Chapter 6. Automata and Logic

> "Obstacles are those frightful things you see when you take your eyes off your goal."
>
> —(Henry Ford, 1863–1947)

The model checking method that we will describe in the next few chapters is based on a variation of the classic theory of finite automata. This variation is known as the theory of ω–automata. The main difference with standard finite automata theory is that the acceptance conditions for ω–automata cover not just finite but also infinite executions.

Logical correctness properties are formalized in this theory as ω–regular properties. We will see shortly that ω–automata have just the right type of expressive power to model both process behavior in distributed systems and a broad range of correctness properties that we may be interested in proving about such systems.

Automata

To develop the theory, we begin with a few basic definitions.

Definition 6.1 (FSA)

A finite state automaton is a tuple (S, s_0, L, T, F), where

We will refer to state set S of finite state automaton A with a dot notation: A. S. Similarly, the initial state of A is referred to as A. s_0, etc.

In the simplest case, an automaton is deterministic, with the successor state of each transition uniquely defined by the source state and the transition label. Determinism is defined more formally as follows.

Definition 6.2 (Determinism)

A finite state automaton (S, s_0, L, T, F) is deterministic if, and only if,

$$\forall s \forall l, ((s, l, s') \in T \wedge (s, l, s'') \in T) \rightarrow s' \equiv s''.$$

Many of the automata we will use do not have this property, that is, they will be used to specify non−deterministic behaviors. As we shall see, there is nothing in the theory that would make the handling of non−deterministic automata particularly troublesome.

Definition 6.3 (Runs)

A run of a finite state automaton (S, s_0, L, T, F) is an ordered, possibly infinite, set of transitions (a sequence)

$$\{(s_0, l_0, s_1), (s_1, l_1, s_2), (s_2, l_2, s_3), \ldots\}$$

such that

$$\forall i, (i \geq 0) \rightarrow (s_i, l_i, s_{i+1}) \in T.$$

Occasionally we will want to talk about specific aspects of a given run, such as the sequence of states that is traversed, or the sequence of transition labels that it defines. Note that for non−deterministic automata the sequence of states traversed cannot necessarily be derived from the sequence of transition labels, and vice versa.

Definition 6.4 (Standard acceptance)

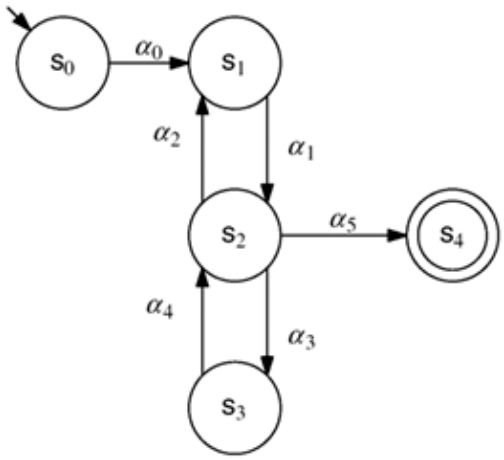
> An accepting run of finite state automaton (S, s_0, L, T, F) is a finite run in which the final transition (s_{n-1}, l_{n-1}, s_n) has the property that $s_n \in F$.

The run is considered accepted if and only if it terminates in a final state of the automaton.

Figure 6.1 shows a simple finite state automaton with five states. It is defined as follows:

$$
\begin{aligned}
S &= \{ s_0, s_1, s_2, s_3, s_4 \}, \\
L &= \{ \alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \}, \\
F &= \{ s_4 \}, \text{ and} \\
T &= \{ (s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), \\
&\quad (s_2, \alpha_2, s_1), (s_2, \alpha_3, s_3), \\
&\quad (s_3, \alpha_4, s_2), (s_2, \alpha_5, s_4) \}.
\end{aligned}
$$

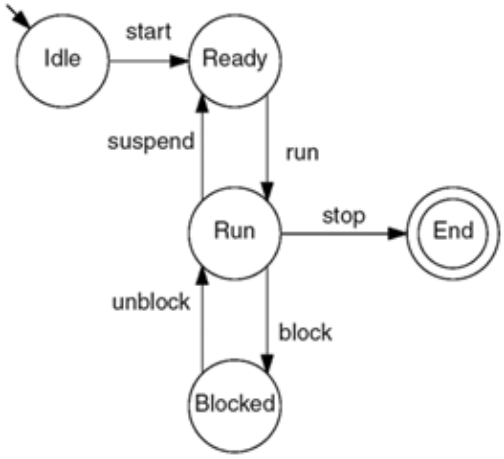
Figure 6.1. A Simple Finite State Automaton



The initial state s_0 is traditionally marked with a short arrow, and the elements of set F are marked with a double circle in graphical representations, as we have done in Figure 6.1.

The labels on the transitions in a finite state automaton do not themselves have any inherent semantics. For the purpose of the definitions, they are just symbols or uninterpreted text strings. We can freely choose to interpret some of these labels as inputs or outputs or, where appropriate, as conditions or actions. The sample automaton from Figure 6.1, for instance, could be interpreted as modeling the life of a user process in a time−sharing system, controlled by a process scheduler, as illustrated in Figure 6.2. State s_0 then represents the "Initial" state where the process is being instantiated, state s_1 is the "Ready" state, s_2 is the "Run" state, s_3 is the "Blocked" state, for example, where the process may be waiting for a system call to complete, and s_4 is the "End" state, reached if and when the process terminates.

Figure 6.2. A Possible Interpretation of the Automaton in Figure 6.1



One possible accepting run of this system is represented by the sequence

$(s_0,\alpha_0 s_1)$, (s_1,α_1,s_2), (s_2,α_5,s_4),

which, under our chosen interpretation, corresponds to the sequence of scheduler actions: start, run, stop.

Omega Acceptance

With the definition of a finite state automaton given here, we can model terminating executions, but we still cannot decide on acceptance or non−acceptance of ongoing, potentially infinite, executions. Looking at Figure 6.2, for instance, if we were to model the underlying scheduler, rather than the processes being scheduled, the termination of the scheduler itself would not necessarily be a desirable result. The same is true for many other interesting systems, such as the control software for a nuclear power plant, a telephone switch, an ATM machine, or a traffic light.

An infinite run is often called an ω−run (pronounced: "omega run"). Acceptance properties for ω−runs can be defined in a number of different ways. The one we will adopt here was introduced by J.R. Büchi [1960].

If σ is an infinite run, let the symbol σ^{ω} represent the set of states that appear infinitely often within σ's set of transitions, and σ^{+} the set of states that−appear only finitely many times. The notion of Büchi acceptance is defined as follows.

Definition 6.5 (Büchi acceptance)

> An accepting ω−run of finite state automaton (S, s_0, L, T, F) is any infinite run σ such that $\exists s_f,\ s_f \in F \wedge s_f \in \sigma^{\omega}$.

That is, an infinite run is accepted if and only if some state in F is visited infinitely often in the run. Without further precautions then, in the automaton from Figure 6.1 we could only see an accepting run under the definition of Büchi acceptance if at least one of the states s_1, s_2, or s_3 were members of final set F, since only these states can be visited infinitely often in a run.

With these definitions it is also easy to formalize the notion of non−progress that we discussed before. Let $P \subseteq S$ be the set of progress states. An ω−run σ then corresponds to a non−progress cycle if: $\forall s_f,\ s_f \in \sigma^{\omega} \rightarrow s_f \notin P$.

The Stutter Extension Rule

The given formalization of acceptance applies only to infinite runs. It would clearly be convenient if we could somehow find a way to extend it so that the classic notion of acceptance for finite runs (cf. Definition 6.4) would be included as a special case. This can be done with the adoption of a stuttering rule. To apply the rule, we must extend our label sets with a fixed predefined null−label ε, representing a no−op operation that is always executable and has no effect (much like PROMELA's `skip` statement). The stutter extension of a finite run can now be defined as follows.

Definition 6.6 (Stutter Extension)

The stutter extension of finite run σ with final state s_n is the ω−run $\sigma,(s_n, \varepsilon, s_n)^\omega$.

The final state of the run, then, is thought to persist forever by infinitely repeating the null action ε. It follows that such a run would satisfy the rules for Büchi acceptance if, and only if, the original final state s_n is in the set of accepting states F, which means that it indeed generalizes the classical definition of finite acceptance.

A couple of abbreviations are so frequently used that it is good to summarize them here. The set of runs that is accepted by an automaton is often referred to as the language of the automaton. An automaton with acceptance conditions that are defined over infinite runs is often called an ω−automaton.

Accepting ω−runs of a finite state automaton can always be written in the form of an expression, using a dot to represent concatenation and the superfix ω to represent infinite repetition:

$$\bigcup_{i=1}^{N} U_i \cdot V_i^\omega$$

with U_i and V_i regular expressions over transitions in the automaton. That is, each such run consists of a finite prefix U, corresponding to an initial part of the run that is executed just once, and a finite suffix V, corresponding to a part of the run that is repeated ad infinitum. These expressions are called ω−regular expressions, and the class of properties that they express are called ω−regular properties. As a final bit of terminology, it is common to refer to automata with Büchi acceptance conditions simply as Büchi Automata.

Finite States, Infinite Runs

It is clear that also an automaton with only finitely many states can have runs that are infinitely long, as already illustrated by Figure 6.1. With some reflection, it will also be clear that a finite automaton can have infinitely many distinct infinite runs.

To see this, imagine a simple automaton with eleven states, ten states named s_0 to s_9, and one final (accepting) state s_{10}. Define a transition relation for this automaton that connects every state to every other state, and also include a transition from each state back to itself (a self–loop). Label the transition from state s_i to s_j with the PROMELA print statement `printf("%d\n", i)`. Use this labeling rule for all transitions except the self–loop on the one final state s_{10}, which is labeled `skip`. That is, if the transition from s_i to s_j is taken, the index number of the source transition will be printed, with as the single exception the transition from s_{10} back to s_{10}, which produces no output.

Every accepting run of this automaton will cause a number to be printed. This automaton has precisely one accepting run for every imaginable non–negative integer number, and there are infinitely many distinct numbers of this type.

Other Types of Acceptance

There are of course many other ways to formalize the acceptance conditions of ω–automata. Most of these methods are named after the authors that first proposed them.

- Define F to be a set of subsets from state set S, that is, $F \subseteq 2^S$. We can require that the set of all states that are visited infinitely often in run σ equals one of the subsets in F:

  $$\exists f,\ f \in F\ \wedge\ \sigma^{\omega} \equiv f\ .$$

  This notion of acceptance is called Muller acceptance, and the corresponding automata are called Muller Automata.
- We can also define a finite set of n pairs, where for each pair (L_i, U_i) we have $L_i \subseteq S$ and $U_i \subseteq S$. We can now require that there is at least one pair i in the set for which none of the states in L_i appear infinitely often in σ, but at least one state in U_i does:

  $$\exists i, (1 \le i \le n),\ \forall s,\ (s \in L_i \rightarrow s \notin \sigma^{\omega})\ \wedge\ \exists t, (t \in U_i \wedge t \in \sigma^{\omega}).$$

  This notion of acceptance is called Rabin acceptance, and the corresponding automata are called Rabin Automata
- Using the same definition of pairs of sets states, we can also define the opposite condition that for all pairs in the set either none of the states in U_i appear infinitely often in σ, or at least one state in L_i does:

  $$\forall i, (1 \le i \le n),\ \exists s, (s \in L_i \wedge s \in \sigma^{\omega})\ \vee\ \forall t, (t \in U_i \rightarrow t \notin \sigma^{\omega}).$$

  This notion of acceptance is called Streett acceptance, and the corresponding automata are called Streett Automata.

All these types of acceptance are equally expressive and define the same class of ω–regular properties as Büchi Automata.

Many interesting properties of Büchi automata have been shown to be decidable. Most importantly, this applies to checks for language emptiness (i.e., deciding whether the set of accepting runs of a given Büchi automaton is empty), and language intersection (i.e., generating a single Büchi automaton that accepts precisely those ω–runs that are accepted by all members of a given set of Büchi automata). We shall see in Chapter 8 that the verification problem for SPIN models is equivalent to an emptiness test for an intersection product of Büchi automata.

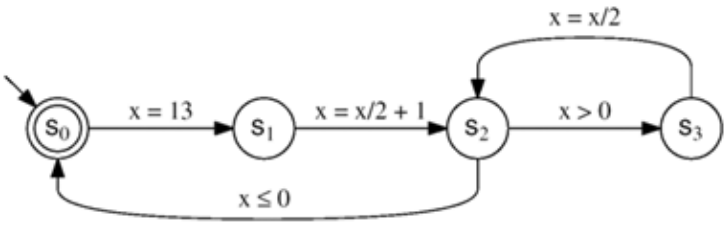
Logic

Automata models offer a good formalism for the analysis of distributed system models. As noted in earlier chapters, though, a complete verification model contains not just the specification of system behavior but also a formalization of the correctness requirements that apply to the system. We will now show how higher–level requirements can be expressed in a special type of logic that has a direct connection to the formalism of Büchi Automata.

A run of an automaton captures the notion of system execution. Through the definition of acceptance conditions we can already distinguish the runs that satisfy a given set of requirements from those that violate it. It can be a daunting task, though, to express correctness requirements at this relatively low level of abstraction. We need a more convenient method.

Consider the automaton from Figure 6.3 with initial and final state s_0. The formulation of a correctness property for this automaton requires the ability to interpret its runs, minimally to distinguish the good runs from the bad ones. To do so, we can define a semantics on the labels that are used. We will use the semantics interpretation of PROMELA. An integer variable named x is assigned values, and is tested in conditionals.

Figure 6.3. Model of a Simple Computation



The sequence of states traversed during a run of this (deterministic) automaton is:

$s_0, s_1, s_2, s_3, s_2, s_3, s_2, s_3, s_2, s_0, ...$

Of course, not every accepting run of the automaton will automatically be consistent with our chosen semantics interpretation of the labels. Given an initial value for x, we can write down the new value for x that is consistent with the PROMELA assignment statements from the labels. A run is only consistent with the semantics interpretation if all condition labels that appear in the run, such as $x > 0$ and $x \leq 0$, evaluate to true.

Given the initial value zero of x, we can annotate each state in the run above with the corresponding value for x, as follows:

$$(s_0, 0), (s_1, 13), (s_2, 7), (s_3, 7), (s_2, 3), (s_3, 3), (s_2, 1), (s_3, 1), (s_2, 0), (s_0, 0), ...$$

Jointly, the state number s_i and the value of variable x define an extended system state. We can derive a pure (non–extended) finite state automaton from the one that is specified in Figure 6.3 by expanding the set of states. State s_2, for instance, would generate four copies in such an unfolding. For values 7, 3, 1, and 0 of x, and state s_3 would generate three copies, one for each of the values 7, 3, and 1. The resulting expanded finite state automaton has nine states, all of which appear in the annotated state sequence.

We can formulate properties of runs of the expanded automaton. The most interesting properties would deal with the achievable and non−achievable values of x during a run. Consider, for instance, the properties

p: "the value of x is odd"

q: "the value of x is 13"

We can deduce a truth value for p and for q at each extended system state. These types of properties are formally called state formulae, to distinguish them from the temporal formulae we will discuss next. The sequence of truth values for p and q then are:

p: false, true, true, true, true, true, true, true, false, false, ...

q: false, true, false, false, false, false, false, false, false, false, ...

To pursue this idea one step further still, we can make statements about possible and impossible sequences of boolean values for p and q throughout a run, as in:

- p is invariantly true,
- p eventually becomes invariantly false,
- p always eventually becomes false at least once more,
- q always implies ¬p,
- p always implies eventually q.

Note that the latter types of statements can only be fully evaluated for complete runs of the system, not just for individual system states in isolation. It is clear that the first two properties above do not hold for the (single) accepting run of our extended finite state automaton. The third and the fifth properties both hold, but the fourth does not.

There are two questions that remain to be answered: can we formalize the properties of the type we have just discussed in terms of a simple logic, and can we express the properties in terms of Büchi acceptance conditions? The next two sections provide the answers to these two questions.

Temporal Logic

The branch of logic that allows one to reason about both causal and temporal relations of properties is called temporal logic. Temporal logic was first studied in the late sixties and early seventies, but primarily as a tool in philosophical arguments that involved the passage of time. A first paper proposing the application of this type of logic for the analysis of distributed system was authored by Amir Pnueli in 1977. It took more than a decade, though, for the fundamental importance of these ideas to be more generally accepted.

Temporal logic allows us to formalize the properties of a run unambiguously and concisely with the help of a small number of special temporal operators. Most relevant to the verification of asynchronous process systems is a specific branch of temporal logic that is known as linear temporal logic, commonly abbreviated as LTL. The semantics of LTL is defined over infinite runs. With the help of the stutter extension rule, however, it applies equally to finite runs, as we shall see in more detail shortly.

A well−formed temporal formula is built from state formulae and temporal operators, using the following two basic rules:

Definition 6.7 (Well−Formed Temporal Formulae)

- All state formulae, including `true` and `false`, are well−formed temporal formulae.
- If α is a unary temporal operator, β is a binary temporal operator, and p and q are well−formed temporal formulae, than so are α p, p β q, (p), and !p (¬ p).

The first temporal operator we will discuss is the binary operator until, which we will represent by the symbol U. The truth of a formula such as pUq can be evaluated for any given ω−run σ. The symbols p and q can be replaced with arbitrary state formulae or with temporal sub−formulae. If a temporal formula f holds for ω−run σ, we write:

σ |= f.

In the definitions that follow, we use the notational convention that σ_i represents the i−th element of the run σ, and σ[i] represents the suffix of σ that starts at the i−th element. Trivially σ ≡ σ[1] ≡ σ_1σ[2].

There are two variations of the until operator that are distinguished by the adjective weak and strong. The definition of the weak until operator is:

Definition 6.8 (Weak Until)

$$\sigma[i] \models (p \mathbin{U} q) \;\Leftrightarrow\; \sigma_i \models q \;\vee\; (\sigma_i \models p \wedge \sigma[i+1] \models (p \mathbin{U} q)).$$

Notice that this definition does not require that the sub−formula q ever become true. The second variant of the operator, called strong until and written U, adds that requirement.

Definition 6.9 (Strong Until)

$$\sigma[i] \models (p \; U \; q) \quad \Leftrightarrow \quad \sigma[i] \models (p \cup q) \; \wedge \exists j, j \geq i, \sigma_j \models q.$$

There are two special cases of these two definitions that prove to be especially useful in practice. The first is a formula of the type p $\cup$ false. Note that the truth of this formula only depends on the value of sub−formula p. We introduce a special operator to capture this:

Definition 6.10 (Always)

$$\sigma \models \Box p \quad \Leftrightarrow \quad \sigma \models (p \cup \mathit{false}).$$

The formula $\Box$ p captures the notion that the property p remains invariantly true throughout a run. The operator $\Box$ is therefore pronounced always or box.

The second special case is a formula of the type true $\cup$ q, which again reduces the number of operands from two to one. The case is important enough to warrant the introduction of another shorthand.

Definition 6.11 (Eventually)

$$\sigma \models \Diamond q \quad \Leftrightarrow \quad \sigma \models (\mathit{true} \; U \; q).$$

The formula $\Diamond$ p captures the notion that the property p is guaranteed to eventually become true at least once in a run. The operator $\Diamond$ is therefore pronounced eventually, or diamond. It conveniently captures the notion of liveness.

There is just one other temporal operator that we have to discuss here to complete the basic temporal logic framework. This is the unary next operator which is represented by the symbol X. The semantics of the X operator can be defined as follows.

Definition 6.12 (Next)

$$\sigma[i] \models X \; p \quad \Leftrightarrow \quad \sigma_{i+1} \models p.$$

The formula X p then simply states that property p is true in the immediately following state of the run.

Recurrence and Stability

There are many standard types of correctness properties that can be expressed with the temporal operators we have defined. Two important types are defined next.

Definition 6.13

A recurrence property is any temporal formula that can be written in the form $\Box\Diamond p$, where p is a state formula.

The recurrence property $\Box\Diamond p$ states that if p happens to be false at any given point in a run, it is always guaranteed to become true again if the run is continued.

Table 6.1. Frequently Used LTL Formulae

Formula	Pronounced	Type/Template
$\Box p$	always p	invariance
$\Diamond p$	eventually p	guarantee
$p \longrightarrow \Diamond q$	p implies eventually q	response
$p \longrightarrow q \cup r$	p implies q until r	precedence
$\Box\Diamond p$	always eventually p	recurrence (progress)
$\Diamond\Box p$	eventually always p	stability (non−progress)
$\Diamond p \longrightarrow \Diamond q$	eventually p implies eventually q	correlation

Definition 6.14

A stability property is any temporal formula that can be written in the form $\Diamond\Box p$, where p is a state formula.

The stability property $\Diamond\Box p$ states that there is always a point in a run where p will become invariantly true for the remainder of the run.

Recurrence and stability are in many ways dual properties that reflect a similar duality between the two earlier canonical correctness requirements we discussed: absence of non−progress cycles and absence of acceptance cycles.

There are other interesting types of duality. For instance, if "!" denotes logical negation, it is not hard to prove that in any context:

Equation 1

$$!\Box p \quad \Leftrightarrow \quad \Diamond !p$$

Equation 2

$$!\Diamond p \quad \Leftrightarrow \quad \Box !p$$

Which also implies, for instance, $\Box p \Leftrightarrow !\Diamond !p$ and $\Diamond p \Leftrightarrow !\Box !p$. We will refer to the above two standard equivalence rules by number in the remainder of this chapter. Some other commonly used rules are:

$$\begin{array}{lcl}
!(p\ U\ q) & \Leftrightarrow & (!q)U(!p \wedge !q) \\
!(p\ U\ q) & \Leftrightarrow & (!q)U(!p \wedge !q) \\
\Box(p \wedge q) & \Leftrightarrow & \Box p \wedge \Box q \\
\Diamond(p \vee q) & \Leftrightarrow & \Diamond p \vee \Diamond q \\
p\ U\ (q \vee r) & \Leftrightarrow & (p\ U\ q) \vee (p\ U\ r) \\
(p \wedge q)\ U\ r & \Leftrightarrow & (p\ U\ r) \wedge (q\ U\ r) \\
p\ U\ (q \vee r) & \Leftrightarrow & (p\ U\ q) \vee (p\ U\ r) \\
(p \wedge q)\ U\ r & \Leftrightarrow & (p\ U\ r) \wedge (q\ U\ r) \\
\Box\Diamond(p \vee q) & \Leftrightarrow & \Box\Diamond p \vee \Box\Diamond q \\
\Diamond\Box(p \wedge q) & \Leftrightarrow & \Diamond\Box p \wedge \Diamond\Box q
\end{array}$$

Many types of temporal logic formula are used so frequently that they have special names. One can consider such formulae templates for expressing common types of properties. Table 6.1 lists the most popular of these templates. We use the symbols $\longrightarrow$ and $\longleftrightarrow$ for logical implication and equivalence, defined as follows.

Definition 6.15 (Implication)

$$p \rightarrow q \models (!p) \vee q.$$

Definition 6.16 (Equivalence)

$$p \leftrightarrow q \models (p \rightarrow q) \wedge (q \rightarrow p).$$

Armed with this logic, we can now revisit the earlier examples of temporal properties that we wanted to be able to express (p. 134), as shown in Table 6.2.

Table 6.2. Formalization of Properties

Formula	English
$\Box p$	p is invariantly true,
$\Diamond \Box \, !p$	p eventually becomes invariantly false
$\Box \Diamond \, !p$	p always eventually becomes false at least once more
$\Box (q \longrightarrow !p)$	q always implies !p
$\Box (p \longrightarrow \Diamond \, q)$	p always implies eventually q

Using Temporal Logic

The logic looks straightforward enough, and it is not difficult to develop an intuition for the meaning of the operators. Still, it can sometimes be difficult to find the right formalization in temporal logic of informally stated requirements. As an example, consider the informal system requirement that p implies q. The formalization that first comes to mind is

$$p \rightarrow q$$

which is almost certainly wrong. Note that as a formula in temporal logic this property must hold for every run of the system. There are no temporal operators used here, just logical implication. This means that it holds if and only if

$$\sigma \models (!p) \vee q$$

which holds if in the first state of the run either p is false or q is true. It says nothing about the remaining steps in σ[2]. To make the property apply to all steps in the run we would have to change it into

$$\Box (p \rightarrow q)$$

but also that is most likely not what is meant, because this expresses merely a logical implication between p and q, not a temporal implication. If a temporal implication was meant, the formula should be written as follows:

$$\Box (p \rightarrow \Diamond q)$$

This still leaves some room for doubt, since it allows for the case where q becomes true in precisely the same state as where p becomes true. It would be hard to argue that this accurately captures the notion that the truth of q is somehow caused by the truth of p. To capture this, we need to modify the formula again, for instance, by adding a next operator.

$$\Box (p \rightarrow X (\Diamond q))$$

After all this work, this formula may still prove to be misleading. If the antecedent p is invariantly false throughout each run, for instance, the property will be satisfied. In this case, we call the property vacuously true. It is almost surely not what we meant when we formalized the property. This brings us to the final revision of the formula by adding the statement that we expect p to become true at some point. This produces the final form

$$\Box (p \rightarrow X (\Diamond q)) \wedge \Diamond p$$

which is quite different from the initial guess of ($p \longrightarrow q$).

Valuation Sequences

Let P be the set of all state formulae that are used in a given temporal logic formula. Each such state formula is typically represented by a lower−case propositional symbol (say p or q). Let, further, V represent the set of all the possible boolean truth assignments to the propositional symbols in P (i.e., set V has $2^{|P|}$ elements, where |P| is the number of elements of set P). We call V the set of valuations of P. With each run σ of a system we can now associate a sequence of valuations from V, denoting the specific boolean values that all propositional symbols take, as we illustrated earlier for the system in Figure 6.3. We will refer to that sequence as $V(\sigma)$.

Stutter Invariance

The next operator, X, can be useful to express complex system requirements, but it should be used with caution. Note first that in a distributed system the very notion of a "next" state is somewhat ambiguous. It is usually unknown and unknowable how the executions of asynchronously execution processes are going to be interleaved in time. It can therefore usually not be determined with certainty how a given current system state relates to the one that happens to follow it after one more step in a run. We can assume that every process will make finite progress, unless it becomes permanently blocked, but it is usually not justified to make more specific assumptions about the precise rate of progress, relative to other processes, since this may depend on uncontrollable and often unobservable aspects of a distributed system, such as the relative speeds of processors or subtle details of process scheduling.

These somewhat vague notions about what can and what cannot safely be stated about the runs of a distributed system can be made more precise with the help of the notion of stutter invariance.

Consider a run σ and its valuation ϕ = V(σ). Remember that ϕ is a sequence of truth assignments to the elements of a finite set of boolean propositions P used in a given temporal formula. Two subsequent elements of this sequence are either equal or they differ. We will replace series of equal consecutive elements in a valuation with a single symbol with the number of repetitions recorded in a superfix.

Let N be a sequence of positive numbers N_1, N_2, ... Each valuation ϕ of a run can then be written in the form

$$\phi_1^{N_1}, \phi_2^{N_2}, \cdots$$

with an appropriate choice for N. Given such a sequence ϕ, we can derive a stutter−free variant of ϕ by setting all elements of N to one: N= 1, 1, ... We can also derive a set of variants of ϕ that includes all possible choices for N. Such a set is called the stutter extension of ϕ and written as E(ϕ).

For temporal logic formula f to be satisfied on some run σ, the valuation V(σ) must satisfy the formula, that is, we must have V(σ) |= f. We are now ready to define stutter invariance.

Definition 6.17 (Stutter invariance)

A temporal logic formula f is stutter invariant if and only if

$$V(\sigma) \models f \rightarrow \forall \phi, \phi \in \mathsf{E}(\sigma),\ V(\phi) \models f.$$

This means that the property is stutter invariant if it is insensitive to the number of steps that individual valuations of the boolean propositions remain in effect.

We argued earlier that it would be dangerous for the correctness of a system to depend on execution speed, so this nicely captures our intuition about well−formed formulae in temporal logic: such formulae should be stutter invariant.

It can be shown that if we bar the next operator, X, from temporal logic, the temporal formulae that we write will be guaranteed to be stutter invariant. Moreover, it can also be shown that without the next operator we can precisely express all stutter invariant properties. This does not mean that a temporal logic formula that includes a X operator is necessarily not stutter invariant: it may well be so, but it is not guaranteed.

In a later chapter we will see another reason why we will want to be cautious with the use of X: properties that are known to be stutter invariant can be verified more efficiently than properties that do not have this property.

Fairness

One of the attractive features of LTL is that it can be used to express a fairly broad range of fairness assumptions. SPIN itself supports only a limited notion of fairness that applies only to the specific way in which process−level non−determinism is resolved (i.e., it applies only to process scheduling decisions). In some cases, we may want to express that also non−deterministic choices within a process are resolved conform some user−specified notion a fairness. These types of fairness conditions are readily expressed in LTL. If, for instance, our correctness claim is ϕ, we can add the fairness constraint that if a process of type P visits a state labeled U infinitely often, it must also visit a state labeled L infinitely often, by adding a conjunct to the property:

$$\phi \ /\backslash ((\Box \Diamond P@U) \rightarrow (\Box \Diamond P@L)).$$

From Logic To Automata

It was shown in the mid eighties that for every temporal logic formula there exists a Büchi automaton that accepts precisely those runs that satisfy the formula. There are algorithms that can mechanically convert any temporal logic formulae into the equivalent Büchi automaton. One such algorithm is built into SPIN.

Strictly speaking, the system description language PROMELA does not include syntax for the specification of temporal logic formulae, but SPIN does have a separate parser for such formulae and it can mechanically translate them into PROMELA syntax, so that LTL can effectively become part of the language that is accepted by SPIN. LTL, however, can only be used for specifying correctness requirements on PROMELA verification models. The models themselves cannot be specified in LTL. SPIN's conversion algorithm translates LTL formulae into `never` claims, and it automatically places accept labels within the claim to capture the semantics of the ω–regular property that is expressed in LTL.

To make it easier to type LTL formulae, the box (always) operator is written as a combination of the two symbols `[]`, and the diamond operator (eventually) is written as the two symbols <>. SPIN only supports the strong version of the until operator, represented by the capital letter `U`. To avoid confusion, state properties are always written with lower–case symbols such as `p` and `q`.

We can, for instance, invoke SPIN as follows:

```
$ spin -f '<>[] p"
never {    /* <>[]p */
T0_init:
        if
        :: (p) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (p) -> goto accept_S4
        fi
}
```

Note carefully that in a UNIX environment the temporal logic formula must be quoted to avoid misinterpretation of the angle symbols by the command interpreter, and to secure that the formula is seen as a single argument even if it contains spaces.[1] SPIN places braces around all expressions to make sure that there can be no surprises in the enforcement of precedence rules in their evaluation. Note also that the guard condition `(1)` is equivalent to the boolean value true.

> [1] On some, but not all, UNIX systems, the argument with the formula can also be enclosed in double quotes.
>
> A note on syntax: SPIN accepts LTL formulae that consist of propositional symbols (including the predefined terms `true` and `false`), unary and binary temporal operators, and the three logical operators ! (logical negation), ∧ (logical and), and ∨ (logical or). The logical and operator can also be written in C style as `&&`, and the logical or operator can also be written as `||`. Also supported are the abbreviations –> for logical implication and <–> for

logical equivalence (see Definitions 6.15 and 6.16). Arithmetic operators (e.g., +, −, *, /) and relational operators (e.g., >, >=, <, <=, ==, !=) cannot appear directly in LTL formulae.

For example, the following attempt to convert a formula fails:

```
$ spin -f '([] p -> <> (a+b <= c))'
tl_spin: expected ')', saw '+'
tl_spin: ([] p -> <> (a+b <= c))
-------------------------^
```

To succesfully convert this formula, we must introduce a new propositional symbol to hide the arithmetic and relational operators, as follows:

```
#define q            (a+b <= c)
```

We can now invoke SPIN's converter on the new formula:

```
$ spin -f ' ([] p -> <> q)'
```

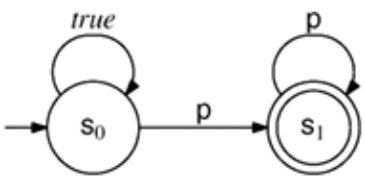
Also be aware of operator precedence rules. The formula above is quite different, for instance, from:

```
$ spin -f ' ([] (p -> <> q))'
```

(Hint: try the conversions with SPIN to see the difference).

The automaton for the formula $\Diamond \Box p$ is shown in Figure 6.4. The automaton has two states, one initial state s_0, which corresponds to the `never` claim state `T0_init`, and one accepting state s_1, which corresponds to `never` claim state `accept_S4`.

Figure 6.4. Automaton for $\Diamond \Box p$



Note that the automaton contains no transitions labelled `(!p)`, in state `accept_S4` or s_1. The reason is that when `p` becomes false after we have reach this state, the continuation of the run can no longer lead to acceptance. The automaton needs to monitor only those runs that may still produce a counterexample if

continued sufficiently far. All other runs are irrelevant to the verification procedure and can summarily be rejected. A direct benefit of leaving the automaton incomplete in this sense is therefore that the verifier will have to inspect fewer runs, which makes it more efficient.

The `never` claim that is generated by SPIN can be included into a PROMELA model to check if the behavior defined by the temporal logic formula can occur in the model, provided that the propositional symbols that are used (here only `p`) are defined. Typically, the definitions are done with macros, for instance, as follows:

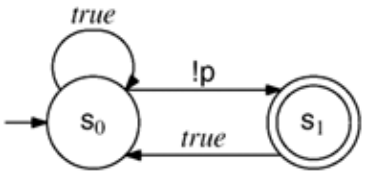
```
#define p      (x > 0 && x <= 100)
```

SPIN will flag it as an error if an accepting run can be found that matched the behavior expressed. That is, SPIN will find those runs that satisfy the LTL formula that was used to generate the `never` claim. To show that a property such as ◊□ p should not be violated, we can simply negate the formula before the claim is generated, for instance, as follows:

```
$ spin -f ' !<>[]p'
never {     /* !<>[]p */
T0_init:
        if
        :: (!(p)) -> goto accept_S9
        :: (1) -> goto T0_init
        fi;
accept_S9:
        if
        :: (1) -> goto T0_init
        fi;
}
```

The corresponding automaton is shown in Figure 6.5.

Figure 6.5. Automaton for ¬◊□ p = □ ◊ ¬ p



Because of the known equivalences we can deduce that !<>□p is equivalent to □!□p which in turn is equivalent to □◊!p. This means that if we ask SPIN to convert the formula □◊!p we should get the same result as for !<>□p. This is readily confirmed by a little experiment:

```
$ spin -f ' []<>!p'
never {     /*   []<>!p */
T0_init:
        if
        :: (!(p)) -> goto accept_S9
```

```
        :: (1) -> goto T0_init
        fi;
accept_S9:
        if
        :: (1) -> goto T0_init
        fi;
}
```

It should be carefully noted that the automaton for neither the original nor the negated version of this formula can accept finite runs (cf. p. 130), and neither version necessarily matches all possible system behaviors. The claim is meant to capture only those accepting runs that satisfy the temporal formula; no more and no less.

The automata that are produced by most conversion algorithms are not always the smallest possible. It can be useful to understand how `never` claims work, so that in special cases, when performance is at a premium, the claim automaton can be edited to remove unnecessary states or transitions. A detailed explanation of the working of `never` claims can be found in Chapter 4, p. 85.

In some cases, also, it can be difficult, or risky, to manually change a SPIN generated claim. It can then be useful to compare the SPIN generated claims with those produced by alternate converters. As an example, consider the following suboptimal claim for the requirement $\Box(p \longrightarrow \Diamond q)$:

```
$ spin -f ' [] (p -> <> q)'
never {     /* [] (p -> <> q) */
T0_init:
        if
        :: ((! (p)) || (q)) -> goto accept_S20
        :: (1) -> goto T0_S27
        fi;

accept_S20:
        if
        :: ((! (p)) || (q)) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;

accept_S27:
        if
        :: (q) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;

T0_S27:
        if
        :: (q) -> goto accept_S20
        :: (1) -> goto T0_S27
        :: (q) -> goto accept_S27
        fi;
}
```

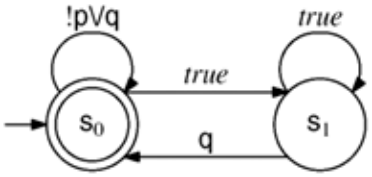
An alternative converter, [2] called `ltl2ba`, written by Paul Gastin and Denis Oddoux, produces a smaller claim:

[2] Gastin and Oddoux's converter is available as part of the standard SPIN distribution. See Appendix D.

```
$ ltl2ba -f '[] (p -> <> q)'
never { /* [] (p -> <> q) */
accept_init:
        if
        :: (!p) || (q) -> goto accept_init
        :: (1) -> goto T0_S2
        fi;
T0_S2:
        if
        :: (1) -> goto T0_S2
        :: (q) -> goto accept_init
        fi;
}
```

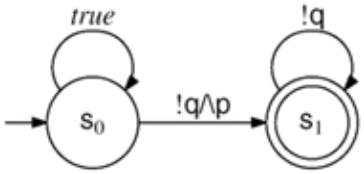
The corresponding automaton structure is shown in Figure 6.6.

Figure 6.6. Automaton for $\Box(p \longrightarrow \Diamond q)$



The logical negation of this last requirement is also interesting. It can be derived as follows:
The corresponding `never` claim is shown in Figure 6.7.

Figure 6.7. `Never` Automaton for $\Diamond(p \wedge \Box !q)$



We can of course also let SPIN do the derivation of the negated formula. The `never` claim that matches Figure 6.7 is generated as follows:

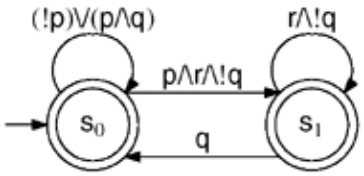
```
$ spin -f '![] (p -> <> q)'
never {    /* ![] (p -> <> q) */
T0_init:
        if
        :: (! ((q)) && (p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
```

```
        :: (! ((q))) -> goto accept_S4
        fi;
}
```

This time the built–in algorithm from SPIN does much better in generating a small automaton. Note carefully that using `else` instead of `(1)` (i.e., `true`) in the first selection construct would imply a check for just the first occurrence of expression (!(q)&&(p)) becoming true in the run. The claim as generated checks for all such occurrences, anywhere in a run.

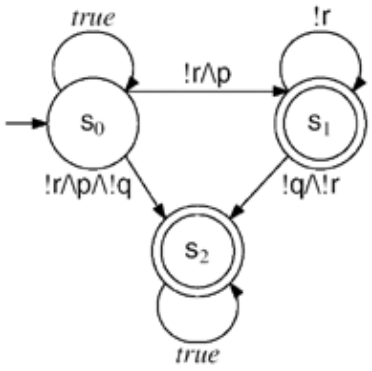
As a slightly more complicated variation on the previous example, consider the requirement $\Box(p \rightarrow (r \mathcal{U} q))$. The corresponding automaton is shown in Figure 6.8. In this case, all possible (infinite) runs of the automaton are accepting. But not all runs are possible. Runs in which, for instance, r becomes false before q becomes true in state s_1 are not accepted. The negation of the property can be derived as follows.

Figure 6.8. Automaton for $\Box(p \rightarrow (r \mathcal{U} q))$



Of course, it is typically easier to let SPIN handle the negation and do the conversion than to work out the negations manually. The automaton that corresponds to either the LTL formula $\Diamond(p \wedge !(r \mathcal{U} q))$ or the formula $!\Box(p \rightarrow (r \mathcal{U} q))$ is shown in Figure 6.9.

Figure 6.9. Automaton for $\Diamond(p \wedge !(r \mathcal{U} q))$



This three–state ω–automaton can be generated in SPIN's `never` claim format as follows:

```
$ spin -f '![] (p -> (q U r))'
never {    /* ![] (p -> (q U r)) */
T0_init:
        if
        :: (!(r) && (p)) -> goto accept_S4
        :: (!(r) && (p) && (!(q))) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
```

```
        :: (!(r)) -> goto accept_S4
        :: (!(q)) && (!(r)) -> goto accept_all
        fi;
accept_all:     /* instant violation */
        skip
}
```

Note again the use of `(1)` (or equivalently `true`) instead of `else` to allow us to initiate the check anywhere in an infinite run.

Not all properties of interest can be specified in LTL, so in some cases we can build a stronger property by hand–writing a `never` claim directly, instead of deriving one from an LTL formula. Technically, the properties that can be express in LTL are a subset of the set of ω–regular properties. PROMELA `never` claims can express all ω–regular properties, and are therefore some what more expressive than LTL alone.

An Example

Consider the following model:

```
int x = 100;

active proctype A()
{
        do
        :: x%2 -> x = 3*x+1
        od
}

active proctype B()
{
        do
        :: !(x%2) -> x = x/2
        od
}
```

What can the range of values of `x` be? We may want to prove that `x` can never become negative or, more interestingly, that it can never exceed its initial value of one hundred. We can try to express this in the following formula:

```
$ spin -f ' [] (x > 0 && x <= 100)'     # wrong
tl_spin: expected ')', saw '>'
tl_spin: [] (x > 0 && x <= 100)
-----------------^
```

But this is not right for two reasons. The first is indicated by the syntax error that SPIN flags. For SPIN to be able to translate a temporal formula it may contain only logical and temporal operators and propositional symbols: it cannot directly include arithmetic or relational expressions. We therefore have to introduce a propositional symbol to represent the expression `(x > 0 && x <= 100)`, and write

```
$ spin -f '[] p'      # better, but still wrong
never {    /* []p */
accept_init:
T0_init:
        if
        :: p -> goto T0_init
        fi;
}
```

Elsewhere in the model itself we can now define `p` as:

```
#define p       (x > 0 && x <= 100)
```

To see that this is still incorrect, notice that we have used SPIN to generate a `never` claim: expressing behavior that should never happen. We can use SPIN only to check for violations of requirements. So what we really meant to state was that the following property, the violation of the first, cannot be satisfied:

```
$ spin -f '![] p'       # correct
never {    /* ![]p */
T0_init:
        if
        :: (!(p)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        true
}
```

This automaton exits, signifying a violation, when the condition `p` ever becomes false, which is what we intended to express. SPIN can readily prove that violations are not possible, proving that with the given initialization the value of `x` is indeed bounded.

Another property we can try to prove is that the value of `x` eventually always returns to one. Again, to check that this is true we must check that the opposite cannot happen. First we introduce a new propositional symbol `q`.

```
#define q       (x == 1)
```

Using `q`, the negated property is written as follows:

```
$ spin -f '![]<>q'
never {    /* ![]<>q */
T0_init:
        if
        :: (!(q)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (!(q)) -> goto accept_S4
        fi;
}
```

Also in this case SPIN readily proves the truth of the property.

Note also that to make these claims about the possible values of the integer variable `x`, the scope of the variable must include the `never` claim. This means that we can only make these claims about global variables. A local variable is never within the scope of a PROMELA `never` claim.

Omega–Regular Properties

PROMELA `never` claims can express a broader range of properties that can be expressed in temporal logic, even when the next operator is allowed. Formally, a `never` claim can express any ω–regular property. To capture a property that is not expressible in temporal logic we can try to encode it directly into a `never` claim, but this is an error–prone process. There is an alternative, first proposed by Kousha Etessami. The alternative is to extend the formalism of temporal logic. A sufficient extension is to allow each temporal logic formula to be prefixed by an existential quantifier that is applied to one of the propositional symbols in the formula. Etessami proved that this single extension suffices to extend the power of temporal logic to cover all ω–regular properties.

Consider the case where we want to prove that it is impossible for `p` to hold only in even steps in a run, but never at odd steps. The temporal logic property `[] X p` would be too strong for this, since it would require `p` to hold on all even steps. This property can be expressed with existential quantification over some pseudo variable `t` as follows.

```
{E t} t && [](t -> X !t) && [](!t -> X t) && [](p -> !t)
```

This property states that there exists a propositional symbol `t` that is initially true and forever alternates between true and false, much like a clock. The formula further states that the truth of the `p` that we are interested in always logically implies the falseness of the alternating `t`. The automaton that corresponds to this formula is as follows.

```
$ eqltl -f '{E t} t && [](t -> X !t) && \
        [] (!t -> X t) && [](p -> !t)'
never {
accept0:
        if
        :: (!(p)) -> goto accept1
        fi;
accept1:
        if
        :: (1) -> goto accept0
        fi
}
```

In the first step, and every subsequent odd step in the run, `p` is not allowed to be true. No check on the value of `p` is in effect during the even steps.

The `eqltl` program was developed by Kousha Etessami.

Other Logics

The branch of temporal logic that we have described here, and that is supported by the SPIN system, is known as linear time temporal logic. The prefix linear is due to the fact that these formulae are evaluated over single sequential runs of the system. With the exception of the small extension that we discussed in the previous section, no quantifiers are used. Linear temporal logic is the dominant formalism in software verification. In applications of model checking to hardware verification another version of temporal logic is frequently used. This logic is known as branching time temporal logic, with as the best known example CTL (an acronym for computation tree logic), which was developed at Carnegie Mellon University. CTL includes both universal and existential quantification, and this additional power means that CTL formulae are evaluated over sets of executions (trees), rather than over individual linear execution paths. There has been much debate in the literature about the relative merits of branching and linear time logics, a debate that has never culminated in any clear conclusions. Despite many claims to the contrary, there is no definitive advantage to the use of either formalism with regard to the complexity of verification. This complexity is dominated by the size of the verification model itself, not by the type of logic used to verify it. This issue, and some other frequently debated issues in model checking, is explored in greater detail in Appendix B.

Bibliographic Notes

The basic theory of finite automata was developed in the fifties. A good summary of the early work can be found in Perrin [1990]. The theory of ω–automata dates back almost as far, starting with the work of Büchi [1960]. An excellent survey of this work, including definitions of the various types of acceptance conditions, can be found in Thomas [1990].

Amir Pnueli's influential paper, first proposing the use of temporal logic in the analysis of distributed systems, is Pnueli [1977]. The main notions used in the definition of temporal logic were derived from earlier work on tense logics. Curiously, this work, including the definition of some of the key operators from temporal logic, did not originate in computer science but in philosophy, see, for instance, Prior [1957,1967], and Rescher and Urquhart [1971]. An excellent overview of temporal logic can be found in Emerson [1990].

The correspondence between linear temporal logic formulae and Büchi automata was first described in Wolper, Vardi, and Sistla [1983]. An efficient conversion procedure, which forms the basis for the implementation used in SPIN, was given in Gerth, Peled, Vardi, and Wolper [1995]. The SPIN implementation uses some further optimizations of this basic procedure that are described in Etessami and Holzmann [2000].

There are several other implementations of the LTL conversion procedure, many of which can outperform the procedure that is currently built into SPIN. A good example is the procedure outlined in Gastin and Oddoux [2001]. Their converter, called `ltl2ba`, is available as part of the SPIN distribution.

The notion of stuttering is due to Lamport [1983], see also Peled, Wilke, and Wolper [1996] and Peled and Wilke [1997]. Etessami's conversion routine for handling LTL formulae with existential quantification is described in Etessami, Wilke, and Schuller [2001].