Chapter 7

Temporal Logic

7.1 Introduction

The idea of temporal logic is to supply a vehicle that allows one to reason about system behaviour as it evolves in time. We are already familiar with traditional propositional logic as a means to reason about combinational circuits. We could roughly say that temporal logic is its counterpart for sequential machines.

In this chapter we first look at a general structure that defines a model for the subsequent temporal logic. This means that we use the structure to define the semantics of the various logic symbols. The temporal logic that we study in detail is known as *Computation Tree Logic* or CTL. It has been invented with the primary purpose to allow efficient testing of certain system properties. These properties are often classified as *liveness* properties and *safety* properties. Informally speaking, liveness properties express that something good will eventually happen; safety properties express that nothing bad will ever happen. For instance, in a practical situation where we have several processors competing to gain access to a shared bus, we could assert the typical safety property that at any time at most one processor gains access to the bus. A typical liveness property in this case would be to require that every processor eventually gets its turn. Problems of this sort can be solved by a verification method known as *model checking*: properties expressed in CTL are checked against a state model of the system. We will see more of this at the end of this chapter.

7.2 A Few Words About Time

As the name suggests, temporal logic has to do with time. The way we model time depends on our application. Throughout this chapter we will choose to consider time to be discrete; let time start at some initial timepoint denoted time 0; and assume an infinite future. This is convenient because we set out to study state-based models of digital systems. Many of such systems are synchronous, it is therefore natural to let our time model coincide with the ticking of the master system clock. But even in asynchronous systems or hybrid systems, it is often accurate enough to only analyse the system's state at discrete points in time.

7.3 Kripke Structures

A Kripke (or temporal) structure is a triple $M = (S, R, L)$ with

S : a (possibly infinite) set of states,

$R \subseteq S \times S$: a total binary relation, i.e., $\forall_{s \in S} \underset{t \in S}{\exists} (s, t) \in R$, and

L : a state labelling function $S \to 2^{AP}$.

The labelling function L is intended to associate with each state of S an interpretation of the *atomic propositions* AP, i.e., through L we know for each state which atomic propositions are assigned true and which are assigned false. The atomic propositions are meant to convey particular facts about the system under study and for now are left without any further interpretation. There are several alternative ways to express the above assignment:

$L : AP \to 2^S$, which gives for each atomic proposition the states it is assigned true.

$L : S \times AP \to \{\mathsf{false}, \mathsf{true}\}$, making L a Boolean function that evaluates to true when a certain atomic proposition is assigned true in a certain state, evaluating to false otherwise.

$L : S \to (AP \to \{\mathsf{false}, \mathsf{true}\})$, which makes $L(s)$ an interpretation function of an atomic proposition at state s.

In the sequel, we will use whichever definition is the most convenient.

A Kripke structure may be viewed as a labelled directed graph: the states are the graph's vertices and the relation R defines the edges. Note that because R is required to be total, each vertex in the graph must have at least one outgoing edge. To map this graph onto our model of time, we single out a certain vertex s_0

and announce that to be at timepoint 0. Its immediate successors will then be at timepoint 1, et cetera. This operation will effectively 'unwind' the relation R and cause the graph to be drawn as a tree (figure 7.1).

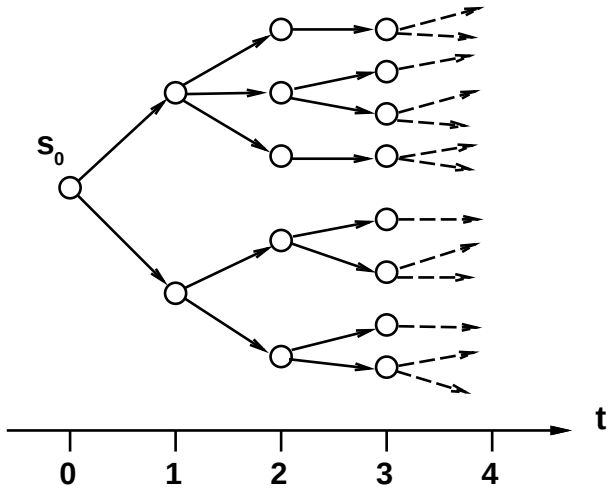

Figure 7.1. Kripke structure 'projected' on time line.

What use is a Kripke structure? Well, it can be regarded as a state model of a system: the atomic propositions labelling a vertex define the 'state' of the system at that vertex. The possible behaviours (or execution traces) of the system are paths through the graph. Note that a Kripke structure reflects a so-called branching-time model; each state corresponds to a point in time and branches (via its outgoing edges) to a number of possible futures. Note also that a Kripke structure is very similar to a *State Transition Graph* (or state diagram), however, in the former the vertices are labelled whereas in a state transition graph the edges are labelled.

7.4 Computation Tree Logic

Computation tree logic is a logic that is specifically tailored to reason about atomic propositions and their change of 'value' in time as laid down by a given Kripke structure. One also says that the Kripke structure is a model for CTL or that CTL formulas are interpreted over Kripke structures.

The syntax of the most general logic called CTL* consists of 2 groups of only 3 rules each:

S_1 Each Atomic Proposition is a *state* formula;

S_2 If f, g are *state* formulas, so are $\neg f$, $f \wedge g$;
S_3 If f is a *path* formula then $E f$, $A f$ are *state* formulas.

P_1 Each *state* formula is a *path* formula;
P_2 If f, g are *path* formulas, so are $\neg f$, $f \wedge g$;
P_3 If f, g are *path* formulas, so are $X f$, $f U g$.

What is known as CTL is a restricted form of the above logic. It consists of the same state formulas generated by the rules S_1, S_2, and S_3, but with the rules for the path formulas replaced by a single new rule:

P_0 If f, g are *state* formulas then $X f$, $f U g$ are *path* formulas.

Observe that the difference in CTL* and CTL syntax is that in the latter path formulas may no longer be nested; they require the use of an E or A operator to make a path formula into a state formula. It can be shown that CTL is strictly weaker in expressiveness than CTL*. So, there are properties that can be expressed as a CTL* formula but no equivalent CTL formula exists.

When no distinction is made between CTL* and CTL we will denote this by CTL$^{(*)}$. We define the language of CTL$^{(*)}$, i.e., the set of all formulas, to be all the *state* formulas generated by the above syntax rules. Hence, from now on when formulas are not explicitly qualified, state formulas are to be understood.

7.4.1 Semantics

Here we will define the semantics for both CTL* and CTL, although we will only be using the simpler logic CTL in the sequel. The meaning of a CTL$^{(*)}$ formula is defined with respect to a Kripke structure $M = (S, R, L)$ with designated initial state s_0. An infinite path in the graph of the Kripke structure will be called a *fullpath*, e.g., $x = (s_0, s_1, \cdots)$ denotes a fullpath starting at state s_0 followed by state s_1 and so on. We use the notation x^i to denote the suffix fullpath $(s_i, s_{i+1}, \cdots)$ of x, i.e., the fullpath x after deletion of a prefix of length i.

In figure 7.2 the semantics of a formula is inductively defined according the syntax rules. These semantic definitions should be read as follows. For a state formula f, "$M, s_0 \models f$ iff condition" means that the formula f holds in (or is satisfied by) the model M with initial state s_0 when the "condition" is met (= true). Of course, the condition may refer to the model. So the semantic rule S_1 says that an atomic proposition (which is itself a state formula) is satisfied by the model (M, s_0) when that atomic proposition is assigned true by the labelling of s_0. For a

path formula f, "$M, x \models f$ iff condition" means that the formula f holds for the fullpath x in the model M when the condition is true. So the semantic rule P_3 for the formula $p\,U\,q$ says that $p\,U\,q$ is satisfied by (M, x) when there exists a suffix fullpath x^i such that $M, x^i \models q$ holds and for all suffix fullpaths x^j, such that $0 \le j < i$, $M, x^j \models p$ holds. Clearly an inductive definition.

S_1	$M, s_0 \models p$	iff	$p \in L(s_0)$
S_2	$M, s_0 \models \neg f$	iff	not $M, s_0 \models f$
	$M, s_0 \models f \wedge g$	iff	$M, s_0 \models f$ and $M, s_0 \models g$
S_3	$M, s_0 \models E\,f$	iff	$\exists_{x = (s_0, s_1, \cdots)} M, x \models f$
	$M, s_0 \models A\,f$	iff	$\forall_{x = (s_0, s_1, \cdots)} M, x \models f$
P_1	$M, x \models f$	iff	$M, s_0 \models f$
P_2	$M, x \models \neg f$	iff	not $M, x \models f$
	$M, x \models f \wedge g$	iff	$M, x \models f$ and $M, x \models g$
P_3	$M, x \models X\,f$	iff	$M, x^1 \models f$
	$M, x \models f\,U\,g$	iff	$\exists_{i \ge 0}\, M, x^i \models g$ and $\forall_{j<i}\, M, x^j \models f$
P_0	$M, x \models X\,f$	iff	$M, s_1 \models f$
	$M, x \models f\,U\,g$	iff	$\exists_{i \ge 0}\, M, s_i \models g$ and $\forall_{j<i}\, M, s_j \models f$

Figure 7.2. Semantics of CTL$^{(*)}$.

A CTL$^{(*)}$ formula ϕ is said to be satisfiable iff there exists a model for it, i.e., there exists a Kripke structure M and a state s such that $M, s \models \phi$ holds.

A CTL$^{(*)}$ formula ϕ is said to be valid (the word "tautology" would be appropriate as well) iff for every structure M and for every state s of M, $M, s \models \phi$ holds.

When needed, these definitions can easily be rephrased for path formulas.

7.4.2 CTL operators

In CTL path formulas cannot be nested. Path formulas are constructed using the X and U operators and must immediately be preceded by a unary E or A operator to turn them into state formulas again. We therefore combine these

possibilities into 4 separate operators with a slightly different notation for ease of writing:

$\mathsf{E}\,\mathsf{X}\,f$	becomes	$\mathsf{EX}\,f$
$\mathsf{E}\,(f\,\mathsf{U}\,g)$	becomes	$f\,\mathsf{EU}\,g$
$\mathsf{A}\,\mathsf{X}\,f$	becomes	$\mathsf{AX}\,f$
$\mathsf{A}\,(f\,\mathsf{U}\,g)$	becomes	$f\,\mathsf{AU}\,g$

The abstract syntax of CTL may now be expressed by the single BNF production rule:

$$SF ::= AP \mid \neg\, SF \mid SF \wedge SF \mid \mathsf{EX}\, SF \mid \mathsf{AX}\, SF \mid SF\, \mathsf{EU}\, SF \mid SF\, \mathsf{AU}\, SF\ .$$

In practice, one often chooses a different set of basic CTL operators. Here we select the operators EX, EG, and EU. EG is a new operator we haven't seen before. It is intended to express that *there exists a fullpath such that the operand holds for all states on that path*. A more formal definition is given in figure 7.3. The Kripke structure M is assumed to be known, and hence M is left out of the equations.

$$s_0 \models \mathsf{EX}\,f \quad \text{iff} \quad \underset{(s_0, s_1, \cdots)}{\exists}\ s_1 \models f$$

$$s_0 \models \mathsf{EG}\,f \quad \text{iff} \quad \underset{(s_0, s_1, \cdots)}{\exists}\ \underset{i \geq 0}{\forall}\ s_i \models f$$

$$s_0 \models f\,\mathsf{EU}\,g \quad \text{iff} \quad \underset{(s_0, s_1, \cdots)}{\exists}\ \underset{j \geq 0}{\exists}\ s_j \models g \wedge \underset{i<j}{\forall}\ s_i \models f$$

Figure 7.3. Selection of basic CTL operators.

Check for yourself that $\mathsf{EG}\,f = \neg\,(\mathsf{true}\,\mathsf{AU}\,\neg f)$. We will further include the following set of derived operators:

$$\begin{aligned}
f \vee g &= \neg\,(\neg f \wedge \neg g) \\
\mathsf{EF}\,f &= \mathsf{true}\,\mathsf{EU}\,f \\
\mathsf{AX}\,f &= \neg\,\mathsf{EX}\,\neg f \\
\mathsf{AG}\,f &= \neg\,\mathsf{EF}\,\neg f \\
f\,\mathsf{AU}\,g &= \neg\,((\neg g\,\mathsf{EU}\,(\neg f \wedge \neg g)) \vee \mathsf{EG}\,\neg g) \\
\mathsf{AF}\,f &= \mathsf{true}\,\mathsf{AU}\,f
\end{aligned}$$

In total we now have 8 temporal operators in our version of CTL. Figure 7.4 provides a memory aid.

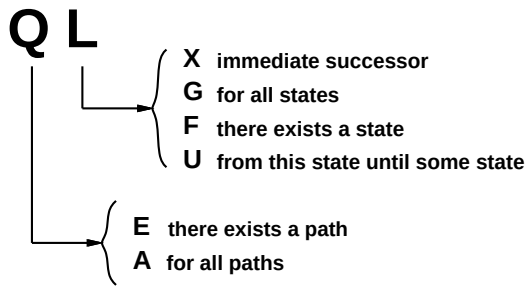

Figure 7.4. Nomenclature for CTL temporal operators.

7.4.3 Examples of CTL use

Figure 7.5 shows a sample Kripke structure (note that every state has at least 1 successor). The set of atomic propositions is $\{\mathsf{white}, \mathsf{black}\}$. The labelling function is given implicitly by the arrangement of white and black vertices: a white vertex assigns true to the white atomic proposition and false to black; for the black vertex the assignment is just the other way around. We of course also could have vertices that assign true or false to both the white and black atomic propositions, but this is not the case in this example.

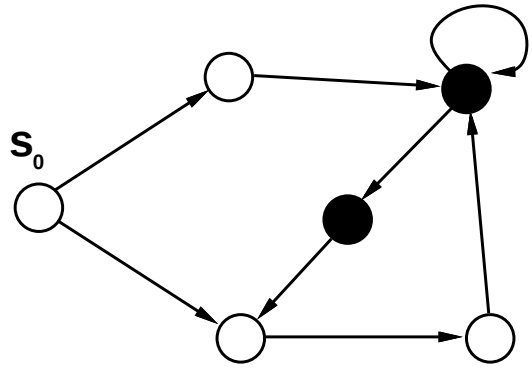

Figure 7.5. Kripke structure M with initial state s_0.

In fact we can express this simply by a CTL formula: $\mathsf{AG}\,(\mathsf{white} \oplus \mathsf{black})$, where $\oplus$ is the exclusive-or operator, which might be defined as an abbreviation by $f \oplus g = f \wedge \neg g \vee \neg f \wedge g$. The AG operator ensures that *for every path* out of state s_0 and *for every state* on such a path, $\mathsf{white} \oplus \mathsf{black}$ will be true. How about the formula $\mathsf{white}\,\mathsf{EU}\,\mathsf{black}$; does this hold for our model as well? Yes, because EU expresses that there exists a path out of s_0 such that on that path black eventually

gets true, and that in all states preceding that state white is assigned true. How do we express the fact that it is not possible to have a fullpath consisting of only white vertices? Simple: *it is not the case (¬) that there exists a path (E) such that all states on that path (G) have* true *assigned to* white. In a formula: $\neg\,\mathsf{EG}\,\mathsf{white}$.

And now for something tougher. How do we express that no fullpath can have a *single* black vertex? This suggests the use of a U-type operator: on such a path we see white vertices *until* a black one, and from then on (AX) only white vertices follow (AG). Solution: $\neg\,(\,\mathsf{white}\,\mathsf{EU}\,(\,\mathsf{black}\wedge\mathsf{AX}\,(\,\mathsf{AG}\,\mathsf{white}\,)\,)\,)$.

7.5 Interlude: Linear-Time Temporal Logic

Many flavours of temporal logic exist. The CTL$^{(*)}$ class of temporal logic belongs to the so-called branching-time temporal logics. If we throw away the possibility to quantify over paths, and in effect only consider a single fullpath of states as our model, the class of so-called linear-time temporal logics ensues. A typical representative is the system of Propositional Linear-Time Temporal Logic or PTL. The abstract syntax of PTL formulas is:

$$F ::= AP \mid \neg F \mid F \wedge F \mid \mathsf{X}F \mid F\,\mathsf{U}\,F\,.$$

Given a fullpath $\mathsf{x} = (\mathsf{s}_0, \mathsf{s}_1, \cdots)$, the meaning of the basic PTL operators is defined as follows:

$\mathsf{s}_0 \models \mathsf{p}$	iff	$\mathsf{p} \in \mathsf{L}(\mathsf{s}_0)$
$\mathsf{s}_0 \models \neg\,\mathsf{f}$	iff	not $\mathsf{s}_0 \models \mathsf{f}$
$\mathsf{s}_0 \models \mathsf{f} \wedge \mathsf{g}$	iff	$\mathsf{s}_0 \models \mathsf{f}$ and $\mathsf{s}_0 \models \mathsf{g}$
$\mathsf{s}_0 \models \mathsf{X}\,\mathsf{f}$	iff	$\mathsf{s}_1 \models \mathsf{f}$
$\mathsf{s}_0 \models \mathsf{f}\,\mathsf{U}\,\mathsf{g}$	iff	$\underset{\mathsf{j}\geq 0}{\exists}\,\mathsf{s}_\mathsf{j} \models \mathsf{g} \wedge \underset{\mathsf{i}<\mathsf{j}}{\forall}\,\mathsf{s}_\mathsf{i} \models \mathsf{f}$

Figure 7.6. Semantics of basic PTL operators.

Linear-time temporal logic has a more natural appeal to it: we intuitively consider time to progress along a non-branching time line, i.e., we only consider a single future. PTL therefore is a serious competitor of CTL in describing system behaviour. PTL is less suitable for model checking (the computational complexity is much worse). PTL is commonly used with a satisfiability checker tool. Such a program determines whether a PTL formula can be made true. It may do so by constructing a fullpath using the identity $\mathsf{f}\,\mathsf{U}\,\mathsf{g} = \mathsf{g} \vee \mathsf{f} \wedge \mathsf{X}\,(\mathsf{f}\,\mathsf{U}\,\mathsf{g})$. By complementing the PTL formula input we can use the same tool as a tautology

checker.

7.6 CTL Model Checking

Our purpose of introducing CTL is to arrive at a method for automatically verifying properties of systems. The idea is to model the system as a finite Kripke structure M with a distinguished initial state s_0 and then check whether a certain property expressed as a CTL formula ϕ holds within that model, i.e., whether we can prove that indeed $M, s_0 \models \phi$.

On the other hand, observe that any CTL formula ϕ could also be interpreted within a given Kripke structure M as denoting a set of states, namely those states s for which $M, s \models \phi$ holds. We define therefore:

$Q(\phi) \subseteq S$ is a set of states associated with CTL formula ϕ:

$Q(\phi) = \{ s \mid M, s \models \phi \}$

One way to find out whether a certain property holds for a given system, is to compute Q and check whether $s_0 \in Q$. The Q sets for each possible form of CTL formula are easily derived from the semantics. Figure 7.6 provides a complete list based on a given Kripke structure $M = (S, R, L)$. We use the notation $R(s)$ to stand for the set $\{ t \in S \mid (s,t) \in R \}$.

$$
\begin{array}{lcl}
Q(\mathsf{false}) & = & \varnothing \\
Q(\mathsf{true}) & = & S \\
Q(P) & = & \{ s \mid P \in L(s) \} \\
Q(\neg f) & = & S \setminus Q(f) \\
Q(f \wedge g) & = & Q(f) \cap Q(g) \\
Q(\mathsf{EX}\, f) & = & \{ s \mid R(s) \cap Q(f) \neq \varnothing \} \\
Q(\mathsf{EG}\, f) & = & Q(f) \cap Q(\mathsf{EX}\, \mathsf{EG}\, f) \\
Q(f\, \mathsf{EU}\, g) & = & Q(g) \cup Q(f) \cap Q(\mathsf{EX}\, (f\, \mathsf{EU}\, g))
\end{array}
$$

Figure 7.7. State-sets for the basic CTL formulas.

The last 2 equations are recurrent. Luckily their solutions are well-defined and can easily computed as we shall see in the next section.

7.6.1 Model checking algorithms

The state-sets for EG and EU are expressed as recurrent equations. These equations are derived from the following logical equivalences for these operators:

$\mathsf{EG}\, f = f \wedge \mathsf{EX}\, \mathsf{EG}\, f$

$$f\,EU\,g = g \vee f \wedge EX\,(f\,EU\,g)$$

You might look at these equations as theorems of CTL and prove them by resorting to their semantic definitions. Note that $Q(EXf)$ is defined to be all those states that have at least one successor that belongs to $Q(f)$. In other words, $Q(EXf)$ is the *image* of $Q(f)$ under the converse relation R^{-1}; this is usually called the *pre-image* under R. Apart from R^{-1} we will use the following notation for the various sets associated with R:

$R \subseteq A \times A$	Relation
$R : A \to 2^A$	Function
$R : 2^A \to 2^A$	Extended function
$R(s) = \{t \mid (s,t) \in R\}$	(definition)
$R(S) = \bigcup_{s \in S} R(s)$	(extension)
$R^0(S) = S$	Identity
$R^1(S) = R(S)$	Image
$R^k(S) = R^{k-1}(R(S))$	Iterated application
$R^{-1}(t) = \{s \mid (s,t) \in R\}$	(converse)
$R^{-1}(T) = \bigcup_{t \in T} R^{-1}(t)$	Pre-image
$R^{-k}(T) = R^{-(k-1)}(R^{-1}(T))$	Iterated application

Using $Q(EXf) = R^{-1}(Q(f))$ (check this) we can rewrite the state-sets for EG and EU as follows:

$$Q(EGf) = Q(f) \cap R^{-1}(Q(EGf))$$

$$Q(f\,EU\,g) = Q(g) \cup Q(f) \cap R^{-1}(Q(f\,EU\,g))$$

To solve these equations we need to apply a bit of fixpoint theory. Assuming that $Q(f)$ and $Q(g)$ are known, i.e., the terms may be considered constant, say Q_f and Q_g, what we are dealing with are functions F of signature $2^S \to 2^S$ for which we like to find a fixpoint value. A fixpoint $Z \in S$ is a value such that $Z = F(Z)$. Our task is to solve the following fixpoint equations:

$$Z_{EG} = F_{EG}(Z_{EG}) = Q_f \cap R^{-1}(Z_{EG})$$

$$Z_{EU} = F_{EU}(Z_{EU}) = Q_g \cup Q_f \cap R^{-1}(Z_{EU})$$

Remember from algebra that $(2^S, \subseteq)$ is a complete lattice, and so (trivially) is $(2^S, \supseteq)$. $(2^S, \subseteq)$ has bottom (or least) element $\bot = \varnothing$ and top (or greatest) element $\top = S$; for $(2^S, \supseteq)$ it is the other way around. From the Tarski-Knaster fixpoint theorem we learn that for a complete lattice over 2^S the least and greatest

fixpoints for *continuous* functions $F : 2^S \to 2^S$ do exist and are unique, and further that they may be calculated as follows:

- *least fixpoint*: $\mathsf{lfp}(F) = \mu Z.\, F(Z) = \mathsf{supremum}\{F^i(\bot) \mid i \geq 0\} = \bigcup_{i \geq 0} F^i(\bot)$, provided F is $\cup$-continuous, i.e., for any chain $X_0 \subseteq X_1 \subseteq X_2 \subseteq \cdots$ we have $F(\bigcup_{i \geq 0} X_i) = \bigcup_{i \geq 0} F(X_i)$.
- *greatest fixpoint*: $\mathsf{gfp}(F) = \nu Z.\, F(Z) = \mathsf{infinum}\{F^i(\top) \mid i \geq 0\} = \bigcap_{i \geq 0} F^i(\top)$, provided F is $\cap$-continuous, i.e., for any chain $X_0 \supseteq X_1 \supseteq X_2 \supseteq \cdots$ we have $F(\bigcap_{i \geq 0} X_i) = \bigcap_{i \geq 0} F(X_i)$.

Since monotonicity follows from continuity, we have:

$\forall_{i \geq 0}\ F^i(\bot) \subseteq F^{i+1}(\bot)$, and hence $\mu Z.\, F(Z) = \lim_{i \to \infty} F^i(\bot)$

$\forall_{i \geq 0}\ F^i(\top) \supseteq F^{i+1}(\top)$, and hence $\nu Z.\, F(Z) = \lim_{i \to \infty} F^i(\top)$

For finite lattices, i.e., the set 2^S is finite, these results immediately lead to a successive approximation algorithm to compute the fixpoints. For our CTL state-sets application we have to decide whether we want the least fixpoint solution or the greatest fixpoint. This of course is determined by the semantics of the operators. Without proof we here state the correct fixpoint characterizations of the state-sets for the EG and EU operators:

$Q(\mathsf{EG}\, f) = \nu Z.\, Q(f) \cap R^{-1}(Z)$

$Q(f\, \mathsf{EU}\, g) = \mu Z.\, Q(g) \cup Q(f) \cap R^{-1}(Z)$

The computation of $Q(\mathsf{EG}\, f)$ proceeds as follows. We start with our initial approximation $Z_0 = S$ because we have a greatest fixpoint at hand. Then we

```
2^S EG (CTL f)
{
  k:=0;   Z_k:=S;
  do {
    Z_{k+1}:=Q(f) ∩ R^{-1}(Z_k);
    if (Z_{k+1}=Z_k) return Z_k;
    k++;
  } forever;
}
```

Algorithm 7.1. Iterative calculation of $Q(\mathsf{EG}\, f)$.

calculate $Z_1 = Q(f) \cap R^{-1}(Z_0) = Q(f) \cap R^{-1}(S)$. Using this result, we calculate Z_2, and so on, till we find at some step $k \geq 0$ that $Z_{k+1} = Z_k$, in which case we are done and the solution is Z_k. Algorithm 7.1 presents the pseudo-C code for this procedure.

The following example is intended to illustrate the workings of this simple algorithm. We consider the Kripke structure of figure 7.7. There are 6 states numbered 0 through 5: $S = \{0, 1, 2, 3, 4, 5\}$. There is a single atomic proposition P which is assigned true in the states labelled with P and false otherwise: $Q(P) = \{0, 1, 3, 4\}$.

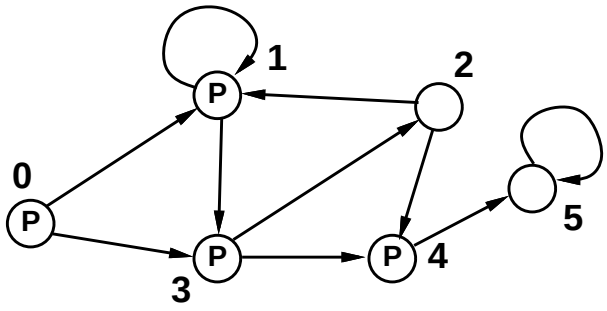

Figure 7.8. Kripke structure used in $Q(EG\,P)$ example.

Now follow all the steps necessary in the calculation of $Q(EG\,P)$:

$$Z_0 = S = \{0, 1, 2, 3, 4, 5\}$$

$$(R^{-1}(Z_0) = R^{-1}(\{0, 1, 2, 3, 4, 5\}) = \{0, 1, 2, 3, 4, 5\})$$

$$Z_1 = \{0, 1, 3, 4\} \cap R^{-1}(\{0, 1, 2, 3, 4, 5\}) = \{0, 1, 3, 4\}$$

$$(R^{-1}(Z_1) = R^{-1}(\{0, 1, 3, 4\}) = \{0, 1, 2, 3\})$$

$$Z_2 = \{0, 1, 3, 4\} \cap R^{-1}(Z_1) = \{0, 1, 3\}$$

$$(R^{-1}(Z_2) = R^{-1}(\{0, 1, 3\}) = \{0, 1, 2\})$$

$$Z_3 = \{0, 1, 3, 4\} \cap R^{-1}(Z_2) = \{0, 1\}$$

$$(R^{-1}(Z_3) = R^{-1}(\{0, 1\}) = \{0, 1, 2\})$$

$$Z_4 = \{0, 1, 3, 4\} \cap R^{-1}(Z_3) = \{0, 1\} = Z_3$$

We see that the solution is $Q(EG\,P) = \{0, 1\}$.

In an analogous way we can derive the procedure to compute $Q(f\,EU\,g)$. This is shown in algorithm 7.2.

```
2^S EU (CTL f, CTL g)
{
  k:=0;  Z_k:=∅;
  do {
    Z_{k+1}:=Q(g) ∪ (Q(f) ∩ R^{-1}(Z_k));
    if (Z_{k+1}=Z_k) return Z_k;
    k++;
  } forever;
}
```

Algorithm 7.2. Iterative calculation of $Q(f\,EU\,g)$.

Note that for a Kripke structure $R^{-1}(S) = S$ and $R^{-1}(\emptyset) = \emptyset$. Therefore we could have slightly simplified the above algorithms by using a different initialisation and then skipping the first iteration step. Apart from the need to calculate R^{-1}, i.e., the pre-image, the algorithms solely use set operations which may be implemented in variety of ways. In the next chapter we will learn about a data structure that is particularly suited to implement a model checking algorithm.

7.7 References

[**Emer90**] Emerson, E.A., "Chapter 16: Temporal and Modal Logic," in *Handbook of Theoretical Computer Science*, ed. Jan van Leeuwen, vol. B: Formal Models and Semantics, pp. 996-1072, Elsevier Science Publishers B.V., 1990.

[**McMi93**] McMillan, Kenneth L., in *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.