Chapter 5

Model checking, verification of CTL

One must verify or expel ... doubts, and convert them into the certainty of YES or NO. [Thomas Carlyle]

Concepts introduced: LTL, CTL, linear and branching time, Kripke models, control state reachability, liveness analysis, model checking, ROBDDs.

THE BEHAVIOUR of a transition system is its set of runs, or its set of computations. To verify behaviours, we can consider questions like: "*Does every computation (run) of the transition system have a desired property* `X` *?*" or "*Is it true that in no computation,* `C` *is immediately followed by* `on-ac`*?*".

5.1 The verification setting

Properties may involve *reachability* of states. For example, is there a run leading to *deadlock*? A deadlocked system can do no more computation, more formally:

Definition 29 *The run* $s_0 \stackrel{*}{\Longrightarrow} s_k$ *with* $s_0 \in S_{\text{in}}$ *is in* ***deadlock*** *if no action is enabled at* s_k.

We would like to pose more sophisticated questions (other than reachability questions). For example, we may have *qualitative* questions like:

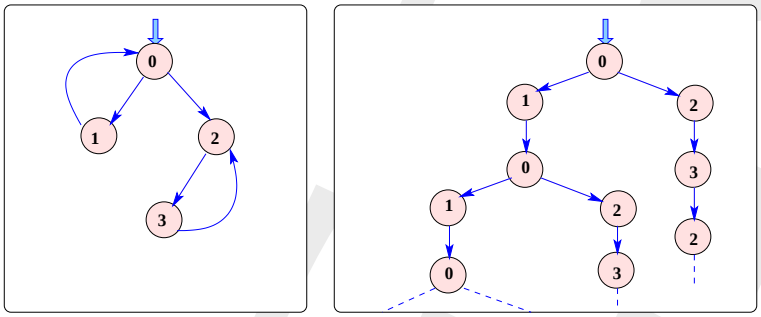
- ❖ Every request is eventually served.
- ❖ The sensor signal x11 is sensed infinitely often.
- ❖ From any stage of the computation the all clear state can be reached within 3 steps

We may also be interested in *quantitative* questions like:

- ❖ Every request is served within 3 microseconds.
- ❖ The sensor signal x11 is sensed every 10 milliseconds for ever.
- ❖ From any stage of the computation the all clear state can be reached within 1 second.

An extensional logic is one in which the truth value of a complex formula can be determined by the truth values of its components. By contrast, modal or intensional logics deal with sentences qualified by modal operators such as could or eventually or must and so on. If the modal operators refer to time, then we call these logics *temporal* logics.

Consider an assertion like “The engine is too hot." The *meaning* is clear, it does not vary with time, but the truth value of the assertion *can* vary in time. Sometimes it is true, and sometimes it is false, and it is never true and false simultaneously. Temporal logics are a good mechanism for expressing *qualitative* temporal properties of reactive systems. The basic modal operators are $\Box$ which represents necessity and its dual $\Diamond$ which represents possibility ($\Diamond A = \neg\Box\neg A$). The language of modal logic consists of propositional variables, a set of Boolean connectives such as $\{\wedge, \vee, \neg\}$, and the modal operators.



(a) Transition system (b) Branching time view

Figure 5.1: Notions of computation

Consider the transition system in Figure 5.1(a). A *linear* time view of the computations induced by this system is just the set of all runs:

$$\{\,0\,1\,0\,1\,0\,1\,0\,1\ldots,\;\; 0\,1\,0\,2\,3\,2\,3\,2\ldots,\;\; \ldots\,\}$$

A branching time view of the same system is shown in Figure 5.1(b), where the temporal order forms a tree which branches into the future. LTL (Linear Temporal Logic) is a modal linear-time temporal logic, built up from a set of proposition variables $p_1, p_2, \ldots$, logic connectives $\neg, \vee, \wedge, \rightarrow$ and the temporal operators:

- $\mathrm{X}\,\phi$ indicating that ϕ must hold in the next state.
- $\mathrm{G}\,\phi$ (or $\Box\,\phi$) indicating that ϕ must hold in all the following states.
- $\mathrm{F}\,\phi$ (or $\Diamond\,\phi$) indicating that eventually (finally) ϕ must hold somewhere.
- $\phi\,\mathrm{U}\,\psi$ indicating that ϕ has to hold until ψ holds at the current or a future position.
- $\phi\,\mathrm{R}\,\psi$ The dual of U, ψ holds until the first state where ϕ holds.

In LTL, one can encode formulæ about events along a single computation path. By contrast, CTL (Computation Tree Logic) is a modal *branching-time* temporal logic. The operators quantify over all possible future paths from a given state.

CTL and LTL are both subsets of a more general temporal logic CTL*. There are expressions in CTL that cannot be expressed in LTL and vice versa.

As an example, consider $\mathrm{A}(\mathrm{FG}\,p)$, which is an LTL formula representing the idea that: for all paths, eventually p holds globally (i.e. from then on). $\mathrm{AF}(\mathrm{AG}\,p)$ is a

CTL formula representing the idea that: for all paths, eventually you get to a state where for all paths p holds globally (i.e. from then on). This CTL formula does not represent the same thing as the original LTL formula, even though at first sight it seems to be similar.

A counter-example is given below in Figure 5.2.

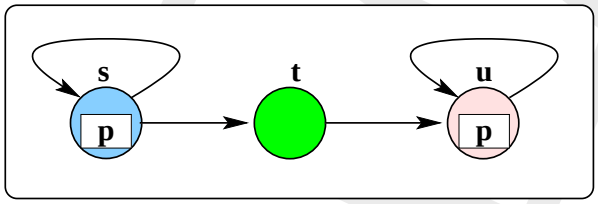


Figure 5.2: Counter example

It is fairly easy to see that in the LTL linear time view, all runs that start in state s have p eventually holding globally. In a linear time view, the possible (infinite) runs from s are:

$$\left.\begin{array}{l} sssssssss\ldots \end{array}\right\} \quad i.e.\ \ s^{\infty}$$

or...

$$\left.\begin{array}{l} stuuuuuu\ldots \\ sstuuuuu\ldots \\ ssstuuuu\ldots \\ \ldots \end{array}\right\} \quad i.e.\ \ ss^{*}tu^{\infty}$$

where s^{∞} means an infinite run of state s, and s^{*} means a finite run of s.

In both state s and state u, the property p holds, so in the linear time view, for state s, $\mathrm{A}(\mathrm{FG}\,p)$.

However, let us now consider the branching time view, and the CTL formula $\mathrm{AF}(\mathrm{AG}\,p)$. Figure 5.3 has the unfolded branching time view.

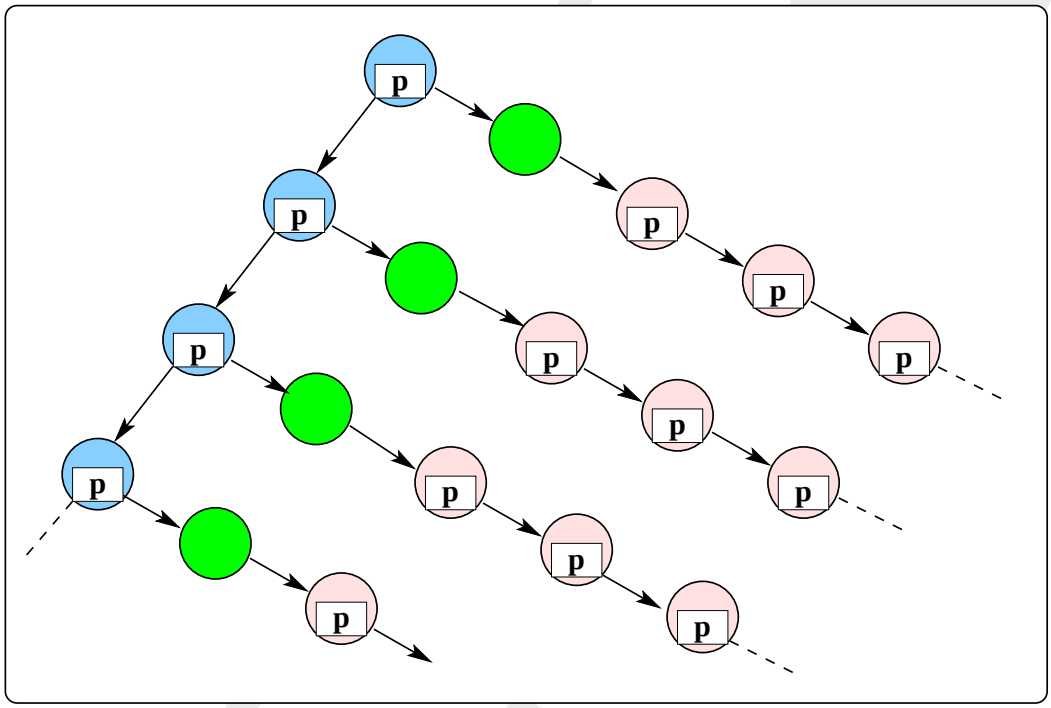


Figure 5.3: Unfolded counter example

The CTL formula $\mathrm{AF}(\mathrm{AG}\, p)$ represents the idea that: for all paths, eventually you get to a state where for all paths p holds globally (i.e. from then on). If you traverse down the left hand sequence in the figure, it does not matter how far you go, you will never get to a state where for all paths p holds globally.

In other words, this does not represent the same assertion as the LTL one.

CTL formulæ are differentiated from LTL formulæ, because each of the temporal operators must be preceded by a *path* quantifier: A (for all paths), and E (there exists a path). Since CTL expressions require every temporal operator to be preceded by a quantifier, and since there are five temporal operators, and two quantifiers, we have ten base expression types.

The ten base expression types may be expressed in terms of just three expressions:

- ❖ EX p : For one computation path, property p holds in the next state;
- ❖ A$(p \,\mathrm{U}\, q)$: For all computation paths, property p holds until q holds.
- ❖ E$(p \,\mathrm{U}\, q)$: For one computation path, property p holds until q holds.

We focus here on CTL specifications. The *model-checking* verification setting may be visualized as in Figure 5.4, where we extract a (state transition) model TS from a system.

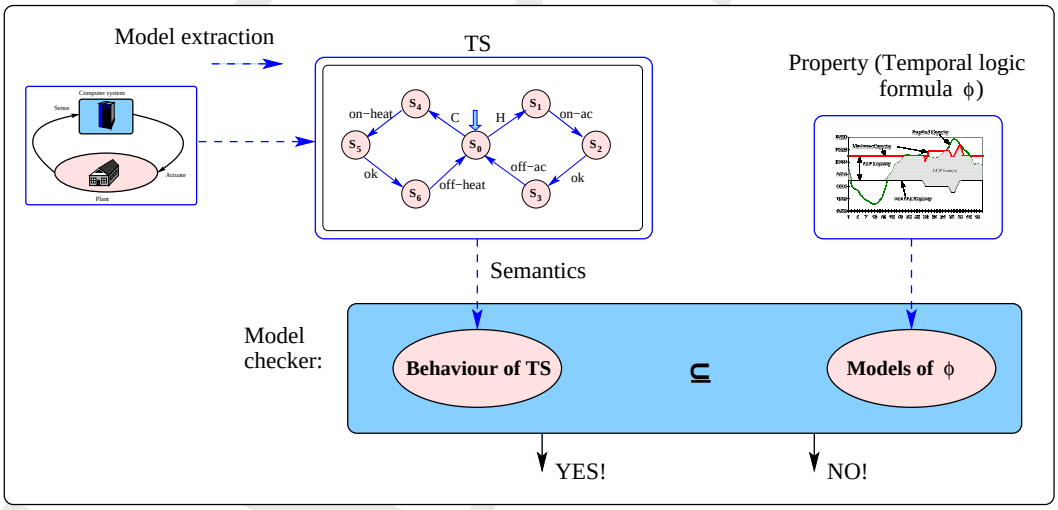
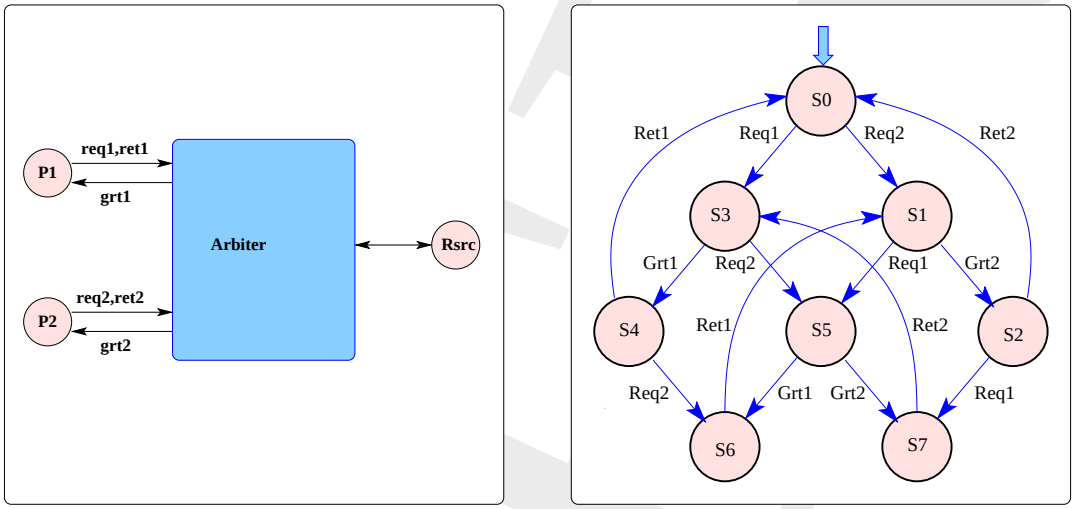


Figure 5.4: Verification setting

The meaning that we attach to TS is that it correctly represents the behaviour of the system, expressed as the allowable set of runs (or computations) of the system. A model-checker checks if this *behaviour* of the system is a subset of the set of runs (or computations) induced by an arbitrary property ϕ, returning YES or NO.



(a) Resource arbiter

(b) Arbiter transition system

Figure 5.5: Arbiter for an operating system

Consider the following scenario: We have an arbiter in an operating system, which is controlling access by two processes to a single resource, only allowing one at a time to gain access to the resource. Each process can request access to the resource, by (perhaps) making a `req()` call. When the resource is free, the arbiter can grant access by signalling the process using the `grt()` signal. When a process no longer needs the resource, it signals the arbiter using the `ret()` signal. In Figure 5.5(a) we see the arbiter, with Figure 5.5(b) showing a transition system for its correct operation.

When modelling this system, it is important to identify suitable atomic propositions relevant to the system. Suitable propositions might be:

- i_1, i_2 : Processes 1 and 2 are *idle*. In the starting state both processes are idle.
- w_1, w_2 : Processes 1 and 2 are *waiting* for the resource.
- u_1, u_2 : Processes 1 and 2 are *using* the resource.

5.2 Foundations for CTL model checking

The $\Box$ operator cannot be formalized with an extensional semantics. The Kripke semantics is a formal semantics for modal logic systems.

Definition 30 *A Kripke structure $\mathcal{K}$ over a set AP of atomic propositions is a 4-tuple $(S, \Delta, AP, \mathcal{L})$, where*

1. *S is a finite set of **states***
2. *$\Delta \subseteq S \times S$ is a **transition relation** that must be* total
3. *AP is a finite set of **atomic propositions***
4. *$\mathcal{L} : S \rightarrow 2^{AP}$ is a function which labels each state with the set of atomic propositions true in that state*

Consider the transition system shown in Figure 5.5(b). The atomic propositions represent the system's view of the state of the two tasks. They are idle, waiting or using the *resource*, and we have the following set of atomic propositions:

$$AP = \{i_1, w_1, u_1, i_2, w_2, u_2\}$$

We can convert the transition system into a Kripke structure, by writing out $\mathcal{L}(s)$ for each state s. The labelling function $\mathcal{L} : S \rightarrow 2^{\mathrm{AP}}$ for each state returns the atomic propositions that are valid in each state. A suitable function is:

$$\begin{aligned}\mathcal{L} \ = \{ \ & (s_0, \{i_1, i_2\}),\\ & (s_1, \{i_1, w_2\}),\\ & (s_2, \{i_1, u_2\}),\\ & (s_3, \{w_1, i_2\}),\\ & (s_4, \{u_1, i_2\}),\\ & (s_5, \{w_1, w_2\}),\\ & (s_6, \{u_1, w_2\}),\\ & (s_7, \{w_1, u_2\}) \ \}\end{aligned}$$

5.2.1 The unfolding of $\mathcal{K}$

When investigating a Kripke structure $\mathcal{K}$, it is often easier to visualize the unfolded version $\mathrm{UF}(\mathcal{K})$ of the structure.

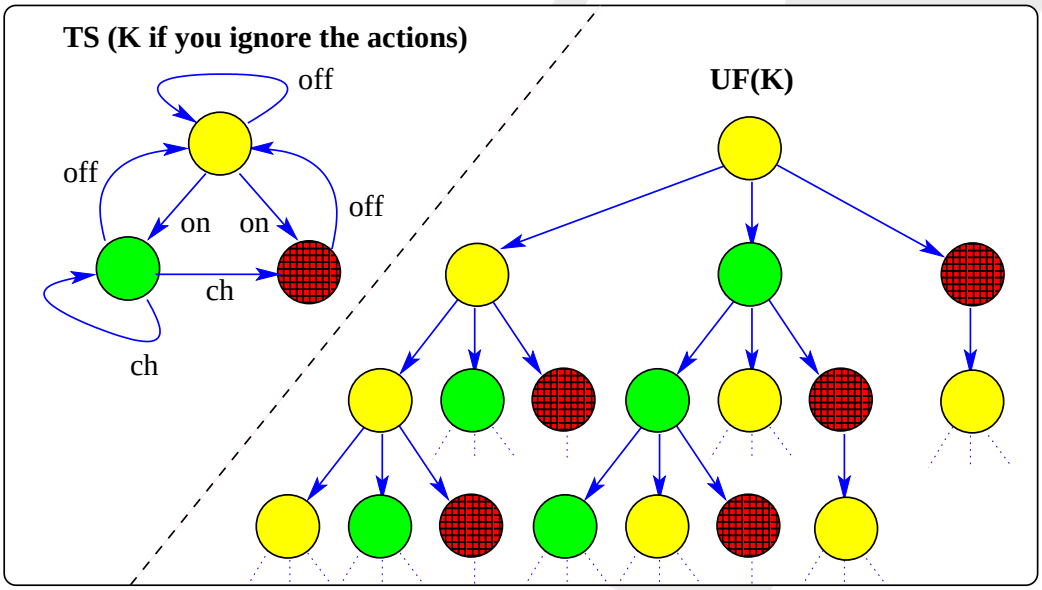


Figure 5.6: Unfolding of the TV Kripke structure

In Figure 5.6 the Kripke structure $\mathcal{K}$ represents a transition system for the *irritable* TV viewer. The viewer can switch the TV on and off, and also change channels. As soon as the irritable TV viewer switches to a TV channel that is not worth watching, the TV must be switched off.

$\mathrm{UF}(\mathcal{K})$ is another Kripke structure, equivalent in terms of the path structure of $\mathcal{K}$, but much easier to visualize possible runs, and to see when a run repeats.

Definition 31 *The **unfolding** of a Kripke structure $\mathcal{K}$, from an identified starting state s_0, is* $\mathrm{UF}(\mathcal{K}) = (S, \Delta, AP, \mathcal{L})$, *where*

1. $S = \{(s, \pi) \mid \pi \text{ is a path from } s_0 \text{ to } s \text{ in } \mathcal{K}$
2. $\Delta((s, \pi), (s', \pi'))$ iff $\Delta(s, s')$ in $\mathcal{K}$ and $\pi' = \pi s'$.
3. $\mathcal{L}(s, \pi) = \mathcal{L}(s)$

5.2.2 CTL and CTL-

We define a fragment of CTL, called CTL-, which we use to specify the formulæ defining the properties to be checked against the Kripke structure. When used in this context, such a Kripke structure is often called a CTL-model (or just a *model*), and we will refer to such models as M. The CTL- fragment is sufficient to encode any (traditional) CTL formula.

Definition 32 *Given a proposition $p \in AP$ (a finite set of atomic propositions), then p is a CTL- formula, and if ψ_1 and ψ_2 are CTL- formulæ, then*

1. $\neg\psi_1$ *is a CTL- formula*
2. $\psi_1 \wedge \psi_2$ *is a CTL- formula*
3. $\psi_1 \vee \psi_2$ *is a CTL- formula*
4. $\mathrm{EX}(\psi_1)$ *is a CTL- formula*
5. $\mathrm{A}(\psi_1 \,\mathrm{U}\, \psi_2)$ *is a CTL- formula*
6. $\mathrm{E}(\psi_1 \,\mathrm{U}\, \psi_2)$ *is a CTL- formula*

Formulæ with this syntax can be viewed as a tree. For example, the formula

$$\mathrm{EX}(p \vee \mathrm{EX}(\mathrm{E}(p \,\mathrm{U}\, r)))$$

could be represented by the tree in Figure 5.7.

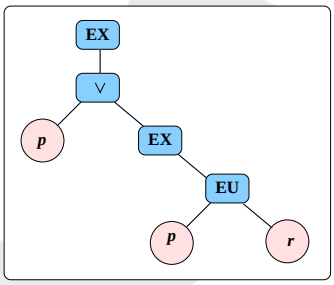


Figure 5.7: A CTL- formula shown as a tree

5.2.3 Model checking

Model checking is commonly expressed as a ternary relation ($\models$):

$$M, s \models P$$

The relation is true when the property P holds in state s for a given model M. It is normally defined inductively, with a set of interlocking rules. A labelling algorithm may then be used to establish the set of states satisfying the relation.

There are two kinds of properties that are of interest:

1. **safety:** nothing bad will ever happen
2. **liveness:** eventually something good will happen

(One is like "*there exists*" and the other like "*for all*"). The techniques for assessing safety and liveness are different. For example, in model checking we use Büchi-automata to express and check liveness, and state-space enumeration for safety. Safety is about reachability, but with liveness we must consider infinite non-terminating execution, checking for the reachability of certain (control) states, and so an overall goal is to solve a control state reachability problem.

Let us return to the *arbiter* example in Figure 5.5(b), and let M be the resulting CTL-model. We can determine whether or not $M, s_0 \models \psi$ for the following example properties: $\psi_1 = \text{EF}(u_1 \wedge u_2)$, $\psi_2 = \text{AF}(u_1 \vee u_2)$ and $\psi_3 = \text{AG}(u_1 \vee u_2)$.

Given the Kripke structure, it is fairly easy to see that:

1. $M, s_0 \not\models \text{EF}(u_1 \wedge u_2)$ as there is no path from s_0 leading to a state in which both u_1 and u_2 are true. (In fact there is no state in which both u_1 and u_2 are true).
2. $M, s_0 \models \text{AF}(u_1 \vee u_2)$ as every path from s_0 must lead to one of s_4, s_6, s_2 or s_7, and each of these has either u_1 or u_2 true.
3. $M, s_0 \not\models \text{AG}(u_1 \vee u_2)$ as $M, s_0 \not\models u_1$ and $M, s_0 \not\models u_2$.

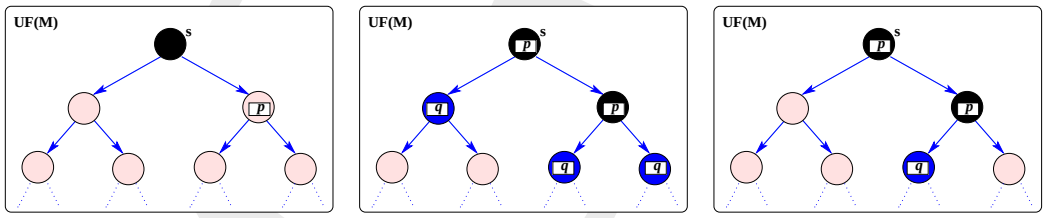
5.2.4 The definition of $\models$

The model checking relation $\models$ (In LaTeX, `\models`) can be read various ways. In $M, s \models \psi$ we might say that ψ *holds* or is *satisfied* at state s.

A path from one state s is a sequence of states $\pi = s_0\ s_1\ \ldots$ such that $s = s_0$ and $\Delta(s_i, s_{i+1})$ for every i. The i-th element of π is $\pi(i)$. The model checking relation is defined for each atomic proposition p and each CTL- formula ψ_1, ψ_2 as:

$M, s \models p$	$\Leftrightarrow$	$p \in \mathcal{L}(s)$
$M, s \models \neg\psi_1$	$\Leftrightarrow$	iff it is not the case that $M, s \models \psi_1$
$M, s \models \psi_1 \wedge \psi_2$	$\Leftrightarrow$	iff $M, s \models \psi_1$ and $M, s \models \psi_2$
$M, s \models \psi_1 \vee \psi_2$	$\Leftrightarrow$	iff $M, s \models \psi_1$ or $M, s \models \psi_2$
$M, s \models \mathrm{EX}(\psi_1)$	$\Leftrightarrow$	iff $\Delta(s, s')$ and $M, s' \models \psi_1$ (i.e. s has a successor state at which ψ_1 holds)
$M, s \models \mathrm{A}(\psi_1 \,\mathrm{U}\, \psi_2)$	$\Leftrightarrow$	iff for every path $\pi = s_0\ s_1\ \ldots$ from s, for some j, $M, \pi(j) \models \psi_2$, and $\forall i < j\ M, \pi(i) \models \psi_1$
$M, s \models \mathrm{E}(\psi_1 \,\mathrm{U}\, \psi_2)$	$\Leftrightarrow$	iff there is a path $\pi = s_0\ s_1\ \ldots$ from s, where for some j, $M, \pi(j) \models \psi_2$, and $\forall i < j\ M, \pi(i) \models \psi_1$

Figure 5.8 shows each of the temporal CTL- operators in an unfolded Kripke structure $\mathrm{UF}(M)$.



(a) $M, s \models \mathrm{EX}(p)$ (b) $M, s \models \mathrm{A}(p \,\mathrm{U}\, q)$ (c) $M, s \models \mathrm{E}(p \,\mathrm{U}\, q)$

Figure 5.8: EX, AU and EU in the unfolded Kripke structure

The table clearly defines the model checking relation for EX, AU and EU in CTL-, but not for the other CTL operators AX, AG and so on. The following list shows some of the other operators that can be derived in terms of the three just given.

1. $\mathrm{AX}(\psi) = \neg\mathrm{EX}(\neg\psi)$ For every next state ψ holds. It is not the case that there exists a next state at which ψ does not hold.
2. $\mathrm{EG}(\psi) = \neg\mathrm{A}(\mathrm{true}\,\mathrm{U}\,\neg\psi)$ There exists a path π from s such that for every $k \geq 0$: $M, \pi(k) \models \psi$. It is not the case that ...

5.3 Model checking algorithm for CTL

The label field in the Kripke structure gives us a mechanism for calculating when a formula ψ satisfies the associated model M in a particular state s: $M, s \models \psi$. The model checking process has two steps, visualized in Figure 5.9.

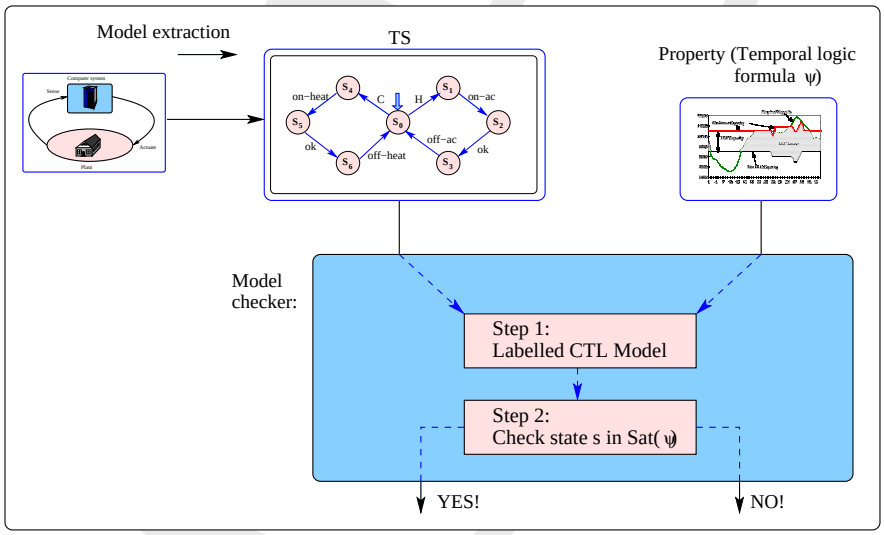


Figure 5.9: Model checking CTL models

Firstly, we label each state with the atomic propositions that are true in each state.

Secondly, we calculate the sets of states that satisfy the property, and if our state s is in this set, we can return YES, otherwise NO. We show an algorithm for calculating the satisfaction set for a model, given a formula. This can be calculated using a recursive function (corresponding to a recursive traversal of the structure), which returns the set of states that satisfy a given formula:

```
set_of_States sat(Property ψ) =
  if ψ ∈ AP then {s | ψ ∈ L(s)}
  else case ψ of
    true:        S
    false:       ∅
    ¬ψ:          S − sat(ψ)
    ψ1 ∧ ψ2:     sat(ψ1) ∩ sat(ψ2)
    ψ1 ∨ ψ2:     sat(ψ1) ∪ sat(ψ2)
    EX(ψ1):      {s ∈ S | s' ∈ s↑ ∧ s' ∈ sat(ψ1)}
    A(ψ1 U ψ2):  lfp(g(Z) = sat(ψ2) ∪ (sat(ψ1) ∩ {s ∈ S | ∀s' ∈ s↑ ∩ Z}))
    E(ψ1 U ψ2):  lfp(h(Z) = sat(ψ2) ∪ (sat(ψ1) ∩ {s ∈ S | ∃s' ∈ s↑ ∩ Z}));
```

where $s^{\uparrow}$ represents the set of successor states of s.

The last two lines in the function express the idea that we can calculate the sets of states for $\mathrm{A}(\psi_1 \,\mathrm{U}\, \psi_2)$ and $\mathrm{E}(\psi_1 \,\mathrm{U}\, \psi_2)$, by taking the least fix-point (and hence the function name `lfp`) of functions g and h (sometimes expressed as the algorithms $\mathrm{sat}_{\mathrm{AU}}$ and $\mathrm{sat}_{\mathrm{EU}}$). What are the functions g and h? Some investigation will show that

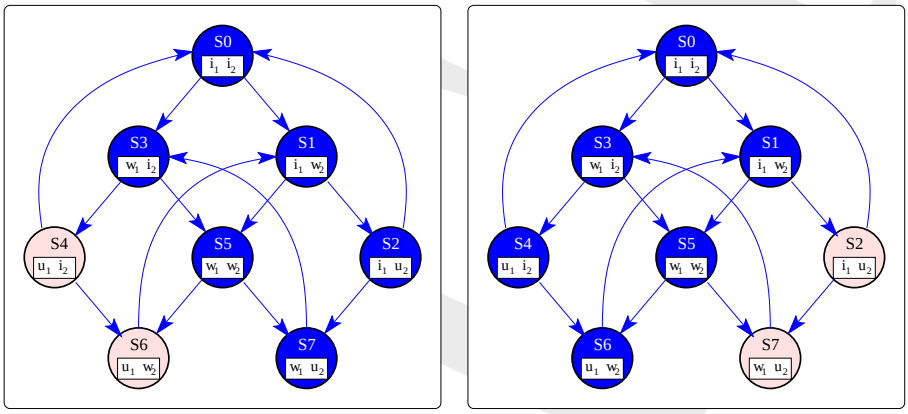
$$\begin{aligned} \mathrm{A}(\psi_1 \,\mathrm{U}\, \psi_2) &= \psi_2 \vee (\psi_1 \wedge \mathrm{AX}(\mathrm{A}(\psi_1 \,\mathrm{U}\, \psi_2))), \text{ and} \\ \mathrm{E}(\psi_1 \,\mathrm{U}\, \psi_2) &= \psi_2 \vee (\psi_1 \wedge \mathrm{EX}(\mathrm{E}(\psi_1 \,\mathrm{U}\, \psi_2))) \end{aligned}$$

and it seems appropriate to express these as fix-points of the corresponding functions

$$\begin{aligned} g(Z) &= \psi_2 \vee (\psi_1 \wedge \mathrm{AX}(Z)), \text{ and} \\ h(Z) &= \psi_2 \vee (\psi_1 \wedge \mathrm{EX}(Z)) \end{aligned}$$

We can view this as a labelling procedure for $M, s \models \psi$ where we build up extra labels for each state in the model, progressively uncovering the subformulas of ψ.

In Figure 5.10, we see the labelled arbiter model, and the states that satisfy $\mathrm{EX}(w_1)$ and $\mathrm{E}(i_2 \,\mathrm{U}\, w_2)$.



(a) $\mathrm{EX}(w_1)$ (b) $\mathrm{E}(i_2 \,\mathrm{U}\, w_2)$

Figure 5.10: Arbiter model checking

5.3.1 Example and optimizations

The model checking approach to verification [CGP99] is to abstract out key elements of the software and to verify just these elements. For efficiency, various optimizations can be made if the elements reduce to binary values. Given the underlying reliance on binary abstractions, it is no surprise that model checking is being used in the analysis of digital electronic circuits, but it has also proved effective in the software domain, in the areas of protocol analysis, the behaviour of reactive systems, and for checking concurrent systems.

It is difficult to find examples convincing enough to demonstrate a technique, but which are small enough to fully describe in a short chapter. We choose to use as an example a simple mutual exclusion protocol in which two processes, P_1 and P_2 share six (boolean) variables, and co-operate to ensure mutually exclusive access to a critical section of code. A third process T_1 monitors the variables and changes a turn variable. The entire system is the parallel composition of these three processes, and is continuous (indicated by the trailing recursive call). Each line of code is considered to be atomic, and we use 1 to represent true, 0 to represent false.

```
P1  = if idle1 then (wait1 := 1; idle1 := 0) else
      if wait1 ∧ idle2 then (active1 := 1; wait1 := 0) else
      if wait1 ∧ wait2 ∧ ¬turn then (active1 := 1; wait1 := 0);
      if active1 then (CritSect; idle1 := 1; active1 := 0);
P2  = if idle2 then (wait2 := 1; idle2 := 0) else
      if wait2 ∧ idle1 then (active2 := 1; wait2 := 0) else
      if wait2 ∧ wait1 ∧ turn then (active2 := 1; wait2 := 0);
      if active2 then (CritSect; idle2 := 1; active2 := 0);
T1  = if idle1 ∧ wait2 then turn := 1 else
      if idle2 ∧ wait1 then turn := 0;
System = (P1 || P2 || T1); System;
```

The specific protocol is chosen because it only uses boolean variables, simplifying and shortening this presentation. We can represent a state s_i of this system as a valuation of the relevant variables. The states for this system are listed in Table 5.1.

State $\mathcal{S}$	**P_1 vars**			**P_2 vars**			**T_1 vars**	**Next state(s)**
	idle_1	wait_1	active_1	idle_2	wait_2	active_2	turn	
s_0	1	0	0	1	0	0	0	s_1, s_2
s_1	0	1	0	1	0	0	0	s_3, s_5
s_2	1	0	0	0	1	0	0	s_4, s_5, s_6
s_3	0	0	1	1	0	0	0	s_0, s_7
s_4	1	0	0	0	0	1	0	s_0, s_8
s_5	0	1	0	0	1	0	0	s_7
s_6	1	0	0	0	1	0	1	s_9, s_{10}
s_7	0	0	1	0	1	0	0	s_2
s_8	0	1	0	0	0	1	0	s_1
s_9	0	1	0	0	1	0	1	s_{11}
s_{10}	1	0	0	0	0	1	1	s_{11}, s_{15}
s_{11}	0	1	0	0	0	1	1	s_{12}
s_{12}	0	1	0	1	0	0	1	s_1, s_9, s_{13}
s_{13}	0	0	1	1	0	0	1	s_{14}, s_{15}
s_{14}	0	0	1	0	1	0	1	s_6
s_{15}	1	0	0	1	0	0	1	s_6, s_{12}

Table 5.1: States for the complete system

We may also characterize this system using a transition diagram as in Figure 5.11, where the labels on the transitions relate to the line-numbers of the program.

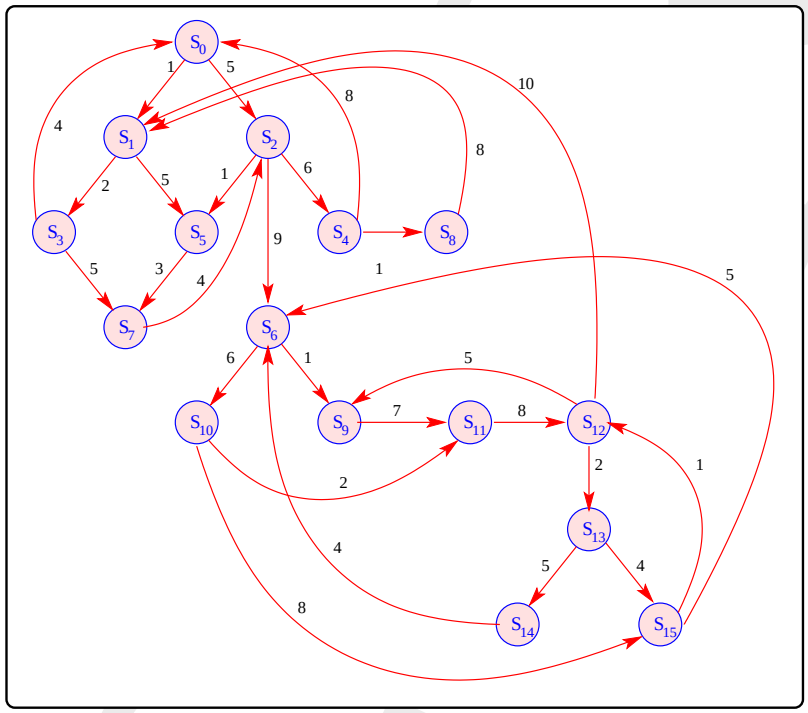


Figure 5.11: State transition diagram

The transition relation may be compactly expressed (using short names for the variables) as the following predicate t_1:

$$\begin{aligned} t_1 \quad = \quad & (i_1 \wedge w_1' \wedge \overline{i_1'}) \quad \vee \quad (w_1 \wedge i_2 \wedge a_1' \wedge \overline{w_1'}) \quad \vee \quad (w_1 \wedge w_2 \wedge \overline{t} \wedge a_1' \wedge \overline{w_1'}) \quad \vee \quad (a_1 \wedge i_1' \wedge \overline{a_1'}) \\ \vee \quad & (i_2 \wedge w_2' \wedge \overline{i_2'}) \quad \vee \quad (w_2 \wedge i_1 \wedge a_2' \wedge \overline{w_2'}) \quad \vee \quad (w_2 \wedge w_1 \wedge t \wedge a_2' \wedge \overline{w_2'}) \quad \vee \quad (a_2 \wedge i_2' \wedge \overline{a_2'}) \\ \vee \quad & (i_1 \wedge w_2 \wedge t') \quad \vee \quad (i_2 \wedge w_1 \wedge \overline{t'}) \end{aligned}$$

Note that the predicate has been ordered to clearly show a correlation with the program. The first line corresponds to P_1, the second to P_2 and the third to T_1.

5.3.2 BDD representation of transition relation

A predicate such as just seen may also be encoded as an ordered binary decision tree (OBDT), in which the levels denote the different variables, and paths through the tree represent valuations of the transition relation. In Figure 5.12 we have an OBDT for the expression $(i_1 \wedge i_2) \vee (i_3 \wedge \overline{i_4})$.

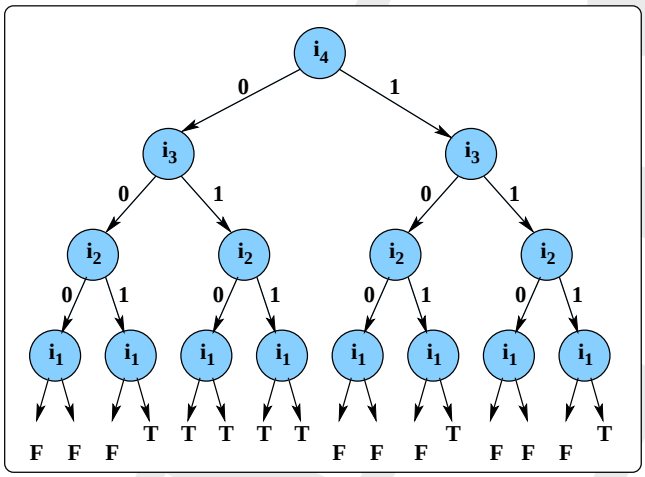


Figure 5.12: OBDT for $(i_1 \wedge i_2) \vee (i_3 \wedge \overline{i_4})$

Note that if we reorder the variables, we get a different decision tree, but this new tree still represents the predicate. In other words, it is independent of the order of the variables. The OBDT does not scale well, but there are optimizations that may be done. An optimization to exploit repetition on OBDTs leads to reduced ordered binary decision diagrams (ROBDDs).

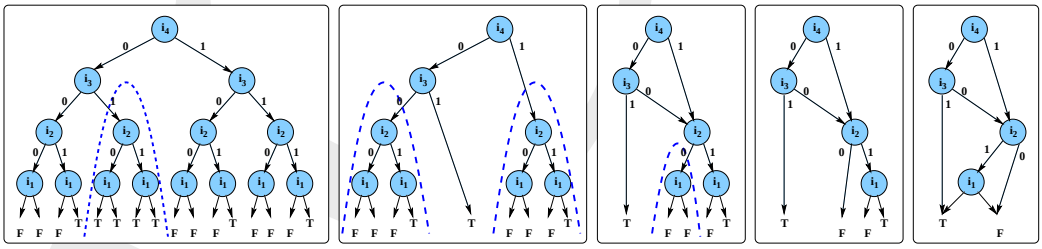


Figure 5.13: ROBDD reduction for $(i_1 \wedge i_2) \vee (i_3 \wedge \overline{i_4})$

ROBDDs provide a canonical form for the OBDTs, but more significantly, similar sub-trees of a OBDT result in the ROBDD merging the two subtrees.

Bryant [Bry86] introduced these data structures, showing how such representations of functions may be manipulated efficiently. In the paper, fast algorithms for common boolean operations are described, with complexities proportional to the sizes of the graphs.

The ROBDD optimization for the purpose of model checking was first identified by McMillan [McM93], and resulted in significant improvements in the number of states that could be model-checked. In problems which exhibit a pattern regularity,

the ROBDD representation can be many orders of magnitude smaller than an equivalent (OBDT or enumerated) representation. Modern branching time model checkers all use the ROBDD optimization.

5.4 Summary of topics

In this section, we introduced the following topics:

Verification setting. *Basic setting for the verifi cation problem.*
Theoretical foundations. *Formal foundations for CTL.*
Model checking algorithms. *Algorithms for checking systems.*
Example and optimizations. *Using BDDs for CTL model checking.*