CTL Model Checking

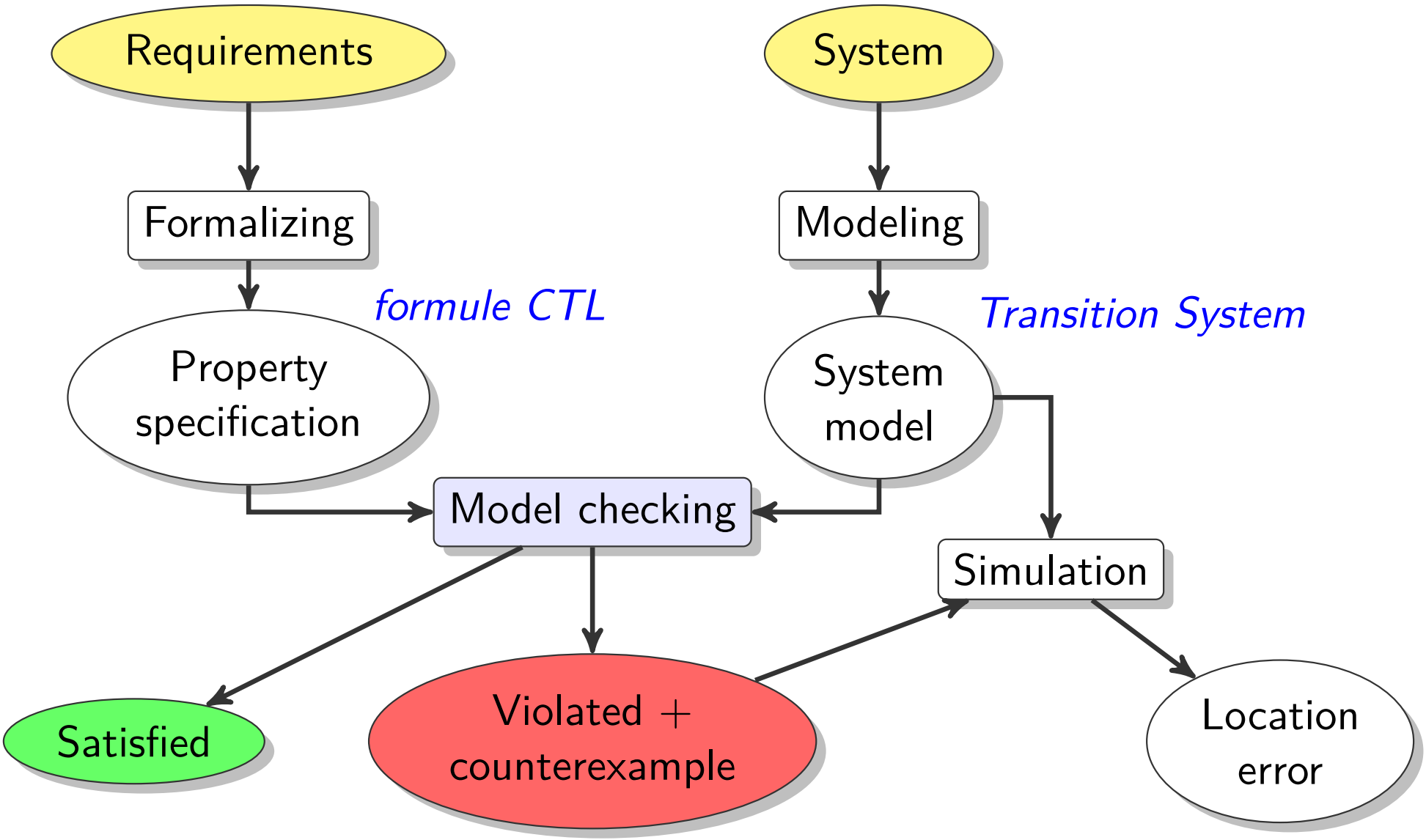
Marco Buracchi
marco.buracchi1@stud.unifi.it

Stefano Martina
stefano.martina@stud.unifi.it

UNIVERSITÀ
DEGLI STUDI
FIRENZE

February 22, 2016

Model checking



Computation Tree Logic (CTL)

Grammatica

$$\begin{aligned} \Phi &::= true | a | \Phi_1 \wedge \Phi_2 | \neg\Phi | \exists\varphi | \forall\varphi \\ \varphi &::= \bigcirc\Phi | \Phi_1 \, \mathbf{U} \, \Phi_2 \end{aligned}$$

- ✓ Φ sono state formula
- ✓ φ sono path formula

Operatori derivati

$$\begin{aligned} \exists\Diamond\Phi &= \exists(true \, \mathbf{U} \, \Phi) \\ \forall\Diamond\Phi &= \forall(true \, \mathbf{U} \, \Phi) \\ \exists\Box\Phi &= \neg\forall\Diamond\neg\Phi \\ \forall\Box\Phi &= \neg\exists\Diamond\neg\Phi \end{aligned}$$

CTL Existential Normal Form (ENF)

Grammatica

$$\Phi \quad ::= \quad true | a | \Phi_1 \wedge \Phi_2 | \neg\Phi | \exists \bigcirc \Phi | \exists(\Phi_1 \, \mathbf{U} \, \Phi_2) | \exists\Box\Phi$$

Equivalenze

$$\begin{aligned} \forall \bigcirc \Phi &\equiv \neg\exists \bigcirc \neg\Phi \\ \forall\Diamond\Phi &\equiv \neg\exists\Box\neg\Phi \\ \forall\Box\Phi &\equiv \neg\exists\Diamond\neg\Phi = \neg\exists(true \, \mathbf{U} \, \neg\Phi) \end{aligned}$$

✓ utile anche la formula vista prima: $\exists\Diamond\Phi = \exists(true \, \mathbf{U} \, \Phi)$

Equivalenze non banali

$\forall$ until

$$\forall(\Phi \mathbf{U} \Psi) \equiv \neg\exists(\neg\Psi \mathbf{U} (\neg\Phi \wedge \neg\Psi)) \wedge \neg\exists\Box\neg\Psi$$

Weak until

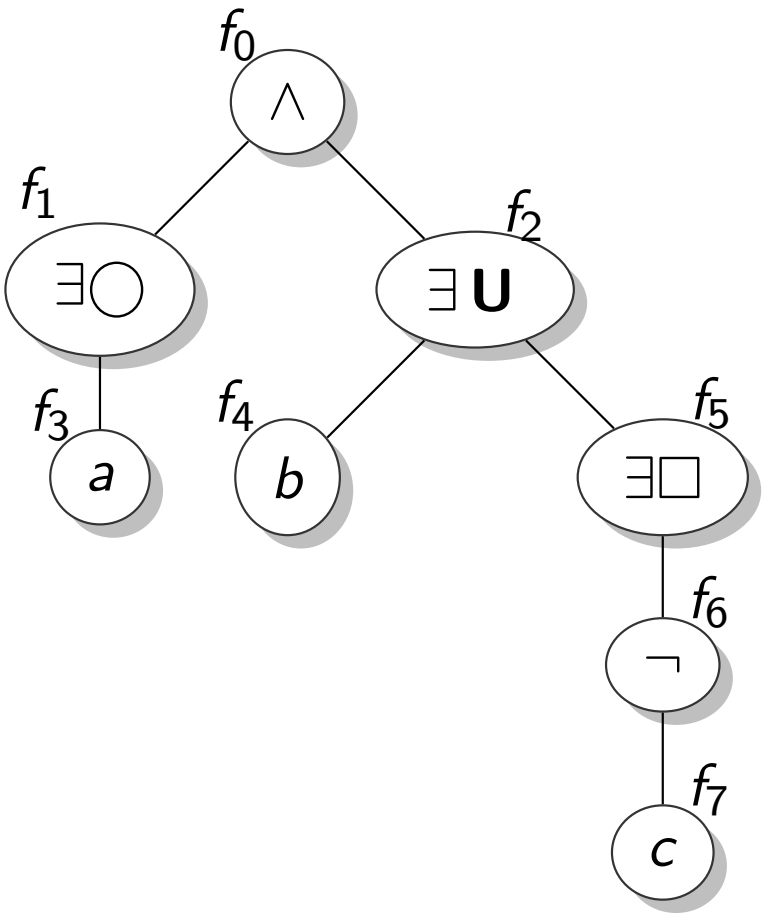
$$\begin{aligned}
\exists(\Phi \mathbf{W} \Psi) &= \neg\forall((\Phi \wedge \neg\Psi) \mathbf{U} (\neg\Phi \wedge \neg\Psi)) \\
&= \exists(\Phi \mathbf{U} \Psi) \vee \exists\Box\Phi \\
&= \neg(\neg\exists(\Phi \mathbf{U} \Psi) \wedge \neg\exists\Box\Phi) \\
\forall(\Phi \mathbf{W} \Psi) &= \neg\exists((\Phi \wedge \neg\Psi) \mathbf{U} (\neg\Phi \wedge \neg\Psi))
\end{aligned}$$

- ✓ Comportano esplosione esponenziale della formula

Formula example

Formula

$$\Phi = \exists \bigcirc a \wedge \exists (b \, \mathbf{U} \, \exists \Box \neg c) \equiv$$ `EX(a) & (b EU (EG !c))`



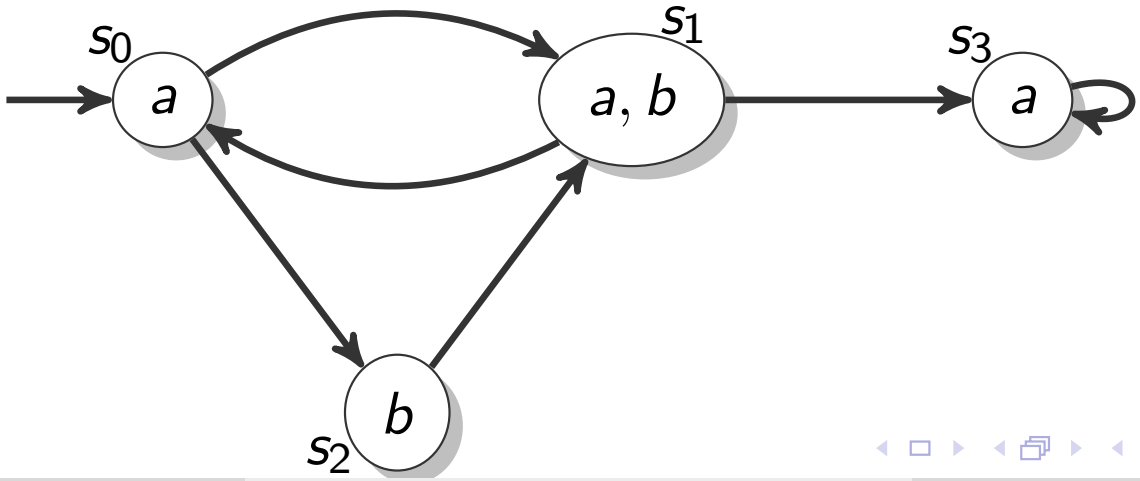

```
1   f0 &
2   f1 EX
3   f2 EU
4   f3 ap a
5   f4 ap b
6   f5 EG
7   f6 !
8   f7 ap c
9
10  f0 f1
11  f0 f2
12  f1 f3
13  f2 f4 <
14  f2 f5 >
15  f5 f6
16  f6 f7
```

Transition System (TS)

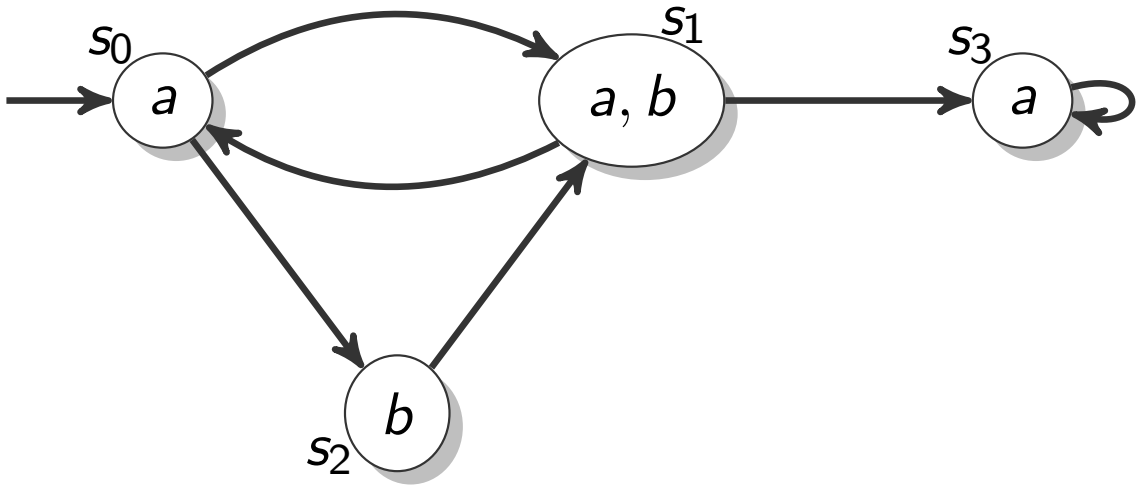
Definizione

$TS = (S, Act, \longrightarrow, I, AP, L)$

- ✓ S insieme di stati
- ✓ Act insieme di azioni (singola azione per CTL)
- ✓ $\longrightarrow \subseteq S \times Act \times S$
- ✓ $I \subseteq S$ insieme di stati iniziali
- ✓ AP insieme di proposizioni atomiche
- ✓ $L : S \to 2^{AP}$



Implementazione TS




```
1   s0 true a
2   s1 false a,b
3   s2 false b
4   s3 false a
5
6   s0 s1
7   s0 s2
8   s1 s0
9   s1 s3
10  s2 s1
11  s3 s3
```

CTL Model Checking

✓ Verificare se $TS \models \Phi$

- calcolare ricorsivamente $Sat(\Phi)$

$$
\begin{aligned}
Sat(true) &= S \\
Sat(a) &= \{s \in S | a \in L(s)\} \qquad , \forall a \in AP \\
Sat(\Phi \wedge \Psi) &= Sat(\Phi) \cap Sat(\Psi) \\
Sat(\neg\Phi) &= S \setminus Sat(\Phi) \\
Sat(\exists \bigcirc \Phi) &= \{s \in S | Post(s) \cap Sat(\Phi) \neq \varnothing\} \\
Sat(\exists(\Phi \,\mathbf{U}\, \Psi)) &= \text{il più piccolo } T \subseteq S \text{ t.c.} \\
&\quad Sat(\Psi) \cup \{s \in Sat(\Phi) | Post(s) \cap T \neq \varnothing\} \subseteq T \\
Sat(\exists\Box\Phi) &= \text{il più grande } T \subseteq S \text{ t.c.} \\
&\quad T \subseteq \{s \in Sat(\Phi) | Post(s) \cap T \neq \varnothing\}
\end{aligned}
$$

  - ★ formule non ENF vengono convertite

- $TS \models \Phi \Leftrightarrow I \subseteq Sat(\Phi)$

Complessità CTL Model Checking

Algoritmi di base $Sat(\cdot)$, formula ENF

$$\mathcal{O}((N+K)\cdot|\Phi|)$$

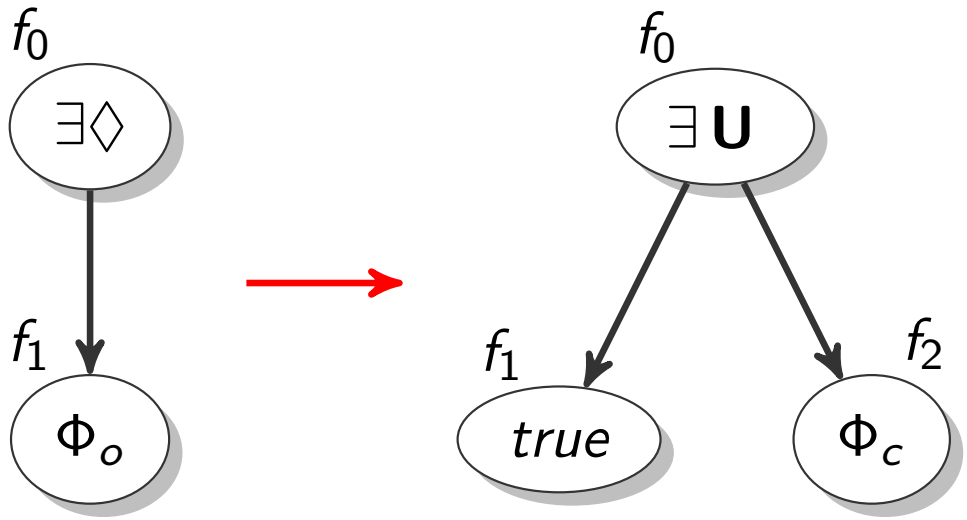
- ✓ N numero di stati del TS
- ✓ K numero di transizioni del TS
- ✓ $|\Phi|$ lunghezza della formula Φ

Complessità $TS \models \Phi$

- ✓ La trasformazione da formula CTL generica a CTL ENF sarebbe esponenziale, però:
  - Esistono algoritmi lineari analoghi a quelli per formule ENF
  - Implementata conversione/calcolo lineare di $Sat(\cdot)$ per formule generiche
- ✓ La complessità totale rimane invariata

Formule non ENF

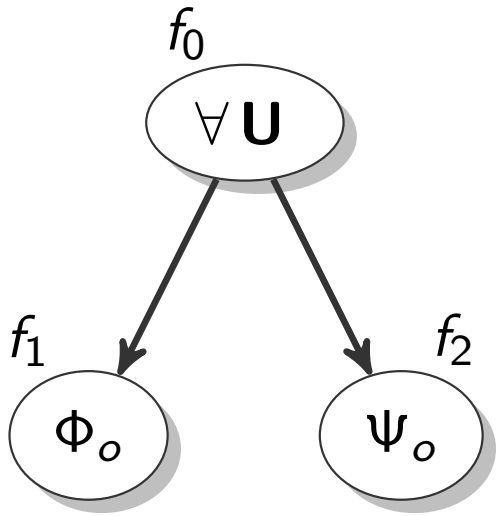
- ✓ Alberi di conversione generici
  - Foglie speciali per agganciarci alla formula originale

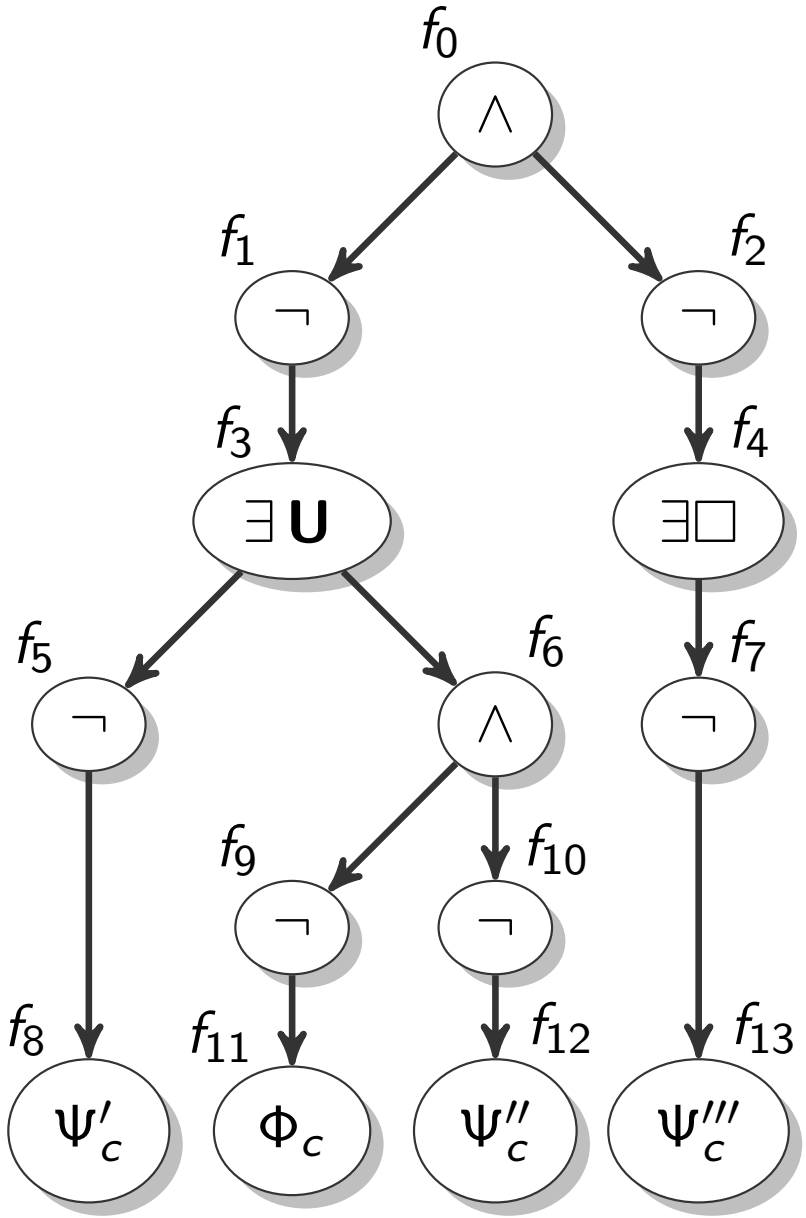


1. Calcolo di $Sat(\Phi_o)$
2. Salva risultato in nodo speciale Φ_c
   - Evita esplosione della formula $\forall(\Phi\,\mathbf{U}\,\Psi)$
3. Calcolo di $Sat(\exists\,\mathbf{U}\,)$ su albero di conversione

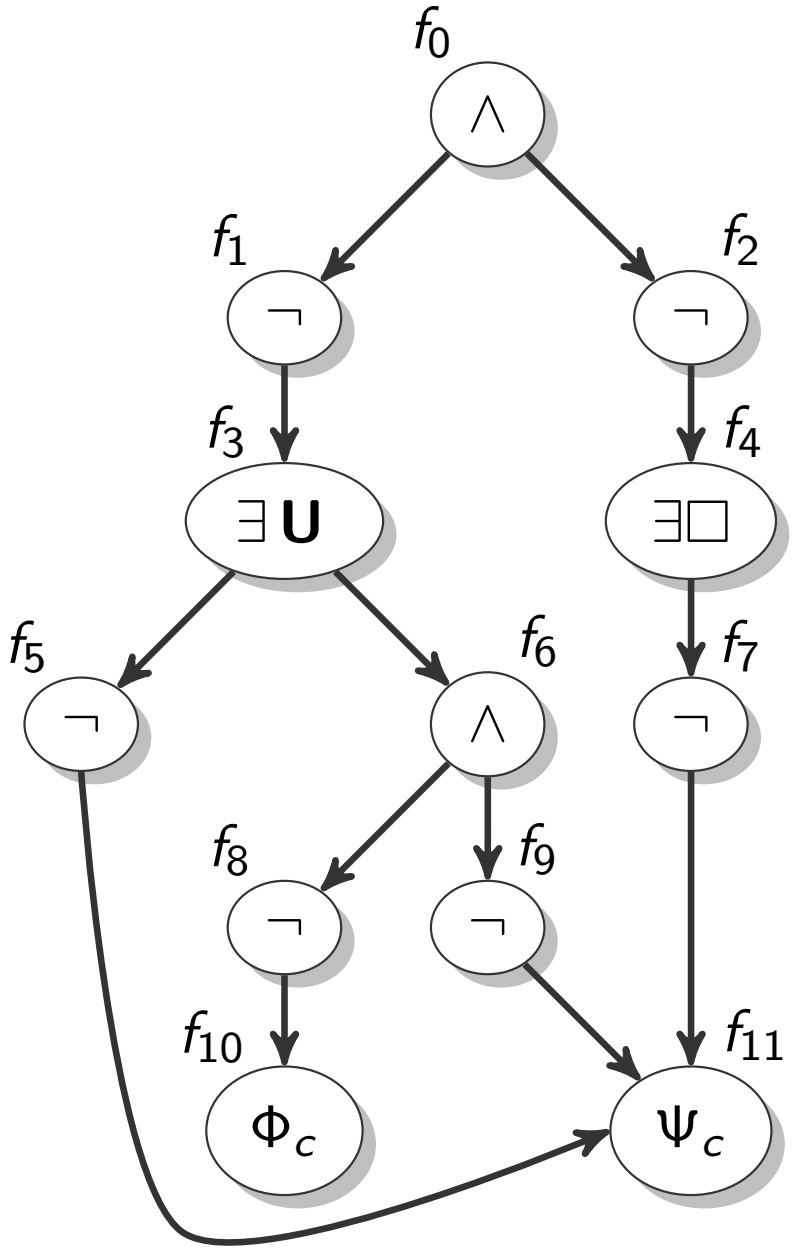
$\forall\,\mathbf{U}$ explosion

- ✓ Formula da convertire:





✓ $Sat(\Psi)$ da calcolare tre volte



✓ $Sat(\Psi)$ calcolata una volta

Questions? Thank you!

Implementazione costruttore

```
10      def __init__(self, transitionSystem):
11          """
12          A ctlChecker is linked to a certain transition system. When
13          you build a CtlChecker you need to pass a TransitionSystem to
14          it.
15          """
16          self._syntax = syntax.Syntax()
17          self._conv = conversions.Conversions()
18          self._ts = transitionSystem
19
20          self._callDic = {
21              self._syntax.true           :  self._satTrue,
22              self._syntax.ap             :  self._satAp,
23              self._syntax.land           :  self._satAnd,
24              self._syntax.lor            :  self._satConversionTwoSons,
25              self._syntax.lnot           :  self._satNot,
26              self._syntax.implies        :  self._satConversionTwoSonsOrdered,
27              self._syntax.equals         :  self._satConversionTwoSons,
28              self._syntax.exNext         :  self._satExNext,
29              self._syntax.exUntil        :  self._satExUntil,
30              self._syntax.exAlways       :  self._satExAlways,
31              self._syntax.exEventually   :  self._satConversionOneSon,
32              self._syntax.faNext         :  self._satConversionOneSon,
33              self._syntax.faUntil        :  self._satConversionTwoSonsOrdered,
34              self._syntax.faAlways       :  self._satConversionOneSon,
35              self._syntax.faEventually   :  self._satConversionOneSon,
36              self._syntax.exWeakUntil    :  self._satConversionTwoSonsOrdered,
37              self._syntax.faWeakUntil    :  self._satConversionTwoSonsOrdered,
38              self._syntax.phiNode        :  self._satPhi,
39          }
```

Implementazione $Sat(\cdot)$

```
200     def sat(self, ctlFormule):
201         """
202         The function that compute the satisfaction set of a formula
203         and is callable by outside the class. Basically it calls _sat
204         initializing the current node with the root of the formula.
205         """
206         return self._sat(ctlFormule.graph.copy(), [s for s,a in
                ↪ ctlFormule.graph.nodes(data=True) if a['root'] ==True][0])
```

```
191     def _sat(self, tree, currNode):
192         """
193         The generic function for the calculus of the satisfaction set
194         of a formula. It uses a dictionary for calling the right
195         function for the type of formula.
196         """
197         if (tree.node[currNode]['form'] in self._callDic.keys()) :
198             return self._callDic[tree.node[currNode]['form']](tree, currNode)
```

Implementazione $Sat(\cdot)$

```
41      def _satTrue(self, tree, currNode):
42          """
43          Return satisfation set for 'true', that is all the nodes of
44          the transition system.
45          """
46          return set(self._ts.graph.nodes())
```

```
48      def _satAp(self, tree, currNode):
49          """
50          Return satisfation set for an atomic proposition, that is all
51          the nodes of the transition system that contain that atom.
52          """
53          retSet = set()
54          for stato,att in self._ts.graph.nodes(data=True):
55              if tree.node[currNode]['val'] in att['att']:
56                  retSet.add(stato)
57
58          return retSet
```

```
60      def _satAnd(self, tree, currNode):
61          """
62          Return satisfation set for 'phi and psi', that is the
63          intersection of the satisfation sets of phi and psi.
64          """
65          sonA = tree.successors(currNode)[0]
66          sonB = tree.successors(currNode)[1]
67
68          return self._sat(tree, sonA).intersection(self._sat(tree, sonB))
```

Implementazione $Sat(\cdot)$

```
70      def _satNot(self, tree, currNode):
71          """
72          Return satisfation set for 'not phi', that is the complement
73          of the satisfation set of phi.
74          """
75          return set(self._ts.graph.nodes()).difference(self._sat(tree,
                ↪ tree.successors(currNode)[0]))
```

```
77      def _satExNext(self, tree, currNode):
78          """
79          Return satisfation set for 'EX phi', that is the set of nodes that
80          have a successor that satisfy phi.
81          """
82          retSet = set()
83          satPhi = self._sat(tree, tree.successors(currNode)[0])
84
85          for stato in self._ts.graph.nodes():
86              if set(self._ts.graph.successors(stato)).intersection(satPhi): #true if
                    ↪ not empty
87                  retSet.add(stato)
88
89          return retSet
```

Implementazione $Sat(\cdot)$

```
91      def _satExUntil(self, tree, currNode):
92          """
93          Return satisfation set for 'E(phi U psi)', that is the set of nodes that
94          have a track that satisfy phi ended by a state that satisfy
95          psi. This set is calculated going backward starting from the
96          states that satisfy psi, adding the states that satisfy phi
97          and have an edge to the already found states.
98          """
99          leftSon = [x for x in tree[currNode] if tree[currNode][x]['son'] ==
                ↪ self._syntax.leftSon][0]
100         rightSon = [x for x in tree[currNode] if tree[currNode][x]['son'] ==
                ↪ self._syntax.rightSon][0]
101
102         S = self._sat(tree, leftSon)
103         E = self._sat(tree, rightSon)
104         T = E.copy()
105
106         while E: #while not empty
107             r = E.pop()
108             for s in self._ts.graph.predecessors(r):
109                 if s in S.difference(T):
110                     E.add(s)
111                     T.add(s)
112         return T
```

Implementazione $Sat(\cdot)$

```
115         def _satExAlways(self, tree, currNode):
116             """
117             Return satisfation set for 'EG phi', that is the set of nodes
118             that have a track that satisfy always phi. This set is
119             calculated using counters that start with the numbers of
120             neighbours for each state that satisfy phi, and then remove a
121             state from the set only if every neighbour don't satisfy phi.
122             """
123 
124             T = self._sat(tree, tree.successors(currNode)[0])
125             E = set(self._ts.graph.nodes()).difference(T)
126             count = dict()
127             for s in T:
128                 count[s] = len(self._ts.graph.successors(s))
129 
130             while E:
131                 r = E.pop()
132                 for s in self._ts.graph.predecessors(r):
133                     if s in T:
134                         count[s] = count[s]-1
135                         if count[s] == 0:
136                             T.remove(s)
137                             E.add(s)
138 
139             return T
```

Implementazione $Sat(\cdot)$ formule non ENF

```
141     def _satConversionOneSon(self, tree, currNode):
142         """
143         Return satisfation set for every conversion tree that has only
144         one son phi. That is calculated saving the satisfation set of
145         phi in a special node of the conversion tree before calling
146         the calculus.
147         """
148         form = tree.node[currNode]['form']
149         son = tree.successors(currNode)[0]
150 
151         self._conv.trees[form].node[self._conv.phis[form]]['sat'] = self._sat(tree,
                ↪ son)
152         return self._sat(self._conv.trees[form], self._conv.roots[form])
```

```
154     def _satConversionTwoSons(self, tree, currNode):
155         """
156         Return satisfation set for every conversion tree that has two
157         sons phi and psi. That is calculated saving the satisfation
158         sets of phi and psi in special nodes of the conversion tree
159         before calling the calculus.
160         """
161         form = tree.node[currNode]['form']
162         sonA = tree.successors(currNode)[0]
163         sonB = tree.successors(currNode)[1]
164 
165         self._conv.trees[form].node[self._conv.phis[form]]['sat'] = self._sat(tree,
                ↪ sonA)
166         self._conv.trees[form].node[self._conv.psis[form]]['sat'] = self._sat(tree,
                ↪ sonB)
167         return self._sat(self._conv.trees[form], self._conv.roots[form])
```

Implementazione $Sat(\cdot)$ formule non ENF

```
169     def _satConversionTwoSonsOrdered(self, tree, currNode):
170         """
171         Return satisfation set for every conversion tree that has two
172         sons phi and psi whit a proper order. That is calculated
173         saving the satisfation sets of phi and psi in special nodes of
174         the conversion tree before calling the calculus.
175         """
176         form = tree.node[currNode]['form']
177         leftSon = [x for x in tree[currNode] if tree[currNode][x]['son'] ==
                ↪ self._syntax.leftSon][0]
178         rightSon = [x for x in tree[currNode] if tree[currNode][x]['son'] ==
                ↪ self._syntax.rightSon][0]
179 
180         self._conv.trees[form].node[self._conv.phis[form]]['sat'] = self._sat(tree,
                ↪ leftSon)
181         self._conv.trees[form].node[self._conv.psis[form]]['sat'] = self._sat(tree,
                ↪ rightSon)
182         return self._sat(self._conv.trees[form], self._conv.roots[form])
```

Implementazione $TS \models \Phi$

```
208         def check(self, ctlFormule):
209             """
210             The function that do the model checking. It check if all the
211             initial states of the transition system are contained inside
212             the calculated satisfation set.
213             """
214             sats = self.sat(ctlFormule)
215             initials = set([s for s,a in self._ts.graph.nodes(data=True) if
                    ↪ a['initial'] ==True])
216             return initials.issubset(sats)
```

Complessità rispetto a Linear Temporal Logic (LTL)

1. Problema Hamiltoniano (HP) è NP-Completo
   - Trovare un percorso in un certo grafo G che passa esattamente una volta per ogni nodo

2a. è possibile descrivere TS_G e una formula LTL Φ_{LTL} di lunghezza polinomiale nel numero di stati del grafo del problema
   - tali che $TS_G \not\models \Phi_{LTL} \quad \Leftrightarrow \quad G$ contiene un percorso hamiltoniano

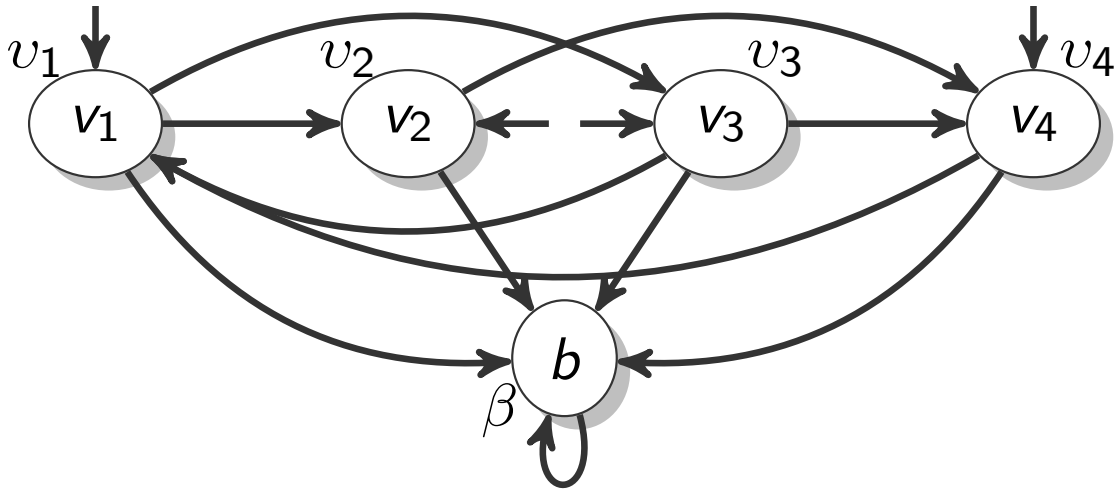
2b. il model checking di Linear Temporal Logic (LTL) ha complessità esponenziale in $|\Phi|$

3a. è possibile descrivere un TS_G e una formula CTL Φ_{CTL} di lunghezza esponenziale nel numero di stati del grafo
   - tali che $TS_G \not\models \neg\Phi_{CTL} \quad \Leftrightarrow \quad G$ contiene un percorso hamiltoniano

3b. il model checking di CTL ha complessità lineare in $|\Phi|$

4. se $P \neq NP$ non esiste una formula Φ_{CTL} equivalente a Φ_{LTL} e non esponenzialmente più lunga.

Implementazione TS_G, Φ_{LTL}, Φ_{CTL}



$$\Phi_{LTL} = \neg \bigwedge_{v \in V} (\Diamond v \wedge \Box(v \rightarrow \bigcirc \Box \neg v))$$

$$\Phi_{CTL} = \bigvee_{(i_1,\ldots,i_n)} \Psi(v_{i_1}, \ldots, v_{i_n}) \quad \in \mathcal{O}(n!)$$

dove n è il numero di stati, e

$\Psi(v_i) = v_i$; $\quad \Psi(v_{i_1}, v_{i_2}, \ldots, v_{i_n}) = v_{i_1} \wedge \exists \bigcirc \Psi(v_{i_2}, \ldots, v_{i_n})$.