CTL Model Checking

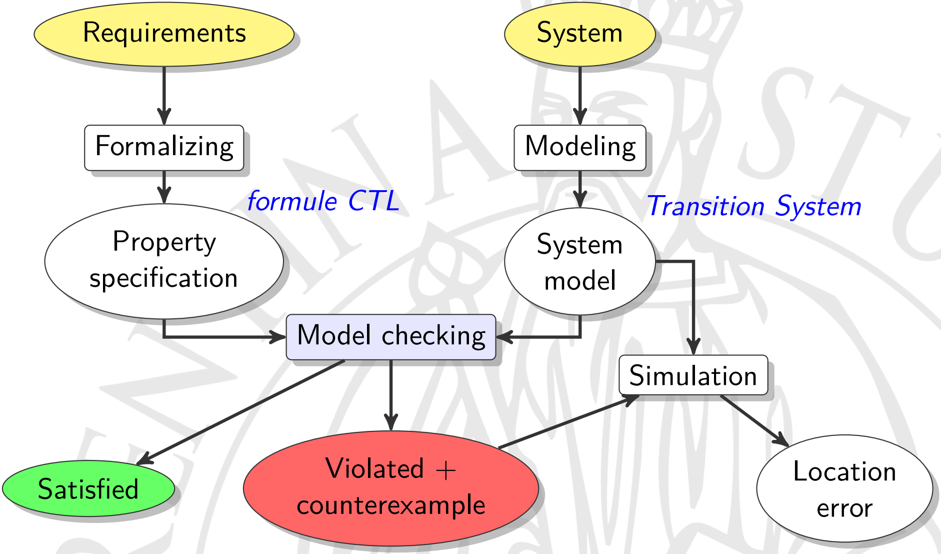
Marco Buracchi
marco.buracchi1@stud.unifi.it

Stefano Martina
stefano.martina@stud.unifi.it

UNIVERSITÀ
DEGLI STUDI
FIRENZE

January 25, 2016

Model checking



Computation Tree Logic (CTL)

Grammatica

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

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

Operatori derivati

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

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

∀ 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

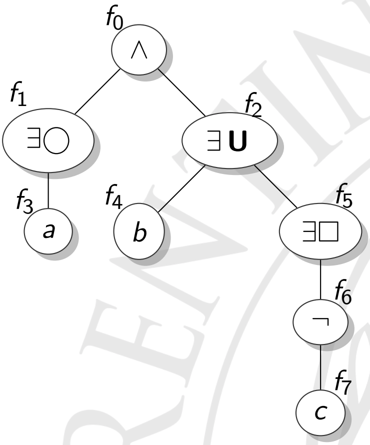
$$\begin{aligned}
\exists(\Phi\,\mathbf{W}\,\Psi) &= \neg\forall((\Phi\wedge\neg\Psi)\,\mathbf{U}\,(\neg\Phi\wedge\neg\Psi)) \\
&= \exists(\Phi\,\mathbf{U}\,\Psi)\vee\exists\Box\Phi \\
&= \neg(\neg\exists(\Phi\,\mathbf{U}\,\Psi)\wedge\neg\exists\Box\Phi) \\
\forall(\Phi\,\mathbf{W}\,\Psi) &= \neg\exists((\Phi\wedge\neg\Psi)\,\mathbf{U}\,(\neg\Phi\wedge\neg\Psi))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

✓ Comportano esplosione esponenziale della formula

Formula example

Formula

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



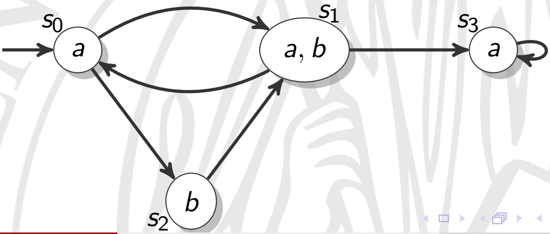

```
f0 &
f1 EX
f2 EU
f3 ap a
f4 ap b
f5 EG
f6 !
f7 ap c

f0 f1
f0 f2
f1 f3
f2 f4 <
f2 f5 >
f5 f6
f6 f7
```

Transition System (TS)

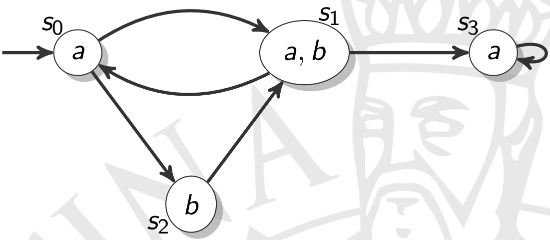
Definizione

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

- ✓ 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
- ✓ $L : S \rightarrow 2^{AP}$



Implementazione TS




```
s0 true a
s1 false a,b
s2 false b
s3 false a

s0 s1
s0 s2
s1 s0
s1 s3
s2 s1
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

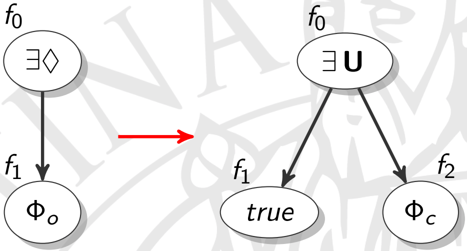
Complessità $TS \models \Phi$

- ✓ 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

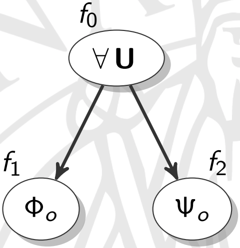
- Foglie speciali per agganciarci alla formula originale

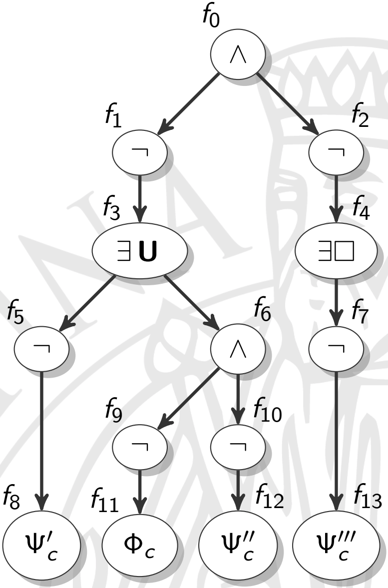


1. Calcolo di $Sat(\Phi_o)$
2. Salva risultato in nodo speciale Φ_c
   - Evita esplosione della formula $\forall(\Phi\,\mathbf{U}\,\Psi)$
3. Calcolo di $Sat(\exists\,\mathbf{U}\,)$ su albero di conversione

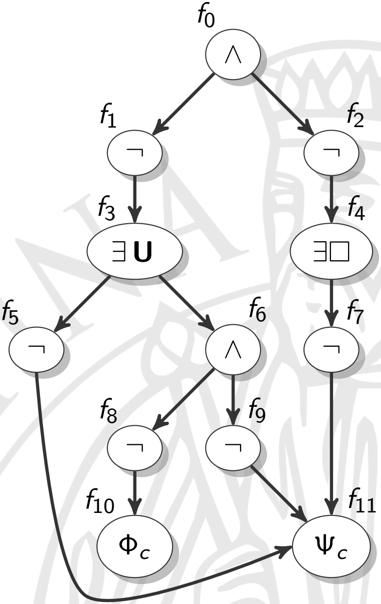
$\forall$ **U** explosion

✓ Formula da convertire:





- ✓ *Sat*(Ψ) da calcolare tre volte



✓ $Sat(\Psi)$ calcolata una volta

The End.

Questions? Thank you!

Implementazione costruttore

```python
    def __init__(self, transitionSystem):
        """
        A ctlChecker is linked to a certain transition system. When
        you build a CtlChecker you need to pass a TransitionSystem to
        it.
        """
        self._syntax = syntax.Syntax()
        self._conv = conversions.Conversions()
        self._ts = transitionSystem

        self._callDic = {
            self._syntax.true          :  self._satTrue,
            self._syntax.ap            :  self._satAp,
            self._syntax.land          :  self._satAnd,
            self._syntax.lor           :  self._satConversionTwoSons,
            self._syntax.lnot          :  self._satNot,
            self._syntax.implies       :  self._satConversionTwoSonsOrdered,
            self._syntax.equals        :  self._satConversionTwoSons,
            self._syntax.exNext        :  self._satExNext,
            self._syntax.exUntil       :  self._satExUntil,
            self._syntax.exAlways      :  self._satExAlways,
            self._syntax.exEventually  :  self._satConversionOneSon,
            self._syntax.faNext        :  self._satConversionOneSon,
            self._syntax.faUntil       :  self._satConversionTwoSonsOrdered,
            self._syntax.faAlways      :  self._satConversionOneSon,
            self._syntax.faEventually  :  self._satConversionOneSon,
            self._syntax.exWeakUntil   :  self._satConversionTwoSonsOrdered,
            self._syntax.faWeakUntil   :  self._satConversionTwoSonsOrdered,
            self._syntax.phiNode       :  self._satPhi,
        }
```

Implementazione $Sat(\cdot)$

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

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

Implementazione $Sat(\cdot)$

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

```python
    def _satAp(self, tree, currNode):
        """
        Return satisfation set for an atomic proposition, that is all
        the nodes of the transition system that contain that atom.
        """
        retSet = set()
        for stato,att in self._ts.graph.nodes(data=True):
            if tree.node[currNode]['val'] in att['att']:
                retSet.add(stato)

        return retSet
```

```python
    def _satAnd(self, tree, currNode):
        """
        Return satisfation set for 'phi and psi', that is the
        intersection of the satisfation sets of phi and psi.
        """
        sonA = tree.successors(currNode)[0]
        sonB = tree.successors(currNode)[1]

        return self._sat(tree, sonA).intersection(self._sat(tree, sonB))
```

Implementazione $Sat(\cdot)$

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

```
    def _satExNext(self, tree, currNode):
        """
        Return satisfation set for 'EX phi', that is the set of nodes that
        have a successor that satisfy phi.
        """
        retSet = set()
        satPhi = self._sat(tree, tree.successors(currNode)[0])

        for stato in self._ts.graph.nodes():
            if set(self._ts.graph.successors(stato)).intersection(satPhi): #true if
                ↪ not empty
                retSet.add(stato)

        return retSet
```

Implementazione $Sat(\cdot)$

```python
    def _satExUntil(self, tree, currNode):
        """
        Return satisfation set for 'E(phi U psi)', that is the set of nodes that
        have a track that satisfy phi ended by a state that satisfy
        psi. This set is calculated going backward starting from the
        states that satisfy psi, adding the states that satisfy phi
        and have an edge to the already found states.
        """
        leftSon = [x for x in tree[currNode] if tree[currNode][x]['son'] == self._syntax.leftSon][0]
        rightSon = [x for x in tree[currNode] if tree[currNode][x]['son'] == self._syntax.rightSon][0]

        S = self._sat(tree, leftSon)
        E = self._sat(tree, rightSon)
        T = E.copy()

        while E: #while not empty
            r = E.pop()
            for s in self._ts.graph.predecessors(r):
                if s in S.difference(T):
                    E.add(s)
                    T.add(s)
        return T
```

Implementazione $Sat(\cdot)$

```python
    def _satExAlways(self, tree, currNode):
        """
        Return satisfation set for 'EG phi', that is the set of nodes
        that have a track that satisfy always phi. This set is
        calculated using counters that start with the numbers of
        neighbours for each state that satisfy phi, and then remove a
        state from the set only if every neighbour don't satisfy phi.
        """

        T = self._sat(tree, tree.successors(currNode)[0])
        E = set(self._ts.graph.nodes()).difference(T)
        count = dict()
        for s in T:
            count[s] = len(self._ts.graph.successors(s))

        while E:
            r = E.pop()
            for s in self._ts.graph.predecessors(r):
                if s in T:
                    count[s] = count[s]-1
                    if count[s] == 0:
                        T.remove(s)
                        E.add(s)

        return T
```

Implementazione $Sat(\cdot)$ formule non ENF

```python
    def _satConversionOneSon(self, tree, currNode):
        """
        Return satisfation set for every conversion tree that has only
        one son phi. That is calculated saving the satisfation set of
        phi in a special node of the conversion tree before calling
        the calculus.
        """
        form = tree.node[currNode]['form']
        son = tree.successors(currNode)[0]

        self._conv.trees[form].node[self._conv.phis[form]]['sat'] = self._sat(tree,
             son)
        return self._sat(self._conv.trees[form], self._conv.roots[form])
```

```python
    def _satConversionTwoSons(self, tree, currNode):
        """
        Return satisfation set for every conversion tree that has two
        sons phi and psi. That is calculated saving the satisfation
        sets of phi and psi in special nodes of the conversion tree
        before calling the calculus.
        """
        form = tree.node[currNode]['form']
        sonA = tree.successors(currNode)[0]
        sonB = tree.successors(currNode)[1]

        self._conv.trees[form].node[self._conv.phis[form]]['sat'] = self._sat(tree,
             sonA)
        self._conv.trees[form].node[self._conv.psis[form]]['sat'] = self._sat(tree,
             sonB)
        return self._sat(self._conv.trees[form], self._conv.roots[form])
```

Implementazione $Sat(\cdot)$ formule non ENF

```
    def _satConversionTwoSonsOrdered(self, tree, currNode):
        """
        Return satisfation set for every conversion tree that has two
        sons phi and psi whit a proper order. That is calculated
        saving the satisfation sets of phi and psi in special nodes of
        the conversion tree before calling the calculus.
        """
        form = tree.node[currNode]['form']
        leftSon = [x for x in tree[currNode] if tree[currNode][x]['son'] == self._syntax.leftSon][0]
        rightSon = [x for x in tree[currNode] if tree[currNode][x]['son'] == self._syntax.rightSon][0]

        self._conv.trees[form].node[self._conv.phis[form]]['sat'] = self._sat(tree, leftSon)
        self._conv.trees[form].node[self._conv.psis[form]]['sat'] = self._sat(tree, rightSon)
        return self._sat(self._conv.trees[form], self._conv.roots[form])
```

Implementazione $TS \models \Phi$

```python
    def check(self, ctlFormule):
        """
        The function that do the model checking. It check if all the
        initial states of the transition system are contained inside
        the calculated satisfation set.
        """
        sats = self.sat(ctlFormule)
        initials = set([s for s,a in self._ts.graph.nodes(data=True) if
             a['initial'] ==True])
        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

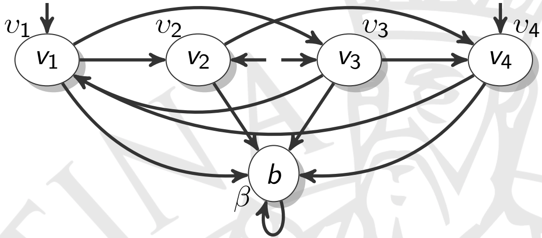
2b. il model checking di Linear Temporal Logic (LTL) ha complessità esponenziale in $|\Phi|$

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}



$$\begin{aligned}\Phi_{LTL} &= \neg \bigwedge_{\upsilon \in V} (\Diamond \upsilon \wedge \Box(\upsilon \rightarrow \bigcirc \Box \neg \upsilon)) \\ \Phi_{CTL} &= \bigvee_{(i_1,\ldots,i_n)} \Psi(\upsilon_{i_1}, \ldots, \upsilon_{i_n}) \quad \in \mathcal{O}(n!)\end{aligned}$$

dove n è il numero di stati, e
$\Psi(\upsilon_i) = \upsilon_i; \quad \Psi(\upsilon_{i_1}, \upsilon_{i_2}, \ldots, \upsilon_{i_n}) = \upsilon_{i_1} \wedge \exists \bigcirc \Psi(\upsilon_{i_2}, \ldots, \upsilon_{i_n}).$