Chapter 4

Temporal Logics

Temporal logics are logics about time. Time is considered a discrete entity. We consider properties over sequence of events. Time is here the separation between events. We can then distinguish:

- the current event, that is, the current position in a sequence;
- the previous event, that is, the previous position in a sequence;
- the next event, that is, the next position in a sequence.

This intuition supposes linear sequences. The first logic – LTL for Linear Time Logic – we will study indeed assumes linear sequences. In contrast, Computation Tree Logic - CTL, the second logic we will study – is defined over branching sequences. In this case, we can speak about a *set* of possible next positions in sequences. We will see that these two logics are incomparable. There are properties that can be expressed in LTL that cannot be expressed in CTL and vice versa. Finally, CTL* includes both LTL and CTL.

4.1 Kripke structures

4.1.1 Definition

To define semantics for the temporal logics, we need a formal model. The standard model is a Kripke structure. The later is similar to a state graph. Edges have no label. Vertices represent states. There is a labelling function for states. This function represents the set of propositions that are true in the state. Formally, we have the following:

Definition 4.1.1. A Kripke structure is a tuple M = (S, I, T, L) with a finite set of state S, a finite set of initial states $I \subseteq S$, a finite transition relation between states $T \subseteq S \times S$, and a labelling function $L : S \rightarrow \mathcal{P}(A)$ with atomic propositions A.

This definition has four elements:

- a finite set of states;
- a finite set of initial states;
- a transition relation that gives for every state the possible successor states. There may be more than one successor state. A common restriction on the transition relation is to be total, that is, there is always a successor state;
- a labelling function that tells which propositions are true in a state. This is the central aspect of Kripke structures: information is stored in states.

Defining a Kripke structures means giving definitions to these four elements.

A *path* is an infinite sequence of states. Formally, a path π from state s is the infinite sequence $s_0, s_1, ...$ such that $s_0 = s$ and $\forall i \geq 0. T(s_i, s_{i+1})$, that is, there is a transition between all states.

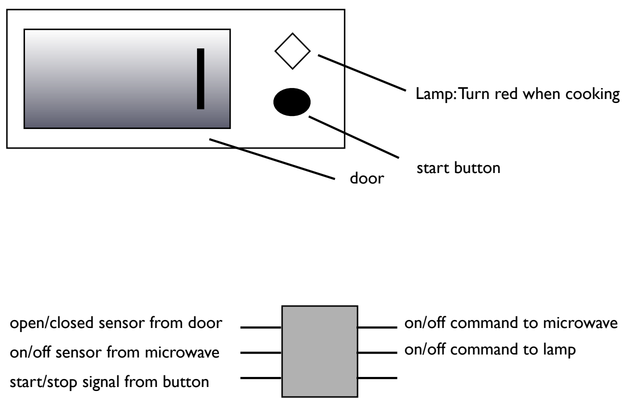
Lamp: Turn red when cooking
start button
door
open/closed sensor from door
on/off sensor from microwave
start/stop signal from button
on/off command to microwave
on/off command to lamp

Figure 4.1: A simple microwave-oven and its controller

Consider the example pictured in Figure 4.1. It represents a simple microwave oven with a door, a start button, and a lamp. The door can be closed or open. Users push the start button to cook for 10 seconds. The lamp turns red when the oven is cooking. We focus on the control software running on the micro-controller inside the oven. This micro-controller is also pictured in Figure 4.1. It is connected to five wires. It receives two signals from sensors giving information about the status of the door and the oven. A door sensor signal set to 1 means that the door is open. This signal is 0 when the door is closed. The oven sensor signal is 1 when the oven is cooking and 0 otherwise. The micro-controller also receives a pulse when the button is pushed. When the button is pushed, the button signal is set to 1 for 10 ms before coming back to 0. The micro-controller has two command signals. A 1 is sent to the lamp or the oven to turn these devices on. When the command is put back to 0, these devices are turned off. The oven must satisfy the safety property that it is never cooking while the door is open. This completes an informal description of the oven and its expected behaviour.

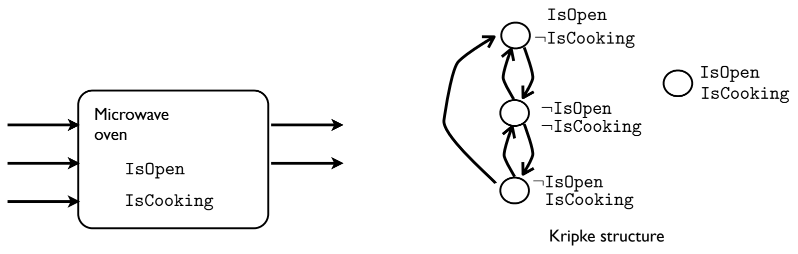

Figure 4.2: Modelling the microwave-oven for model-checking

The Kripke structure used to represent the behaviour of state variables `IsCooking` and `IsOpen` is given in Figure 4.2. Initially, the door is opened and the oven is not cooking. Once the door has been closed, either the oven starts cooking or the door is open again. When the oven is cooking, the oven may stop and keep its door closed or stop with an open door.

Exercise 4.1.2. Give a formal definition of the Kripke structure representing the oven example and pictured in Figure 4.2. What are execution paths of this Kripke structure?

4.1.2 First order representation

Defining all components of the definition of a Kripke structure is a tedious task. First order logic provides a more compact way to represent a Kripke structure. In particular, it introduces the "prime" notation to encode transition. For instance, the transition of Boolean variable a from true to false is encoded as $a \wedge \neg a'$. Here, a denotes the value before the transition and a' denotes the value after the transition.

Note that the notation for the first order presentation of a Kripke differs from the notation used to represent the Kripke structure. In general, caligraphic letters are used for the first order representation. For instance, $\mathcal{S}$ represents the first order representation of the set of states S and $\mathcal{T}$ represents the first order representation of the transition relation T.

Let us consider a simple example. Consider the program below:

```
0: x = x + y
1: y = x - y
2: x = x - y
```

These three lines are executed over and over again. The initial values of x and y are 3 and 5. This is represented by the following formula defining the set of initial states:

$$\mathcal{S}_0 \equiv x = 3 \wedge y = 5$$

Each line of the program is represented by a transition. Using the prime notation the three transitions can be represented as follows:

$$\mathcal{T}(x, y, x', y') \equiv (x' = x+y \wedge y' = y) \vee (x' = x \wedge y' = x-y) \vee (x' = x-y \wedge y' = y)$$

We assume synchronous semantics, that is, the sequential execution of the three lines of our program example. This synchronous execution needs to be expressed in the Kripke model. This is achieved by introducing an extra variable pc as a "program counter". Variable pc takes values in $\{0, 1, 2\}$ and it points to the line that should be executed. In every transition, variable pc is incremented cyclically, that is, $pc' = (pc + 1) \bmod 3$. At the end, the complete transition relation is as follows:

$$\begin{array}{rcl} & & (x' = x + y \wedge y' = y \wedge pc = 0 \wedge pc' = (pc+1) \bmod 3) \\ \mathcal{T}(x, y, x', y') \equiv & \vee & (x' = x \wedge y' = x - y \wedge pc = 1 \wedge pc' = (pc+1) \bmod 3) \\ & \vee & (x' = x - y \wedge y' = y \wedge pc = 2 \wedge pc' = (pc+1) \bmod 3) \end{array}$$

The state graph of the Kripke structure is given in Figure 4.3. States are triples representing the values of x, y, and pc.

$$\begin{tikzcd}
{} \arrow[rd] & & & & \\
& (3,5,0) \arrow[r] & (8,5,1) \arrow[r] & (8,3,2) \arrow[r] & (5,3,0) \arrow[ld] \\
& & (8,5,2) \arrow[lu] & (8,3,1) \arrow[l] &
\end{tikzcd}$$

Figure 4.3: State graph for our example

As expected the program simply exchanges the values of x and y.

From the first order encoding, one can extract the Kripke structure defined as follows:

- $D = \{3, 5, 8\}$ and $S = D \times D \times \{0, 1, 2\}$
- $S_0 = \{(3, 5)\}$
- $T = \{((3, 5, 0), (8, 5, 1)), ((8, 5, 1), (8, 3, 2)), ((8, 3, 2), (5, 3, 0)), ((5, 3, 0), (8, 3, 1)), ((8, 3, 1), (8, 5, 2)), ((8, 5, 2), (3, 5, 0))\}$
- $L((i, j, k)) = \{x = i \wedge y = j \wedge pc = k\}$

The first order representation is a symbolic representation of the Kripke structure. A state is a particular assignments of the state variables. A valuation is a function assigning concrete values to variables. The states of a system are all possible valuations for all the variables. For instance, a system with a 64-bit register has 2^{64} possible states. Transitions also are encoded using first order logic using the prime notation. As we sketched it earlier, it is a convention where v denotes the current value of variable – that is, before a transition – and v' denotes the next value of a variable – that is, the value after a transition. A function incrementing a 64-bit register would be defined as $\mathit{reg}' = (\mathit{reg} + 1) \bmod 64$.

Exercise 4.1.3. Give the first order representation of the Kripke structure of the oven example and given as answer to the previous question.

4.1.3 Conclusion

We shortly described Kripke structures as a formal model of state-based systems. Kripke are therefore an apposite model for sequential hardware. Also, Kripke structures are the formal models used to define the temporal logics. The next sections introduces one of these logics, namely, Linear Time Logic (LTL).

4.2 LTL

LTL considers linear sequences. LTL formula's are defined over paths, that is, infinite linear sequences of states. The main operators are "next p" – notation $\mathbf{X}p$ – meaning p is true *at the next time*; and "p until q – notation $p\ \mathbf{U}\ q$ – meaning that p holds for some time after which q holds.

The core syntax of an LTL formula φ is defined by the following grammar:

$$\varphi := \neg\varphi \mid \varphi \wedge \psi \mid a \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\psi$$

That is, LTL formulas are constructed according to the following rules:

- A proposition is a valid LTL formula.
- Given two valid LTL formulas φ and ψ, $\neg\varphi$ and $\varphi \wedge \psi$ are also valid LTL formulas.

- Given two valid LTL formulas φ and ψ, $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\psi$ are also valid LTL formulas.

We introduce the following derived operators:

- $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \Leftrightarrow \psi \equiv \varphi \Rightarrow \psi \wedge \psi \Rightarrow \varphi$
- $\top \equiv \varphi \vee \neg\varphi$
- $\bot \equiv \neg\top$
- $\mathbf{F}\varphi \equiv \top\mathbf{U}\varphi$ ”eventually φ”
- $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$ ”globally φ”

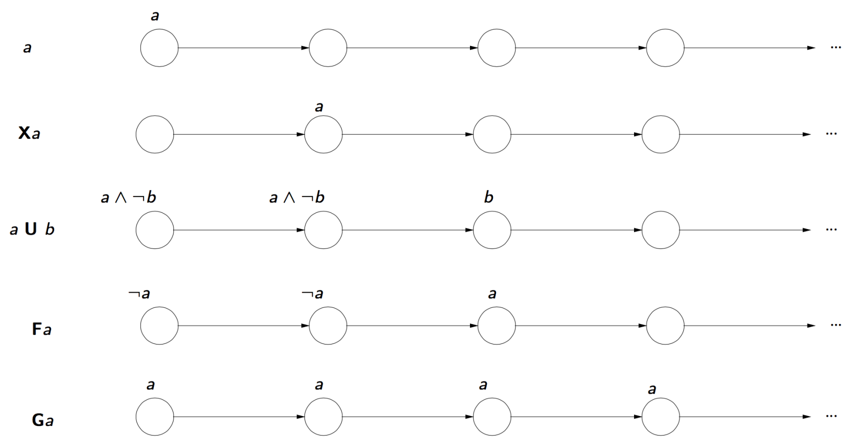

Figure 4.4: Intuitive semantics for LTL operators.

The intuitive semantics of these operators is illustrated in Figure 4.4 that shows several infinite paths starting in an arbitrary state. Proposition a is true if it is true in the first state of the path. Formula $\mathbf{X}a$ is true if a is true in the next state. Formula $a\mathbf{U}b$ holds if a holds for some state until a state is reached where b holds. Formula $\mathbf{F}a$ (”eventually a”) holds if there exists a state where a holds. Formula $\mathbf{G}a$ (”globally a”) holds if a holds in all states along the paths. The next definition gives the formal semantics of LTL formulas over an infinite path. Given an infinite path π, we define $\pi(i)$ to be the state at position i in the path, noted s_i. We also define π^i to be the suffix of path π from position i, that is, $s_i, s_{i+1}, ...$

Definition 4.2.1. Let π be an infinite path in a Kripke structure M.

$$\begin{array}{lllc}
\pi \models & p & \text{iff} & p \in L(\pi(0)) \\
\pi \not\models & \neg p & \text{iff} & p \notin L(\pi(0)) \\
\pi \models & f \wedge g & \text{iff} & \pi \models f \text{ and } \pi \models g \\
\pi \models & \mathbf{X}f & \text{iff} & \pi^1 \models f \\
\pi \models & f\mathbf{U}g & \text{iff} & \exists j.\pi^i \models g \wedge \forall j.j < i.\pi^j \models f
\end{array}$$

Exercise 4.2.2. The above definition only considers the basic operations. Give a formal definition of the derived operators $\vee$, $\mathbf{F}$, $\mathbf{G}$. Prove that the syntactic definition of these derived operators satisfy these semantics.

This defines the validity of a formula over an infinite path. In LTL, a Kripke structure M satisfies an LTL formula if this formula holds *on all paths* starting in one of the initial states.

Considering the oven example of the previous section, we can express the property that the oven is never cooking with an open door as variable `IsOpen` is false whenever variable `IsCooking` is true; or, in LTL syntax, $\mathbf{G}(\texttt{IsCooking} \Rightarrow \neg\texttt{IsOpen})$. Here are a few more examples about LTL properties:

- Whenever the light is red, it cannot become green immediately:

$$\mathbf{G}(\texttt{red} \Rightarrow \neg\mathbf{X}\texttt{green})$$

- The traffic light eventually becomes green:

$$\mathbf{F}\texttt{green}$$

- Once red, the light eventually becomes green:

$$\mathbf{G}(\texttt{red} \Rightarrow \mathbf{F}\texttt{green})$$

- After being red, the light goes yellow and then eventually becomes green:

$$\mathbf{G}(\texttt{red} \Rightarrow \mathbf{X}(\texttt{red}\mathbf{U}(\texttt{yellow} \wedge \mathbf{X}(\texttt{yellow}\mathbf{U}\texttt{green}))))$$

We can classify LTL properties in different categories:

- Reachability properties express the fact that something will eventually happen. They have the form $\mathbf{F}\varphi$. We distinguish the following:
  - negated reachability: $\mathbf{F}\neg\varphi$
  - conditional reachability: $\varphi\mathbf{U}\psi$
  - reachability from every state: this cannot be expressed in LTL. But it can be expressed in CTL (see next section).
- Safety properties express the fact that something bad will never happen. They have the form $\mathbf{G}\neg\varphi$. We define a weak form of conditional safety: $(\varphi\mathbf{U}\psi) \vee \mathbf{G}\varphi$. Such a property means ψ may only happen after φ holds.

- Liveness properties express the fact that something must always eventually happen. The main property is $\mathbf{G}(\varphi \Rightarrow \mathbf{F}\psi)$ and is called a *request-response* property. Very often φ is a request and ψ is the grant response. Such properties express that all requests must be eventually granted.
- Fairness properties express the fact that something happens infinitely often. They have the form $\mathbf{GF}\varphi$.

Another interesting class of properties are of the form $\mathbf{FG}p$ meaning eventually p always hold. This can be useful to express equivalence of two state machines after some initialising sequence. In this case, p is the output of a miter and must eventually always be 0.

Exercise 4.2.3. Give the formal semantics of properties of the form $\mathbf{GF}p$ and $\mathbf{FG}p$.

An essential duality in LTL is that the negation of an eventually is equivalent to an inevitably property, formally, we have the following:

$$\neg\mathbf{F}\neg p \equiv \mathbf{G}p$$

Exercise 4.2.4. Prove this duality.

4.3 CTL

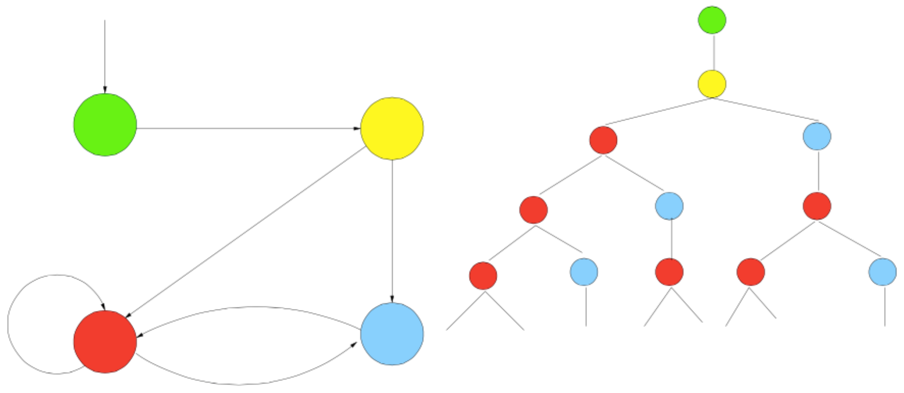

Figure 4.5: Unwinding of a Kripke structure into an infinite tree.

The Computation Tree Logic (CTL) considers execution trees instead of linear paths of states. The idea is to unwind a Kripke structure into an infinite tree. Figure 4.5 shows an example of a Kripke structure and its unwinding into an infinite tree.

A consequence of looking at trees instead of paths is that a state now may have more than one successor. CTL introduces *path* quantifiers to express that a property must all on all paths starting at a given state, or that a property must hold for some path starting from a given state. Therefore, the syntax and semantics of CTL distinguish between state formulas and path formulas.

Let A be a set of atomic propositions. The syntax of state formulas is given by the following rules:

- If $p \in A$, p is a state formula.
- If φ_1 and φ_2 are state formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$ are state formulas.
- If ψ is a *path* formula, $\mathbf{E}\psi$ and $\mathbf{A}\psi$ are state formulas.

Two additional rules specify the syntax of path formulas:

- If φ is a state formula, then φ is a path formula.
- If φ_1 and φ_2 are *state* formula, then $\mathbf{X}\varphi_1$, $\mathbf{F}\varphi_1$, $\mathbf{G}\varphi_1$, and $\varphi_1\mathbf{U}\varphi_2$ are path formulas.

The second rule ensures that in any CTL formula, temporal operators (e.g. $\mathbf{X}$ and $\mathbf{U}$) are *immediately* preceded by a path quantifier. In other words, path quantifiers and temporal operators must alternate.

Let have a look at some examples:

- Is $(\mathbf{A}\neg\mathbf{X}p)$ a valid CTL formula? The answer is no. p is an atomic proposition, and therefore is a valid state formula. We can then form the valid path formula $\mathbf{X}p$. There is no rule in the semantics that from a valid path formula allows the creation of its negation. The syntax only allows the negation of state formulas. For a similar reason, we cannot syntactically defined $\mathbf{AF}p$ in CTL as being $\neg E\neg F\neg p$, because the $\neg$ between the $\mathbf{E}$ and the $\mathbf{F}$ is not allowed by the syntax.
- Is $\mathbf{XA}p$ a valid CTL formula? The answer is yes. It is a valid *path* formula that can be constructed as follows. p is an atomic proposition and therefore a valid state formula. A valid state formula is also a valid path formula. We can then construct the state formula $\mathbf{A}p$. We then construct from this valid state formula, the valid path formula $\mathbf{XA}p$.
- Note that $\mathbf{EXA}p$ is a valid state formula.

Here are a few other examples of CTL formulas:

- It is possible to get to a state where `start` holds but `ready` does not hold:

$$\mathbf{EF}(\texttt{start} \wedge \neg\texttt{ready})$$

- If a request occurs, then it will be eventually acknowledged:

$$\mathbf{AG}(\texttt{req} \Rightarrow \mathbf{AF}\ \texttt{req})$$

- `ready` holds infinitely often on every path:

$$\mathbf{AG}(\mathbf{AF}\ \texttt{ready})$$

- it is possible to `reset` from every state:

$$\mathbf{AG}(\mathbf{EF}\ \texttt{reset})$$

To give formal semantics of CTL, we add the notation $s \models \varphi$ to denote that state formula φ holds in state s. It is assumed that formulas φ_1 and φ_2 are state formulas and ψ_1 and ψ_2 are path formulas.

Definition 4.3.1.

$$\begin{array}{rlcc}
s \models & p & \text{iff} & p \in L(s) \\
s \models & \neg\varphi_1 & \text{iff} & s \not\models \varphi_1 \\
s \models & \varphi_1 \vee \varphi_2 & \text{iff} & s \models \varphi_1 \text{ or } s \models \varphi_2 \\
s \models & \varphi_1 \wedge \varphi_2 & \text{iff} & s \models \varphi_1 \text{ and } s \models \varphi_2 \\
s \models & \mathbf{E}\psi_1 & \text{iff} & \exists\pi.\pi(0) = s \wedge \pi \models \psi_1 \\
s \models & \mathbf{A}\psi_1 & \text{iff} & \forall\pi.\pi(0) = s \wedge \pi \models \psi_1 \\
\pi \models & \varphi_1 & \text{iff} & \pi(0) \models \varphi_1 \\
\pi \models & \mathbf{X}\psi_1 & \text{iff} & \pi^1 \models \psi_1 \\
\pi \models & \mathbf{F}\psi_1 & \text{iff} & \exists i.\pi^i \models \psi_1 \\
\pi \models & \mathbf{G}\psi_1 & \text{iff} & \forall i.\pi^i \models \psi_1 \\
\pi \models & \psi_1\mathbf{U}\psi_2 & \text{iff} & \exists i.\pi^i \models \psi_2 \wedge \forall j < i.\pi^j \models \psi_1
\end{array}$$

All CTL properties used in specifications must be state formulas. Because, a CTL formula holds for Kripke structure M if and only if it holds in all initial states of M.

We can then define several dereived CTL operators:

- Potentially φ: $\mathbf{EF}\varphi \equiv \mathbf{E}(\top\mathbf{U}\varphi)$.
- Inevitable φ: $\mathbf{AF}\varphi \equiv \mathbf{A}(\top\mathbf{U}\varphi)$.
- Potentially always φ: $\mathbf{EG}\varphi \equiv \neg\mathbf{AF}\neg\varphi$.
- Invariantly φ: $\mathbf{AG}\varphi \equiv \neg\mathbf{EF}\neg\varphi$.

For model checking, we basically consider a restricted set of operators from which all other ones can be derived. This set consists in the operators $\mathbf{EX}$, $\mathbf{EG}$ and $\mathbf{EU}$.

The semantics of several operators are illustrated in Figure 4.6.

4.4 CTL*

CTL* is CTL without the restriction that temporal operators and path quantifiers must alternate. Syntactically it allows Boolean operations between path formulas. The syntax for state formulas is the same as the syntax defined earlier for CTL. For path formulas, the rules defining the syntax are the following:

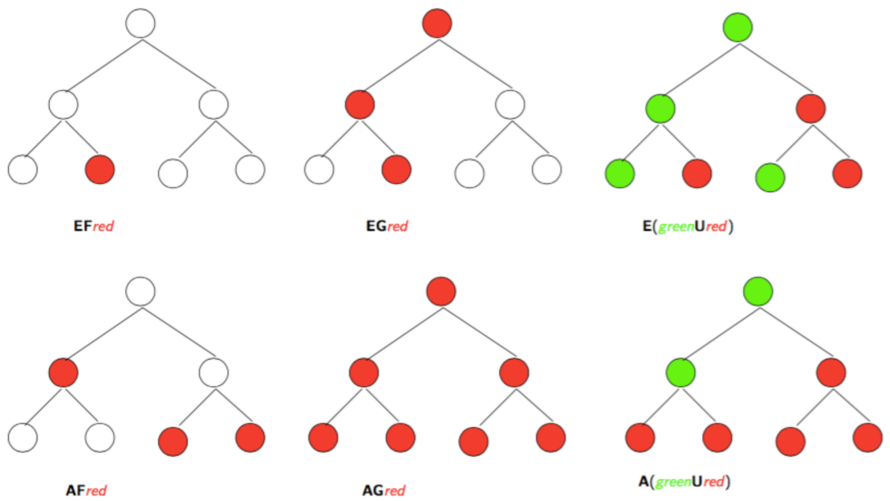

Figure 4.6: Intuitive semantics for several CTL operators.

- If φ is a state formula, then φ is also a path formula.
- If ψ_1 and ψ_2 are path formulas, then $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, $\mathbf{X}\psi_1$, $\mathbf{F}\psi_1$, $\mathbf{G}\psi_1$, $\psi_1\mathbf{U}\psi_2$ are path formulas.

In CTL*, we can therefore define $\mathbf{A}\varphi$ as $\neg\mathbf{E}\neg\varphi$, which is not allowed in CTL. The formal semantics of CTL* formula is the following:

Definition 4.4.1.

$$
\begin{array}{lllc}
s \models & p & \text{iff} & p \in L(s) \\
s \models & \neg\varphi_1 & \text{iff} & s \not\models \varphi_1 \\
s \models & \varphi_1 \vee \varphi_2 & \text{iff} & s \models \varphi_1 \text{ or } s \models \varphi_2 \\
s \models & \varphi_1 \wedge \varphi_2 & \text{iff} & s \models \varphi_1 \text{ and } s \models \varphi_2 \\
s \models & \mathbf{E}\psi_1 & \text{iff} & \exists\pi.\pi(0) = s \wedge \pi \models \psi_1 \\
s \models & \mathbf{A}\psi_1 & \text{iff} & \forall\pi.\pi(0) = s \wedge \pi \models \psi_1 \\
\pi \models & \varphi_1 & \text{iff} & \pi(0) \models \varphi_1 \\
\pi \models & \neg\psi_1 & \text{iff} & \pi \not\models \psi_1 \\
\pi \models & \psi_1 \vee \psi_2 & \text{iff} & \pi \models \psi_1 \text{ or } \pi \models \psi_2 \\
\pi \models & \psi_1 \wedge \psi_2 & \text{iff} & \pi \models \psi_1 \text{ and } \pi \models \psi_2 \\
\pi \models & \mathbf{X}\psi_1 & \text{iff} & \pi^1 \models \psi_1 \\
\pi \models & \mathbf{F}\psi_1 & \text{iff} & \exists i.\pi^i \models \psi_1 \\
\pi \models & \mathbf{G}\psi_1 & \text{iff} & \forall i.\pi^i \models \psi_1 \\
\pi \models & \psi_1\mathbf{U}\psi_2 & \text{iff} & \exists i.\pi^i \models \psi_2 \wedge \forall j < i.\pi^j \models \psi_1
\end{array}
$$

4.5 Comparison

It can be actually shown the three logics discussed so far have different expressive powers. CTL* is the most expressive logic. It includes CTL and LTL. We can embed LTL in CTL* by making the implicit universal path quantifiers of LTL explicit. Given a state s we have:

$$s \models_{LTL} \varphi \Leftrightarrow s \models_{CTL*} \mathbf{A}\varphi$$

CTL and LTL are incomparable. There are LTL formulas for which no equivalent CTL formula exists and vice versa.

We say that CTL-formula φ and LTL-formula ψ are equivalent if for any Kripke structure M, M satisfies φ if and only it satisfies ψ. The other important observation is that removing all path quantifiers from a CTL-formula φ either result in an equivalent LTL-formula or there does not exist any LTL-formula that is equivalent to φ. So, to prove that there a CTL-formula has no LTL equivalent it is enough to exhibit a Kripke structure that satisfies the CTL formula but not the LTL formula or vice versa.

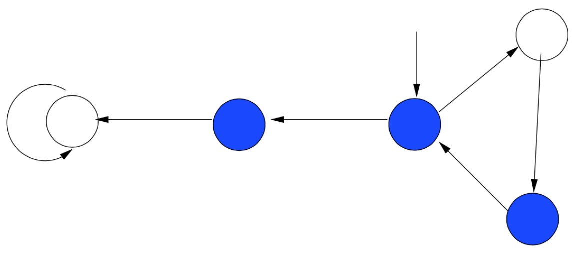
Figure 4.7: Kripke structure satisfying $\mathbf{F}(a \wedge \mathbf{X}\ a)$ but not $\mathbf{AF}(a \wedge \mathbf{AX}\ a)$.

Consider the Kripke structure in Figure 4.7, LTL-formula $\mathbf{F}(a \wedge \mathbf{X}\ a)$, and CTL-formula $\mathbf{AX}(a \wedge \mathbf{AX}\ a)$. The Kripke structure satisfies $\mathbf{F}(a \wedge \mathbf{X}\ a)$ as all paths either satisfy a in the first two states or in the second, third, and fourth states. So, in all paths there exists a position where a is true at the current and the next step. The CTL counterpart does not hold because $\mathbf{AX}a$ does not hold in the initial state. It is possible to reach a state where a does not hold. Therefore we have found a CTL formula that has no equivalent in LTL.

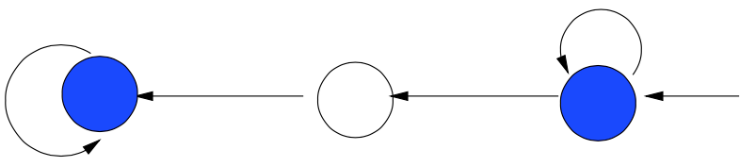
Figure 4.8: Kripke structure satisfying $\mathbf{FG}\ a$ but not $\mathbf{AFAG}\ a$.

There is no CTL formula equivalent to the LTL formula $\mathbf{FG}a$. This property expresses the fact that in all path there is a position from which a always hold. A CTL property that would express this would be $\mathbf{AFAG}a$. Remember that we need to alternate between path quantifiers and temporal operators. The Kripke structure in Figure 4.8 satisfies $\mathbf{FG}a$ but not $\mathbf{AFAG}a$. Paths from the initial state are either an infinite sequence of a state, or a non a state followed by an infinite sequence of a state. But, the initial state has a non a state as successor, therefore $\mathbf{AG}\ a$ does not hold initially.

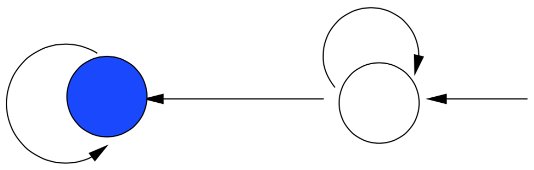

Figure 4.9: Kripke structure satisfying $\mathbf{AGEF}\ a$ but not $\mathbf{GF}\ a$.

Similarly, there is no LTL formula equivalent to the CTL formula $\mathbf{AGEF}a$. This is the "reset" property, that a state is reachable from any other state. The LTL candidate would be $\mathbf{GF}a$ meaning always eventually a. Figure 4.9 shows a Kripke structure satisfying $\mathbf{AGEF}a$ but not $\mathbf{GF}a$. There are paths that are infinite sequences of the initial state, which does not satisfy a. Whereas all paths in the computation tree have always the possibility to go to the a state and remain in this state.

4.6 Conclusion

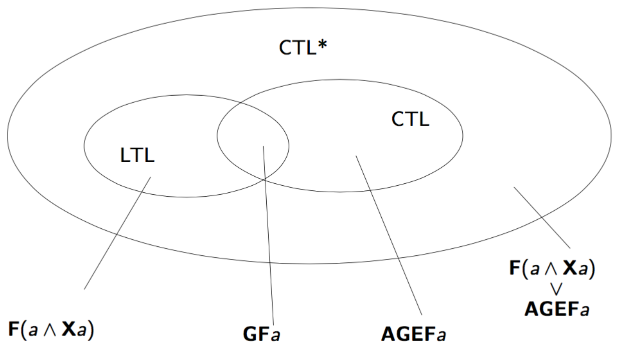
CTL*
CTL
LTL
$\mathbf{F}(a \wedge \mathbf{X}a)$
$\mathbf{GF}a$
$\mathbf{AGEF}a$
$\mathbf{F}(a \wedge \mathbf{X}a) \vee \mathbf{AGEF}a$

Figure 4.10: Summary of the differences between LTL, CTL, and CTL*.

In this chapter we discussed three temporal logics. LTL is a logic where a formula holds if it holds for all paths starting from the initial state. CTL and CTL* considers a branching notion of time where formulas are defined over infinite trees. CTL and CTL* allows the expression of possibilities. It is possible to express that reaching a state is *possible*, for instance $\mathbf{EG}p$ means that there is the possibility that p holds globally. Note that there is no guarantee for p to hold at all! The three logic have different expressive powers. These differences are summarised in Figure 4.10. CTL* is the most expressive logic as it is more than CTL and LTL. CTL and LTL are incomparable. There are LTL formulas for which no CTL equivalent formulas exists and vice versa.

4.7 Exercises

Exercise 4.7.1. Consider a circuit with one input bit and one output bit. Let use i to denote the boolean value of the input and o the boolean value of the output. Write an LTL formula for each one of the following statements:

1. The system shall never output two successive 1's.
2. Whenever the input bit is 1, the output will be 1 in at most two cycles.
3. Whenever the input bit is 1, the output bit does not change in the next cycle.
4. The output is 1 infinitely often.

Exercise 4.7.2. Suppose we have two processes (*A* and *B*) connected to a single *Queue*. These processes execute several tasks, and from time to time they want to write a message in the queue. Present a set of atomic propositions – try to minimise the number of propositions – that are needed to describe the following LTL formulas and give the corresponding LTL formulas.

1. Mutual exclusion, that is, only one process at a time can write to the queue.
2. Finite time of usage, that is, a process can write to the queue only for a finite amount of time.
3. Absence of individual starvation, that is, if a process wants to write to the queue, it eventually is able to do so.
4. Absence of blocking, that is, a process can always request to write to the queue. Note that here the assumption that processes on all paths from time to time want to write to the queue is important.
5. Alternating access, that is, processes must strictly alternate in writing to the queue.

4.8 Solution to exercises

1. Formally defining the Kripke structure means giving definitions for the different components of the definition. There are two variables, therefore the set of atomic propositions is $AP = \{\texttt{IsCooking}, \texttt{IsOpen}\}$. Each proposition is a Boolean. It can be true (1) or false (0). The set of states is the set of all possible pairs of Booleans, that is, $S = \mathbb{B}^2$. The first element of the pair represents variable

`IsOpen` and the second element of the pair variable `IsCooking`. The initial state is when the door is open and the oven is not cooking, so $S_0 = \{10\}$. The labelling function must be defined for each state. This function returns the set of atomic propositions that are true in a state. For the oven example, we have the following:

- $L(00) = \emptyset$
- $L(01) = \{\texttt{IsCooking}\}$
- $L(10) = \{\texttt{IsOpen}\}$
- $L(11) = \{\texttt{IsOpen}, \texttt{IsCooking}\}$

2. The set of atomic propositions is unchanged,$AP = \{\texttt{IsCooking}, \texttt{IsOpen}\}$. There is one initial state encoded as $\mathcal{S}_0 = \texttt{IsOpen} \wedge \neg\texttt{IsCooking}$. The transition relation, noted $\mathcal{T}(\texttt{IsOpen}, \texttt{IsCooking}, \texttt{IsOpen}', \texttt{IsCooking}')$, is encoded as the following disjunction:

$$\begin{array}{ll}
 & \texttt{IsOpen} \wedge \neg\texttt{IsCooking} \wedge \neg\texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \\
\vee & \neg\texttt{IsOpen} \wedge \neg\texttt{IsCooking} \wedge \texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \\
\vee & \neg\texttt{IsOpen} \wedge \neg\texttt{IsCooking} \wedge \neg\texttt{IsOpen}' \wedge \texttt{IsCooking}' \\
\vee & \neg\texttt{IsOpen} \wedge \texttt{IsCooking} \wedge \neg\texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \\
\vee & \neg\texttt{IsOpen} \wedge \texttt{IsCooking} \wedge \texttt{IsOpen}' \wedge \neg\texttt{IsCooking}'
\end{array}$$