

Temporal Logic and Model Checking

June 2, 2008

1 Temporal Logic

Temporal logic is an instance of modal logic, which provides a language to specify temporal properties of sequences of events. In particular, it can be used to describe (desirable or undesirable) temporal properties of systems, that one wants to verify.

1.1 Transition Systems

Transition systems provide an abstraction of several entities that can be object of verification, such as digital circuits, software modules, protocol specifications and so on.

Definition 1 A *transition system* is a tuple $\mathcal{T} = (S, I, \delta, P, \sigma)$, where:

- S is a finite set of *states*;
- $I \subseteq S$ is the set of *initial* states;
- $\delta \subseteq S \times S$ is the (total) *transition relation*;
- P is a finite set of *propositions*;
- $\sigma : S \rightarrow 2^P$ is the function associating each state some “observable” properties, i.e., propositions, that are true in that state.

As an example of transition system, Figure 1 shows a graphical representation. Edges outgoing from “nothing” individuate the (unique, in this case) initial states, edges between states represent transition relation elements and each p_i in state s_j represents an observable property such that $p_i \in \sigma(s_j)$. Observe that it is very similar to a finite state automaton, though its purpose is not acceptance or rejection of words.

Evolutions of a transition system start from initial states and proceed along *runs* of states, according to transition relation δ . Observe that, as δ is total, every state has always at least a successor. Formally:

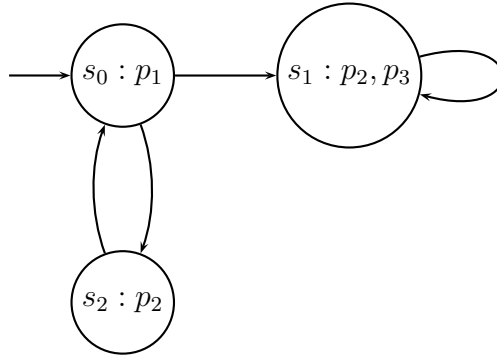


Figure 1: Graphical representation of a transition system.

Definition 2 A run ρ of \mathcal{T} is an infinite sequence $\rho = s_0, s_1, \dots$ where $s_0 \in I$ and $(s_i, s_{i+1}) \in \delta$.

And consequently, we naturally extend σ :

Definition 3 Given a run $\rho = s_0, s_1, \dots$ of \mathcal{T} , we define $\sigma(\rho) = \sigma(s_0), \sigma(s_1), \dots$

Note that in order to be able to specify eventual properties of runs as finite statements, we need to deal with infinite sequences. If runs were finite sequences of states and we wish to say “something eventually happens”, we should use an expression like “something happens after one step **or** after 2 steps **or**...” , thus, as in general we know not a priori whether or when such “something” happens, we would need an infinite statement. By defining runs as infinite sequences, we overcome this obstacle.

It is important to highlight that states themselves have no particular semantics. This is provided by observable propositions which represent relevant properties of the system. For this reason, temporal properties are not verified with respect to states but, rather, to their associated properties. Examples of common relevant properties of $\sigma(\rho)$ include:

- some bad property never holds –e.g., no state where a resource is free and taken is reached;
- some good property always holds (invariant);
- if some property p is true, then some other property p' will eventually be true (eventuality).

Now, we show a formal logic that allows for specifying statements like these.

1.2 Propositional Linear Time Logic: LTL

This is one of the simplest types of temporal logic. LTL is an extension of propositional logic with temporal operators. Statements represent properties of sequences of propositions and

are generated by using propositions, temporal and boolean connectors.

1.2.1 Syntax

Definition 4 The syntax of LTL is defined over a set of propositional symbols P . An LTL formula is defined as follows:

1. every atomic proposition $p \in P$ is a formula;
2. boolean combinations of formulas are formulas;
3. if $\varphi, \psi \in LTL$ then:
 - $X\varphi \in LTL$ (φ is true at next state);
 - $\varphi U \psi \in LTL$ (φ holds until ψ is true).

Temporal operators X and U are called *next* and *until*, respectively. This set of operators is minimal but “complete”, in a sense that will be soon defined.

1.2.2 Semantics

LTL formulas are evaluated over runs.

Definition 5 Let $\rho = s_0, s_1, \dots$ be a run of \mathcal{T} and let $\rho|_i = s_i, s_{i+1}, \dots$ represent the (infinite) suffix of ρ starting from (and including) s_i .

$\rho \models \varphi$ (ρ satisfies φ) is defined inductively as follows:

- $\rho \models p$, for $p \in P$, iff $p \in \sigma(s_0)$;
- extension to boolean combinations of formulas is obvious;
- $\rho \models X\psi$ iff $\rho|_1 \models \psi$;
- $\rho \models \phi U \psi$ iff $\exists k \geq 0$ s.t. $\rho|_k \models \psi$ and $\forall 0 \leq i \leq k$ $\rho|_i \models \phi$;

First three cases are pretty intuitive. As for the *until* operator, Figure 2 provides a graphical representation of a run satisfying $\varphi U \psi$, where state names are removed as irrelevant. Observe that the existence of a state satisfying ψ is necessary in order for the formula to be satisfied.

Definition 6 A transition system \mathcal{T} satisfies an LTL formula φ , $\mathcal{T} \models \varphi$, iff every run ρ of \mathcal{T} is such that $\rho \models \varphi$.

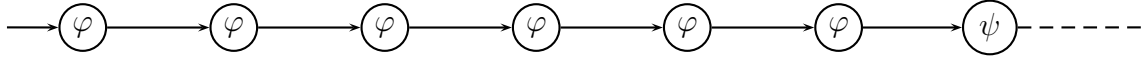


Figure 2: An example of run satisfying $\varphi U \psi$.

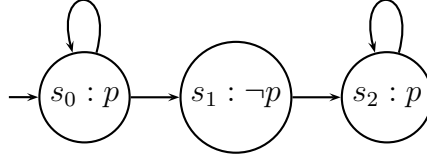


Figure 3: Transition system of Example 1.

1.2.3 Additional Operators

Additional useful operators can be derived by X and U :

- $F\varphi \doteq trueU\varphi$ (φ eventually holds);
- $G\varphi \doteq \neg F\neg\varphi$ (φ always holds);
- $\varphi W\psi \doteq (\varphi U\psi) \vee G\varphi$ (φ waits until ψ).

Example 1 Figure 3 shows an example of transition system \mathcal{T} , with only one proposition p . Consider the temporal property $\varphi = FGp$. Clearly, $\mathcal{T} \models FGp$ as every run starting from s_0 will be eventually such that all of its states satisfy p (\mathcal{T} can loop in either s_0 or s_2 , both satisfying p). Now, consider the temporal formula $\varphi' = G(p \rightarrow F(\neg p))$. Informally, it can be rephrased as: *it is always the case that from a state satisfying p a state satisfying $\neg p$ is eventually reached*. Since \mathcal{T} can loop in state s_0 , φ' is not satisfied ($\mathcal{T} \not\models \varphi'$) as there exists at least one run that satisfies p infinitely many often but never satisfies $\neg p$.

A natural question that arises is the following: *given a transition system \mathcal{T} and an LTL formula ϕ , is it the case that either $\mathcal{T} \models \phi$ or $\mathcal{T} \models \neg\phi$ holds?* The answer is *no*: it is not hard to find a formula ϕ such that both $\mathcal{T} \not\models \phi$ and $\mathcal{T} \not\models \neg\phi$ hold.

Example 2 A classical example of LTL application involves protocols for resource management. Assume to have two processes, represented by transition systems, which may access a same resource. In order for concurrent processes to execute correctly, some properties concerning the resource accesses need be guaranteed. Consider the set of propositions $\{req_1, req_2, owns_1, owns_2\}$, where req_i stands for “process i has requested the resource” and $owns_i$ stands for “process i owns the resource”. Some of the typical properties required to guarantee correctness of concurrent process executions are:

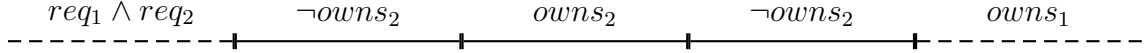


Figure 4: Representation of a run satisfying the “precedence property”.

- $G\neg(owns_1 \wedge owns_2)$ (“system invariant”): it is never the case that both processes own the resource at the same time;
- $G(req_1 \rightarrow Fowns_1)$ (“response property”): it is always the case that if process 1 requests the resource, it eventually owns it;
- $GF(req_1 \wedge \neg(owns_1 \vee owns_2)) \rightarrow (GFowns_1)$ (“strong fairness condition”): if it is true infinitely often that process 1 has requested the resource when it was free, then process 1 owns the resource infinitely often;
- $G((req_1 \wedge req_2) \rightarrow (\neg owns_2 W(owns_2 W(\neg owns_2 W owns_1))))$ (“precedence property”): whenever both processes compete for resource, process 2 will be granted the resource at most once before it is granted to process 1 (see Figure 4).

1.3 On the choice of temporal operators

A relevant question that may arise about the LTL definition provided above is *why the particular operators X and U have been chosen*. In other words, one may argue that the selected operators are somehow arbitrary.

Given a transition system \mathcal{T} , consider the logic FO^{temp} (with explicit access to time), defined as follows:

- assume a structure $\langle \mathbb{N}, \leq \rangle$ is defined, where \leq is a total order relation over \mathbb{N} ;
- for each $p \in P$ (propositions of \mathcal{T}), R_p is a unary relation such that: $R_p(i)$ is true $\Leftrightarrow p \in \sigma(s_i)$, where σ is the mapping from \mathcal{T} ’s states to propositions.

Such a logic allows to talk about temporal properties of \mathcal{T} , by explicitly referring to (discrete) time and avoiding the use of temporal operators. For example “ p until q ” can be encoded as:

$$\exists k (R_q(k) \wedge \forall i (0 \leq i \leq k \rightarrow R_p(i)))$$

Similarly, one can encode other temporal properties. The following result holds:

Theorem 7 [Kemp’s Theorem] $LTL \equiv FO^{temp}$.

The proof for $LTL \subseteq FO^{temp}$ is simple, while the converse inclusion, $FO^{temp} \subseteq LTL$, is non-trivial. The theorem shows that LTL is “complete” in the sense that it has exactly the same expressive power as FO^{temp} .

2 Application: Model Checking

The *model checking* problem (for finite transition systems) can be formally stated as follows:

given a transition system \mathcal{T} and a temporal formula φ , check whether $\mathcal{T} \models \varphi$.

That is, *does each run of \mathcal{T} satisfy φ ?* To answer this question, an automata-based approach is typically adopted (though, different ones do exist). Note that, in general, the temporal formula φ can be written in several temporal logics. However, here we refer to LTL.

First, observe that verifying whether $\mathcal{T} \models \varphi$ corresponds to check that *there is no run ρ of \mathcal{T} violating φ , i.e., such that $\rho \models \neg\varphi$* . Now, given an LTL formula φ assume to be able to build a finite automaton A_φ that accepts exactly all the *infinite sequences of sets of propositions* that satisfy φ (recall that semantics of a run ρ is actually given by $\sigma(\rho)$). The adopted strategy is as follows:

- from φ build $A_{\neg\varphi}$;
- check that there is no run of \mathcal{T} accepted by $A_{\neg\varphi}$.

Informally, we *execute \mathcal{T} against $A_{\neg\varphi}$* . If $A_{\neg\varphi}$ accepts some run of \mathcal{T} then a counterexample is found and we know *why $\mathcal{T} \not\models \varphi$* , otherwise we conclude $\mathcal{T} \models \varphi$. In other words, we execute the cross product $\mathcal{T} \times A_{\neg\varphi}$ and check whether it is empty. If so then $\mathcal{T} \models \varphi$, otherwise $\mathcal{T} \not\models \varphi$.

In order to show the strategy in details, we need the following basics:

1. extend automata to infinite words (a.k.a. ω -words): when is an infinite word accepted?
2. find an actual algorithm that given φ builds the corresponding automaton A_φ .

2.1 Automata over infinite words: Büchi Automata

A Büchi automaton is similar to an FSA but has a different accepting condition that allows to deal with infinite words. In particular, an ω -word is accepted iff, when provided as input, makes the automaton *go infinitely often through an accepting state*. In details:

Definition 8 A Büchi automaton is a tuple $B = (Q, I, \delta, F)$ over a finite input alphabet Σ , where:

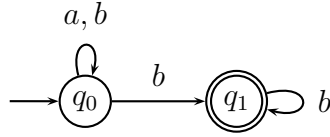


Figure 5: Nondeterministic Büchi automaton with no deterministic counterpart.

- Q is the set of states;
- $I \subseteq Q$ is the set of initial states;
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition function;
- $F \subseteq Q$ is the set of accepting states.

Definition 9 A *run* of B over an ω -word $a_0a_1\dots \in \Sigma^\omega$ is an infinite sequence of states q_0, q_1, \dots where $q_0 \in I$, and $(q_i, a_{i+1}, q_{i+1}) \in \delta$.

Definition 10 A run is *accepting* iff $\exists q \in F$ s.t. $q_i = q$ for infinitely many i .

Definition 11 Given a Büchi automaton B , $\mathcal{L}(B)$ is the language accepted by B .

Definition 12 A set of ω -words is called an *ω -regular language* iff it equals $\mathcal{L}(B)$ for some Büchi automaton B .

2.2 Properties of Büchi Automata

Some relevant (and useful) properties of Büchi automaton.

1. *Emptiness* is decidable in linear time with respect to $|Q|$: one can enumerate strongly connected components and check that at least one reachable from I contains an accepting state. Note: it is only required that an accepting computation *exists*.
2. Unlike FSAs, *nondeterministic Büchi automata are more powerful than deterministic ones*. Figure 5 shows an example of a nondeterministic Büchi automaton accepting a language of ω -words where a occurs finitely many times. Double lined nodes represent accepting states. It can be shown that no deterministic automaton accepting the same language can be built.
3. Closure Properties:
 - Union: Proved using the classical construction, as in FSAs.

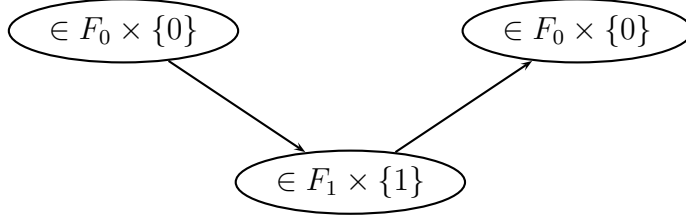


Figure 6: Final state “switch” in a Büchi automaton simulating a generalized one.

- Intersection: given two Büchi automata A_1 and A_2 we want to build a third automaton $A_1 \cap A_2$ such that $\mathcal{L}(A_1 \cap A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. In this case, the classical “cross” construction doesn’t work because the two automata are not guaranteed to go *simultaneously* through their accepting states.

To show that closure holds under intersection, we first need to introduce a *generalized acceptance condition*. Instead of a single set of accepting states, we consider a Büchi automaton with multiple accepting state sets F_1, \dots, F_m and, in order for an ω -word to be accepted, require that it goes infinitely often through all F_i . Based on this, from $A_i = (Q_i, I_i, \delta_i, F_i)$ ($i = 1, 2$) we define the *generalized* Büchi automaton $A_1 \cap A_2 = (Q_1 \times Q_2, I_1 \times I_2, \delta_1 \times \delta_2, F_1 \times Q_2, Q_1 \times F_2)$ which accepts exactly $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

In order to prove that the property holds, we need to show that a generalized Büchi automaton can be simulated by a Büchi automaton. This is true in general. Instead of the complete proof, however, we provide just an example which shows the basic intuition behind it. Let $A = (Q, I, \delta, F_0, F_1)$ be a generalized Büchi automaton. Define the Büchi automaton $A' = (q \times \{0, 1\}, I \times \{0\}, \delta', F_0 \times \{0\})$, where:

$$\delta'((q, i), a) = \{(t, i) \mid t \in \delta(q, a), q \notin F_i\} \cup \{(t, (i + 1) \bmod 2) \mid t \in \delta(q, a), q \in F_i\}$$

It can be seen that, due to δ' , in order for A' to go infinitely often through $F_0 \times \{0\}$ –i.e., to accept– it needs to go infinitely often through $F_1 \times \{1\}$, as depicted in Figure 6.

- Closure under complementation holds but cannot be proven by “determinization”. The proof is non-trivial and will be omitted. However, the complexity of building the complemented automaton is $2^{O(n \log(n))}$, where n is the number of states in the original automaton.

2.3 From LTL to Büchi Automata

Now, let see how, starting from an LTL formula φ , we can build a Büchi automaton A_φ that accepts exactly all and only runs satisfying φ . Recall that the semantics of a run is given by the *sets of propositions* its states are labeled with. So, if P is the set of propositions runs refer

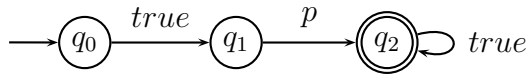


Figure 7: Büchi automaton for Xp .

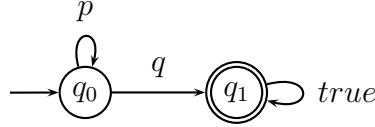


Figure 8: Büchi automaton for pUq .

to, the automaton to be built has input alphabet 2^P . Consequently, automaton transitions are required to be labeled by subsets of P . For convenience, we adopt a different convention and use propositional formulas: the subsets of P a formula represents corresponds to exactly all models of the formula. For instance, formula $\phi = (p \vee q)$ represents sets $\{p\}$, $\{q\}$ and $\{p, q\}$. Figures 7 and 8 show two examples of automata that accept ω -words satisfying Xp and pUq , respectively. Observe that labels p , q in both Figures are *formulas*.

Now, we take a look at formal procedures to construct the Büchi automaton. A first, naive, approach is based on a syntax-directed procedure which applies structural recursion on φ , exploiting the following basic translation schemas:

- for $\varphi \vee \psi$ see Figure 9;
- for $\neg\varphi$ build A_φ and take its complement A_φ^C ;
- for $X\varphi$ see Figure 10;
- for $\varphi U\psi$ we omit the schema as it is more complicated.

Unfortunately, this approach has a major drawback: construction of A_φ^C yields an exponential blowup in the number of states and the resulting automaton contains 2^{2^m} states, where $m = |\varphi|$ and the number of nested exponents is the number of occurrences of “ \neg ”. This can be avoided by using a direct translation with no structural recursion, which avoids multiple complementation steps. The resulting automaton has $2^{|\varphi|}$ states. In order to show

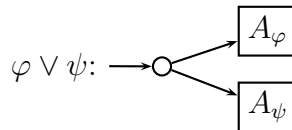


Figure 9: Büchi automaton for $\varphi \vee \psi$.

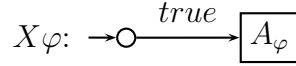


Figure 10: Büchi automaton for $X\varphi$.

its construction, we refer to the so-called set of φ 's *clauses* $\mathcal{Cl}(\varphi)$ ¹, containing all and only subformulae of φ and their negation ($\neg\neg\psi = \psi$). Automaton A_φ consists of:

States. Each state of A_φ is a set of clauses in $\mathcal{Cl}(\varphi)$, *satisfied by all accepted (infinite) runs starting in that state*. To enforce this, each state q must satisfy a *consistency criterion*, i.e., $\forall\psi, \psi_1, \psi_2 \in \mathcal{Cl}(\varphi)$:

- either $\psi \in q$ or $\neg\psi \in q$, not both;
- $\psi_1 \vee \psi_2 \in q$ iff $\psi_1 \in q$ or $\psi_2 \in q$;
- if $\psi_1 U \psi_2 \in q$ then $\psi_1 \in q$ or $\psi_2 \in q$;
- if $\psi_1 U \psi_2 \notin q$ then $\psi_2 \notin q$.

Initial States. All and only states containing φ ;

Transitions. $(q, s, q') \in \delta$ iff:

1. $s = q \cap P$, i.e, s is the set of propositions in q ;
2. q' contains ψ iff $X\psi \in q$;
3. if $\psi_1 \cup \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1 \cup \psi_2 \in q'$;
4. if $\psi_1 \cup \psi_2 \notin q$ and $\psi_1 \in q$ then $\psi_1 \cup \psi_2 \notin q'$.

Acceptance condition. Consider all formulas of the form $fUg \in \mathcal{Cl}(\varphi)$ and refer to them as $\psi_1^i U \psi_2^i$, for $1 \leq i \leq k$. We define a set of accepting states F_1, \dots, F_k , where F_i contains all states q such that $\psi_1^i U \psi_2^i \notin q$ or $\psi_2^i \in q$.

As for the complexity of computing A_φ , observe that explicit computation of A_φ is not required. In fact, given a state $q \in Q$ and $s \in 2^P$, one can compute the set of possible next states directly from φ . Therefore, the complexity is *PSPACE*.

Example 3 We show how to construct the automaton A_φ of Figure 11, for $\varphi = pUq$.

- we have $\mathcal{Cl}(\varphi) = \{pUq, \neg(pUq), p, \neg p, q, \neg q\}$;
- from $\mathcal{Cl}(\varphi)$, we derive the following set of consistent states:

$$Q = \{\{p, \neg q, pUq\}, \{p, \neg q, \neg(pUq)\}, \{\neg p, q, pUq\}, \{p, q, pUq\}, \{\neg p, \neg q, \neg(pUq)\}\};$$

¹Here *clause* has not the usual meaning.

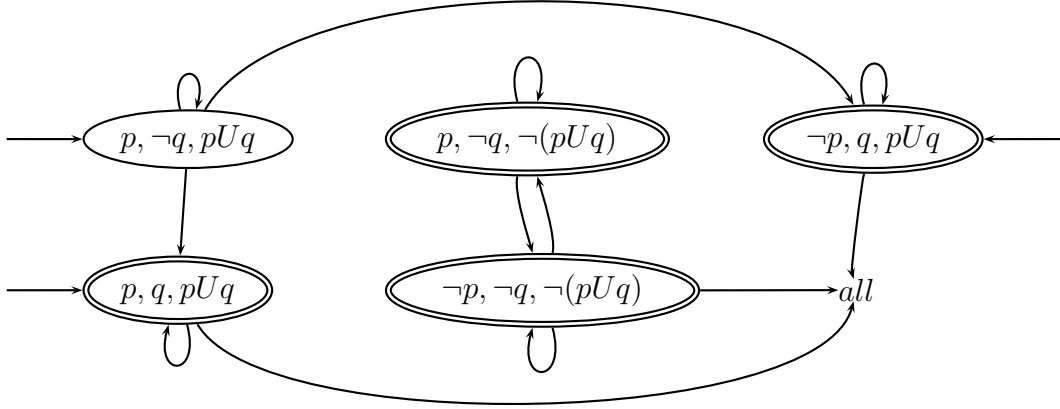


Figure 11: Büchi automaton resulting from direct translation of pUq .

- initial states are all those containing pUq and are individuated in Figure 11 by incoming edges from “nothing”;
- transitions are as in Figure 11. An edge from state s to state all means that there exists a transition from s to state s' , for every s' ;
- as pUq is the only formula of the form fUg in $\mathcal{Cl}(\varphi)$, we have only one set of accepting states F_{pUq} whose states are represented in Figure 11 by double lined nodes.

2.4 Putting things together

Recall that our original problem was Model Checking for finite transition systems:

- we are given a finite transition system \mathcal{T} with observable properties P ;
- we have an LTL formula φ over P ;
- we want to check whether every run of \mathcal{T} satisfies φ . Clearly, this is equivalent to check that there exists no run of \mathcal{T} satisfying $\neg\varphi$.

Our plan for solving the problem was: generate $A_{\neg\varphi}$ from φ and execute \mathcal{T} against it, to check whether any run is accepted by $A_{\neg\varphi}$. If so, then $\mathcal{T} \not\models \varphi$, otherwise $\mathcal{T} \models \varphi$.

Consider a run $\rho = s_0, s_1, \dots$ of \mathcal{T} , accepted by $A_{\neg\varphi}$. To accept, $A_{\neg\varphi}$ must go through some run $\varrho = q_0, q_1, \dots, f, \dots, f, \dots$ such that $f \in F$. Now, observe that *if there exists a \mathcal{T} 's run accepted by $A_{\neg\varphi}$ then there exists a periodic run accepted by $A_{\neg\varphi}$* . To see this, consider runs in Figure 12, the upper representing a \mathcal{T} run accepted by $A_{\neg\varphi}$, and the lower being its corresponding accepting $A_{\neg\varphi}$ run. Due to state finiteness, for p sufficiently large, there exists a state s_n that occurs twice and, in both occurrences, “matches” a final state $f \in F$

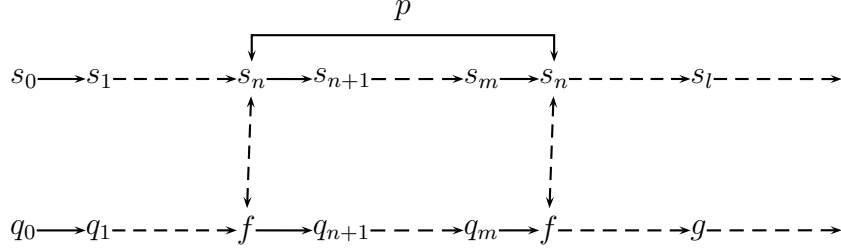


Figure 12: Non-periodic \mathcal{T} 's and $A_{-\varphi}$'s runs.

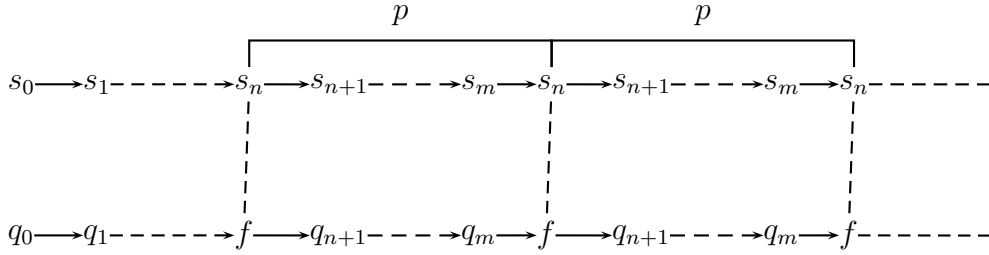


Figure 13: Periodic \mathcal{T} run, obtained from \mathcal{T} 's run of Figure 12, and its accepting $A_{-\varphi}$ run.

(see Figure). Therefore, we can derive from \mathcal{T} 's run another one which is periodic: starting from the first occurrence of s_n matching f , we simply concatenate the same portion of run s_n, \dots, s_m infinitely many times, as shown in Figure 13. Note that from s_n , transition to s_{n+1} and subsequent states are allowed in \mathcal{T} and yield the same respective transitions in $A_{-\varphi}$. Clearly, such run is still accepted as it induces a respective $A_{-\varphi}$ periodic run, where f occurs infinitely often.

Based on this, we can adopt the following nondeterministic algorithm, which searches for periodic runs of \mathcal{T} accepted by $A_{-\varphi}$:

1. start with $(s_0, q_0) \rightarrow (s, q)$;
2. if $q \in F$ guess that $f = q$ and $s_n = s$ or continue;
3. generate (s', q') from (s, q) , i.e., one possible successor state of \mathcal{T} and $A_{-\varphi}$, respectively;
4. if $(s', q') = (s_n, f)$ then *accept* else go to 2.

It can be seen that the algorithm accepts if and only if there exists a run of \mathcal{T} accepted by $A_{-\varphi}$.

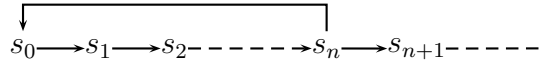


Figure 14: A run whose states allow to reach s_0 .

Concerning complexity, note that instead of \mathcal{T} , one can provide as input a specification $\mathcal{S}_{\mathcal{T}}$ (\mathcal{T} is typically exponential in $\mathcal{S}_{\mathcal{T}}$). Moreover, we need not to generate all \mathcal{T} 's transitions at a time, but we can do this *lazily*, while generating \mathcal{T} 's runs. This can be done in PSPACE. Similarly, given φ , $A_{\neg\varphi}$ need not be explicitly generated, but we can compute its transitions directly from φ , again, in PSPACE. Consequently, the above algorithm is in NPSPACE, that is PSPACE. The corresponding deterministic algorithm turns out to be a depth-first search with some bookkeeping and has time complexity $O(2^{|\varphi|}|\mathcal{T}|)$.

2.5 Branching Time Logics

So far, given a transition system we have considered only properties of its runs, expressed in LTL. This approach does not allow for talking about *interactions between runs*. We could not express properties such as “there is always a way to go back to the initial state”. This property requires that every state of every run has s_0 as a *possible successor*. So, for instance, take run of Figure 14 and assume that, for $i \geq 0$, s_i has s_0 as a possible successor (only one edge is depicted in the Figure), i.e., $(s_i, s_0) \in \delta$ for $i \geq 0$. Even if $s_i \neq s_0$ for $i \geq 0$, the property is satisfied. The point here is not whether the run *goes* through s_0 but, rather, whether it *can* do that. It should be clear that LTL is not powerful enough to represent such properties. Indeed, to do so, we need a representation of runs which takes into account *all possible* successors of a state. This is a *tree* model of time.

Consider the transition system of Figure 15. Its evolution can be represented by the infinite tree of Figure 16, which, intuitively, corresponds to the unfolding of the transition system. Clearly, it contains more information than a single run, as it tells us how a run *can* be continued. So, referring to such tree, we can express statements such as:

- for every run starting at a given point something happens, or
- there exists a run starting at a given point such that something will happen.

That is, we gain the possibility to *quantify over runs* (*Branching Time*). According to how we combine temporal operators and path quantifiers A (for all) and E (there exists), we get two branching time logics: CTL and CTL* (*Computation Tree Logics*).

2.5.1 CTL*

CTL* is the augmentation of LTL with path quantifiers and no restriction on how operators

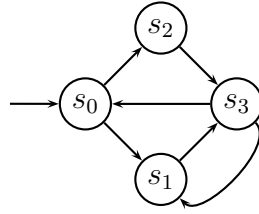


Figure 15: A transition system.

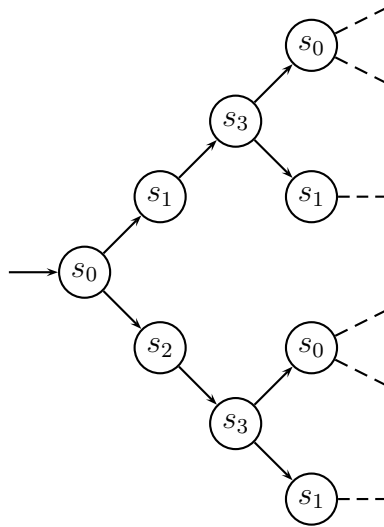


Figure 16: Unfolding of transition system of Figure 15.

can be nested. It contains both LTL and CTL (see below). Note that an LTL formula φ corresponds to a CTL* formula $A\varphi$. Some examples of CTL* formulas are:

1. AFp , in every run p has to hold eventually. It is satisfied, e.g., by tree of Figure 17;
2. AGp , equivalent to Gp in LTL;
3. EGp , there exists some path where p is always satisfied. See Figure 18 for a model of this formula;

An interesting result about CTL* is that Model Checking is PSPACE-complete. The automaton approach can also deal with trees, instead of runs, by extending Büchi automata. We get the same complexity as in the case of LTL. Unfortunately, it is still too high. To get better complexity results, we need to reduce the expressive power of the logic.

2.5.2 CTL

CTL formulas are very similar to CTL*'s but require *each temporal operator F, G, X, U , to be preceded by A or E* . Previous examples of CTL* formulas are also valid for CTL. However, there are examples of CTL* formulas not in CTL, such as *fairness assumption*:

$$A(GFp \rightarrow GFq) \tag{1}$$

Note that the formula can be expressed in LTL. This is not in CTL because path quantifiers are needed before temporal operators, as in:

$$AGAFp \rightarrow AGAFq \tag{2}$$

which is not the same as formula (1). It can be shown that formula (1) *has no CTL reformulation*, thus implying that *CTL* is strictly more powerful than CTL*. In fact, CTL* contains both LTL and CTL. In addition, CTL and LTL are incomparable.

Concerning complexity, we get that Model Checking for CTL is $O(|\varphi||\mathcal{T}|)$. The intuition why it is more efficient is that when verifying a property, one can restrict only to *local* properties. That is, as a consequence of the syntactic restriction, one can look only at each state and its “neighbors”, without needing to go arbitrarily deep in the tree. It is very restricted but in many cases is enough, e.g.:

- $AG(request \rightarrow AFgrant)$, non-starvation property;
- $AG(n \rightarrow EXt)$, non-blocking property.

Summarizing:

- $CTL^* = LTL + A, E$;

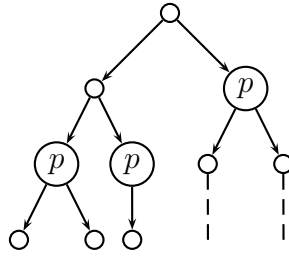


Figure 17: A tree satisfying AFp .

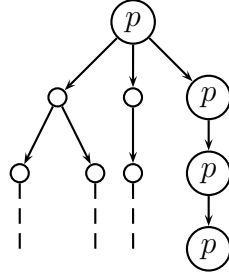


Figure 18: A tree satisfying EGp .

- CTL is a restricted version of CTL^* , with improved model checking complexity: $O(|\varphi||\mathcal{T}|)$;
- CTL is orthogonal to LTL, i.e., they are incomparable;
- CTL^* is more expressive than LTL and CTL.