By contrast, Chapter 4 describes a very different verification technique which in terms of the above classification is a proof-based, computer-assisted, property-verification approach. It is intended to be used for programs which we expect to terminate and produce a result.

Model checking is based on *temporal logic*. The idea of temporal logic is that a formula is not *statically* true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others. Thus, the static notion of truth is replaced by a *dynamic* one, in which the formulas may change their truth values as the system evolves from state to state. In model checking, the models $\mathcal{M}$ are *transition systems* and the properties ϕ are formulas in temporal logic. To verify that a system satisfies a property, we must do three things:

- model the system using the description language of a model checker, arriving at a model $\mathcal{M}$;
- code the property using the specification language of the model checker, resulting in a temporal logic formula ϕ;
- Run the model checker with inputs $\mathcal{M}$ and ϕ.

The model checker outputs the answer 'yes' if $\mathcal{M} \models \phi$ and 'no' otherwise; in the latter case, most model checkers also produce a trace of system behaviour which causes this failure. This automatic generation of such 'counter traces' is an important tool in the design and debugging of systems.

Since model checking is a *model-based* approach, in terms of the classification given earlier, it follows that in this chapter, unlike in the previous two, we will not be concerned with semantic entailment ($\Gamma \models \phi$), or with proof theory ($\Gamma \vdash \phi$), such as the development of a natural deduction calculus for temporal logic. We will work solely with the notion of satisfaction, i.e. the satisfaction relation between a model and a formula ($\mathcal{M} \models \phi$).

There is a whole zoo of temporal logics that people have proposed and used for various things. The abundance of such formalisms may be organised by classifying them according to their particular view of 'time.' *Linear-time* logics think of time as a set of paths, where a path is a sequence of time instances. *Branching-time* logics represent time as a tree, rooted at the present moment and branching out into the future. Branching time appears to make the non-deterministic nature of the future more explicit. Another quality of time is whether we think of it as being *continuous* or *discrete*. The former would be suggested if we study an analogue computer, the latter might be preferred for a synchronous network.

Temporal logics have a dynamic aspect to them, since the truth of a formula is not fixed in a model, as it is in predicate or propositional logic, but depends on the time-point inside the model. In this chapter, we study a logic where time is linear, called *Linear-time Temporal Logic* (LTL), and another where time is branching, namely *Computation Tree Logic* (CTL). These logics have proven to be extremely fruitful in verifying hardware and communication protocols; and people are beginning to apply them to the verification of software. *Model checking* is the process of computing an answer to the question of whether $\mathcal{M}, s \models \phi$ holds, where ϕ is a formula of one of these logics, $\mathcal{M}$ is an appropriate model of the system under consideration, s is a state of that model and $\models$ is the underlying satisfaction relation.

Models like $\mathcal{M}$ should not be confused with an actual physical system. Models are abstractions that omit lots of real features of a physical system, which are irrelevant to the checking of ϕ. This is similar to the abstractions that one does in calculus or mechanics. There we talk about *straight* lines, *perfect* circles, or an experiment *without friction*. These abstractions are very powerful, for they allow us to focus on the essentials of our particular concern.

3.2 Linear-time temporal logic

Linear-time temporal logic, or LTL for short, is a temporal logic, with connectives that allow us to refer to the future. It models time as a sequence of states, extending infinitely into the future. This sequence of states is sometimes called a computation path, or simply a path. In general, the future is not determined, so we consider several paths, representing different possible futures, any one of which might be the 'actual' path that is realised.

We work with a fixed set `Atoms` of atomic formulas (such as $p, q, r, \ldots$, or $p_1, p_2, \ldots$). These atoms stand for atomic facts which may hold of a system, like *'Printer Q5 is busy,'* or *'Process 3259 is suspended,'* or *'The content of register* `R1` *is the integer value* 6*.'* The choice of atomic descriptions obviously depends on our particular interest in a system at hand.

3.2.1 Syntax of LTL

Definition 3.1 Linear-time temporal logic (LTL) has the following syntax given in Backus Naur form:

$$\begin{aligned}\phi ::= {} & \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \\ & \mid (\mathrm{X}\,\phi) \mid (\mathrm{F}\,\phi) \mid (\mathrm{G}\,\phi) \mid (\phi\ \mathrm{U}\ \phi) \mid (\phi\ \mathrm{W}\ \phi) \mid (\phi\ \mathrm{R}\ \phi) \end{aligned} \tag{3.1}$$

where p is any propositional atom from some set `Atoms`.

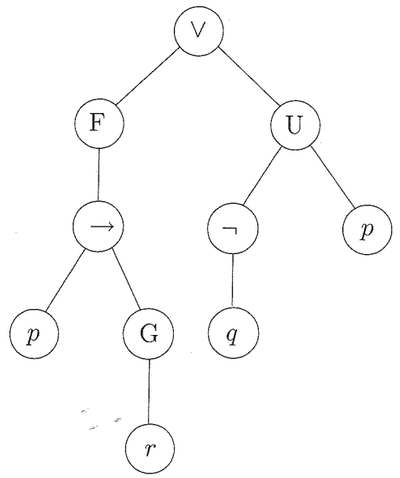

Figure 3.1. The parse tree of $(\mathrm{F}\,(p \to \mathrm{G}\,r) \vee ((\neg q)\ \mathrm{U}\ p))$.

Thus, the symbols $\top$ and $\bot$ are LTL formulas, as are all atoms from `Atoms`; and $\neg\phi$ is an LTL formula if ϕ is one, etc. The connectives X, F, G, U, R, and W are called *temporal connectives*. X means 'neXt state,' F means 'some Future state,' and G means 'all future states (Globally).' The next three, U, R and W are called 'Until,' 'Release' and 'Weak-until' respectively. We will look at the precise meaning of all these connectives in the next section; for now, we concentrate on their syntax.

Here are some examples of LTL formulas:

- $(((\mathrm{F}\,p) \wedge (\mathrm{G}\,q)) \to (p\ \mathrm{W}\ r))$
- $(\mathrm{F}\,(p \to (\mathrm{G}\,r)) \vee ((\neg q)\ \mathrm{U}\ p))$, the parse tree of this formula is illustrated in Figure 3.1.
- $(p\ \mathrm{W}\ (q\ \mathrm{W}\ r))$
- $((\mathrm{G}\,(\mathrm{F}\,p)) \to (\mathrm{F}\,(q \vee s)))$.

It's boring to write all those brackets, and makes the formulas hard to read. Many of them can be omitted without introducing ambiguities; for example, $(p \to (\mathrm{F}\,q))$ could be written $p \to \mathrm{F}\,q$ without ambiguity. Others, however, are required to resolve ambiguities. In order to omit some of those, we assume similar binding priorities for the LTL connectives to those we assumed for propositional and predicate logic.

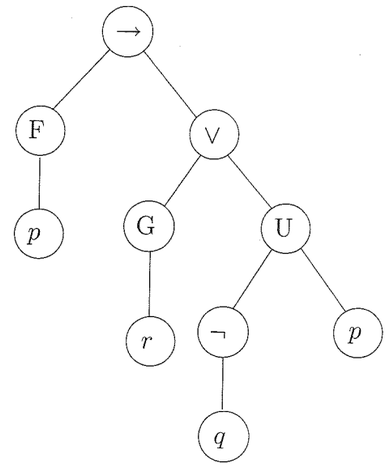

Figure 3.2. The parse tree of $\mathrm{F}\,p \to \mathrm{G}\,r \vee \neg q\ \mathrm{U}\ p$, assuming binding priorities of Convention 3.2.

Convention 3.2 The unary connectives (consisting of $\neg$ and the temporal connectives X, F and G) bind most tightly. Next in the order come U, R and W; then come $\wedge$ and $\vee$; and after that comes $\to$.

These binding priorities allow us to drop some brackets without introducing ambiguity. The examples above can be written:

- $\mathrm{F}\,p \wedge \mathrm{G}\,q \to p\ \mathrm{W}\ r$
- $\mathrm{F}\,(p \to \mathrm{G}\,r) \vee \neg q\ \mathrm{U}\ p$
- $p\ \mathrm{W}\ (q\ \mathrm{W}\ r)$
- $\mathrm{G}\,\mathrm{F}\,p \to \mathrm{F}\,(q \vee s)$.

The brackets we retained were in order to override the priorities of Convention 3.2, or to disambiguate cases which the convention does not resolve. For example, with no brackets at all, the second formula would become $\mathrm{F}\,p \to \mathrm{G}\,r \vee \neg q\ \mathrm{U}\ p$, corresponding to the parse tree of Figure 3.2, which is quite different.

The following are *not* well-formed formulas:

- $\mathrm{U}\,r$ – since U is binary, not unary
- $p\ \mathrm{G}\ q$ – since G is unary, not binary.

Definition 3.3 A subformula of an LTL formula ϕ is any formula ψ whose parse tree is a subtree of ϕ's parse tree.

The subformulas of p W $(q$ U $r)$, e.g., are p, q, r, q U r and p W $(q$ U $r)$.

3.2.2 Semantics of LTL

The kinds of systems we are interested in verifying using LTL may be modelled as transition systems. A transition system models a system by means of *states* (static structure) and *transitions* (dynamic structure). More formally:

Definition 3.4 A transition system $\mathcal{M} = (S, \rightarrow, L)$ is a set of states S endowed with a transition relation $\rightarrow$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$, and a labelling function $L\colon S \rightarrow \mathcal{P}(\texttt{Atoms})$.

Transition systems are also simply called *models* in this chapter. So a model has a collection of states S, a relation $\rightarrow$, saying how the system can move from state to state, and, associated with each state s, one has the set of atomic propositions $L(s)$ which are true at that particular state. We write $\mathcal{P}(\texttt{Atoms})$ for the power set of `Atoms`, a collection of atomic descriptions. For example, the power set of $\{p, q\}$ is $\{\emptyset, \{p\}, \{q\}, \{p, q\}\}$. A good way of thinking about L is that it is just an assignment of truth values to all the propositional atoms, as it was the case for propositional logic (we called that a *valuation*). The difference now is that we have *more than one state*, so this assignment depends on which state s the system is in: $L(s)$ contains all atoms which are true in state s.

We may conveniently express all the information about a (finite) transition system $\mathcal{M}$ using directed graphs whose nodes (which we call states) contain all propositional atoms that are true in that state. For example, if our system has only three states s_0, s_1 and s_2; if the only possible transitions between states are $s_0 \rightarrow s_1$, $s_0 \rightarrow s_2$, $s_1 \rightarrow s_0$, $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_2$; and if $L(s_0) = \{p, q\}$, $L(s_1) = \{q, r\}$ and $L(s_2) = \{r\}$, then we can condense all this information into Figure 3.3. We prefer to present models by means of such pictures whenever that is feasible.

The requirement in Definition 3.4 that for every $s \in S$ there is at least one $s' \in S$ such that $s \rightarrow s'$ means that no state of the system can 'deadlock.' This is a technical convenience, and in fact it does not represent any real restriction on the systems we can model. If a system did deadlock, we could always add an extra state s_d representing deadlock, together with new

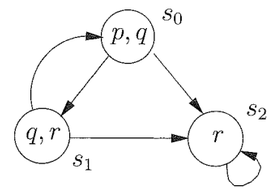

Figure 3.3. A concise representation of a transition system $\mathcal{M} = (S, \rightarrow, L)$ as a directed graph. We label state s with l iff $l \in L(s)$.

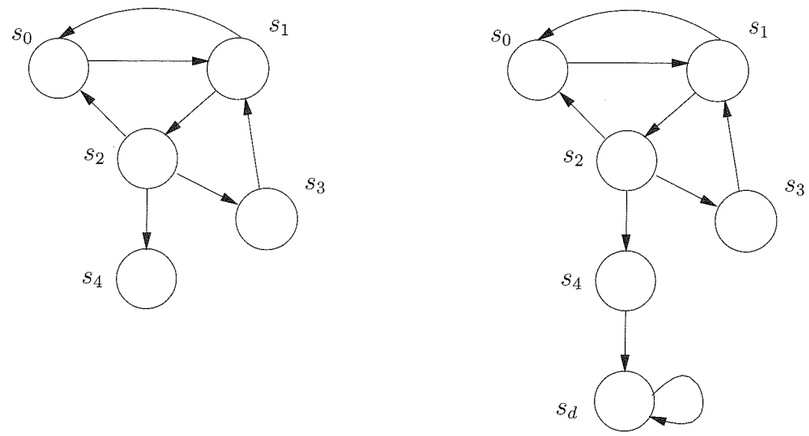

Figure 3.4. On the left, we have a system with a state s_4 that does not have any further transitions. On the right, we expand that system with a 'deadlock' state s_d such that no state can deadlock; of course, it is then our understanding that reaching the 'deadlock' state s_d corresponds to deadlock in the original system.

transitions $s \rightarrow s_d$ for each s which was a deadlock in the old system, as well as $s_d \rightarrow s_d$. See Figure 3.4 for such an example.

Definition 3.5 A path in a model $\mathcal{M} = (S, \rightarrow, L)$ is an infinite sequence of states $s_1, s_2, s_3, \ldots$ in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$. We write the path as $s_1 \rightarrow s_2 \rightarrow \ldots$.

Consider the path $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$. It represents a possible future of our system: first it is in state s_1, then it is in state s_2, and so on. We write π^i for the suffix starting at s_i, e.g., π^3 is $s_3 \rightarrow s_4 \rightarrow \ldots$.