where

\[ R_i(V, V') \equiv (v_i \Leftrightarrow f_1(V)) \land \bigwedge_{j \neq i} (v_j \Leftrightarrow v_j). \]

Note that some component may change repeatedly, without another component ever making a step. In practice, this is extremely unlikely. It is possible to augment the model with an additional fairness constraint that will disallow such behaviors. This topic will be discussed further in the next chapter.

To illustrate the difference between the synchronous and the asynchronous models, consider the following example. Let \( V = \{v_0, v_1\} \), \( v'_0 = v_0 \oplus v_1 \) and \( v'_1 = v_0 \oplus v_1 \). Let \( s \) be a state with \( v_0 = 1 \land v_1 = 1 \). According to the synchronous model, the only successor of \( s \) is the state with \( v_0 = 0 \land v_1 = 0 \), since both assignments are executed simultaneously. According to the asynchronous model, the state \( s \) has two successors:

1. \( v_0 = 0 \land v_1 = 1 \) (the assignment to \( v_0 \) is taken first).
2. \( v_0 = 1 \land v_1 = 0 \) (the assignment to \( v_1 \) is taken first).

2.2.2 Programs

All of the programs we consider are asynchronous. We start by discussing sequential programs because concurrent programs are composed of sequential components. The approach that we use is similar to the approach used in the book by Manna and Pnueli [186]. For a more detailed treatment of these issues, we refer the reader to that book. A program consists of states that are sequentially composed with each other. We describe a translation procedure \( C \) that takes the text of a sequential program \( P \) and transforms it into a first order formula \( R \) that represents the set of transitions of the program. Without loss of generality, we assume that each statement has a unique entry point and a unique exit point. The transition procedure is simplified significantly if each entry and exit point of a statement in the program is uniquely labeled. Thus, we define a labeling transformation that given an unlabeled program \( P \) results in a labeled program \( P^C \).

The labeling transformation defined below attaches a single label with the entry point of each statement in \( P \), except for \( P \) itself. No two attached labels are identical. In sequential programs, the exit point of a statement is identical to the entry point of the following statement. Thus, it is sufficient to label entry points. If we also provide labels for the entry and the exit points of \( P \), then we get a unique labeling of the entry and exit points of all statements of the program.

Since we do not restrict ourselves to a specific programming language, we define the labeling transformation for a number of statement types. It is easy to extend the definition to other statement types. Given a statement \( P \), the labeled statement \( P^C \) is defined as follows:
If $P$ is not a composite statement (e.g., $P$ is $x := e$, skip, wait, lock, unlock, etc.), then $P^C = P$.

- If $P = P_1 ; P_2$ then $P^C = P_1^C ; l'' : P_2^C$.

- If $P = \text{if } b \text{ then } P_1 \text{ else } P_2 \text{ end if}$, then $P^C = \text{if } b \text{ then } l_1 : P_1^C \text{ else } l_2 : P_2^C \text{ end if}$.

- If $P = \text{while } b \text{ do } P_1 \text{ end while}$, then $P^C = \text{while } b \text{ do } l_1 : P_1^C \text{ end while}$.

In the remainder of this section, we assume that $P$ is a labeled statement and that the entry and exit points of $P$ are labeled by $m$ and $m'$ respectively. Let $pc$ be a special variable called the \textit{program counter} that ranges over the set of program labels and an additional value $\perp$ called the \textit{undefined value}. The undefined value is needed when concurrent programs are considered. In this case, $pc = \perp$ indicates that the program is not active.

Let $V$ denote the set of program variables. Let $V'$ be the set of primed variables $v'$ for each $v \in V$, and let $pc'$ be the primed variable for $pc$. Recall that the unprimed copy refers to the value of the variables before a transition, whereas the primed copy refers to the value after the transition. Because each transition typically changes only a small number of the program variables, we will use $\text{same}(V)$ as an abbreviation for the formula

$$\bigwedge_{y \in V} (y' = y).$$

We first give the formula that describes the set of initial states of the program $P$. Given some condition $\text{pre}(V)$ on the initial values of the variables of $P$,

$$S_0(V, pc) \equiv \text{pre}(V) \land pc = m.$$  

The translation procedure $\mathcal{C}$ depends on three parameters: the entry label $l$, the labeled statement $P$, and the exit label $l'$. The procedure is defined recursively with one rule for each statement type in the language. $\mathcal{C}(l, P, l')$ describes the set of transitions in $P$ as a disjunction of all the transitions in the set. The disjunct for an individual transition determines the value of the boolean condition and the value of the program counter for which the transition may be executed. It is true whenever the transition is enabled and false otherwise.

- \textbf{Assignment:}

$$\mathcal{C}(l, v \leftarrow e, l') \equiv pc = l \land pc' = l' \land v' = e \land \text{same}(V \setminus \{v\})$$

- \textbf{Skip:}

$$\mathcal{C}(l, \text{skip}, l') \equiv pc = l \land pc' = l' \land \text{same}(V)$$
Sequential composition:
\[
C(l, P_1; l'': P_2, l') \equiv C(l, P_1, l'') \lor C(l'', P_2, l')
\]
The formula for the transitions of \(P_1; l'': P_2\) is a disjunction of the formulas for the transitions of \(P_1\) and of \(P_2\). Because of the intermediate label \(l''\), statement \(P_2\) will be executed after statement \(P_1\).

Conditional:
\[
C(l, \text{if } b \text{ then } l_1 : P_1 \text{ else } l_2 : P_2 \text{ end if}, l')
\]
is the disjunction of the following four formulas:
- \(pc = l \land pc' = l_1 \land b \land \text{same}(V)\)
- \(pc = l \land pc' = l_2 \land \neg b \land \text{same}(V)\)
- \(C(l_1, P_1, l')\)
- \(C(l_2, P_2, l')\)
The first disjunct corresponds to the case where condition \(b\) is true. In this case, statement \(P_1\) will be executed next. The second disjunct corresponds to the case where condition \(b\) is false. In this case, statement \(P_2\) will be executed next. Both disjuncts describe transitions that involve only a change of the program counter. The third and fourth disjuncts are formulas for the transitions of \(P_1\) and \(P_2\), respectively. Note that \(l'\) is the exit point for both \(P_1\) and \(P_2\). The translation for the if statement can easily be extended to handle nondeterministic choice between several alternatives.

While:
\[
C(l, \text{while } b \text{ do } l_1 : P_1 \text{ end while}, l')
\]
is the disjunction of the following three formulas:
- \(pc = l \land pc' = l_1 \land b \land \text{same}(V)\)
- \(pc = l \land pc' = l' \land \neg b \land \text{same}(V)\)
- \(C(l_1, P_1, l)\)
The first disjunct corresponds to the case where condition \(b\) is true. In this case, statement \(P_1\) will be executed next. The second disjunct corresponds to the case where condition \(b\) is false, in which case, the execution of the while statement terminates. The third disjunct is a formula for the set of transitions of \(P_1\). Note that the exit point of \(P_1\) is identical to the entry point of the while statement. Thus, if \(P_1\) terminates the execution of the while statement will restart.

### 2.2.3 Concurrent Programs

A concurrent program consists of a set of processes that can be executed in parallel. A process is a sequential statement as described in the previous section. Concurrent programs in which processes do not interact by means of message passing or shared variables are
usually easy to analyze and will not be considered further. In this section, we will consider asynchronous programs in which exactly one process can make a transition at any time. We begin by introducing some terminology that will be used throughout the section. \( V_i \) is the set of variables that can be changed by process \( P_i \). We do not require that these sets be disjoint. As before, \( V \) is the set of all program variables. The program counter of a process \( P_i \) is \( pc_i \). \( PC \) is the set of all program counters.

A concurrent program \( P \) has the form

\[
\text{cobegin } P_1 \parallel P_2 \parallel \ldots \parallel P_n \text{ coend}
\]

where \( P_1, \ldots, P_n \) are processes. The labeling transformation for sequential programs is extended so that a concurrent program can occur as a statement in a sequential program. The transformation attach the label to the entry point and to the exit point of each process. Unlike exit points in sequential programs, no exit point of a concurrent process is identical to an entry point. As a result, the exit points of processes must be explicitly labeled. As before, we assume that no two labels are identical and that the entry and exit points of \( P \) are labeled \( m \) and \( m' \), respectively.

- If \( P = \text{cobegin } P_1 \parallel P_2 \parallel \ldots \parallel P_n \text{ coend} \), then

\[
P^L = \text{cobegin } l_1 : P_1^L \parallel l_2 : P_2^L \parallel \ldots \parallel l_n : P_n^L \text{ coend}.
\]

The formula that describes the initial states of a concurrent program \( P \) is

\[
\mathcal{S}_0(V, PC) \equiv pre(V) \land pc = m \land \bigwedge_{i=1}^{n}(pc_i = \bot),
\]

where \( pc_i = \bot \) indicates that process \( P_i \) has not been activated yet and therefore cannot be executed from the current state.

The translation procedure \( C \) is extended to concurrent programs as follows: \( C(l, \text{cobegin } l_1 : P_1 \parallel l'_1 \parallel \ldots \parallel l_n : P_n \parallel l'_n \text{ coend}, l') \) is the disjunction of three formulas:

- \( pc = l \land pc'_1 = l_1 \land \ldots \land pc'_n = l_n \land pc' = \bot \)
- \( pc = \bot \land pc_1 = l'_1 \land \ldots \land pc_n = l'_n \land pc' = l' \land \bigwedge_{i=1}^{n}(pc'_i = \bot) \)
- \( \bigvee_{i=1}^{n} (C(l, P, l') \land same(V \setminus V_i) \land same(PC \setminus \{pc_i\})) \)

The first disjunct describes the initialization of the concurrent processes. A transition is made from the entry point of the \texttt{cobegin} statement to the entry points of the individual processes. The second disjunct describes the termination of the concurrent program. A transition is made from the exit points of the processes to the exit of the \texttt{cobegin} statement. This transition will only be executed if all the processes terminate. The third disjunct
describes the interleaved execution of the concurrent processes. The formula for the transition relation of process $P_i$ is conjected with $\text{same} \ (V \setminus V_i) \land \text{same}(PC \setminus \{pc_i\})$. This guarantees that a transition in process $P_i$ can only change variables in $V_i$. It also ensures that only one process can make a transition at any time.

**Shared Variables**

Recall that $V_i$ is the set of variables that may be changed by process $P_i$. Concurrent programs for which the sets $V_i$ overlap are called *shared variable* programs. We show how to extend the translation procedure $\mathcal{C}$ to some commonly used *process synchronization* statements. Such statements are frequently needed to provide processes with exclusive access to shared variables. These statements are atomic and treated by the labeling transformation accordingly. Assume that the statement belongs to the text of process $P_i$.

- **Wait:** Because our primary interest is in finite state programs, we only describe how to implement this statement using *busy waiting*. In particular, we do not consider implementations that require complex data structures like process queues. The statement $\text{wait}(b)$ repeatedly tests the value of the boolean variable $b$ until it determines that $b$ is true. When $b$ becomes true, a transition is made to the next program point.

$\mathcal{C}(l, \text{wait}(b), l')$ is a disjunction of the following two formulas:

- $(pc_i = l \land pc'_i = l \land \neg b \land \text{same}(V_i))$
- $(pc_i = l \land pc'_i = l' \land b \land \text{same}(V_i))$

- **Lock:** The statement $\text{lock}(v)$ is similar to the statement $\text{wait}(v = 0)$, except that when $v = 0$ is true the transition changes the value of $v$ to 1. This statement is often used to guarantee *mutual exclusion* by preventing more than one process from entering its critical region.

$\mathcal{C}(l, \text{lock}(v), l')$ is a disjunction of the following two formulas:

- $(pc_i = l \land pc'_i = l \land v = 1 \land \text{same}(V_i))$
- $(pc_i = l \land pc'_i = l' \land v = 0 \land \forall v' = 1 \land \text{same}(V_i \setminus \{v\}))$

- **Unlock:** The statement $\text{unlock}(v)$ assigns the value 0 to the variable $v$. Typically, this statement enables some other process to enter its critical region.

$\mathcal{C}(l, \text{unlock}(v), l') \equiv pc_i = l \land pc'_i = l' \land v' = 0 \land \text{same}(V_i \setminus \{v\})$

### 2.3 Example of Program Translation

Consider a simple *mutual exclusion* program

$P = m : \text{cobegin } P_0 || P_1 \text{ coend } m'$
with two processes $P_0$ and $P_1$, where

\[
\begin{align*}
P_0:: & \quad l_0: \quad \textbf{while True do} \\
& \quad NC_0: \quad \textsf{wait}(turn = 0); \\
& \quad CR_0: \quad turn := 1; \\
& \quad \textbf{end while}; \\
& l_0'
\end{align*}
\]

\[
\begin{align*}
P_1:: & \quad l_1: \quad \textbf{while True do} \\
& \quad NC_1: \quad \textsf{wait}(turn = 1); \\
& \quad CR_1: \quad turn := 0; \\
& \quad \textbf{end while}; \\
& l_1'
\end{align*}
\]

The program counter $pc$ of the program $P$ takes only three values: $m$, the label of the entry point of $P$; $m'$, the label of the exit point of $P$; and $\bot$, the value of $pc$ when $P_1$ and $P_2$ are active. Each process $P_i$ has a program counter $pc_i$ that ranges over the labels $l_i, l_i', NC_i, CR_i$, and $\bot$. The two processes share a single variable $turn$. Thus, $V = V_0 = V_1 = \{ \text{turn} \}$ and $PC = \{ pc, pc_0, pc_1 \}$. When the value of the program counter of a process $P_i$ is $CR_i$, the process is in its critical region. Both processes are not allowed to be in their critical regions at the same time. When the value of the program counter is $NC_i$, the process is in its noncritical region. In this case it waits until $turn = i$ in order to gain exclusive entry into the critical region.

The initial states of $P$ are described by the formula

\[
S_0(V, PC) \equiv pc = m \land pc_0 = \bot \land pc_1 = \bot.
\]

Note that no restriction is imposed on the value of $turn$. Thus, it may initially be either 0 or 1. Applying the translation procedure $C$ we obtain the formula for the transition relation of $P$, $\mathcal{R}(V, PC, V', PC')$, which is the disjunction of the following four formulas:

- $pc = m \land pc_0' = l_0 \land pc_1' = l_1 \land pc' = \bot$
- $pc_0 = l_0' \land pc_1 = l_1' \land pc' = m' \land pc_0' = \bot \land pc_1' = \bot$
- $C(l_0, P_0, l_0') \land \text{same}(V \setminus V_0) \land \text{same}(PC \setminus \{ pc_0 \})$, which is equivalent to $C(l_0, P_0, l_0') \land \text{same}(pc, pc_1)$
- $C(l_1, P_1, l_1') \land \text{same}(V \setminus V_1) \land \text{same}(PC \setminus \{ pc_1 \})$, which is equivalent to $C(l_1, P_1, l_1') \land \text{same}(pc, pc_0)$
For each process $P_i$, $\mathcal{C}(l_i, P_i, l'_i)$ is the disjunction of:

- $pc_i = l_i \land pc'_i = NC_i \land \text{True} \land \text{same(turn)}$
- $pc_i = NC_i \land pc'_i = CR_i \land \text{turn} = i \land \text{same(turn)}$
- $pc_i = CR_i \land pc'_i = l_i \land \text{turn'} = (i + 1) \mod 2$
- $pc_i = NC_i \land pc'_i = NC_i \land \text{turn} \neq i \land \text{same(turn)}$
- $pc_i = l_i \land pc'_i = l'_i \land \text{False} \land \text{same(turn)}$

The Kripke structure in Figure 2.2 is derived from the formulas $S_0$ and $\mathcal{R}$ as described in Section 2.1.1. By examining the state space of the Kripke structure, it is easy to see that the processes will never be in their critical regions at the same time. Thus, the program guarantees the required mutual exclusion property. However, this program fails to guarantee absence of starvation, since one of the processes may continuously try to enter its critical region without ever being able to do so, while the other process stays in its critical region forever. Later, we will see how to formulate and model check such properties.
3 Temporal Logics

In this chapter we describe a logic for specifying properties of the state transition systems or Kripke structures introduced in Section 2.1. The logic uses atomic propositions and boolean connectives such as conjunction, disjunction, and negation to build up complicated expressions describing properties of states. In reactive systems, we are also interested in describing the transitions between states. This is important because such systems interact with and continually respond to their environment. Traditional software verification methodologies, such as those due to Floyd [114] and Hoare [136], deal with the input-output semantics of programs. The internal details of how the computation is carried out are not reflected in the properties that can be specified and proved; only the input at the start of execution and the output at termination are described. In contrast, for reactive systems, the computation sequence is of primary importance, and many reactive systems are designed not to terminate.

Temporal logic is a formalism for describing sequences of transitions between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that eventually some designated state is reached, or that an error state is never entered. Properties like eventually or never are specified using special temporal operators. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. We will focus on a powerful logic called CTL* [61, 63, 105].

3.1 The Computation Tree Logic CTL*

Conceptually, CTL* formulas describe properties of computation trees. The tree is formed by designating a state in a Kripke structure as the initial state and then unwinding the structure into an infinite tree with the designated state at the root, as illustrated in Figure 3.1. The computation tree shows all of the possible executions starting from the initial state.

In CTL* formulas are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers A ("for all computation paths") and E ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- **X** ("next time") requires that a property holds in the second state of the path.
- The **F** ("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path.
- The **G** ("always" or "globally") specifies that a property holds at every state on the path.
The U ("until") operator is a bit more complicated since it is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.

R ("release") is the logical dual of the U operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

The remainder of this section contains a precise description of the syntax and semantics of CTL*. There are two types of formulas in CTL*: state formulas (which are true in a specific state) and path formulas (which are true along a specific path). Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then $p$ is a state formula.
- If $f$ and $g$ are state formulas, then $\neg f$, $f \lor g$ and $f \land g$ are state formulas.
- If $f$ is a path formula, then $E f$ and $A f$ are state formulas.

Two additional rules are needed to specify the syntax of path formulas:
If $f$ is a state formula, then $f$ is also a path formula.

If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $f \land g$, $X f$, $F f$, $G f$, $f U g$, and $f R g$ are path formulas.

$CTL^*$ is the set of state formulas generated by the above rules.

We define the semantics of $CTL^*$ with respect to a Kripke structure. Recall that a Kripke structure $M$ is a triple $(S, R, L)$, where $S$ is the set of states; $R \subseteq S \times S$ is the transition relation, which must be total (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \rightarrow 2^A$ is a function that labels each state with a set of atomic propositions true in that state. A path in $M$ is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. (Alternatively, we can think of a path as an infinite branch in the computation tree that corresponds to the Kripke structure.)

We use $\pi^i$ to denote the suffix of $\pi$ starting at $s_i$. If $f$ is a state formula, the notation $M, s \models f$ means that $f$ holds at state $s$ in the Kripke structure $M$. Similarly, if $f$ is a path formula, $M, \pi \models f$ means that $f$ holds along path $\pi$ in the Kripke structure $M$. When the Kripke structure $M$ is clear from the context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that $f_1$ and $f_2$ are state formulas and $g_1$ and $g_2$ are path formulas):

1. $M, s \models p \iff p \in L(s)$.
2. $M, s \models \neg f_1 \iff M, s \not\models f_1$.
3. $M, s \models f_1 \lor f_2 \iff M, s \models f_1$ or $M, s \models f_2$.
4. $M, s \models f_1 \land f_2 \iff M, s \models f_1$ and $M, s \models f_2$.
5. $M, s \models E g_1 \iff$ there is a path $\pi$ from $s$ such that $M, \pi \models g_1$.
6. $M, s \models A g_1 \iff$ for every path $\pi$ starting from $s$, $M, \pi \models g_1$.
7. $M, \pi \models f_1 \iff s$ is the first state of $\pi$ and $M, s \models f_1$.
8. $M, \pi \models \neg g_1 \iff M, \pi \not\models g_1$.
9. $M, \pi \models g_1 \lor g_2 \iff M, \pi \models g_1$ or $M, \pi \models g_2$.
10. $M, \pi \models g_1 \land g_2 \iff M, \pi \models g_1$ and $M, \pi \models g_2$.
11. $M, \pi \models X g_1 \iff M, \pi^i \models g_1$.
12. $M, \pi \models F g_1 \iff$ there exists a $k \geq 0$ such that $M, \pi^k \models g_1$.
13. $M, \pi \models G g_1 \iff$ for all $i \geq 0$, $M, \pi^i \models g_1$.
14. $M, \pi \models g_1 \ U g_2 \iff$ there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq j < k$, $M, \pi^j \not\models g_1$.
15. $M, \pi \models g_1 \ R g_2 \iff$ for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models g_1$ then $M, \pi^j \not\models g_2$.

It is easy to see that the operators $\lor$, $\neg$, $X$, $U$, and $E$ are sufficient to express any other $CTL^*$ formula.
\[ f \land g \equiv \neg (\neg f \lor \neg g) \]
\[ f R g \equiv \neg (\neg f U \neg g) \]
\[ F f \equiv \text{True} U f \]
\[ G f \equiv \neg F \neg f \]
\[ A(f) \equiv \neg E(\neg f) \]

### 3.2 CTL and LTL

In this section we consider two useful sublogics of CTL*: one is a branching-time logic and one is a linear-time logic. The distinction between the two is in how they handle branching in the underlying computation tree. In branching-time temporal logic the temporal operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for describing events along a single computation path.

Computation Tree Logic (CTL) [19, 61, 104] is a restricted subset of CTL* in which each of the temporal operators X, F, G, U, and R must be immediately preceded by a path quantifier. More precisely, CTL is the subset of CTL* that is obtained by restricting the syntax of path formulas using the following rule.

- If \( f \) and \( g \) are state formulas, then \( X f, F f, G f, f U g, \) and \( f R g \) are path formulas.

Linear Temporal Logic (LTL) [217], on the other hand, will consist of formulas that have the form \( A f \) where \( f \) is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an LTL path formula is either:

- If \( p \in AP \), then \( p \) is a path formula.
- If \( f \) and \( g \) are path formulas, then \( \neg f, f \lor g, f \land g, X f, F f, G f, f U g, \) and \( f R g \) are path formulas.

It can be shown [59, 105, 166] that the three logics that we have discussed have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula \( A(FG p) \). This formula expresses the property that along every path, there is some state from which \( p \) will hold forever. Likewise, there is no LTL formula that is equivalent to the CTL formula \( AG(\text{EF} p) \). The disjunction of these two formulas \( A(FG p) \lor AG(\text{EF} p) \) is a CTL* formula that is not expressible in either CTL or LTL.

Most of the specifications in this book will be written in the logic CTL. There are ten basic CTL operators:

- \( AX \) and \( EX \),
- AF and EF,
- AG and EG
- AU and EU,
- AR and ER.

Each of the ten operators can be expressed in terms of three operators EX, EG, and EU:

- $AX\ f = \neg EX(\neg f)$
- $EF\ f = E[True \ U\ f]$
- $AG\ f = \neg EF(\neg f)$
- $AF\ f = \neg EG(\neg f)$
- $A[f \ U\ g] \equiv \neg E[\neg g \ U\ (\neg f \land \neg g)] \land \neg EG\ \neg g$
- $A[f \ R\ g] \equiv \neg E[\neg f \ U\ \neg g]$
- $E[f \ R\ g] \equiv \neg A[\neg f \ U\ \neg g]$

The four operators that are used most widely are illustrated in Figure 3.2. The operators are easiest to understand in terms of the computation tree obtained by unfolding the Kripke model. Each computation tree has the state $s_0$ as its root.

Some typical CTL formulas that might arise in verifying a finite state concurrent program are given below:

- $EF(Start \land \neg Ready)$: It is possible to get to a state where $Start$ holds but $Ready$ does not hold.
- $AG(Req \rightarrow AF\ Ack)$: If a request occurs, then it will be eventually acknowledged.
- $AG(AF\ DeviceEnabled)$: The proposition $DeviceEnabled$ holds infinitely often on every computation path.
- $AG(EF\ Restart)$: From any state it is possible to get to the $Restart$ state.

Many of the methods to avoid the state explosion problem rely on compositional reasoning or abstraction. The logic that is typically used in these cases is more restricted and allows only universal path quantifiers. The restriction of $CTL^*$ to universal path quantifiers is called $ACTL^*$, and the restriction of $CTL$ to universal path quantifiers is called $ACTL$.

In order to avoid implicit existential path quantifiers resulting from the use of negation, we assume that the formulas are given in positive normal form, that is, negations are applied only to atomic propositions. To avoid the loss of expressive power, we need conjunction and disjunction, and both the $U$ and $R$ operators.
The formulas $\text{AF AG } a$ and $\text{AF AX } a$ are examples of ACTL formulas. These formulas are not expressible in LTL [59]. Because ACTL is a subset of CTL, the logics ACTL and LTL are incomparable. Moreover, ACTL* is more expressive than LTL. The formulas $\text{AG EF Start}$ and $\text{AG } \neg\text{ AF Start}$ are not in ACTL.

### 3.3 Fairness

Finally, we consider the issue of fairness. In many cases, we are only interested in the correctness along fair computation paths. For example, if we are verifying an asynchronous circuit with an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs forever. Alternatively, we may want to consider communication protocols that operate over reliable channels which have the property that no message is ever continuously transmitted but never received. Such properties cannot be expressed directly in CTL [59, 104, 105] but can be expressed in CTL*. In order to deal with fairness in CTL we must modify its semantics slightly. We call the new semantics of the logic the fair semantics. A fairness constraint can be an arbitrary set of states, usually
described by a formula of the logic. If fairness constraints are interpreted as sets of states, then a fair path must contain an element of each fairness constraint infinitely often. If fairness constraints are interpreted as CTL formulas, then a path is fair if each constraint is true infinitely often along the path. The path quantifiers in the logic are then restricted to fair paths.

Formally, a fair Kripke structure is a 4-tuple \( M = (S, R, L, F) \), where \( S, L, \) and \( R \) are defined as before and \( F \subseteq 2^S \) is a set of fairness constraints (often called generalized Büchi acceptance conditions). Let \( \pi = s_0, s_1, \ldots \) be a path in \( M \). Define

\[
\inf(\pi) = \{ s \mid s = s_i \text{ for infinitely many } i \}.
\]

We say that \( \pi \) is fair if and only if for every \( P \in F \), \( \inf(\pi) \cap P \neq \emptyset \). The semantics of CTL* with respect to a fair Kripke structure is very similar to the semantics of CTL* with respect to an ordinary Kripke structure. We will write \( M, s \models_F f \) to indicate that the state formula \( f \) is true in state \( s \) of the fair Kripke structure \( M \). Similarly, we write \( M, \pi \models_F g \) to indicate that the path formula \( g \) is true along path \( \pi \) in \( M \). Only clauses 1, 5 and 6 in the original semantics change.

1. \( M, s \models_F p \) ⇔ there exists a fair path starting from \( s \) and \( p \in L(s) \).
2. \( M, s \models_F E(g_1) \) ⇔ there exists a fair path \( \pi \) starting from \( s \) such that \( \pi \models_F g_1 \).
3. \( M, s \models_F A(g_1) \) ⇔ for all fair paths \( \pi \) starting from \( s, \pi \models_F g_1 \).

To illustrate the use of fairness, consider again the communication protocol for reliable channels. There is one fairness constraint for each channel that expresses the reliability of that channel. A possible choice for the fairness constraint associated with channel \( i \) is the set of states that satisfy the formula \( \neg \text{send}_i \lor \text{receive}_i \). Thus, a computation path is fair if and only if for every channel, infinitely often either a message is not sent or a message is received. Other notions of fairness are dealt with in \([116]\).