Lectures 10-12

Model Checking

Lecture Outline

- References
- Definition of Model Checking
- The Model Checking Process
- Model Checking Benefits and Drawbacks
- System Modeling. Transition Systems
- Reactive Systems
- Propositional Linear Temporal Logic (PLTL)
- Computation Tree Logic (CTL)
- CTL Model Checking Algorithms
- Symbolic Model Checking
- Model Checking Tools

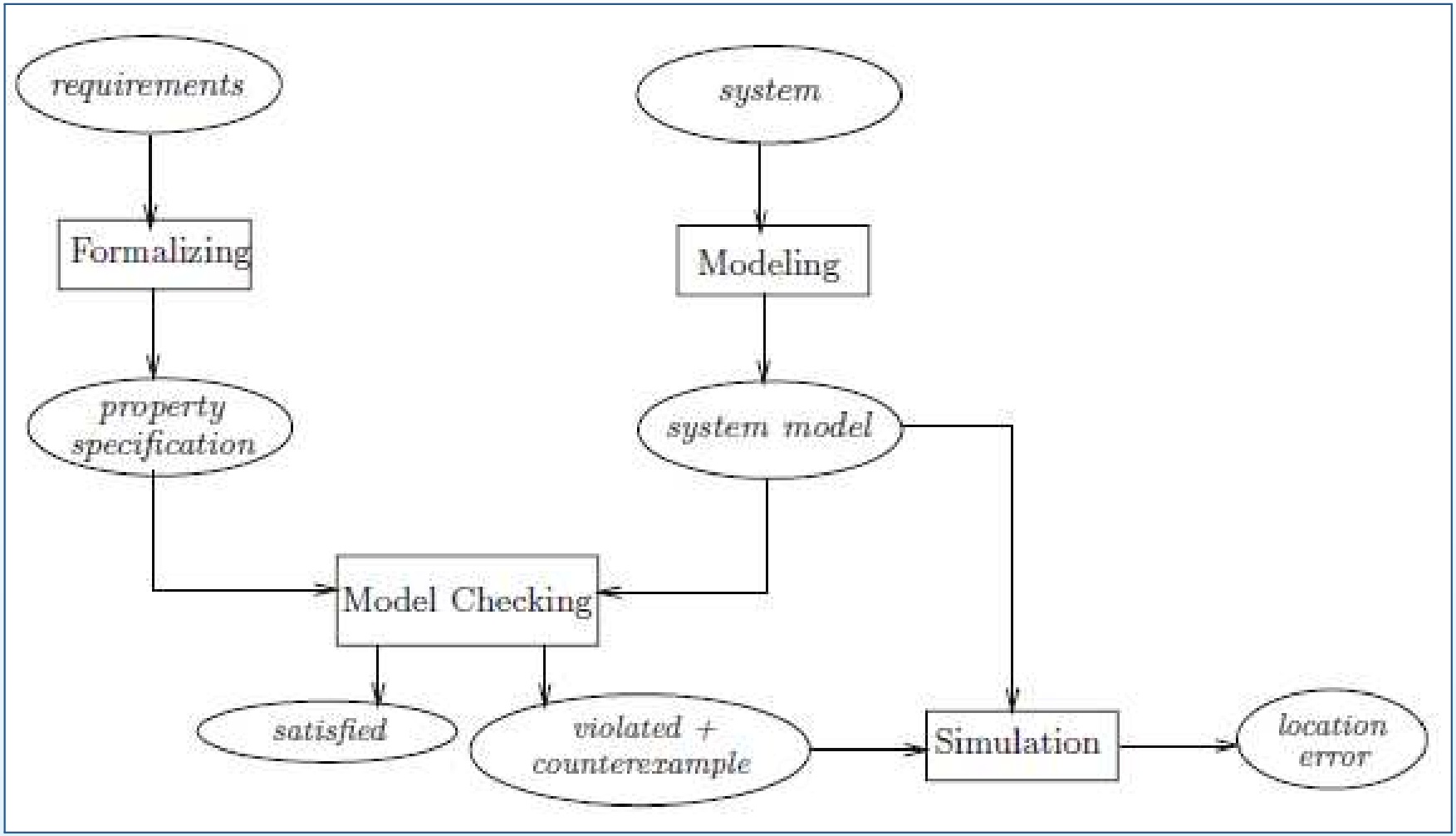
References

- [1] Baier, C. and Katoen, J.-P., *Principles of Model Checking*, The MIT Press, 2008.
- [2] Merz., S., *Model Checking: A Tutorial Overview*, Lecture Notes in Computer Science 2067, pp. 3 - 38, 2001.
- [3] Muler-Olm, M., Schmidt, D., and Steffen, B., *Model Checking: A Tutorial Introduction*, Lecture Notes in Computer Science 1694, pp. 330 - 354, 1999.
- [4] Clarke, E.M. and Lerda, F., *Model Checking: Software and Beyond*, Journal of Universal Computer Science, 13(5), 639-649, 2007.
- [5] Katoen, J.-P., *Concepts, Algorithms and Tools for Model Checking*, Friedrich-Alexander Universitat Erlangen-Nurnberg, 1999.
- [6] SPIN homepage - http://spinroot.com/spin/whatispin.html
- [7] NuSMV homepage - http://nusmv.fbk.eu/

Definition of Model Checking

- State of the art approaches in formal verification
  - theorem proving
  - model checking
- *Definition 10.1*: *Model checking* [1] is an automated verification technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

The Model Checking Scheme [1]



The Model Checking Process [1]

- *Modeling phase*
  - model the system under consideration using the *model description language* of the employed model checker
  - formalize the desired property using the *property specification language* at hand
- *Running phase*
  - run the model checker tool to verify whether the property holds for the system model or not
- *Analysis phase*
  - property satisfied $\Rightarrow$ check the next one (if any)
  - property not satisfied $\Rightarrow$
    - perfom simulation using the provided counterexample
    - correct the model, design or property specification
    - re-enter the running phase
  - out of memory $\Rightarrow$ reduce the model state space and start over

Modeling phase

- *Model description language*
  - usually some sort of *finite-state automaton* (finite set of states + transitions among them)
    - states enclose information regarding current values of variables, previously executed statement (program counter position), etc.
  - for real systems, the model description language is dialect/extension of a known language (e.g. C, Java, etc.)
- *Property specification language*
  - usually, some sort of *temporal logic*
    - an extension of propositional logic with operators describing the behavior of systems over time
    - allows the specification of properties such as
      - functional correctness (does the system do what it should?)
      - reachability (is it possible to end up in a deadlock state?)
      - safety (something bad never happens)
      - liveness (something good will eventually happen)
      - fairness (does an event occur repeatedly under certain conditions?)
      - real-time properties (is the system acting in time?)

Running phase

- A "push button" algorithmic approach
- The validity of the considered property is checked in each state of the model system

Analysis phase

- A property which is not satisfied may indicate
  - *a system design error*
    - the system design should be improved, the model synchronized with it and verification restarted for all the properties
  - *a system modeling error* (the model is not consistent with the system design)
    - the model should be corrected and verification restarted for all properties
  - *a property formalization error* (the formal property statement does not correspond to its informal specification)
    - the property should be adjusted and a new verification performed for it
    - verification of the previous properties should not be repeated, since the model has not changed
- An "out of memory" outcome may be handled by
  - applying symbolic techniques for state space representation (e.g. binary decision diagrams, partial order reduction)
  - using rigorous abstractions of the complete system
  - giving up the precision of the result with probabilistic verification approaches

Model Checking Benefits [1]

- *sound math foundations* (graph theory, logic, data structures)
- *general approach*
  - may be applied to a wide variety of domains, e.g. software engineering, hardware design, communication protocols, embedded systems, etc.
- allows *partial verifications*
  - properties can be checked individually, in the natural order of their relevance; a design can be verified against a partial specification
- provides *diagnosing information*
  - useful for debugging purposes
- *"push button" technology*
  - low degree of user interaction and expertise required, short learning curve
- *easy integrable* into existing development processes, leading to *shorter development cycles*
- *increasing interest from industry*
  - many commercial tools have become available

Model Checking Drawbacks [1]

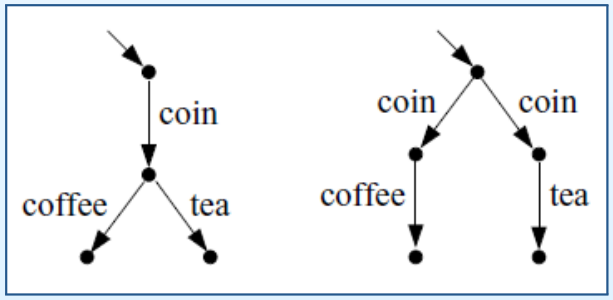
- *it works on the model*, not the system itself, therefore its results can be no better than the model is
  - complementary techniques such as testing are needed to find coding errors (for software) or fabrication faults (for hardware)
- *more appropriate for control-intensive systems* than data-intensive ones (which have infinite state spaces)
- *state explosion problem*, i.e. the number of states needed to model realistic systems can easily exceed the available computer resources
- *requires expertise* in finding abstractions for building smaller system models and state properties in temporal logic
- *no guarantee of completeness*, since it only checks explicitely states requirements
- *no absolute guarantee of correctness*, the tool may contain software defects

System Modeling

- Model checking dependends on a dicrete graph-like model of the system
  - nodes correspond to system states
  - arcs represent transitions among states
- Simple graphs are two weak to provide a meaningful description, therefore they should be annotated with specific information
- Two approaches are in common use
  - labeled transition systems (LTS) - arcs are annotated with actions
  - Kripke structures (KS) - states are annotated with atomic propositions
- A mixed approach, Kripke Transition Systems (KTS), is often more convenient/intuitive for modeling purposes

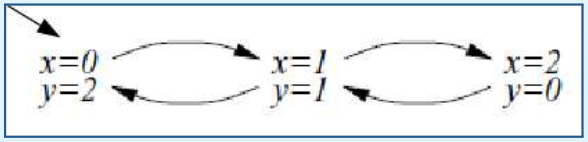
Labeled Transition Systems

- *Definition 10.2*: A *labeled transition system* (*LTS*) [3] is a 3-tuple LTS=(S, A, δ), where
  - S is a set of *states*
  - A is a set of *actions*
  - $\delta \subseteq S \times A \times S$ is a *transition relation*
    - If $(s, a, s') \in \delta$, then we adopt the notation $s \xrightarrow{a} s'$, that reads as "the system can evolve from state s to state s' under action a"
      - s' is called an a-*successor* of s
    - in model checking applications, S and A are usually finite
- LTS originate from concurency theory, where they are used as operational models of process behavior
- LTS examples modeling the behavior of a vending machine [3]



Kripke Structures

- *Definition 10.3*: A *Kripke structure* (*KS*) over a set of atomic propositions AP is a tuple KS=(S, δ, L, I), where
  - S is a set of *states*
  - $\delta \subseteq S \times S$ is a *transition relation*
  - $I \subseteq S$ is the set of *initial states*
  - $L : S \to 2^{AP}$ is a *labeling function* (or *interpretation function*)
    - the atomic propositions represent basic properties of system states
    - the function L assigns to each state the properties enjoyed by it
    - we assume that $\{true, false\} \subseteq AP$ and $true \in L(s)$ and $false \notin L(s)$, $\forall\ s \in S$
    - KS is called *total* if δ is a total relation
    - for model checking purposes, S and AP are usually finite
- KS example representing the states arising when two program components x and y trade two resources back and forth. Propositions are of the shape $var = num$



Kripke Structures (cont.)

- Let $KS = (S, \delta, L, I)$ be a Kripke structure
  - a *finite path* is a finite, non-empty sequence $\pi =< \pi_0, \ldots, \pi_{n-1} >$ of states $\pi_0, \ldots, \pi_{n-1} \in S$, such that $(\pi_i, \pi_{i+1}) \in \delta$, $\forall\ 0 \leq i < n-1$
    - $|\pi| = n$ is the *length* of the path
  - an *infinite path* is an infinite sequence $\pi =< \pi_0, \pi_1, \pi_2, \ldots >$ of states in S, such that $(\pi_i, \pi_{i+1}) \in \delta$, $\forall\ i \geq 0$
    - the length of an infinite path in ∞
    - π_i is the i + 1 - th state in path π
    - π^i is the sufix/tail of π, following the first i elements, $\pi^i =< \pi_i, \pi_{i+1}, \ldots >$
  - a path is called *maximal* if it cannot be extended
  - The set of paths starting from s

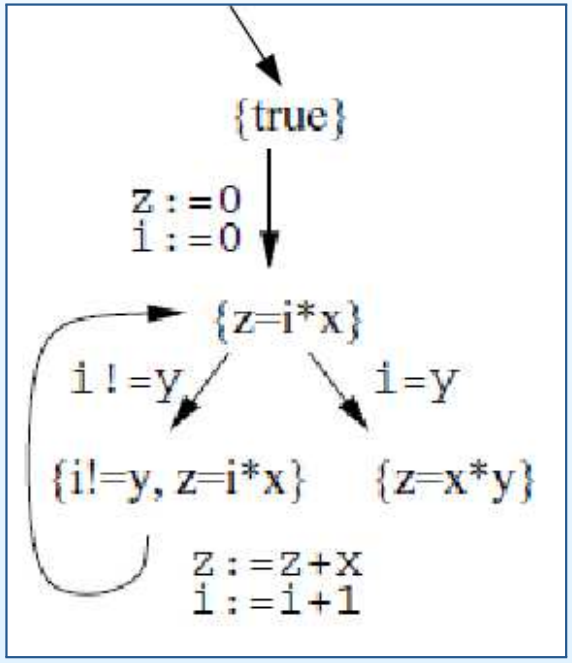
$$Path_K(s) = \{\pi \mid \pi\ is\ a\ path\ in\ K\ and\ \pi_0 = s\}$$

Kripke Transition Systems

- Labels on arcs (as in LTS) are needed when the labeling captures the dynamics of a system, while labels on states (as in KS) - when the labeling captures static properties of states.
- There are ways of encoding arcs' labeling into states' labeling and viceversa, therefore theoretical analyses deal with a single form of labeling
- In modeling however, it is more intuitive to have the possibility to use both forms of labeling at once, therefore the need of a structure combining both LTS and KS.
- *Definition 10.4*: A *Kripke transition system* (*KTS*) over a set of atomic propositions AP is a tuple KTS=(S, A, δ, L, I), where
  - S is a set of *states*
  - A is a set of *actions* assumed to be disjoint with AP
  - $\delta \subseteq S \times A \times S$ is a *transition relation*
  - $I \subseteq S$ is the set of *initial states*
  - $L : S \to 2^{AP}$ is a *labeling function* (or *interpretation function*)

Kripke Transition Systems (cont.)

- KTS are appropriate for the modeling of sequencial imperative programs when performing dataflow analysis
  - nodes are labeled by predicates meaningfull in the context of the performed analysis
  - arcs are labeled by statements
- KTS example [3]



$$z := 0; i := 0$$

$$while\ i \neq y\ do$$

$$z := z + x$$

$$i := i + 1$$

$$endwhile$$

Kripke Transition Systems (cont.)

- Trafic Lights example
  (see slide 15 of Lecture 11)

Reactive Systems

- Unlike traditional sequencial (transformational) systems, which are viewed as functions from inputs to outputs, *reactive systems* are characterized by a continuous, "non-terminating" interaction with the environment
  - such a system interracts by reacting to stimuli from its environment
  - e.g. an operating system, a traffic control system, a cofee machine
- Properties of reactive systems are generally concerned with the behavior of the system over time (relative ordering of events)
  - These properties tipically fall in two (almost) disjoint classes
    - *safety* properties ("something bad never happens") - e.g. the coffee machine will never provide tea if the user has requested cofee
    - *liveness* properties ("something good will eventually happen") - e.g. the user will eventually receive the coffee if he has paid for it
    - most reactive systems properties are a combination of liveness and safety
  - Such properties can be described by means of temporal logics
    - linear-time temporal logics, e.g. Propositional Linear Temporal Logic (PLTL)
    - branching-time temporal logics, e.g. Computation Tree Logic (CTL)

PLTL - Syntax

- **Definition 11.1** [PLTL syntax]: Let AP be a set of atomic propositions. Then
  1. For all $p \in AP$, p is a formula.
  2. If ϕ is a formula, then $\neg\phi$ is a formula.
  3. If ϕ and ψ are formulas, then $\phi \vee \psi$ is a formula.
  4. If ϕ is a formula, then $\mathsf{X}\,\phi$ is a formula.
  5. If ϕ and ψ are formulas, then $\phi\,\mathsf{U}\,\psi$ is a formula.

  The set of formulas constructed by these rules is denoted by *PLTL*.
- **Observations**:
  1. The first 3 items of the above definition give the formulas of propositional logic, of which PLTL is a superset
  2. The temporal operators are X (read as "neXt") and U (read as "Until")
  3. BNF equivalent of the PLTL syntax definition above

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi$$

PLTL - Syntax (cont.)

- The other boolean operators are defined as usual

$$\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$$
$$\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$$
$$\phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$$

- *true* and *false* are defined as

$$true \equiv \phi \vee \neg\phi$$
$$false \equiv \neg true$$

- The other two temporal operators used by PLTL, namely F (read as "eventually" or "Finally") and G (read as "Globally" or "always") are defined as

$$\mathsf{F}\,\phi \equiv true\,\mathsf{U}\,\phi$$
$$\mathsf{G}\,\phi \equiv \neg\mathsf{F}\,\neg\phi$$

PLTL - Syntax (cont.)

- Informal meaning of temporal operators
  - A formula without temporal operators at the top level refers to the current state
  - Xϕ refers to the next state
  - Fϕ refers to some future state
  - Gϕ refers to all future states
  - U refers to all future states until a particular condition is valid
- Operators' precedence
  - Unary operators have higher precedence then binary ones
  - $\neg$ and X have equal precedence
  - G and F have equal precedence, but lower then the previous two
  - U has higher precedence then $\wedge$, $\vee$ and $\Rightarrow$
  - $\wedge$ and $\vee$ have equal precedence and higher than $\Rightarrow$
- **Example 11.1**: If $AP = \{x = 1, x < 2, x \geq 3\}$ the the following are PLTL formulas: $\neg(x < 2)$, $(x < 2) \vee (x = 1)$, X$(x = 1)$, F$(x < 2)$, G$(x = 1)$, $(x < 2)$U$(x \geq 3)$.

PLTL - Semantics

- The PLTL syntax only gives rules for constructing syntactically correct PLTL formulas, without any reference to their meaning/ interpretation
- PLTL is interpreted over sequences of states (or paths)
- The formal meaning/semantics of temporal logic fomulas is defined in terms of a model
- A PLTL model is a Kripke structure $\mathcal{M} = (S, \delta, L, I)$ in which the transition relation δ is, in fact, a total function (each state has a unique successor)
  - If $s \in S$, $\delta(s)$ is the unique next state of s (or direct successor)
  - δ acts like a generator of infinite sequences of states $s, \delta(s), \delta(\delta(s)) = \delta^2(s), \delta^3(s), \ldots$
  - The labeling/interpretation function L assigns to each state of S the atomic properties from AP that it enjoys of
  - The meaning of logical formulas is given by means of a satisfaction relation (denoted $\models$) between a model $\mathcal{M}$, one of its states s, and a formula ϕ. We write $\mathcal{M}, s \models \phi$, or just $s \models \phi$ iff ϕ is valid in state s of $\mathcal{M}$

PLTL - Semantics (cont.)

- **Definition 11.2** [PLTL Semantics]: Let $p \in AP$ be an atomic proposition, ϕ, ψ PLTL formulas, $\mathcal{M} = (S, \delta, L, I)$ a PLTL model over AP, and $s \in S$ a state. The satisfaction relation $\models$ is defined by:

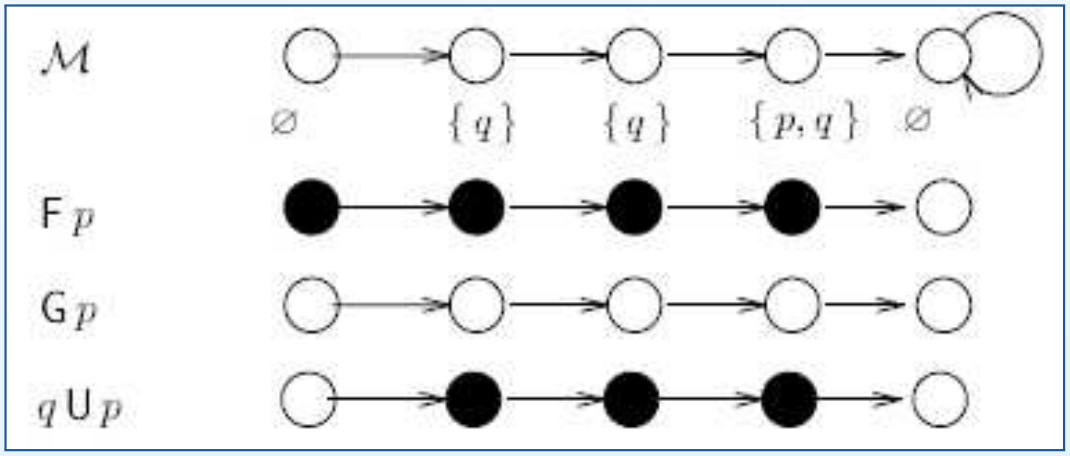
$$
\begin{aligned}
s &\models p && \text{iff } p \in L(s) \\
s &\models \neg\phi && \text{iff } \neg(s \models \phi) \\
s &\models \phi \vee \psi && \text{iff } (s \models \phi) \vee (s \models \psi) \\
s &\models \mathsf{X}\,\phi && \text{iff } \delta(s) \models \phi \\
s &\models \phi\,\mathsf{U}\,\psi && \text{iff } \exists\, j \geq 0 \text{ such that } \delta^j(s) \models \psi \text{ and } \delta^k(s) \models \phi\ \forall\, 0 \leq k < j
\end{aligned}
$$

  (Here, $\delta^0(s) = s$ and $\delta^{n+1}(s) = \delta(\delta^n(s)),\ \forall\, n \geq 0$.)

- The semantics of the other logical and temporal operators and logical constants can be easily derived:
  - $true$ is valid in all states, $false$ is not valid in any state
  - $s \models \mathsf{F}\,\phi$ iff $\exists\, j \geq 0$ such that $\delta^j(s) \models \phi$
  - $s \models \mathsf{G}\,\phi$ iff $\forall\, j \geq 0\ \delta^j(s) \models \phi$

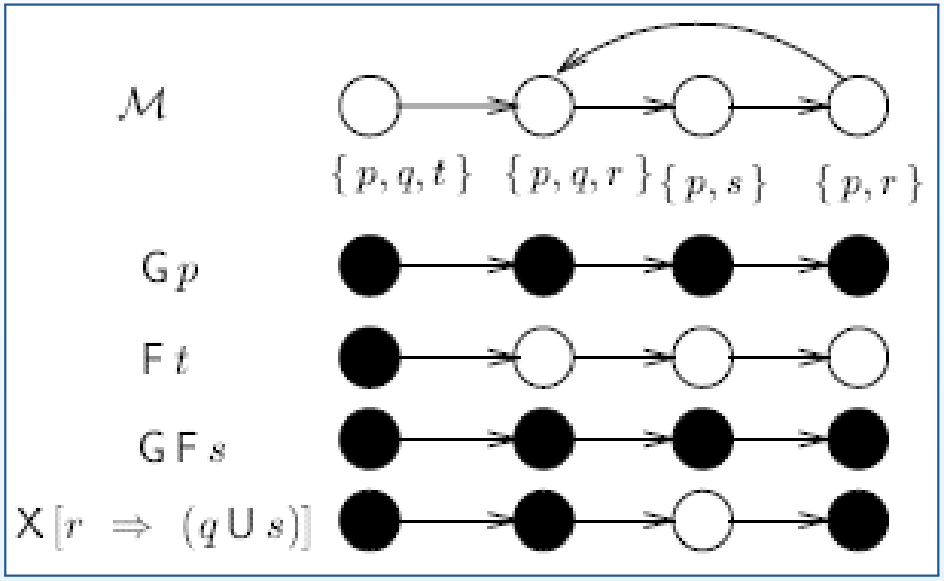
PLTL - Semantics (cont.)

- **Definition 11.3** [Formal definition of the model checking problem]: The model checking problem is: given a finite model $\mathcal{M}$, a state s, and a property specification ϕ, do we have $\mathcal{M}, s \models \phi$?
- **Example 11.2**: Let $\mathcal{M}$ be the PLTL model in the next figure. Below there are illustrations of the validity of three PLTL formulas, for each state of the model (a state is colored black if the formula is satisfied in that particular state, and white otherwise).



PLTL - Semantics (cont.)

- **Example 11.3**: Let $\mathcal{M}$ be the PLTL model in the next figure. Below there are illustrations of the validity of four PLTL formulas, for each state of the model (a state is colored black if the formula is satisfied in that particular state, and white otherwise).



PLTL - Semantics (cont.)

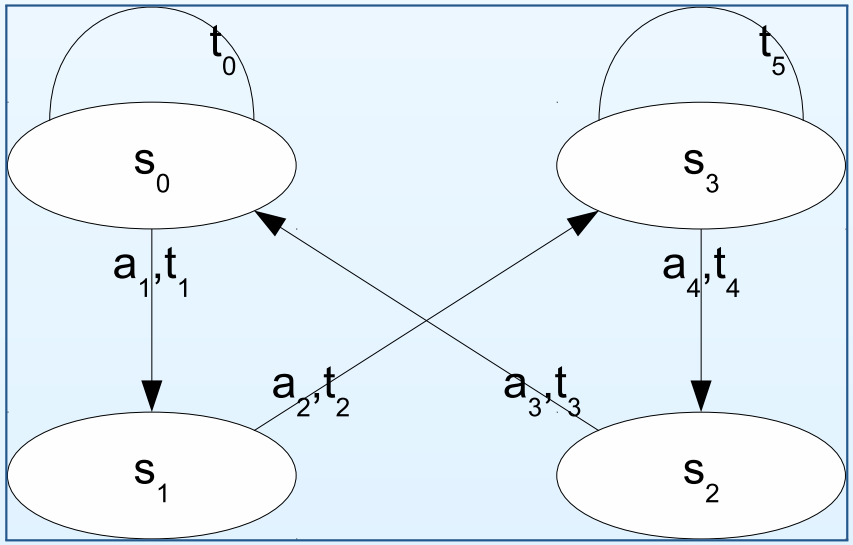
- Frequently used PLTL propositions and their interpretations in state s
  - $\phi \Rightarrow \mathsf{F}\,\psi$ - if initially (in s) ϕ, then eventually ψ
  - $\mathsf{G}[\phi \Rightarrow \mathsf{F}\,\psi]$ - if ϕ, then eventually ψ (for s and all its successor states)
  - $\mathsf{GF}\phi$ - ϕ holds infinitely often
  - $\mathsf{FG}\phi$ - eventually permanently (globally) ϕ
  - $\mathsf{G}[\phi \Rightarrow \mathsf{X}\phi]$ - if ϕ is valid in an some state, then it is also valid in its direct successor state
  - $\mathsf{G}[\phi \Rightarrow \mathsf{G}\phi]$ - once ϕ, allways ϕ

PLTL - Property specification

- **Example 11.4** [A communication channel]: Consider two processes, S (sender) and R (receiver), communicating via an unidirectional channel. S has an output buffer ($S.out$) and R has an input one ($R.in$), both with infinite capacity. If S sends a message m to R, it inserts m into $S.out$; R reads messages by deleting them from its input stream $R.in$. We assume that all messages are uniquely identified and that $AP = \{m \in S.out, m \in R.in\}$
  - A message cannot be in both buffers simultaneously:
    $\mathsf{G}\neg(m \in S.out \wedge m \in R.in)$
  - The channel does not loose messages:
    $\mathsf{G}(m \in S.out \Rightarrow \mathsf{F}(m \in R.in))$
  - The channel is order preserving:
    $\mathsf{G}[m \in S.out \wedge \neg m' \in S.out \wedge \mathsf{F}(m' \in S.out) \Rightarrow$
    $\Rightarrow \mathsf{F}(m \in R.in \wedge \neg m' \in R.in \wedge \mathsf{F}(m' \in R.in))]$

PLTL - Property specification (cont.)

- **Example 11.5** [Traffic lights]: Consider a system composed of two traffic lights, A and B, placed on an intersection, modeled by the Kripke transition system below. States are labeled by pairs consisting of the first letters of the corresponding colors of the two traffic lights, as follows: $L(s_0) = \{(g, r)\}$, $L(s_1) = \{(y, r)\}$, $L(s_3) = \{(r, g)\}$, $L(s_2) = \{(r, y)\}$. Transitions among states are labeled so as to reflect the change in traffic lights colors, namely: $a_1 = A := y$, $a_2 = (A, B) := (r, g)$, $a_3 = (A, B) := (g, r)$, $a_4 = B := y$,



  - todo

PLTL - Property specification (cont.)

- A and B cannot be green simultaneously:
  $\mathsf{G}\neg(A = g \wedge B = g)$
- If A is yellow, it will eventually be red:
  $\mathsf{G}(A = y \Rightarrow \mathsf{F}(A = r))$
- If A is yellow, it will be red in the next state:
  $\mathsf{G}(A = y \Rightarrow \mathsf{X}(A = r))$
- B will not be green until A becomes red:
  $\mathsf{G}(\neg(B = g)\mathsf{U}(A = r))$

CTL - Syntax

- **Definition 11.4** [CTL syntax]: Let AP be a set of atomic propositions. For $p \in AP$, the set of *CTL formulas* is defined as

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{EX}\,\phi \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi] \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi].$$

- **Observations**:
  1. Four temporal operators are used: EX (read as "for some path next"), E (read as "for some path"), A (read as "for all paths"), U (read as "Until")
  2. X and U are linear temporal operators, expressing properties over a single path, E expresses a property over some path and A over all paths
  3. $true$, $false$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined as usual
  4. The rules for the other temporal operators can be easily computed

$$\mathsf{AX}\phi \equiv \neg\mathsf{EX}\neg\phi$$
$$\mathsf{EF}\phi \equiv \mathsf{E}[true\,\mathsf{U}\,\phi] \quad \mathsf{AF}\phi \equiv \mathsf{A}[true\,\mathsf{U}\,\phi]$$
$$\mathsf{EG}\phi \equiv \neg\mathsf{AF}\neg\phi \qquad \mathsf{AG}\phi \equiv \neg\mathsf{EF}\neg\phi$$

CTL - Syntax (cont.)

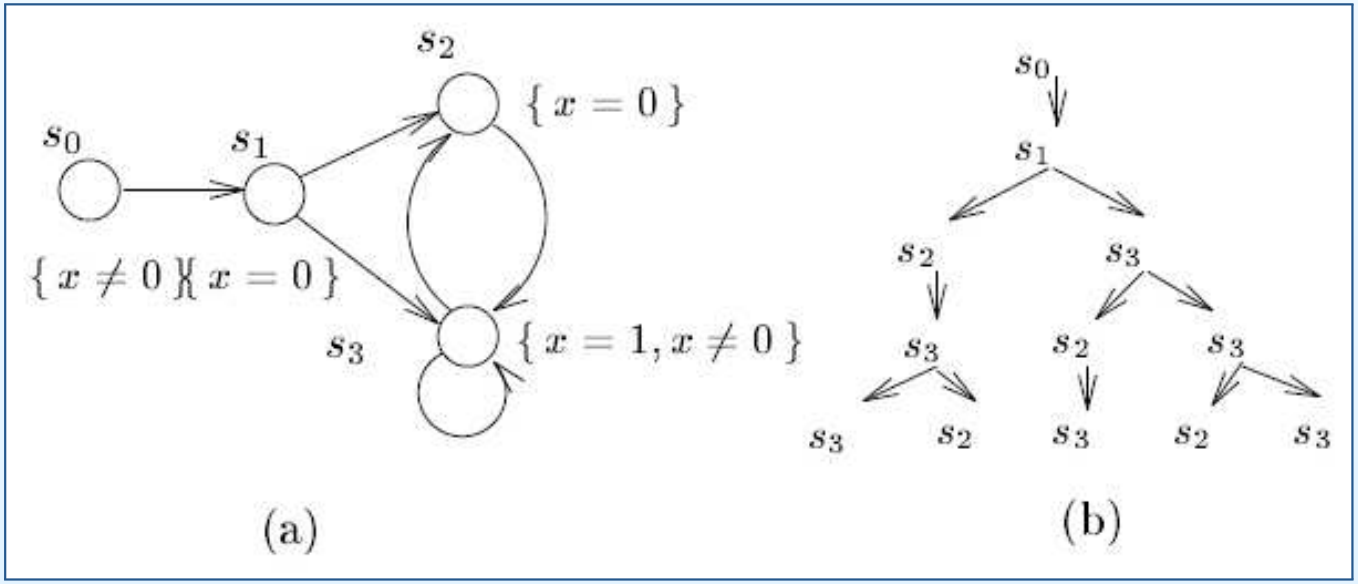
- **Observations**:
  5. AX reads as "for all paths next", EF ϕ as "ϕ holds potentially", AF ϕ as "ϕ is inevitable", EG ϕ as " potentially always ϕ" and AG ϕ as "invariantly ϕ"
  6. The operators A and E bind equally strong, having the highest precedence among unary operators. All the other operatos have the same precedence as in PLTL
  7. In CTL the linear temporal operators (X, F, G, U) should be immediately preceded by A or E.
- **Example 11.6**: Let $AP = \{x = 1, x < 2, x \geq 3\}$.
  - E$[x = 1 \wedge$ AX$(x \geq 3)]$ and EF[G$(x = 1)]$ are not valid CTL formulas
  - EG$[x = 1 \wedge$ AX$(x \geq 3)]$ and EF[EG$(x = 1)]$ are valid CTL formulas

CTL - Semantics

- The model used to define the PLTL semantics generates for each state s a unique infinite sequence of successor states (aka computation path starting at s), therefore the semantics of PLTL is given in terms of a single such computation path
- Branching-time logics, however, do not refer to a single computation path, but to several/all of them
- Therefore, in order to define the CTL semantics, the sequence generating model is replaced by a tree-generating model
- A CTL model is a Kripke structure, in which the transition relation is total (each state has at least one direct successor)
- A CTL model generates for each state s an infinite computation tree rooted at s

CTL - Semantics (cont.)

- **Example 11.7**: Let $AP = \{x = 0, x = 1, x \neq 0\}$ be a set of atomic propositions, $S = \{s_0, s_1, s_2, s_3\}$ be a set of states, with the labeling $L(s_0) = \{x \neq 0\}$, $L(s_1) = L(S_2) = \{x = 0\}$ and $L(s_3) = \{x = 1, x \neq 0\}$ and the transition relation $\delta = \{(s_0, s_1), (s_1, s_2), (s_1, s_3), (s_3, s_3), (s_3, s_2), (s_2, s_3)\}$. $\mathcal{M} = (S, \delta, L, \{s_0\})$ is a CTL model. $\mathcal{M}$ (a) and a prefix of its infinite computation tree (b), are shown below.



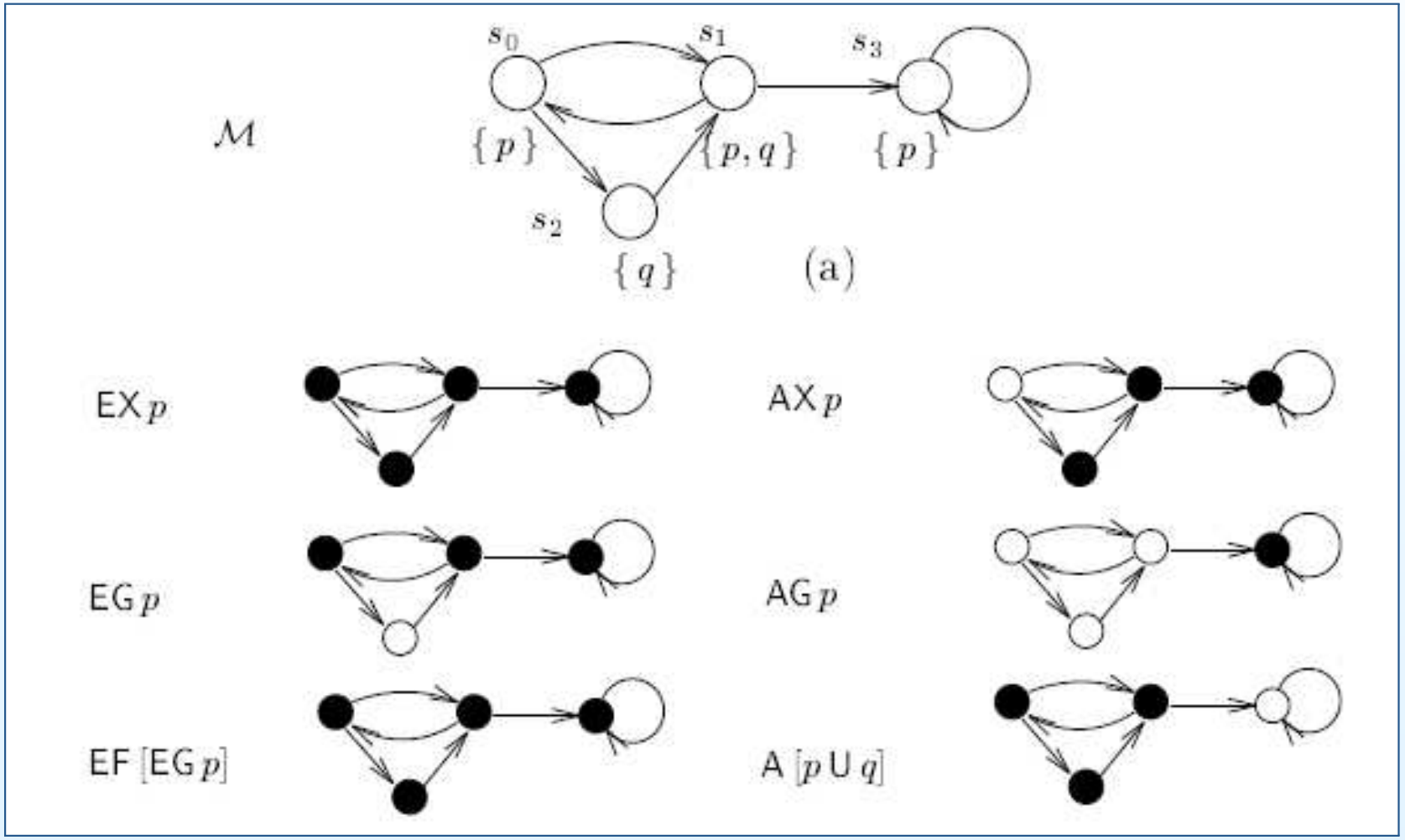
CTL - Semantics (cont.)

- **Definition 11.5** [CTL Semantics]: Let $p \in AP$ be an atomic proposition, ϕ, ψ CTL formulas, $\mathcal{M} = (S, \delta, L, I)$ a CTL model over AP, and $s \in S$ a state. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{rll}
s \models p & \text{iff} & p \in L(s) \\
s \models \neg\phi & \text{iff} & \neg(s \models \phi) \\
s \models \phi \vee \psi & \text{iff} & (s \models \phi) \vee (s \models \psi) \\
s \models \mathsf{EX}\,\phi & \text{iff} & \exists\, \pi \in Path_{\mathcal{M}}(s)\,.\,\pi_1 \models \phi \\
s \models \mathsf{E}[\phi\,\mathsf{U}\,\psi] & \text{iff} & \exists\, \pi \in Path_{\mathcal{M}}(s)\,.\,(\exists\, j \geq 0\,.\,\pi_j \models \psi \ \wedge\ \pi_k \models \phi\ \forall\, 0 \leq k < j) \\
s \models \mathsf{A}[\phi\,\mathsf{U}\,\psi] & \text{iff} & \forall\, \pi \in Path_{\mathcal{M}}(s)\,.\,(\exists\, j \geq 0\,.\,\pi_j \models \psi \ \wedge\ \pi_k \models \phi\ \forall\, 0 \leq k < j)
\end{array}
$$

CTL - Semantics (cont.)

- **Example 11.8**:



CTL - Property specification

- **Example 11.9** [Mutual Exclusion]: Consider a two process mutual exclusion program. Each of the two processes (P_1 and P_2) can be in one of the following states: critical section (C), attempting section (T) and non-critical section (N). A process starts in the non-critical section and shows that it wants to enter the critical one, by entering the attempting section. Once it receives access, it enters the critical section, and, when leaving it, it reenters the non-critical one. The state of a process P_i is denoted by $P_i.s$. Below are some properties in CTL specification.
  - The two processes are never simultaneously in the critical section: AG$[\neg(P_1.s = C \wedge P_2.s = C)]$
  - A process that wants to enter the critical section eventually does so: AG$[P_1.s = T \Rightarrow$ AF$(P1.s = C)]$
  - Processes must strictly alternate in having access to the critical section: AG$[P_1.s = C \Rightarrow$ A$[P_1.s = C$ U $(P_1.s \neq C \wedge$ A $(P_1.s \neq C$ U $P_2.s = C))]]$

CTL Model Checking (CTL MC)

- *CTL model checking problem*: Given a Kripke structure $\mathcal{M} = (S, \delta, I, L)$, with a total transition relation, over a set AP of atomic propositions, a state $s \in S$ and a CTL formula ϕ, decide whether $\mathcal{M}, s \models \phi$.
- The CTL model checking algorithm provided in [5] iteratively computes, for an arbitrary CTL formula ϕ, the set

$$Sat(\phi, \mathcal{M}) = \{s \in S \mid \mathcal{M}, s \models \phi\}$$

- The model checking problem $\mathcal{M}, s \models \phi$ is thus reduced to investigating whether $s \in Sat(\phi, \mathcal{M})$ or not
- Computing $Sat(\phi, \mathcal{M})$ solves a more general problem than the CTL model checking problem stated above
  - The entire set of states satisfying a formula is computed
  - The iterative algorithm computes $Sat(\psi, \mathcal{M})$, for any subformula ψ of ϕ, therefore allows investigating whether $\mathcal{M}, s \models \psi$

Main CTL MC Algorithm [5]

Function $Sat(\phi, \mathcal{M})$ is

{

$Pre : \mathcal{M} = (S, \delta, I, L) -$ Kripke structure over AP, $\delta -$ total relation

$\phi -$ CTL formula

$Post : Sat(\phi, \mathcal{M}) = \{s \in S \mid \mathcal{M}, s \models \phi\}$

}

Case ϕ **of**

$true$, then $Sat(\phi, \mathcal{M}) \leftarrow S$

$false$, then $Sat(\phi, \mathcal{M}) \leftarrow \varnothing$

$p \in AP$, then $Sat(\phi, \mathcal{M}) \leftarrow \{s \in S \mid p \in L(s)\}$

$\neg\psi$, then $Sat(\phi, \mathcal{M}) \leftarrow S - Sat(\psi, \mathcal{M})$

$\psi \vee \eta$, then $Sat(\phi, \mathcal{M}) \leftarrow Sat(\psi, \mathcal{M}) \cup Sat(\eta, \mathcal{M})$

EXψ, then $Sat(\phi, \mathcal{M}) \leftarrow \{s \in S \mid \exists\, s' \in \delta(s)\ .\ s' \in Sat(\psi, \mathcal{M})\}$

E[ψUη], then $Sat(\phi, \mathcal{M}) \leftarrow Sat_{EU}(\psi, \eta, \mathcal{M})$

A[ψUη], then $Sat(\phi, \mathcal{M}) \leftarrow Sat_{AU}(\psi, \eta, \mathcal{M})$

End{Case}

End{Function}

Main CTL MC Algorithm [5] (cont.)

Function $Sat_{EU}(\psi, \eta, \mathcal{M})$ is

{

$Pre : \mathcal{M} = (S, \delta, I, L)$ – Kripke structure over AP, δ – total relation

ψ, η – CTL formulas

$Post : Sat_{EU}(\psi, \eta, \mathcal{M}) = \{s \in S \mid \mathcal{M}, s \models \mathsf{E}[\psi \mathsf{U} \eta]\}$

}

$Q \leftarrow Sat(\eta, \mathcal{M});$

$Q' \leftarrow \varnothing;$

While $Q \neq Q'$ **do**

$Q' \leftarrow Q;$

$Q \leftarrow Q \cup (Sat(\psi, \mathcal{M}) \cap \{s \in S \mid \exists\, s' \in \delta(s) \cap Q\})$

End{While};

$Sat_{EU}(\psi, \eta, \mathcal{M}) \leftarrow Q$

End{Function}

Main CTL MC Algorithm [5] (cont.)

Function $Sat_{AU}(\psi, \eta, \mathcal{M})$ is

{

$Pre : \mathcal{M} = (S, \delta, I, L)$ – Kripke structure over AP, δ – total relation

ψ, η – CTL formulas

$Post : Sat_{AU}(\psi, \eta, \mathcal{M}) = \{s \in S \mid \mathcal{M}, s \models \mathsf{A}[\psi \mathsf{U} \eta]\}$

}

$Q \leftarrow Sat(\eta, \mathcal{M});$

$Q' \leftarrow \varnothing;$

While $Q \neq Q'$ **do**

$Q' \leftarrow Q;$

$Q \leftarrow Q \cup (Sat(\psi, \mathcal{M}) \cap \{s \in S \mid \forall\, s' \in \delta(s)\ .\ s' \in Q\})$

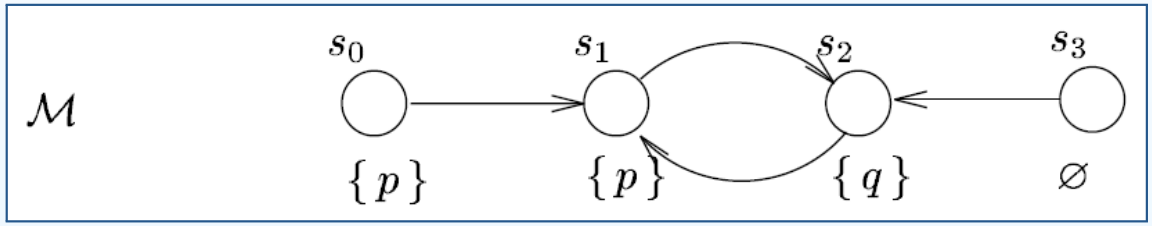
End{While};

$Sat_{AU}(\psi, \eta, \mathcal{M}) \leftarrow Q$

End{Function}

CTL MC Examples

- **Example 12.1**:Given the Kripke structure $\mathcal{M} = (S, \delta, L, I)$, with $S = \{s_0, s_1, s_2, s_3\}$, $\delta = \{(s_0, s_1), (s_1, s_2), (s_2, s_1), (s_3, s_2)\}$, $I = \{s_0\}$, $L(s_0) = \{p\}, L(s_1) = \{p\}, L(s_2) = \{q\}, L(s_3) = \varnothing$, check if $\mathcal{M}, s_0 \models \mathsf{E}\,[p\,\mathsf{U}\,q]$.



Solution: Compute $Sat_{EU}(p, q, \mathcal{M})$ and check if $s_0 \in Sat_{EU}(p, q, \mathcal{M})$.

Init: $Q := \{s_2\}$, $Q' := \varnothing$

Iter 1: $Q' := \{s_2\}$, $Q := \{s_2\} \cup (\{s_0, s_1\} \cap \{s_1, s_3\}) = \{s_1, s_2\}$

Iter 2: $Q' := \{s_1, s_2\}$, $Q := \{s_1, s_2\} \cup (\{s_0, s_1\} \cap \{s_0, s_1, s_2, s_3\}) = \{s_0, s_1, s_2\}$

Iter 3: $Q' := \{s_0, s_1, s_2\}$,

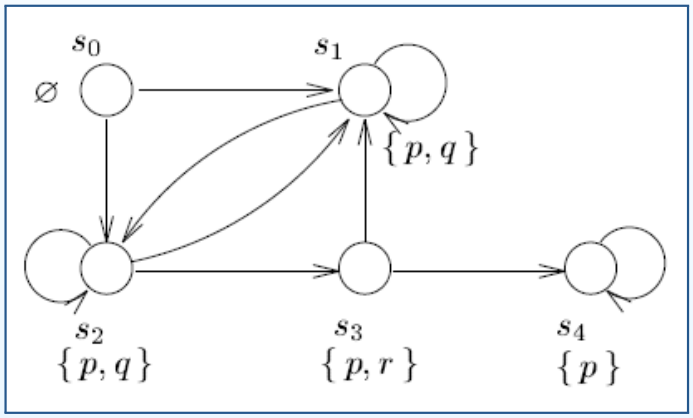
$Q := \{s_0, s_1, s_2\} \cup (\{s_0, s_1\} \cap \{s_0, s_1, s_2, s_3\}) = \{s_0, s_1, s_2\}$,

thus $Q = Q' = Sat_{EU}(p, q, \mathcal{M})$

Since $s_0 \in Sat_{EU}(p, q, \mathcal{M}) \Rightarrow \mathcal{M}, s_0 \models \mathsf{E}\,[p\,\mathsf{U}\,q]$

CTL MC Examples

- **Example 12.2**: Consider the model below, with initial state $\{s_0\}$. Check whether $\mathcal{M}, s_0 \models$ EF[AG p] using the previous CTL MC algorithm.



Solution: Bring the CTL formula to a shape containing only (some of) the operators $\neg, \vee$, EX, EU, AU and then perform the computation incrementally for subformulas (starting from elementary ones), until reaching the entire formula

EF[AG p] = EF[$\neg$EF$\neg p$] = EF[$\neg$E[$true$ U $\neg p$]] = E[$true$ U [$\neg$E[$true$ U $\neg p$]]]

$Sat(\neg p) = S - Sat(p) = S - \{s_1, s_2, s_3, s_4\} = \{s_0\}$

$Sat(\mathsf{E}[true\,\mathsf{U}\,\neg p]) = Sat_{EU}(true, \neg p) = \{s_0\}$

$Sat(\neg\mathsf{E}[true\,\mathsf{U}\,\neg p]) = S - Sat(\mathsf{E}[true\,\mathsf{U}\,\neg p]) = \{s_1, s_2, s_3, s_4\}$

$Sat(\mathsf{E}[true\,\mathsf{U}\,[\neg\mathsf{E}[true\,\mathsf{U}\,\neg p]]]) = \{s_0, s_1, s_2, s_3, s_4\}$

Alternative CTL MC Algorithm [4]

- Every CTL formula ϕ can be rewritten using only $\neg, \wedge, \mathsf{EX}, \mathsf{EG}, \mathsf{EU}$
- The algorithm recursivelly labels each state with all subformulas of ϕ that hold in that state
  - if the formula has the shape $\phi = p \in AP$, the labeling is done by L
  - if the formula has the shape $\phi = \neg\psi$, all states not labeled by ψ are labeled by ϕ
  - if the formula has the shape $\phi = \psi \wedge \eta$, all states labeled by both ψ and η are labeled by ϕ
  - if the formula has the shape $\phi = \mathsf{EX}\psi$, all predecessors of states labeled by ψ are labeled by ϕ
  - if the formula has the shape $\phi = \mathsf{E}[\psi\,\mathsf{U}\,\eta]$, all states labeled by η, as well as all predecessors of states labeled by ϕ that are themselves labeled by ψ are labeled by ϕ (while possible)
  - if the formula has the shape $\phi = \mathsf{EG}\psi$, the subgraph G_ψ of $\mathcal{M}$, containing all states labeled by ψ is considered. Each state in a non-trivial, strongly connected component of G_ψ is labeled by ϕ, as well as each predecessor of such a state (while possible)

Symbolic Model Checking [4]

- In case of sizable systems or complex specifications, the explicit-state model checking algorithms may not terminate, due to either insufficient memory or running time
- The above problem is known as the *state explosion problem*, being one of the main limitations of model checking
- An alternative to the explicit enumeration of all reachable states would be a state-space traversal which considers large numbers of states at a single step. This, in turn, requires efficient data structures for representing items such as sets of states or transition systems (formulas, binary decision diagrams, etc.). Such representations are called *symbolic*, and the model checking techniques employing them are referred as *simbolic model checking*
- Symbolic model checking taxonomy
  - techniques based on *(Reduced Order) Binary Decision Diagrams* (*ROBDDs*, data structures for the symbolic representation of sets)
  - *bounded model checking*

Model Checking Tools

- SPIN (**S**imple **P**romela **IN**terpreter) [6]
  - developed beginning with the 80's by members of the CS Research Center at Bell Labs, US
  - general tool for verifying the correctness of distributed systems
  - systems are described in Promela (**Pro**cess **Me**ta **La**nguage)
  - properties to be verified are expresses as PLTL formulas
  - the tool does also work as a simulator
  - freely available since 1991
  - applications / success stories
    - verification of control algorithms of a flood control barrier in Netherlands (late 90's)
    - verification of algorithms for space mission critical software (2000)
    - verification of medical device transmission protocols

Model Checking Tools

- NuSMV (new **S**ymbolic **M**odel **V**erifier) [7]
  - reimplementation and extension of SMV, the first model checking tool based on BDDs
  - developed as a joint project between Instituto Trentino di Cultura, Italy, Carnegie Mellon University, University of Genoa and University of Trento
  - aimed at the reliable verification of industrially sized designs, as a backend for other verification tools, as well as a research tool for formal verification techniques
  - allows checking both LTL and CTL specifications
  - freely available as an Eclipse plugin
- Java Pathfinder - used to verify Java bytecode programs
- Bandera - toolset for modelcheking concurrent Java software