Relazione per l'approfondimento su Computational Tree Logic

Filippo Mameli, Federico Schipani

31 gennaio 2017

Indice

1 Introduzione a Computational Tree Logic (CTL) 1
1.1 Sintassi di Computational Tree Logic (CTL) . . . 2
1.2 Semantica di CTL . . . 3

2 Model Checking di CTL 3
2.1 Complessità dell'algoritmo . . . 6

3 Model Checker in Python per CTL 7
3.1 Come vengono rappresentati i Transition System (TS) . . . 7
3.2 Come vengono rappresentate le formule . . . 8
3.2.1 Il parsing della formula . . . 9
3.2.2 La conversione in ENF . . . 10
3.3 Gli algoritmi di model checking . . . 11
3.4 Una versione migliorata del Model Checker . . . 14

4 Esempi di utilizzo del Model Checker di CTL in Python 15
4.1 Formula $\exists \Diamond \phi$ . . . 15
4.2 Formula $\exists \Box b$ . . . 15
4.3 Fisolofi a cena . . . 17

1 Introduzione a CTL

Computational Tree Logic (CTL) è una logica proposta da Clarke e Emerson per far fronte ad alcuni problemi noti di Linear Temporal Logic (LTL). In LTL il concetto di tempo è lineare, ciò vuol dire che in un determinato istante abbiamo un unico possibile futuro. Ciò comporta che una determinata formula ϕ è valida in uno stato s, se e solo se tutte le possibili computazioni che partono da quello stato soddisfano la formula. Più formalmente:

$$s \vDash \phi \iff \pi \vDash \phi \ \forall \ \mathtt{path}\ \pi\ \mathtt{che}\ \mathtt{inizia}\ \mathtt{in}\ s \tag{1.0.1}$$

Come si può notare dalla Formula (1.0.1) non è possibile imporre facilmente condizioni di soddisfacibilità solo su alcuni di questi path. Dato uno stato s, si può verificare che solo alcune computazioni soddisfano una formula ϕ

usando la dualità tra l'operatore universale ed esistenziale. Quindi verificare $s \models \exists\phi$ corrisponde a verificare $s \models \neg\forall\neg\phi$. Se quest'ultima non è soddisfatta allora esisterà una computazione che soddisfa ϕ, altrimenti non esisterà.

Non è possibile usare questo sotterfugio per proprietà più complicate. Per esempio la proprietà

Proprietà 1. *Per ogni computazione è sempre possibile ritornare in uno stato iniziale*

non è esprimibile in LTL. Un tentativo potrebbe essere $\Box\Diamond start$, dove $start$ indica uno stato iniziale. Tuttavia una formula di questo tipo è troppo forte, in quanto impone che una computazione ritorni sempre in uno stato iniziale, e non soltanto eventualmente.

CTL risolve questi problemi introducendo una nozione di tempo che si basa sulle diramazioni. Quindi non abbiamo più un'infinita sequenza di stati, ma un infinito albero di stati. Questo comporta che in un determinato istante avremo diversi possibili futuri.

La semantica di questa logica è definita in termini di infiniti alberi, dove ogni diramazione rappresenta un singolo percorso. L'albero quindi è una fedele rappresentazione di tutti i possibili path, e si può facilmente ottenere srotolando il TS.

In CTL sono presenti quantificatori, definiti sui path, di tipo esistenziale ($\exists$) ed universale ($\forall$). La Proprietà $\exists\Diamond\psi$ dice che esiste una computazione che soddisfa $\Diamond\psi$, più intuitivamente vuol dire che esisterà almeno una possibile computazione nel quale uno stato s che soddisfa ψ verrà eventualmente raggiunto. Tuttavia questo non esclude la possibilità che ci possono essere computazioni per le quali questa proprietà non viene soddisfatta. La proprietà 1 citata in precedenza è possibile ottenerla annidando quantificatori esistenziali ed universali in questo modo:

$$\forall\,\Box\,\exists\Diamond start \tag{1.0.2}$$

La Formula (1.0.2) si legge come: in ogni stato ($\Box$) di ogni possibile computazione ($\forall$), è possibile ($\exists$) eventualmente ritornare in uno stato iniziale ($\Diamond start$).

1.1 Sintassi di CTL

CTL ha una sintassi a due livelli, dove le formule sono classificate in *formule sugli stati* e *formule sui path*.

$$\boldsymbol{\phi} ::= \text{true} \mid a \mid \boldsymbol{\phi}_1 \wedge \boldsymbol{\phi}_2 \mid \neg\boldsymbol{\phi} \mid \exists\varphi \mid \forall\varphi$$

Grammatica 1: Grammatica per le formule sugli stati

Le prime sono definite dalla Grammatica 1 dove $a \in AP$ e φ è una formula sui path.

$$\varphi ::= \bigcirc \phi \mid \phi_1 \mathbin{\mathcal{U}} \phi_2$$

Grammatica 2: Grammatica per le formule sui path

Le *formule sui path* sono invece definite dalla Grammatica 2. Intuitivamente si può dire che le formule sugli stati esprimono una proprietà su uno stato, mentre le formule sui path esprimono proprietà sui infinite sequenze di stati. Per esempio la formula $\bigcirc \phi$ è vera per un path se lo stato successivo, in quel path, soddisfa ϕ. Una formula sugli stati può essere trasformata in una formula sui path aggiungendo all'inizio un quantificatore esistenziale ($\exists$) o universale ($\forall$). Per esempio la formula $\exists\varphi$ è valida in uno stato se esiste almeno un percorso che soddisfa φ.

1.2 Semantica di CTL

Le formule CTL sono interpretate sia sugli stati che sui path di un TS. Formalmente, dato un TS, la semantica di una formula è definita da due relazioni di soddisfazione: una per le formule di stato ed una per le formule di path. Per le formule di stato è un tipo di relazione tra gli stati del TS e la formula di stato. Si scrive che $s \vDash \phi$ se e solo se la formula di stato ϕ è vera nello stato s.

Per le formule di path la relazione $\vDash$ è una relazione definita tra un frammento di path massimale nel TS e una formula di path. Si scrive che $\pi \vDash \varphi$ se e solo se il path π soddisfa la formula φ.

Definizione 1. *Sia* $a \in AP$ *una proposizione atomica,* $TS = (S, act, \to, I, AP, L)$ *un Transition System (TS) senza stati terminali, stati* $s \in S$, ϕ, Ψ *formule CTL di stato e* φ *una formula CTL di path. La relazione di soddisfazione* $\vDash$ *per le formule di stato è definita come:*

$$s \vDash a \iff a \in L(s)$$

$$s \vDash \neg\phi \iff \text{not } s \vDash \phi$$

$$s \vDash \phi \wedge \Psi \iff (s \vDash \phi) \text{ e } (s \vDash \Psi)$$

$$s \vDash \exists\varphi \iff \pi \vDash \varphi \text{ per alcuni } \pi \in Paths(s)$$

$$s \vDash \forall\varphi \iff \pi \vDash \text{ per tutti i } \pi \in Paths(s)$$

Per un path π*, la relazione di soddisfazione* $\vDash$ *per le formule di path è definita da:*

$$\pi \vDash \bigcirc \phi \iff \pi[1] \vDash \phi$$

$$\pi \vDash \phi \mathbin{\mathcal{U}} \Psi \iff \exists j \geqslant 0.(\pi[j] \vDash \Psi \wedge (\forall 0 \leqslant k < j.\pi[k] \vDash \phi)$$

2 Model Checking di CTL

Data una formula CTL ϕ ed un TS l'obiettivo dell'algoritmo di Model Checking è quello di dire se il TS soddisfa o meno la formula. Gli algoritmi proposti lavorano su formule in Existential Normal Form (ENF), definite dalla Grammatica 3

$$\phi ::= \text{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists \bigcirc \phi \mid \exists \Box \phi \mid \exists(\phi_1 \mathbf{U} \phi_2)$$

Grammatica 3: Grammatica delle formule in ENF

il che non è limitate in quanto il Teorema 2.1 dimostra che per ogni formula CTL esiste la corrispondente formula in ENF.

Teorema 2.1. *Per ogni formula CTL esiste un equivalente formula CTL in ENF*

Dimostrazione. Grazie alle leggi di dualità si ottengono delle regole di traduzione:

$$\forall \bigcirc \phi \equiv \neg\exists \bigcirc \neg\phi$$

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

□

Algoritmo 1 Algoritmo di model checking base per CTL

```
1: procedure CTLModelChecking(TS, φ)
2:     for all i ≤ |φ| do
3:         for all Ψ ∈ Sub(φ) con |Ψ| = i do
4:             calcola Sat(Ψ) da Sat(Ψ')
5:         end for
6:     end for
7:     return I ⊆ Sat(φ)
8: end procedure
```

L'algoritmo base, mostrato in Algoritmo 1, risolve ricorsivamente il problema di verificare se un determinato TS soddisfa una formula ϕ. Fondamentalmente il calcolo consiste in un attraversamento dalle foglie alla radice dell'albero di parsing della formula sugli stati ϕ. In questo albero i nodi rappresentano le sottoformule Ψ di ϕ, mentre le foglie rappresentano le proposizioni atomiche $a \in AP$ e la costante $true$. Durante la computazione vengono calcolati ricorsivamente gli insiemi $Sat\Psi$ per ogni sottoformula Ψ di ϕ. Ad ogni passo, per stabilire quali sono gli stati che soddisfano un nodo v, si combinano le valutazioni (già effettuate) dei suoi nodi figli. Il tipo di computazione quando si raggiunge il nodo v dipende dal tipo di operatore che contiene, che può essere $\wedge$, $\exists \bigcirc$ oppure $\exists \mathbf{U}$.

Il seguente teorema definisce come vengono generati gli insiemi di sottoformule.

Teorema 2.2. *Sia* $TS = (S, Act, \rightarrow, I, AP, L)$ *un TS senza stati terminali. Per tutte le formule CTL* ϕ, Ψ *su AP è vero che:*

$$Sat(true) = S \tag{2.2.1}$$

$$Sat(a) = \{s \in S | a \in L(s)\} \tag{2.2.2}$$

$$Sat(\phi \wedge \Psi) = Sat(\phi) \cap Sat(\Psi) \tag{2.2.3}$$

$$Sat(\neg\phi) = S \backslash Sat(\phi) \tag{2.2.4}$$

$$Sat(\exists \bigcirc \phi) = \{s \in S | Post(s) \cap Sat(\phi) \neq \varnothing\} \tag{2.2.5}$$

$$\begin{aligned} &Sat(\exists(\phi \mathbf{U} \Psi)) \textit{ è il più piccolo sottoinsieme T di S tale per cui} \\ &(Sat(\Psi) \subseteq T \wedge (s \in Sat(\phi) \wedge Post(s) \cap T \neq \varnothing)) \implies s \in T \end{aligned} \tag{2.2.6}$$

$$\begin{aligned} &Sat(\exists(\Box\phi))\textit{è il più grande sottoinsieme T di S tale che} \\ &T \subseteq Sat(\phi) \wedge s \in T \implies Post(s) \cap T \neq \varnothing \end{aligned} \tag{2.2.7}$$

Le caratterizzazioni fornite dal Teorema 2.2 forniscono una base per la costruzione di algoritmi per calcolare gli insiemi di soddisfacibilità per i vari operatori.

Per l'operatore Until $\mathbf{U}$ la (2.2.6) del Teorema 2.2 suggerisce di usare una procedura iterativa tale per cui $T_0 = Sat(\Psi)$ e $T_{i+1} = T_i \cup \{s \in Sat\phi | Post(s) \cap T_i \neq \varnothing\}$. L'insieme T_i contiene tutti gli stati che possono raggiungere uno stato $s \in Sat\Psi$ in i passi attraverso path che passano per stati $s^1 \in Sat\psi$.

Algoritmo 2 Algoritmo per $Sat(\exists(\phi \mathbf{U} \Psi))$

```
1:  procedure ComputeExistsUntil(TS, φUΨ)
2:      E := Sat(Ψ)
3:      T :=E
4:      while E ≠ ∅ do
5:          let s¹ ∈ E
6:          E := E\{s¹}
7:          for all  s ∈ Pre(s¹) do
8:              if s ∈ Sat(φ)\T then
9:                  E := E ∪ {s}
10:                 T := T ∪ {s}
11:             end if
12:         end for
13:     end while
14:     return T
15: end procedure
```

L'Algoritmo 2 parte calcolando tutti gli stati che soddisfano Ψ, che vengono poi copiati in due insiemi: E e T. Successivamente parte un ciclo che effettua una ricerca andando ad analizzare i predecessori degli stati. In questo ciclo viene preso un elemento s' dall'insieme E e ne vengono analizzati i predecessori. Se uno stato s appartiene a $Pre(s')\backslash T$ allora viene inserito in E ed in T. Alla fine l'insieme T conterrà tutti gli stati che soddisfano la formula $\exists(\phi \mathbf{U} \Psi)$.

L'algoritmo per calcolare $Sat(\exists \Box \phi)$ sfrutta la caratterizzazione fornita da 2.2.7. L'idea base è computare $Sat(\exists \Box \phi)$ iterativamente in questo modo:

$$T_0 = Sat(\phi) \; e \; T_{i+1} = T_i \cap \{s \in Sat(\phi) \mid Post(s) \cap T_i \neq \varnothing\}$$

L'algoritmo 3 realizza questa procedura con una ricerca volta all'indietro che

Algoritmo 3 Algoritmo per $Sat(\exists \Box \phi)$

```
1:  procedure ComputeExistsAlways(TS, (∃ □ φ))
2:      E := S\Sat(φ)
3:      T := Sat(φ)
4:      for all s ∈ Sat(φ) do
5:          count[s] := |Post(s)|
6:      end for
7:      while E ≠ ∅ do
8:          let s¹ ∈ E
9:          E := E\{s¹}
10:         for all  s ∈ Pre(s¹) do
11:             if s ∈ T then
12:                 count[s] := count[s] − 1
13:                 if count[s] = 0 then
14:                     T := T\{s}
15:                     E := E ∪ {s}
16:                 end if
17:             end if
18:         end for
19:     end while
20:     return T
21: end procedure
```

inizia con

$$T = Sat(\phi) \qquad e \qquad E = S\backslash Sat(\phi)$$

in questo caso T è uguale a T_0 ed E contiene tutti gli stati per cui $\exists \Box \phi$ è falsa. Durante la ricerca gli stati $s \in T$ che non soddisfano $\exists \Box \phi$ vengono rimossi se $Post(s) \cap T = \varnothing$. Questa verifica è resa possibile da un array chiamato $count[s]$ definito $\forall s \in Sat(\phi)$ che conta quanti successori ha lo stato s. Questo contatore verrà decrementato ogni volta che uno stato contenuto nei predecessori di uno stato $s' \in E$ è anche in T. Quando il contatore è 0 lo stato verrà marcato come stato che non soddisfa $\forall s \in Sat(\phi)$, quindi sarà rimosso da T ed aggiunto ad E.

2.1 Complessità dell'algoritmo

Teorema 2.3. *Per un Transition System (TS) con N stati e K transizioni, ed una formula CTL* ϕ*, il problema di* $TS \models \phi$ *può essere risolto in tempo*

$$O((N + K) \cdot |\phi|)$$

Dimostrazione. La complessità in tempo di questo algoritmo è determinata come segue. Sia TS un Transition System (TS) finito con N stati e K transizioni. Sotto l'assunzione che l'insieme di predecessori di uno stato sono

rappresentati da una *Linked List*, la complessità degli Algoritmi 3 e 2 sono $O(N + K)$. Visto che la computazione del $Sat(\phi)$ viene effettuata in una maniera *bottom-up* la complssità risulta lineare nella dimensione della formula, quindi la complessità dell'algoritmo 1 è data da

$$O((N + K) \cdot |\phi|)$$

□

Va però ricordato che l'algoritmo proposto lavora con formule in ENF, il che può portare ad una crescita esponenziale della dimensione della formula. Fortunatamente però esistono algoritmi per calcolare gli insiemi di stati soddisfatti per formule non in ENF, che hanno complessità $O(N + K)$.

3 Model Checker in Python per CTL

Il programma è stato realizzato utilizzando Python con l'ausilio delle librerie `pyparser` e `networkx`. La prima libreria è stata usata per definire la grammatica di CTL ed effettuare il parsing di una formula passata in input come una stringa. La seconda libreria è stata invece utilizzata per la rappresentazione dei TS. Questa libreria contiene molte funzioni che si sono rivelate essenziali durante lo sviluppo del model checker, come per esempio la possibilità di ottenere liste di predecessori e successori di uno stato con una sola riga di codice.

3.1 Come vengono rappresentati i TS

Un Transition System (TS) viene rappresentato come un grafo G dove i nodi sono gli stati del TS e gli archi sono rappresentati dagli archi del TS. Più formalmente:

Definizione 2. *Dato un Transition System (TS)* $= \{S, Act, \rightarrow, I, AP, L\}$ *si considera il grafo sottostante* $G = (V, E)$ *tale che* $V = S$ *e* $E = \{(s, s') \in SxS | \ s' \in Post(s)\}$

Grazie alla libreria `networkx` è possibile leggere un grafo da un file eXtensible Markup Language (XML) in linguaggio Graph Exchange XML Format (GEXF). GEXF è un linguaggio usato per la definizione di strutture complesse, come grafi, in maniera semplice. Un esempio di grafo in GEXF è mostrato in Codice 1. Questo grafo è di tipo diretto ed è formato da due nodi e un arco che li collega, il primo nodo ha label `Hello` ed il secondo ha label `World`.

Codice 1: Esempio di grafo rappresentato in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft" version="1.2">
    <graph mode="static" defaultedgetype="directed">
        <nodes>
            <node id="0" label="Hello" />
            <node id="1" label="World" />
        </nodes>
        <edges>
            <edge id="0" source="0" target="1" />
```

```
10          </edges>
11      </graph>
12  </gexf>
```

Per leggere il grafo in Python sono sufficienti poche righe di codice, mostrate in Codice 2.

Codice 2: Lettura di un file GEXF

```
1  import networkx as nx
2  if __name__ == "__main__":
3      path = '../inputfiles/ts6_11es.gexf' #Path grafo
4      graph = nx.read_gexf(path)
```

Un TS è rappresentato da un nodo principale `graph` con attributi `mode` impostato a `static` e `defaultedgetype` impostato a `directed`. A sua volta questo nodo conterrà due nodi figli chiamati `nodes` e `edges`, che conterranno a loro volta una lista di nodi e di archi. Uno stato di un TS viene rappresentato assegnando, nel file GEXF un figlio `node` al padre `nodes`, con attributo `id` scelto arbitrariamente e attributo `label` contenente le proposizioni atomiche che sono soddisfatte dallo stato considerato. Per rappresentare uno stato iniziale del TS è sufficiente anteporre al `id` dello stato una `S`. Un arco è semplicemente rappresentato da un figlio `edge` del nodo padre `edges`. Gli attributi di un nodo `edges` sono `id`, `target` e `source`. L'identificativo può essere scelto arbitrariamente, mentre `target` e `source` sono rispettivamnete la destinazione e la sorgente dell'arco.

3.2 Come vengono rappresentate le formule

Sono state definite due grammatiche, una per le formule CTL una per quelle in ENF riportata in Grammatica 5. Una formula del tipo $\exists \Box \forall (a \mathcal{U} b)$ può essere scritta utilizzando la grammatica riportata in Grammatica 4 in questo modo:

$$[](a \text{ FU } b)$$

Si distinguono gli operatori unari, come la negazione, e gli operatori binari, come & o UNTIL. Più specificatamente si indica gli operatori unari:

- $\neg\varphi$ con !φ
- $\exists \Box \varphi$ con [] φ
- $\exists \bigcirc \varphi$ con NEXT φ
- $\exists \Diamond \varphi$ con EE φ
- $\forall \Diamond \varphi$ con FE φ
- $\forall \Box \varphi$ con FA φ
- $\forall \bigcirc \varphi$ con FN φ

Per gli operatori binari:

- $\exists(\varphi_1 \mathcal{U} \varphi_2)$ con φ_1 UNTIL φ_2

- φ_1 & φ_2 con φ_1 & φ_2
- $\forall(\varphi_1 \mathbf{U} \varphi_2)$ con φ_1 FU φ_2

Le variabili atomiche si possono scrivere utilizzando concatenazioni di lettere minuscole dell'alfabeto o numeri interi. Si può scrivere ad esempio (a & b), ma anche (a1 & b32) oppure (zty & x6).

In molti casi si possono omettere le parentesi, ad esempio si può scrivere a UNTIL b UNTIL c che sarà valutata come (a UNTIL b) UNTIL c.

Sono invece scartate formule con due o più operatori unari consecutivi in cui non si specifica le parentesi, ad esempio []!a non sarà valutata mentre la formula [](!a) sì.

$$\begin{aligned} \text{atom} &::= \text{true} \mid \text{a} \\ \varphi &::= \text{atom} \mid \neg\varphi \mid []\varphi \mid \text{NEXT } \varphi \mid \text{EE } \varphi \mid \text{FE } \varphi \mid \text{FA } \varphi \mid \text{FN } \varphi \mid \phi \\ \phi &::= \varphi_1 \,\&\, \varphi_2 \mid \varphi_1 \text{ UNTIL } \varphi_2 \mid \varphi_1 \text{ FU } \varphi_2 \end{aligned}$$

Grammatica 4: Grammatica per le formule CTL

$$\begin{aligned} \text{atom} &::= \text{true} \mid \text{a} \\ \varphi &::= \text{atom} \mid \neg\varphi \mid []\varphi \mid \text{NEXT } \varphi \mid \phi \\ \phi &::= \varphi_1 \,\&\, \varphi_2 \mid \varphi_1 \text{ UNTIL } \varphi_2 \end{aligned}$$

Grammatica 5: Grammatica per le formule ENF

3.2.1 Il parsing della formula

Per analizzare la formula si utilizza la libreria `pyparsing`. Si possono definire facilmente delle grammatiche e ricavare da una formula in notazione infissa uno stack in notazione postfissa, contenente tutti gli elementi della formula.

Nel Codice 3 dalla linea 31 alla 42 si definiscono gli operatori unari e le variabili atomiche. Gli operatori unari possono essere:

- un operatore e la variabile atomica semplice, es. !a
- un operatore e la variabile atomica con le parentesi, es. !(a)
- un operatore e un'espressione tra parentesi, es. !(a & b)

Le variabili atomiche sono "TRUE" oppure tutte le possibili concatenazioni di lettere minuscole o numeri interi quindi: ((a..z)*(1..9)*)*.

Alla linea 48 nella variabile `factor` si salvano le possibili operazioni binarie utilizzando la funzione di `pyparsing` `ZeroOrMore`. Infatti nella formula

possono esserci zero operazioni di & o UNTIL e quindi factor sarà formato solo da `atoms` cioè le combinazioni di espressioni con operazioni unarie o variabili atomiche. Se presenti, le operazioni binarie sono del tipo:

- un operatore e le due espressioni, es. a & !b
- una serie di operatori e le espressioni, es. a & b & (c UNTIL d) & e

Ad ogni elemento è associata un'azione che inserisce il valore nello stack, es & ha `pushBinary`. La funzione `parseString` (linea 54) identifica gli elementi appartenenti alla formula e invoca le azioni al momento dell'idividuazione dell'espressione.

Codice 3: Parser CTL

```
31      def CTL(self):
32          bnf = None
33          if not bnf:
34              atomicVal = Word(srange("[a-zo-9]"), max=10) | 'TRUE'
35              lpar = Literal('(').suppress()
36              rpar = Literal(')').suppress()
37              expr = Forward()
38              atomo = (Optional('[]') + (atomicVal | lpar + atomicVal +
                     rpar).setParseAction(self.pushAtom) | Optional('[]')
                     + (lpar + expr + rpar)).setParseAction(self.
                    pushUAlways)
39              atom1 = (Optional('!') + (atomicVal | lpar + atomicVal +
                    rpar).setParseAction(self.pushAtom) | Optional('!') +
                     (lpar + expr + rpar)).setParseAction(self.pushUNot)
40              atom2 = (Optional('NEXT') + (atomicVal | lpar + atomicVal
                     + rpar).setParseAction(self.pushAtom) | Optional('
                    NEXT') + (lpar + expr + rpar)).setParseAction(self.
                    pushNext)
41
42              atoms = atomo | atom1 | atom2
43
44              andOp = Literal('&')
45              untilOp = Literal('UNTIL')
46
47              factor = Forward()
48              factor = (atoms) + ZeroOrMore((andOp + (atoms) | untilOp
                     + (atoms)).setParseAction(self.pushBinary))
49              expr << factor
50              bnf = expr
51          return bnf
52
53      def getParsedFormula(self, formula):
54          results = self.CTL().parseString(formula)
55          return self.exprStack
```

3.2.2 La conversione in ENF

Per convertire la formula in ENF si crea con stessa metodologia usata nel parsing uno stack contenente gli elementi della formula in notazione postfissa, e in aggiunta si valuta lo stack per restituire la stringa convertita. Una volta creato lo stack si invoca la funzione `evaluateStack` che ricorsivamente costruisce la stringa di output. In Codice 4 si mostrano i casi base in cui non si fanno modifiche alla formula.

Codice 4: ENF Converter

```
77      def evaluateStack(self):
78          op = self.exprStack.pop()
79          if op == '[]':
80              op1 = '[](' + self.evaluateStack() + ')'
81              return op1
82          if op == '!':
83              op1 = '!(' + self.evaluateStack() + ')'
84              return op1
```

In Codice 5 si mostra invece la conversione nel caso di FU(Forall until), dove viene utilizzata la seguente equivalenza:

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

per convertire la seguente formula:

$$\phi \text{ FU } \Psi$$

nella controparte in ENF: !(Ψ UNTIL (!ϕ & Ψ)) & !([]!(Ψ))

Codice 5: ENF Converter

```
102         if op == 'FU':
103             op1 = self.evaluateStack()  # phi
104             op2 = self.evaluateStack()  # psi
105             op3 = '!(!(' + op2 + ')UNTIL(!(' + op1 + ')&!(' + op2 + '
                    ))) & !([](!(' + op2 + ')))'
106             return op3
```

3.3 Gli algoritmi di model checking

Il model checker vero e proprio è definito come una classe in python, che per essere istanziata necessita di un TS e di una formula CTL in ENF. Quindi in fase di costruzione si occuperà di leggere il file GEXF e convertire la formula in notazione postfissa, memorizzandola in uno stack, come descritto in Sottosezione 3.2.1.

Una volta creata l'istanza della classe `CTLModelChecker` è necessario chiamare la funzione mostrata in Codice 6, che rappresenta una possibile implementazione dell'Algoritmo 1.

Codice 6: Funzione che effettua il model checking base

```
1       def iterativeCheckFormula(self):
2           satisfactionSet = []
3           for i in self.__formula:
4               if i == '!':
5                   elo = satisfactionSet.pop()
6                   satisfactionSet.append(self.__checkNot(elo))
7               elif i == '&':
8                   elo = satisfactionSet.pop()
9                   el1 = satisfactionSet.pop()
10                  satisfactionSet.append(self.__checkAnd(elo, el1))
11              elif i == 'UNTIL':
12                  elo = satisfactionSet.pop()
13                  el1 = satisfactionSet.pop()
14                  satisfactionSet.append(self.__checkUntil(elo, el1))
15              elif i == 'NEXT':
16                  elo = satisfactionSet.pop()
```

```
17                  satisfactionSet.append(self.__checkNext(elo))
18              elif i == '[]':
19                  elo = satisfactionSet.pop()
20                  satisfactionSet.append(self.__checkAlways(elo))
21              elif i == 'TRUE':
22                  satisfactionSet.append(self.__checkTrue())
23              elif i[0].isalpha():
24                  satisfactionSet.append(self.__checkSingle(i))
25
26          return (self.__checkInitialStates(satisfactionSet),
                satisfactionSet[0])
```

Questa funzione scorre iterativamente la formula effettuando diverse operazioni in base all'elemento analizzato, effettuandone così la valutazione. Il caso base è quando viene analizzato un singoletto, ovvero una singola proposizione senza operatori logici. In questo caso viene richiamato il metodo `__checkSingle()`, mostrato in Codice 7, il quale ritornerà un insieme di stati che soddisfano il singoletto. Questo insieme sarà aggiunto ad una pila temporanea, che alla fine conterrà come singolo elemento solamente l'insieme di stati che soddisfano la formula data in input.

Codice 7: Funzione per il calcolo dell'insieme di soddisfacibilità di una singola proposizione

```
1       def __checkSingle(self, i):
2           tempList = []
3           for node in self.__nodes:
4               if i in self.__nodes[node]:
5                   tempList.append(node)
6           return tempList
```

Il compito di questo metodo è molto semplice, itera la lista dei nodi e verifica se la proposizione atomica in questione è presente all'interno dell'attributo `label` del nodo preso in considerazione.

Se durante la valutazione della formula si incorre in un operatore bisogna estrarre dalla pila uno, o due insiemi di stati soddisfatti ed effettuare l'operazione. L'operatore più semplice da gestire è il NOT, in quanto, come mostrato in Codice 8 è sufficiente sottrarre a tutti gli stati del TS gli stati passati in input alla funzione `__checkNot()`.

Codice 8: Funzione per il calcolo del NOT

```
1       def __checkNot(self, elo):
2           if elo is not None:
3               return list(set(self.__nodes) - set(elo))
4           return []
```

Il calcolo dell'AND richiede due insiemi di stati, che saranno estratti dalla pila e passati in input alla funzione mostrata in Codice 9. Questa funzione trasforma in oggetti di tipo `set` le due liste passate in input, ne effettua l'intersezione e le ritorna sottoforma di lista.

Codice 9: Funzione per il calcolo del AND

```
1       def __checkAnd(self, elo, el1):
2           return list(set(elo).intersection(el1))
```

L'operatore $\bigcirc$ richiede solamente una lista di stati e iterando sulla lista dei nodi effettua l'operazione di intersezione tra i successori del nodo preso in considerazione e la lista di stati passata in input. Se la dimensione dell'insieme risultante è maggiore di 0 allora il nodo verrà aggiunto alla lista di stati che soddisfa la formula per cui è stato chiamato il metodo. L'implementazione di questa funzione è mostrata in Codice 10.

Codice 10: Funzione per il calcolo del NEXT

```
1     def __checkNext(self, elo):
2         tempList = []
3         for node in self.__nodes:
4             successors = self.__ts.successors(node)
5             if len(set(successors).intersection(set(elo))) > 0:
6                 tempList.append(node)
7         return tempList
```

Nel caso in cui si incorre nell'operatore $\Box$ viene richiamato il metodo `__checkAlways()`, che prende in input una singola lista di stati. Questo metodo è una implementazione dell'Algoritmo 3 ed è mostrato in Codice 11

Codice 11: Funzione per il calcolo del ALWAYS

```
1     def __checkAlways(self, elo):
2         E = list(set(self.__nodes) - set(elo))
3         T = elo
4         count = dict()
5         for el in elo:
6             count[el] = len(self.__ts.successors(el))
7         while len(E) > 0:
8             s1 = E.pop()
9             s1Preset = self.__ts.predecessors(s1)
10            for s1pre in s1Preset:
11                if s1pre in T:
12                    count[s1pre] = count[s1pre] - 1
13                    if count[s1pre] == 0:
14                        E.append(s1pre)
15                        T.remove(s1pre)
16        return T
```

Per l'operatore $\mathbf{U}$ è stato implementato l'Algoritmo 2, mostrato in Codice 12.

Codice 12: Funzione per il calcolo del UNTIL

```
1     def __checkUntil(self, elo, el1):
2         E = elo
3         T = E[:]
4         while len(E) > 0:
5             s1 = E.pop()
6             s1Preset = self.__ts.predecessors(s1)
7             for s in s1Preset:
8                 if s in list(set(el1) - set(T)):
9                     E.append(s)
10                    T.append(s)
11        return T
```

L'ultima operazione effettuata dall'Algoritmo 1 è verificare se gli stati iniziali sono un sottoinsieme degli stati soddisfatti dalla formula. Il Codice 6 effettua questa operazione alla riga 26.

3.4 Una versione migliorata del Model Checker

Con lo scopo di migliorare le prestazioni dell'algoritmo di Model Checking è stata sviluppata un'ulteriore versione che fa uso di una tecnica di caching. Usando questa tecnica ad ogni passo viene memorizzato in un dizionario il risultato delle sottoformule valutate. In questo modo, se la formula contiene più sottoformule uguali non è necessario rivalutarle ogni volta, ma basta accedere al dizionario per prenderne il risultato. Per realizzare questa ottimizzazione è stato inserito un dizionario all'interno della classe, ed il metodo `iterativeCheckFormula` mostrato in Codice 6 è stato modificato. Il nuovo metodo `iterativeCheckFormula` per ogni elemento dello stack verifica prima se è già stato valutato effettuando un accesso nel dizionario. Nel caso in cui sia già stato valutato non effettua una chiamata al metodo che si occupa della valutazione, ma aggiunge alla pila degli stati soddisfatti l'elemento corrispondente del dizionario. L'implementazione di questa funzione è mostrata in Codice 13.

Codice 13: Funzione per il model checking base ottimizzata con la tecnica del caching

```
1     def iterativeCheckFormula(self):
2         satisfactionSet = []
3         for i in self.__formula:
4             if i == '!':
5                 elo = satisfactionSet.pop()
6                 satisfactionSet.append(self.__checkNot(elo))
7             elif i == '&':
8                 elo = satisfactionSet.pop()
9                 el1 = satisfactionSet.pop()
10                if str(elo)+"&"+str(el1) in self.__cache:
11                    satisfactionSet.append(self.__cache[str(elo)+"&"+
                          str(el1)])
12                else:
13                    self.__cache[str(elo)+"&"+str(el1)] = self.
                          __checkAnd(elo, el1)
14                    satisfactionSet.append(self.__cache[str(elo)+"&"+
                          str(el1)])
15            elif i == 'UNTIL':
16                elo = satisfactionSet.pop()
17                el1 = satisfactionSet.pop()
18                if str(elo)+"UNTIL"+str(el1) in self.__cache:
19                    satisfactionSet.append(self.__cache[str(elo)+"
                          UNTIL"+str(el1)])
20                else:
21                    self.__cache[str(elo)+"UNTIL"+str(el1)] = self.
                          __checkUntil(elo, el1)
22                    satisfactionSet.append(self.__cache[str(elo)+"
                          UNTIL"+str(el1)])
23            elif i == 'NEXT':
24                elo = satisfactionSet.pop()
25                satisfactionSet.append(self.__checkNext(elo))
26                if "NEXT"+str(elo) in self.__cache:
27                    satisfactionSet.append(self.__cache["NEXT"+str(
                          elo)])
28                else:
29                    self.__cache["NEXT"+str(elo)] = self.__checkNext(
                          elo)
30                    satisfactionSet.append(self.__cache["NEXT"+str(
                          elo)])
```

```
31              elif i == '[]':
32                  elo = satisfactionSet.pop()
33                  if "[]"+str(elo) in self.__cache:
34                      satisfactionSet.append(self.__cache["[]"+str(elo)
                            ])
35                  else:
36                      self.__cache["[]"+str(elo)] = self.__checkAlways(
                            elo)
37                      satisfactionSet.append(self.__cache["[]"+str(elo)
                            ])
38              elif i == 'TRUE':
39                  satisfactionSet.append(self.__checkTrue())
40              elif i[0].isalpha():
41                  satisfactionSet.append(self.__checkSingle(i))
42
43          return (self.__checkInitialStates(satisfactionSet),
                satisfactionSet[0])
```

4 Esempi di utilizzo del Model Checker di CTL in Python

Per provare il model checker sono state verificate le formule di esempio analizzate nel libro di testo, inoltre è stato sviluppato un piccolo script Python che costruisce il TS che modella il problema dei filosofi a cena.

4.1 Formula $\exists\Diamond\phi$

Sul TS riportato in Figura 1 è stato verificata la formula $\exists\Diamond\phi$ in cui

$$\phi = ((a = c) \text{ or } (a != b))$$

cioè ϕ =!(!(a & c & !b) & !(!a & !c & b)). La formula nella sintassi descritta in Grammatica 4 equivale a

$$EE(!(!(a \,\&\, c \,\&\, !b) \,\&\, !(!a \,\&\, !c \,\&\, b)))$$

e sarà tradotta in ENF in

$$TRUE\ UNTIL(!(!(a \,\&\, c \,\&\, !(b)) \,\&\, !(!(a) \,\&\, !(c) \,\&\, b)))$$

per poi essere passata al modelChecker. Questo restituisce il risultato della valutazione e l'insieme di stati soddisfatti cioè:

$$(\text{False}, ['5','4','6','7'])$$

Si può anche utilizzare l'insieme ricavato per mostrare visivamente gli stati soddisfatti come mostrato in Figura 2.

4.2 Formula $\exists\,\Box\,b$

La formula in questo caso è già in ENF quindi il convertitore non la modifica. $\exists\,\Box\,b$ equivale a

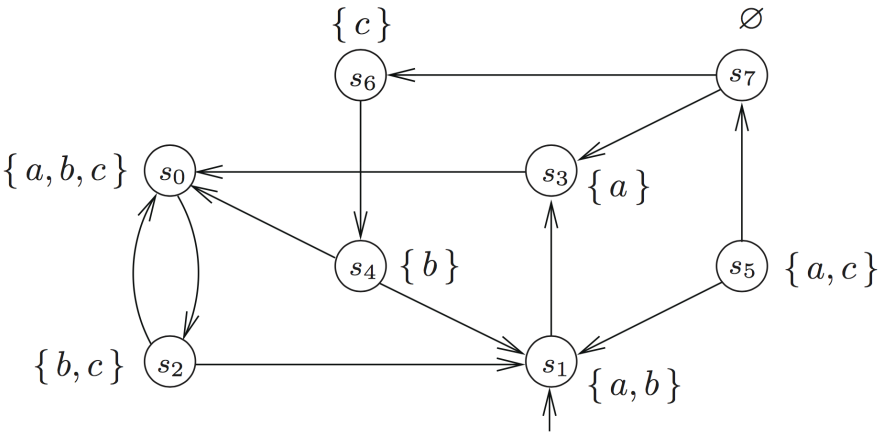

Figura 1: Transition system

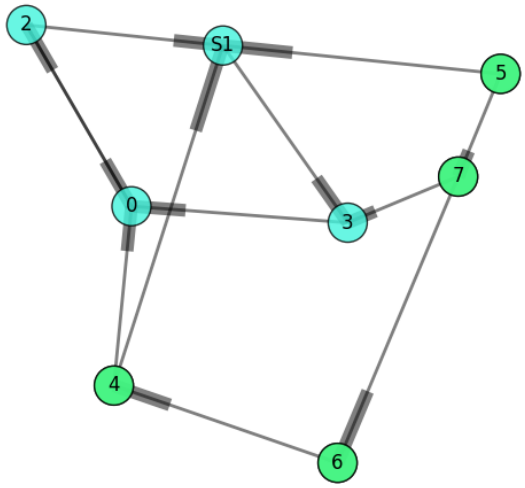

Figura 2: Transition system risultante

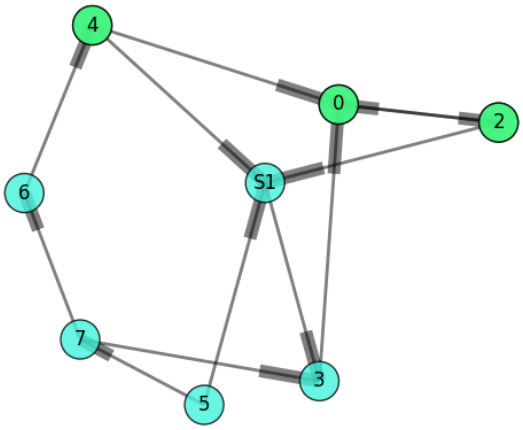

Figura 3: Transition system risultante

$$[]b$$

e il modelChecker restituisce

$$(\text{False}, ['0','2','4'])$$

. Il risultato in questo esempio si mostra in Figura 3

4.3 Fisolofi a cena

Lo script creato per modellare il problema dei fisolofi a cena, dato in input un intero n restituisce il GEXF che rappresenta il TS con n filosofi. Le proposizioni atomiche del TS generato sono:

$$AP = \{e_0, \ldots, e_n, t_0, \ldots, t_n, h_0, \ldots, h_n, w_0, \ldots, w_n\}$$

che rappresentano gli stati possibili in cui si può trovare ognuno degli n filosofi:

- $t_0 \ldots t_n$ rappresentano le proposizioni atomiche dove il filosofo è nello stato "thinking", cioè quando non è in attesa di niente
- $h_0 \ldots h_n$ rappresentano le proposizioni atomiche dove il filosofo è nello stato "hungry", ovvero quando richiede la prima forchetta
- $w_0 \ldots w_n$ rappresentano le proposizioni atomiche dove il filosofo è nello stato "wait", ovvero quando è in attesa della seconda forchetta

- $e_0 \dots e_n$ rappresenta le prooposizioni atomiche dove il filosofo è nello stato "eating", ovvero quando mangia

Lo stato iniziale del TS generato sarà lo stato che conterrà come label l'insieme $\{t_0 \dots t_n\}$, cioè quando tutti i filosofi sono nello stato di "thinking".

Prendendo ad esempio il transition system per tre filosofi, si può verificare la starvation freedom con questa formula:

$$\forall \Box \forall \Diamond e1 \& \forall \Box \forall \Diamond e2 \& \forall \Box \forall \Diamond e3$$

cioè

FA (FE e1) & FA (FE e2) & FA (FE e3)

. Il modelChecker restituisce `False` in quanto il deadlock può verificarsi nello stato `w1w2w3` in cui ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra.

Si può inoltre utilizzare il ts dei filosofi per esaminare l'uso del caching su una formula di grandi dimensioni. Prendendo ad esempio la formula:

t1 FU h1 FU w1 FU e1

. Questa dopo la conversione in ENF diventa:

$$!(!(!(!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))\&!([](!(t1))))\text{UNTIL}$$

$$(!(w1)\&!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))\&!([](!(t1))))))$$

$$\&!([](!(!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))\&!([](!(t1)))))))$$

$$\text{UNTIL}(!(e1)\&!(!(!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))$$

$$\&!([](!(t1))))\text{UNTIL}(!(w1)\&!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))$$

$$\&!([](!(t1))))))\&!([](!(!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))$$

$$\&!([](!(t1)))))))))\&!([](!(!(!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))$$

$$\&!([](!(t1))))\text{UNTIL}(!(w1)\&!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))$$

$$\&!([](!(t1))))))\&!([](!(!(!(t1)\text{UNTIL}(!(h1)\&!(t1)))\&!([](!(t1)))))))))$$

Lista di acronimi

CTL Computational Tree Logic

LTL Linear Temporal Logic

TS Transition System

ENF Existential Normal Form

XML eXtensible Markup Language

GEXF Graph Exchange XML Format