5

Liveness Requirements

As discussed in chapter 3, requirements can be classified into two broad categories: *safety* requirements assert that "nothing bad ever happens," and *liveness* requirements assert that "something good eventually happens." For instance, in the leader election problem, the central safety requirement is that no two nodes should ever declare themselves to be the leaders, and the central liveness requirement is that each node should eventually make a decision. In chapter 3, we studied how to specify and verify safety requirements. Now we turn our attention to liveness requirements. Such requirements are specified using a formalism called *temporal logic*. The problem of checking whether a model satisfies its specification expressed in temporal logic is known as *model checking*.

5.1 Temporal Logic

Let us revisit our example of the system of traffic lights for a railroad from section 3.1.2. Given a model of the trains and the desired requirements, the design problem is to construct a controller so that the system composed of the trains and the controller satisfies the requirements. One basic requirement is that the two trains should not be on the bridge simultaneously. This safety requirement is captured by the property

$$\texttt{TrainSafety}: \neg[(mode_W = \texttt{bridge}) \wedge (mode_E = \texttt{bridge})]$$

and we require this property to be an invariant of the composite system. Obviously, this is not a *complete* specification for the desired controller: a controller that keeps both traffic lights to be red all the time is safe with respect to the property `TrainSafety` but is not an acceptable solution as it would never let any train onto the bridge. We need to augment the safety requirement with a liveness requirement that asserts that the controller should allow the trains onto the bridge. For resource allocation problems exemplified by our railroad system, while there is usually a canonical safety requirement, the liveness requirements can make differing demands. For instance, we may require that "if one of the

trains wants to enter the bridge, then eventually some train should be allowed to enter," or we may demand a stronger requirement that "if a train wants to enter the bridge, then eventually that specific train should be allowed to enter."

Violation of a safety requirement is demonstrated by a finite execution that leads the system from an initial state to an erroneous state. For instance, the counterexample of figure 3.7 is a finite execution demonstrating that our first attempt at designing the controller for the railroad system is incorrect. Violation of a liveness requirement, in contrast, is not exhibited by such a finite execution. Instead, it consists of a cycle of states such that the cycle is reachable from an initial state, and if the cycle is executed repeatedly, the demand made by the liveness requirement is unmet. Hence, the mathematical formalization of liveness requirements considers infinite executions of the system. A natural formalism for specifying requirements about infinite executions is *temporal logic*. Temporal logics come in many varieties and can express safety as well as liveness requirements. We will study the classical temporal logic LTL, which stands for Linear Temporal Logic. This logic forms the core of the *Property Specification Language* (PSL), which has been standardized by IEEE and supported by commercial simulation and verification tools used in the electronic design automation industry.

5.1.1 Linear Temporal Logic

Let V be a set of typed variables, and suppose we are writing requirements to constrain the values these variables are allowed to take. Given a valuation q over V, that is, a type-consistent assignment of values to V and a Boolean expression e over V, $q(e)$ denotes the result of evaluating the expression e using the values assigned by the valuation q. When $q(e)$ equals 1, we say that the valuation q *satisfies* the expression e. We can thus interpret a Boolean expression e to express a constraint or requirement on individual valuations: the requirement e is satisfied when the expression evaluates to 1 according to the values assigned by the valuation. For example, suppose the set V contains two Boolean variables x and y. Then the expression $(x = y)$ expresses the requirement that both these variables should take the same value: a valuation q satisfies the requirement precisely when the value $q(x)$ is same as the value $q(y)$.

While an expression over variables V is evaluated with respect to a valuation for V, a temporal logic formula over V is evaluated with respect to an *infinite* sequence of valuations. That is, to interpret a temporal logic formula, we need to consider an infinite sequence $q_1 q_2 \ldots$ where each element q_i of the sequence is a valuation. For example, when the set V contains two Boolean variables x and y, each valuation is an assignment to the Boolean variables x and y, and the temporal logic formulas are evaluated with respect to an infinite sequence $\rho = (x_1, y_1)(x_2, y_2)\cdots$. We call such an infinite sequence of valuations a *trace* over V.

Boolean expressions are used to express constraints over individual valuations,

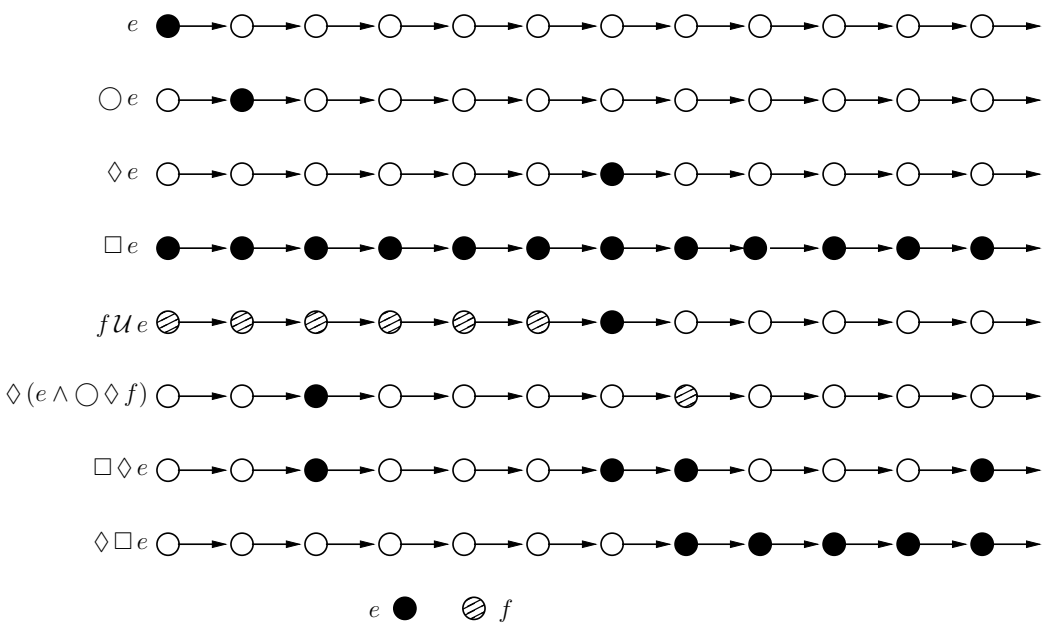

Figure 5.1: Illustrating Temporal Operators of LTL

and such expressions are combined using temporal operators that capture requirements for the sequence of valuations in a trace. Thus, a Boolean expression e over V is the simplest form of a temporal logic formula. We say that a trace ρ satisfies the Boolean expression e if the *first* valuation in the trace satisfies e. In our example, the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies the LTL-formula $(x = y)$ precisely when the first valuation in the trace satisfies the expression, that is, when x_1 equals y_1.

Temporal Operators

Let us consider the temporal operator *always*, denoted $\Box$. The LTL-formula $\Box e$ is satisfied by a trace when every valuation in the trace satisfies e. For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies the LTL-formula $\Box(x = y)$ precisely when *every* valuation in the trace satisfies the expression $(x = y)$, that is, when $x_j = y_j$ for every j. Thus, the formula $\Box(x = y)$ expresses the requirement that the variable x should always be equal to y. Figure 5.1 illustrates the requirements imposed by different temporal logic formulas.

The dual of the *always* operator is the temporal operator *eventually*, denoted $\Diamond$. The LTL-formula $\Diamond e$ is satisfied by a trace when *some* valuation in the trace satisfies e. For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies the LTL-formula $\Diamond(x = y)$ precisely when some valuation in the trace satisfies the expression $(x = y)$, that is, when $x_j = y_j$ for some j. Thus, the formula

$\Diamond\,(x = y)$ expresses the requirement that eventually at some step, the values of the variables x and y coincide.

The temporal operator *next*, denoted $\bigcirc$, is used to assert requirements for the *next* valuation in the trace. The LTL-formula $\bigcirc\, e$ is satisfied by the trace $q_1 q_2 \ldots$ when the valuation q_2 satisfies the expression e. For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies the LTL-formula $\bigcirc\,(x = y)$ precisely when $x_2 = y_2$.

The final temporal operator we consider is called the *until* operator, denoted $\mathcal{U}$, that takes two formulas as arguments. The LTL-formula $f\,\mathcal{U}\,e$ is satisfied in a trace if the expression f is satisfied in every valuation in the sequence until we encounter a valuation that satisfies e. That is, the trace $q_1 q_2 \ldots$ satisfies the formula $f\,\mathcal{U}\,e$ precisely when there exists a position j such that the valuation q_j satisfies the expression e and each of the valuations from q_1 to q_{j-1} satisfies the expression f. For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies the LTL-formula $(x = 0)\,\mathcal{U}\,(y = 1)$ precisely when there is some position j such that $y_j = 1$ and $x_k = 0$ for all positions $1 \leq k < j$. This expresses the requirement that eventually y should become 1, and until then x should stay 0.

Temporal logic formulas can be combined using the standard logical operators: conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and negation ($\neg$). For example, if φ_1 and φ_2 are two LTL-formulas, then we can combine them using the conjunction operator to obtain the LTL-formula $\varphi_1 \wedge \varphi_2$. A trace ρ satisfies the conjunction $\varphi_1 \wedge \varphi_2$ precisely when it satisfies both φ_1 and φ_2. Thus, a trace ρ satisfies the LTL-formula $\Box\,(x = y) \wedge \Diamond\,(x = 0)$ if in every valuation of the trace, the value of x is equal to the value of y, and there is a valuation in the trace which assigns the value 0 to x.

So far we have considered LTL-formulas in which the arguments to the temporal operators are expressions constraining individual valuations. In general, temporal operators can be nested, that is, arguments of temporal operators may be complex temporal logic formulas. For example, consider the LTL-formula $\bigcirc\,\Box\,(x = y)$. This formula says that in the next step, always $(x = y)$ holds: the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies this formula at the first position precisely when it satisfies the formula $\Box\,(x = y)$ at position 2, that is, $x_j = y_j$ for all positions $j \geq 2$.

To formalize the meaning of LTL-formulas with nested temporal operators, we will define what it means for a trace to satisfy an LTL-formula at a given position: for a trace ρ, a position $j \geq 1$, and an LTL-formula φ, the notation $(\rho, j) \models \varphi$ stands for "the trace ρ satisfies the formula φ at position j." The trace $\rho = q_1 q_2 \ldots$ satisfies a Boolean expression e (without any temporal operators) at position j if the valuation q_j satisfies e. The trace ρ satisfies the next formula $\bigcirc\,\varphi$ at position j, where φ may be an arbitrary LTL-formula, if the trace ρ satisfies the formula φ at position $j + 1$. That is, "next φ" holds at a position if φ holds at the next position. Similarly, the trace ρ satisfies the eventually formula $\Diamond\,\varphi$ at position j, where φ may be an arbitrary LTL-formula, if the trace

ρ satisfies the formula φ at some position k with $k \geq j$. That is, "eventually φ" holds at a position if φ holds at some later (or future) position. Similarly, "always φ" holds at a position if φ holds at every subsequent position.

Syntax and Semantics

Now we can define the logic LTL precisely. The definition below defines both the *syntax* of the logic—what are the syntactically correct formulas of the logic and the *semantics* of the logic—the meaning of the formulas given by the rules to evaluate the formulas over traces. The definition is inductive; for instance, it describes the rule for evaluating the formula $\Box\,\varphi$ assuming we have already defined how to evaluate the simpler formula φ.

LINEAR TEMPORAL LOGIC

Given a set V of typed variables, the set of formulas of *linear temporal logic* (LTL) is defined inductively by the rules below:

- If e is a Boolean expression over V, then e is an LTL-formula.
- If φ is an LTL-formula, then so are $\neg\,\varphi$, $\bigcirc\,\varphi$, $\Diamond\,\varphi$, and $\Box\,\varphi$.
- If φ_1 and φ_2 are LTL-formulas, then so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\varphi_1\,\mathcal{U}\,\varphi_2$.

Given a trace $\rho = q_1 q_2 \ldots$ (that is, an infinite sequence of valuations over V), a position $j \geq 1$, and an LTL-formula φ, the *satisfaction* relation, $(\rho, j) \models \varphi$, meaning that the trace ρ satisfies the LTL-formula φ at position j, is defined inductively by the following rules:

- $(\rho, j) \models e$ if the valuation q_j satisfies the Boolean expression e.
- $(\rho, j) \models \neg\,\varphi$ if it is not the case that $(\rho, j) \models \varphi$.
- $(\rho, j) \models \varphi_1 \wedge \varphi_2$ if both $(\rho, j) \models \varphi_1$ and $(\rho, j) \models \varphi_2$.
- $(\rho, j) \models \varphi_1 \vee \varphi_2$ if either $(\rho, j) \models \varphi_1$ or $(\rho, j) \models \varphi_2$.
- $(\rho, j) \models \varphi_1 \rightarrow \varphi_2$ if either $(\rho, j) \models \neg\,\varphi_1$ or $(\rho, j) \models \varphi_2$.
- $(\rho, j) \models \bigcirc\,\varphi$ if $(\rho, j+1) \models \varphi$.
- $(\rho, j) \models \Box\,\varphi$ if for every position $k \geq j$, $(\rho, k) \models \varphi$.
- $(\rho, j) \models \Diamond\,\varphi$ if for some position $k \geq j$, $(\rho, k) \models \varphi$.
- $(\rho, j) \models \varphi_1\,\mathcal{U}\,\varphi_2$ if for some position $k \geq j$, $(\rho, k) \models \varphi_2$, and for all positions i such that $j \leq i < k$, $(\rho, i) \models \varphi_1$.

The trace ρ satisfies the LTL-formula φ if $(\rho, 1) \models \varphi$.

Notice that the satisfaction of a formula at a position j of a trace $\rho = q_1 q_2 \ldots$

depends only on the *suffix* of the trace starting at position j, that is, on the the sequence of valuations $q_j q_{j+1} \cdots$. This is because all the temporal operators refer to the *future* positions. It is worth emphasizing that the current position is considered part of the future: satisfaction of "eventually φ" at a position j demands φ to be satisfied at some position $k \geq j$. Thus, if a formula φ is satisfied at a position, then so is "eventually φ" satisfied at that position. Also note that for the until-formula $\varphi_1 \mathcal{U} \varphi_2$ to be satisfied at a position j, we demand that the formula φ_2 is satisfied at some position $k \geq j$, and the formula φ_1 holds at all positions following j, including j itself, and *strictly preceding* k. The eventuality operator is just a special case of the until-operator: $\Diamond \varphi$ has the same meaning as the until-formula $1 \mathcal{U} \varphi$, where 1 is the Boolean constant that is satisfied by every valuation.

Illustrative Temporal Patterns

Nesting of temporal operators give interesting and useful formulas. We highlight some typical patterns (see figure 5.1).

Sequencing: Nested applications of eventually operators can be used to require a sequence of events in a particular order. For instance, consider two events that correspond to satisfaction of formulas φ_1 and φ_2. Then a trace satisfies the LTL-formula $\Diamond (\varphi_1 \wedge \bigcirc \Diamond \varphi_2)$ if there are two positions i and j with $i < j$, such that the formula φ_1 is satisfied at position i and the formula φ_2 is satisfied at position j. For example, the trace $\rho = (x_1, y_1)(x_2, y_2) \cdots$ satisfies the LTL-formula $\Diamond ((x = 1) \wedge \bigcirc \Diamond (y = 1))$ precisely when we can find two positions i and j such that $i < j$ and $x_i = 1$ and $y_j = 1$. Note that the use of the next operator requires the two events to occur at *distinct* positions. The modified formula $\Diamond (\varphi_1 \wedge \Diamond \varphi_2)$ is satisfied by a trace if there are two positions i and j with $i \leq j$, such that the formula φ_1 is satisfied at position i and the formula φ_2 is satisfied at position j.

Recurrence Formulas: Consider the *always-eventually* formula $\Box \Diamond \varphi$. A trace ρ satisfies the formula $\Box \Diamond \varphi$ at the initial position if $\Diamond \varphi$ is satisfied at every position i. This condition holds if for every position i there exists a future position $j \geq i$ such that the trace ρ satisfies φ at position j. With a little bit of reasoning, convince yourself that this condition can be reformulated as: there exists an infinite sequence of positions $j_1 < j_2 < j_3 < \cdots$ such that φ is satisfied at each of these positions. In other words, $\Box \Diamond \varphi$ is satisfied if φ is satisfied in a *recurrent* or repeating manner. For example, the trace $\rho = (x_1, y_1)(x_2, y_2) \cdots$ satisfies the recurrence formula $\Box \Diamond (x = 0)$ precisely when for infinitely many positions j, $x_j = 0$. This expresses the requirement that x is assigned the value 0 repeatedly.

Persistence Formulas: The dual of the recurrence requirement expressed by the always-eventually formula is the *eventually-always* formula $\Diamond \Box \varphi$. It is satisfied if there exists a position j where the always-formula $\Box \varphi$ is satisfied, that is, φ is satisfied in every position following j. In other words, the requirement is

that eventually the formula φ is satisfied and continues to hold in a persistent manner. For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\cdots$ satisfies the persistence formula $\Diamond\Box\,(x = 0)$ precisely when for some position j, for every $k \geq j$, $x_k = 0$ (equivalently, if x is non-zero only at finitely many positions).

Let us consider another example to understand nested temporal formulas. Suppose there is a single variable x of type `nat`. Consider the expressions $\texttt{even}(x)$, $\texttt{odd}(x)$, and $\texttt{prime}(x)$ that are satisfied when the value of x is an even number, an odd number, and a prime number, respectively. Consider the trace $\rho = 1, 2, 3, \cdots$; that is, the jth valuation in the trace assigns the value j to the variable x. Then the trace ρ satisfies the following formula

$$\Box\ [\,\texttt{even}(x)\ \rightarrow\ (\bigcirc\ \texttt{odd}(x)\ \wedge\ \bigcirc\bigcirc\ \texttt{even}(x)\,)\,]$$

which asserts that at every position, if x is even, then in the next position, x is odd, and in the next-to-next position, x is even. The trace ρ also satisfies the recurrence formula

$$\Box\Diamond\ \texttt{prime}(x)$$

which asserts that the trace contains infinitely many prime numbers.

As another example, suppose there is a single variable x of type `real`, and consider the trace $\rho = 1, 1/2, 1/4, 1/8, \cdots$ (that is, the jth valuation in the trace assigns the value 2^{-j} to the variable x. Then the trace ρ does *not* satisfy the eventuality formula $\Diamond\,(x = 0)$ but for every $\epsilon > 0$, however small, satisfies the persistence formula $\Diamond\Box\,(x \leq \epsilon)$.

Temporal Implications and Equivalences

An LTL-formula φ is said to be *valid* if every trace ρ satisfies φ. A valid LTL-formula is also called a *temporal tautology*: it holds no matter how we choose to assign values to the variables at every step. For two LTL-formulas φ_1 and φ_2, if the implication $\varphi_1 \rightarrow \varphi_2$ is valid, then whenever a trace ρ satisfies the formula φ_1, we are guaranteed that the trace ρ also satisfies the formula φ_2. In such a case, the requirement expressed by φ_1 is *stronger* than the requirement expressed by φ_2 since the satisfaction of one implies the satisfaction of the other. Two LTL-formulas φ_1 and φ_2 are *equivalent*, written $\varphi_1 \leftrightarrow \varphi_2$, if both the implications $\varphi_1 \rightarrow \varphi_2$ and $\varphi_2 \rightarrow \varphi_1$ are valid. For two equivalent formulas, a trace satisfies either both or none of them.

Let us consider a few valid implications and equivalences aimed at gaining a better intuition about the meaning of temporal operators.

If a trace ρ satisfies the LTL-formula $\Box\varphi$, then it satisfies φ at all positions, in particular at the initial position, and thus ρ satisfies φ. This argument is independent of the choice of φ. Thus, for every LTL-formula φ, the implication $\Box\varphi \rightarrow \varphi$ is valid, that is, the formula $\Box\varphi$ is a stronger requirement than the formula φ itself. The converse is not true: it is easy to find instances where

a trace satisfies an LTL-formula φ but not the formula $\Box\varphi$. The following equivalence, however, is valid for every LTL-formula φ:

$$\Box\varphi \;\leftrightarrow\; [\varphi \wedge \bigcirc\Box\varphi].$$

It says that the always-formula $\Box\varphi$ is satisfied precisely when the current position satisfies φ and the next position satisfies the always-formula $\Box\varphi$. This equivalence can be viewed as an "inductive" definition of the always operator. Similar inductive definitions of the eventually and until operators can be obtained (see exercise 5.1).

Observe that if a trace satisfies the recurrence formula $\Box\Diamond\varphi$ at a particular position, say the first position, then it satisfies the same recurrence formula $\Box\Diamond\varphi$ at the next position also. The converse also holds. In fact, for every trace ρ, for all positions i and j, $(\rho, i) \models \Box\Diamond\varphi$ if and only if $(\rho, j) \models \Box\Diamond\varphi$. As a result, for any LTL-formula φ, all the following three formulas are equivalent:

$$\Box\Diamond\varphi \;\leftrightarrow\; \bigcirc\Box\Diamond\varphi \;\leftrightarrow\; \Diamond\Box\Diamond\varphi.$$

Laws for Temporal and Logical Operators

The interplay between temporal and logical operators can be understood by considering how the logical and temporal operators distribute with respect to one another. Let φ_1 and φ_2 be two LTL-formulas, and let ρ be a trace. Then the trace ρ satisfies the always-formula $\Box(\varphi_1 \wedge \varphi_2)$ precisely when $(\rho, j) \models (\varphi_1 \wedge \varphi_2)$ for every position j. This holds precisely when $(\rho, j) \models \varphi_1$ and $(\rho, j) \models \varphi_2$ for every position j. This is equivalent to saying that the trace ρ satisfies $\Box\varphi_1$ as well as $\Box\varphi_2$, which holds precisely when the trace ρ satisfies the conjunction $\Box\varphi_1 \wedge \Box\varphi_2$. Thus, we have established that the always-operator distributes over conjunction. Thus, for any two LTL-formulas φ_1 and φ_2, the following equivalence is valid:

$$\Box(\varphi_1 \wedge \varphi_2) \;\leftrightarrow\; (\Box\varphi_1 \wedge \Box\varphi_2).$$

Let us examine if a similar distributivity property holds for disjunction: do the formulas $\Box(\varphi_1 \vee \varphi_2)$ and $(\Box\varphi_1 \vee \Box\varphi_2)$ mean the same? Consider a trace ρ. Suppose ρ satisfies $\Box\varphi_1$. Then for every position j, $(\rho, j) \models \varphi_1$. By the semantics of disjunction, we have that for every position j, $(\rho, j) \models (\varphi_1 \vee \varphi_2)$. It follows that ρ satisfies the always-formula $\Box(\varphi_1 \vee \varphi_2)$. Symmetric reasoning allows us to conclude that if a trace ρ satisfies $\Box\varphi_2$, then it also satisfies $\Box(\varphi_1 \vee \varphi_2)$. A trace satisfies the disjunction $\Box\varphi_1 \vee \Box\varphi_2$ precisely when it satisfies either $\Box\varphi_1$ or $\Box\varphi_2$, and in either case, we have established that it must then satisfy $\Box(\varphi_1 \vee \varphi_2)$. Thus, the following implication is valid:

$$(\,\Box\varphi_1 \vee \Box\varphi_2\,) \;\rightarrow\; \Box(\varphi_1 \vee \varphi_2).$$

However, the converse implication is not valid. If we know that a trace ρ satisfies the always-formula $\Box(\varphi_1 \vee \varphi_2)$, then we know that at each position, either φ_1

or φ_2 is satisfied. But this does not necessarily mean that either all positions satisfy φ_1 or all positions satisfy φ_2. As a concrete counterexample, suppose the valuation assigns a Boolean value to a variable x. Consider the trace $\rho = 010101\cdots$ in which the values 0 and 1 are assigned to x in an alternate manner. This trace satisfies $\Box(x = 0 \vee x = 1)$ but satisfies neither $\Box(x = 0)$ nor $\Box(x = 1)$.

To conclude this section, let us note the interplay between the temporal operators and logical negation. First note that $\neg\Box\varphi$ is equivalent to $\Diamond\neg\varphi$ (and similarly, $\neg\Diamond\varphi$ is equivalent to $\Box\neg\varphi$). This results in a duality between the recurrence and persistence formulas: a property φ is not recurrent precisely when the negated property $\neg\varphi$ is persistent. That is, $\neg\Box\Diamond\varphi$ is equivalent to $\Diamond\Box\neg\varphi$. The dual of the next operator is itself: a trace does not satisfy the next-formula $\bigcirc\varphi$ at a position j, precisely when the trace does not satisfy the formula φ at position $(j+1)$, precisely when the trace satisfies the formula $\bigcirc\neg\varphi$ at position j. Thus, the LTL-formulas $\neg\bigcirc\varphi$ and $\bigcirc\neg\varphi$ are equivalent.

Exercise 5.1: We saw that the always-formula $\Box\varphi$ is equivalent to $\varphi \wedge \bigcirc\Box\varphi$. Find analogous formulas equivalent to the eventually-formula $\Diamond\varphi$ and to the until-formula $\varphi_1 \mathcal{U} \varphi_2$. Justify your answers. ■

Exercise 5.2: For each of the pair of formulas below, say whether the two are equivalent and if not whether one of them is a stronger requirement than the other. In each case, justify your answer.

1. $\Diamond(\varphi_1 \wedge \varphi_2)$ and $(\Diamond\varphi_1 \wedge \Diamond\varphi_2)$.
2. $\Diamond(\varphi_1 \vee \varphi_2)$ and $(\Diamond\varphi_1 \vee \Diamond\varphi_2)$.
3. $\Box\Diamond(\varphi_1 \wedge \varphi_2)$ and $(\Box\Diamond\varphi_1 \wedge \Box\Diamond\varphi_2)$.
4. $\Box\Diamond(\varphi_1 \vee \varphi_2)$ and $(\Box\Diamond\varphi_1 \vee \Box\Diamond\varphi_2)$.

■

Exercise 5.3: Are the LTL-formulas $\neg(\varphi_1 \mathcal{U} \varphi_2)$ and $(\neg\varphi_2)\,\mathcal{U}\,(\neg\varphi_1)$ equivalent? If not, is one of them a stronger requirement than the other? Justify your answer. ■

Exercise 5.4: Are the two LTL formulas $\Box\Diamond(\varphi_1 \wedge \Diamond\varphi_2)$ and $\Box\Diamond(\varphi_2 \wedge \Diamond\varphi_1)$ equivalent? Justify your answer clearly. ■

5.1.2 LTL Specifications

We can use LTL formulas to specify requirements for both synchronous and asynchronous systems. Let us first focus on synchronous systems.

Consider a synchronous reactive component C with input variables I and output variables O. The natural choice of observable variables for such a component

is the set $I \cup O$ of input and output variables. An LTL specification for the component C is an LTL-formula φ over the set $I \cup O$ of observable variables. As the component executes, the infinite sequence of inputs and outputs it produces is a trace of the component. Formally, an infinite execution of the component C consists of an infinite sequence of the form

$$s_0 \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \xrightarrow{i_3/o_3} s_3 \cdots$$

such that s_0 is an initial state of C, and for each $j > 0$, $s_{j-1} \xrightarrow{i_j/o_j} s_j$ is a reaction C. Given such an execution, the infinite sequence $(i_1, o_1)(i_2, o_2)(i_3, o_3) \cdots$ of inputs and outputs is a *trace* of C. The component C satisfies the specification φ if *every* trace of C satisfies φ. An infinite execution is called a counterexample to the specification φ if the corresponding trace does not satisfy φ. The problem of checking whether a component satisfies a temporal logic specification is known as model checking.

For example, our very first component `Delay` (figure 2.1) has input variable *in* and output variable *out*. LTL-formulas over these variables can be used to express constraints over the desired temporal behavior. In particular, consider the specification:

$$\Box\ [\,(in = 0)\ \rightarrow\ \bigcirc\,(out = 0)\,]\ \wedge\ \Box\ [\,(in = 1)\ \rightarrow\ \bigcirc\,(out = 1)\,],$$

which says that at every position of a trace, if the value of *in* is 0, then in the next position the value of *out* is 0, and if the value of *in* is 1, then in the next position the value of *out* is 1. Indeed, every trace of the component `Delay` satisfies this specification, so we will say that the component satisfies the specification.

As another example, consider the component `ClockedCopy` (figure 2.6) with input variables *in* and *clock* and output variable *out*. Consider the following LTL formula:

$$\Box\ [\,(out = 0)\ \rightarrow (\,out = 0\,)\ \mathcal{U}\ clock?\,]\wedge\ \Box\ [\,(out = 1)\ \rightarrow (\,out = 1\,)\ \mathcal{U}\ clock?\,].$$

It says that if the value of the output variable *out* is 0 (or 1) in a given round, then it is guaranteed to stay 0 (or 1, respectively) until the event *clock* is present. It captures the requirement that the output should not change in rounds in which the event *clock* is absent. The component `ClockedCopy` does satisfy this requirement.

Requirements for Leader Election

Let us recall the leader election problem discussed in section 2.4.3. The decision of each node is captured by the output variable *status* that ranges over the enumerated type {`unknown`, `leader`, `follower`}. While the nodes use the variables *in* and *out* for exchanging messages with one another, the design requirements

of the problem specify which traces of values of the *status* variables of different processes are acceptable. The requirement that a node n should eventually make a decision is expressed by the formula

$$\Diamond [\, status_n \neq \texttt{unknown} \,].$$

The formula states that for a given node n, eventually the value of the status variable of the instance of the process `SyncLENode` corresponding to this node should be different from `unknown`. The safety requirement that two nodes should not consider themselves to be leaders can be expressed by the following formula, which states that for every pair of distinct nodes m and n, either m is never a leader or n is never a leader:

$$\Box (\, status_m \neq \texttt{leader} \,) \vee \Box (\, status_n \neq \texttt{leader} \,).$$

Requirements for Railroad Controller

Let us revisit the railroad controller system of section 3.1.2. Let us consider the observable variables of the system to be $signal_W$ and $signal_E$ capturing the traffic lights and the variables $mode_W$ and $mode_E$ capturing the train states. While the latter are not modeled as output variables, it is acceptable to write requirements that refer to the state of the models capturing the environment since the modeling of the environment is part of the specification of the design problem.

When an LTL-formula refers to a state variable x of a component, the valuation at each position of a trace should specify the value for x also: in a trace corresponding to an infinite execution, the value of x at position j is the value of x at the beginning of the jth round.

The basic safety requirement that the two trains should not be on the bridge simultaneously is expressed by the always-formula:

$$\Box \neg [\, (mode_W = \texttt{bridge}) \wedge (mode_E = \texttt{bridge}) \,].$$

Consider the following liveness requirement, which asserts that the west train should enter the bridge repeatedly:

$$\Box \Diamond (\, mode_W = \texttt{bridge} \,).$$

For the given model of the trains, no controller can satisfy this requirement since the model does not require the train to arrive at the bridge: a train could always stay in the mode `away` forever. Indeed, this is not an appropriate requirement for resource allocation problems. The granting of the response—setting the signal green by the controller should be preconditioned on the request—waiting at the bridge by the train. Consider the following revised liveness requirement, which asserts that if the west train is waiting then eventually the west traffic signal should turn green:

$$\Box [\, (mode_W = \texttt{wait}) \rightarrow \Diamond (signal_W = \texttt{green}) \,].$$

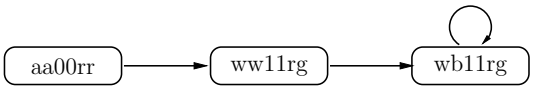

Figure 5.2: Cyclic Counterexample Illustrating Liveness Violation

This LTL-formula says that at every step, if the condition $mode_W = \texttt{wait}$ holds, then at some later step, the condition $signal_W = \texttt{green}$ must hold. This is a typical pattern for LTL formulas: *Always (Request implies Eventually Response)*. We want to check if every trace of the system `RailRoadSystem2` (see figure 3.8) satisfies this specification. It turns out that this is not the case. The counterexample, which consists of an initial sequence of steps followed by a cyclic execution that repeats, is illustrated in figure 5.2. As in figure 3.7, each state is denoted by listing the values of the variables $mode_W$, $mode_E$, $near_W$, $near_E$, *west*, and *east*, in that order, and a, w, b, g, and r are abbreviations for `away`, `wait`, `bridge`, `green`, and `red`, respectively. The cycle in the counterexample corresponds to the case when the east train is on the bridge refusing to leave, while the west train keeps waiting. We can conclude that if the controller lets the east train on the bridge and the train does not ever leave the bridge, a scenario consistent with the given model of the train, then the controller cannot possibly let the west train in. Since no controller can satisfy the requirement as specified, we need to modify our specification of the requirement.

An alternative standard form of liveness requirement is captured by the revised formula φ_{df}, which states that if the west train is waiting, then eventually either the corresponding signal is green or the east train is on the bridge:

$$\Box[\ (mode_W = \texttt{wait}) \ \rightarrow \ \Diamond \ [(signal_W = \texttt{green}) \ \vee \ (mode_E = \texttt{bridge})]\ ].$$

This form of requirement is called *deadlock freedom*: while it does not ensure that the controller is responsive to the west train when it requests an entry, it does ensure utilization of the resource. In particular, a controller that keeps both the traffic lights red all the time would violate this requirement, and so would a controller that keeps both trains waiting for one another in a deadlocked manner due to a buggy design. The controller `Controller2` of figure 3.8 is free of such deadlocks and meets the specification φ_{df}.

The more stringent requirement that every request should be fulfilled by granting the resource to the requester is called *starvation freedom*. In our example, if the west train wants to enter, then starvation freedom requires that it should be allowed to enter. As discussed already, this is feasible only when the east train is well behaved in the sense that it does not stay on the bridge forever. The formula φ_{sf} below asserts that under the assumption that the east train is repeatedly off the bridge, if the west train is waiting, then eventually the west traffic signal should turn green:

$$\Box\Diamond\ (mode_E \neq \texttt{bridge}) \ \rightarrow \ \Box[\ (mode_W = \texttt{wait}) \ \rightarrow \ \Diamond \ (signal_W = \texttt{green})\ ].$$

Note that the requirement expressed by φ_{sf} is stronger than φ_{df}: any trace that satisfies φ_{sf} also satisfies φ_{df} but not vice versa. The controller `Controller2` of figure 3.8 is in fact starvation-free and meets the specification φ_{sf}. In particular, the counterexample of figure 5.2 is ruled out: since the precondition $\Box \Diamond (mode_E \neq \texttt{bridge})$ is violated by this trace, it satisfies φ_{sf}.

Exercise 5.5: Consider the design of the synchronous three-bit counter from section 2.4.1. Write an LTL-formula to express the requirement that if the input signal *inc* is repeatedly high, then it is guaranteed that the counter will be repeatedly at its maximum value (that is, all the three output bits out_0, out_1, and out_2 are 1). Does the circuit `3BitCounter` of figure 2.27 satisfy this specification? ■

Exercise 5.6: Recall the synchronous design of a cruise controller system from section 2.4.2. Consider the following requirement: when the cruise-controller is "on," assuming the driver does not issue any further input events, eventually the speed becomes equal to the desired cruising speed and stays equal. Express this requirement in LTL using the variables *on*, *speed*, *cruiseSpeed*, *cruise*, *inc*, and *dec*. ■

5.1.3 LTL Specifications for Asynchronous Processes *

LTL-formulas can be used to specify constraints on executions of asynchronous processes also. Consider an asynchronous process P with state variables S, input channels I, and output channels O. Then LTL-formulas over the set $I \cup O$ can be used to specify desired requirements on sequences of inputs and outputs. We can associate infinite traces with infinite executions of P in the same manner we associated traces with executions of synchronous components with the following two changes. First, in the asynchronous model, each action is either an internal action that does not involve any of the input or output channels, an input action involving a single input channel x, or an output action involving a single output channel y. To interpret an LTL-formula over the set $I \cup O$ of variables, we need an infinite sequence of valuations, where each valuation needs to assign values to all the input and output variables. To interpret a single action of P as a valuation for all the input and output variables, we can assign the undefined value $\perp$ to each input and output channel not involved in the action. Second, since an asynchronous process has (weak or strong) fairness assumptions associated with its tasks, to check whether the process satisfies an LTL-specification, we consider the traces corresponding only to the fair executions: the asynchronous process P satisfies the LTL-specification φ if the trace corresponding to every *fair* infinite execution of P satisfies φ.

Suppose for the reliable communication buffer with input channel *in* and output channel *out*, we want to specify that a message sent on the input channel eventually appears on the output channel (see section 4.3.2). The following LTL-formula specifies this requirement for a given value v of type `msg`:

$$\Box \, (\, (in = v) \; \rightarrow \; \Diamond \, (out = v) \,).$$

The alternating-bit protocol discussed in section 4.3.2 satisfies this requirement under the fairness assumptions discussed there.

In some problems, the requirements for the asynchronous solutions are no different from the corresponding requirements for the synchronous designs. One such instance is the leader election problem. We have already studied LTL-formulas corresponding to the requirements that every node n eventually decides, and for every pair of nodes m and n, either the node m is never a leader or the node n is never a leader. The same formulas can be used as requirements for the asynchronous case.

The difference in the requirements for the synchronous and asynchronous cases is highlighted by the specification of logical gates. Consider an inverter with input *in* and output *out*. The natural specification in the synchronous case is the always formula

$$\Box\,(\,out = \neg\, in\,),$$

which says that the output is always equal to the negation of the input. In the asynchronous case, this specification cannot be satisfied due to the decoupling of the changes in the output in response to the changes in the input. We can demand that if the input is 0, then we expect the output to eventually become 1, provided the input is maintained unchanged at 0. This is expressed by the following formula:

$$\Box\,[\,(\,in = 0\,)\;\rightarrow\;(\,in = 0\,)\;\mathcal{U}\;(\,in = 1\;\vee\;out = 1\,)\,].$$

A symmetric formula can express the requirement that if the input is 1, then unless the input is changed back to 0, and the output will eventually become 1.

We can also write LTL-requirements that refer to state variables as well as input and output channels. To interpret such a formula, the trace corresponding to an ω-execution retains the values of state variables also.

Fairness Assumptions

In section 4.2.4, we discussed how to annotate tasks of an asynchronous process with fairness assumptions so that we consider only those infinite executions in which enabled tasks are not delayed forever. To check whether an asynchronous process meets its specification given as an LTL-formula, we check if every fair execution satisfies the specification. Now we discuss how to capture fairness assumptions within LTL-specifications. LTL-formulas corresponding to fairness assumptions can be useful for a better understanding of the distinction between weak and strong fairness and also suggest how an analysis tool designed for checking whether all executions satisfy a given LTL-formula can easily be adapted to check whether all fair executions satisfy an LTL-specification.

Consider an asynchronous process P with state variables S, input channels I, and output channels O. To express fairness requirements in LTL, at every step of the execution, we need to be able to express whether a task is enabled and

`nat` $x := 0;\ y := 0;\ \{A_x, A_y\}$ *taken*
$A_x :\ x := x + 1;\ \mathit{taken} := A_x$
$A_y :\ \texttt{even}(x)\ \rightarrow\ \{\, y := y + 1;\ \mathit{taken} := A_y \,\}$

Figure 5.3: Modified Version of `AsyncEvenInc`

whether a task is taken. For each output and internal task A of P, let $\mathit{Guard}(A)$ be the guard condition for the task A, which is a Boolean-valued expression over the state variables whose value in a state indicates whether the task A is enabled in that state. Thus, an infinite execution satisfies the LTL-formula $\Box\Diamond\, \mathit{Guard}(A)$ exactly when the task A is enabled at infinitely many steps of the execution. While a state of an asynchronous process contains enough information about whether a task is enabled, to refer to whether a task is taken, we introduce an additional variable, *taken*, that ranges over the set of tasks: its value at each step indicates the most recent task executed. The update code of a task A is modified so that it sets this variable to A.

To illustrate this, recall the asynchronous process `AsyncEvenInc` from section 4.2.4 (see figure 4.18). Figure 5.3 shows the corresponding process with the additional variable *taken*. An infinite execution of this modified process satisfies the recurrence formula $\Box\Diamond\,(\mathit{taken} = A_y)$ exactly when the task A_y is executed infinitely often.

With this modification, for a given output or an internal task A, consider the following formula

$$\mathit{wf}(A) :\ \Diamond\Box\ \mathit{Guard}(A)\ \rightarrow\ \Box\Diamond\ (\mathit{taken} = A).$$

This formula expresses the requirement that if the task A is persistently enabled, then it must be repeatedly taken. Thus, a trace satisfies this formula precisely when the trace corresponds to an infinite execution that is weakly fair with respect to the task A. To check if the process `AsyncEvenInc` guarantees the value of x to eventually exceed 10 assuming weak fairness for the task A_x, for which the guard condition is always true, we check if the modified process of figure 5.3 satisfies the LTL-formula

$$\Box\Diamond\ (\mathit{taken} = A_x)\ \rightarrow\ \Diamond\,(x > 10).$$

Indeed this LTL-formula is satisfied along every infinite execution of the process. To check if the value of y is guaranteed to eventually exceed 10 assuming weak fairness for the task A_y, we check if the process of figure 5.3 satisfies the LTL-formula

$$[\,\Diamond\Box\ \texttt{even}(x) \rightarrow \Box\Diamond\ (\mathit{taken} = A_y)\,]\ \rightarrow\ \Diamond\,(y > 10).$$

This requirement does not hold: the infinite execution in which only the task A_x is executed at every step does not satisfy this formula, and thus weak fairness assumption for the task A_y does not suffice to ensure satisfaction of the eventuality $\Diamond\,(y > 10)$.

Note that the formula $wf(A)$ is equivalent to the following formula, which asserts that if the task A is enabled at a given step, then at a later position it is either taken or disabled:

$$wf(A):\ \Box\,[\,Guard(A)\ \rightarrow\ \Diamond\,(\,(taken = A)\ \vee\ \neg\,Guard(A)\,)\,].$$

The strong fairness assumption for a given task A is expressed by the formula:

$$sf(A):\ \Box\Diamond\ Guard(A)\ \rightarrow\ \Box\Diamond\ (taken = A).$$

This formula asserts that if the task is repeatedly enabled, then it must be repeatedly executed. Thus, a trace satisfies this formula precisely when the trace corresponds to an infinite execution that is strongly fair with respect to the task A.

To check if the process `AsyncEvenInc` guarantees the value of y to eventually exceed 10 assuming strong fairness for the task A_y, we check if the process of figure 5.3 satisfies the LTL-formula

$$[\,\Box\Diamond\ \texttt{even}(x)\ \rightarrow\ \Box\Diamond\,(taken = A_y)\,]\ \rightarrow\ \Diamond\,(y > 10).$$

Indeed this LTL-formula is satisfied along every infinite execution of the process.

Note that, independent of exactly how a trace assigns values to the expressions $Guard(A)$ and $(taken = A)$ at each step, the following temporal implication is valid:

$$sf(A)\ \rightarrow\ wf(A).$$

This explains that "strong fairness" is indeed a stronger requirement than "weak fairness."

Instead of requiring that all fair executions of an asynchronous process P satisfy an LTL-formula φ, we can require all executions of the process P to satisfy the conditional LTL-formula $\varphi_{fair} \rightarrow \varphi$, where φ_{fair} is the conjunction of $wf(A)$ and $sf(A)$ formulas for tasks A for which weak and strong fairness assumptions are made. For example, for the process `UnrelFIFO` of figure 4.19, we assume strong fairness for the internal task A_1 that correctly transfers an element from the queue x to the queue y and weak fairness for the output task A_{out} that transmits the elements from the internal queue y to the output channel. To demand that all fair executions satisfy an LTL-formula φ is equivalent to requiring that all executions satisfy the formula

$$(\,sf(A_1)\ \wedge\ wf(A_{out})\,)\ \rightarrow\ \varphi.$$

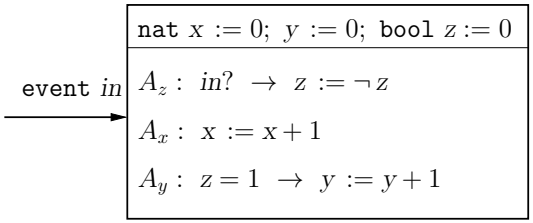

Figure 5.4: Exercise: Satisfaction Under Fairness Assumptions

Exercise 5.7: Consider the two specifications of weak fairness:

$$\varphi_1 : \Diamond\Box\ \mathit{Guard}(A)\ \rightarrow\ \Box\Diamond\ (\mathit{taken} = A)$$

and

$$\varphi_2 : \Box\ [\mathit{Guard}(A)\ \rightarrow\ \Diamond\ ((\mathit{taken} = A)\ \vee\ \neg\ \mathit{Guard}(A))].$$

Prove that these two LTL-formulas are equivalent. ■

Exercise 5.8: Consider an asynchronous process P shown in figure 5.4 with the input task A_z and internal tasks A_x and A_y. For each of the LTL-formulas below, does the process P satisfy the formula? If not, is there a suitable fairness assumption regarding execution of tasks under which the process satisfies this specification? When adding fairness assumptions, clearly specify whether you are using strong fairness or weak fairness and for which tasks with a justification.

(1) $\Diamond\ (x > 5)$;
(2) $\Diamond\ (y > 5)$;
(3) $\Box\Diamond\ (z = 1)\ \rightarrow\ \Diamond\ (y > 5)$.

■

5.1.4 Beyond LTL *

We conclude this section by noting some of the limitations of the logic LTL as a specification language for writing requirements. These limitations have spawned a number of extensions and variations of LTL. While a detailed study of various temporal logics and their comparative merits is beyond the scope of this textbook, the discussion below is a glimpse into the rich variety of alternative temporal logics.

Branching-Time Temporal Logics

An LTL-formula is evaluated over a trace corresponding to a single execution of a system, and a system satisfies an LTL-formula when *all* executions of the system satisfy the formula. With this interpretation, there is no way to demand

that some executions satisfy one type of a requirement and some others satisfy another kind. In particular, for the consensus problem discussed in section 4.3.3, consider the requirement that "if the preferences of the two processes P_1 and P_2 are different initially, then both decisions are possible." This requirement cannot be specified in LTL but can be stated in the so-called "branching-time" temporal logics. Recall that the variables $pref_1$ and $pref_2$ capture the initial preferences of the two processes, and the variables dec_1 and dec_2 are assigned the decision values on termination. Then the following formula of the branching-time temporal logic CTL (Computation Tree Logic) captures the desired requirement:

$$(pref_1 \neq pref_2) \;\rightarrow\; [\,\exists\Diamond\,(\,dec_1 = dec_2 = 0\,)\;\wedge\;\exists\Diamond\,(\,dec_1 = dec_2 = 1\,)\,].$$

The logic CTL, in addition to the logical and temporal operators, allows existential ($\exists$) and universal ($\forall$) quantifiers over executions. The formulas are interpreted over the tree of all executions where nodes correspond to states and branching corresponds to possible choices of the successor state at each node. A quantified branching-time formula $\exists\,\varphi$ is satisfied at a node if there is an execution ρ starting at the corresponding state such that ρ satisfies the formula φ, which may contain temporal operators.

Stateful Temporal Logics

Given a Boolean variable e, consider the following requirement: the value of e is 1 in every even position. One can prove that no LTL-formula exactly captures this requirement. Note that the LTL-formula

$$\bigcirc\,(e = 1)\;\wedge\;\Box\,[\,(e = 1)\;\rightarrow\bigcirc\,\bigcirc\,(e = 1)\,]$$

expresses a much stronger requirement: for a trace to satisfy this formula, not only is it necessary that at every even position the value of e must be 1, but if the value of e happens to be 1 at some odd position, then the formula can be satisfied only when the value of e is 1 in all subsequent odd positions. Intuitively, the desired requirement "the value of e is 1 in every even position" requires the specification logic to maintain an internal state variable that captures whether a position is odd or even, and LTL-formulas cannot maintain such a state. This shortcoming has led to extensions of LTL with regular expressions (or equivalently deterministic finite automata) to express *stateful* temporal constraints. The IEEE standard Property Specification Language PSL allows a combination of temporal operators and regular expressions.

Interpretation over Finite Traces

In our formalization of LTL, LTL-requirements specify constraints only on infinite executions of a system. As a result, if a synchronous reactive component C has no infinite executions (this may happen if the component C is not input enabled), then no matter which LTL-formula φ we consider, the component C satisfies the requirement φ vacuously. Also, since not every reachable state

necessarily appears on some infinite execution, given a state property φ, the system may satisfy the always-formula $\Box\,\varphi$, even though the property φ is not an invariant of the system. This anomaly can be avoided if we redefine the semantics of LTL-formulas so that a formula can be evaluated on a finite trace also. If $\rho = q_1 q_2 \cdots q_m$ is a finite trace and $1 \leq j \leq m$ is a position in the trace, then

$(\rho, j) \models \Box\,\varphi$ if $(\rho, k) \models \varphi$ for all positions k with $j \leq k \leq m$; and
$(\rho, j) \models \bigcirc\,\varphi$ if $j < k$ and $(\rho, j+1) \models \varphi$.

Thus, the main difference is that $\bigcirc\,\varphi$ now means that it's not yet the end of the trace and the next position satisfies φ. Now, a component C satisfies an LTL-formula φ if every infinite trace as well as every finite trace corresponding to a maximal execution of C satisfies φ. Here, a maximal execution is a finite execution of C that ends in a state that has no successors (that is, the execution cannot be extended by an additional state). Intuitively, maximal traces correspond to terminating (or deadlocked) executions, and including such executions ensures that while evaluating an LTL-formula, we examine all the reachable states of the component. Note that evaluating an LTL-formula on *all* finite executions is not meaningful: if a system satisfies an eventuality φ after, say five rounds, then executions of the system of length less than five do not satisfy the eventually formula $\Diamond\,\varphi$ but should not be considered as counterexamples (however if there is some maximal execution that does not contain a state satisfying φ, then it does indicate a violation of the requirement $\Diamond\,\varphi$). All the analysis techniques for establishing that a system satisfies its LTL-specification can be easily modified to account for such a revised interpretation.

5.2 Model Checking

In chapter 3, we saw that the canonical safety verification problem is the invariant verification problem: given a transition system T and a property φ over its state variables, we want to check if all the reachable states of the system T satisfy the property φ. For automated verification, we reduced the invariant verification to the reachability problem: to check whether the property φ is an invariant of the transition system T, we check whether a state violating φ is reachable and, if so, the corresponding execution is a counterexample to the invariant verification question. We then studied both enumerative and symbolic algorithms for solving the reachability problem.

Repeatability Problem

In model checking, given the system described as a synchronous reactive component, or as an asynchronous process, and an LTL-specification, we want to check if every execution of the system satisfies the given LTL-specification. The core computational problem for verification of liveness requirements turns out to be the *repeatability* problem: given a transition system T and a property φ

over its state variables, does there exist an infinite execution of the system T that repeatedly encounters states satisfying the property φ (that is, whether the recurrence formula $\Box\Diamond\varphi$ is satisfied by some infinite execution of the system)? This form of repeated reachability is also known as *Büchi reachability*, named after the logician J. Richard Büchi who studied finite automata over infinite words resulting in an elegant theory of ω-regular languages that mirrors the classical theory of regular languages over finite words. We will show that the LTL model-checking problem, namely, checking whether every trace of a given system satisfies a given LTL-specification, can be reduced to the repeatability problem for the composition of the system and a monitor derived from the LTL-specification. In the safety case, the finite execution that demonstrates the reachability of certain error states of the monitor corresponds to a counterexample indicating a violation of the requirement. Similarly, in model checking, the infinite execution that demonstrates the repeated reachability of certain error states of the monitor is a counterexample indicating a violation of the liveness requirement.

> REPEATABILITY PROBLEM FOR TRANSITION SYSTEMS
>
> An *infinite execution* of a transition system T consists of an infinite sequence of the form $\rho = s_0, s_1, \ldots$ such that s_0 is an initial state of T and for each $j > 0$, (s_{j-1}, s_j) is a transition of T. A property φ over the state variables of T is said to be *repeatable* if there exists *some* infinite execution ρ of T such that the execution ρ satisfies the recurrence LTL-formula $\Box\Diamond\varphi$. The *repeatability problem* is to check, given a transition system T and a property φ over its state variables, whether φ is repeatable.

When the answer to the repeatability problem, for a given transition system T and property φ, is positive, we want to demonstrate an infinite execution in which the property φ is repeatedly satisfied. Typically such an execution is illustrated by a state s, such that (1) the state s is reachable from some initial state, (2) the state s is reachable from itself using one or more transitions, and (3) the state s satisfies the property φ. That is, as evidence, we will produce a cycle that is reachable from some initial state and contains a state satisfying φ.

Recall the transition system $\texttt{GCD}(m, n)$ of figure 3.1 capturing the program for computing the greatest common divisor of two numbers m and n. Note that as long as both the variables x and y have positive values, the system stays in the mode `loop` updating the variables. Suppose we want to check the liveness requirement that the loop always terminates. This corresponds to checking repeatability of the property $(mode = \texttt{loop})$: an infinite execution where this condition is satisfied repeatedly corresponds to a nonterminating execution.

5.2.1 Büchi Automata

Now we describe how to compile LTL-formulas into a special kind of monitor, called Büchi automata, so that the model checking problem can be reduced to the repeatability problem for the composition of the system and the monitor.

Definition

Given a set V of Boolean variables, a Büchi automaton over V is a synchronous reactive component M with the set V as input variables, described as an extended-state machine. The only state variable for the automaton is its mode, and thus it has only finitely many states. It has no outputs. Thus, a mode-switch, or an edge, is completely described by the source and the target states of the switch and the guard condition, which is a Boolean expression over the input variables. Given an infinite sequence of inputs, that is, a trace over V, an execution of the machine produces an infinite sequence of states. A subset F of the states is declared as *accepting*. The execution corresponding to an input trace is accepting if some accepting state repeats infinitely often along this execution. The automaton can be nondeterministic: from a given state and a given input, the guard conditions of multiple outgoing switches can be true simultaneously. Thus, for a given input trace, multiple executions are possible. The automaton M accepts an infinite trace ρ over V if there exists an accepting infinite execution when supplied with the sequence ρ of inputs. The formal definition is summarized below:

BÜCHI AUTOMATON

A Büchi automaton M consists of

- a finite set V of Boolean input variables,
- a finite set Q of states,
- a set $\mathit{Init} \subseteq Q$ of initial states,
- a set $F \subseteq Q$ of accepting states, and
- a finite set E of edges, where each edge is of the form (q, Guard, q') consisting of a source state $q \in Q$, a target state $q' \in Q$, and a Boolean expression *Guard* over the input variables V.

For two states q and q' and an input $v \in Q_V$, $q \xrightarrow{v} q'$ is a transition of the automaton if there exists an edge (q, Guard, q') such that the input v satisfies the expression *Guard*. Given a trace $\rho = v_1 v_2 \cdots$ over the input variables, an execution of the automaton over the input trace ρ is an infinite sequence of the form $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots$ such that q_0 is an initial state, and for each $i \geq 1$, $q_{i-1} \xrightarrow{v_i} q_i$ is a transition. The Büchi automaton M *accepts* the input trace ρ if there exists an execution over ρ such that for infinitely many indices i, $q_i = q$ for some accepting state $q \in F$.

The Büchi automaton M is said to be deterministic if from every state on a given input only one edge can be chosen: for every state q and every pair of edges $(q, \mathit{Guard}_1, q_1)$ and $(q, \mathit{Guard}_2, q_2)$, the conjunction $\mathit{Guard}_1 \wedge \mathit{Guard}_2$ is unsatisfiable (ensuring that an input v that satisfies the guard Guard_1 cannot satisfy the guard Guard_2, and vice versa).

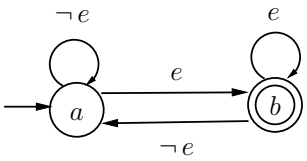

Figure 5.5: Büchi Automaton for $\Box\Diamond\, e$

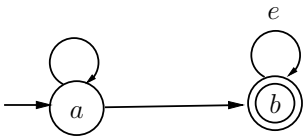

Figure 5.6: Büchi Automaton for $\Diamond\Box\, e$

Examples

Figure 5.5 shows the Büchi automaton M that accepts only those traces that satisfy the recurrence formula $\Box\Diamond\, e$. It has a single Boolean input variable e and is a state machine with two states a and b. The state a is initial, and the state b is accepting (indicated by a double circle). In each round, if the input e is 1, the automaton transitions to the state b, and if the input e is 0, the automaton transitions to the state a. Given a trace $v_1 v_2 \cdots$ of inputs, the automaton M has a unique execution, and the state b repeats infinitely often along this execution precisely when the input sequence contains infinitely many 1 s. Thus, the automaton M accepts an input trace ρ precisely when the input trace contains infinitely many 1 s, that is, when the trace ρ satisfies the LTL-formula $\Box\Diamond\, e$.

Figure 5.6 shows another Büchi automaton. It also has a single Boolean input variable e and is a state machine with two states a and b, of which the state a is initial and the state b is accepting. This automaton is nondeterministic. In the initial state, in each round, independent of whether the input value is 0 or 1, the automaton can either stay in the state a or switch to the state b. Once the automaton switches to the state b, if the input is 0, then no transition is enabled, and the automaton gets stuck. Since the only accepting state is b, and once in the state b, the automaton can generate an infinite execution only when all the subsequent input values are 1 s, the automaton has an infinite execution with the state b repeating precisely when from some position onward the input trace contains only 1 s. That is, the automaton accepts exactly those traces that satisfy the LTL-formula $\Diamond\Box\, e$. There is no deterministic Büchi automaton corresponding to the LTL-formula $\Diamond\Box\, e$, and thus nondeterminism can be crucial for constructing Büchi automata corresponding to LTL formulas.

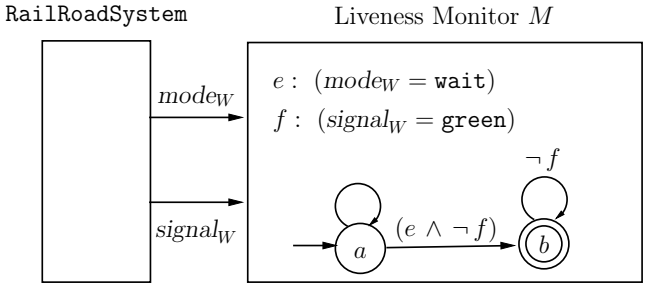

Figure 5.7: Monitoring Liveness Violations Using a Büchi Automaton

Monitoring for Violations of Liveness Requirements

To illustrate how Büchi automata can be used as monitors to detect violations of liveness requirements, let us revisit the railroad controller example and the liveness requirement, which asserts that if the west train is waiting, then eventually the west traffic signal should turn green:

$$\Box [\ (mode_W = \texttt{wait}) \ \rightarrow \ \Diamond \ (signal_W = \texttt{green})\ ].$$

This is a a commonly occurring pattern for liveness requirements of the form $\varphi : \ \Box (e \rightarrow \Diamond f)$, where e and f are expressions over the observable variables of the system. To check whether every trace of the system satisfies φ, we first *negate the specification* and check if there is some execution of the system that satisfies the negated specification. The negated specification is $\neg \Box (e \rightarrow \Diamond f)$, which is equivalent to the formula $\Diamond (e \wedge \Box \neg f)$. Thus, a violation of the requirement is an infinite execution in which the property e holds at some position, and then onward the property f never holds. Consider the nondeterministic Büchi automaton M shown in figure 5.7 that accepts exactly those traces that satisfy this negated formula. The automaton can switch to the accepting state b only when it encounters an input that satisfies the property e, and once in the state b, it can continue execution only when the input at every step does not satisfy the property f. In the composed system $\texttt{RailRoadSystem} \| M$, there is an infinite execution in which the automaton state is repeatedly b if and only if the component `RailRoadSystem` can produce a trace that satisfies $\neg \varphi$. Thus, the model checking problem of verifying that every trace of the system satisfies the LTL-formula φ has been reduced to checking the repeatability of the property $(M.mode = b)$ for the composed system. As already discussed, `RailRoadSystem2` (see figure 3.8) does not satisfy the specification: in the system $\texttt{RailRoadSystem2} \| M$, the property $(M.mode = b)$ is repeatable since the state with $mode_W = \texttt{wait}$, $mode_E = \texttt{bridge}$, $near_W = near_E = 1$, $signal_W = \texttt{red}$, $signal_E = \texttt{green}$, and $M.mode = b$ is reachable and has a self-loop.

As another example of monitoring for violations of liveness requirements, consider the LTL specification $\varphi : \ \Box \Diamond \ e \rightarrow \Box \Diamond \ f$, where e and f are expressions

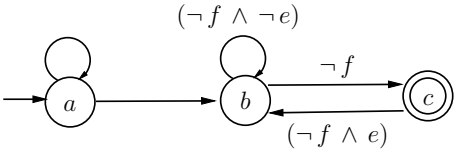

Figure 5.8: Büchi Automaton for Detecting Violation of $\Box \Diamond\ e \rightarrow \Box \Diamond\ f$

over the observable variables of the component C. It asserts conditional recurrence demanding that if the property e is recurrent, then so should be the property f. To check whether every trace of the component C satisfies φ, we first negate the specification and check if there is some execution of the component C that satisfies the negated specification. The negated specification is equivalent to $\Box \Diamond\ e \wedge \Diamond \Box \neg f$, stating that the property e is recurrent and the property $\neg f$ is persistent. Thus, we want to find an infinite execution in which after a certain position, $\neg f$ holds continuously and e holds repeatedly. This is captured by the Büchi automaton M shown in figure 5.8 with three states. Initially the state is a. The automaton loops in this initial state for an arbitrary number of steps and then nondeterministically switches to the state b. Subsequently, every time the condition e is satisfied, it transitions to the state c and switches back to the state b in the next step. In both states b and c, the execution can continue only if the condition $\neg f$ holds. An infinite execution can visit the state c repeatedly only if the property e is recurrent and the property $\neg f$ is persistent. Thus, the model checking problem can be reformulated as checking the repeatability of the property $(mode = c)$ for the composed system $C \| M$.

Generalized Büchi Automata

Consider the LTL-formula $\Box \Diamond\ e \wedge \Box \Diamond \neg e$, which says that the variable e should be 1 infinitely often and should also be 0 infinitely often. A convenient way to capture this requirement is to use the same automaton structure as the one shown in figure 5.5 and to use *two* accepting sets: $F_1 = \{a\}$ and $F_2 = \{b\}$. An execution of such a machine over an input trace is accepting if both of these sets repeat infinitely often. This extension of Büchi automata with such a conjunctive accepting requirement is called *generalized* Büchi automata.

Formally, a generalized Büchi automaton has input variables V, states Q, initial states *Init*, and edges of the form $(m, Guard, m')$, as in the case of a Büchi automaton, and has sets $F_1, F_2, \ldots F_k$ of accepting sets of states. An execution $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots$ of the automaton over an input trace $v_1 v_2 \cdots$ is said to be accepting if for each j, for infinitely many indices i, the state q_i belongs to the accepting set F_j. In other words, the trace $q_0 q_1 \ldots$ over states corresponding to the execution should satisfy the formula $\wedge_{j=1,\ldots k} \Box \Diamond\ (mode \in F_j)$.

It turns out that generalized Büchi automata are no more expressive than Büchi automata. Expressing a requirement on input traces as a generalized Büchi

automaton with multiple accepting sets can allow the design of a machine with fewer states, but it is possible to compile it into a Büchi automaton (with a single accepting set) without changing the set of input traces it accepts.

Proposition 5.1 [From Generalized Büchi automata to Büchi automata] *For a given generalized Büchi automaton M over the input variables V, there exists a Büchi automaton M' over the input variables V such that for every trace ρ over V, the automaton M accepts the trace ρ exactly when the automaton M' accepts it.*

Proof. Let M be a generalized Büchi automaton over inputs V with states Q, initial states *Init*, edges E, and accepting sets $F_1, \ldots F_k$. We want to construct a Büchi automaton M' such that visiting one of its accepting states repeatedly ensures that the original automaton M has visited each of the sets F_j repeatedly. For this purpose, the automaton M' maintains the state of M and, additionally, a counter that cycles through the values $1, 2, \ldots k, 0$. Initially the counter is 1. When the state of M is in the set F_1, the counter is incremented to 2. When a state in the accepting set F_2 is encountered, it is incremented to 3. More generally, when the counter is j, the automaton is waiting to visit a state in the accepting set F_j. When such a state is encountered, the counter is incremented to $j + 1$. When the counter j equals k, when a state in the accepting set F_k is encountered, the counter is updated to 0, and in the next transition, it is changed to 1. If along the execution, the counter is 0 repeatedly, then it has cycled through all the values repeatedly, and the execution has visited each of the accepting sets F_j repeatedly. Conversely, if an accepting set F_j repeats infinitely often along an execution, then the counter cannot get "stuck" at the value j, and thus if all the accepting sets repeat infinitely often, then the counter cycles through 0 repeatedly.

Formally, the set of states of M' is the set $Q \times \{0, 1, \ldots k\}$. The initial states of M' are of the form $\langle q, 1\rangle$ with $q \in \mathit{Init}$. For every edge (q, Guard, q') of M, the automaton M' has the edge $(\langle q, 0\rangle, \mathit{Guard}, \langle q', 1\rangle)$; for every $1 \leq c < k$, if $q \in F_c$, then the automaton M' has the edge $((q, c), \mathit{Guard}, (q', c + 1))$ or else the edge $(\langle q, c\rangle, \mathit{Guard}, \langle q', c\rangle)$; and if $q \in F_k$, then the automaton M' has the edge $(\langle q, k\rangle, \mathit{Guard}, \langle q', 0\rangle)$ or else the edge $(\langle q, k\rangle, \mathit{Guard}, \langle q', k\rangle)$. The accepting set for M' is the set of states of the form $\langle q, 0\rangle$.

To complete the proof, we need to show that an input trace ρ is accepted by the generalized Büchi automaton M exactly when it is accepted by the Büchi automaton M', but this follows in a straightforward manner from the definitions. ■

Exercise 5.9: For each of the LTL-formulas below, construct a Büchi automaton that accepts exactly those traces that satisfy the formula:

(1) $\Box\Diamond\, e \vee \Diamond\Box\, f$;
(2) $\Box\Diamond\, e \wedge \Box\Diamond\, f$;
(3) $\Box(\, e \rightarrow e\, \mathcal{U}\, f\,)$.

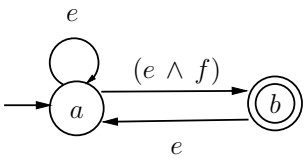

Figure 5.9: Exercise: From a Büchi Automaton to LTL

■

Exercise 5.10: Write an LTL-formula that exactly describes the set of traces that are accepted by the Büchi automaton shown in figure 5.9. Explain your answer. ■

Exercise 5.11*: Given two Büchi automata M_1 and M_2, both over the same set V of input variables, show how to construct a Büchi automaton M over the inputs V such that the automaton M accepts an input trace ρ over V exactly when both the automata M_1 and M_2 accept the trace ρ. ■

Exercise 5.12*: Given a Büchi automaton M with states Q and accepting states F, consider the Büchi automaton M' obtained by toggling the role of accepting states in M: the states, the initial states, and the edges of the automaton M' are the same as the ones in the original automaton M, but its accepting states are $Q \setminus F$ (that is, a state is accepting in M' exactly when it is not accepting in M). Consider the claim "an input trace ρ is accepted by the automaton M' exactly when it is not accepted by the automaton M." Does the claim hold? If your answer is "yes," justify with a proof. If your answer is "no," give a counterexample. In this latter case, does the claim hold if the automaton M is deterministic? ■

5.2.2 From LTL to Büchi Automata*

The construction of Büchi automata for detecting violations of LTL specifications can be automated. An LTL-formula φ can be compiled into a (generalized) Büchi automaton M_φ, which accepts exactly those traces that satisfy the formula φ. States of the desired automaton are *sets of subformulas* of φ. Such an automaton is called a *tableau*. We first illustrate the construction using an example.

Sample Tableau Construction

To illustrate the principles of the tableau construction, let us consider the LTL-formula $\varphi = \Box(e \vee f) \wedge \Diamond e$. The states of the tableau are collections of LTL-formulas derived from φ. Each state q is a set of formulas, and we would like to ensure that every formula contained in a state q is satisfied by the input trace along every infinite accepting execution starting in the state q.

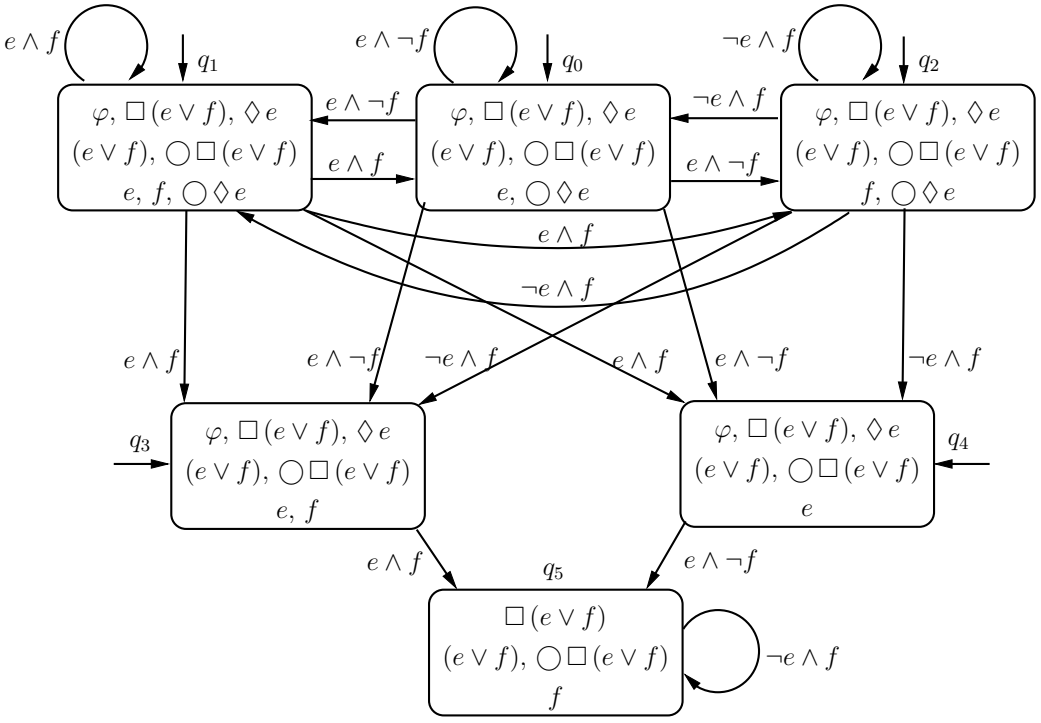

Figure 5.10: Tableau Construction for $\varphi = \Box(e \vee f) \wedge \Diamond e$

An initial state of the tableau is required to contain the given formula φ. From the semantics of conjunction, the formula φ is satisfied when both $\Box(e \vee f)$ and $\Diamond e$ are satisfied, so an initial state must contain both these formulas. From the semantics of the always operator, $\Box(e \vee f)$ is satisfied if both $(e \vee f)$ and the next-formula $\bigcirc\Box(e \vee f)$ are satisfied, and hence we add both of these to the desired initial state. To satisfy the disjunction $(e \vee f)$, a state must contain either e or f or both e and f. The inclusion of the next-formula $\bigcirc\Box(e \vee f)$ does not create any additional requirements on the current state, but the rules for adding transitions between states will ensure its satisfaction. From the semantics of the eventuality operator, $\Diamond e$ is satisfied if at least one of e and the next-formula $\bigcirc\Diamond e$ is satisfied. Combining the resulting cases with different possible ways of satisfying $(e \vee f)$ gives us five initial states q_0, q_1, q_2, q_3, and q_4 as shown in figure 5.10. For example, the state q_0 corresponds to the set $\{\varphi, \Box(e \vee f), \Diamond e, (e \vee f), \bigcirc\Box(e \vee f), e, \bigcirc\Diamond e\}$ of formulas: for an execution starting in a state q_0, we want each formula in this set to be satisfied and a formula such as f, which is not included in this set to be not satisfied.

Whenever a state includes an atomic expression, say e, it means that the input to be processed must satisfy this expression, and thus e should appear as a

conjunct in the guard of the edges out of this state. Similarly, when a state does not contain an atomic expression e, the input to be processed must not satisfy e, and thus $\neg e$ should appear as a conjunct in the guard of the edges out of this state. This explains the guards on all the edges of the automaton. In particular, each edge out of the state q_0 has the guard $(e \wedge \neg f)$.

To obtain successors of a state, we examine the next formulas in the state. For every formula of the form $\bigcirc \psi$, $\bigcirc \psi$ should belong to the current state if and only if the successor state contains ψ. Since the state q_0 contains both $\bigcirc \Box (e \vee f)$ and $\bigcirc \Diamond e$, its successor is required to contain $\Box (e \vee f)$ as well as $\Diamond e$. Such a successor state then must satisfy the conjunction of these two, which is the original formula φ. This means that the successors of the state q_0, and also of q_1 and q_2 by the same logic, are exactly the initial states containing φ. Now let us consider the state q_3, which contains $\bigcirc \Box (e \vee f)$ but not $\bigcirc \Diamond e$. Hence, its successor state should contain $\Box (e \vee f)$ but not $\Diamond e$ (and hence cannot contain φ). To satisfy the always-formula $\Box (e \vee f)$, a state must contain $(e \vee f)$ and $\bigcirc \Box (e \vee f)$. Since it does not contain $\Diamond e$, it cannot contain e, and thus the only way to satisfy $(e \vee f)$ is by including f. The state $q_5 = \{\Box (e \vee f), (e \vee f), \bigcirc \Box (e \vee f), f\}$ is thus the sole successor of the state q_3 and also of states q_4 and q_5 by the same logic.

We would like to ensure that if $p_0 \xrightarrow{v_1} p_1 \xrightarrow{v_2} \cdots$ is an infinite execution through the tableau corresponding to the input trace $\rho = v_1 v_2 \cdots$ starting at some state p_0, then a formula ψ is in p_0 if and only if the input trace ρ satisfies the formula ψ. This is not quite true yet. For instance, an execution can loop forever at the state q_2 provided each input satisfies f: every state contains $\Diamond e$, but no input satisfies the atomic expression e. Intuitively, along this infinite execution, the choice to satisfy $\Diamond e$ is postponed forever. This can be avoided by adding a Büchi acceptance condition that requires that to satisfy $\Diamond e$, one must satisfy e eventually. This is expressed by the Büchi accepting set $F_1 = \{q_0, q_1, q_3, q_4\}$ containing states that either contain e or do not contain $\Diamond e$. For the always-formula $\Box (e \vee f)$, if a state does not contain this formula, then we want to make sure that the formula is indeed not satisfied. Note that the negation of an always-formula is an eventually-formula, so the Büchi accepting condition F_2 contains all states that do not contain $(e \vee f)$ or contain $\Box (e \vee f)$, and in our example, this turns out to be the set of all states. Thus, an infinite execution is accepting according to both the accepting sets if it does not end up looping at state q_2 forever. Verify that the Büchi automaton, with the states and edges given by the tableau of figure 5.10, accepts an input trace over $\{e, f\}$ exactly when it satisfies the formula φ.

In summary, in a tableau construction, states are subsets of formulas. Each formula stipulates requirements concerning other formulas that must be satisfied by the sequence of inputs along the paths starting in the current state. The edges are defined so as to ensure propagation of the next formulas from one state to its successor. The generalized Büchi accepting requirements ensure eventual fulfillment of eventuality formulas.

Tableau Construction

We proceed to formalize the tableau construction. For the formal construction, let us assume that LTL-formulas are constructed from atomic expressions using the logical connectives of negation, conjunction, and disjunction and the temporal operators next, always, and eventually. Extending the construction to handle the until operator is left as an exercise.

Given an LTL-formula φ, let V_φ be the set of atomic expressions that occur in φ. To evaluate the formula φ, at every step we need to know whether each of the expressions in V_φ is satisfied. Thus, we can treat every atomic expression as a Boolean variable and interpret the formula φ with respect to a trace over the set V_φ of Boolean variables. These variables are the set of input variables for the Büchi automaton M_φ.

Let us first define the set of formulas that are relevant to evaluating the given formula. The *closure* $Sub(\varphi)$ of an LTL-formula φ is defined as:

1. if ψ is a syntactic subexpression occurring in φ, then ψ belongs to $Sub(\varphi)$; and
2. if ψ is a subexpression of the form $\Diamond\,\psi'$ or $\Box\,\psi'$, then $\bigcirc\,\psi$ also belongs to $Sub(\varphi)$.

For the illustrative tableau construction in figure 5.10, for $\varphi = \Box\,(e \vee f) \wedge \Diamond\, e$,

$$Sub(\varphi) \;=\; \{\, e, f, (e \vee f), \Diamond\, e, \bigcirc \Diamond\, e, \Box\,(e \vee f), \bigcirc \Box\,(e \vee f), \varphi \,\}.$$

As illustrated in the sample construction, whether the satisfaction of the formula φ depends on the satisfaction of the formulas in $Sub(\varphi)$. Verify that if the formula φ has length k, then the number of formulas in its closure is at most $2k$.

Now, a state of the tableau is a subset of formulas from the closure such that the set of constraints expressed by these formulas are locally consistent. A subset $q \subseteq Sub(\varphi)$ of the closure of φ is *consistent* if the following conditions are satisfied:

- $\neg\,\psi$ belongs to q exactly when ψ does not belong to q;
- $\psi_1 \wedge \psi_2$ belongs to q exactly when both ψ_1 and ψ_2 belong to q;
- $\psi_1 \vee \psi_2$ belongs to q exactly when either ψ_1 or ψ_2 or both belong to q;
- $\Diamond\,\psi$ belongs to q exactly when either ψ or $\bigcirc \Diamond\,\psi$ or both belong to q; and
- $\Box\,\psi$ belongs to q exactly when both ψ and $\bigcirc \Box\,\psi$ belong to q.

In our example of figure 5.10, note that all six states are indeed consistent. These are not all the consistent states, and figure 5.10 shows only those states

that are reachable from the initial states. For example, the state $q_6 = \{f, (e \vee f)\}$ is a consistent state but is not reachable.

Given a consistent subset q of $Sub(\varphi)$, let us denote by $Guard_q$ the expression obtained by conjoining each atomic expression e contained in q and the negation of each atomic expression e that does not belong to q. For example, if q contains the atomic expression e but does not contain the atomic expression f, then $Guard_q$ is the expression $e \wedge \neg f$ and captures the constraint on the next input when in state q.

Now we are ready to define the generalized Büchi automaton M_φ, also known as the tableau corresponding to the formula φ:

- the set of input variables is the set V_φ of atomic expressions that appear in φ;
- the set of states is the set of consistent subsets of the closure $Sub(\varphi)$;
- a state $q \subseteq Sub(\varphi)$ is initial exactly when q contains the formula φ;
- for a pair of states q and q', if it is the case that every next formula $\bigcirc \psi$ in $Sub(\varphi)$ belongs to q exactly when ψ belongs to q', then there is an edge $(q, Guard_q, q')$; and
- for each eventually formula $\psi = \Diamond \psi'$ in the closure $Sub(\varphi)$, there is an accepting set F_ψ containing states q such that $\psi' \in q$ or $\psi \notin q$; and for each always formula $\psi = \Box \psi'$ in the closure $Sub(\varphi)$, there is an accepting set F_ψ containing states q such that $\psi' \notin q$ or $\psi \in q$.

The correctness of the construction is established below.

Proposition 5.2 [Correctness of LTL Tableau Construction] *For every* LTL*-formula φ over the atomic expressions V, a trace ρ over V satisfies φ exactly when it is accepted by the generalized Büchi automaton M_φ.*

Proof. Let φ be an LTL formula, and let $\rho = v_1 v_2 \cdots$ be a trace over the set V_φ of expressions appearing in φ. Suppose $\rho \models \varphi$. For $i \geq 0$, let $q_i \subseteq Sub(\varphi)$ be the set $\{\psi \in Sub(\varphi) \mid (\rho, i+1) \models \psi\}$ of formulas true at the position $i+1$ in the trace ρ. From the definitions, it follows that (1) for all i, the set q_i is consistent; (2) for all i and all formulas $\bigcirc \psi \in Sub(\varphi)$, $\bigcirc \psi \in q_i$ exactly when $\psi \in q_{i+1}$; (3) the set q_0 is an initial state of M_φ; (4) for all i, the input v_{i+1} satisfies the guard $Guard_{q_i}$; (5) for each $\Diamond \psi \in Sub(\varphi)$, if $(\rho, i) \models \Diamond \psi$ for infinitely many positions i, then $(\rho, j) \models \psi$ for infinitely many positions j; and (6) for each $\Box \psi \in Sub(\varphi)$, if $(\rho, i) \not\models \Box \psi$ for infinitely many positions i, then $(\rho, j) \not\models \psi$ for infinitely many positions j. It follows that $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots$ is an accepting execution in the tableau M_φ, and the automaton M_φ accepts the trace ρ.

Now consider an accepting execution $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots$ of the automaton M_φ over the input trace $\rho = v_1 v_2 \cdots$. We want to establish that for all $\psi \in Sub(\varphi)$,

for all $i \geq 0$, $\psi \in q_i$ if and only if $(\rho, i+1) \models \psi$. The proof is by induction on the structure of ψ and is left as an exercise. It follows that $\rho \models \varphi$. ■

Note that the number of states of the automaton M_φ is exponential in the size of the formula. This blow-up is unavoidable. The generalized Büchi automaton M_φ can be converted into a Büchi automaton using the construction described in proposition 5.1.

Model Checking

To check whether a component C satisfies an LTL-formula φ, we first negate the formula φ. We build the Büchi automaton M corresponding to the formula $\neg\,\varphi$ such that an infinite execution of the composed system $C \parallel M$ in which the Büchi states of M are repeatedly encountered corresponds to an infinite execution of the component C that satisfies the negated specification $\neg\,\varphi$ and, thus, is a counterexample to the model checking problem. This approach of negating the formula *before* applying the tableau construction avoids the need for the computationally demanding task of complementing the tableau and is essential to the practical applications of model checking. We summarize the result below.

Theorem 5.1 [From LTL Model Checking to Repeatability] *Given an* LTL*-formula φ over the atomic expressions V that refer to the observable variables of a system C, there is an algorithm to construct a nondeterministic Büchi automaton M with the input variables V, with a subset F of accepting states, such that a system C satisfies the specification φ precisely when for the composed system $C \parallel M$, the property "state of M belongs to F" is repeatable.* ■

Exercise 5.13: Consider the LTL-formula $\varphi = \Box \Diamond e \vee \Box f$. First compute the closure $Sub(\varphi)$. Then apply the tableau construction to build the generalized Büchi automaton M_φ. It suffices to show only the reachable states. ■

Exercise 5.14: The formal description of the tableau construction considers formulas where the only temporal operators are next, eventually, and always. Describe how the modifications necessary when the until operator is also allowed. ■

Exercise 5.15: Consider the LTL-formula $\varphi = (e \,\mathcal{U}\, \bigcirc f) \vee \neg e$. First compute the closure $Sub(\varphi)$. Then apply the tableau construction to build the generalized Büchi automaton M_φ. It suffices to construct only the reachable states. ■

Exercise 5.16: In section 5.1.4, we mentioned that the following property cannot be specified in LTL: "the expression e is 1 in every even position." Draw a Büchi automaton M with one input variable e that accepts a trace exactly when it satisfies this property. ■

Exercise 5.17*: Complete the proof of proposition 5.2: for an accepting execution $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots$ of the automaton M_φ over the input trace $\rho = v_1 v_2 \cdots$, prove that for all $\psi \in Sub(\varphi)$, for all $i \geq 0$, $\psi \in q_i$ if and only if $(\rho, i+1) \models \psi$, by induction on the structure of the formula ψ. ■

5.2.3 Nested Depth-First Search *

Given a transition system T and a property φ, to check whether φ is repeatable, we search for a state that violates φ, is reachable from some initial state, and is contained in a cycle. The core computational problem here is detecting cycles. As discussed in section 3.3, we assume that the transition system is countably branching and is represented by the functions *FirstInitState*, *NextInitState*, *FirstSuccState*, and *NextSuccState* that can be used to enumerate initial states and successor states of a given state. We want our algorithm to be *on-the-fly*: it should explore states and transitions out of these states only as needed, and it should terminate returning a counterexample as soon as it finds one. Thus, the ideal algorithm should not first examine all the reachable states and then proceed to finding cycles. As a result, the classical algorithms for detecting cycles in a graph that rely on computation of strongly connected components in the graph are not best suited for our application (a strongly connected component in a directed graph is a maximal subset of the vertices such that there is a path between every pair of vertices in this subset). Instead, we will present a cycle-detection algorithm that employs two depth-first search traversals, one nested in the other.

The algorithm explores the reachable states of the transition system in a manner similar to the depth-first search algorithm of figure 3.16 using the stack *Pending* and the set *Reach* to store states already visited. The key difference is the following: while checking the reachability of a property, when a state satisfying the property is encountered, the search terminates; while checking the repeatability of a property, when a state s satisfying the property is encountered, the algorithm initiates another search for a cycle containing the state s. To implement this, suppose every time a state satisfying the property φ is encountered, a brand-new search to check whether the state s is reachable from itself is initiated, and this search uses its own set of visited states. While such a strategy would lead to a correct algorithm, it has time complexity that is quadratic in the number of states, and this can be significantly improved. The optimal algorithm is shown in figure 5.11.

The algorithm involves two nested searches: a primary search performed by the function *DFS* and a secondary (or nested) search performed by the function *NDFS*. The states encountered during the primary search are stored in the set *Reach*, whereas the states visited during the secondary search are stored in the set *NReach*. As in a standard depth-first search, for every reachable state s of T, the function *DFS* is invoked at most once with the state s as its input. Once the primary search originating at a state s terminates, if the state s satisfies the desired repeatable property φ, then it is a potential candidate for the cyclic counterexample. Then a secondary search is initiated by calling the function *NDFS* with the state s as its input. The objective of this secondary search is to find a cycle starting at the state s. When the function *NDFS*(s) is invoked, the stack *Pending* contains an execution starting from an initial state leading to the state s. The secondary search does not modify the stack *Pending*. Thus,

```
Input: A transition system T and property φ;
Output: If φ is a repeatable property of T return 1, else return 0;

set(state) Reach := EmptySet;
set(state) NReach := EmptySet;
stack(state) Pending := EmptyStack;
state s := FirstInitState(T);

while s ≠ null do {
  if Contains(Reach, s) = 0 then
    if DFS(s) = 1 then return 1;
  s := NextInitState(s, T);
  };
return 0.

bool function DFS(state s)
  Insert(s, Reach);
  Push(s, Pending);
  state t := FirstSuccState(s, T);
  while t ≠ null do {
    if Contains(Reach, t) = 0 then
      if DFS(t) = 1 then return 1;
    t := NextSuccState(s, t, T);
    };
  if Satisfies(s, φ) = 1 then
    if Contains(NReach, s) = 0 then
      if NDFS(s) = 1 then return 1;
  Pop(Pending);
  return 0.

bool function NDFS(state s)
  Insert(s, NReach);
  state t := FirstSuccState(s, T);
  while t ≠ null do {
    if Contains(Pending, t) = 1 then return 1;
    if Contains(NReach, s) = 0 then
      if NDFS(t) = 1 then return 1;
    t := NextSuccState(s, t, T);
    };
  return 0.
```

Figure 5.11: Nested Depth-first Search Algorithm for Checking Repeatability

if the secondary search encounters a transition leading to a state belonging to the stack, then it concludes that there is a cycle that contains the state s. This establishes that whenever the algorithm returns 1, the transition system contains a reachable cycle containing a state that satisfies the property φ.

The secondary search uses a separate set *NReach* to keep track of states encountered during the secondary search. However, this set is shared across all calls to *NDFS*: every time the function *NDFS* is called with a state s as its input, the state s is added to this set, and *NDFS* is invoked with a state t as input only if the state t is not in the set *NReach*. Thus, the secondary search explores a reachable state at most once, and the total running time of the secondary search is the same as the primary search. To understand the interplay between the two searches and the argument about the correctness of the search strategy, consider two states s and t that are encountered by the primary search, and suppose both satisfy the property φ. Suppose the secondary search is invoked from the state s first, and it explores all states that are reachable from the state s, adding them to the set *NReach* but without finding a cycle. Later, when the secondary search is invoked from the state t, it will just skip over states that were added to the set *NReach*. What guarantees that this does not cause the algorithm to miss detection of a cycle containing the state t?

To answer this question, let us order the states according to the termination times of the primary search: with each reachable state s, associate a number d_s such that if the call *DFS*(s) terminates before the call *DFS*(t), then $d_s < d_t$. Let $s_0, \ldots s_k$ be the ordering of the states that are reachable and satisfy the property φ according to this numbering. Let s_i be the first state in this ordering that belongs to a cycle, and let Q be the set of all states that are reachable from the states s_j with $j < i$.

We first claim that the cycle that contains the state s_i is disjoint from the set Q. If not, there is a state t belonging to the set Q such that both states s_i and t belong to a cycle. This implies that the state s_i is reachable from some state s_j with $j < i$ (since the state t must be reachable from some such state). Thus, the exploration from the state s_j is guaranteed to examine state s_i, but given the ordering of the states, we know that the call *DFS*(s_j) terminates before the call *DFS*(s_i). This can happen only if the call *DFS*(s_i) is pending when *DFS*(s_j) is invoked. This implies that the state s_j is also reachable from the state s_i, and thus the state s_j is involved in a cycle, a contradiction to the assumption that the state s_i is the first state in the ordering that belongs to a cycle.

When the primary search from the state s_i terminates, the set *NReach* containing the states visited by the secondary search so far equals the set Q. Since the state s_i does not belong to the set Q, the function *NDFS* will be invoked with the state s_i as its input. Since there is a cycle that contains the state s_i and does not involve any of the states already in the set *NReach*, the secondary search is guaranteed to discover this cycle.

To understand how the algorithm works, let us consider the transition system shown in figure 5.12. The initial state is A, and the property is satisfied in

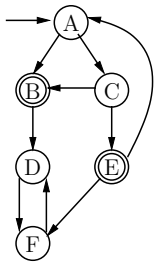

Figure 5.12: Sample Transition System for Illustrating Repeatability Algorithms

states B and E. The execution of the algorithm is illustrated in figure 5.13. The first column lists the calls made to the functions *DFS* and *NDFS*, where the indentation indicates which calls are pending. The subsequent columns list the values of the stack *Pending* (left-most element is at the top of the stack) and the sets *Reach* and *NReach*.

Initially, the function *DFS* is called with the initial state A as input, which invokes $DFS(B)$, which in turn calls $DFS(D)$, which then invokes $DFS(F)$. Since the sole successor of the state F is already in the set *Reach*, the call $DFS(F)$ terminates. Thus, $d_F = 1$. Subsequently, the call $DFS(D)$ also terminates ($d_D = 2$). At this point, in the execution of $DFS(B)$, the state B has no more successor states, but since the state B satisfies the desired property, the secondary search is initiated for the first time via the call $NDFS(B)$. This in turn calls $NDFS(D)$ and then calls $NDFS(F)$. All these calls terminate reporting failure: the stack *Pending* contains the states A and B, and no transitions leading to either of these states are encountered. When the call $DFS(B)$ terminates ($d_B = 3$), the set *NReach* contains all the states reachable from the state B, namely, the states B, D, and F. The primary search from the state A now proceeds to call $DFS(C)$, which in turn calls $DFS(E)$. All the successors of the state E are already in the set *Reach*, but since the state E satisfies the property, another secondary search is invoked using the call $NDFS(E)$. At this point, the states B, D, and F are already assumed to be visited for the secondary search, and hence the transition from the state E to F will not be explored for establishing the repeatability of the state E. This is justified by the correctness argument for the algorithm: since $DFS(B)$ has already terminated without discovering a cycle containing the state B, a cycle containing the state E cannot involve states reachable from the state B. As it turns out, the state E has a successor state, namely, A, which is in the stack. As a result, the call $NDFS(E)$ terminates reporting success, causing all the pending calls to terminate with return value 1.

Note that, just like the depth-first search algorithm of figure 3.16, the nested depth-first search algorithm may detect a cycle before exploring all the reachable

	Pending	*Reach*	*NReach*
$DFS(A)$	$[\,]$	$\{\,\}$	$\{\,\}$
$DFS(B)$	$[A]$	$\{A\}$	$\{\,\}$
$DFS(D)$	$[B, A]$	$\{A, B\}$	$\{\,\}$
$DFS(F)$	$[D, B, A]$	$\{A, B, D\}$	$\{\,\}$
$NDFS(B)$	$[B, A]$	$\{A, B, D, F\}$	$\{\,\}$
$NDFS(D)$	$[B, A]$	$\{A, B, D, F\}$	$\{B\}$
$NDFS(F)$	$[B, A]$	$\{A, B, D, F\}$	$\{B, D\}$
$DFS(C)$	$[A]$	$\{A, B, D, F\}$	$\{B, D, F\}$
$DFS(E)$	$[C, A]$	$\{A, B, D, F, C\}$	$\{B, D, F\}$
$NDFS(E)$	$[E, C, A]$	$\{A, B, D, F, C, E\}$	$\{B, D, F\}$

Figure 5.13: Illustrative Execution of the Nested Depth-first Search Algorithm

states. For instance, for the transition system of figure 5.12, if we introduce a transition from the state D to state B, then the call $NDFS(B)$ will discover a cycle, and the algorithm terminates without ever visiting the states C or E. If the number of reachable states of the transition system is finite, then it is guaranteed to terminate with the correct answer. These properties are summarized in the theorem below.

Theorem 5.2 [Nested Depth-first Search for Repeatability Checking] *Given a countably branching transition system T and a property φ, the nested depth-first search algorithm of figure 5.11 has the following guarantees:*

1. *If the algorithm terminates, then the returned value correctly indicates whether the property φ is a repeatable property of T.*
2. *If the number of reachable states of T is finite, then the algorithm terminates, and the number of calls to DFS and to NDFS are bounded by the number of reachable states.*

■

Exercise 5.18: Modify the nested depth-first search algorithm of figure 5.11 so that it outputs a counterexample consisting of an execution leading from an initial state to a state s satisfying φ and a cyclic execution from s back to itself. ■

5.2.4 Symbolic Repeatability Checking

Recall the symbolic breadth-first search algorithm for invariant verification by iterative image computation discussed in section 3.4. We will now develop a symbolic nested search algorithm to check whether the transition system has an infinite execution that repeatedly visits a given property. As before, we assume that a set of states over a set V of typed variables is represented as a region of type `reg`. In the symbolic representation of a transition system with state

variables S, the initial states are represented by a region *Init* over the variables S, and the transitions are represented by a region *Trans* over the variables $S \cup S'$. The property φ whose repeatability is to be checked is also represented by a region over the variables S. The data type `reg` for regions supports operations such as `Conj`, `Diff`, and `IsSubset`, as discussed in section 3.4.

Image and Pre-image Computation

The core step of symbolic verification algorithms is image computation. Given a region A over the state variables S, the region that contains all the states that can be reached from the states in A using one transition can be computed using the `Post` operation defined as:

$$\texttt{Post}(A, \mathit{Trans}) \;=\; \texttt{Rename}(\texttt{Exists}(\texttt{Conj}(A, \mathit{Trans}), S), S', S).$$

The dual operator corresponds to the pre-image computation: given a region A over the state variables, the region that contains all the states from which some state in A can be reached using one transition is called the *pre-image* of the region A. Given a region A, to compute its pre-image, we first rename the unprimed variables to primed variables and then intersect it with the transition region *Trans* over $S \cup S'$ to obtain all the transitions that lead to the states in A. Then we project the result onto the set S of unprimed state variables by existentially quantifying the variables in S'. Thus, the pre-image operator `Pre` is defined as:

$$\texttt{Pre}(A, \mathit{Trans}) \;=\; \texttt{Exists}(\texttt{Conj}(\texttt{Rename}(A, S, S'), \mathit{Trans}), S').$$

Let us consider an example to illustrate the pre-image computation. Suppose the system has a single variable x of type `real`, and the update is given by the following conditional statement

$$\texttt{if}\ (\,1 \le x \le 5\,)\ \texttt{then}\ x := x - 1\ \texttt{else}\ x := x + 1.$$

Then the transition region is given by the formula

$$[\,(1 \le x \le 5)\ \wedge\ (x' = x - 1)\,]\ \vee\ [\,((1 > x)\ \vee\ (x > 5))\ \wedge\ (x' = x + 1)\,].$$

Consider the region A given by the formula $1 \le x \le 2$, and let us apply the sequence of steps needed to compute the pre-image of the region A. First, we rename the variable x to x', and this gives us the formula $1 \le x' \le 2$. Then we conjoin this region with the transition formula *Trans*, and this gives the result, which simplifies to:

$$\begin{array}{rl} & [\,(\,1 \le x \le 5\,)\ \wedge\ (\,x' = x - 1\,)\ \wedge\ (\,1 \le x' \le 2\,)\,] \\ \vee & [\,(\,(1 > x)\ \vee\ (x > 5)\,)\ \wedge\ (\,x' = x + 1\,)\ \wedge\ (\,1 \le x' \le 2\,)\,]. \end{array}$$

The final step is to apply the operation of existential quantification to eliminate the variable x' leading to the formula

$$[\,(1 \le x \le 5)\ \wedge\ (1 \le x - 1 \le 2)\,]\ \vee\ [\,((1 > x)\ \vee\ (x > 5))\ \wedge\ (1 \le x + 1 \le 2)\,].$$

```
Input: A transition system T given by a region Init for initial states,
    a region Trans for transitions, and a region φ for the property;
Output: If φ is repeatable in T, return 1, else return 0.

reg Reach := Empty;
reg New := Init;
while IsEmpty(New) = 0 do {
  Reach := Disj(Reach, New);
  New := Diff(Post(New, Trans), Reach);
  };
reg Recur := Conj(Reach, φ);
while IsEmpty(Recur) = 0 do {
  Reach := Empty;
  New := Pre(Recur, Trans);
  while IsEmpty(New) = 0 do {
    Reach := Disj(Reach, New);
    if IsSubset(Recur, Reach) = 1 then return 1;
    New := Diff(Pre(New, Trans), Reach);
    };
  Recur := Conj(Recur, Reach);
  };
return 0.
```

Figure 5.14: Symbolic Nested Search Algorithm for Checking Repeatability

This formula simplifies to

$$(2 \leq x \leq 3) \ \vee \ (0 \leq x < 1)$$

which precisely describes the desired set of values of x for which executing the conditional assignment causes the resulting value to belong to the interval $[1, 2]$.

Nested Symbolic Search

The symbolic algorithm for checking repeatability shown in figure 5.14 uses both image computation and pre-image computation. The algorithm has two phases: the first phase consists of a single while loop, and the second phase consists of two nested while loops.

The first phase of the algorithm computes the region *Reach* of all states reachable from the region *Init* of initial states by repeatedly applying the image-computation operator `Post`. This is similar to the algorithm of figure 3.19. Let us illustrate the algorithm using the sample transition system shown in figure 5.12. Figure 5.15 shows the values of the regions *Reach* and *New* at the beginning and, after each iteration, during the first phase of the algorithm.

The second phase attempts to find an infinite execution with repeating occurrences of the property φ. The set of states whose repeated occurrence indicates

	Reach	*New*
Initially	$\{\ \}$	$\{A\}$
After iteration 1	$\{A\}$	$\{B, C\}$
After iteration 2	$\{A, B, C\}$	$\{D, E\}$
After iteration 3	$\{A, B, C, D, E\}$	$\{F\}$
After iteration 4	$\{A, B, C, D, E, F\}$	$\{\ \}$

Figure 5.15: Illustrative Execution of the First Phase of Algorithm 5.14

success is captured by the region *Recur*. This region initially contains all the reachable states that satisfy the property φ and can be computed by intersecting the region *Reach* computed at the end of the first phase and the region representing the property. Let us call this region $Recur_0$. For each of the states s in this set, we want to determine if there exists an execution consisting of one or more transitions starting in the state s and ending in some state in $Recur_0$. To compute this information, the inner loop repeatedly applies the pre-image computation to find those states from which states in $Recur_0$ can be reached in one or more transitions. This computation is similar to the computation of the reachable states: the region *Reach*, initialized to the empty set, contains the states already examined; and the region *New*, initialized to the states from which the current set *Recur* can be reached in one transition, contains the states to be explored. In each iteration of the inner loop, the region *Reach* is updated by adding the unexplored states in the region *New*. The set of states to be newly explored is obtained by computing the pre-image of the current set *New* and removing the already explored states in *Reach* using the set-difference operation. The inner loop terminates when there are no more new states to be examined. At this point, the region *Reach* contains precisely those states that have a path to some state in $Recur_0$. By intersecting this region with *Recur*, we obtain the set $Recur_1$, a subset of $Recur_0$. The outer loop is now repeated again with this revised value of *Recur*.

To illustrate the second phase of the algorithm, let us continue with our example transition system of figure 5.12. Figure 5.16 shows the values of the regions *Recur*, *Reach*, and *New* at each iteration. Initially, the set *Recur* contains the states B and E as potential candidates for the repeating states. One iteration of the outer loop discovers that there is no execution from the state B that can lead back to this set; as a result, *Recur* gets updated to $\{E\}$. During the second iteration of the outer loop, the inner loop computes the set of states from which the state E can be reached. In the third iteration of the inner loop, the state E gets added to the region *Reach*, and this causes the successful termination of the algorithm.

Outer loop	*Recur*	Inner loop	*Reach*	*New*
Initially	$\{B, E\}$			
		Initially	$\{\ \}$	$\{A, C\}$
		After iteration 1	$\{A, C\}$	$\{E\}$
		After iteration 2	$\{A, C, E\}$	$\{\ \}$
After iteration 1	$\{E\}$			
		Initially	$\{\ \}$	$\{C\}$
		After iteration 1	$\{C\}$	$\{A\}$
		After iteration 2	$\{C, A\}$	$\{E\}$
		During iteration 3	$\{C, A, E\}$	

Figure 5.16: Illustrative Execution of the Second Phase of Algorithm 5.14

Correctness

Let $\mathit{Recur}_1, \mathit{Recur}_2, \ldots$ be the successive values assigned to the region variable *Recur* at the end of the outer while loop. Each such set Recur_i is a subset of the set Recur_{i-1} and contains those states in Recur_{i-1} from which some state in Recur_{i-1} can be reached using an execution with one or more transitions. Hence, each such set Recur_i contains the states s such that the state s is reachable from some initial state, the state s satisfies the property φ, and there is an execution starting from the state s that encounters states satisfying the property φ at least i times.

Suppose the property φ is repeatable for the transition system T. Then there is a state s that is reachable from some initial state, the state s satisfies φ, and there is an infinite execution starting from the state s that encounters states satisfying the property φ infinitely many times. Such a state s will belong to every set Recur_i and thus will never be removed from *Recur*. As a result, if the value of *Recur* becomes the empty set at any point during the execution of the algorithm, no such state s exists, and the algorithm can terminate claiming that the property φ is not repeatable.

Conversely, suppose for the current nonempty set Recur_i, from every state in Recur_i, some state in Recur_i can be reached by an execution with one or more transitions. Then in the subsequent iteration of the outer loop, the final value of the set *Reach* is a superset of Recur_i. As the region *Reach* is computed iteratively by adding more and more states that can reach Recur_i in one or more transitions, when the algorithm finds that *Reach* is a superset of *Recur*, it terminates reporting repeatability of the property φ. We argue that in this case, indeed there is an infinite execution in which the property φ repeats. Let s_0 be any state in Recur_i. Since the set Recur_i is a subset of Recur_0, the state s_0 is reachable from an initial state. Hence, it suffices to demonstrate that there exists an infinite execution starting at the state s_0 with the property φ repeating infinitely often. From the state s_0, there is an execution with one or

more transitions leading to some state, say s_1, in $Recur_i$. From the state s_1, there is an execution with one or more transitions leading to some state, say s_2, in $Recur_i$. This process can be repeated forever. Concatenating all these finite executions gives us the desired infinite execution with repeating φ.

Complexity Analysis

If the number of reachable states of T is finite, then the termination is guaranteed. Suppose the number of reachable states of T is n, and k of these satisfy the property φ. Then the set $Recur_0$ contains k states, and the number of iterations of the outer loop is at most k (since states are only removed from *Recur*, and the algorithm terminates if the value of *Recur* does not change). In each iteration of the outer loop, the inner loop can be executed at most n times as it computes the set of states from which *Recur* is reachable, in an iterative manner. The actual running time of the algorithm depends on how efficiently the various operations on regions are executed, but the number of symbolic operations is quadratic.

The correctness and complexity of the algorithm are summarized below.

Theorem 5.3 [Symbolic Nested Search for Checking Repeatability] *Given a symbolic representation of a transition system T and a property φ, the symbolic nested search algorithm of figure 5.14 has the following guarantees:*

1. *If the algorithm terminates, then the returned value correctly indicates whether the property φ is repeatable for the transition system T.*
2. *If the transition system T has n reachable states, of which k satisfy the property φ, then the algorithm terminates after at most $O(nk)$ operations on regions.*

■

Exercise 5.19: Consider a transition system with two variables x and y of type `nat`. Suppose the transitions of the system are described by the conditional statement

$$\texttt{if } (x > y) \texttt{ then } x := x + 1 \texttt{ else } y := x.$$

First, describe the transition region as a formula *Trans* over the variables x, y, x', and y'. Consider the region A given by the formula $1 \leq y \leq 5$. Compute the pre-image of the region A. ■

Exercise 5.20*: Algorithm of figure 5.14 uses both post-image computation and pre-image computation. Suppose we modify the algorithm by replacing both calls to `Pre` by `Post`, that is, in the second phase, replace the assignment *New* := `Pre`(*Recur*, *Trans*) by *New* := `Post`(*Recur*, *Trans*) and the assignment *New* := `Diff`(`Pre`(*New*, *Trans*), *Reach*) by *New* := `Diff`(`Post`(*New*, *Trans*), *Reach*). How will this modification impact the correctness of the algorithm? Justify your answer. ■

5.3 Proving Liveness *

In section 3.2.1, we studied a general proof technique for establishing invariants of transition systems: given a transition system T and a property φ over its state variables, to prove that the property φ is an invariant of the transition system T, we find another property ψ and show that (1) the property ψ is an *inductive* invariant of the transition system T, and (2) the property ψ implies the property φ. This proof technique based on inductive invariants is appealing for the following reasons. First, the method is rooted in the intuitive and informal argument needed to convince oneself about the correctness of the system. Second, the formalization of the rule is mathematically precise, and the rule can be used to produce a machine-checkable proof of the correctness of the system. Third, the rule is general enough so that every invariant property can be established by applying the rule. Now, we focus on identifying proof techniques for proving liveness properties of transition systems.

Consider a transition system T with the state variables S. Liveness properties of such a transition system can be expressed using LTL-formulas over the set S of variables: the transition system T satisfies the LTL-formula φ if every infinite execution of T satisfies φ. The precise details of proof rules for establishing that the transition system satisfies the LTL-formula φ depend on the structure of the formula φ. We focus on the most commonly occurring patterns: eventuality properties and response properties assuming weak fairness.

5.3.1 Eventuality Properties

Let us revisit the leader election protocol of section 2.4.3. We want to establish that every node eventually makes a decision. Since a node announces its decision by updating the output variable *status* when the value of its state variable r equals N, where N is the total number of nodes in the network, we want to show that eventually the value of r becomes N. More precisely, we want to prove that every infinite execution of the transition system corresponding to the protocol satisfies the eventually formula $\Diamond\,(r_n = N)$, where n is an arbitrary node.

To convince yourself that the component of figure 2.35 satisfies the eventuality formula $\Diamond\,(r_n = N)$, observe that the value of the variable r_n is initially 1, and in each round, it is incremented by 1 as long as it is less than N. To formally capture the intuition behind this argument, let us define a function *rank* from the states of the transition system to natural numbers: in a given state s of the transition system, the value of $\mathit{rank}(s)$ is the difference between N and the value that the state s assigns to the variable r_n. The function *rank* captures the *distance* of a state from the desired eventuality goal. To prove that every execution satisfies the eventuality formula $\Diamond\,(r_n = N)$, we show that if a state s does not already satisfy the desired eventuality (that is, the value of the round variable in the state s is not yet N), then executing the protocol for one more step *decreases* the rank; that is, if (s, t) is a transition of the

system, then $\mathit{rank}(t) < \mathit{rank}(s)$. Since the rank is a natural number, it cannot decrease forever, implying that a state satisfying the desired eventuality must be encountered in finitely many steps.

The function *rank* needs to map every state, and not just the states that we informally know to be reachable, to a natural number. In our example, we define the rank $\mathit{rank}(s)$ of a given state s to be $N - s(r_n)$ if $s(r_n) \leq N$ and 0 otherwise. For a state s that assigns, say, the value $N + 1$ to the variable r_n, the rank is 0, and executing a transition in such a state does not decrease the rank. However, we know that such a state is unreachable. More precisely, we show that $0 \leq r_n \leq N$ is an invariant of the system, and this can be established using the proof technique already studied. To show that the execution of one transition decreases the rank, it now suffices to focus on those states that satisfy this invariant. That is, we show that for every state s of the system, assuming that the state s satisfies the invariant $0 \leq r_n \leq N$, if (s, t) is a transition of the system, then either the state t satisfies the eventuality goal or its rank is strictly smaller than the rank of the state s.

This reasoning is summarized in the following proof rule for establishing eventuality properties:

> Proof rule for Eventuality properties
>
> To establish that a transition system T satisfies the eventuality formula $\Diamond\, \varphi$, where φ is a property over the state variables of T, identify a state property ψ and a function *rank* that maps states of T to `nat` and show that:
>
> 1. the property ψ is an invariant of T; and
> 2. for every state s that satisfies ψ and every transition (s, t) of T, either the state t satisfies φ or $\mathit{rank}(t) < \mathit{rank}(s)$.

To establish that the proof rule is sound, we need to argue that if we establish the two premises (1) and (2) of the rule, then the transition system must satisfy the formula $\Diamond\, \varphi$. Consider an infinite execution $s_0, s_1, s_2, \ldots$ of the transition system. Clearly, each state s_j appearing in the execution is reachable and by the first premise satisfies the invariant property ψ. To show that some state s_j in the execution must satisfy the desired eventuality φ, assume to the contrary. Then by the second premise, since there is a transition between every pair of adjacent states (s_j, s_{j+1}), we have that $\mathit{rank}(s_{j+1}) < \mathit{rank}(s_j)$ for each $j \geq 0$. However, this is not possible: if $\mathit{rank}(s_0)$ is K, then the rank can decrease at most K times as the rank of each state is a non-negative number and thus cannot decrease at each step of an infinite execution.

Exercise 5.21: Recall the transition system $\texttt{GCD}(m, n)$ of figure 3.1 capturing the program for computing the greatest common divisor of two numbers m and n. Suppose we want to establish that the program terminates, that is, eventually *mode* equals `stop`. Prove this eventuality property using the proof

rule for eventuality formulas by selecting an appropriate invariant and a ranking function. ■

5.3.2 Conditional Response Properties

A canonical liveness property is the response property "every request φ_1 is eventually followed by the response φ_2," expressed by the LTL-formula $\Box\,(\varphi_1 \rightarrow \Diamond\,\varphi_2)$.

Recall the rule for establishing the eventuality formula $\Diamond\,\varphi$: we find a ranking function that maps states to natural numbers, identify an invariant property ψ, and show that executing a transition in any state that satisfies the invariant causes either the fulfillment of the goal or a decrease in the rank. Let us first consider how to generalize this reasoning to establish the response formula $\Box\,(\varphi_1 \rightarrow \Diamond\,\varphi_2)$. Now we want to show that whenever the property φ_1 holds, executing a sequence of system transitions must result in a state satisfying the goal property φ_2. We again use a ranking function that maps states to natural numbers and show that the rank keeps decreasing until the goal is satisfied. This is achieved by the following proof principle:

> PROOF RULE FOR RESPONSE PROPERTIES
>
> To establish that a transition system T satisfies the response formula $\Box\,(\varphi_1 \rightarrow \Diamond\,\varphi_2)$, where φ_1 and φ_2 are state properties, identify a state property ψ and a function *rank* that maps states of T to `nat` and show that:
>
> 1. every state that satisfies the property φ_1 also satisfies the property ψ, and
> 2. for every state s that satisfies the property ψ and for every transition (s,t) of T, either the state t satisfies the response property φ_2 or the state t satisfies the property ψ and $\mathit{rank}(t) < \mathit{rank}(s)$.

As in the case of the eventuality rule, the property ψ captures the states from which executing a transition causes either the fulfillment of the goal or a decrease in the rank. However, instead of requiring that the property ψ be an invariant of the system, we require that whenever the request property φ_1 holds, the property ψ should hold, and it should continue to hold until the response φ_2 is satisfied.

To illustrate the proof technique for establishing response properties, let us consider a transition system with two variables x and y of type `int`. Suppose initially x equals 1 and y equals 0, and the transitions of the system correspond to executing the following code at each step:

$$\texttt{if}\ (\,x > 0\,)\ \texttt{then}\ \{x := x - 1;\ y := y + 1\}\ \texttt{else}\ x := y.$$

The sequence of values of x is $1, 0, 1, 0, 2, 1, 0, 4, 3, 2, 1, 0, 8, 7, \ldots$ and thus, the program satisfies the recurrence property $\Box\,\Diamond\,(x = 0)$. To prove that the system

satisfies this recurrence formula, we can apply the rule for response properties using φ_1 as 1 (that is, always true) and φ_2 as $(x = 0)$. We choose ψ to be the same as φ_1, and thus premise 1 of the rule holds immediately. Consider a state $s = (a, b)$ (that is, x equals a and y equals b). The ranking function *rank* maps such a state s to a if $a > 0$ and to $b+1$ if $a = 0$. To establish premise 2, consider a state $s = (a, b)$. If $a > 0$, then $\mathit{rank}(s) = a$, and letting the system execute one step leads to the state $t = (a - 1, b + 1)$. If $a = 1$, then the state t satisfies the goal $(x = 0)$ (and in this case, $\mathit{rank}(t) = b + 2$, which could be much higher than $\mathit{rank}(s)$), and if $a > 1$, then $a - 1 > 0$ and $\mathit{rank}(t) = a - 1$, which is less than $\mathit{rank}(s)$. However, if $a = 0$, then $\mathit{rank}(s) = b + 1$, and letting the system execute one step leads to the state $t = (b, b)$ with $\mathit{rank}(t) = b$, which is less than $\mathit{rank}(s)$. This means that the proof rule is applicable and allows us to conclude that the system satisfies $\Box \Diamond (x = 0)$.

Establishing Eventual Delivery of Messages for Merge

Let us revisit the example of the asynchronous process `Merge` of figure 4.3. Suppose we want to establish that if a message v is received on the input channel in_1, it will eventually be delivered on the output channel *out*. This is captured by the response property:

$$\Box\, (\, in_1\,?\,v \;\rightarrow \Diamond\; out\,!\,v\,).$$

Here, the request property φ_1 is $in_1\,?\,v$, and its fulfillment is the response property $\varphi_2 = out\,!\,v$. We first choose the strengthening ψ of the request property to be $\texttt{Contains}(x_1, v)$. Note that whenever the process executes the input action $in_1\,?\,v$, the message v is enqueued in the state variable x_1, and thus, the property $\texttt{Contains}(x_1, v)$ holds.

To define the ranking function, consider the following question: when a message v is in the queue x_1, which quantity do we expect to monotonically decrease until the message v gets removed from the queue? It is the number of messages queued up ahead of the message v. Hence, given a state s, let $\mathit{rank}(s)$ be 0 if the queue $s(x_1)$ does not contain the message v; otherwise, let $\mathit{rank}(s)$ be k if v is the kth message in the queue $s(x_1)$. For instance, if the queue x_1 contained five messages when the message v gets enqueued, then it will be the sixth message in the queue, and the rank will be 6. Whenever a message gets dequeued from the queue x_1 (and transmitted on the output channel), the message v moves up one slot, causing the rank to become 5. This process repeats until the message v is at the front of the queue. In such a state, the rank is 1, and dequeuing a message from the queue x_1 causes the rank to become 0, and in the resulting state, the goal property $out\,!\,v$ is satisfied. Now the condition $\texttt{Contains}(x_1, v)$ no longer holds, and the rank stays unchanged. If at a later step the message v is received again on the input channel in_1, it gets enqueued in the queue x_1, and the rank can increase arbitrarily. For instance, if the size of the queue x_1 is 12 upon the arrival of the next instance of the message v on the input channel in_1, then it gets enqueued in the 13th slot, and the rank is 13. It keeps decreasing

from 13 down to 0 until the message v is again transferred from the queue x_1 to the output channel.

In order to apply the proof rule for the response property, let us check the premises with the property φ_1 equal to $in_1\,?\,v$, the property φ_2 equal to $out\,!\,v$, and the property ψ equal to $\texttt{Contains}(x_1, v)$. Clearly, the premise 1 of the rule holds: a state satisfying the property φ_1 is guaranteed to satisfy the property ψ. Furthermore, in a transition (s, t), where the state s satisfies ψ (that is, the message v is in the queue x_1 in state s), and the state t does not satisfy the goal (that is, the message v is not transmitted from the queue in_1 to the output channel during this transition), we are guaranteed that the state t continues to satisfy the property ψ. Intuitively, the property $\texttt{Contains}(x_1, v)$ continues to stay true and is preserved in every transition until the goal property holds.

However, the proof of the response formula fails. The second premise of the rule requires that in states satisfying $\texttt{Contains}(x_1, v)$, executing a transition either satisfies the goal property $out\,!\,v$ or causes the rank to decrease. That is, when the message v is in the queue x_1, executing a step of the process `Merge`, either leads to a state with the message v is output on the output channel or causes the message v to shift one slot in the queue x_1. This condition is satisfied only by the execution of the output task A_1^o that dequeues a message from the queue x_1 and transmits it on the channel out. If the transition corresponds to executing any task other than the task A_1^o, then while the condition $\texttt{Contains}(x_1, v)$ continues to hold, the rank stays the same. If the task A_1^o is never executed, then the rank will stay unchanged, and the message v will never be output. In fact, not all executions of the process `Merge` satisfy the response formula $\Box\,(\,in_1\,?\,v\;\rightarrow\,\Diamond\;out\,!\,v\,)$. The formula, however, is satisfied by all the executions that satisfy the weak fairness assumption for the task A_1^o. We need to *strengthen* the proof rule for response properties in order to establish such *conditional* response properties.

Proof Rule for Conditional Response

Suppose we want to establish that the process `Merge` satisfies the response formula $\Box\,(\,in_1\,?\,v\;\rightarrow\,\Diamond\;out\,!\,v\,)$ assuming weak fairness for the output task A_1^o. We use the same choice for ψ, namely, $\texttt{Contains}(x_1, v)$, and the same ranking function, namely, for a given state s, $rank(s)$ is 0 if the queue $s(x_1)$ does not contain the message v, and $rank(s)$ is k if v is the kth message in the queue $s(x_1)$. Consider a state s that satisfies the property $\texttt{Contains}(x_1, v)$ and a transition (s, t) of the process such that the transition does not involve sending the message v on the output channel. Instead of insisting that $rank(t) < rank(s)$ as required by the earlier response rule, we now consider two cases. When the transition from the state s to the state t involves the execution of the task A_1^o, we require that $rank(t) < rank(s)$; otherwise, it suffices that the rank does not decrease (that is, $rank(t) \leq rank(s)$), but the task A_1^o should be enabled in the state t. Intuitively, the task A_1^o has the responsibility to decrease the rank and, hence, cause progress toward the fulfillment of the goal. The execution of a task

other than the task A_1^o should maintain enabledness of the task A_1^o without increasing the rank. The weak fairness assumption for the task A_1^o ensures that if the task A_1^o stays continuously enabled, it will eventually be executed, and this would decrease the rank.

Establishing the response formula assuming weak fairness for the task A_1^o is the same as establishing the following conditional response formula for all executions:

$$\Box\,[\,\mathit{Guard}(A_1^o)\ \rightarrow\ \Diamond\,(\,(\mathit{taken} = A_1^o)\ \vee\ \neg\,\mathit{Guard}(A_1^o)\,)\,] \\ \rightarrow\ \Box\,[\,\mathit{in}_1\,?\,v\ \rightarrow \Diamond\ \mathit{out}\,!\,v\,].$$

The proof rule formalized below shows how to establish the response formula $\Box\,[\,\varphi_1\ \rightarrow\ \Diamond\,\varphi_2\,]$ under the assumption of the form $\Box\,[\,\psi_1\ \rightarrow\ \Diamond\,(\,\psi_2 \vee \neg\psi_1\,)\,]$. When the property ψ_1 is $\mathit{Guard}(A)$ and the property ψ_2 is $(\mathit{taken} = A)$, the assumption corresponds to the weak fairness assumption for the task A. If we pick the property ψ_1 to be the constant 1, then the assumption simplifies to $\Box\Diamond\,\psi_2$.

Proof rule for Conditional Response properties

To establish that a transition system T satisfies the conditional response formula

$$\Box\,[\,\psi_1\ \rightarrow\ \Diamond\,(\psi_2\ \vee\ \neg\,\psi_1)\,]\ \rightarrow\ \Box\,[\varphi_1\ \rightarrow\ \Diamond\ \varphi_2],$$

where φ_1, φ_2, ψ_1, and ψ_2 are state properties, identify a state property ψ and a function *rank* that maps states of T to `nat` and show that:

1. every state that satisfies the property φ_1 also satisfies the property ψ,
2. every state that satisfies the property ψ also satisfies the property ψ_1, and
3. for every state s that satisfies the property ψ and for every transition (s,t) of T, either the state t satisfies the response φ_2 or the state t satisfies the property ψ and $\mathit{rank}(t) \leq \mathit{rank}(s)$ and if the state t satisfies the property ψ_2 then $\mathit{rank}(t) < \mathit{rank}(s)$.

Let us summarize the proof that the process `Merge` satisfies the response property $\Box\,(\,\mathit{in}_1\,?\,v\ \rightarrow\ \Diamond\ \mathit{out}\,!\,v\,)$ assuming weak fairness for the output task A_1^o using the notation of this proof rule. For the proof rule above, we have the property φ_1 equal to $\mathit{in}_1\,?\,v$, the property φ_2 equal to $\mathit{out}\,!\,v$, the property ψ_1 equal to $\mathit{Guard}(A_1^o)$, and the property ψ_2 equal to $(\mathit{taken} = A_1^o)$. We choose the property ψ to be $\texttt{Contains}(x_1, v)$ and *rank* to be the function that maps a state s to the position of the message v in the queue x_1 (and 0 if the message is not in the queue). Verify that all three premises as required by the rule indeed are satisfied.

We conclude by sketching a proof that the reasoning behind the proof rule for conditional response is *sound*: if we show all the premises of the rule, then

the desired conditional response formula is indeed satisfied by every execution of the transition system. It turns out that this proof rule, coupled with the application of valid temporal patterns such as the chain rule, is *complete*: if the transition system indeed satisfies the conditional response formula, there exist appropriate choices for the strengthening property ψ and the ranking function *rank* for which all the premises of the rule hold.

Theorem 5.4 [Soundness of the Proof Rule for Conditional Response] *The proof rule for establishing that a transition system T satisfies the conditional response formula φ given by $\Box[\,\psi_1 \rightarrow \Diamond(\psi_2 \vee \neg\psi_1)\,] \rightarrow \Box[\,\varphi_1 \rightarrow \Diamond\,\varphi_2\,]$ is sound.*

Proof. Let T be a transition system, and consider the LTL-formula φ of the above form. Let ψ be a state property and *rank* be a function that maps states of T to `nat`, such that (1) every state that satisfies the property φ_1 also satisfies the property ψ, (2) every state that satisfies the property ψ also satisfies the property ψ_1, and (3) for every state s that satisfies the property ψ and every transition (s,t) of T such that the state t does not satisfy the property φ_2, the state t satisfies the property ψ, and $\mathit{rank}(t) \leq \mathit{rank}(s)$, and if the state t satisfies the property ψ_2, then $\mathit{rank}(t) < \mathit{rank}(s)$. Under these assumptions, we want to show that every infinite execution of T satisfies the formula φ.

Let $\rho = s_0 s_1 s_2 \cdots$ be an infinite execution of the transition system T. Assume that the execution ρ satisfies the formula $\Box[\,\psi_1 \rightarrow \Diamond(\psi_2 \vee \neg\psi_1)\,]$. We want to show that the execution ρ satisfies the formula $\Box[\varphi_1 \rightarrow \Diamond\,\varphi_2]$. Let i be a position in the execution. Assume that the state s_i satisfies the request property φ_1. We want to establish that there exists a position $j \geq i$ such that the state s_j satisfies the response property φ_2. We will prove this by contradiction. Assume that the property φ_2 is not satisfied in every state s_j for $j \geq i$.

We claim that for every position $j \geq i$, the state s_j satisfies the property ψ. The proof is by induction on j, where $j = i$ is the base case. Since the state s_i satisfies the property φ_1, by premise 1, it also satisfies the property ψ. Now consider an arbitrary state s_j, with $j \geq i$. Assume that the state s_j satisfies the property ψ. There is a transition from the state s_j to the state s_{j+1} since they appear as consecutive states along the execution ρ. By assumption, the state s_{j+1} does not satisfy the property φ_2. By premise 3, we conclude that the state s_{j+1} satisfies the property ψ.

By premise 2, we conclude that every state s_j, for $j \geq i$, also satisfies the property ψ_1. Since the execution ρ satisfies the formula $\Box[\,\psi_1 \rightarrow \Diamond(\psi_2 \vee \neg\psi_1)\,]$, by the semantics of temporal operators, we conclude that there exist infinitely many positions $j_1, j_2, \ldots$ with $i < j_1 < j_2 < \cdots$ such that for each k, the state s_{j_k} satisfies the property ψ_2.

Let $\mathit{rank}(s_i) = m$. We know that at every step $j \geq i$, the state s_j satisfies the property ψ, the state s_{j+1} does not satisfy the property φ_2, and (s_j, s_{j+1}) is a transition of T. By premise 3, at every step $j \geq i$, $\mathit{rank}(s_{j+1}) \leq \mathit{rank}(s_j)$. Since

for each k the state s_{j_k} satisfies the property ψ_2, rank must strictly decrease at this step: $rank(s_{j_k}) < rank(s_{j_k-1})$.

To summarize, the rank is m at step i, it stays the same or decreases at every step after i, and it strictly decreases at infinitely many steps $j_1, j_2, \cdots$. This is a contradiction. ■

Exercise 5.22: Consider an asynchronous process with the state variables x and y, both of type `nat`, initialized to 0. The process consists of two tasks both of which are always enabled. For the task A_1, the update is specified by

$$\texttt{if } (x > 0) \texttt{ then } x := x - 1 \texttt{ else } x := y$$

and for the task A_2, the update is specified by $y := y + 1$. Weak fairness is assumed for both the tasks. Prove that the process satisfies the recurrence property $\Box \Diamond\, x = 0$ using the proof rule for conditional response properties. ■

Bibliographic Notes

While the origins of *temporal logic* are rooted in philosophy, the use of linear temporal logic for expressing formal requirements of reactive systems was introduced by Pnueli [Pnu77]. Subsequently, the expressiveness and decision procedures for many variants of temporal logics were studied (see [Eme90] for a survey of the theoretical foundations). The specification language PSL is an industrial standard that is supported by commercial tools for hardware design [EF06] (see also integration of specifications in the hardware description language Verilog [BKSY12]).

The concept of *model checking* was introduced in [CE81] and [QS82] in the context of checking branching-time temporal properties of finite-state protocols and has received significant attention in both academia and industry (see the textbooks [CGP00] and [BK08] for an introduction and the 2009 ACM Turing Award lecture for an overview of its impact [CES09]).

Automata over infinite traces were introduced by Büchi in the context of decision procedures for monadic second-order logic [Büc62] (see [Tho90] for a survey of results on such automata). The translation from Ltl to Büchi automata appears in [VW86], and this work led to the *automata-theoretic* approach to model checking. The *nested* depth-first search algorithm of figure 5.11 was introduced in [CVWY92]. The model checker Spin [Hol04] includes state-of-the-art implementations of these techniques.

The symbolic nested fixpoint computation of figure 5.14 was introduced in [BCD+92, McM93] and is supported by the model checker NuSmv [CCGR00].

The proof rules for establishing liveness properties of transitions systems were first studied in [MP81] (see also [Lam94]) and are supported by the verification toolkit TLA+ [Lam02].