

5. LTL, CTL

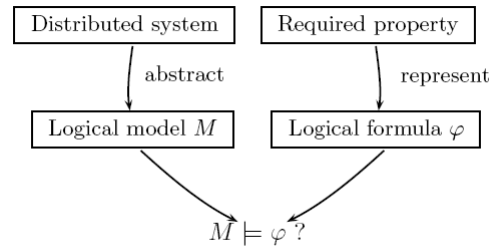
- Last part:
 - Alloy
 - logic
 - language, static/dynamic models
 - SAT solvers
- Today:
 - Temporal Logic (LTL, CTL)

Overview

- **How can we model check of a program or system?**
 - Modeling
 - Build a model of the program, system ???
 - Specification
 - Describe requirement properties using a form of Temporal Logic
 - Verification
 - Automatically (semi-automatic)

Model checking: definition

- Given a model M and a formula φ , model checking is the problem of verifying whether or not φ is true in M (written $M \models \varphi$).



Objectives

- Understand why temporal logic can be a useful formalism for specifying properties of concurrent/reactive systems.
- Understand the intuition behind Computation Tree Logic (CTL) – the specification logic used e.g., in the well-known (Nu)SMV model-checker.
- Be able to confidently apply Linear Temporal Logic (LTL) – the specification logic used in e.g., TLA+ and SPIN – to specify properties of systems.
- Understand the formal semantics of LTL.

Reasoning about Executions

- We've seen specifications that are about individual program states
 - e.g., assertions, invariants, **scoped** traces
- Sometimes we want to reason about the relationship between multiple states
 - Must one state always precede another?
 - Does seeing one state preclude the possibility of subsequently seeing another?
- We need to shift our thinking from states to paths in the state space

Examples

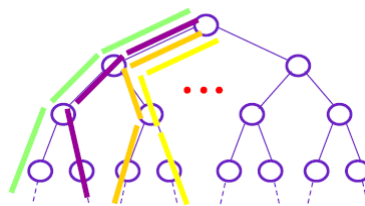
- A use of a variable must be preceded by a definition
- When a file is opened it must subsequently be closed
- You cannot shift from drive to reverse without passing through neutral
- The program will eventually terminate
- Two concurrent processes will each be able to enter its critical sections
- Will a print request be served in the future by a Printer Manager

Why Use Temporal Logic?

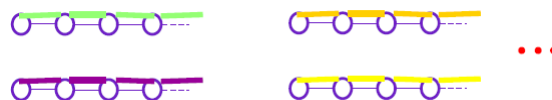
- Requirements of concurrent, distributed, and reactive systems are often phrased as constraints on *sequences of events or states* or constraints on *execution paths*.
- Temporal logic provides a formal, expressive, and compact notation for realizing/modeling such requirements.
- The temporal logics we consider are also strongly tied to various computational frameworks (e.g., automata theory) which provides a foundation for building verification tools.

Linear Time Logic we want to understand

- Restrict path quantification to "ALL" (no "EXISTS")



- Reason in terms of branching traces instead of branching trees



Overview

- *Temporal Property*: A property with time-related operators such as “**invariant**” or “**eventually**”
- *Invariant(p)*: is true in a state if property p is true in every state on all execution paths starting at that state
 - The Invariant operator has different names in different temporal logics:
 - **G, AG, □** (“goal” or “box” or “forall”)
- *Eventually(p)*: is true in a state if property p is true at some state on every execution path starting from that state
 - **F, AF, ◇** (“diamond” or “future” or “exists”)

An example concurrent program

- A simple concurrent mutual exclusion program
 - Two processes execute asynchronously
 - There is a shared variable **turn**
 - Two processes use the shared variable to ensure that they are not in the critical section at the same time
 - Can be viewed as a “fundamental” program: any bigger concurrent one would include this one
- ```
10: while True do
11: wait(turn = 0);
 // critical section
12: turn := 1;
13: end while;

|| // concurrently with

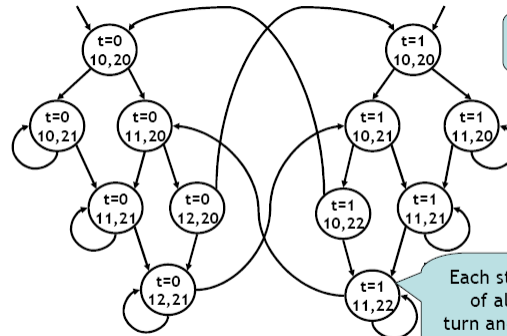
20: while True do
21: wait(turn = 1);
 // critical section
22: turn := 0;
23: end while;
```

## Reachable states of the program

```
10: while True do
11: wait(turn = 0);
 // critical section
12: turn := 1;
13: end while;
```

|| // concurrently with

```
20: while True do
21: wait(turn = 1);
 // critical section
22: turn := 0;
23: end while
```



Next: formalize  
this intuition ...

Each state is a valuation  
of all the variables:  
turn and the two program  
counters for two processes

## Properties of the program

- Example: "In all the reachable states (configurations) of the system, the two processes are *never in the critical section at the same time*"
  - Equivalently, we can say that
    - $Invariant(\neg(pc1=12 \wedge pc2=22))$
- Also: "Eventually the first process enters the critical section"
  - $Eventually(pc1=12)$
- "pc1=12", "pc2=22" are atomic properties

# Transition Systems

---

- In model checking the system being analyzed is represented as a **labeled transition system**

$$T = (S, I, R, L)$$

- Also called a **Kripke Structure**
- $S$  = Set of states // standard FSM
- $I \subseteq S$  = Set of initial states // standard FSM
- $R \subseteq S \times S$  = Transition relation // standard FSM
- $L: S \rightarrow P(AP)$  = Labeling function // this is new!
- $AP$ : Set of **atomic propositions** (e.g., "x=5")
  - Atomic propositions capture basic properties
  - For software, atomic props depend on variable values
  - The labeling function labels each state with the set of propositions true in that state

# LTL

---

- In LTL time is
  - implicit,
  - discrete,
  - has an initial moment with no predecessors, and
  - infinite in the future
- The model of LTL formula is infinite sequence of states  $\pi: s_0, s_1, s_2, \dots$

## LTL (cont'd)

- Elements:
  - Atomic propositions **AP**
  - Boolean operators  $\wedge \vee \neg \rightarrow$
  - Temporal operators **G F X U R**
- Syntax:  
 $\phi ::= P$ 
  - |  $\neg \phi$  |  $\phi \wedge \psi$  |  $\phi \vee \psi$  |  $\phi \rightarrow \psi$
  - | **G**  $\phi$  | **F**  $\phi$  | **X**  $\phi$  |  $\phi$  **U**  $\psi$  |  $\phi$  **R**  $\psi$

## Semantic Intuition

- **G**  $f$  - always  $f$
- **F**  $f$  - eventually  $f$
- **X**  $f$  - next state
- $f$  **U**  $r$  - until
- $f$  **R**  $r$  - releases

## Semantic

---

- Semantic is given with respect to path
  - $\pi = s_0 s_1 s_2 \dots$
- Suffix of trace starting at  $s_j$ 
  - $\pi_j = s_j s_{j+1} s_{j+2} \dots$
- A system satisfies an LTL formula  $f$  if **each** possible path through the system satisfy  $f$ .

## Semantic (cont'd)

---

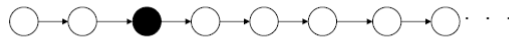
- $n \models a$       iff  $a \in s_0$  [ $a \in L(s_0)$ ]
- $n \models \neg\phi$     iff  $n \not\models \phi$
- $n \models \phi \wedge \psi$  iff  $n \models \phi$  and  $n \models \psi$
- $n \models \phi \vee \psi$  iff  $n \models \phi$  or  $n \models \psi$
- $n \models X\phi$     iff  $n_1 \models \phi$
- $n \models F\phi$     iff exists  $i \geq 0$   $n_i \models \phi$
- $n \models G\phi$     iff for all  $i \geq 0$   $n_i \models \phi$
- $n \models \phi U \psi$  iff exists  $i \geq 0$   $n_i \models \psi$   
and for all  $0 \leq j < n$ .  $n_j \models \phi$
- $n \models \phi R \psi$  iff for all  $j \geq 0$ , if for every  $i < j$   
not  $n_i \models \phi$  then  $n_j \models \psi$

## LTL Identities

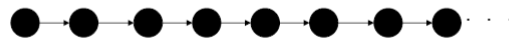
- $G \varphi = \neg F \neg \varphi$
- $F \varphi = (true \ U \ \varphi)$
- $\varphi \ R \ \psi = \neg(\neg \varphi \ U \ \neg \psi)$
  
- Every LTL formula  $f$  can be rewritten using only operators  $\neg \vee X \ U$   
This is important for implementing tools for verification.

## LTL Satisfiability Examples

○  $p$  does not hold      ●  $p$  holds



On this path:  $F p$  holds,  $G p$  does not hold,  $p$  does not hold,  
 $X p$  does not hold,  $X (X p)$  holds,  $X (X (X p))$  does not hold



On this path:  $F p$  holds,  $G p$  holds,  $p$  holds,  
 $X p$  holds,  $X (X p)$  holds,  $X (X (X p))$  holds

## Practical patterns of specifications

---

- It is impossible to get to a state where *started* holds, but *ready* does not hold

$G \neg (\text{started} \wedge \neg \text{ready})$

- For any state, if a *request* (of some resource) occurs, then it will eventually be *acknowledged*

$G(\text{requested} \rightarrow F \text{ acknowledged})$ .

- A certain process is *enabled* infinitely often on every computation path

$GF \text{ enabled}$

## Practical patterns of ...

---

- Whatever happens, a certain process will eventually be permanently *deadlocked*

$FG \text{ deadlock}$

- If the process is enabled infinitely often, then it runs infinitely often

$GF \text{ enabled} \rightarrow GF \text{ running}$ .

- An upwards travelling lift at the *second floor* does not change its *direction* when it has passengers wishing to go to the *fifth floor*

$G(\text{floor2} \wedge \text{directionup} \wedge \text{ButtonPressed5} \rightarrow (\text{directionup} U \text{floor5}))$

## Practical patterns of ...

---

- From any state *it is possible* to get to a restart state (i.e., there is a path from all states to a state satisfying restart).

Not expressible! Why??

---

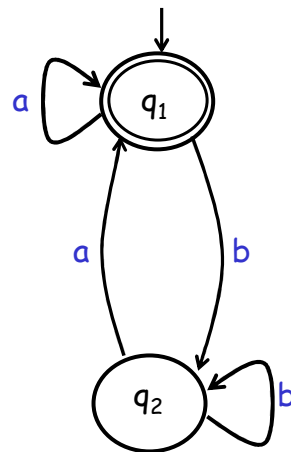
## LTL Verification

## State Sequences as Words

- Let  $AP$  be the finite set of atomic propositions of the formula  $f$ .
- Let  $\Sigma = 2^{AP}$  be the alphabet over  $AP$ .
- Every sequence of states is an  $\omega$  word in  $\Sigma^\omega$ 
  - $\alpha = P_0, P_1, P_2, \dots$  where  $P_i = L(s_i)$ .
- A word  $\alpha$  is a model of formula  $f$  iff  $\alpha \models f$
- Example: for  $f = p \wedge (\neg q \cup q)$   $\{p\}, \{\}, \{q\}, \{p, q\}^\omega$
- Let  $\text{Mod}(f)$  denote the set of models of  $f$ .
- Automata theory: automata are computational devices to determine whether a word belongs to language.

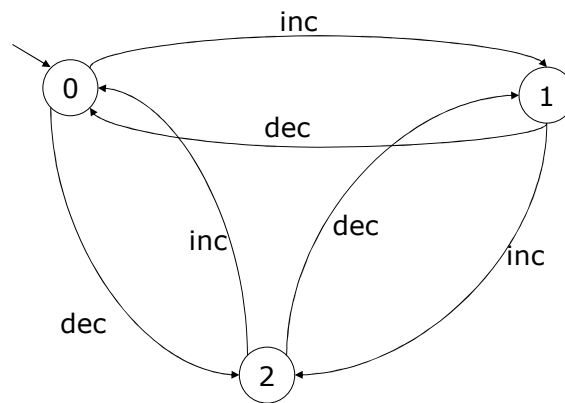
## Automata Theory

- Automaton  $A = (Q, \Sigma, \delta, I, F)$ 
  - $Q$  – set of states
  - $\Sigma$  – finite alphabet
  - $\delta$  – transition relation
  - $I$  – set of initial states
  - $F$  – set of acceptance states
- A run  $\rho$  of  $A$  on  $\omega$  word  $\alpha$   
 $\rho = q_0, q_1, q_2, \dots$ , s.t.  $q_0 \in I$  and  
 $(q_i, \alpha_i, q_{i+1}) \in \delta$
- The run  $\rho$  is accepting if  
 $\text{Inf}(\rho) \cap F \neq \emptyset$



## An Example

- An automaton for a Modula 3 counter with operations *inc*, *dec* ?

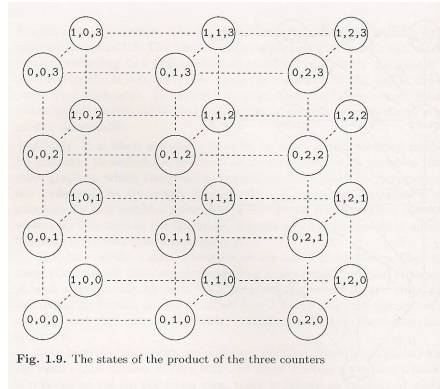
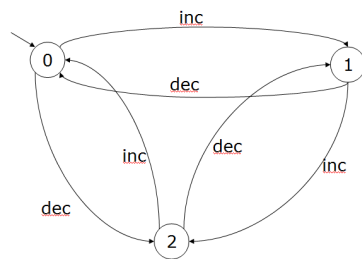


## Observation

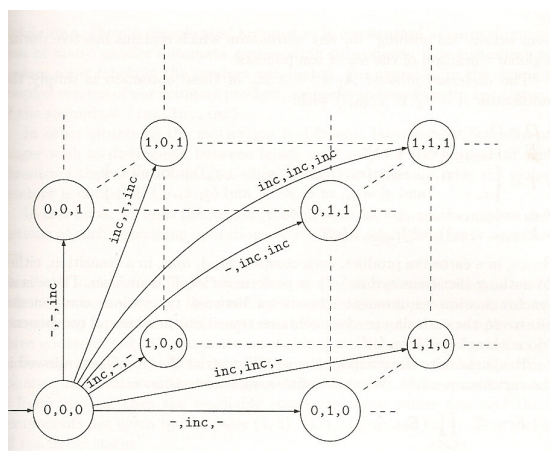
- We can use automata to model services like Counter, Printer Manager, ...
- When we deal with real-life systems, these are often broken up into components.
- We first model the components.
- Then, the global automaton is obtained from the component automata by having them to cooperate.

## An Example without synchronisation

- A system composed of a modulo 2, modulo 3 and modulo 4 counter.
- Means composing the three automata
- Possible states are:



## A few transition of the product



## An example with synchronization

- We forbid them to evolve independently.

Sync =  $\{(inc, inc, inc), (dec, dec, dec)\}$

and we can build the reachability automaton.

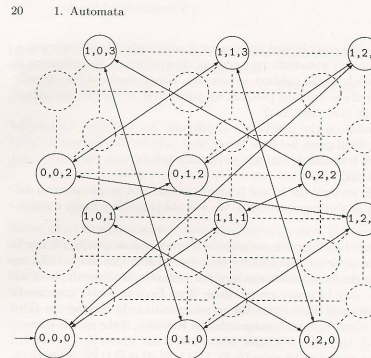


Fig. 1.11. The automaton  $\mathcal{A}_{ccc}^{comp}$  restricted to reachable states

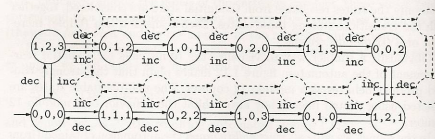


Fig. 1.12. The rearranged automaton  $\mathcal{A}_{ccc}^{comp}$

## Coming back to LTL verification

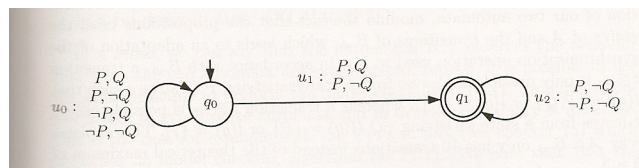
- LTL are path formulas. The viewpoint adopted will be language theory.
- Consider the formula  $p$ : **GFP**  
An execution  $q_0, q_1, \dots$  satisfying  $p$  must contain infinitely many positions  $q_x$  at which  $p$  holds.
- The execution is of the form  $((\neg P)^* . P)^\omega$
- An execution not satisfying  $p$  must, from a certain point on, only contain states satisfying  $\neg P$ .  $(\neg P + P)^* . (\neg P)^\omega$

## Basic Strategy

- Checking relies on the possibility to associate with each LTL formula  $p$  an  $\omega$ -regular expression describing the form imposed on an execution by its satisfaction of  $p$ .  
The “does  $A \models p$ ?” question then reduces to a “are all the executions of  $A$  of the form described by the regular expression.”
- Build  $B_{\neg p}$ . Strongly synchronize  $A$  and  $B_{\neg p}$ . Build the product automaton  $A \otimes B_{\neg p}$ .  
“does  $A \models p$ ?” reduces to “is the language of  $A \otimes B_{\neg p}$  empty?”

## Example

- Create an automata for  $B_{\neg \Phi}$  where  $\Phi$  is  $G(P \rightarrow \mathbf{X}FQ)$ . The negation of  $\Phi$  means that there exists an occurrence of  $P$  after which we will never encounter  $Q$ .



# The Automata A

- The model to be checked

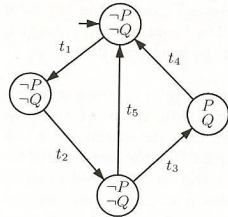
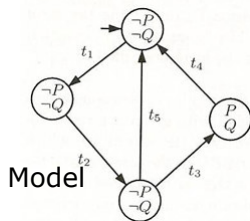
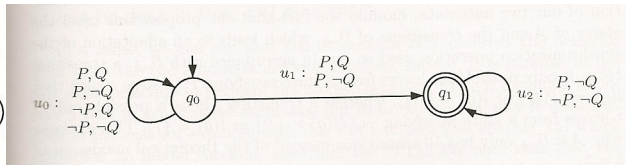


Fig. 3.2. Does the automaton A verify  $\phi : G(P \Rightarrow X FQ)$  ?

# The product automaton



Model

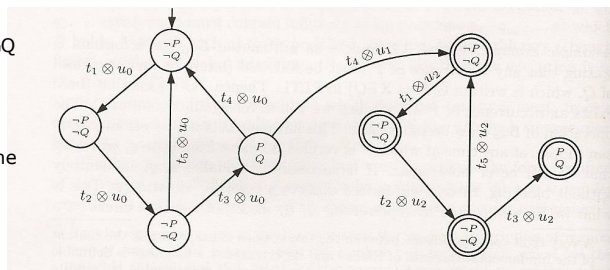


Property

Observations:  
 $t \otimes u_1$  if  $t$  leaves a state sat.  $P$   
 $t \otimes u_2$  if  $t$  leaves a state sat.  $\neg Q$

We can observe behaviors of A that are in  $A \otimes B \neg p$ .

If the language is not empty the property is not satisfied.

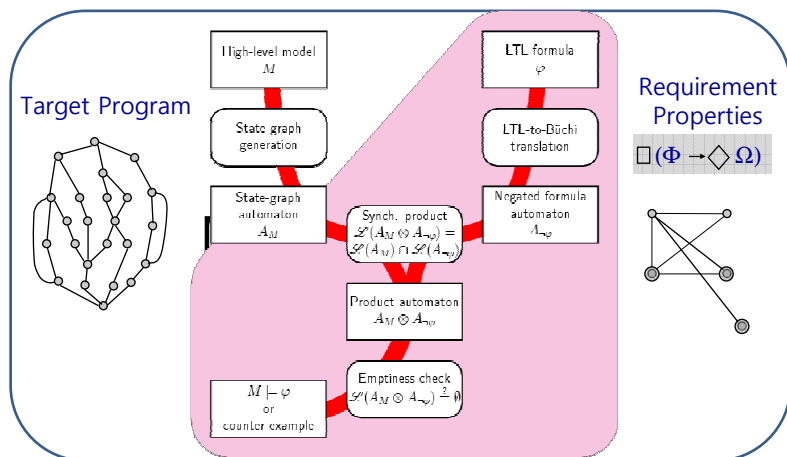


Product Automaton

## Automata for LTL Formula and GBA

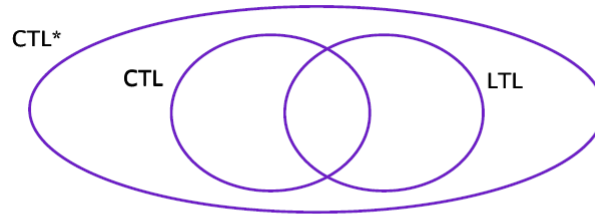
- Generalized Büchi Automaton (GBA)  $A$  is  
 $A = (Q, \Sigma, \delta, I, FT)$ , where  $FT$  is  
 $FT = \langle F_1, F_2, \dots, F_k \rangle$
- A run  $\rho$  is accepting if  $\text{Inf}(\rho) \cap F_i \neq \emptyset$   
for every  $1 \leq i \leq k$
- A set of models  $\text{Mod}(f)$  of formula  $f$  is  $\omega$ -regular and we  
can construct a GBA  $A$  for the formula s.t.  
 $\text{Mod}(f) = L(A(f))$ , where  $L(A(f))$  is a language accepted  
by Automata  $A$

## Process of Model Checking LTL



# CTL – Computational Tree Logic

---



- CTL is not strictly more expressive than LTL (and vice versa)
- CTL\* invented by Emerson and Halpern in 1986 to unify CTL and LTL
- We believe that almost all properties that one wants to express about software lie in intersection of LTL and CTL

# Basic Concepts CTL

---

## Basic concepts: syntax and semantics

- *Syntax* tells how to write formulas: syntax gives the rules to write *correct* (i.e., *well-formed*) formulas (wff).
- Semantics gives a *meaning* to well-formed formulas. Semantics is used to decide whether or not a given wff is true or false.

# CTL Syntax

---

We start from a set of *atomic propositions*  $AP = \{p, q, \dots\}$ . Atomic propositions stand for atomic facts which may hold in a system, e.g. “Printer *ps706* is busy”, “Process *1486* is idle”, “The value of *x* is 5”, etc.

The Backus-Naur form form CTL formulae is the following:

$$\varphi ::= \top \mid \perp \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid AX\varphi \mid EX\varphi \mid AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A[\varphi U \varphi] \mid E[\varphi U \varphi]$$

# CTL Syntax

---

Each CTL operator is a pair of symbols. The first one is either A (“for All paths”), or E (“there Exists a path”). The second one is one of X (“neXt state”), F (“in a Future state”), G (“Globally in the future”) or U (“Until”).

**NOTICE:** U is a *binary* operator, it could be written  $EU(\varphi, \psi)$  or  $AU(\varphi, \psi)$ . Notice that the quantifier is graphically separated (e.g.,  $E[pUq]$ ), but it is in fact a single operator  $EU$ , which could be written  $EU(p, q)$ .

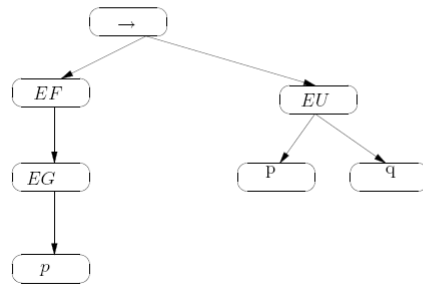
Example:  $AG(p \rightarrow (EFq))$  is read as “It is Globally the case that, if  $p$  is true, then there Exists a path such that at some point in the Future  $q$  is true”.

## CTL Parse trees

---

Parse trees are very useful to understand CTL formulas. For instance:

$$(EF(EGp)) \rightarrow E[pUq]$$



## Exercise

---

Is it a wff? Why?

1.  $EFGr$
2.  $A\neg G\neg p$
3.  $A[pU(EFr)]$
4.  $F[rUq]$
5.  $EF(rUq)$
6.  $AEFr$
7.  $A[rUA[pUq]]$
8.  $A[(rUq) \wedge (pUr)]$

# Answer

---

1.  $EFGr$  NO
2.  $A\neg G\neg p$  NO
3.  $A[pU(EFr)]$  YES
4.  $F[rUq]$  NO
5.  $EF(rUq)$  NO
6.  $AEFr$  NO
7.  $A[rUA[pUq]]$  YES
8.  $A[(rUq) \wedge (pUr)]$  NO

# CTL Semantics

---

You should be able to identify well-formed CTL formulae. Now: how to evaluate formulae, i.e., how to decide whether or not a formula is true.

You should know the meaning of *tautology* and *unsatisfiable formulae*:

- $AG(p \vee \neg p)$  : tautology
- $AG(p \wedge \neg p)$ : unsatisfiable

But what about  $EFp$ ? it may be true or not, depending on how we evaluate formulae.

# CTL Semantics

---

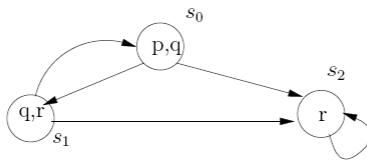
We evaluate formulae in *transition systems*. A transition system model a system by means of *states* and *transitions* between states. Formally:

A transition system  $M = (S, R_t, L)$  is a set of states  $S$  with a binary relation  $R_t \subseteq S \times S$  and a labelling function  $L : S \rightarrow 2^{AP}$  ( $AP$  is a set of atomic propositions, see above). The relation  $R_t$  is *serial*, i.e., for every state  $s \in S$ , there exists a state  $s'$  s.t.  $sR_t s'$ .

# CTL Semantics: transition system

---

An example  $M = (S, R_t, L)$

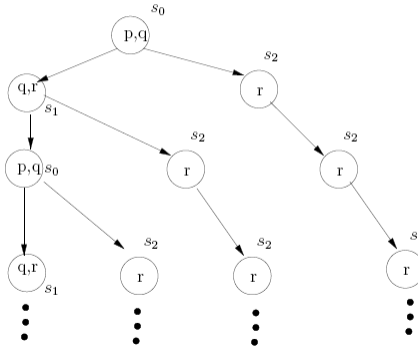


Here  $S = \{s_0, s_1, s_2\}$ ,  $R_t = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$ , and  $L(s_0) = \{p, q\}$ ,  $L(s_1) = \{q, r\}$ ,  $L(s_2) = \{r\}$ .

# CTL semantics

- from transition systems to computation paths

It is useful to visualise all possible computation paths by *unwinding* the transition system:

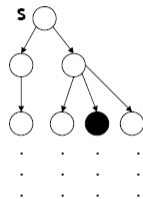


# CTL Examples

○ p does not hold

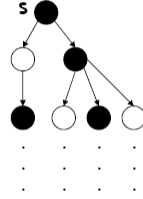
● p holds

## CTL Examples



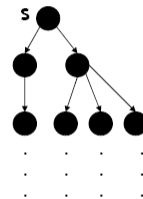
At state s:  
 EF p, EX (EX p),  
 AF (¬p), ¬p holds

AF p, AG p,  
 AG (¬p), EX p,  
 EG p, p does not hold



At state s:  
 EF p, AF p,  
 EX (EX p),  
 EX p, EG p, p holds

AG p, AG (¬p),  
 AF (¬p) does not hold



At state s:  
 EF p, AF p,  
 AG p, EG p,  
 Ex p, AX p, p holds

EG (¬p), EF (¬p),  
 does not hold

## Short Summary

---

- You should be able to recognise well-formed CTL formulas.
- You know what a transition system is ( $M = (S, R_t, L)$ ).
- You know how to unwind a transition system and obtain computation paths.

Next: Given a CTL formula  $\varphi$  and a transition system  $M$ , establish whether or not  $\varphi$  is true at a given state  $s$  in  $M$ , written as:

$$M, s \models \varphi$$

## CTL semantics

---

Let  $M = (S, R_t, L)$  be a transition system (also called a *model* for CTL). Let  $\varphi$  be a CTL formula and  $s \in S$ .  $M, s \models \varphi$  is defined inductively on the structure of  $\varphi$ , as follows

$$\begin{aligned} M, s &\models \top \\ M, s &\not\models \perp \\ M, s &\models p \quad \text{iff } p \in L(s) \\ M, s &\models \neg\varphi \quad \text{iff } M, s \not\models \varphi \\ M, s &\models \varphi \wedge \psi \quad \text{iff } M, s \models \varphi \text{ and } M, s \models \psi \\ M, s &\models \varphi \vee \psi \quad \text{iff } M, s \models \varphi \text{ or } M, s \models \psi \end{aligned}$$

## CTL Semantics (temporal operators)

---

- $M, s \models AX\varphi$  iff  $\forall s'$  s.t.  $sR_t s', M, s' \models \varphi$
- $M, s \models EX\varphi$  iff  $\exists s'$  s.t.  $sR_t s'$  and  $M, s' \models \varphi$
- $M, s \models AG\varphi$  iff for all paths  $(s, s_2, s_3, s_4, \dots)$  s.t.  $s_i R_t s_{i+1}$  and for all  $i$ , it is the case that  $M, s_i \models \varphi$
- $M, s \models EG\varphi$  iff *there is a path*  $(s, s_2, s_3, s_4, \dots)$  s.t.  $s_i R_t s_{i+1}$  and for all  $i$  it is the case that  $M, s_i \models \varphi$
- $M, s \models AF\varphi$  iff for all paths  $(s, s_2, s_3, s_4, \dots)$  s.t.  $s_i R_t s_{i+1}$ , there is a state  $s_i$  s.t.  $M, s_i \models \varphi$
- $M, s \models EF\varphi$  iff *there is a path*  $(s, s_2, s_3, s_4, \dots)$  s.t.  $s_i R_t s_{i+1}$ , and there is a state  $s_i$  s.t.  $M, s_i \models \varphi$

## CTL Semantics (temporal operators)

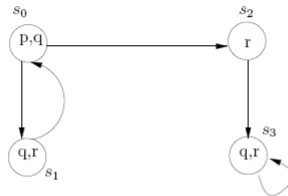
---

- $M, s \models A[\varphi U \psi]$  iff for all paths  $(s, s_2, s_3, s_4, \dots)$  s.t.  $s_i R_t s_{i+1}$  there is a state  $s_j$  s.t.  $M, s_j \models \psi$  and  $M, s_i \models \varphi$  for all  $i < j$ .
- $M, s \models E[\varphi U \psi]$  iff there exists a path  $(s, s_2, s_3, s_4, \dots)$  s.t.  $s_i R_t s_{i+1}$  and there is a state  $s_j$  s.t.  $M, s_j \models \psi$  and  $M, s_i \models \varphi$  for all  $i < j$ .

# Exercise

## CTL semantics: EXERCISE

Consider the following transition system:



Verify whether or not: (1)  $M, s_0 \models EX(\neg p)$ ; (2)  $M, s_0 \models EXEG(r)$ ;  
(3)  $M, s_1 \models AG(q \vee r)$ ; (4)  $M, s_2 \models A[rUq]$ ; (5)  $M, s_1 A[qUAG(r)]$ ;  
(6)  $M, s_1 E[qUEG(r)]$ ; (7)  $M, s_0 \models \neg EG(q)$ ; (8)  $M, s_1 \models EFAG(q)$ .

### CTL semantics: EXERCISE SOLUTIONS

(1) YES; (2) YES; (3) YES; (4) YES; (5) NO (because  $AG(r)$  is never true if you keep looping between  $s_0$  and  $s_1$ ); (6) YES; (7) YES; (8) YES.

# Equivalences between CTL formulas

In the syntax of CTL we introduced all the operators AX, EX, AF, EF, AG, EG, AU, and EU. However, some formulas are equivalent:

$$AX\varphi \equiv \neg EX\neg\varphi$$

$$AG\varphi \equiv \neg EF\neg\varphi$$

$$AF\varphi \equiv \neg EG\neg\varphi$$

Moreover,  $EF\varphi \equiv E[\top U \varphi]$ . Therefore, only three operators are required to express all the remaining:  $EX, EG, EU$  (this is called an *adequate set of operators*). This is useful when developing algorithms for model checking.

# Specification patterns

---

Temporal logics are useful to express requirements of systems. Typically, requirements have *common and recurring patterns*. For instance, two example of patterns:

- **Liveness**: “Something good will eventually happen”. For instance: “Whenever any process requests to enter its critical section, it will eventually be permitted to do so”. In CTL:

$$AG(request \rightarrow AF(critical))$$

- **Safety**: “Nothing bad will happen”. For instance, “Only one process is in its critical section at any time”. In CTL (with 2 processes only):

$$AG(\neg(critical_1 \wedge critical_2))$$

# Practical Patterns of Specifications

---

- A certain process is enabled infinitely often on every path

$$AG(AF \text{ enabled})$$

- An upward traveling lift at the second floor does not change its direction if it has passengers that want to go to the fifth floor.

$$AG(\text{floor2} \wedge \text{directionup} \wedge \text{Button5} \rightarrow A[\text{directionup} \text{ U } \text{floor5}])$$

- It is possible to get to a state where started holds, but ready does not

$$EF(\text{started} \wedge \neg \text{ready})$$

# Model Checking CTL

---

We have seen very simple example in these slides. However, real systems may be composed of hundred of thousand states. **Efficient algorithms are needed** to verify  $M, s \models \varphi$ .

We will see example of systems using NuSMV (a model checker) later in the course.

How do you verify a formula in a model? What we did: unwind the transition system  $M$ . However, a computer *cannot* check infinite data structures: we need to check *finite* data structure.

Next: an algorithm to compute the set of states of a model  $M$  in which  $\varphi$  holds, **the labelling algorithm**.

# The Labeling algorithm

---

INPUT: a CTL model  $\mathcal{M} = (S, \rightarrow, L)$  and a CTL formula  $\phi$ .

OUTPUT: the set of states of  $\mathcal{M}$  which satisfy  $\phi$ .

First, change  $\phi$  to the output of TRANSLATE( $\phi$ ), i.e., we write  $\phi$  in terms of the connectives AF, EU, EX,  $\wedge$ ,  $\neg$  and  $\perp$  using the equivalences given earlier in the chapter. Next, label the states of  $\mathcal{M}$  with the subformulas of  $\phi$  that are satisfied there, starting with the smallest subformulas and working outwards towards  $\phi$ .

# Labeling Informally

---

Suppose  $\psi$  is a subformula of  $\phi$  and states satisfying all the *immediate* subformulas of  $\psi$  have already been labelled. We determine by a case analysis which states to label with  $\psi$ . If  $\psi$  is

- $p$ : then label  $s$  with  $p$  if  $p \in L(s)$ .
- $\psi_1 \wedge \psi_2$ : label  $s$  with  $\psi_1 \wedge \psi_2$  if  $s$  is already labelled both with  $\psi_1$  and with  $\psi_2$ .
- $\neg\psi_1$ : label  $s$  with  $\neg\psi_1$  if  $s$  is not already labelled with  $\psi_1$ .
- $\text{AF } \psi_1$ :
  - If any state  $s$  is labelled with  $\psi_1$ , label it with  $\text{AF } \psi_1$ .
  - Repeat: label any state with  $\text{AF } \psi_1$  if all successor states are labelled with  $\text{AF } \psi_1$ , until there is no change. This step is illustrated in Figure 3.24.
- $\text{E}[\psi_1 \text{ U } \psi_2]$ :
  - If any state  $s$  is labelled with  $\psi_2$ , label it with  $\text{E}[\psi_1 \text{ U } \psi_2]$ .
  - Repeat: label any state with  $\text{E}[\psi_1 \text{ U } \psi_2]$  if it is labelled with  $\psi_1$  and at least one of its successors is labelled with  $\text{E}[\psi_1 \text{ U } \psi_2]$ , until there is no change. This step is illustrated in Figure 3.25.
- $\text{EX } \psi_1$ : label any state with  $\text{EX } \psi_1$  if one of its successors is labelled with  $\psi_1$ .

# Labeling for EG

---

**Handling EG directly** Instead of using a minimal adequate set of connectives, it would have been possible to write similar routines for the other connectives. Indeed, this would probably be more efficient. The connectives  $\text{AG}$  and  $\text{EG}$  require a slightly different approach from that for the others, however. Here is the algorithm to deal with  $\text{EG } \psi_1$  *directly*:

- $\text{EG } \psi_1$ :
  - Label *all* the states with  $\text{EG } \psi_1$ .
  - If any state  $s$  is *not* labelled with  $\psi_1$ , *delete* the label  $\text{EG } \psi_1$ .
  - Repeat: *delete* the label  $\text{EG } \psi_1$  from any state if *none* of its successors is labelled with  $\text{EG } \psi_1$ ; until there is no change.

---

## LTL vs CTL

## CTL vs. LTL

---

- If a process is enabled infinitely often, then it runs infinitely often. LTL: GF enabled  $\rightarrow$  GF running

CTL: - (AGAF enabled  $\rightarrow$  AGAF running)?

- From any state it is possible to get to a restart state

CTL: AG(EF restart) LTL: -

- The lift can remain idle on the third floor with its doors closed

CTL: AG(floor3  $\wedge$  idle  $\wedge$  doorclosed  $\rightarrow$  EG(floor3  $\wedge$  idle  $\wedge$  doorclosed))

LTL: -

## Comparison

---

- Different views of time – branching with linear
- Incomparable expressive power:
  - FGp is not expressible in CTL
  - AGEFp is not expressible in LTL
- Performance:
  - CTL run in time  $O(|P| \times |f|)$
  - LTL run in time  $O(|P| \times 2^{O(|f|)})$  and space  $O((|f| + \log(|P|))^2)$

## Disadvantages of CTL

---

- CTL is unintuitive and difficult to use
- In most important cases complexity is comparable to LTL or even worse (reactive systems, hierarchical systems)
- Hardly applicable for compositional verification and semi-formal verification
- On formulas which expressible both in LTL and CTL performance is equal or may be made equal

## Advantages of LTL

---

- LTL is more natural for verification
- Suits well for compositional and semi-formal verification
- Allow application of different techniques for state-space reduction
- Uniform approach for model treatment (language containment and automata theory)
- Bounded model checking

## Disadvantages of LTL

---

- For compositional reasoning specification language should be as expressive as modeling language
  - **BUT!** LTL cannot express finite-state systems

Corollary: LTL is too weak

- We need more expressive temporal specification language

## Solutions

---

- ETL [VW94] –extension with temporal connectives representing  $\omega$ -automata
- Linear  $\mu$ -calculus [BB87] – extended LTL with fixpoint operators
- QPTL [SVW87] extension of LTL with quantification over propositional variables
- FTL by Intel [Armoni02]

## Conclusions

---

- LTL suits better for verification of concurrent reactive systems
- CTL can be used as back-end for linear-time model checkers
- The linear-time model checking technology is still an open issue