Chapter 8

Liveness and Fairness

The specifications we have written so far say what a system must *not* do. The clock must not advance from 11 to 9; the receiver must not receive a message if the FIFO is empty. They don't require that the system ever actually do anything. The clock need never tick; the sender need never send any messages. Our specifications have described what are called *safety properties*. If a safety property is violated, it is violated at some particular point in the behavior—by a step that advances the clock from 11 to 9, or that reads the wrong value from memory. Therefore, we can talk about a safety property being satisfied by a finite behavior, which means that it has not been violated by any step so far.

We now learn how to specify that something *does* happen—that the clock keeps ticking, or that a value is eventually read from memory. We specify *liveness properties*—ones that cannot be violated at any particular instant. Only by examining an entire infinite behavior can we tell that the clock has stopped ticking, or that a message is never sent.

We express liveness properties as temporal formulas. This means that, to add liveness conditions to your specifications, you have to understand temporal logic—the logic of temporal formulas. The chapter begins, in Section 8.1, with a more rigorous look at what a temporal formula means. To understand a logic, you have to understand what its true formulas are. Section 8.2 is about temporal tautologies, the true formulas of temporal logic. Sections 8.4–8.7 describe how to use temporal formulas to specify liveness properties. Section 8.8 completes our study of temporal logic by examining the temporal quantifier $\exists$. Finally, Section 8.9 reviews what we've done and explains why the undisciplined use of temporal logic is dangerous.

This chapter is the only one that contains proofs. It would be nice if you learned to write similar proofs yourself, but it doesn't matter if you don't. The proofs are here because studying them can help you develop the intuitive understanding of temporal formulas that you need to write specifications—

an understanding that makes the truth of a simple temporal tautology like $\Box\Box F \equiv \Box F$ as obvious as the truth of a simple theorem about numbers like $\forall\, n \in Nat : 2 * n \geq n$.

Many readers will find that this chapter taxes their mathematical ability. Don't worry if you have trouble understanding it. Treat this chapter as an exercise to stretch your mind and prepare you to add liveness properties to your specifications. And remember that liveness properties are likely to be the least important part of your specification. You will probably not lose much if you simply omit them.

8.1 Temporal Formulas

Recall that a state assigns a value to every variable, and a behavior is an infinite sequence of states. A temporal formula is true or false of a behavior. Formally, a temporal formula F assigns a Boolean value, which we write $\sigma \models F$, to a behavior σ. We say that F is true of σ, or that σ satisfies F, iff $\sigma \models F$ equals TRUE. To define the meaning of a temporal formula F, we have to explain how to determine the value of $\sigma \models F$ for any behavior σ. For now, we consider only temporal formulas that don't contain the temporal existential quantifier $\exists$.

It's easy to define the meaning of a Boolean combination of temporal formulas in terms of the meanings of those formulas. The formula $F \wedge G$ is true of a behavior σ iff both F and G are true of σ, and $\neg F$ is true of σ iff F is not true of σ. These definitions are written more formally as

$$\sigma \models (F \wedge G) \triangleq (\sigma \models F) \wedge (\sigma \models G) \qquad \sigma \models \neg F \triangleq \neg (\sigma \models F)$$

These are the definitions of the meaning of $\wedge$ and of $\neg$ as operators on temporal formulas. The meanings of the other Boolean operators are similarly defined. We can also define in this way the ordinary predicate-logic quantifiers $\forall$ and $\exists$ as operators on temporal formulas—for example:

$$\sigma \models (\exists\, r : F) \triangleq \exists\, r : (\sigma \models F)$$

Ordinary quantification over constant sets is defined the same way. For example, if S is an ordinary constant expression—that is, one containing no variables—then

$$\sigma \models (\forall\, r \in S : F) \triangleq \forall\, r \in S : (\sigma \models F)$$

Quantifiers are discussed further in Section 8.8 below.

All the unquantified temporal formulas that we've seen have been Boolean combinations of three simple kinds of formulas, which have the following meanings:

- A state predicate, viewed as a temporal formula, is true of a behavior iff it is true in the first state of the behavior.

State function and *state predicate* are defined on page 25.

- A formula $\Box P$, where P is a state predicate, is true of a behavior iff P is true in every state of the behavior.
- A formula $\Box[N]_v$, where N is an action and v is a state function, is true of a behavior iff every successive pair of steps in the behavior is a $[N]_v$ step.

Since a state predicate is an action that contains no primed variables, we can both combine and generalize these three kinds of temporal formulas into the two kinds of formulas A and $\Box A$, where A is an action. I'll first explain the meanings of these two kinds of formulas, and then define the operator $\Box$ in general. To do this, I will use the notation that σ_i is the $(i+1)^{\text{st}}$ state of the behavior σ, for any natural number i, so σ is the behavior $\sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots$.

We interpret an arbitrary action A as a temporal formula by defining $\sigma \models A$ to be true iff the first two states of σ are an A step. That is, we define $\sigma \models A$ to be true iff $\sigma_0 \to \sigma_1$ is an A step. In the special case when A is a state predicate, $\sigma_0 \to \sigma_1$ is an A step iff A is true in state σ_0, so this definition of $\sigma \models A$ generalizes our interpretation of a state predicate as a temporal formula.

We have already seen that $\Box[N]_v$ is true of a behavior iff each step is a $[N]_v$ step. This leads us to define $\sigma \models \Box A$ to be true iff $\sigma_n \to \sigma_{n+1}$ is an A step, for all natural numbers n.

We now generalize from the definition of $\sigma \models \Box A$ for an action A to the definition of $\sigma \models \Box F$ for an arbitrary temporal formula F. We defined $\sigma \models \Box A$ to be true iff $\sigma_n \to \sigma_{n+1}$ is an A step for all n. This is true iff A, interpreted as a temporal formula, is true of a behavior whose first step is $\sigma_n \to \sigma_{n+1}$, for all n. Let's define σ^{+n} to be the suffix of σ obtained by deleting its first n states:

$$\sigma^{+n} \triangleq \sigma_n \to \sigma_{n+1} \to \sigma_{n+2} \to \cdots$$

Then $\sigma_n \to \sigma_{n+1}$ is the first step of σ^{+n}, so $\sigma \models \Box A$ is true iff $\sigma^{+n} \models A$ is true for all n. In other words

$$\sigma \models \Box A \equiv \forall\, n \in Nat : \sigma^{+n} \models A$$

The obvious generalization is

$$\sigma \models \Box F \triangleq \forall\, n \in Nat : \sigma^{+n} \models F$$

for any temporal formula F. In other words, σ satisfies $\Box F$ iff every suffix σ^{+n} of σ satisfies F. This defines the meaning of the temporal operator $\Box$.

We have now defined the meaning of any temporal formula built from actions (including state predicates), Boolean operators, and the $\Box$ operator. For example:

$$\begin{aligned}
&\sigma \models \Box((x = 1) \Rightarrow \Box(y > 0)) \\
&\quad \equiv \forall\, n \in Nat : \sigma^{+n} \models ((x = 1) \Rightarrow \Box(y > 0)) && \text{By the meaning of } \Box. \\
&\quad \equiv \forall\, n \in Nat : (\sigma^{+n} \models (x = 1)) \Rightarrow (\sigma^{+n} \models \Box(y > 0)) && \text{By the meaning of } \Rightarrow. \\
&\quad \equiv \forall\, n \in Nat : (\sigma^{+n} \models (x = 1)) \Rightarrow && \text{By the meaning of } \Box. \\
&\qquad\qquad\qquad (\forall\, m \in Nat : (\sigma^{+n})^{+m} \models (y > 0))
\end{aligned}$$

Thus, $\sigma \models \Box((x = 1) \Rightarrow \Box(y > 0))$ is true iff, for all $n \in Nat$, if $x = 1$ is true in state σ_n, then $y > 0$ is true in all states σ_{n+m} with $m \geq 0$.

To understand temporal formulas intuitively, think of σ_n as the state of the universe at time instant n during the behavior σ.[1] For any state predicate P, the expression $\sigma^{+n} \models P$ asserts that P is true at time n. Thus, $\Box((x = 1) \Rightarrow \Box(y > 0))$ asserts that, any time $x = 1$ is true, $y > 0$ is true from then on. For an arbitrary temporal formula F, we also interpret $\sigma^{+n} \models F$ as the assertion that F is true at time instant n. The formula $\Box F$ then asserts that F is true at all times. We can therefore read $\Box$ as *always* or *henceforth* or *from then on*.

We saw in Section 2.2 that a specification should allow stuttering steps—ones that leave unchanged all the variables appearing in the formula. A stuttering step represents a change only to some part of the system not described by the formula; adding it to the behavior should not affect the truth of the formula. We say that a formula F is *invariant under stuttering*[2] iff adding or deleting a stuttering step to a behavior σ does not affect whether σ satisfies F. A sensible formula should be invariant under stuttering. There's no point writing formulas that aren't sensible, so TLA allows you to write only temporal formulas that are invariant under stuttering.

A state predicate (viewed as a temporal formula) is invariant under stuttering, since its truth depends only on the first state of a behavior, and adding a stuttering step doesn't change the first state. An arbitrary action is not invariant under stuttering. For example, the action $[x' = x + 1]_x$ is satisfied by a behavior σ in which x is left unchanged in the first step and incremented by 2 in the second step; it isn't satisfied by the behavior obtained by removing the initial stuttering step from σ. However, the formula $\Box[x' = x + 1]_x$ is invariant under stuttering, since it is satisfied by a behavior iff every step that changes x is an $x' = x + 1$ step—a condition not affected by adding or deleting stuttering steps.

In general, the formula $\Box[A]_v$ is invariant under stuttering, for any action A and state function v. However, $\Box A$ is not invariant under stuttering for an arbitrary action A. For example, $\Box(x' = x + 1)$ can be made false by adding a step that does not change x. So, even though we have assigned a meaning to $\Box(x' = x + 1)$, it isn't a legal TLA formula.

Invariance under stuttering is preserved by $\Box$ and by the Boolean operators—that is, if F and G are invariant under stuttering, then so are $\Box F$, $\neg F$, $F \wedge G$, $\forall\, x \in S : F$, etc. So, state predicates, formulas of the form $\Box[N]_v$, and all formulas obtainable from them by applying $\Box$ and Boolean operators are invariant under stuttering.

[1] It is because we think of σ_n as the state at time n, and because we usually measure time starting from 0, that I number the states of a behavior starting with 0 rather than 1.

[2] This is a completely new sense of the word *invariant*; it has nothing to do with the concept of invariance discussed already.

We now examine five especially important classes of formulas that are constructed from arbitrary temporal formulas F and G. We introduce new operators for expressing the first three.

$\Diamond F$ is defined to equal $\neg\Box\neg F$. It asserts that F is not always false, which means that F is true at some time:

$$\begin{array}{lll}
\sigma \models \Diamond F & & \\
\quad \equiv \ \sigma \models \neg\Box\neg F & & \text{By definition of } \Diamond. \\
\quad \equiv \ \neg(\sigma \models \Box\neg F) & & \text{By the meaning of } \neg. \\
\quad \equiv \ \neg(\forall\, n \in Nat : \sigma^{+n} \models \neg F) & & \text{By the meaning of } \Box. \\
\quad \equiv \ \neg(\forall\, n \in Nat : \neg(\sigma^{+n} \models F)) & & \text{By the meaning of } \neg. \\
\quad \equiv \ \exists\, n \in Nat : \sigma^{+n} \models F & & \text{Because } \neg\forall\neg \text{ is equivalent to } \exists.
\end{array}$$

We usually read $\Diamond$ as *eventually*, taking eventually to include now.

$F \rightsquigarrow G$ is defined to equal $\Box(F \Rightarrow \Diamond G)$. The same kind of calculation we just did for $\sigma \models \Diamond F$ shows

$$\sigma \models (F \rightsquigarrow G) \quad \equiv \quad \forall\, n \in Nat : (\sigma^{+n} \models F) \Rightarrow (\exists\, m \in Nat : (\sigma^{+(n+m)} \models G))$$

The formula $F \rightsquigarrow G$ asserts that whenever F is true, G is eventually true—that is, G is true then or at some later time. We read $\rightsquigarrow$ as *leads to*.

$\Diamond\langle A\rangle_v$ is defined to equal $\neg\Box[\neg A]_v$, where A is an action and v a state function. It asserts that not every step is a $(\neg A) \vee (v' = v)$ step, so some step is a $\neg((\neg A) \vee (v' = v))$ step. Since $\neg(P \vee Q)$ is equivalent to $(\neg P) \wedge (\neg Q)$, for any P and Q, action $\neg((\neg A) \vee (v' = v))$ is equivalent to $A \wedge (v' \neq v)$. Hence, $\Diamond\langle A\rangle_v$ asserts that some step is an $A \wedge (v' \neq v)$ step—that is, an A step that changes v. We define the action $\langle A\rangle_v$ by

I pronounce $\langle A\rangle_v$ as *angle A sub v*.

$$\langle A\rangle_v \triangleq A \wedge (v' \neq v)$$

so $\Diamond\langle A\rangle_v$ asserts that eventually an $\langle A\rangle_v$ step occurs. We think of $\Diamond\langle A\rangle_v$ as the formula obtained by applying the operator $\Diamond$ to $\langle A\rangle_v$, although technically it's not because $\langle A\rangle_v$ isn't a temporal formula.

$\Box\Diamond F$ asserts that at all times, F is true then or at some later time. For time 0, this implies that F is true at some time $n_0 \geq 0$. For time $n_0 + 1$, it implies that F is true at some time $n_1 \geq n_0 + 1$. For time $n_1 + 1$, it implies that F is true at some time $n_2 \geq n_1 + 1$. Continuing the process, we see that F is true at an infinite sequence of time instants $n_0, n_1, n_2, \ldots$. So, $\Box\Diamond F$ implies that F is true at infinitely many instants. Conversely, if F is true at infinitely many instants, then, at every instant, F must be true at some later instant, so $\Box\Diamond F$ is true. Therefore, $\Box\Diamond F$ asserts that F is *infinitely often* true. In particular, $\Box\Diamond\langle A\rangle_v$ asserts that infinitely many $\langle A\rangle_v$ steps occur.

$\Diamond\Box F$ asserts that eventually (at some time), F becomes true and remains true thereafter. In other words, $\Diamond\Box F$ asserts that F is *eventually always* true. In particular, $\Diamond\Box[N]_v$ asserts that, eventually, every step is a $[N]_v$ step.

The operators $\Box$ and $\Diamond$ have higher precedence (bind more tightly) than the Boolean operators, so $\Diamond F \vee \Box G$ means $(\Diamond F) \vee (\Box G)$. The operator $\rightsquigarrow$ has lower precedence than $\wedge$ and $\vee$.

8.2 Temporal Tautologies

A temporal theorem is a temporal formula that is satisfied by all behaviors. In other words, F is a theorem iff $\sigma \models F$ equals TRUE for all behaviors σ. For example, the *HourClock* module asserts that $HC \Rightarrow \Box HCini$ is a theorem, where HC and $HCini$ are the formulas defined in the module. This theorem expresses a property of the hour clock.

The formula $\Box HCini \Rightarrow HCini$ is also a theorem. However, it tells us nothing about the hour clock because it's true regardless of how $HCini$ is defined. For example, substituting $x > 7$ for $HCini$ yields the theorem $\Box(x > 7) \Rightarrow (x > 7)$. A formula like $\Box HCini \Rightarrow HCini$ that is true when any formulas are substituted for its identifiers is called a *tautology*. To distinguish them from the tautologies of ordinary logic, tautologies containing temporal operators are sometimes called *temporal* tautologies.

Let's prove that $\Box HCini \Rightarrow HCini$ is a temporal tautology. To avoid confusing the arbitrary identifier $HCini$ in this tautology with the formula $HCini$ defined in the *HourClock* module, let's replace it by F, so the tautology becomes $\Box F \Rightarrow F$. There are axioms and inference rules for temporal logic from which we can prove any temporal tautology that, like $\Box F \Rightarrow F$, contains no quantifiers. However, it's often easier and more instructive to prove them directly from the meanings of the operators. We prove that $\Box F \Rightarrow F$ is a tautology by proving that $\sigma \models (\Box F \Rightarrow F)$ equals TRUE, for any behavior σ and any formula F. The proof is simple:

$$
\begin{array}{lll}
\sigma \models (\Box F \Rightarrow F) & \equiv (\sigma \models \Box F) \Rightarrow (\sigma \models F) & \text{By the meaning of } \Rightarrow. \\
& \equiv (\forall\, n \in Nat : \sigma^{+n} \models F) \Rightarrow (\sigma \models F) & \text{By definition of } \Box. \\
& \equiv (\forall\, n \in Nat : \sigma^{+n} \models F) \Rightarrow (\sigma^{+0} \models F) & \text{By definition of } \sigma^{+0}. \\
& \equiv \text{TRUE} & \text{By predicate logic.}
\end{array}
$$

The temporal tautology $\Box F \Rightarrow F$ asserts the obvious fact that, if F is true at all times, then it's true at time 0. Such a simple tautology should be obvious once you become accustomed to thinking in terms of temporal formulas. Here are three more simple tautologies, along with their English translations.

$\neg\Box F \equiv \Diamond\neg F$

F is not always true iff it is eventually false.

$\Box(F \land G) \equiv (\Box F) \land (\Box G)$

F and G are both always true iff F is always true and G is always true. Another way of saying this is that $\Box$ distributes over $\land$.

$\Diamond(F \lor G) \equiv (\Diamond F) \lor (\Diamond G)$

F or G is eventually true iff F is eventually true or G is eventually true. Another way of saying this is that $\Diamond$ distributes over $\lor$.

At the heart of the proof of each of these tautologies is a tautology of predicate logic. For example, the proof that $\Box$ distributes over $\land$ relies on the fact that $\forall$ distributes over $\land$:

$\sigma \models (\Box(F \land G) \equiv (\Box F) \land (\Box G))$

$$
\begin{array}{lll}
\equiv & (\sigma \models \Box(F \land G)) \equiv (\sigma \models (\Box F) \land (\Box G)) & \text{By the meaning of } \equiv. \\
\equiv & (\sigma \models \Box(F \land G)) \equiv (\sigma \models \Box F) \land (\sigma \models \Box G) & \text{By the meaning of } \land. \\
\equiv & (\forall\, n \in Nat : \sigma^{+n} \models (F \land G)) \equiv & \text{By definition of } \Box. \\
 & \quad (\forall\, n \in Nat : \sigma^{+n} \models F) \land (\forall\, n \in Nat : \sigma^{+n} \models G) & \\
\equiv & \text{TRUE} & \text{By the predicate-logic tautology } (\forall\, x \in S : P \land Q) \equiv (\forall\, x \in S : P) \land (\forall\, x \in S : Q).
\end{array}
$$

The operator $\Box$ doesn't distribute over $\lor$, nor does $\Diamond$ distribute over $\land$. For example, $\Box((n \geq 0) \lor (n < 0))$ is not equivalent to $(\Box(n \geq 0) \lor \Box(n < 0))$; the first formula is true for any behavior in which n is always a number, but the second is false for a behavior in which n assumes both positive and negative values. However, the following two formulas are tautologies:

$(\Box F) \lor (\Box G) \Rightarrow \Box(F \lor G)$ $\qquad$ $\Diamond(F \land G) \Rightarrow (\Diamond F) \land (\Diamond G)$

Either of these tautologies can be derived from the other by substituting $\neg F$ for F and $\neg G$ for G. Making this substitution in the second tautology yields

$$
\begin{array}{lll}
\text{TRUE} \equiv & \Diamond((\neg F) \land (\neg G)) \Rightarrow (\Diamond \neg F) \land (\Diamond \neg G) & \text{By substitution in the second tautology.} \\
\equiv & \Diamond \neg(F \lor G) \Rightarrow (\Diamond \neg F) \land (\Diamond \neg G) & \text{Because } (\neg P \land \neg Q) \equiv \neg(P \lor Q). \\
\equiv & \neg \Box(F \lor G) \Rightarrow (\neg \Box F) \land (\neg \Box G) & \text{Because } \Diamond \neg H \equiv \neg \Box H. \\
\equiv & \neg \Box(F \lor G) \Rightarrow \neg((\Box F) \lor (\Box G)) & \text{Because } (\neg P \land \neg Q) \equiv \neg(P \lor Q). \\
\equiv & (\Box F) \lor (\Box G) \Rightarrow \Box(F \lor G) & \text{Because } (\neg P \Rightarrow \neg Q) \equiv (Q \Rightarrow P).
\end{array}
$$

This pair of tautologies illustrates a general law: from any temporal tautology, we obtain a *dual* tautology by making the replacements

$\Box \leftarrow \Diamond$ $\qquad$ $\Diamond \leftarrow \Box$ $\qquad$ $\land \leftarrow \lor$ $\qquad$ $\lor \leftarrow \land$

and reversing the direction of all implications. (Any $\equiv$ or $\neg$ is left unchanged.) As in the example above, the dual tautology can be proved from the original by replacing each identifier with its negation and applying the (dual) tautologies $\Diamond \neg F \equiv \neg \Box F$ and $\neg \Diamond F \equiv \Box \neg F$ along with propositional-logic reasoning.

Another important pair of dual tautologies assert that $\Box\Diamond$ distributes over $\vee$ and $\Diamond\Box$ distributes over $\wedge$:

(8.1) $\Box\Diamond(F \vee G) \equiv (\Box\Diamond F) \vee (\Box\Diamond G) \qquad \Diamond\Box(F \wedge G) \equiv (\Diamond\Box F) \wedge (\Diamond\Box G)$

The first asserts that F or G is true infinitely often iff F is true infinitely often or G is true infinitely often. Its truth should be fairly obvious, but let's prove it. To reason about $\Box\Diamond$, it helps to introduce the symbol $\exists_\infty$, which means *there exist infinitely many*. In particular, $\exists_\infty i \in Nat : P(i)$ means that $P(i)$ is true for infinitely many natural numbers i. On page 91, we showed that $\Box\Diamond F$ asserts that F is true infinitely often. Using $\exists_\infty$, we can express this as

(8.2) $(\sigma \models \Box\Diamond F) \equiv (\exists_\infty i \in Nat : \sigma^{+i} \models F)$

The same reasoning proves the following more general result, where P is any operator:

(8.3) $(\forall\, n \in Nat : \exists\, m \in Nat : P(n+m)) \equiv \exists_\infty i \in Nat : P(i)$

Here is another useful tautology involving $\exists_\infty$, where P and Q are arbitrary operators and S is an arbitrary set:

(8.4) $(\exists_\infty i \in S : P(i) \vee Q(i)) \equiv (\exists_\infty i \in S : P(i)) \vee (\exists_\infty i \in S : Q(i))$

Using these results, it's now easy to prove that $\Box\Diamond$ distributes over $\vee$:

$$\begin{array}{lll} \sigma \models \Box\Diamond(F \vee G) & & \\ \quad \equiv \exists_\infty i \in Nat : \sigma^{+i} \models (F \vee G) & & \text{By (8.2).} \\ \quad \equiv (\exists_\infty i \in Nat : \sigma^{+i} \models F) \vee (\exists_\infty i \in Nat : \sigma^{+i} \models G) & & \text{By (8.4).} \\ \quad \equiv (\sigma \models \Box\Diamond F) \vee (\sigma \models \Box\Diamond G) & & \text{By (8.2).} \end{array}$$

From this, we deduce the dual tautology, that $\Diamond\Box$ distributes over $\wedge$.

In any TLA tautology, replacing a temporal formula by an action yields a tautology—a formula that is true for all behaviors—even if that formula isn't a legal TLA formula. (Remember that we have defined the meaning of nonTLA formulas like $\Box(x' = x + 1)$.) We can apply the rules of logic to transform those nonTLA tautologies into TLA tautologies. Among these rules are the following dual equivalences, which are easy to check:

$$[A \wedge B]_v \equiv [A]_v \wedge [B]_v \qquad \langle A \vee B \rangle_v \equiv \langle A \rangle_v \vee \langle B \rangle_v$$

(The second asserts that an $A \vee B$ step that changes v is either an A step that changes v or a B step that changes v.)

As an example of substituting actions for temporal formulas in TLA tautologies, let's substitute $\langle A \rangle_v$ and $\langle B \rangle_v$ for F and G in the first tautology of (8.1) to get

(8.5) $\Box\Diamond(\langle A \rangle_v \vee \langle B \rangle_v) \equiv (\Box\Diamond\langle A \rangle_v) \vee (\Box\Diamond\langle B \rangle_v)$

This isn't a TLA tautology, because $\Box\Diamond(\langle A\rangle_v \lor \langle B\rangle_v)$ isn't a TLA formula. However, a general rule of logic tells us that replacing a subformula by an equivalent one yields an equivalent formula. Substituting $\langle A \lor B\rangle_v$ for $\langle A\rangle_v \lor \langle B\rangle_v$ in (8.5) gives us the following TLA tautology:

$$\Box\Diamond\langle A \lor B\rangle_v \equiv (\Box\Diamond\langle A\rangle_v) \lor (\Box\Diamond\langle B\rangle_v)$$

8.3 Temporal Proof Rules

A proof rule is a rule for deducing true formulas from other true formulas. For example, the *Modus Ponens* Rule of propositional logic tells us that, for any formulas F and G, if we have proved F and $F \Rightarrow G$, then we can deduce G. Since the laws of propositional logic hold for temporal logic as well, we can apply the *Modus Ponens* Rule when reasoning about temporal formulas. Temporal logic also has some proof rules of its own. One is

> **Generalization Rule** From F we can infer $\Box F$, for any temporal formula F.

This rule asserts that, if F is true for all behaviors, then so is $\Box F$. To prove it, we must show that, if $\sigma \models F$ is true for every behavior σ, then $\tau \models \Box F$ is true for every behavior τ. The proof is easy:

$$\begin{array}{lll} \tau \models \Box F & \equiv & \forall\, n \in Nat : \tau^{+n} \models F \qquad \text{By definition of } \Box. \\ & \equiv & \forall\, n \in Nat : \text{TRUE} \qquad \text{By the assumption that } \sigma \models F \text{ equals TRUE, for all } \sigma. \\ & \equiv & \text{TRUE} \qquad \text{By predicate logic.} \end{array}$$

Another temporal proof rule is

> **Implies Generalization Rule** From $F \Rightarrow G$ we can infer $\Box F \Rightarrow \Box G$, for any temporal formulas F and G.

The Generalization Rule can be derived from the Implies Generalization Rule and the tautology $\text{TRUE} = \Box\text{TRUE}$ by substituting TRUE for F and F for G.

The difference between a temporal proof rule and a temporal tautology can be confusing. In propositional logic, every proof rule has a corresponding tautology. The *Modus Ponens* Rule, which asserts that we can deduce G by proving F and $F \Rightarrow G$, implies the tautology $F \land (F \Rightarrow G) \Rightarrow G$. But in temporal logic, a proof rule need not imply a tautology. The Generalization Rule, which states that we can deduce $\Box F$ by proving F, does not imply that $F \Rightarrow \Box F$ is a tautology. The rule means that, if $\sigma \models F$ is true for all σ, then $\sigma \models \Box F$ is true for all σ. That's different from the (false) assertion that $F \Rightarrow \Box F$ is a tautology, which would mean that $\sigma \models (F \Rightarrow \Box F)$ is true for all σ. For example, $\sigma \models (F \Rightarrow \Box F)$ equals FALSE if F is a state predicate that is true in the first state of σ and is false in some other state of σ. Forgetting the distinction between a proof rule and a tautology is a common source of mistakes when using temporal logic.

8.4 Weak Fairness

It's easy to specify liveness properties with the temporal operators $\Box$ and $\Diamond$. For example, consider the hour-clock specification of module *HourClock* in Figure 2.1 on page 20. We can require that the clock never stops by asserting that there must be infinitely many *HCnxt* steps. The obvious way to write this assertion is $\Box\Diamond HCnxt$, but that's not a legal TLA formula because *HCnxt* is an action, not a temporal formula. However, an *HCnxt* step advances the value *hr* of the clock, so it changes *hr*. Therefore, an *HCnxt* step is also an *HCnxt* step that changes *hr*—that is, it's an $\langle HCnxt\rangle_{hr}$ step. We can thus write the liveness property that the clock never stops as $\Box\Diamond\langle HCnxt\rangle_{hr}$. So, we can take $HC \wedge \Box\Diamond\langle HCnxt\rangle_{hr}$ to be the specification of a clock that never stops.

Before continuing, I must make a confession and then lead you on a brief digression about subscripts. Let me first confess that the argument I just gave, that we can write $\Box\Diamond\langle HCnxt\rangle_{hr}$ in place of $\Box\Diamond HCnxt$, was sloppy (a polite term for *wrong*). Not every *HCnxt* step changes *hr*. Consider a state in which *hr* has some value that is not a number—perhaps a value ∞. An *HCnxt* step that starts in such a state sets the new value of *hr* to $\infty + 1$. We don't know what $\infty + 1$ equals; it might or might not equal ∞. If it does, then the *HCnxt* step leaves *hr* unchanged, so it is not an $\langle HCnxt\rangle_{hr}$ step. Fortunately, states in which the value of *hr* is not a number are irrelevant. Because we are conjoining the liveness condition to the safety specification *HC*, we care only about behaviors that satisfy *HC*. In all such behaviors, *hr* is always a number, and every *HCnxt* step is an $\langle HCnxt\rangle_{hr}$ step. Therefore, $HC \wedge \Box\Diamond\langle HCnxt\rangle_{hr}$ is equivalent to the nonTLA formula $HC \wedge \Box\Diamond HCnxt$.[3]

When writing liveness properties, the syntax of TLA often forces us to write $\langle A\rangle_v$ instead of A, for some action A. As in the case of *HCnxt*, the safety specification usually implies that any A step changes some variable. To avoid having to think about which variables A actually changes, we generally take the subscript v to be the tuple of all variables, which is changed iff any variable changes. But what if A does allow stuttering steps? It's silly to assert that a stuttering step eventually occurs, since such an assertion is not invariant under stuttering. So, if A does allow stuttering steps, we want to require not that an A step eventually occurs, but that a nonstuttering A step occurs—that is, an $\langle A\rangle_v$ step, where v is the tuple of all the specification's variables. The syntax of TLA forces us to say what we should mean.

When discussing formulas, I will usually ignore the angle brackets and subscripts. For example, I might describe $\Box\Diamond\langle HCnxt\rangle_{hr}$ as the assertion that there are infinitely many *HCnxt* steps, rather than infinitely many $\langle Hnxt\rangle_{hr}$, which is what it really asserts. This finishes the digression; we now return to specifying liveness conditions.

[3] Even though $HC \wedge \Box\Diamond HCnxt$ is not a TLA formula, its meaning has been defined, so we can determine whether it is equivalent to a TLA formula.

Let's modify specification *Spec* of module *Channel* (Figure 3.2 on page 30) to require that every value sent is eventually received. We do this by conjoining a liveness condition to *Spec*. The analog of the liveness condition for the clock is $\Box\Diamond\langle Rcv\rangle_{chan}$, which asserts that there are infinitely many *Rcv* steps. However, only a value that has been sent can be received, so this condition would also require that infinitely many values be sent—a requirement we don't want to make. We want to permit behaviors in which no value is ever sent, so no value is ever received. We require only that any value that is sent is eventually received.

To assure that all values that should be received are eventually received, it suffices to require only that the next value to be received eventually is received. (When that value has been received, the one after it becomes the next value to be received, so it must eventually be received, and so on.) More precisely, we need only require it always to be the case that, if there is a value to be received, then the next value to be received eventually is received. The next value is received by a *Rcv* step, so the requirement is[4]

$$\Box(\text{There is an unreceived value } \Rightarrow \Diamond\langle Rcv\rangle_{chan})$$

There is an unreceived value iff action *Rcv* is enabled, meaning that it is possible to take a *Rcv* step. TLA$^+$ defines ENABLED A to be the predicate that is true iff action A is enabled. The liveness condition can then be written

$$\Box(\text{ENABLED } \langle Rcv\rangle_{chan} \Rightarrow \Diamond\langle Rcv\rangle_{chan}) \tag{8.6}$$

In the ENABLED formula, it doesn't matter if we write *Rcv* or $\langle Rcv\rangle_{chan}$. We add the angle brackets so the two actions appearing in the formula are the same.

In any behavior satisfying the safety specification *HC*, it's always possible to take an *HCnxt* step that changes *hr*. Action $\langle HCnxt\rangle_{hr}$ is therefore always enabled, so ENABLED $\langle HCnxt\rangle_{hr}$ is true throughout such a behavior. Since TRUE $\Rightarrow \Diamond\langle HCnxt\rangle_{hr}$ is equivalent to $\Diamond\langle HCnxt\rangle_{hr}$, we can replace the liveness condition $\Box\Diamond\langle HCnxt\rangle_{hr}$ for the hour clock with

$$\Box(\text{ENABLED } \langle HCnxt\rangle_{hr} \Rightarrow \Diamond\langle HCnxt\rangle_{hr})$$

This suggests the following general liveness condition for an action A:

$$\Box(\text{ENABLED } \langle A\rangle_v \Rightarrow \Diamond\langle A\rangle_v)$$

This condition asserts that, if A ever becomes enabled, then an A step will eventually occur—even if A remains enabled for only a fraction of a nanosecond and is never again enabled. The obvious practical difficulty of implementing such a condition suggests that it's too strong. So, we replace it with the weaker formula $\text{WF}_v(A)$, defined to equal

$$\Box(\Box\text{ENABLED } \langle A\rangle_v \Rightarrow \Diamond\langle A\rangle_v) \tag{8.7}$$

[4] $\Box(F \Rightarrow \Diamond G)$ equals $F \rightsquigarrow G$, so we could write this formula more compactly with $\rightsquigarrow$. However, it's more convenient to keep it in the form $\Box(F \Rightarrow \Diamond G)$

This formula asserts that, if A ever becomes forever enabled, then an A step must eventually occur. WF stands for Weak Fairness, and the condition $\mathrm{WF}_v(A)$ is called *weak fairness on* A. We'll soon see that our liveness conditions for the clock and the channel can be written as WF formulas. But first, let's examine (8.7) and the following two formulas, which turn out to be equivalent to it:

(8.8) $\Box\Diamond(\neg\textsc{enabled}\,\langle A\rangle_v)\ \vee\ \Box\Diamond\langle A\rangle_v$

(8.9) $\Diamond\Box(\textsc{enabled}\,\langle A\rangle_v)\ \Rightarrow\ \Box\Diamond\langle A\rangle_v$

These three formulas can be expressed in English as

(8.7) It's always the case that, if A is enabled forever, then an A step eventually occurs.

(8.8) A is infinitely often disabled, or infinitely many A steps occur.

(8.9) If A is eventually enabled forever, then infinitely many A steps occur.

The equivalence of these three formulas isn't obvious. Trying to deduce their equivalence from the English expressions often leads to confusion. The best way to avoid confusion is to use mathematics. We show that the three formulas are equivalent by proving that (8.7) is equivalent to (8.8) and that (8.8) is equivalent to (8.9). Instead of proving that they are equivalent for an individual behavior, we can use tautologies that we've already seen to prove their equivalence directly. Here's a proof that (8.7) is equivalent to (8.8). Studying it will help you learn to write liveness conditions.

$\Box(\Box\textsc{enabled}\,\langle A\rangle_v \Rightarrow \Diamond\langle A\rangle_v)$

$\equiv\ \Box(\neg\Box\textsc{enabled}\,\langle A\rangle_v \vee \Diamond\langle A\rangle_v)$ Because $(F \Rightarrow G) \equiv (\neg F \vee G)$.

$\equiv\ \Box(\Diamond\neg\textsc{enabled}\,\langle A\rangle_v \vee \Diamond\langle A\rangle_v)$ Because $\neg\Box F \equiv \Diamond\neg F$.

$\equiv\ \Box\Diamond(\neg\textsc{enabled}\,\langle A\rangle_v \vee \langle A\rangle_v)$ Because $\Diamond F \vee \Diamond G \equiv \Diamond(F \vee G)$.

$\equiv\ \Box\Diamond(\neg\textsc{enabled}\,\langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v$ Because $\Box\Diamond(F \vee G) \equiv \Box\Diamond F \vee \Box\Diamond G$.

The equivalence of (8.8) and (8.9) is proved as follows:

$\Box\Diamond(\neg\textsc{enabled}\,\langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v$

$\equiv\ \neg\Diamond\Box(\textsc{enabled}\,\langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v$ Because $\Box\Diamond\neg F \equiv \Box\neg\Box F \equiv \neg\Diamond\Box F$.

$\equiv\ \Diamond\Box(\textsc{enabled}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$ Because $(F \Rightarrow G) \equiv (\neg F \vee G)$.

We now show that the liveness conditions for the hour clock and the channel can be written as weak fairness conditions.

First, consider the hour clock. In any behavior satisfying HC, an $\langle HCnxt\rangle_{hr}$ step is always enabled, so $\Diamond\Box(\textsc{enabled}\,\langle HCnxt\rangle_{hr})$ equals TRUE. Therefore, HC implies that $\mathrm{WF}_{hr}(HCnxt)$, which equals (8.9), is equivalent to formula $\Box\Diamond\langle HCnxt\rangle_{hr}$, our liveness condition for the hour clock.

Now, consider the channel. I claim that the liveness condition (8.6) can be replaced by $\mathrm{WF}_{chan}(Rcv)$. More precisely, *Spec* implies that these two formulas are equivalent, so conjoining either of them to *Spec* yields equivalent specifications. The proof rests on the observation that, in any behavior satisfying *Spec*, once *Rcv* becomes enabled (because a value has been sent), it can be disabled only by a *Rcv* step (which receives the value). In other words, it's always the case that if *Rcv* is enabled, then it is enabled forever or a *Rcv* step eventually occurs. Stated formally, this observation asserts that *Spec* implies

$$(8.10)\ \Box\,(\,\text{ENABLED}\,\langle Rcv\rangle_{chan}\ \Rightarrow\ \Box(\text{ENABLED}\,\langle Rcv\rangle_{chan})\ \vee\ \Diamond\langle Rcv\rangle_{chan}\,)$$

We show that we can take $\mathrm{WF}_{chan}(Rcv)$ as our liveness condition by showing that (8.10) implies the equivalence of (8.6) and $\mathrm{WF}_{chan}(Rcv)$.

The proof is by purely temporal reasoning; we need no other facts about the channel specification. Both for compactness and to emphasize the generality of our reasoning, let's replace $\text{ENABLED}\,\langle Rcv\rangle_{chan}$ by E and $\langle Rcv\rangle_{chan}$ by A. Using version (8.7) of the definition of WF, we must prove

$$(8.11)\ \Box(E \Rightarrow \Box E \vee \Diamond A)\ \ \Rightarrow\ \ (\Box(E \Rightarrow \Diamond A)\ \equiv\ \Box(\Box E \Rightarrow \Diamond A))$$

So far, all our proofs have been by calculation. That is, we have proved that two formulas are equivalent, or that a formula is equivalent to TRUE, by proving a chain of equivalences. That's a good way to prove simple things, but it's usually better to tackle a complicated formula like (8.11) by splitting its proof into pieces. We have to prove that one formula implies the equivalence of two others. The equivalence of two formulas can be proved by showing that each implies the other. More generally, to prove that P implies $Q \equiv R$, we prove that $P \wedge Q$ implies R and that $P \wedge R$ implies Q. So, we prove (8.11) by proving the two formulas

$$(8.12)\ \Box(E \Rightarrow \Box E \vee \Diamond A)\ \wedge\ \Box(E \Rightarrow \Diamond A)\ \Rightarrow\ \Box(\Box E \Rightarrow \Diamond A)$$

$$(8.13)\ \Box(E \Rightarrow \Box E \vee \Diamond A)\ \wedge\ \Box(\Box E \Rightarrow \Diamond A)\ \Rightarrow\ \Box(E \Rightarrow \Diamond A)$$

Both (8.12) and (8.13) have the form $\Box F \wedge \Box G \Rightarrow \Box H$. We first show that, for any formulas F, G, and H, we can deduce $\Box F \wedge \Box G \Rightarrow \Box H$ by proving $F \wedge G \Rightarrow H$. We do this by assuming $F \wedge G \Rightarrow H$ and proving $\Box F \wedge \Box G \Rightarrow \Box H$ as follows:

1. $\Box(F \wedge G) \Rightarrow \Box H$

   PROOF: By the assumption $F \wedge G \Rightarrow H$ and the Implies Generalization Rule (page 95), substituting $F \wedge G$ for F and H for G in the rule.

2. $\Box F \wedge \Box G \Rightarrow \Box H$

   PROOF: By step 1 and the tautology $\Box(F \wedge G) \equiv \Box F \wedge \Box G$.

This shows that we can deduce $\Box F \wedge \Box G \Rightarrow \Box H$ by proving $F \wedge G \Rightarrow H$, for any F, G, and H. We can therefore prove (8.12) and (8.13) by proving

(8.14) $(E \Rightarrow \Box E \vee \Diamond A) \wedge (E \Rightarrow \Diamond A) \;\Rightarrow\; (\Box E \Rightarrow \Diamond A)$

(8.15) $(E \Rightarrow \Box E \vee \Diamond A) \wedge (\Box E \Rightarrow \Diamond A) \;\Rightarrow\; (E \Rightarrow \Diamond A)$

The proof of (8.14) is easy. In fact, we don't even need the first conjunct; we can prove $(E \Rightarrow \Diamond A) \Rightarrow (\Box E \Rightarrow \Diamond A)$ as follows:

$(E \Rightarrow \Diamond A)$

$\equiv \quad (\Box E \Rightarrow E) \wedge (E \Rightarrow \Diamond A)$ Because $\Box E \Rightarrow E$ is a temporal tautology.

$\Rightarrow \quad (\Box E \Rightarrow \Diamond A)$ By the tautology $(P \Rightarrow Q) \wedge (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$.

The proof of (8.15) uses only propositional logic. We deduce (8.15) by substituting E for P, $\Box E$ for Q, and $\Diamond A$ for R in the following propositional-logic tautology:

$$(P \Rightarrow Q \vee R) \wedge (Q \Rightarrow R) \;\Rightarrow\; (P \Rightarrow R)$$

A little thought should make the validity of this tautology seem obvious. If not, you can check it by constructing a truth table.

These proofs of (8.14) and (8.15) complete the proof that we can take $\mathrm{WF}_{chan}(Rcv)$ instead of (8.7) as our liveness condition for the channel.

8.5 The Memory Specification

8.5.1 The Liveness Requirement

Let's now strengthen the specification of the linearizable memory of Section 5.3 with the liveness requirement that every request must receive a response. (We don't require that a request ever be issued.) The liveness requirement is conjoined to the internal memory specification, formula *ISpec* of the *InternalMemory* module (Figure 5.2 on pages 52–53).

We want to express the liveness requirement in terms of weak fairness. To do this, we must understand when actions are enabled. The action $Rsp(p)$ is enabled only if the action

(8.16) $Reply(p, buf[p], memInt, memInt')$

is enabled. Recall that the operator *Reply* is a constant parameter, declared in the *MemoryInterface* module (Figure 5.1 on page 48). Without knowing more about this operator, we can't say when action (8.16) is enabled.

Let's assume that *Reply* actions are always enabled. That is, for any processor p and reply r, and any old value $miOld$ of $memInt$, there is a new value

miNew of *memInt* such that $Reply(p, r, miOld, miNew)$ is true. For simplicity, we just assume that this is true for all p and r, and add the following assumption to the *MemoryInterface* module:

$$\text{ASSUME } \forall\, p,\ r,\ miOld\ :\ \exists\, miNew\ :\ Reply(p,\ r,\ miOld,\ miNew)$$

We should also make a similar assumption for *Send*, but we don't need it here.

We will subscript our weak fairness formulas with the tuple of all variables, so let's give that tuple a name:

$$vars\ \triangleq\ \langle\, memInt,\ mem,\ ctl,\ buf\,\rangle$$

When processor p issues a request, it enables the $Do(p)$ action, which remains enabled until a $Do(p)$ step occurs. The weak fairness condition $\text{WF}_{vars}(Do(p))$ implies that this $Do(p)$ step must eventually occur. A $Do(p)$ step enables the $Rsp(p)$ action, which remains enabled until a $Rsp(p)$ step occurs. The weak fairness condition $\text{WF}_{vars}(Rsp(p))$ implies that this $Rsp(p)$ step, which produces the desired response, must eventually occur. Hence, the requirement

$$\text{WF}_{vars}(Do(p))\ \wedge\ \text{WF}_{vars}(Rsp(p)) \tag{8.17}$$

assures that every request issued by processor p must eventually receive a reply. We want this condition to hold for every processor p, so we can take, as the liveness condition for the memory specification, the formula

$$Liveness\ \triangleq\ \forall\, p \in Proc\ :\ \text{WF}_{vars}(Do(p))\ \wedge\ \text{WF}_{vars}(Rsp(p)) \tag{8.18}$$

The internal memory specification is then $ISpec \wedge Liveness$.

8.5.2 Another Way to Write It

I find a single fairness condition simpler than the conjunction of fairness conditions. Seeing the conjunction of the two weak fairness formulas in the definition of *Liveness* leads me to ask if it can be replaced by a single weak fairness condition on $Do(p) \vee Rsp(p)$. Such a replacement isn't always possible; in general, the formulas $\text{WF}_v(A) \wedge \text{WF}_v(B)$ and $\text{WF}_v(A \vee B)$ are not equivalent. However, in this case, we can replace the two fairness conditions with one. If we define

$$Liveness2\ \triangleq\ \forall\, p \in Proc\ :\ \text{WF}_{vars}(Do(p) \vee Rsp(p)) \tag{8.19}$$

then $ISpec \wedge Liveness2$ is equivalent to $ISpec \wedge Liveness$. As we will see, this equivalence holds because any behavior satisfying *ISpec* satisfies the following two properties:

DR1. Whenever $Do(p)$ is enabled, $Rsp(p)$ can never become enabled unless a $Do(p)$ step eventually occurs.

DR2. Whenever $Rsp(p)$ is enabled, $Do(p)$ can never become enabled unless a $Rsp(p)$ step eventually occurs.

These properties are satisfied because a request to p is issued by a $Req(p)$ step, executed by a $Do(p)$ step, and responded to by a $Rsp(p)$ step; and then, the next request to p can be issued by a $Req(p)$ step. Each of these steps becomes possible (the action enabled) only after the preceding one occurs.

Let's now show that DR1 and DR2 imply that the conjunction of weak fairness of $Do(p)$ and of $Rsp(p)$ is equivalent to weak fairness of $Do(p) \lor Rsp(p)$. For compactness, and to emphasize the generality of what we're doing, let's replace $Do(p)$, $Rsp(p)$, and *vars* by A, B, and v, respectively.

First, we must restate DR1 and DR2 as temporal formulas. The basic form of DR1 and DR2 is

Whenever F is true, G can never be true unless H is eventually true.

This is expressed in temporal logic as $\Box(F \Rightarrow \Box\neg G \lor \Diamond H)$. (The assertion "$P$ unless Q" just means $P \lor Q$.) Adding suitable subscripts, we can therefore write DR1 and DR2 in temporal logic as

$$DR1 \triangleq \Box(\,\text{ENABLED}\,\langle A\rangle_v \Rightarrow \Box\neg\,\text{ENABLED}\,\langle B\rangle_v \lor \Diamond\langle A\rangle_v\,)$$
$$DR2 \triangleq \Box(\,\text{ENABLED}\,\langle B\rangle_v \Rightarrow \Box\neg\,\text{ENABLED}\,\langle A\rangle_v \lor \Diamond\langle B\rangle_v\,)$$

Our goal is to prove

$$(8.20)\quad DR1 \land DR2 \Rightarrow (\,\text{WF}_v(A) \land \text{WF}_v(B) \equiv \text{WF}_v(A \lor B)\,)$$

This is complicated, so we split the proof into pieces. As in the proof of (8.11) in Section 8.4 above, we prove an equivalence by proving two implications. To prove (8.20), we prove the two theorems

$$DR1 \land DR2 \land \text{WF}_v(A) \land \text{WF}_v(B) \Rightarrow \text{WF}_v(A \lor B)$$
$$DR1 \land DR2 \land \text{WF}_v(A \lor B) \Rightarrow \text{WF}_v(A) \land \text{WF}_v(B)$$

We prove them by showing that they are true for an arbitrary behavior σ. In other words, we prove

$$(8.21)\quad (\sigma \models DR1 \land DR2 \land \text{WF}_v(A) \land \text{WF}_v(B)) \Rightarrow (\sigma \models \text{WF}_v(A \lor B))$$
$$(8.22)\quad (\sigma \models DR1 \land DR2 \land \text{WF}_v(A \lor B)) \Rightarrow (\sigma \models \text{WF}_v(A) \land \text{WF}_v(B))$$

These formulas seem daunting. Whenever you have trouble proving something, try a proof by contradiction; it gives you an extra hypothesis for free—namely, the negation of what you're trying to prove. Proofs by contradiction are especially useful in temporal logic. To prove (8.21) and (8.22) by contradiction, we need to compute $\neg(\sigma \models \text{WF}_v(C))$ for an action C. From the definition (8.7) of WF, we easily get

$$(8.23)\quad (\sigma \models \text{WF}_v(C)) \equiv \forall\, n \in Nat : (\sigma^{+n} \models \Box\,\text{ENABLED}\,\langle C\rangle_v) \Rightarrow (\sigma^{+n} \models \Diamond\langle C\rangle_v)$$

This and the tautology

$$\neg(\forall\, x \in S \,:\, P \Rightarrow Q) \;\equiv\; (\exists\, x \in S \,:\, P \wedge \neg Q)$$

of predicate logic yields

(8.24) $\neg(\sigma \models \mathrm{WF}_v(C)) \;\equiv$
$\exists\, n \in \mathit{Nat} \,:\, (\sigma^{+n} \models \Box\, \textsc{enabled}\, \langle C \rangle_v) \wedge \neg(\sigma^{+n} \models \Diamond \langle C \rangle_v)$

We also need two further results, both of which are derived from the tautology $\langle A \vee B \rangle_v \equiv \langle A \rangle_v \vee \langle B \rangle_v$. Combining this tautology with the temporal tautology $\Diamond(F \vee G) \equiv \Diamond F \vee \Diamond G$ yields

(8.25) $\Diamond \langle A \vee B \rangle_v \;\equiv\; \Diamond \langle A \rangle_v \vee \Diamond \langle B \rangle_v$

Combining the tautology with the observation that an action $C \vee D$ is enabled iff action C or action D is enabled yields

(8.26) $\textsc{enabled}\, \langle A \vee B \rangle_v \;\equiv\; \textsc{enabled}\, \langle A \rangle_v \vee \textsc{enabled}\, \langle B \rangle_v$

We can now prove (8.21) and (8.22). To prove (8.21), we assume that σ satisfies *DR1*, *DR2*, $\mathrm{WF}_v(A)$, and $\mathrm{WF}_v(B)$, but it does not satisfy $\mathrm{WF}_v(A \vee B)$, and we obtain a contradiction. By (8.24), the assumption that σ does not satisfy $\mathrm{WF}_v(A \vee B)$ means that there exists some number n such that

(8.27) $\sigma^{+n} \models \Box\, \textsc{enabled}\, \langle A \vee B \rangle_v$

(8.28) $\neg(\sigma^{+n} \models \Diamond \langle A \vee B \rangle_v)$

We obtain a contradiction from (8.27) and (8.28) as follows:

1. $\neg(\sigma^{+n} \models \Diamond \langle A \rangle_v) \wedge \neg(\sigma^{+n} \models \Diamond \langle B \rangle_v)$
   Proof: By (8.28) and (8.25), using the tautology $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$.
2. (a) $(\sigma^{+n} \models \textsc{enabled}\, \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \Box\, \neg \textsc{enabled}\, \langle B \rangle_v)$
   (b) $(\sigma^{+n} \models \textsc{enabled}\, \langle B \rangle_v) \Rightarrow (\sigma^{+n} \models \Box\, \neg\, \textsc{enabled}\, \langle A \rangle_v)$
   Proof: By definition of *DR1*, the assumption $\sigma \models \mathit{DR1}$ implies
   $$(\sigma^{+n} \models \textsc{enabled}\, \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \Box\, \neg\, \textsc{enabled}\, \langle B \rangle_v) \vee (\sigma^{+n} \models \Diamond \langle A \rangle_v)$$
   and part (a) then follows from 1. The proof of (b) is similar.
3. (a) $(\sigma^{+n} \models \textsc{enabled}\, \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \Box\, \textsc{enabled}\, \langle A \rangle_v)$
   (b) $(\sigma^{+n} \models \textsc{enabled}\, \langle B \rangle_v) \Rightarrow (\sigma^{+n} \models \Box\, \textsc{enabled}\, \langle B \rangle_v)$
   Proof: Part (a) follows from 2(a), (8.27), (8.26), and the temporal tautology
   $$\Box(F \vee G) \wedge \Box\neg G \;\Rightarrow\; \Box F$$
   The proof of part (b) is similar.

4. (a) $(\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \Diamond \langle A \rangle_v)$
   (b) $(\sigma^{+n} \models \text{ENABLED } \langle B \rangle_v) \Rightarrow (\sigma^{+n} \models \Diamond \langle B \rangle_v)$

   PROOF: The assumption $\sigma \models \text{WF}_v(A)$ and (8.23) imply

   $$(\sigma^{+n} \models \Box \text{ENABLED } \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \Diamond \langle A \rangle_v)$$

   Part (a) follows from this and 3(a). The proof of part (b) is similar.

5. $(\sigma^{+n} \models \Diamond \langle A \rangle_v) \vee (\sigma^{+n} \models \Diamond \langle B \rangle_v)$

   PROOF: Since $\Box F$ implies F, for any F, (8.27) and (8.26) imply

   $$(\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \vee (\sigma^{+n} \models \text{ENABLED } \langle B \rangle_v)$$

   Step 5 then follows by propositional logic from step 4.

Steps 1 and 5 provide the required contradiction.

We can prove (8.22) by assuming that σ satisfies *DR1*, *DR2*, and $\text{WF}_v(A \vee B)$, and then proving that it satisfies $\text{WF}_v(A)$ and $\text{WF}_v(B)$. We prove only that it satisfies $\text{WF}_v(A)$; the proof for $\text{WF}_v(B)$ is similar. The proof is by contradiction; we assume that σ does not satisfy $\text{WF}_v(A)$ and obtain a contradiction. By (8.24), the assumption that σ does not satisfy $\text{WF}_v(A)$ means that there exists some number n such that

(8.29) $\sigma^{+n} \models \Box \text{ ENABLED } \langle A \rangle_v$

(8.30) $\neg (\sigma^{+n} \models \Diamond \langle A \rangle_v)$

We obtain the contradiction as follows:

1. $\sigma^{+n} \models \Diamond \langle A \vee B \rangle_v$

   PROOF: From (8.29) and (8.26) we deduce $\sigma^{+n} \models \Box \text{ ENABLED } \langle A \vee B \rangle_v$. By the assumption $\sigma \models \text{WF}_v(A \vee B)$ and (8.23), this implies $\sigma^{+n} \models \Diamond \langle A \vee B \rangle_v$.

2. $\sigma^{+n} \models \Box \neg \text{ENABLED } \langle B \rangle_v$

   PROOF: From (8.29) we deduce $\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v$, which by the assumption $\sigma \models$ *DR1* and the definition of *DR1* implies

   $$(\sigma^{+n} \models \Box \neg \text{ENABLED } \langle B \rangle_v) \vee (\sigma^{+n} \models \Diamond \langle A \rangle_v)$$

   The assumption (8.30) then implies $\sigma^{+n} \models \Box \neg \text{ENABLED } \langle B \rangle_v$.

3. $\neg (\sigma^{+n} \models \Diamond \langle B \rangle_v)$

   PROOF: The definition of ENABLED implies $\neg \text{ENABLED } \langle B \rangle_v \Rightarrow \neg \langle B \rangle_v$. (A $\langle B \rangle_v$ step can occur only when it is enabled.) From this, simple temporal reasoning implies

   $$(\sigma^{+n} \models \Box \neg \text{ENABLED } \langle B \rangle_v) \Rightarrow \neg (\sigma^{+n} \models \Diamond \langle B \rangle_v)$$

   (A formal proof uses the Implies Generalization Rule and the tautology $\Box \neg F \equiv \neg \Diamond F$.) We then deduce $\neg (\sigma^{+n} \models \Diamond \langle B \rangle_v)$ from 2.

4. $\neg (\sigma^{+n} \models \Diamond \langle A \vee B \rangle_v)$

   PROOF: By (8.30), 3, and (8.25), using the tautology $\neg P \wedge \neg Q \equiv \neg (P \vee Q)$.

Steps 1 and 4 provide the necessary contradiction. This completes our proof of (8.22), which completes our proof of (8.20).

8.5.3 A Generalization

Formula (8.20) provides a rule for replacing the conjunction of weak fairness requirements on two actions with weak fairness of their disjunction. We now generalize it from two actions A and B to n actions $A_1, \ldots, A_n$. The generalization of *DR1* and *DR2* is

$$DR(i,j) \triangleq \Box(\text{ENABLED}\,\langle A_i\rangle_v \Rightarrow \Box\neg\text{ENABLED}\,\langle A_j\rangle_v \vee \Diamond\langle A_i\rangle_v)$$

If we substitute A_1 for A and A_2 for B, then *DR1* becomes $DR(1,2)$ and *DR2* becomes $DR(2,1)$. The generalization of (8.20) is

(8.31) $$(\forall\, i,j \in 1\,..\,n : (i \neq j) \Rightarrow DR(i,j)) \Rightarrow (\text{WF}_v(A_1) \wedge \ldots \wedge \text{WF}_v(A_n) \equiv \text{WF}_v(A_1 \vee \ldots \vee A_n))$$

To decide if you can replace the conjunction of weak fairness conditions by a single one in a specification, you will probably find it easier to use the following informal statement of (8.31):

> **WF Conjunction Rule** If $A_1, \ldots, A_n$ are actions such that, for any distinct i and j, whenever $\langle A_i\rangle_v$ is enabled, $\langle A_j\rangle_v$ cannot become enabled unless an $\langle A_i\rangle_v$ step occurs, then $\text{WF}_v(A_1) \wedge \ldots \wedge \text{WF}_v(A_n)$ is equivalent to $\text{WF}_v(A_1 \vee \ldots \vee A_n)$.

Perhaps the best way to think of this rule is as an assertion about an arbitrary individual behavior σ. Its hypothesis is then that $\sigma \models DR(i,j)$ holds for all distinct i and j; its conclusion is

$$\sigma \models (\text{WF}_v(A_1) \wedge \ldots \wedge \text{WF}_v(A_n) \equiv \text{WF}_v(A_1 \vee \ldots \vee A_n))$$

To replace $\text{WF}_v(A_1) \wedge \ldots \wedge \text{WF}_v(A_n)$ by $\text{WF}_v(A_1 \vee \ldots \vee A_n)$ in a specification, you have to check that any behavior satisfying the safety part of the specification also satisfies $DR(i,j)$, for all distinct i and j.

Conjunction and disjunction are special cases of quantification:

$$F_1 \vee \ldots \vee F_n \equiv \exists\, i \in 1\,..\,n : F_i$$
$$F_1 \wedge \ldots \wedge F_n \equiv \forall\, i \in 1\,..\,n : F_i$$

We can therefore easily restate the WF Conjunction Rule as a condition on when $\forall\, i \in S : \text{WF}_v(A_i)$ and $\text{WF}_v(\exists\, i \in S : A_i)$ are equivalent, for a finite set S. The resulting rule is actually valid for any set S:

> **WF Quantifier Rule** If, for all $i \in S$, the A_i are actions such that, for any distinct i and j in S, whenever $\langle A_i\rangle_v$ is enabled, $\langle A_j\rangle_v$ cannot become enabled unless an $\langle A_i\rangle_v$ step occurs, then $\forall\, i \in S : \text{WF}_v(A_i)$ is equivalent to $\text{WF}_v(\exists\, i \in S : A_i)$.

8.6 Strong Fairness

We define $\mathrm{SF}_v(A)$, *strong fairness* of action A, to be either of the following two equivalent formulas:

(8.32) $\Diamond\Box(\neg \text{ENABLED } \langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v$

(8.33) $\Box\Diamond\text{ENABLED } \langle A\rangle_v \Rightarrow \Box\Diamond\langle A\rangle_v$

Intuitively, these two formulas assert

(8.32) A is eventually disabled forever, or infinitely many A steps occur.

(8.33) If A is infinitely often enabled, then infinitely many A steps occur.

The proof that (8.32) and (8.33) are equivalent is similar to the proof on page 98 that the two formulations (8.8) and (8.9) of $\mathrm{WF}_v(A)$ are equivalent.

Definition (8.32) of $\mathrm{SF}_v(A)$ is obtained from definition (8.8) of $\mathrm{WF}_v(A)$ by replacing $\Box\Diamond(\neg \text{ENABLED } \langle A\rangle_v)$ with $\Diamond\Box(\neg \text{ENABLED } \langle A\rangle_v)$. Since $\Diamond\Box F$ (eventually always F) is stronger than (implies) $\Box\Diamond F$ (infinitely often F) for any formula F, strong fairness is stronger than weak fairness. We can express weak and strong fairness as follows:

- Weak fairness of A asserts that an A step must eventually occur if A is *continuously* enabled.
- Strong fairness of A asserts that an A step must eventually occur if A is *continually* enabled.

Continuously means without interruption. *Continually* means repeatedly, possibly with interruptions.

Strong fairness need not be strictly stronger than weak fairness. Weak and strong fairness of an action A are equivalent iff A infinitely often disabled implies that either A eventually becomes forever disabled, or else infinitely many A steps occur. This is expressed formally by the tautology

$$(\mathrm{WF}_v(A) \equiv \mathrm{SF}_v(A)) \equiv (\Box\Diamond(\neg \text{ENABLED } \langle A\rangle_v) \Rightarrow \Diamond\Box(\neg \text{ENABLED } \langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v)$$

In the channel example, weak and strong fairness of *Rcv* are equivalent because *Spec* implies that, once enabled, *Rcv* can be disabled only by a *Rcv* step. Hence, if *Rcv* is disabled infinitely often, then it either eventually remains disabled forever, or else it is disabled infinitely often by *Rcv* steps.

The analogs of the WF Conjunction and WF Quantifier Rules (page 105) hold for strong fairness—for example:

> **SF Conjunction Rule** If $A_1, \ldots, A_n$ are actions such that, for any distinct i and j, whenever action A_i is enabled, action A_j cannot become enabled until an A_i step occurs, then $\mathrm{SF}_v(A_1) \wedge \ldots \wedge \mathrm{SF}_v(A_n)$ is equivalent to $\mathrm{SF}_v(A_1 \vee \ldots \vee A_n)$.

Strong fairness can be more difficult to implement than weak fairness, and it is a less common requirement. A strong fairness condition should be used in a specification only if it is needed. When strong and weak fairness are equivalent, the fairness property should be written as weak fairness.

Liveness properties can be subtle. Expressing them with *ad hoc* temporal formulas can lead to errors. We will specify liveness as the conjunction of weak and/or strong fairness properties whenever possible—and it almost always is possible. Having a uniform way of expressing liveness makes specifications easier to understand. Section 8.9.2 below discusses an even more compelling reason for using fairness to specify liveness.

8.7 The Write-Through Cache

Let's now add liveness to the write-through cache, specified in Figure 5.5 on pages 57–59. We want our specification to guarantee that every request eventually receives a response, without requiring that any requests are issued. This requires fairness on all the actions that make up the next-state action *Next* except for the following:

- A $Req(p)$ action, which issues a request.
- An $Evict(p, a)$ action, which evicts an address from the cache.
- A $MemQWr$ action, if $memQ$ contains only write requests and is not full (has fewer than $QLen$ elements). Since a response to a write request can be issued before the value is written to memory, failing to execute a $MemQWr$ action can prevent a response only if it prevents the dequeuing of a read operation in $memQ$ or the enqueuing of an operation (because $memQ$ is full).

For simplicity, let's require fairness for the $MemQWr$ action too; we'll weaken this requirement later. Our liveness condition then has to assert fairness of the actions

$$MemQWr \quad MemQRd \quad Rsp(p) \quad RdMiss(p) \quad DoRd(p) \quad DoWr(p)$$

for all p in $Proc$. We now must decide whether to assert weak or strong fairness for these actions. Weak and strong fairness are equivalent for an action that, once enabled, remains enabled until it is executed. This is the case for all of these actions except $DoRd(p)$, $RdMiss(p)$, and $DoWr(p)$.

The $DoRd(p)$ action can be disabled by an *Evict* step that evicts the requested data from the cache. In this case, fairness of other actions should imply that the data will eventually be returned to the cache, re-enabling $DoRd(p)$.

The data cannot be evicted again until the $DoRd(p)$ action is executed, and weak fairness then suffices to ensure that the necessary $DoRd(p)$ step eventually occurs.

The $RdMiss(p)$ and $DoWr(p)$ actions append a request to the $memQ$ queue. They are disabled if that queue is full. A $RdMiss(p)$ or $DoWr(p)$ could be enabled and then become disabled because a $RdMiss(q)$ or $DoWr(q)$, for a different processor q, appends a request to $memQ$. We therefore need strong fairness for the $RdMiss(p)$ and $DoWr(p)$ actions. So, the fairness conditions we need are

Weak Fairness for $Rsp(p)$, $DoRd(p)$, $MemQWr$, and $MemQRd$

Strong Fairness for $RdMiss(p)$ and $DoWr(p)$.

As before, let's define $vars$ to be the tuple of all variables.

$$vars \triangleq \langle memInt,\ wmem,\ buf,\ ctl,\ cache,\ memQ \rangle$$

We could just write the liveness condition as

$$\begin{aligned}(8.34)\ \land\ \forall\, p \in Proc\ :\ &\land\ \mathrm{WF}_{vars}(Rsp(p))\ \land\ \mathrm{WF}_{vars}(DoRd(p)) \\ &\land\ \mathrm{SF}_{vars}(RdMiss(p))\ \land\ \mathrm{SF}_{vars}(DoWr(p)) \\ \land\ &\mathrm{WF}_{vars}(MemQWr)\ \land\ \mathrm{WF}_{vars}(MemQRd)\end{aligned}$$

However, I prefer replacing the conjunction of fairness conditions by a single fairness condition on a disjunction, as we did in Section 8.5 for the memory specification. The WF and SF Conjunction Rules (pages 105 and 106) imply that the liveness condition (8.34) can be rewritten as

$$\begin{aligned}(8.35)\ \land\ \forall\, p \in Proc\ :\ &\land\ \mathrm{WF}_{vars}(Rsp(p) \lor DoRd(p)) \\ &\land\ \mathrm{SF}_{vars}(RdMiss(p) \lor DoWr(p)) \\ \land\ &\mathrm{WF}_{vars}(MemQWr \lor MemQRd)\end{aligned}$$

We can now try to simplify (8.35) by moving the quantifier inside the WF and SF formulas. First, because $\forall$ distributes over $\land$, we can rewrite the first conjunct of (8.35) as

$$\begin{aligned}(8.36)\ \land\ &\forall\, p \in Proc\ :\ \mathrm{WF}_{vars}(Rsp(p) \lor DoRd(p)) \\ \land\ &\forall\, p \in Proc\ :\ \mathrm{SF}_{vars}(RdMiss(p) \lor DoWr(p))\end{aligned}$$

We can now try to apply the WF Quantifier Rule (page 105) to the first conjunct of (8.36) and the corresponding SF Quantifier Rule to its second conjunct. However, the WF quantifier rule doesn't apply to the first conjunct. It's possible for both $Rsp(p) \lor DoRd(p)$ and $Rsp(q) \lor DoRd(q)$ to be enabled at the same time, for two different processors p and q. The formula

(8.37) $\mathrm{WF}_{vars}(\exists\, p \in Proc \,:\, Rsp(p) \lor DoRd(p))$

is satisfied by any behavior in which infinitely many $Rsp(p)$ and $DoRd(p)$ actions occur for some processor p. In such a behavior, $Rsp(q)$ could be enabled for some other processor q without an $Rsp(q)$ step ever occurring, making $\mathrm{WF}_{vars}(Rsp(q) \lor DoRd(q))$ false, which implies that the first conjunct of (8.36) is false. Hence, (8.37) is not equivalent to the first conjunct of (8.36). Similarly, the analogous rule for strong fairness cannot be applied to the second conjunct of (8.36). Formula (8.35) is as simple as we can make it.

Let's return to the observation that we don't have to execute $MemQWr$ if the $memQ$ queue contains only write requests and is not full. In other words, we have to execute $MemQWr$ only if $memQ$ is full or contains a read request. Let's define

$$\begin{aligned} QCond \;\triangleq\; & \lor\; Len(memQ) = QLen \\ & \lor\; \exists\, i \in 1 \,..\, Len(memQ) \,:\, memQ[i][2].op = \text{“Rd”} \end{aligned}$$

so we need eventually execute a $MemQWr$ action only when it's enabled and $QCond$ is true, which is the case iff the action $QCond \land MemQWr$ is enabled. In this case, a $MemQWr$ step is a $QCond \land MemQWr$ step. Hence, it suffices to require weak fairness of the action $QCond \land MemQWr$. We can therefore replace the second conjunct of (8.35) with

$$\mathrm{WF}_{vars}((QCond \land MemQWr) \lor MemQRd)$$

We would do this if we wanted the specification to describe the weakest liveness condition that implements the memory specification's liveness condition. However, if the specification were a description of an actual device, then that device would probably implement weak fairness on all $MemQWr$ actions, so we would take (8.35) as the liveness condition.

8.8 Quantification

Section 8.1 describes the meaning of ordinary quantification of temporal formulas. For example, the meaning of the formula $\forall\, r : F$, for any temporal formula F, is defined by

$$\sigma \models (\forall\, r \,:\, F) \;\triangleq\; \forall\, r \,:\, (\sigma \models F)$$

where σ is any behavior.

The symbol r in $\exists\, r : F$ is usually called a bound variable. But we've been using the term *variable* to mean something else—something that's declared by a VARIABLE statement in a module. The bound "variable" r is actually a constant

in these formulas—a value that is the same in every state of the behavior.[5] For example, the formula $\exists\, r : \Box(x = r)$ asserts that x has the same value in every state of a behavior.

Bounded quantification over a constant set S is defined by

$$\sigma \models (\forall\, r \in S \,:\, F) \;\triangleq\; (\forall\, r \in S \,:\, \sigma \models F)$$
$$\sigma \models (\exists\, r \in S \,:\, F) \;\triangleq\; (\exists\, r \in S \,:\, \sigma \models F)$$

The symbol r is declared to be a constant in formula F. The expression S lies outside the scope of the declaration of r, so the symbol r cannot occur in S. It's easy to define the meanings of these formulas even if S is not a constant—for example, by letting $\exists\, r \in S : F$ equal $\exists\, r : (r \in S) \wedge F$. However, for nonconstant S, it's better to write $\exists\, r : (r \in S) \wedge F$ explicitly.

It's also easy to define the meaning of CHOOSE as a temporal operator. We can just let $\sigma \models (\text{CHOOSE } r : F)$ be an arbitrary constant value r such that $\sigma \models F$ equals TRUE, if such an r exists. However, a temporal CHOOSE operator is not needed for writing specifications, so CHOOSE $r : F$ is not a legal TLA$^+$ formula if F is a temporal formula.

We now come to the temporal existential quantifier $\mathbf{\exists}$. In the formula $\mathbf{\exists}\, x : F$, the symbol x is declared to be a variable in F. Unlike $\exists\, r : F$, which asserts the existence of a single value r, the formula $\mathbf{\exists}\, x : F$ asserts the existence of a value for x in each state of a behavior. For example, if y is a variable, then the formula $\mathbf{\exists}\, x : \Box(x \in y)$ asserts that y always has some element x, so y is always a nonempty set. However, the element x could be different in different states, so the values of y in different states could be disjoint.

We have been using $\mathbf{\exists}$ as a hiding operator, thinking of $\mathbf{\exists}\, x : F$ as F with variable x hidden. The precise definition of $\mathbf{\exists}$ is a bit tricky because, as discussed in Section 8.1, the formula $\mathbf{\exists}\, x : F$ should be invariant under stuttering. Intuitively, $\mathbf{\exists}\, x : F$ is satisfied by a behavior σ iff F is satisfied by a behavior τ that is obtained from σ by adding and/or deleting stuttering steps and changing the value of x. A precise definition appears in Section 16.2.4 (page 314). However, for writing specifications, you can simply think of $\mathbf{\exists}\, x : F$ as F with x hidden.

TLA also has a temporal universal quantifier $\mathbf{\forall}$, defined by

$$\mathbf{\forall}\, x \,:\, F \;\triangleq\; \neg\, \mathbf{\exists}\, x \,:\, \neg F$$

This operator is hardly ever used. TLA$^+$ does not allow bounded versions of the operators $\mathbf{\exists}$ and $\mathbf{\forall}$.

[5] Logicians use the term *flexible variable* for a TLA variable, and the term *rigid variable* for a symbol like r that represents a constant.

8.9 Temporal Logic Examined

8.9.1 A Review

Let's look at the shapes of the specifications that we've written so far. We started with the simple form

(8.38) $Init \wedge \Box[Next]_{vars}$

where *Init* is the initial predicate, *Next* the next-state action, and *vars* the tuple of all variables. This kind of specification is, in principle, quite straightforward. We then introduced hiding, using $\exists$ to bind variables that should not appear in the specification. Those bound variables, also called *hidden* or *internal* variables, serve only to help describe how the values of the free variables (also called visible variables) change.

Hiding variables is easy enough, and it is mathematically elegant and philosophically satisfying. However, in practice, it doesn't make much difference to a specification. A comment can also tell a reader that a variable should be regarded as internal. Explicit hiding allows implementation to mean implication. A lower-level specification that describes an implementation can be expected to imply a higher-level specification only if the higher-level specification's internal variables, whose values don't really matter, are explicitly hidden. Otherwise, implementation means implementation under a refinement mapping. (See Section 5.8.) However, as explained in Section 10.8 below, implementation often involves a refinement of the visible variables as well.

To express liveness, the specification (8.38) is strengthened to the form

(8.39) $Init \wedge \Box[Next]_{vars} \wedge Liveness$

where *Liveness* is the conjunction of formulas of the form $\mathrm{WF}_{vars}(A)$ and/or $\mathrm{SF}_{vars}(A)$, for actions A. (I'm considering universal quantification to be a form of conjunction.)

8.9.2 Machine Closure

In the specifications of the form (8.39) we've written so far, the actions A whose fairness properties appear in formula *Liveness* have one thing in common: they are all *subactions* of the next-state action *Next*. An action A is a subaction of *Next* iff every A step is a *Next* step. Equivalently, A is a subaction of *Next* iff A implies *Next*.[6] In almost all specifications of the form (8.39), formula *Liveness*

[6] We can also use the following weaker definition of subaction: A is a subaction of formula (8.38) iff, for every state s of every behavior satisfying (8.38), if A is enabled in state s then $Next \wedge A$ is also enabled in s.

should be the conjunction of weak and/or strong fairness formulas for subactions of *Next*. I'll now explain why.

When we look at the specification (8.39), we expect *Init* to constrain the initial state, *Next* to constrain what steps may occur, and *Liveness* to describe only what must eventually happen. However, consider the following formula:

$$(8.40)\ \ (x = 0) \land \Box[x' = x + 1]_x \land \mathrm{WF}_x((x > 99) \land (x' = x - 1))$$

The first two conjuncts of (8.40) assert that x is initially 0 and that any step either increments x by 1 or leaves it unchanged. Hence, they imply that if x ever exceeds 99, then it forever remains greater than 99. The weak fairness property asserts that, if this happens, then x must eventually be decremented by 1—contradicting the second conjunct. Hence, (8.40) implies that x can never exceed 99, so it is equivalent to

$$(x = 0) \land \Box[(x < 99) \land (x' = x + 1)]_x$$

Conjoining the weak fairness property to the first two conjuncts of (8.40) forbids an $x' = x + 1$ step when $x = 99$.

A specification of the form (8.39) is called *machine closed* iff the conjunct *Liveness* constrains neither the initial state nor what steps may occur. A more general way to express this is as follows. Let a finite behavior be a finite sequence of states.[7] We say that a finite behavior σ satisfies a safety property S iff the behavior obtained by adding infinitely many stuttering steps to the end of σ satisfies S. If S is a safety property, then we define the pair of formulas S, L to be machine closed iff every finite behavior that satisfies S can be extended to an infinite behavior that satisfies $S \land L$. We call (8.39) machine closed if the pair of formulas $\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$, *Liveness* is machine closed.

We seldom want to write a specification that isn't machine closed. If we do write one, it's usually by mistake. Specification (8.39) is guaranteed to be machine closed if *Liveness* is the conjunction of weak and/or strong fairness properties for subactions of *Next*.[8] This condition doesn't hold for specification (8.40), which is not machine closed, because $(x > 99) \land (x' = x - 1)$ is not a subaction of $x' = x + 1$.

Liveness requirements are philosophically satisfying. A specification of the form (8.38), which specifies only a safety property, allows behaviors in which the system does nothing. Therefore, the specification is satisfied by a system that does nothing. Expressing liveness requirements with fairness properties is less satisfying. These properties are subtle and it's easy to get them wrong.

[7] A finite behavior therefore isn't a behavior, which is an infinite sequence of states. Mathematicians often abuse language in this way.

[8] More precisely, this is the case for a finite or countably infinite conjunction of properties of the form $\mathrm{WF}_v(A)$ and/or $\mathrm{SF}_v(A)$, where each $\langle A \rangle_v$ is a subaction of *Next*. This result also holds for the weaker definition of subaction in the footnote on the preceding page.

It requires some thought to determine that the liveness condition for the write-through cache, formula (8.35) on page 108, does imply that every request receives a reply.

It's tempting to express liveness properties more directly, without using fairness properties. For example, it's easy to write a temporal formula asserting for the write-through cache that every request receives a response. When processor p issues a request, it sets $ctl[p]$ to "rdy". We just have to assert that, for every processor p, whenever a state in which $ctl[p] = \text{“rdy”}$ is true occurs, there will eventually be a $Rsp(p)$ step:

$$(8.41)\ \forall\, p \in Proc : \Box((ctl[p] = \text{“rdy”}) \Rightarrow \Diamond\langle Rsp(p)\rangle_{vars})$$

While such formulas are appealing, they are dangerous. It's very easy to make a mistake and write a specification that isn't machine closed.

Except in unusual circumstances, you should express liveness with fairness properties for subactions of the next-state action. These are the most straightforward specifications, and hence the easiest to write and to understand. Most system specifications, even if very detailed and complicated, can be written in this straightforward manner. The exceptions are usually in the realm of subtle, high-level specifications that attempt to be very general. An example of such a specification appears in Section 11.2.

8.9.3 Machine Closure and Possibility

Machine closure can be thought of as a possibility condition. For example, machine closure of the pair S, $\Box\Diamond\langle A\rangle_v$ asserts that for every finite behavior σ satisfying S, it is possible to extend σ to an infinite behavior satisfying S in which infinitely many $\langle A\rangle_v$ actions occur. If we regard S as a system specification, so a behavior that satisfies S represents a possible execution of the system, then we can restate machine closure of S, $\Box\Diamond\langle A\rangle_v$ as follows: in any system execution, it is always possible for infinitely many $\langle A\rangle_v$ actions to occur.

TLA specifications express safety and liveness properties, not possibility properties. A safety property asserts that something is impossible—for example, the system cannot take a step that doesn't satisfy the next-state action. A liveness property asserts that something must eventually happen. System requirements are sometimes stated informally in terms of what is possible. Most of the time, when examined rigorously, these requirements can be expressed with liveness and/or safety properties. (The most notable exceptions are statistical properties, such as assertions about the probability that something happens.) We are never interested in specifying that something *might* happen. It's never useful to know that the system *might* produce the right answer. We never have to specify that the user *might* type an "a"; we must specify what happens if he does.

Machine closure is a property of a pair of formulas, not of a system. Although a possibility property is never a useful assertion about a system, it can be a useful assertion about a specification. A specification S of a system with keyboard input should always allow the user to type an "a". So, every finite behavior satisfying S should be extendable to an infinite behavior satisfying S in which infinitely many "a"s are typed. If the action $\langle A \rangle_v$ represents the typing of an "a", then saying that the user should always be able to type infinitely many "a"s is equivalent to saying that the pair S, $\Box\Diamond\langle A \rangle_v$ should be machine closed. If S, $\Box\Diamond\langle A \rangle_v$ isn't machine closed, then it could become impossible for the user ever to type an "a". Unless the system is allowed to lock the keyboard, this would mean that there was something wrong with the specification.

This kind of possibility property can be proved. For example, to prove that it's always possible for the user to type infinitely many "a"s, we show that conjoining suitable fairness conditions on the input actions implies that the user *must* type infinitely many "a"s. However, proofs of this kind of simple property don't seem to be worth the effort. When writing a specification, you should make sure that possibilities allowed by the real system are allowed by the specification. Once you are aware of what should be possible, you will usually have little trouble ensuring that the specification makes it possible. You should also make sure that what the system *must* do is implied by the specification's fairness conditions. This can be more difficult.

8.9.4 Refinement Mappings and Fairness

Section 5.8 (page 62) describes how to prove that the write-through memory implements the memory specification. We have to prove $Spec \Rightarrow \overline{ISpec}$, where *Spec* is the specification of the write-through memory, *ISpec* is the internal specification of the memory (with the internal variables made visible), and, for any formula F, we let $\overline{F}$ mean F with expressions *omem*, *octl*, and *obuf* substituted for the variables *mem*, *ctl*, and *buf*. We could rewrite this implication as (5.3) because substitution (overbarring) distributes over operators like $\wedge$ and $\Box$, so we had

$$
\begin{array}{lll}
\overline{IInit \wedge \Box[INext]_{\langle memInt,\, mem,\, ctl,\, buf \rangle}} & & \\
\quad \equiv \overline{IInit} \wedge \overline{\Box[INext]_{\langle memInt,\, mem,\, ctl,\, buf \rangle}} & & \text{Because } \bar{} \text{ distributes over } \wedge. \\
\quad \equiv \overline{IInit} \wedge \Box[\overline{INext}]_{\overline{\langle memInt,\, mem,\, ctl,\, buf \rangle}} & & \text{Because } \bar{} \text{ distributes over } \Box[\cdots]_{\ldots}. \\
\quad \equiv \overline{IInit} \wedge \Box[\overline{INext}]_{\langle \overline{memInt},\, \overline{mem},\, \overline{ctl},\, \overline{buf} \rangle} & & \text{Because } \bar{} \text{ distributes over } \langle \ldots \rangle. \\
\quad \equiv \overline{IInit} \wedge \Box[\overline{INext}]_{\langle memInt,\, \overline{mem},\, \overline{ctl},\, \overline{buf} \rangle} & & \text{Because } \overline{memInt} = memInt.
\end{array}
$$

Adding liveness to the specifications adds conjuncts to the formulas *Spec* and *ISpec*. Suppose we take formula *Liveness*2, defined in (8.19) on page 101, as

the liveness property of *ISpec*. Then $\overline{ISpec}$ has the additional term $\overline{Liveness2}$, which can be simplified as follows:

$$\begin{array}{ll}
\overline{Liveness2} & \\
\quad \equiv \overline{\forall\, p \in Proc : \mathrm{WF}_{vars}(Do(p) \lor Rsp(p))} & \text{By definition of } Liveness2. \\
\quad \equiv \forall\, p \in Proc : \overline{\mathrm{WF}_{vars}(Do(p) \lor Rsp(p))} & \text{Because } \bar{} \text{ distributes over } \forall.
\end{array}$$

But we cannot automatically move the $\bar{}$ inside the WF because substitution does not, in general, distribute over ENABLED, and hence it does not distribute over WF or SF. For the specifications and refinement mappings that occur in practice, including this one, simply replacing each $\overline{\mathrm{WF}_v(A)}$ by $\mathrm{WF}_{\overline{v}}(\overline{A})$ and each $\overline{\mathrm{SF}_v(A)}$ by $\mathrm{SF}_{\overline{v}}(\overline{A})$ does give the right result. However, you don't have to depend on this. You can instead expand the definitions of WF and SF to get, for example:

$$\begin{array}{lll}
\overline{\mathrm{WF}_v(A)} & \equiv \overline{\Box\Diamond\neg \text{ENABLED } \langle A \rangle_v \lor \Box\Diamond\langle A \rangle_v} & \text{By definition of WF.} \\
& \equiv \Box\Diamond\neg\overline{\text{ENABLED } \langle A \rangle_v} \lor \Box\Diamond\langle \overline{A} \rangle_{\overline{v}} & \text{By distributivity of } \bar{}.
\end{array}$$

You can compute the ENABLED predicates "by hand" and then perform the substitution. When computing ENABLED predicates, it suffices to consider only states satisfying the safety part of the specification, which usually means that ENABLED $\langle A \rangle_v$ equals ENABLED A. You can then compute ENABLED predicates using the following rules:

1. ENABLED $(A \lor B) \equiv (\text{ENABLED } A) \lor (\text{ENABLED } B)$, for any actions A and B.
2. ENABLED $(P \land A) \equiv P \land (\text{ENABLED } A)$, for any state predicate P and action A.
3. ENABLED $(A \land B) \equiv (\text{ENABLED } A) \land (\text{ENABLED } B)$, if A and B are actions such that the same variable does not appear primed in both A and B.
4. ENABLED $(x' = exp) \equiv$ TRUE and ENABLED $(x' \in exp) \equiv (exp \neq \{\})$, for any variable x and state function exp.

For example:

$$\begin{array}{ll}
\overline{\text{ENABLED } (Do(p) \lor Rsp(p))} & \\
\quad \equiv \overline{(ctl[p] = \text{“rdy”}) \lor (ctl[p] = \text{“done”})} & \text{By rules 1–4.} \\
\quad \equiv (octl[p] = \text{“rdy”}) \lor (octl[p] = \text{“done”}) & \text{By the meaning of } \bar{}.
\end{array}$$

8.9.5 The Unimportance of Liveness

While philosophically important, in practice the liveness property of (8.39) is not as important as the safety part, $Init \wedge \Box[Next]_{vars}$. The ultimate purpose of writing a specification is to avoid errors. Experience shows that most of the benefit from writing and using a specification comes from the safety part. On the other hand, the liveness property is usually easy enough to write. It typically constitutes less than five percent of a specification. So, you might as well write the liveness part. However, when looking for errors, most of your effort should be devoted to examining the safety part.

8.9.6 Temporal Logic Considered Confusing

The most general type of specification I've discussed so far has the form

(8.42) $\exists\, v_1, \ldots, v_n \,:\, Init \wedge \Box[Next]_{vars} \wedge Liveness$

where *Liveness* is the conjunction of fairness properties of subactions of *Next*. This is a very restricted class of temporal-logic formulas. Temporal logic is quite expressive, and one can combine its operators in all sorts of ways to express a wide variety of properties. This suggests the following approach to writing a specification: express each property that the system must satisfy with a temporal formula, and then conjoin all these formulas. For example, formula (8.41) above expresses the property of the write-through cache that every request eventually receives a response.

This approach is philosophically appealing. It has just one problem: it's practical for only the very simplest of specifications—and even for them, it seldom works well. The unbridled use of temporal logic produces formulas that are hard to understand. Conjoining several of these formulas produces a specification that is impossible to understand.

The basic form of a TLA specification is (8.42). Most specifications should have this form. We can also use this kind of specification as a building block. Chapters 9 and 10 describe situations in which we write a specification as a conjunction of such formulas. Section 10.7 introduces an additional temporal operator $\stackrel{+}{\triangleright}$ and explains why we might want to write a specification $F \stackrel{+}{\triangleright} G$, where F and G have the form (8.42). But such specifications are of limited practical use. Most engineers need only know how to write specifications of the form (8.42). Indeed, they can get along quite well with specifications of the form (8.38) that express only safety properties and don't hide any variables.