c \) can only execute when \( h \)'s program counter equals \( \alpha \). Different actions have different values for \( \alpha \), 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 \( h.\text{pc} := \beta \), 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.

\footnote{1 The variables declared by a program are not allowed to have labels as their values, hence there is no \texttt{Label} type.}

\footnote{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.}

\footnote{3 This works for non-recursive procedures. To accommodate recursive procedures, the state must involve something equivalent to a stack. Probably the simplest solution is to augment the state space by tacking on the nesting depth of the procedure to all the names and program counter values defined above. For example, \( h + "P.v" + d.enc \) for every positive integer \( d \). An invocation transition at depth \( d \) goes to depth \( d+1 \).}
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 unstruck later, because of the transitions of some other threads. Thus, a command failing does not necessarily (or even usually) mean that the thread fails.

We won’t give the non-atomic semantics precisely here as we did for the atomic semantics in handout 9, since it involves a number of fussy details that don’t add much insight. Also, it’s somewhat arbitrary. You can always get exactly the atomicity you want by adding local variables and semicolons to increase the number of atomic transitions (see the examples below), or <<...>> brackets to decrease it.

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

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

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

Suppose the label before the command $c$ is $a$ and the one after the command is $b$, and the transition function defined by MC($c$) in handout 9 is $\langle s, a | o \rangle$. Then if $c$ is in thread $h$, its transition function is

$\langle s, o | s(h â¨ .Spc) = a \backslash o(hâ¨.Sp) = b \backslash o \rangle$

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

$\langle s, o | (EXISTS h: Thread | s(hâ¨.P.$Sp) = a \backslash o(hâ¨.$P.$Sp) = b \backslash o \rangle)$

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

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

\[
\begin{align*}
\text{[a1]} & \text{ C1;} & \text{[a2]} & \text{ C2;} & \text{[a3]} & \text{ C3} & \text{[a3]} \\
\text{pc} = \alpha_1 & \text{ => } \text{[C1]} & \text{pc} = \alpha_2 & \text{ => } \text{[C2]} & \text{pc} = \alpha_3 & \text{ => } \text{[C3]}
\end{align*}
\]

As you can see, $[\alpha]$ $C_1$; $[\beta]$ translates to $pc = \alpha$ => $[C_1]$; $pc = \beta$. If there’s a non-deterministic choice, that shows up as a choice in one of the actions of the translation:

\[
\begin{align*}
\text{[a1]} & \text{ IF } \text{C1;} & \text{[a2]} & \text{ C1;} & \text{[a3]} & \text{ C1;} & \text{[a3]} \\
\text{pc} = \alpha_1 & \text{ => IF } \text{[C1]} & \text{pc} = \alpha_2 & \text{ => } \text{[C1]} & \text{pc} = \alpha_3 & \text{ => } \text{[C1]}
\end{align*}
\]

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

The second example is a simple real Spec program. The next section explains the detailed rules for where the labels go in Spec; note that an assignment is two atomic actions, one to evaluate

\[
\begin{align*}
\text{[a1]} & \text{ DO } i < n => \text{[a2]} & \text{ sum := [a3] sum + x(i);} & \text{[a4]} & \text{ if } \text{pc} = \alpha_2 & \text{ => } \text{[a3]} & \text{pc} = \alpha_3 & \text{ => } \text{[C1]} & \text{pc} = \alpha_4 & \text{ => } \text{[C2]} & \text{pc} = \alpha_5 & \text{ => } \text{[C3]} \\
\text{pc} = \alpha_6 & \text{ => } \text{[C4]}
\end{align*}
\]

The expression on the right and the other to change the variable on the left. The extra local variable $t$ is a place to store the value between these two actions.

\[
\begin{align*}
\text{[a1]} & \text{ i := [a3] i + 1} & \text{[a4]} & \text{ if } \text{pc} = \alpha_4 & \text{ => } \text{[a3]} & \text{pc} = \alpha_5 & \text{ => } \text{[C1]} & \text{pc} = \alpha_6 & \text{ => } \text{[C2]}
\end{align*}
\]

Labels in Spec

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

\[
\begin{align*}
\text{VAR t | [a1]} & \text{ if } t < n => \text{[a2]} & \text{ sum := [a3] sum + x(i);} & \text{[a4]} & \text{ if } t < i + 1 => \text{[a5]} & \text{ i := 1} & \text{[a6]} & \text{ pc := a6 => [C4]}
\end{align*}
\]

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

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

For an invocation, there is a transition to evaluate the argument and set the $sa$ variable, and one to send control to the start of the body. The RET command’s transition sets $sa$ 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 $sa$, send control to the body, do the RET and set $sa$, 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 $x := a + b + c$ translates to

\[
\begin{align*}
\text{VAR t | [a1]} & \text{ if } t < a => \text{[a2]} & \text{ t := a;} & \text{[a3]} & \text{ if } t < t + b => \text{[a4]} & \text{ t := t + b} & \text{[a5]} & \text{ if } x < t + c => \text{[a6]} & \text{ x := t + c}
\end{align*}
\]

Handout 17. Formal Concurrency
when the program counter is the label on the $\text{DO}$, and the last transition sets the program counter back to that label. When $\varepsilon$ fails, the program counter is set to the label following the $\text{DO}$.

To sum up, there’s a label on each $\text{:=}, \text{THEN}, \text{EXCEPT}$, and $\text{DO}$ outside of $\langle\ldots\rangle$. 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 $\text{=}$ 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 $\text{	extquoteleft}\text{	extquoteleft}$. After the invocation is complete, the PC goes to the next label after the invocation, which is the one on the $\text{:=}$ if the invocation is 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.

\[
\begin{align*}
\text{PROC } P(\cdot) &= \{P_1\} \text{ VAR } x, y | \\
&\quad \text{IF } x > 5 \Rightarrow \{P_2\} x := [P_4] Q(x + 1, 2 \cdot [P_3]); \ [P_5] y := [P_6] 3 \\
&\quad \{P_7\} \text{ FI; } \\
&\quad \text{VAR } z | \text{ DO } [P_8] \langle P() \rangle \text{ OD } [P_9]
\end{align*}
\]

### 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 $\text{PROC}$ 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 $\text{RET}$ command in the procedure to just after the invocation in the caller. These external transitions do not affect the $\Sigma$ variable that communicates the argument and result values. That variable is set by the internal transitions that compute the argument and do the $\text{RET}$ command.

There’s another style of defining external interfaces in which every external action is an $\text{APROC}$. If you want to get the effect of a non-atomic procedure, you have to break it into two $\text{APROC}$’s, one that delivers the arguments and sets the internal state so that internal actions will do the work of the procedure body, and a second that retrieves the result. This style is used in I/O automata\(^4\), but we will not use it.

\[^{4}\text{Nancy Lynch, } \textit{Distributed Algorithms}, \text{Morgan Kaufmann, 1996, Chapter 8.}\]

### Examples

Here are two Spec programs that search for prime numbers and collect the result in a set $\text{primes}$; both omit the even numbers, initializing $\text{primes}$ to $\{2\}$. Both are based on the sieve of Eratosthenes, testing each prime less than $n^{1/2}$ to see whether it divides $n$. Since the threads may not be synchronized, we must ensure that all the numbers $\leq n^{1/2}$ 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.

\[
\begin{align*}
\text{CONST } \text{Odds} &= \{i: \text{Nat} \mid i // 2 = 1 \land i > 1\} \\
\text{VAR } \text{primes} : \text{SET Nat} := \{2\} \\
\text{done} : \text{SET Nat} := \{\} \\
\text{INVARIANT} \ (\forall n: \text{Nat} \mid n \IN \text{done} \land \text{IsPrime}(n) \Rightarrow n \IN \text{primes} \\
&\quad \land \text{IsPrime}(n))
\end{align*}
\]

\[
\begin{align*}
\text{THREAD } \text{Sieve1}(n: \IN \text{Odds}) = \\
&\quad \{i: \IN \text{Odds} \mid i =< \text{Sqrt}(n)\} \langle \text{done} =\Rightarrow \rangle \% \text{Wait for possible factors} \\
&\quad \{\text{IF} (\forall p: \IN \text{primes} \mid p =< \text{Sqrt}(n) \Rightarrow n = p \# 0) =\Rightarrow \}
\end{align*}
\]

\[
\begin{align*}
&\quad \{\langle \text{primes} \langle \\
&\quad \langle \langle \langle \text{done} \\langle \langle \langle \text{Sqrt}(n) \Rightarrow \text{Int} = \text{RET} \{i: \text{Nat} \mid i =< n\} \max \}
\end{align*}
\]

\[
\begin{align*}
\text{FUNC } \text{Sqrt}(n: \text{Nat}) \Rightarrow \text{Int} &= \text{RET} \{i: \text{Nat} \mid i\cdot i =< n\}.\max
\end{align*}
\]

The second example, on the other hand, is closer to code, running ten parallel searches. Although there is one thread for every integer, only threads $\text{Sieve0}$, $\text{Sieve1}$, ..., $\text{Sieve9}$ are “active”, because of the initial guard, Differences from $\text{Sieve1}$ are boxed.

\[
\begin{align*}
\text{CONST } \text{nThreads} &= 10 \\
\text{VAR } \text{primes} : \text{SET Int} := \{2\} \\
\text{next} &:= \text{nThreads}.\seq
\end{align*}
\]

\[
\begin{align*}
\text{THREAD } \text{Sieve}(i: \text{Int}) = \text{next}(!i) = \\
&\quad \text{next}(!i) := 2*i + 3; \\
&\quad \text{DO } \text{VAR } n: \text{Int} := \text{next}(!i) | \\
&\quad \{\text{ALL } j: \IN \text{next}.\rng \mid j =\langle \text{Sqrt}(n)\} =\Rightarrow \}
\end{align*}
\]

\[
\begin{align*}
&\quad \{\text{IF} (\forall p: \IN \text{primes} \mid p =\langle \text{Sqrt}(n) \Rightarrow n = p \# 0) =\Rightarrow \}
\end{align*}
\]

\[
\begin{align*}
&\quad \{\langle \text{primes} \langle \\
&\quad \langle \langle \langle \text{Sqrt}(n) \Rightarrow \text{SKIP} \}
\end{align*}
\]

\[
\begin{align*}
\text{OD} \\
&\quad \text{next}(!i) := n + 2*\text{nThreads}
\end{align*}
\]

### 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 $\langle\ldots\rangle$ brackets.\(^5\) It doesn’t tell you where to write the brackets. If the environment in which the program has to run doesn’t impose any

\[^{5}\text{As we have seen, Spec does treat expression evaluation as atomic. Recall that if you are dealing with an environment in which an expression like } x(i) + f(y) \text{ can’t be evaluated atomically, you should model this by writing } \text{VAR } t1, t2 | t1 := x(i); t2 := f(y); \ldots t1 + t2 \ldots.\]

Handout 17. Formal Concurrency
constraints, it’s usually best to make the atomic actions as big as possible, because big atomic actions are easier to reason about. But big atomic actions are often too hard or too expensive to code, or the reduction in concurrency hurts performance too much, so that we have to make do with small ones. For example, in a shared-memory multiprocessor typically only the individual instructions are atomic, and we can only write one disk block atomically. So we are faced with the problem of showing that code with small atomic actions satisfies a spec with bigger ones.

**The idea**

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

\[
A; B
\]

(1)

as well as an arbitrary collection of other commands. We denote the program counter value before \( \alpha \) by \( \beta \) and at the semi-colon by \( \gamma \). We are thinking of the program as

\[
h.\$pc = \alpha \Rightarrow A [ ] h.\$pc = \beta \Rightarrow B [ ] C_1 [ ] C_2 [ ] \ldots
\]

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

Suppose \( B \) denotes an arbitrary atomic command, and \( A \) denotes an atomic command that commutes with every command in the program (other than \( h \)) that is enabled when \( h \) is at the semicolon, that is, when \( h.\$pc = \beta \). (We give a precise meaning for ‘commutes’ below.) In addition, both \( A \) and \( B \) have only internal actions. Then it’s intuitively clear that the program with

\[
\text{\textquotesingle\textquotesingle} A; B \text{\textquotesingle\textquotesingle}
\]

(1) simulates a program with the same commands except that instead of (1) it has

\[
\text{\textquotesingle\textquotesingle} A; B \text{\textquotesingle\textquotesingle}
\]

(2)

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

Thus we have achieved the goal of making a bigger atomic command \( \text{\textquotesingle\textquotesingle} A; B \text{\textquotesingle\textquotesingle} \) out of two small ones \( A \) and \( B \). We can call the big command \( B \) and repeat the process on \( E; D \) to get a still bigger command \( \text{\textquotesingle\textquotesingle} E; A; B \text{\textquotesingle\textquotesingle} \).

How do we ensure that only one command \( C \) that commutes with \( A \) can execute while \( h.\$pc = \beta \)?

The simplest way is to ensure that the variables that \( A \) touches (reads or writes) are disjoint from the variables that \( C \) writes, and vice versa; then they will surely commute. Two such commands are called ‘non-interfering’. There are two easy ways to show that commands are non-interfering. One is that \( A \) touches only local variables of \( h \). Only actions of \( h \) that touch local variables of \( h \), and the only action of \( h \) that is enabled when \( h.\$pc = \beta \) is \( B \). So any sequence of commands that touch only local variables is atomic, and if it is preceded or followed by a single arbitrary atomic command the whole thing is still atomic.\(^6\)

The other easy case is a critical section protected by a mutex. Recall that a critical section for \( v \) is a command with the property that if some thread’s \( \text{PC} \) 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 \( v \) and commands that don’t touch \( v \) can execute while \( h.\$pc = \beta \). So even 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):

\[
\text{\textquotesingle\textquotesingle} A; B \text{\textquotesingle\textquotesingle}
\]

(1)

\[
\text{\textquotesingle\textquotesingle} C \text{\textquotesingle\textquotesingle}
\]

(2)

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>Possible sequence?</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.acq(h)</td>
<td>m.acq(h')</td>
<td>No: ( C ) is blocked by ( h ) holding ( m )</td>
</tr>
<tr>
<td>m.acq(h)</td>
<td>m.rel(h')</td>
<td>No: ( C ) won’t be reached because ( h' ) doesn’t hold ( m )</td>
</tr>
<tr>
<td>m.rel(h)</td>
<td>m.acq(h')</td>
<td>OK</td>
</tr>
<tr>
<td>m.rel(h)</td>
<td>m.rel(h')</td>
<td>No: one thread doesn’t hold ( m ), hence won’t do rel</td>
</tr>
</tbody>
</table>

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

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

A third important case of commuting operations, producer-consumer, is 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 \( \text{get} \), and 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 \( m.\text{acq}(h') \) is blocked until \( m.\text{rel}(h) \) 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).

\[
\text{\textquotesingle\textquotesingle} A; B \text{\textquotesingle\textquotesingle}
\]

(1)

\[
\text{\textquotesingle\textquotesingle} A; B \text{\textquotesingle\textquotesingle}
\]

(2)

As usual, we call the complete program containing (2) the spec \( S \) and the complete program containing (1) the code \( T \). We need an abstraction relation \( \lambda \) between the states of \( T \) and the

\[\text{Handout 17. Formal Concurrency}\]

---


\(^7\) 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.
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 $h$.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 $t$.

First we need a precise definition of "c is enabled at $\beta$ and commutes with $A$". For any command $X$, we write $u \mathbin{X} u'$ for $\mathsf{MC}(X)(u, u')$, that is, if $X$ relates $u$ to $u'$. The idea of 'commutes' is that $\mathsf{AR}$ relates $u$ to $u'$, because only equal states are related when $h$.Spc # $\beta$. We use $u$ and $u'$, for states of $S$ and $T$, to avoid confusion with various other uses of $t$.

Example 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 $\text{Read}$ and $\text{Write}$ operations. Here is the unsurprising spec of the register, which makes both operations atomic:

\begin{verbatim}
MODULE Register
EXPORT Read, Write =
VAR x : Int := 0
APROC Read() -> Int = << RET x >>
APROC Write(i: Int) = << x := i >>
\end{verbatim}
To increment the register, we could use the following procedure:

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

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.

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

Exercise: 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.

```plaintext
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.

    APROC acq() = << m = nil => m := SELF; RET >>
    APROC rel() = << m = SELF => 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 spin lock.

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

    PROC acq() =
        DO << fh = held => SKIP >> OD;
        IF fh = free => RET[*] HAVOC
    PROC rel() = << fh := free >>
END BadSpinLock
```

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

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

A thread acquires the lock when it is the one that changes it from `free` to `held`, which it checks by testing the value returned by the test-and-set.

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

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

Of course 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.

Wait-free primitives

It’s also possible to implement spin locks with a wait-free primitive rather than with test-and-set, although this rather misses the point of wait-free synchronization, which is discussed informally in handout 14.

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

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

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

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

Now we can write acq:

PROC acq() = VAR fh’, OK: Bool |
DO fh’ := LL(); IF fh’ = free => OK := SC(held); IF OK => RET [*] SKIP FI [*] SKIP FI
OD

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

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

or with LL/SC:

VAR fh := free
VAR x : Any
PROC acq() = VAR t := CAS(free, held); IF t => RET [*] SKIP FI OD
PROC SC(new: Any) -> BOOL = << IF lock => x := new; RET true [*] RET false >>

More generally, you can update an arbitrary data structure with an arbitrary function f by replacing i+1 in the CAS implementation with f(i). The way to think about this is that f computes a new version of i, and you install it if the version hasn’t changed since you started. This is a form of optimistic concurrency control; see handout 20 for a more general discussion of this subject. Like optimistic concurrency control in general, the approach runs the danger of doing a lot of work that you then have to discard, or of starving some of the threads because they never get done before other threads sneak in and change the version out from under them. There are clever tricks for minimizing this danger; the basic idea is to queue your f for some other thread to execute along with its own.

Read/write locks

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

CLASS RWLock
EXPORT rAcq, rRel, wAcq, wRel =

TYPE ST = SET Thread
VAR r : ST := {}
VAR w : ST := {}
VAR lock : Bool := false
APROC rAcq() = << SELF IN (r / w) => HAVOC [*] w = {} => r / := {SELF} >>
APROC wAcq() = << SELF IN (r / w) => HAVOC [*] (r / w) = {} => w := {SELF} >>
APROC rRel() = << ~ (SELF IN r) => HAVOC [*] r := {} >>
APROC wRel() = << ~ (SELF IN w) => HAVOC [*] w := {} >>
END RWLock

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

CLASS ForgetfulRWL
EXPORT rAcq, rRel, wAcq, wRel =

VAR rw := 0 % >0 gives number of readers, 0 means free, -1 means one writer
APROC rAcq() = << rw >= 0 => rw + := 1 >>
APROC wAcq() = << rw = 0 => rw := -1 >>
APROC rRel() = << rw - := 1 >>
APROC wRel() = << rw := 0 >>
END ForgetfulRWL

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

Condition variables

Mutexes are used to protect shared variables. Often a thread 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, has to release the lock. It is not efficient to repeatedly release the lock and then re-acquire it to check the condition. Instead, it’s better for h to wait on a condition variable, as we saw in handout 14. Whenever any thread changes the shared variables in such a way that the condition might become true, it signals the threads waiting on that variable. Sometimes we say that the waiting threads ‘wake up’ when they are signaled. Depending on the application, a thread may signal one or several of the waiting threads.

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

CLASS Condition
EXPORT wait, signal, broadcast =

TYPE M = Mutex
VAR c : SET Thread := {}% Each condition variable is the set of waiting threads.
PROC wait(m) =
<< c / := {SELF}; m.rel >>;% m.rel=HAVOC unless SELF IN m
PROC signal() =
% Remove at least one thread from c.  In practice, usually just one.
% IF VAR t: SET Thread | t <= c /
APROC broadcast() = << c := {} >>
END Condition

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

Coding read/write lock using condition variables

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

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

CLASS RWLockImpl EXPORT rAcq, rRel, wAcq, wRel = % implements ForgetfulRWL
VAR rw : Int := 0
m := m.new()
c := c.new()
PROC rAcq(l) = m.acq; DO ~ rw >= 0 => c.wait(m) OD; rw + := 1; m.rel
PROC wAcq(l) = m.acq; DO ~ rw = 0 => c.wait(m) OD; rw := -1; m.rel
PROC rRel(l) = m.acq; rw - := 1; IF rw = 0 => c.signal [*] SKIP FI; m.rel
PROC wRel(l) = m.acq; rw := 0; c.broadcast; m.rel
END RWLockImpl

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

An unbounded FIFO buffer

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

MODULE Buffer[T] EXPORT Produce, Consume =
VAR b : SEQ T := {}
APROC Produce(t) = << b + := t >>
APROC Consume() -> T = VAR t | << b # {} => t := b.head; b := b.tail; RET t >>
END Buffer

The code is another example of easy concurrency.

MODULE BufferImpl[T] EXPORT Produce, Consume =
VAR b m := m.new()
c := c.new()
% ABSTRACTION FUNCTION Buffer.b = b
APROC Produce(t) = m.acq; IF b = {} => c.signal [*] SKIP FI; b := t; m.rel
APROC Consume() -> T = VAR t | << b # {} => t := b.head; b := b.tail; RET t >>
END Buffer

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
Coding Mutex with memory

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

CLASS Mutex2Impl EXPORT acq, rel =

VAR req : Thread -> Bool := {* -> false}
lastReq : Int
PROC acq() =
[a\(_2\)] req(SELF) := true;
[a\(_2\)] lastReq := SELF;
DO [a\(_2\)] (req(SELF*) \(\text{\slash}\) lastReq = SELF) \(\Rightarrow\) SKIP OD [a\(_3\)]
PROC rel() = req(SELF) := false
END Mutex2Impl

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

PROC acq0() =
[a\(_3\)] req(SELF) := true;
[a\(_2\)] lastReq := SELF;
DO [a\(_2\)] (req(SELF*) \(\text{\slash}\) lastReq = SELF) \(\Rightarrow\) SKIP OD [a\(_3\)]

We get mutual exclusion because once \(\text{req}(h)\) is true, \(h^*\) can’t get from \(a\(_2\)\) to \(a\(_3\)\). Thus \(\text{req}(h)\) acts as a lock that keeps the predicate \(h^*.\text{spc} = a\(_2\)\) true once it becomes true. Only one of the threads can get to \(a\(_3\)\) and acquire the lock. We might call the algorithm ‘polite’ because each thread defers to the other one at \(a\(_2\)\).

Of course, acq0 is no good because it can deadlock—if both threads get to \(a\(_2\)\) then neither can progress. acq avoids this problem by making it a little easier for a thread to progress: even if \(\text{req}(h^*)\), \(h^*\) can take \((a\(_2\), a\(_3\))\) if \(\text{lastReq} \neq h\). Intuitively this maintains mutual exclusion because:

- If both threads are at \(a\(_2\)\), only the one \(\neq \text{lastReq}\), say \(h\), can progress to \(a\(_3\)\) and acquire the lock. Since \(\text{lastReq}\) won’t change, \(h^*\) will remain at \(a\(_2\)\) until \(h\) releases the lock.

- Once \(h^*\) has acquired the lock with \(h^*\) not at \(a\(_2\)\), \(h^*\) can only reach \(a\(_3\)\) by setting \(\text{lastReq} := h^*\), and again \(h^*\) will remain at \(a\(_2\)\) until \(h\) releases the lock.

It ensures progress because the DO is the only place to get stuck, and whichever thread is not in lastReq will get past it. It ensures fairness because the first thread to get to \(a\(_3\)\) is the one that will get the lock first.


hi2 : Word := 0

% ABSTRACTION FUNCTION Clock.t = T(lo, hi1, hi2), Clock.Read.t1 = Read.tHist,
% Clock.Read.t2 = T(Read.tLo, Read.tH1, read.tH2)

THREAD Tick() = DO VAR newLo: Word, newHi: Word |
  << newLo := lo + 1 // base; newHi := hi1 + 1 >>;
  IF << newLo # 0 => lo := newLo >> [*] << hi2 := newHi >>; << lo := newLo >>; << hi1 := newHi >>
  FI OD

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word |
  << tH1 := h1 >>; << tLo := lo >>;  << tH2 := h2; RET T(tLo, tH1, tH2) >>
END ClockImpl

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

User and kernel mutexes and condition variables

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

Notes on the code for mutexes:

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

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

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

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

This opposite-order access pattern often works in hard concurrency, that is, when there’s not enough locking to do the job in a straightforward way. We saw another version of it in

Mutex2Impl, which sets req(h) before reading req(h*). In this case req(h) acts like a lock to keep h*.%pc = ap from changing from true to false. We also saw it in ClockImpl, where the reader and the writer of the clock touch its pieces in the opposite order.

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

CLASS MutexImpl EXPORT acq, rel = % implements ForgetfulMutex
  TYPE FH = Mutex.FH
  VAR fh := free

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

  END MutexImpl

% 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.
% indented code holds the lock

PROC KernelRelease() = SpinLock(); % indented code holds the lock
  q + := {SELF};
  IF fh = free => q := q.reml % undo previous line; will retry at acq
    [*] Deschedule(SELF) % wait, then retry at acq
    FI;
  ReleaseSpinLock()

% 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.
  % indented code holds the lock

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.
  % indented code holds the lock

% The newly scheduled thread competes with others to acquire the mutex.

END MutexImpl

Now for conditions. Note that:

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

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

CLASS ConditionImpl EXPORT wait, signal, broadcast = % implements Condition
  TYPE M = Mutex

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

  PROC KernelRelease() =
    SpinLock();
    IF q # {} => Schedule(q.head); q := q.tail [*] SKIP FI;
    ReleaseSpinLock()

  END ConditionImpl

Handout 17. Formal Concurrency 2006
THEOREM: If there exists an abstraction function or relation from states(T) to states(S) such that:

Start: If u is any initial state of T then F(u) is an initial state of S.

Step: If u and F(u) are reachable states of T and S respectively, and (u, π, u') is a step of T, then there is an execution fragment of S from F(u) to F(u'), having the same trace.

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

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

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

Start: If u is any initial state of T then there exists an initial state s of S such that (u, s) ∈ AR.

Step: If u and s are reachable states of T and S respectively, (u, s) ∈ AR, and (u, π, u') is a step of T, then there is an execution fragment of S from s to some s' having the same trace, and such that (u', s') ∈ AR.

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

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

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

The answer is that we need some invariants for `RWLockImpl`. The first, a “dominant thread invariant”, says that only a thread whose name is in `m` (a “dominant thread”) can be in certain portions of its code (those guarded by the mutex). The dominant thread invariant is in turn used to prove other invariants called “data protection invariants”.

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

### BufferImpl Implements Buffer

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

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

We use a slightly simplified version of `Produce` that always signals, and we introduce a local variable `temp` to make explicit the atomicity of assignment to the shared variable `b`.

#### Abstraction Function

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

To describe the abstraction function, we thus need to explain how to construct a state of the spec from a state of the code. So what is a state of the `Buffer` module above? It consists of:

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

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

---

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

- \( P_1 \) — before the body of \( \text{Produce} \)
- \( P_2 \) — after the body of \( \text{Produce} \)
- \( C_1 \) — before the body of \( \text{Consume} \)
- \( C_2 \) — after the body of \( \text{Consume} \)

or as annotations to the code:

\[
\text{PROC } \text{Produce}(t) = [P_1] \langle \langle b \leftarrow b + \{t\} \rangle \rangle [P_2] \\
\text{PROC } \text{Consume}() \rightarrow T = [C_1] \langle \langle b \notin \{\} \Rightarrow \text{VAR } t \leftarrow b.\text{head} | b \leftarrow b.\text{tail} ; \text{RET } t \rangle \rangle [C_2]
\]

For the code, they are:

- For a thread in \( \text{Produce} \):
  - \( p_1 \) — before \( \text{m.acq} \)
    - in \( \text{m.acq} \) — either before or after the action
  - \( p_2 \) — before \( \text{temp} := b + \{t\} \)
  - \( p_3 \) — before \( b := \text{temp} \)
  - \( p_4 \) — before \( \text{c.signal} \)
    - in \( \text{c.signal} \) — either before or after the action
  - \( p_5 \) — before \( \text{m.rel} \)
    - in \( \text{m.rel} \) — either before or after the action
  - \( p_6 \) — after \( \text{m.rel} \)

- For a thread in \( \text{Consume} \):
  - \( c_1 \) — before \( \text{m.acq} \)
    - in \( \text{m.acq} \) — either before or after action
  - \( c_2 \) — before \( \text{the test } b \notin \{\} \)
  - \( c_3 \) — before \( \text{c.wait} \)
    - in \( \text{c.wait} \) — at beginning, in middle, or at end
  - \( c_4 \) — before \( b := \text{b.head} \)
  - \( c_5 \) — before \( \text{temp} := b.\text{tail} \)
  - \( c_6 \) — before \( b := \text{temp} \)
  - \( c_7 \) — before \( \text{m.rel} \)
    - in \( \text{m.rel} \) — either before or after action
  - \( c_8 \) — before \( \text{RET } t \)
  - \( c_9 \) — after \( \text{RET } t \)

or as annotations to the code:

\[
\text{PROC } \text{Produce}(t) = \text{VAR } \text{temp} | [P_1] \text{m.acq} ; [P_2] \text{temp} := b + \{t\} ; [P_3] b := \text{temp} ; [P_4] \text{c.signal} ; [P_5] \text{m.rel} [P_6] \\
\text{PROC } \text{Consume}() \rightarrow T = \text{VAR } \text{temp} | [C_1] \text{m.acq}
\]

Notice that we have broken the assignment statements into their constituent atomic actions, introducing a temporary variable \( \text{temp} \) to hold the result of evaluating the right hand side. Also, the PC’s in the \text{Mutex} and \text{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.

\[
\text{APROC acq}() = \langle \langle m = \text{SELF} ; \text{RET} \rangle \rangle \text{HAVOC} >> \\
\text{APROC rel}() = \langle \langle m = \text{SELF} \Rightarrow m := \text{nil} ; \text{RET}[*] \rangle \rangle \text{HAVOC} >> \\
\text{APROC signal}() = \langle \langle \text{VAR } hs : \text{SET } \text{Thread} | m.\text{acq} ; \text{c.wait} ; m.\text{rel} \rightarrow T \rangle \rangle \text{HAVOC} >> \\
\text{APROC signal}() = \langle \langle \text{VAR } hs : \text{SET } \text{Thread} | m.\text{acq} ; \text{c.wait} ; m.\text{rel} \rightarrow T \rangle \rangle \text{HAVOC} >>
\]

Now we can define the mapping on program counters:

- If a thread \( h \) is not in \( \text{Produce} \) or \( \text{Consume} \) in the code, then it is not in either procedure in the spec.
- If a thread \( h \) is in \( \text{Produce} \) in the code, then:
  - If \( h.\$pc \) is in \( \{p_1, p_2, p_3\} \) or is in \( \text{m.acq} \), then in the spec \( h.\$pc = P_1 \).
  - If \( h.\$pc \) is in \( \{p_4, p_5, p_6\} \) or is in \( \text{m.rel} \) or \( \text{c.signal} \) then in the spec \( h.\$pc = P_2 \).
  - If a thread \( h \) is in \( \text{Consume} \) in the code, then:
    - If \( h.\$pc \in \{c_1, ..., c_9\} \) or is in \( \text{m.acq} \) or \( \text{c.wait} \) then in the spec \( h.\$pc = C_1 \).
    - If \( h.\$pc \) is in \( \{c_7, c_8, c_9\} \) or is in \( \text{m.rel} \) then in the spec \( h.\$pc = C_2 \).

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

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

- If \( \pi \) is an external transition, \( \pi \) simulates the singleton sequence containing just \( \pi \).
- If \( \pi \) takes a thread from a PC of \( p_1 \) to a PC of \( p_2 \), \( \pi \) simulates the singleton sequence containing just the body of \( \text{Produce} \).
- If \( \pi \) takes a thread from a PC of \( c_1 \) to a PC of \( c_2 \), \( \pi \) simulates the singleton sequence containing just the body of \( \text{Consume} \).
- All other transitions \( \pi \) 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...
To prove this, we use the fact that if \( h.\_\text{spc} = p \), then no other thread is dominant, so no other transition can change \( b \). We also have to show that any transition that puts \( h.\_\text{spc} \) at this point establishes the consequent of the implication — but there is only one transition that does this (the one that assigns to \( \text{temp} \)), and it clearly establishes the desired property.

The transition in \( \text{consume} \) 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 \( \text{ForgetfulMutex} \) implements \( \text{Mutex} \). Second, we show that \( \text{SpinLock} \) implements \( \text{ForgetfulMutex} \). For convenience, we repeat the definitions of the two modules.

**CLASS Mutex**

EXPORT acq, rel =

VAR m : (Thread + Null) := nil
PROC acq() = << m = nil => m := SELF; RET >>
PROC rel() = << m = SELF => m := nil; RET [*] HAVOC >>
END Mutex

**CLASS ForgetfulMutex**

EXPORT acq, rel =

TYPE M = ENUM[free, held]
VAR m := free
PROC acq() = << m = free => m := held; RET >>
PROC rel() = << m := free; RET >>
END ForgetfulMutex

**Proof that ForgetfulMutex implements Mutex**

These two modules have the same atomicity. The difference is that \( \text{ForgetfulMutex} \) 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 \( \text{Spinlock implements Mutex} \) uses history variables.

The main interesting relationship that \( AR \) must express is:

\[ s.m \text{ is non-nil if and only if } u.m = \text{held}. \]

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

The \$pc value for each thread is the same in both modules.
Finally, there is the technicality of the special $\text{havoc} = \text{true}$ state that occurs in $\text{Mutex}$. We handle this by allowing $\text{AR}$ to relate all states of $\text{ForgetfulMutex}$ to any state with $\text{havoc} = \text{true}$.

Having defined $\text{AR}$, we just show that the two conditions of the abstraction relation definition are satisfied.

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

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

So suppose that $\pi$ is an internal action of $\text{Forgetful Mutex}$. We argue that the given step corresponds to a single step of $\text{Mutex}$, with “the same” action. There are two cases:

1. $\pi$ is the body of an $\text{acq}$, by some thread $h$. Since $\text{acq}$ is enabled in $\text{Forgetful Mutex}$, we have $u.m =$ free, and $h.$\text{pc}'s local variable $h.$\text{pc}'s local variable $\text{acq}$ body is right before the $\text{acq}$ body in $u$. Since $(u, s) \in \text{AR}$, we have $s.m =$ nil, and also $h.$\text{pc}'s local variable $h.$\text{pc}'s local variable $\text{acq}$ body is just before the $\text{acq}$ body in $s$. Therefore, the $\text{acq}$ body for thread $h$ is also enabled in $\text{ Mutex}$. Let $s'$ be the resulting state of $\text{ Mutex}$.

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

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

On the other hand, if $u.m =$ held then $\text{Forgetful Mutex}$ sets $u.m :=$ free. Since $(u, s) \in \text{AR}$, we have $s.m =$ nil. Now there are two cases: If $s.m =$ $h.$\text{pc}'s local variable $h.$\text{pc}'s local variable $\text{acq}$ moves to right after the $\text{acq}$ step in $\text{ Mutex}$, then corresponding changes occur in both modules, which allows us to conclude $(u', s') \in \text{AR}$. Otherwise, $\text{ Mutex}$ goes to $\text{havoc} = \text{true}$. But as before, this is OK because $\text{havoc} = \text{true}$ corresponds to everything in $\text{Forgetful Mutex}$.

The conclusion is that every trace of $\text{Forgetful Mutex}$ is also a trace of $\text{ Mutex}$. Note that this proof does not imply anything about liveness, though in fact the two modules have the same liveness properties.

**Proof that SpinLock implements ForgetfulMutex**

We repeat the definition of SpinLock.

**CLASS SpinLock**

**EXPORT** acq, rel =

**TYPE** M = ENUM[free, held]

**VAR** m :::= free

**PROC** acq() = VAR t: FH |

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

Having defined $\text{AR}$, we just show that the two conditions of the abstraction relation definition are satisfied.

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

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

So suppose that $\pi$ is an internal action of $\text{Forgetful Mutex}$. We argue that the given step corresponds to a single step of $\text{Mutex}$, with “the same” action. There are two cases:

1. $\pi$ is the body of an $\text{acq}$, by some thread $h$. Since $\text{acq}$ is enabled in $\text{Forgetful Mutex}$, we have $u.m =$ free, and $h.$\text{pc}'s local variable $h.$\text{pc}'s local variable $\text{acq}$ body is right before the $\text{acq}$ body in $u$. Since $(u, s) \in \text{AR}$, we have $s.m =$ nil, and also $h.$\text{pc}'s local variable $h.$\text{pc}'s local variable $\text{acq}$ body is just before the $\text{acq}$ body in $s$. Therefore, the $\text{acq}$ body for thread $h$ is also enabled in $\text{ Mutex}$. Let $s'$ be the resulting state of $\text{ Mutex}$.

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

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

On the other hand, if $u.m =$ held then $\text{Forgetful Mutex}$ sets $u'.m :=$ free. Since $(u, s) \in \text{AR}$, we have $s.m =$ nil. Now there are two cases: If $s.m =$ $h.$\text{pc}'s local variable $h.$\text{pc}'s local variable $\text{acq}$ moves to right after the $\text{acq}$ step in $\text{ Mutex}$, then corresponding changes occur in both modules, which allows us to conclude $(u', s') \in \text{AR}$. Otherwise, $\text{ Mutex}$ goes to $\text{havoc} = \text{true}$. But as before, this is OK because $\text{havoc} = \text{true}$ corresponds to everything in $\text{Forgetful Mutex}$.

The conclusion is that every trace of $\text{Forgetful Mutex}$ is also a trace of $\text{ Mutex}$. Note that this proof does not imply anything about liveness, though in fact the two modules have the same liveness properties.

**Proof that SpinLock implements ForgetfulMutex**

We repeat the definition of SpinLock.

**CLASS SpinLock**

**EXPORT** acq, rel =

**TYPE** M = ENUM[free, held]

**VAR** m :::= free

**PROC** acq() = VAR t: FH |
4. The step does not change m. Then h.$pc in SpinLock moves to the test, with t = held. Thus, there is no change in the abstract value of h.$pc.

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

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

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

**SpinLock implements Mutex, second version**

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

**Abstraction function**

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

- A value for m (either a thread or nil); and
- for each thread that is active in the module, a program counter.

There are no local variables for threads in the spec.

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

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

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

- We augment the state of the code with two additional variables:
  
  ```
  ms: (Thread + Null) := nil % in the Spec
  hs: Bool := false % $havoc in the Spec
  ```

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

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

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

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

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

- A1 — before the body of acq
- A2 — after the body of acq
- R1 — before the body of rel
- R2 — after the body of rel

For the code, they are:

- For a thread in acq:
  
  ```
  a1 — before the VAR t
  a2 — after the VAR t and before the DO loop
  a3 — before the test-and-set in the body of the DO loop
  a4 — after the test-and-set in the body of the DO loop
  a5 — before the assignment to ms
  a6 — after the assignment to ms
  ```

- For a thread in rel:

  ```
  r1 — before the body
  r2 — after the body
  ```

The transitions in acq may be a little confusing: there’s a transition from a4 to a5, as well as transitions from a2 to a5.

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

APROC acq() = [A1] << m = nil => m := SELF >> [A2]

APROC rel() = [R1] << m # SELF => HAVOC [*] m := nil >> [R2]

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

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

APROC rel() = [r1] << m := free; hs := hs \ (ms # SELF); ms := nil >> [r2]
```
Either

If \( m = \text{free} \) then \( ms = \text{nil} \), and

If a thread is at \( a_0 \) or at \( a_4 \) with \( t = \text{free} \), then \( ms = \text{nil} \), \( m = \text{held} \), and there are no other threads at \( a_0 \), and for all other threads at \( a_4 \), \( t = \text{held} \)

or \( hs \) is true.

Given this invariant, we are done: we have shown the appropriate correspondence for all the transitions in the code. So we must prove the invariant. We do this by induction. It’s vacuously true in the initial state, since no thread could be at \( a_0 \) or \( a_4 \) 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 \( \text{rel} \) preserves the first conjunct of the invariant because afterwards both \( m = \text{free} \) and \( ms = \text{nil} \). To prove that the transition in \( \text{rel} \) preserves the second conjunct of the invariant, there are two cases, depending on whether the spec allows \( \text{HAVOC} \).

1. If it does, then the code sets \( hs \) true; this corresponds to the \( \text{HAVOC} \) 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 \( \text{HAVOC} \) is that the rest of the invariant may not hold after a rogue thread does \( \text{rel} \). Because the rogue thread resets \( m \) to \( \text{free} \), if there’s a thread at \( a_4 \) or at \( a_0 \) with \( t = \text{free} \), and \( m = \text{held} \), then after the rogue \( \text{rel} \), \( m \) is no longer held and hence the second conjunct is false. This means that it’s possible for several threads to get to \( a_4 \), or to \( a_0 \) with \( t = \text{free} \). The invariant still holds, because \( hs \) 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 \( a_4 \) with \( t = \text{free} \) or at \( a_0 \). After the transition to \( r_2 \) it’s still the case that no thread is at \( a_4 \) with \( t = \text{free} \) or at \( a_0 \), so the invariant is still true.

Now we consider the transitions in \( \text{acq} \). The transitions from \( a_1 \) to \( a_2 \) and from \( a_3 \) to \( a_3 \) obviously preserve the invariant. The transition from \( a_4 \) to \( a_3 \) 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 \( a_4 \) to \( a_3 \) also clearly preserves the invariant.

The transition from \( a_1 \) to \( a_2 \) 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 \( a_4 \). (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:

\[
ms = \text{nil} \quad (\text{since when the thread was at } a_3 \text{ m must have been free, hence the first part of the invariant applies});
\]
m = held (as a direct result of the transition);  
there are no threads at a₃ (by assumption); and  
there are no other threads at a₄ with t = free (by assumption).

So the invariant is still true after the transition.

Finally, assume a thread h is at a₄, about to transition to a₆. If the invariant is true here, then h is the only thread at a₄, and all threads at a₄ have t = held. So after it makes the transition, the invariant is vacuously true, because there is no other thread at a₃ and the threads at a₄ haven’t changed their state.

We have proved the invariant. The invariant implies that if a thread is at a₆, ms = 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 a₃ to a₄ to simulate the body of Mutex.acq. We omit the has history variable and augment the code as follows:

```
PROC acq() = [[a₃] VAR t := FH |  
[ a₃] DO [a₃] << t := m := held; [IF t # held -> ms := SELF [*] SKIP FI] >>;  
[a₄] IF t # held -> [a₅] RET [a₇] [*] SKIP FI OD;  
PROC rel() = [r₁] << m := free; ms := nil >> [r₃]
```

The abstraction function maps ms to Mutex.m as before, and it maps PC’s a₁-a₃ to A₁, and a₄-a₅ to A₄. It maps a₃ to A₁ if t = held, and to A₄ if t = free; thus a₃ to a₄ simulates Mutex.acq only if m was free, as we should expect. There is no need for an invariant; we only used it at a₃ to a₄, which no longer exists.

The simulation argument is the same as before except for a₃ to a₄, which is the only place where we changed the code. If m = held, then m and ms don’t change; hence Mutex.m doesn’t change, and neither does the abstract PC; in this case the transition simulates the empty trace. If m = free, then m becomes held, ms becomes SELF, and the abstract PC becomes A₄; in this case the transition simulates A₃ to A₄, as promised.

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

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

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

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

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

\[
\text{Holds0.set.size} <= 1 \land (h.\$pc = a₂ \Rightarrow \text{req(h)})
\]

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 {} to {h}. The second conjunct holds because it can never be true \(\Rightarrow false\), since only the step (a₂, req(h) := true, a₃) can make the antecedent true, this step also makes the consequent true, and no step away from a₂ makes the consequent false.

This argument applies toMutex0 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:

\[
\text{VAR } t \mid \text{DO } [a_2] \quad \text{req(SELF*)} \quad >> \quad [a_2] \quad \text{<< t} \quad >> \quad \text{SKIP} \quad >> \quad \text{OD } [a_3]
\]

We examine several ways to show the correctness of this; they all have the same idea, but the details differ. The most obvious one is to add the conjunct \( h.\text{pc} \neq a_2 \) to \( \text{Holds0} \), and extend the mutual exclusion conjunct of the invariant so that it covers a thread that has reached \( a_2 \) with \( t = \text{false} \):

\[
(\text{Holds0.set}.\text{size} <= 1) \quad \forall \quad h \mid h.\text{pc} = a_2 \quad \land \quad h.t = \text{false} \]

Or we could get the same effect by saying that a thread acquires the lock by reaching \( a_2 \) with \( t = \text{false} \), so that it’s the transition \((a_2, a_2)\) with \( t = \text{false} \) that simulates the body of \text{Mutex.acq}, rather than the transition to \( a_3 \) as before. This means changing the definition of \( \text{Holds0} \) to:

\[
\text{FUNC } \text{Holds0}(h: \text{Thread}) = \\
\quad \text{RET } \text{req}(h) \land h.\text{pc} \neq a_2 \land (h.\text{pc} = a_2 \implies h.t = \text{false})
\]

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

\[
h.\text{pc} = a_{21} \quad \land \quad h.t = \text{false} \implies \text{Holds}(h^*) = \text{false}
\]

and this is enough to ensure that the transition \((a_{21}, a_1)\) maintains the invariant.

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

\[
\text{PROC } \text{acq}() = \\
\quad [a_3] \quad \text{req(SELF)} := \text{true}; \\
\quad [a_1] \quad \text{lastReq} := \text{self}; \\
\quad \text{DO } [a_2] \quad (\text{req(SELF*)} \land \text{lastReq} = \text{SELF}) \implies \text{SKIP} \quad \text{OD } [a_3]
\]

We discussed liveness informally earlier, and we don’t attempt to prove it. To prove mutual exclusion, we need to extend \( \text{Holds0} \) in the obvious way:

\[
\text{FUNC } \text{Holds}(h: \text{Thread}) = \text{req}(h) \land \quad (h.\text{pc} \neq a_1) \land (h.\text{pc} \neq a_2)
\]

and add \( \lor h.\text{pc} = a_1 \) to the antecedent of the invariant. In order to have mutual exclusion, it must be true that \( h \) won’t find \( \text{lastReq} = h^* \) as long as \( h^* \) holds the lock. We need to add a conjunct to the invariant to express this. This leaves us with:

\[
\text{FUNC } \text{Holds0.set}.\text{size} <= 1 \land \\
\quad \forall (h.\text{pc} = a_2 \land h.\text{pc} \neq a_1) \implies \text{req}(h) \\
\quad (\text{Holds}(h^*) \land \quad h.\text{pc} = a_2 \implies \text{lastReq} = h)
\]

The last conjunct holds because \((a_1, a_2)\) makes it true, and the only way to make it false is for \( h^* \) to do \( \text{lastReq} = \text{SELF} \), which it can only do from \( a_1 \), so that \( \text{Holds}(h^*) \) is false. With this invariant it’s obvious that \((a_2, a_3)\) 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 is not too big. In this case, if the two threads are \( a \) and \( b \), the state space is just:

\[
a.\text{pc} \in \{a_0, a_1, a_2, a_3\} \\
b.\text{pc} \in \{a_0, a_1, a_2, a_3\} \\
\text{req}(a) \in \{\text{false}, \text{true}\} \\
\text{req}(b) \in \{\text{false}, \text{true}\} \\
\text{lastReq} \in \{a, b\}
\]

We can write down a state concisely with one digit to represent each PC, \( a \) or \( f \) for each \( \text{req} \), and an \( a \) or \( b \) for \( \text{lastReq} \). Thus \( \text{00ffa} \) is \( a.\text{pc} = a_0, b.\text{pc} = a_0, \text{req}(a) = \text{false}, \text{req}(b) = \text{false}, \text{lastReq} = a \). When the value of a component is unimportant we write \( x \) for it.

Now we can display the complete state machine:

![State Machine Diagram]

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

1. The value of \( \text{req} \) follows from the PC’s, which cuts the number of reachable states to 32.
2. \( 33xxx \) is not reachable. This is the **mutual exclusion invariant**, which is that both PC’s cannot be in the critical section at the end of \text{acq}. This removes 2 states.
3. At the top of the picture the value of \( \text{lastReq} \) 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 \( \text{lastReq} \) 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 transitions from \( 00fx \) and \( 1tttx \) 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.

Unfortunately, this story is incomplete, because it assumed that we can evaluate

\[
\text{(rel(SELF) \&\& lastReq = SELF)} \quad \text{atomically, which is not true.}
\]

To fix this we have to break this evaluation down into two steps, with a new program counter value in the middle. We reproduce the whole procedure for easy reference:

\[
\text{PROC acq0()} = \begin{cases}
\{a_0\} \text{ req(SELF)} := \text{true}; \\
\{a_1\} \text{ lastReq := SELF} \\
\text{DO } \{a_2\} \text{ req(SELF) => SKIP FI } \{a_3\}
\end{cases}
\]

\[
\text{PROC rel0()} = \text{ req(SELF) := false}
\]

This increases the number of states from 128 to 200; the state space is:

\[
\begin{align*}
\text{a.$pc} & \quad \text{IN } \{a_0, a_1, a_2, a_3\} \\
\text{b.$pc} & \quad \text{IN } \{a_0, a_1, a_2, a_3\} \\
\text{req(a)} & \quad \text{IN } \{\text{false, true}\} \\
\text{req(b)} & \quad \text{IN } \{\text{false, true}\} \\
\text{lastReq} & \quad \text{IN } \{a, b\}
\end{align*}
\]

Most of the states are still unreachable, but there are 26 reachable states that we need to distinguish (using \( x \) as before when the value of a component doesn’t affect the possible transitions). Instead of drawing a new state diagram like the previous one, we present a matrix that exposes the symmetries in a different way, by using the two PCs as the \( x \) and \( y \) coordinates. Except for the \( x \) in the \( 2tttx, 24tttx \), and \( 42tttx \) states, the PC values determine the other state components. The convention in this table is that for each state there’s a transition that advances a thread’s PC to the next non-blank row (for \( a \)) or column (for \( b \)) unless there’s an arrow going somewhere else. Like the diagram, the table is symmetric.

The new transitions are the ones that involve a PC of \( a \). These transitions don’t change any state variables. As the table shows, what they do is to articulate the fine structure of the \( 2tttx \) loops in the simpler state diagram. If \( a \) gets to \( 2tttx \) it loops between that state and \( 4tttx \); similarly \( b \) loops between \( x2ttb \) and \( x4ttb \). On the other hand, when \( \text{lastReq = b} \), thread \( a \) can get through the sequence \( x2ttb, x4ttb \rightarrow 3tttb \) (where \( x \) can be \( 2 \) or \( 4 \) and similarly \( b \) can get through the sequence \( x2tta, x4tta \rightarrow 3tta \).

**ClockImpl implements Clock**

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

\[
\text{MODULE Clock EXPORT Read =}
\]

\[
\text{VAR t : Int := 0} \quad \% \text{the current time}
\]

\[
\text{THREAD Tick() = DO } \text{<< t := t + 1 >> OD} \quad \% \text{demon thread advances t}
\]

\[
\text{PROC Read() -> Int = VAR t1: Int \mid}
\]

\[
\begin{align*}
&\text{R1} \quad \text{<< t1 := t >>; } \\
&\text{R2} \quad \text{<< VAR t2 | t1 <= t2 / t2 <= t => RET t2 >> R1}
\end{align*}
\]

\[
\text{END Clock}
\]

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

**MODULE ClockImpl**

```
EXPORT Read =

CONST base := 2**32

TYPE Word = Int SUCHTHAT word IN base.seq)

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

% ABSTRACTION FUNCTION Clock.t = T(lo, hi1, hi2), Clock.Read.t1 = Read.t1Hist,
%   Clock.Read.t2 = T(Read.tLo, Read.tH1, read.tH2)
% The PC correspondence is \( R_1 \leftrightarrow r_1 \), \( R_2 \leftrightarrow r_2 \), \( R_3 \leftrightarrow r_3 \)

THREAD Tick() = DO VAR newLo: Word, newHi: Word |
   << newLo := lo + 1 // base; newHi := hi1 + 1 >>;
   IF << newLo # 0 => lo := newLo >>
   [*] << hi2 := newHi >>; << lo := newLo >>; << hi1 := newHi >>
FI OD

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word, t1Hist: Int |
   \[r_1\] << tH1 := hi1; t1Hist := T(lo, hi1, hi2) >>;
   \[r_2\] % I2: T(lo, tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
   << tLo := lo; >>
   \[r_3\] % I3: T(tLo, tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
   << tH2 := hi2; RET T(tLo, tH1, tH2) >>
   \[r_4\] % I4: \$a IN t1Hist .. T(lo, hi1, hi2)

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

END ClockImpl
```

The whole invariant is thus

\[ h.\text{pc} = r_2 \implies I_2 \quad \text{or} \quad h.\text{pc} = r_3 \implies I_3 \quad \text{or} \quad h.\text{pc} = r_4 \implies I_4 \]

The steps of \( \text{Read} \) clearly maintain this invariant, since they don’t change the value before \( \text{IN} \).

The steps of \( \text{Tick} \) maintain it by case analysis.