

Chapter 4

Model Checking

Recall that requirements can be classified in 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 2, we studied how to specify and verify safety requirements. Now we turn our attention to liveness requirements. Such requirements are specified using a specification formalism called *temporal logic*. The problem of checking whether a model satisfies its specification expressed in temporal logic, is known as *model checking*.

4.1 Requirements Specifications

Let us revisit our example of a system of traffic lights for a railroad from Section 2.1. 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 at the same time. This is a safety requirement that is captured by the property

$$\text{TrainSafety} : \neg (\text{Train}_W.s = \text{bridge} \wedge \text{Train}_E.s = \text{bridge})$$

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` as it would never let any train onto the bridge. We need to augment the specification with liveness requirement that asserts that the controller should allow the trains onto the bridge. For resource allocation problems such as our railroad system, while there is 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 train should be allowed to enter.”

Violation of a safety requirement is demonstrated by a finite execution that leads the system to an erroneous state such as the counterexample of Figure 2.5 that demonstrates that our first attempt at designing the controller was incorrect. Violation of a liveness requirement, on the other hand, is not such a finite execution, rather it consists of a cycle that is reachable from an initial state and if the cycle is executed repeatedly, the demand made by the liveness requirement is unmet. The precise formulation of liveness requirements, hence, considers infinite executions of the system. A natural and increasingly popular formalism for specifying requirements 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.

4.1.1 Linear Temporal Logic

Let V be a set of typed variables and we are writing requirements to constrain the values these variables are allowed to take. For example, the set V can be the set of input and output variables of a synchronous reactive component. An *observation* is a type-consistent assignment of values to V . Recall that we use Boolean expressions to express constraints over variables. Thus, Boolean expressions over V can be used to express requirements on individual observations. For example, suppose 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. While an expression over variables V is evaluated with respect to an observation for V , a temporal logic formula over V is evaluated with respect to an infinite sequence of observations. That is, to interpret a temporal logic formula, we need to consider an infinite sequence $q_1q_2\dots$ where each q_i is an observation. In our example, each observation is an assignment to the Boolean variables x and y , and the temporal logic formulas are evaluated with respect to an infinite sequence $\rho = (x_1, y_1)(x_2, y_2)\dots$. We call such an infinite sequence of observations a *trace* over V .

Boolean expressions are used to express constraints over individual observations, and temporal operators are used to capture requirements, for instance, for all observations in the trace, or for some observation in the trace. A Boolean expression e over V is also a temporal logic formula. We say that a trace ρ satisfies e if the first observation in the trace satisfies e . In our example, the trace $\rho = (x_1, y_1)(x_2, y_2)\dots$ satisfies the formula $(x = y)$ precisely when the initial observation in the trace satisfies the expression, that is, when $x_1 = y_1$.

Let us consider the temporal operator *always*, denoted \square . The formula $\square e$ means that every observation satisfies e . For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\dots$

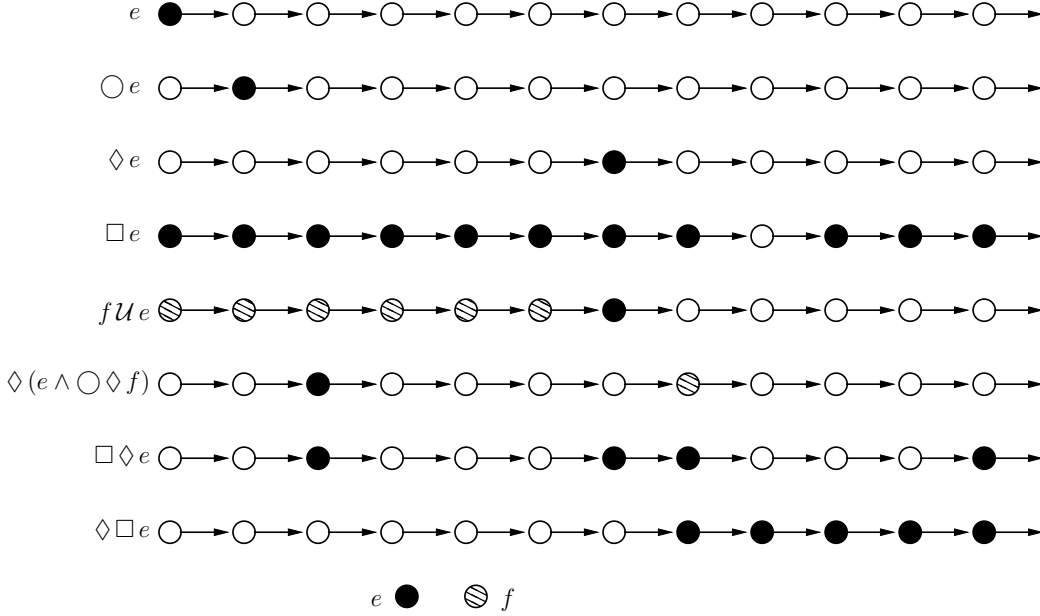


Figure 4.1: Illustrating Temporal operators of LTL

satisfies $\square(x = y)$ precisely when every observation satisfies the expression, that is, when $x_j = y_j$ for every j . Thus, the formula $\square(x = y)$ expresses the requirement that the variable x should always be equal to y . Figure 4.1 illustrates different temporal operators.

The dual of *always* is the operator *eventually*, denoted \diamond . The formula $\diamond e$ means that some observation satisfies e . For example, the trace $\rho = (x_1, y_1)(x_2, y_2) \dots$ satisfies $\diamond(x = y)$ precisely when some observation in the trace satisfies the expression, that is, when $x_j = y_j$ for some j . Thus, the formula $\diamond(x = y)$ expresses the requirement that eventually the variable x must be equal to y .

Temporal logic formulas can be combined using the standard logical operators conjunction (\wedge), disjunction (\vee), implication (\rightarrow), and negation (\neg). The trace ρ satisfies the formula $\square(x = y) \wedge \diamond(x = 0)$ if in every observation of the trace, the value of x is equal to the value of y , and there is an observation in the trace in which x is 0.

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

The final temporal operator we will consider is called the *until* operator, denoted \mathcal{U} , that takes two arguments. The formula $f\mathcal{U}e$ means that the expression f

is satisfied in every observation until we encounter an observation that satisfies e . That is, the trace $q_1q_2\dots$ satisfies the formula $f\mathcal{U}e$ precisely when we can find a position j such that q_j satisfies e and each of the observations from q_1 to q_{j-1} satisfies f . For example, the trace $\rho = (x_1, y_1)(x_2, y_2)\dots$ satisfies $(x = 0)\mathcal{U}(y = 1)$ precisely when there is some j such that $y_j = 1$ and $x_k = 0$ for $k < j$. This expresses the requirement that eventually y should become 1, and until then x should be 0.

So far we have considered formulas in which the arguments to the temporal operators were expressions constraining individual observations. In general, temporal operators can be nested, that is, arguments of temporal operators may themselves be complex temporal logic formulas. For example, consider the formula $\bigcirc\Box(x = y)$. This says that in the next step, always $(x = y)$ holds: the trace $\rho = (x_1, y_1)(x_2, y_2)\dots$ satisfies this formula precisely when $x_j = y_j$ for all $j \geq 2$. That is, ρ satisfies $\bigcirc\Box(x = y)$ at position 1 precisely when it satisfies $\Box(x = y)$ at position 2. To formalize this, we will define what it means for a trace to satisfy a formula at a given position: for an infinite trace ρ , a position $j \geq 1$, and temporal logic formula φ , the notation $(\rho, j) \models \varphi$ stands for “the trace ρ satisfies φ at position j .” The trace $\rho = q_1q_2\dots$ satisfies a Boolean expression (without any temporal operators) at position j , if the observation q_j satisfies e . The trace ρ satisfies the next formula $\bigcirc\varphi$ at position j , where φ may be any temporal logic formula, if ρ satisfies 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 any temporal logic formula, if ρ satisfies φ at position k for some $k \geq j$. That is, “eventually φ ” holds at a position if φ holds at some later position. Similarly, “always φ ” holds at a position if φ holds at every following position.

Now we can define the logic precisely. The definition below defines both the “syntax” of the logic—what are the syntactically correct formulas of the logic, and the “semantics”—how to evaluate the formulas over traces. The definition is inductive; for instance, it describes the rule for evaluating $\Box\varphi$ 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 $\square\varphi$.
- 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 $\varphi_1 \mathcal{U} \varphi_2$.

An observation is a valuation for the variables in V , and a trace is an infinite sequence of observations. Given a trace $\rho = q_1q_2\dots$, a position $i \geq 1$, and an LTL-formula φ , the *satisfaction* relation, $(\rho, i) \models \varphi$, meaning the trace ρ satisfies φ at position i , is defined inductively by the following rules:

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

The trace ρ satisfies φ if $(\rho, 0) \models \varphi$.

Notice that the satisfaction of a formula at a position j of a trace $\rho = q_1q_2\dots$ depends only on the suffix of the trace starting at position j , that is, on the sequence $q_jq_{j+1}\dots$. This is because all the temporal operators refer to the “future” positions. It is worth emphasizing that the current observation is part of the future: satisfaction of “eventually φ ” at a position j demands φ to hold at some position $k \geq j$, so in particular, if φ holds at a position j , then so does “eventually φ ” at position j . Also note that for the until-formula $\varphi_1 \mathcal{U} \varphi_2$ to hold at position j , we demand that φ_2 holds at some position $k \geq j$, and the formula φ_1 holds at all positions following j , including itself, and strictly preceding k (we do not require φ_2 to be satisfied at position k). The eventuality operator is just a special case of the until-operator: $\diamond\varphi$ is same as $1 \mathcal{U} \varphi$, where 1 is the Boolean constant that is always satisfied.

Nesting of temporal operators give interesting and useful formulas. We note

some typical patterns (see Figure 4.1).

Sequencing Nested applications of eventually-operators can 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 the formula $\diamond(\varphi_1 \wedge \bigcirc \diamond \varphi_2)$ holds if there are two position i and j with $i < j$, such that φ_1 holds at position i and φ_2 holds at position j . Note that the use of next operator captures the requirement that the two events happen at distinct positions. For example, the trace $\rho = (x_1, y_1)(x_2, y_2) \cdots$ satisfies $\diamond((x = 1) \wedge \bigcirc \diamond (y = 1))$ if we can find positions $i < j$ with $x_i = 1$ and $y_j = 1$.

Recurrence Formulas Consider the *always-eventually* formula $\square \diamond \varphi$. A trace ρ satisfies $\square \diamond \varphi$ at the beginning if $\diamond \varphi$ is satisfied at every position i . This means for every position i , there exists $j \geq i$ such that ρ satisfies φ at position j . That is, at every position there is a future position where φ holds. 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, φ holds in a recurrent manner. For example, the trace $\rho = (x_1, y_1)(x_2, y_2) \cdots$ satisfies $\square \diamond (x = 0)$ precisely when for infinitely many positions j , $x_j = 0$. This expresses the requirement that x is assigned the value 0 repeatedly. In particular, there is no cycle in which x is non-zero everywhere. Also observe that if a trace satisfies $\square \diamond \varphi$ at a particular position, say at the beginning, then it satisfies $\square \diamond \varphi$ at every position: for all positions i and j , $(\rho, i) \models \square \diamond \varphi$ if and only if $(\rho, j) \models \square \diamond \varphi$.

Persistence Formulas The dual of recurrence requirement expressed by the always-eventually formula is the *eventually-always* formula $\diamond \square \varphi$. It says that there is a position j such that φ holds at every later position; that is, eventually φ holds and continues to hold in a persistent manner. For example, the trace $\rho = (x_1, y_1)(x_2, y_2) \cdots$ satisfies $\diamond \square (x = 0)$ precisely when for some j , for every $k \geq j$, $x_k = 0$. This is same as saying that x is non-zero only at finitely many positions.

4.1.2 LTL as a Specification Logic

We can use LTL formulas to write requirements for both synchronous and asynchronous systems. Let us 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. As the component executes, the trace of inputs and outputs it produces is an *observable* behavior of the component. We call such an (infinite) trace of inputs and outputs, a *trace* of C , and the LTL formula will be evaluated on such traces. For example, our very first component `Delay` (Figure 1.1) has input variable `in` and output variable `out`. LTL-formulas over

these two variables can be used to specify the desired behavior. In particular, consider the specification:

$$\square [in = 0 \rightarrow \bigcirc out = 0] \wedge \square [in = 1 \rightarrow \bigcirc out = 1]$$

which says that at every position 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` can produce, satisfies this specification, so we will say that the component satisfies the specification.

As another example, consider the component `ClockedCopy` (Figure 1.5) with input variables *in* and *clock*, and output variable *out*. Consider the following LTL formula:

$$\square [out = 0 \rightarrow (out = 0) \mathcal{U} clock?] \vee \square [out = 1 \rightarrow (out = 1) \mathcal{U} clock?]$$

It says that if the value of *out* is 0 at a position, then it stays 0 until the event *clock* is present, conjoined with a symmetric formula for *out* = 1. It captures the requirement that *out* should not change in rounds in which the event *clock* is absent. The component `ClockedCopy` does satisfy this requirement.

If there is an execution of the component whose trace violates the formula, then such an execution is a counter-example demonstrating an error. The problem of checking whether a component satisfies a temporal logic specification is known as model checking.

LTL MODEL CHECKING

Given a synchronous reactive component C with input variables I and output variables O , an LTL specification for C is an LTL formula φ over the set $I \cup O$ of variables. An *infinite execution* of C consists of an infinite trace of the form $s_0 \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \cdots$ such that s_0 is an initial state of C ; for each $j > 0$, $s_{j-1} \xrightarrow{i_j/o_j} s_j$ is a reaction C . The infinite trace $(i_1, o_1)(i_2, o_2) \cdots$ of inputs and outputs is called a *trace* of C . The component satisfies the specification φ if every trace of C satisfies φ . An infinite execution is called a counter-example for the specification φ if the corresponding trace does not satisfy φ .

Requirements for Leader Election

Let us recall the leader election problem from Section 1.4.2. The interesting output variable for each node is its *status* that ranges over `{unknown, leader, follower}`. While the nodes use the variables *in* and *out* for exchanging messages with one another, the requirements of the problem should 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

$$\diamond [\text{SyncLENode}_n.\text{status} \neq \text{unknown}]$$

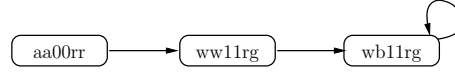


Figure 4.2: Cyclic counter-example for liveness violation

The formula states that for a given node n , eventually the value of $\text{SyncLENode}_n.\text{status}$ (that is, the status of the instance of SyncLENode corresponding to this node) should be different from **unknown**. The safety requirement that two nodes should not consider themselves 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:

$$\Box (\text{SyncLENode}_m.\text{status} \neq \text{leader}) \vee \Box (\text{SyncLENode}_n.\text{status} \neq \text{leader})$$

Requirements for Railroad Controller

Let us revisit the railroad controller system of Section 2.1.2. Let us consider the observable variables of the system to be signal_W and signal_E capturing the traffic lights, and the variables $\text{Train}_W.s$ and $\text{Train}_E.s$ capturing the train states. While the latter are not modeled as outputs, it is acceptable to write requirements that refer to the state of the models capturing the environment since this is part of the specification of the design problem. The same requirements could be captured using the output signals signal_W and signal_E , but the formulas will be more complex.

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

$$\Box \neg (\text{Train}_W.s = \text{bridge} \wedge \text{Train}_E.s = \text{bridge})$$

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

$$\Box \diamond (\text{Train}_W.s = \text{bridge})$$

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 **away** state 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 arrives then eventually the west traffic signal should turn green:

$$\Box [(\text{Train}_W.s = \text{wait}) \rightarrow \diamond (\text{signal}_W = \text{green})]$$

The LTL formula says that at every step, if the condition ($\text{Train}_W.s = \text{wait}$) holds, then at some later position the condition ($\text{signal}_W = \text{green}$) must hold. This is a typical pattern for LTL formulas: “Always if request then Eventually response”. We want to check if every trace of the system `RailRoadSystem2` satisfies this specification. It turns out that this is not the case. The counter-example, which consists of an initial sequence of steps followed by a cyclic execution, is illustrated in Figure 4.2. As in Figure 2.9, each state is denoted by listing the values of $\text{Train}_W.s$, $\text{Train}_E.s$, near_W , near_E , west , and east , in that order, and a , w , b , g , and r , are abbreviations for **away**, **wait**, **bridge**, **green**, **red**, respectively. The cycle in the counter-example 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, which is possible according to the model of the train, then there is not much that the controller can do to let the west train in. Consequently, we need to modify our specification of requirements.

One 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.

$$\Box [(\text{Train}_W.s = \text{wait}) \rightarrow \Diamond ((\text{signal}_W = \text{green}) \vee (\text{Train}_E.s = \text{bridge}))]$$

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, 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 2.6 is deadlock-free, 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, 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:

$$\Box \Diamond (\text{Train}_E.s \neq \text{bridge}) \rightarrow \Box [(\text{Train}_W.s = \text{wait}) \rightarrow \Diamond (\text{signal}_W = \text{green})]$$

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 2.6 is in fact starvation-free, and meets the specification φ_{sf} . In particular, the counter-example of Figure 4.2 is ruled out. Since the pre-condition $\Box \Diamond (\text{Train}_E.s \neq \text{bridge})$ is violated by this trace, it satisfies φ_{sf} .

4.2 Persistence Verification

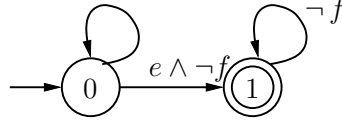
In Chapter 2, we saw that the canonical safety verification problem is the invariant verification problem: given a symbolic transition system T and a property φ over its state variables, we want to check if all reachable states of the system satisfy φ . The dual of the invariant verification is the reachability problem: to check whether φ is an invariant of T , we check whether a state violating φ is reachable, and if so, the corresponding execution is a counter-example to the invariant verification question. The core computational problem for verification of liveness requirements turns out to be the *persistence verification* problem: given a symbolic transition system T and a property φ over its state variables, we want to check if every infinite execution of the system eventually stays only within the states satisfying φ (that is, whether the persistence formula $\diamond\Box\varphi$ holds for every infinite execution). We will show that the LTL model checking problem, namely, checking whether every trace of a given system satisfies an LTL specification, can be reduced to the persistence verification problem for the composition of the system and a monitor derived from the LTL specification.

The dual of the persistence verification question is to check if there exists *some* infinite execution of the system which repeatedly encounters states violating φ . This dual problem is called the *recurrent reachability* question: given a symbolic transition system T and a property φ over its state variables, we want to check if there exists an infinite execution of the system in which states satisfying the property $\neg\varphi$ are encountered repeatedly (that is, whether there exists an infinite execution satisfying the recurrence formula $\Box\diamond\varphi$). If such an erroneous execution is found, it is the counter-example to the original persistence verification problem.

Recall the transition system $\text{GCD}_{m,n}$ from Section 2.1 capturing the program for computing the greatest common divisor of two numbers m and n . To check whether all executions of the program terminate, we check if the property 0 is persistent for this system. Since no state satisfies 0, asking whether all infinite executions stay within 0 is equivalent to absence of infinite executions. A counter-example will be an infinite execution (for any such execution, the property 1 is recurrent). To show that the leader election algorithm eventually makes a decision for every node, we check for every node n , if the property $\text{SyncLENode}_n.\text{status} \neq \text{unknown}$ is persistent.

PERSISTENCE VERIFICATION PROBLEM

An *infinite execution* of a transition system T consists of an infinite sequence of the form s_0, s_1, \dots such that s_0 is an initial state of T and $s_{j-1} \rightarrow s_j$, for $j > 0$, is a transition of T . A property φ over state variables of a transition system T is said to be *persistent* if for every infinite execution s_0, s_1, \dots of T , there exists a position j such that for all positions $k \geq j$, s_k satisfies φ . The persistence verification problem is to check, given a symbolic transition system T and a property φ over its state variables, whether φ is persistent.

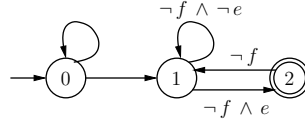
Figure 4.3: Monitor for detecting violation of $\Box(e \rightarrow \Diamond f)$

When the answer to the persistence verification problem with input transition system T and property φ is negative, we want to produce an infinite execution in which the property $\neg\varphi$ is recurrent. Typically such an execution will be demonstrated by a state s such that (1) s is reachable from some initial state, (2) s is reachable from itself, and (3) s violates φ . That is, as evidence, we will produce a cycle that is reachable from some initial state and contains a state violating φ .

4.2.1 LTL Model Checking as Persistence Verification

Many of the LTL specifications we have considered so far are of the form $\varphi : \Box(e \rightarrow \Diamond f)$, where e and f are expressions over the observable variables of the component C . To check whether every trace of C satisfies φ , we first negate the specification and check if there is some execution of C that satisfies the negated specification. The negated specification is $\neg\Box(e \rightarrow \Diamond f)$, and this is equivalent to $\Diamond(e \wedge \Box\neg f)$. Thus, we want to search for an infinite execution in which e holds at some position, and then onwards f never holds. Consider the nondeterministic monitor M shown in Figure 4.3. The input variables of M are the observable variables of the system C . It has a state variable s that ranges over $\{0, 1\}$. Its initial state is 0. When the monitor state is 0, in every round, if the observable variables satisfy the condition e , but not f , the monitor can either stay in state 0, or switch to state 1; otherwise, it can only continue to stay in state 0. When the monitor state is 1, then it can stay in state 1 only if the observed variables satisfy the condition $\neg f$, and otherwise, no transition is enabled. In the composed system $C\|M$, there is an infinite execution in which the monitor state is repeatedly 1 only if the component C can produce a trace that satisfies $\neg\varphi$. Thus, the model checking problem of verifying that every trace of C satisfies φ has been reduced to checking whether the property $s = 0$ over the monitor state is persistent for the composed system $C\|M$ (equivalently, is there an infinite execution in which $s = 1$ is recurrent). The search algorithms try to find a reachable cycle in which the state 1 repeats, and hence, the state 1 is marked by a double circle.

As another example, consider the LTL specification $\varphi : \Box\Diamond e \rightarrow \Box\Diamond f$, where e and f are expressions 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 C satisfies φ , we first negate the specification and check if there is some execution of C that

Figure 4.4: Monitor for detecting violation of $\Box\Diamond e \rightarrow \Box\Diamond f$

satisfies the negated specification. The negated specification is equivalent to $\Box\Diamond e \wedge \Diamond\Box\neg f$, stating that the property e recurrent and the property $\neg f$ is persistent. Thus, we want to find an infinite execution in which after a certain position, all states satisfy $\neg f$, and e holds repeatedly. This is captured by the nondeterministic monitor M shown in Figure 4.4. Its state can take 3 values. Initially the state is 0. The monitor can loop in this initial state for an arbitrary number of steps, and then nondeterministically switches to state 1. Every time the condition e is satisfied, it transitions to state 2, and switches back to state 1 in the next step. In both states 1 and 2, execution can continue only the condition $\neg f$ holds. An infinite execution can visit the state 2 repeatedly only if the property e recurrent and the property $\neg f$ is persistent. Thus, the model checking problem can be reformulated as checking whether $s \neq 2$ is persistent for the composed system $C\|M$.

The construction of the monitors for detecting violations of LTL specifications can be automated. We skip the details, and only give the statement of the result.

Theorem 4.1 [From LTL Model Checking to Persistence Verification] *Given an LTL formula φ over the observable variables V , there is an algorithm to construct a nondeterministic monitor M_φ with input variables V , finitely many states Q , and a subset F of Q such that a system C satisfies the specification φ precisely when for the composed system $C\|M_\varphi$, the property “state of M belongs to F ” is persistent.*

4.3 Nested Depth-first Search

Given a transition system T and a property φ , to check whether φ is a persistent property of all infinite executions of T , we search for a state that is reachable, that violates φ , and is contained in a cycle. The core computational problem here is detecting cycles. As discussed in Section 2.3, we assume that the transition system is represented by functions such as *NextInitState* and *NextSuccState*, and we want to explore states and transitions out of these states only on demand. Furthermore, the algorithm should terminate returning a counter-example as soon as it finds one, and thus, it ideally should not first explore all the reachable states and then proceed to finding cycles. As a result, the classical algorithms for detecting cycles that rely on computation of

strongly-connected-components in a graph are not best suited for our application. Instead, we will present a cycle-detection algorithm that employs nested depth-first-search.

The algorithm explores the reachable states of the transition system in a manner very similar to the depth-first-search algorithm of Figure 2.10 using the stack *Pending* and the set *Reach* to store states already visited. The key difference is the following: in invariant verification, when a state violating the desired property is encountered, the search terminates; for persistence verification, when a state s violating the desired property is encountered, the algorithm initiates another search for a cycle containing s . To implement this, suppose every time a state violating φ is encountered, completely new search to check whether s is reachable from itself is initiated, and this search uses its own set of visited states. While this would be obviously correct, it is too expensive (of time complexity that is quadratic in the number of states), and can be significantly improved. The algorithm is shown in Figure 4.5.

The algorithms 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*, while 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 input state s . Once the primary search originating at s terminates, if the state s violates the desired persistent property φ , it is a potential candidate for the cyclic counter-example. Then a secondary search is initiated by calling *NDFS* with input s . The objective of this secondary search is to find a cycle starting at the state s . When *NDFS*(s) is invoked, the stack *Pending* contains an execution starting from an initial state leading to state s . Thus, if the secondary search visits 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 0, the graph contains a reachable cycle containing a state that violates 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 *NDFS* is called with input state s , s is added to this set, and *NDFS* is invoked with input state s only if the state s is not in *NReach*. Thus, the secondary search explores a reachable state at most once, and the total running time of the algorithm is linear in the number of reachable states. The tricky part is the correctness argument. Suppose s and t are two states that are encountered by the primary search, and suppose both violate φ . Suppose secondary search is invoked with input s first, and it explores all states reachable from s , adding them to the set *NReach*, but without finding a cycle. Later, when secondary search is invoked with input state t , it will just skip over states that were added to *NReach*. What guarantees that this does not cause the algorithm to miss detection of a cycle containing state t ?

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 $DFS(s)$ terminates before $DFS(t)$ then $d_s < d_t$. Suppose the transition system T contains a reachable cycle containing some state violating φ . Let s_0, \dots, s_k be the ordering of such states 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 states s_j with $j < i$. Verify that s_i does not belong to Q (otherwise, there is a cycle containing some s_j for $j < i$). In fact, the cycle that contains s_i is disjoint from Q . When the primary search from state s_i is over, the set $NReach$ containing the states visited by the secondary search so far equals Q . Consequently, $NDFS$ will be invoked with input s_i , and will find a cycle containing s_i .

Note that, just like the depth-first-search algorithm of Figure 2.10, the nested depth-first-search algorithm may detect a cycle before exploring all the reachable states. 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 4.2 [Nested Depth-first-search for Persistence Verification] *Given a countably-branching transition system T and a property φ , the nested-depth-first-search algorithm of Figure 4.5 has the following properties: (1) if the algorithm terminates, then the returned value correctly indicates whether or not the property φ is persistent for T , and (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.*

4.4 Symbolic Persistence Verification

Recall the symbolic breadth-first search algorithm for invariant verification by iterative image computation discussed in Section 2.4. We will develop a nested search algorithm to check whether the transition system has an infinite computation corresponding to a counter-example for persistence verification. 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 S , and the transitions are represented by a region $Trans$ over $S \cup S'$. The negation of the desired persistence property is also represented by a region. The data type **reg** for regions supports operations such as **Conj**, **Diff**, and **IsSubset**.

The core step of symbolic verification algorithms is image computation. Given a region A over state variables, 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:

$$\mathbf{Post}(A, Trans) = \mathbf{Rename}(\mathbf{Exists}(\mathbf{Conj}(A, Trans), S), S', S)$$

The dual operator is the pre-image computation: given a region A over 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 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 region $Trans$ over $S \cup S'$ to obtain all transitions with their targets 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 operation is defined as:

$$\text{Pre}(A, Trans) = \text{Exists}(\text{Conj}(\text{Rename}(A, S, S'), Trans), S')$$

The symbolic algorithm for persistence verification shown in Figure 4.6 uses both image 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 . The second phase attempts to find an infinite execution with recurring violation of the property φ . The set of states whose repeated occurrence indicates a counter-example is captured by the region $Recur$. This region initially contains all states that are reachable and violate φ , and can be computed by intersecting the region $Reach$ computed at the end of the first phase and the region representing the negated property. Let us call this set $Recur_0$. For each of the states s in this set, we want to compute if there exists an execution consisting of one or more transitions starting in 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 reachable states: the region $Reach$, initialized to empty set, contains the states already examined; the region New , initialized to 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, $Reach$ is updated by adding the unexplored states in New ; the set of states to be newly explored is obtained by applying a pre-image computation to 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 $Recur$.

Let $Recur_1, Recur_2, \dots$ be the successive values assigned to $Recur$ at the end of the outer while loop. Each such set $Recur_i$ is a subset of $Recur_{i-1}$, and contains those states in $Recur_{i-1}$ from which some state in $Recur_{i-1}$ can be reached by an execution with one or more transitions. In particular, each such set $Recur_i$ contains states s such that s is reachable from some initial state, s violates φ , and there is an execution starting from s that encounter states violating φ at least i times.

Suppose the property φ is not persistent for the transition system T . Then, there is a state s that is reachable from some initial state, s violates φ , and there is an infinite execution starting from s that encounters states violating φ infinitely many times. Such a state s will never be removed from $Recur$, that is, it will belong to every set $Recur_i$. As a result, if the algorithm finds $Recur$ to be empty at any stage, no such state s exists, and the algorithm can terminate claiming that φ is persistent.

Conversely, suppose for the current value $Recur_i$, which is non-empty, 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 $Reach$ should be a superset of $Recur_i$. As $Reach$ is computed iteratively adding more and more states that can reach $Recur_i$, when the algorithm finds that it is a superset of $Recur$, it terminates saying that the persistence property is violated. We argue that in this case, indeed there is an infinite execution with recurrent $\neg\varphi$. Let s_0 be any state in $Recur_i$. Since $Recur_i$ is a subset of $Recur_0$, s_0 is reachable from the initial state. Hence, it suffices to demonstrate that there exists an infinite execution starting at s_0 with recurrent $\neg\varphi$. From s_0 , there is an execution with one or more transitions leading to some state, say s_1 , in $Recur_i$. From 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 executions guarantees existence of the desired infinite execution that contains states s_0, s_1, \dots all of which violate φ .

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 violate φ . Then, $Recur_0$ contains k states, and the number of iterations of the outer loop is at most k (since states are only removed from $Recur$, and 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 reachable from states in $Recur$ 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.

Theorem 4.3 [Symbolic Nested Search for Persistence Verification] *Given a transition system T and a property φ , the symbolic nested breadth-first-search algorithm of Figure 4.6 has the following properties: (1) If the algorithm terminates, then the returned value correctly indicates whether or not the property φ is persistent for T , (2) If the transition system has n reachable states of which k violate φ , then the algorithm terminates after at most $O(nk)$ operations on regions.*

Input: A countably-branching transition system T and property φ ;
 Output: If φ is a persistent 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) = 0 then return 0;
    s = NextInitState(s, T);
  };
return(1).

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) = 0 then return 0;
      t = NextSuccState(s, t, T);
    };
  if Satisfies(s, φ) = 0 then
    if Contains(NReach, s) = 0 then
      if NDFS(s) = 0 then return 0;
  Pop(Pending);
  return 1.

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

```

Figure 4.5: Nested depth-first search algorithm for persistence verification

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: Is the property φ persistent for the transition system T ?

```

reg  $Reach = \text{Empty}$ ;
reg  $New = Init$ ;
while  $\text{IsEmpty}(New) = 0$  do {
   $Reach = \text{Disj}(Reach, New)$ ;
   $New = \text{Diff}(\text{Post}(New, Trans), Reach)$ ;
};
reg  $Recur = \text{Conj}(Reach, \neg\varphi)$ ;
while  $\text{IsEmpty}(Recur) = 0$  do {
   $Reach = \text{Empty}$ ;
   $New = \text{Pre}(Recur, Trans)$ ;
  while  $\text{IsEmpty}(New) = 0$  do {
     $Reach = \text{Disj}(Reach, New)$ ;
    if  $\text{IsSubset}(Recur, Reach) = 1$  then return 0;
     $New = \text{Diff}(\text{Pre}(New, Trans), Reach)$ ;
  };
   $Recur = \text{Conj}(Recur, Reach)$ ;
};
return 1.

```

Figure 4.6: Symbolic nested search algorithm for persistence verification