Algoritmi e Strutture Dati
-
Elaborato

Matteo Salvalai (715827)
Jacopo Mora (715149)

A.A. 2019/2020

Indice

Introduzione

Nel seguente documento viene descritto il lavoro svolto da Salvalai Matteo e Mora Jacopo nell'ambito dell'elaborato richiesto dal corso *Algoritmi e Strutture Dati*.
L'intero elaborato è stato sviluppato nel linguaggio di programmazione `Java`, scelto in quanto estremamente familiare al gruppo di lavoro.
Considerando le richieste ad un alto livello di astrazione, queste prevedono la creazione di software in grado di costruire reti di automi a stati finiti comportamentali e di poter effettuare, su di esse, la diagnosi relativa ad osservazioni lineari.
Il software sviluppato permette agli utenti di interagire tramite interfaccia testuale o grafica e di svolgere tutte le attività precedentemente descritte, avendo cura di mostrare le prestazioni temporali e spaziali degli algoritmi sviluppati.
Nel primo capitolo vengono descritti gli automi a stati finiti, le reti di automi a stati finiti e ne viene discussa l'implementazione.
L'attenzione, nel secondo capitolo, si concentra sugli spazi comportamentali relativi ad una rete di automi a stati finiti e sull'algoritmo definito per generarli.
Nel terzo capitolo vengono invece considerati gli spazi comportamentali relativi ad un'osservazione lineare e l'algoritmo usato per costruirli.
Nel quarto capitolo viene descritto il procedimento tramite il quale ottenere la diagnosi relativa ad un'osservazione lineare per mezzo dei relativi spazi comportamentali.
Nel quinto capitolo il tema principale è il diagnosticatore. Ne verrà discussa l'implementazione e il relativo algoritmo di generazione.
Nel sesto capitolo viene descritto come utilizzare il diagnosticatore per generare la diagnosi relativa ad un'osservazione lineare.
Nel settimo capitolo vengono discusse le effettive prestazioni spaziali e temporali degli algoritmi implementati nell'ambito di una rete di benchmark e di una rete generata dal gruppo di lavoro.
Nell'ultimo capitolo viene descritta l'interfaccia testuale utilizzata e i comandi previsti.

Capitolo 1

Automi a Stati Finiti

In questo capitolo vengono studiati gli automi a stati finiti, gli automi a stati finiti comportamentali e le reti di automi a stati finiti.

1.1 Automi a stati finiti

Un *Automa a Stati Finiti* è composto da un insieme finito, non nullo, di stati e da un insieme finito di transizioni. Nell'insieme degli stati deve essere presente uno stato iniziale ed è possibile individuare un sottoinsieme di stati di accettazione. Ad ogni transizione viene associato un simbolo, appartenente ad un dato alfabeto finito; sono ammesse auto-transizioni.
Considerando ogni stato di un *Automa a Stati Finiti* come un nodo e ogni sua transizione come un arco, si può osservare come l'oggetto così formato ricordi fortemente un grafo orientato, dotato di autotransizioni. La nostra implementazione rispecchia questa visione di *Automa a Stati Finiti*. Un grafo può essere implementato tramite una *lista di adiacenza*, che fa corrispondere ad ogni vertice v una lista concatenata di vertici adiacenti a v stesso (in ordine arbitrario), oppure tramite una *matrice di adiacenza*. Nel nostro caso, una matrice di adiacenza non sarebbe appropriata, in quanto avremo successivamente a che fare anche con transizioni parallele, le quali non possono essere rappresentate tramite tale struttura. Abbiamo di conseguenza orientato la nostra scelta sulla lista di adiacenza, ottenendo sì un costo temporale maggiore per l'accesso a un vertice rispetto a una matrice di adiacenza, ma in ogni caso un tempo lineare. Come lista di adiacenza, abbiamo usato una mappatura tra uno Stato, la chiave, e un apposito oggetto rappresentante le interconnessioni dello stato stesso, ovvero le transizioni in entrata e le transizioni in uscita. L'unico svantaggio che deriva da tale scelta implementativa è che viene occupato il doppio dello spazio per ogni transizione, questo perchè ciascuna transizione verrà salvata due volte: una volta come transizione di ingresso per il proprio stato destinazione e una volta come transizione di uscita per il proprio stato sorgente. La nostra implementazione delle *interconnessioni* si trova nella classe `Interconnections<S, T>`,

classe parametrica dove `S` è una implementazione di `StateInterface`, mentre `T` è una sottoclasse di `Transition<S>`. Queste due ultime classi verranno trattate in dettaglio nelle sezioni seguenti.

1.1.1 Stato

Dalla definizione iniziale di *Automa a Stati Finiti* si può notare come lo *Stato* giochi un ruolo estremamente importante. Uno *Stato* deve inizialmente essere identificabile e deve poter fornire indicazioni sulla sua eventuale accettabilità.
Uno *Stato* con queste caratteristiche viene implementato tramite la classe `State`. Essa è dotata di un campo booleano, in modo da poter contenere informazioni relative all'accettabilità, e di un campo testuale, atto ad ospitare un identificatore univoco.
Il comportamento generale di uno *Stato* viene descritto all'interno dell'interfaccia `StateInterface`, in modo tale da separare l'implementazione dall'idea di *Stato*.
L'ugualianza fra `State`s è stata implementata in modo da considerare gli identificatore introdotti, sovrascrivendo così la logica nativa di ugualianza.

1.1.2 Transizione

Essenziali per l'implementazione di un *Automa a Stati Finiti*, oltre agli stati, sono le *transizioni*. La nostra implementazione di *transizione* prende forma nella classe parametrica `Transition<S>`, dove il parametro `S` richiede che la classe utilizzata si comporti come uno *stato*, ovvero implementi l'interfaccia precedentemente descritta `StateInterface`.
Così come gli *stati*, anche le *transizioni* devono essere univocamente identificabili. La nostra implementazione definisce un campo testuale atto ad ospitare un identificatore, sulla base del quale è stato ridefinito il meccanismo di ugualianza.
Abbiamo deciso, inoltre, di fornire ad ogni transizione informazioni sugli stati da essa connessi, nella forma di due campi di tipo `S`, uno per l'origine e uno per la destinazione della *transizione*, inizializzati al momento della creazione dell'oggetto. Grazie alla presenza di quest'informazione è possibile riconoscere ed individuare con facilità eventuali autotransizioni ed eventuali transizioni parallele.

1.1.3 Automa

L'implementazione di un *Automa a Stati Finiti* avviene per mezzo della classe parametrica `Automa<S, T>`, dove il parametro `S` indica il tipo degli stati considerati, implementanti l'interfaccia `StateInterface`, e dove il parametro `T` indica il tipo di transizione, parametrizzata con il tipo di stato utilizzato, `S`. Perchè il tipo `T` possa essere valido deve estendere la classe `Transition<S>`.
Così come per `State`, abbiamo deciso anche per il caso di `Automa` di dividere l'implementazione dal comportamento generale di un *Automa a Stati Finiti*, descritto nell'interfaccia parametrica `FiniteStateMachine<S, T>`, dove `S` e `T`

devono sottostare agli stessi vincoli imposti sui parametri di `Automa`.
Ogni `Automa` permette di ospitare un identificatore, tramite il quale viene ridefinita la procedura di ugualianza. Contiene inoltre due campi di tipo `S`, uno atto a contenere lo stato definito come iniziale e uno destinato a contenere lo stato corrente.

1.2 Automi a stati finiti comportamentali

Un *Automa a Stati Finiti Comportamentale*, allo stesso modo di un *Automa a Stati Finiti*, è composto da un insieme finito, non nullo, di stati e da un insieme finito di transizioni. A differenza di quest'ultimo, un Automa Comportamentale è un Automa in cui l'insieme degli stati di accettazione è vuoto. Inoltre, ogni transizione è dotata di un evento in ingresso (che può essere nullo) e di un insieme di eventi in uscita (che può essere vuoto). In generale un *Automa a Stati Finiti Comportamentale* è un automa *non deterministico* sull'alfabeto degli eventi in ingresso, questo perchè possono esistere più transizioni distinte uscenti dal medesimo stato dotate del medesimo evento in ingresso (che potrebbe anche essere nullo).

1.2.1 Stato

La nostra implementazione di uno stato di un *Automa a Stati Finiti Comportamentale* riside nella classe `ComportamentalState`. Tale classe eredita dalla classe astratta `State` e ridefinisce il metodo *isAccepting()* restituendo sempre *false*, in accordo alla definizione di *Automa a Stati Finiti Comportamentale*.

1.2.2 Transizione

Per quanto riguarda le *transizioni*, la relativa implementazione *comportamentale* è stata definita per mezzo della classe `ComportamentalTransition`, la quale estende la classe astratta `Transition<S>` e dove `S` è sostituito dall'implementazio dello stato `ComportamentalState`. Tale classe, oltre a contenere informazioni sugli stati sorgente e destinazione, tiene conto di un eventuale evento in input, con il suo corrispondente link, e di un eventuale insieme di eventi in output, ciascuno con il corrispondente link. Per gestire l'insieme di eventi e link in output, abbiamo utilizzato una `Map`, salvando come chiave l'oggetto evento e come suo valore l'oggetto link. La classe `ComportamentalTransition` tiene anche informazioni sulla etichettatura, sia di osservabilità che di rilevanza.

1.2.3 Event

La classe `Event` è la classe preposta a gestire gli eventi. Essa è caratterizzata da un *id*, che distingue un evento da un altro. Nel caso l'*id* sia uguale a ϵ, allora l'evento è considerato *vuoto*.

1.2.4 Etichettatura

L'etichettatura consiste nel far corrispondere a ogni transizione dei componenti della rete un'etichetta di Ω (nel caso di relazione di osservabilità) e un'etichetta di F (nel caso di relazione di rilevanza). Nel nostro programma, le etichette sono state implementate come sottoclassi della classe `Label`, la quale possiede il campo *symbol* (i.e. "Ω" per l'osservabilità) e il campo *label*, contenente il valore vero e proprio dell'etichetta. Osservabilità e rilevanza sono realizzate rispettivamente mediante le sottoclassi `ObservableLabel` e `RelevantLabel`.

1.3 Reti di automi a stati finiti comportamentali

Una rete di automi a stati finiti comportamentali è costituita da almeno un *Automa a Stati Finiti Comportamentale*. Gli *Automa a Stati Finiti Comportamentale* sono disposti secondo una topologia distribuita orientata connessa, dove ogni singolo *Automa* è un componente.

1.3.1 Link

Ciascuna connessione orientata fra componenti distinti prende il nome di *link*. Possono esistere più link aventi il medesimo componente sorgente e il medesimo componente destinazione. Un link è un buffer di capacità unitaria: ogni evento non nullo in ingresso a una transizione proviene da un link, mentre gli eventi prodotti in uscita da una medesima transizione sono inviati ciascuno a un link distinto (necessariamente vuoto). Abbiamo implementato i Link nella classe omonima `Link`, la quale è caratterizza da un *id* univoco. Inoltre, contiene due ulteriori campi, uno per l'*Automa a Stati Finiti Comportamentale* sorgente e uno per quello di destinazione.

1.3.2 Implementazione della rete

La nostra implementazione di una rete di automi a stati finiti comportamentali è definita nella classe `CFSMnetwork`. L'oggetto si costruisce tramite una lista di `Link`, senza dover specificare ogni *Automa a Stati Finiti Comportamentale* contenuto nella rete, in quanto tale informazione è estratta analizzando l'Automa *sorgente* e l'Automa di *destinazione* di ciascun link.

Capitolo 2

Spazio Comportamentale

In questo capitolo vengono discussi gli Spazi Comportamentali, la loro implementazione e l'algoritmo sviluppato per generarli.

2.1 Definizione di Spazio Comportamentale

Lo *Spazio Comportamentale* è un Automa Finito *deterministico* sull'alfabeto i cui simboli sono gli identificatori di tutte le transizioni dei componenti della rete. Il linguaggio di tale Automa deterministico è l'insieme (che può contenere infiniti elementi) delle traiettorie della rete.

2.1.1 Stato Comportamentale

Lo stato di una rete di *Automi a Stati Finiti Comportamentali*, chiamato per comodità Stato Comportamentale, è costituito dallo stato di tutti i componenti e di tutti i link, dove per stato di un link si intende il contenuto (eventualmente vuoto) del link stesso. Lo Spazio Comportamentale possiede uno stato iniziale e più stati finali: lo stato iniziale della rete è quello in cui ciascun componente è nel suo stato iniziale e i link sono tutti vuoti, mentre uno stato è finale se in esso tutti i link sono vuoti. La nostra implementazione di Stato Comportamentale (`ComportamentalState`) estende dalla classe astratta `State`. Lo stato dei componenti e il contenuto dei link è gestito tramite due mappe, associando nella prima `Map` rispettivamente gli *id* degli *Automi a Stati Finiti Comportamentali* a un oggetto `ComportamentalState`, corrispondente allo stato corrente dell'Automa, e nella seconda ciascun `Link` all'`Event` presente sul link stesso.

2.1.2 Transizione Comportamentale

Una transizione di una rete di *Automi a Stati Finiti Comportamentali*, chiamata per comodità Transizione Comportamentale e definita nella classe `SpaceTransition`, è composta dalla `ComportamentalTransition` che è scattata, e lo Stato Comportamentale *sorgente* e di *destinazione*. Lo scatto di una transizione abilitata di

un singolo componente determina un passaggio di stato della rete. Una Transizione Comportamentale si distingue da un'altra non solo per l'*id*, uguale a quello della propria `ComportamentalTransition`, ma anche per lo stato sorgente e di destinazione nella rete. Questo perchè in uno Spazio Comportamentale, la stessa transizione può scattare in stati diversi dell'Automa.

2.1.3 Spazio Comportamentale come Automa

Per rappresentare il comportamento di uno Spazio Comportamentale, abbiamo definito una sottoclasse parametrica `SpaceAutoma<S>`, dove `S` deve essere una sottoclasse di `SpaceState`, la quale estende da `Automa<S, SpaceTransition<S>>`. La classe `SpaceAutoma` ridefinisce il comportamento per la determinazione degli stati *accettabili*, restituendo tutti gli stati nella rete che sono anche *finali*. Inoltre, include la logica per la *potatura* e *ridenominazione*.

2.1.4 Potatura, ridenominazione e etichettatura

Dopo aver generato lo Spazio Comportamentale di una rete, l'operazione di *potatura* consiste nel rimuovere quegli stati (e transizioni) dello spazio da cui non si può raggiungere alcuno stato finale. La ridenominazione, invece, riguarda l'assegnare a ogni stato dello Spazio Comportamentale un identificatore univoco, senza che questa operazione modifichi lo spazio. Anche la etichettatura non modifica lo spazio, e consiste nel far corrispondere a ogni transizione una etichetta di osservabilità e una di rilevanza. Nel caso in cui la transizione non sia *osservabile* oppure *rilevante*, allora viene applicata una etichetta nulla (ϵ) rispettivamente per l'etichetta *osservabile* o l'etichetta *rilevante*.

2.2 Algoritmi per lo Spazio Comportamentale

In questa sezione verranno presentati gli algoritmi implementati per quanto riguarda le operazioni sullo Spazio Comportamentale, tenendo in considerazione anche delle complessità temporali e spaziali.

2.2.1 Generazione dello Spazio Comportamentale

function GENERA(net)
    $s_0 \Leftarrow$ stato iniziale di ogni Automa e evento di ogni link (inizialmente vuoti)
    inserisce s_0 nell'Automa Spazio Comportamentale, e lo imposta come iniziale
    BUILDSPACE(s_0, transizioni abilitate per s_0)
    effettua la potatura e la ridenominazione dello Spazio Comportamentale generato
    return Spazio Comportamentale generato
end function

function BUILDSPACE(s, $entr$) ▷ $entr$ rappresenta la lista di transizioni abilitate per lo stato s
  if $entr.length > 1$ **then**
    for all $t \in entr$ **do**
      fa tornare la rete allo stato s
      SCATTOTRANSIZIONE(s, t)
    end for
  else if $entr.length = 1$ **then**
    SCATTOTRANSIZIONE(s, unica transizione di $entr$)
  else
    inserisci s nello Spazio Comportamentale
  end if
end function

function SCATTOTRANSIZIONE(s, t)
  effettua la transizione t nella rete
  $s_{dest} \Leftarrow$ stato corrente di ogni automa e contenuto di ogni link dopo la transizione t
  if s_{dest} non è già presente nella rete **then**
    inserisci s_{dest} nello Spazio Comportamentale
  end if
  $t_{comp} \Leftarrow$ transizione da s a s_{dest}
  if t_{comp} non è già presente nella rete **then**
    inserisci t_{comp} nello Spazio Comportamentale
    BUILDSPACE(s_{dest}, transizioni abilitate per s_{dest})
  end if
end function

2.2.2 Potatura

function POTATURA()
  $ns_{prev} \Leftarrow$ numero di stati nello Spazio Comportamentale
  $states_{copy} \Leftarrow$ copia degli stati dello Spazio Comportamentale
  for all $s \in states_{copy}$ **do**
    CHECKPOTATURA(s)
  end for
  $ns_{now} \Leftarrow$ numero di stati nello Spazio Comportamentale dopo potatura
  return $ns_{prev} \neq ns_{now}$ ▷ Se diversi vuol dire che c'è stato almeno uno stato rimosso
end function

function CHECKPOTATURA(s)
  if s è contenuto nello Spazio Comportamentale, è finale e non ha transizioni in uscita **then**
    $t_{in} \Leftarrow$ transizioni in entrata di s
    rimuovi lo stato s dallo Spazio Comportamentale

for all $t \in t_{in}$ **do**
checkPotatura(stato sorgente della transizione t) ▷ Controlla se lo stato precedente sia da potare
end for
end if
end function

2.2.3 Ridenominazione

function ridenominazione()
$r \Leftarrow \emptyset$ ▷ Mappa un intero con uno stato
$i \Leftarrow 0$
for all $s \in$ stati dello Spazio Comportamentale **do**
aggiunge a r la chiave i con valore s
$interc \Leftarrow$ interconnessioni dello stato s
rimuove lo stato s dall'automa
assegna a s l'id i
riaggiungi all'Automa lo stato s con le interconnessioni $interc$
end for
return r
end function

Capitolo 3

Spazio Comportamentale relativo ad una Osservazione Lineare

In questo capitolo vengono discussi gli Spazi Comportamentali relativi a Osservazioni Lineari, la loro implementazione e l'algoritmo sviluppato per generarli.

3.1 Definizione di Spazio Comportamentale relativo ad una Osservazione Lineare

Uno Spazio Comportamentale relativo ad una Osservazione Lineare è uno Spazio Comportamentale che, come dice il nome, è generato a partire da una Osservazione Lineare, cioè una sequenza di eventi osservabili, verificatisi (nell'ordine stabilito dalla sequenza) lungo una traiettoria della rete di FA comportamentali. Per traiettoria di una rete di automi si intende una sequenza di transizioni di componenti che porta dallo stato iniziale della rete a uno stato finale.

3.1.1 Stato Comportamentale di una Osservazione Lineare

Uno Stato Comportamentale di una Osservazione Lineare, in aggiunta agli stati dei componenti e dei link, contiene anche un indice dell'osservazione lineare data. Lo stato iniziale dello Spazio Comportamentale relativo a un'osservazione lineare contiene quindi, oltre lo stato iniziale di tutti i componenti e di tutti i link, anche il valore 0 dell'indice dell'osservazione. Uno stato dello Spazio Comportamentale relativo a un'osservazione lineare è finale se tutti i link sono vuoti e il valore dell'indice è pari alla lunghezza del vettore delle osservazioni lineari. L'implementazione di questo tipo di Stato, nella classe `SpaceStateObs`, include quindi un campo per l'indice dell'osservazione lineare e un campo contenente

la lunghezza del vettore delle osservazioni lineari, in modo da poter stabilire se tale stato può essere considerato finale.

3.2 Algoritmo per la generazione

function GENERA(*net*, *obs*) ▷ *obs* rappresenta il vettore di osservazioni lineari
  index ⇐ 0 ▷ inizializza la variabile globale *index*
  s_0 ⇐ stato iniziale di ogni Automa e evento di ogni link (inizialmente vuoti), indice 0 e *obs.length*
  inserisce s_0 nell'Automa Spazio Comportamentale con Osservazioni, e lo imposta come iniziale
  BUILDSPACE(s_0, transizioni abilitate per s_0)
  effettua la potatura e la ridenominazione dello Spazio Comportamentale con Osservazioni generato
  return Spazio Comportamentale con Osservazioni generato
end function

function BUILDSPACE(*s*, *entr*) ▷ *entr* rappresenta la lista di transizioni abilitate per lo stato *s*
  if *entr.length* > 1 **then**
    for all $t \in entr$ **do**
      fa tornare la rete allo stato *s*
      SCATTOTRANSIZIONE(*s*, *t*)
    end for
  else if *entr.length* = 1 **then**
    SCATTOTRANSIZIONE(*s*, unica transizione di *entr*)
  else
    inserisci *s* nello Spazio Comportamentale
  end if
end function

function SCATTOTRANSIZIONE(*s*, *t*)
  effettua la transizione *t* nella rete
  if *index* < *obs.lenght*&& l'etichetta osservabile di *t* = *obs*[*i*] **then**
    index++
  end if
  s_{dest} ⇐ stato corrente di ogni automa e contenuto di ogni link dopo la transizione *t*, indice *index* e *obs.lenght*
  if s_{dest} non è già presente nella rete **then**
    inserisci s_{dest} nello Spazio Comportamentale con Osservazioni
  end if
  t_{comp} ⇐ transizione da *s* a s_{dest}
  if t_{comp} non è già presente nella rete **then**
    inserisci t_{comp} nello Spazio Comportamentale con Osservazioni
    BUILDSPACE(s_{dest}, enabledTransitions(s_{dest}))

```
    end if
end function

function enabledTransitions(s)
    entr ⇐ ∅
    tr ⇐ transizioni abilitate per lo stato s
    for all t ∈ tr do
        if t non ha una etichetta di osservabilità then
            aggiungi t a entr
        else if index < obs.lenght&& l'etichetta osservabile di t = obs[i] then
            aggiungi t a entr
        end if
    end for
    return entr
end function
```

Capitolo 4

Diagnosi relativa ad una Osservazione Lineare

La diagnosi relativa a un'osservazione lineare è l'alternativa delle espressioni regolari di rilevanza corrispondenti agli stati finali dello spazio comportamentale inerente all'osservazione stessa. A ogni stato finale dello spazio di rilevanza relativo a un'osservazione, corrisponde l'espressione regolare ottenuta concatenando le etichette di rilevanza lungo ciascuna traiettoria che porta dallo stato iniziale allo stato finale considerato.

Capitolo 5

Diagnosticatore

5.1 Il Diagnosticatore

Per definire il concetto di *Diagnosticatore*, è prima necessario introdurre altri concetti, come la *Chiusura Silenziosa* e lo *Spazio delle Chiusure*.

5.1.1 Chiusura Silenziosa

La chiusura silenziosa di uno stato s è costituita dal sottospazio dello Spazio Comportamentale che contiene tutti (e soli) gli stati raggiungibili a partire da s attraverso cammini contenenti esclusivamente transizioni non osservabili (dette anche silenziose). s è lo stato d'ingresso della sua chiusura, mentre uno stato d'uscita è uno stato che sia finale o dotato di transizioni osservabili uscenti entro lo Spazio Comportamentale . Una Chiusura Silenziosa Decorata è una chiusura silenziosa in cui ogni stato s' è stato decorato, cioè è stata associata l'espressione regolare di rilevanza relativa a tutti i cammini che portano dallo stato d'ingresso della chiusura a s' entro la chiusura silenziosa. La nostra implementazione `SilentClosure` estende dalla classe Automa `SpaceAutomaComportamentale`, oltre che implementare l'interfaccia `StateInteface`. La classe estende il comportamento dello `SpaceAutoma`, aggiungendo campi riguardanti le decorazioni degli stati e informazioni sugli stati d'uscita.

5.1.2 Spazio delle Chiusure

Le chiusure silenziose relative allo Spazio Comportamentaledi una rete di *Automa a Stati Finiti Comportamentale*sono fra loro interconnesse mediante transizioni osservabili, uscenti da stati d'uscita della chiusura sorgente e dirette allo stato d'ingresso della chiusura destinazione (dove la chiusura sorgente può coincidere con la chiusura destinazione). Lo stato d'ingresso di una chiusura può appartenere anche ad altre chiusure, dove però non è uno stato d'ingresso. Tale struttura prende il nome di *spazio delle chiusure*. Uno Spazio

delle Chiusure decorato è semplicemente lo spazio delle chiusure in cui ciascuna chiusura è decorata. L'implementazione nel nostro programma dello spazio delle chiusure (classe `ClosureSpace`) eredita da `Automa<SilentClosure, Transition<SilentClosure>>`, utilizza come Stato quindi la classe `SilentClosure`.

5.1.3 Diagnosticatore

Un *Diagnosticatore* può essere visto come un *Automa a Stati Finiti*dove ogni stato corrisponde a una chiusura silenziosa (di cui però non interessa conoscere il contenuto), a ciascuno stato di accettazione è associata una diagnosi, ogni transizione è dotata di etichetta di osservabilità e di una espressione regolare. Per passare da uno *Spazio delle Chiusure* a un *Diagnosticatore* è necessario eseguire questi passi:

- Ricopiare la decorazione di ciascuno stato d'uscita entro le chiusure sulla transizione osservabile da esso uscente, concatenando alla decorazione l'eventuale etichetta di rilevanza relativa alla transizione osservabile stessa
- Associare a ciascuna chiusura che contenga degli stati finali la sua diagnosi

5.2 Algoritmi

5.2.1 Generazione Chiusura Silenziosa

Pseudocodice

function GENERAZIONE(*spazio*, *s*)
  closure $\Leftarrow \emptyset$ ▷ variabile globale rappresentante Chiusura Silenziosa
  if *s* è contenuto in *spazio* e *s* o è lo stato iniziale oppure ha transizioni osservabili in ingresso **then**
    SILENTCLOSURE()
  end if
  return *closure*
end function

function SILENTCLOSURE()
  inserisci *s* in *closure*
  imposta *s* come stato iniziale di *closure*
  queue $\Leftarrow$ transizioni uscenti da *s*
  while *queue.length* > 0 **do**
    current $\Leftarrow$ *queue.pop*()
    if *current* non è osservabile **then**
      $t_{dest} \Leftarrow$ stato destinazione di *current*
      inserisci t_{dest} in *closure*
      aggiungi *current* a *closure*
      aggiungi a *queue* tutte le transizioni uscenti da t_{dest}
    else

imposta lo stato sorgente di *current* come stato di uscita in *closure*
        end if
    end while
end function

Complessità

La funzione GENERAZIONE ha una complessità temporale lineare, $\mathcal{O}(n)$. Il costo è dovuto al controllo di appartenenza di s allo *spazio*, che richiede una scansione lineare di tutti gli stati presenti nello *spazio*, $\mathcal{O}(n_{stati})$, e al controllo della presenza di transizioni osservabili in ingresso per s, il quale richiede anch'esso una scansione lineare ma in questo caso di tutte le transizioni in entrata per s, cioè $\mathcal{O}(n_{t_{in}})$.

Per quanto riguarda il modulo SILENTCLOSURE, il ciclo **while** fa scorrere tutte le transizioni uscenti da s e per ciascuna transizione inserisce, nel caso la transizione non sia osservabile, il suo stato di destinazione nella chiusura. Ciò comporta un costo lineare per la scansione degli stati per controllare se lo stato sia già presente nella *closure*. L'istruzione è eseguita al più una volta per ogni transizione uscente da s, il costo complessivo temporale della funzione è quindi non lineare $\mathcal{O}(n_{t_{out}} \cdot n_s)$.

In merito alla complessità spaziale, l'algoritmo non è ricorsivo, quindi nel calcolo della complessità teniamo conto solo dello spazio occupato dalle strutture dati: nel nostro caso *spazio*, *diagn* e la coda *queue*.

5.2.2 Generazione Diagnosticatore

Pseudocodice

```
function GENERAZIONE(spazio)
    salva spazio come variabile globale
    diagn ⇐ ∅                ▷ variabile globale rappresentante il Diagnosticatore
    initial ⇐ crea chiusura silenziosa dello stato iniziale di space
    queue ⇐ ∅
    aggiungi initial a queue
    COMPOSECLOSURE(queue)
    return diagn
end function

function COMPOSECLOSURE(queue)
    while queue.length >0 do
        closure ⇐ queue.pop()
        inserisci closure in diagn
        for all s ∈ stati di uscita di closure do
            for all t ∈ transizioni in uscita da spazio do
                if t non è osservabile then
                    HANDLETRANSITION(t, closure, queue)
```

end if
end for
end for
end while
end function

function HANDLETRANSITION(t, *closure*, *queue*)
sink $\Leftarrow \emptyset$
$t_{dest} \Leftarrow$ stato destinazione di t
if *diagn* non contiene t_{dest} **then**
sink $\Leftarrow$ nuova chisura silenziosa di *spazio* e stato t_{dest}
decora *sink*
inserisci *sink* in *diagn*
inserisci *sink* in *queue*
else
sink $\Leftarrow$ recupera la chiusura silenziosa da *diagn* con stato t_{dest}
end if
newT $\Leftarrow$ nuova transizione con l'id di t, sorgente *closure* e destinazione *sink*
$t_{source} \Leftarrow$ stato sorgente di t
imposta l'etichetta osservabile di *newT* con l'etichetta osservabile di t
imposta l'etichetta rilevante di *newT* con la concatenazione delle decorazioni di t_{source} e t
aggiunge *newT* a *diagn*
end function

Complessità

Complessità temporale: - creazione initial $\mathcal{O}(n^2)$
- forall stati $\mathcal{O}(n_s)$ - forall transiz $\mathcal{O}(n_t)$ - handleTransition: - creazione sink $\mathcal{O}(n^2)$ - decorazione sink ?
.

Capitolo 6

Diagnosi tramite Diagnosticatore

A differenza del metodo dedicato al calcolo della diagnosi relativa a una singola osservazione lineare, l'uso del diagnosticatore permette di calcolare la diagnosi relativa a qualsiasi osservazione lineare della rete in uso. Dato uno Spazio Comportamentale, una osservazione lineare O e il diagnosticatore, è possibile calcolare la diagnosi seguendo questi passi:

- a partire dallo stato iniziale del diagnosticatore, si seguono tutte le traiettorie del diagnosticatore stesso che producono O.
- per ciascuna traiettoria si costruisce un'espressione regolare che è la concatenazione delle espressioni regolari che si trovano lungo le transizioni (osservabili) della stessa.
- a tale concatenazione si fa seguire (sempre mediante concatenazione) la diagnosi relativa allo stato di accettazione (del diagnosticatore) raggiunto alla fine della traiettoria.
- la diagnosi relativa a O è data dall'alternativa di tutte le espressioni così costruite

La nostra implementazione dell'algoritmo, risiede nella classe `LinearDiagnosis`.

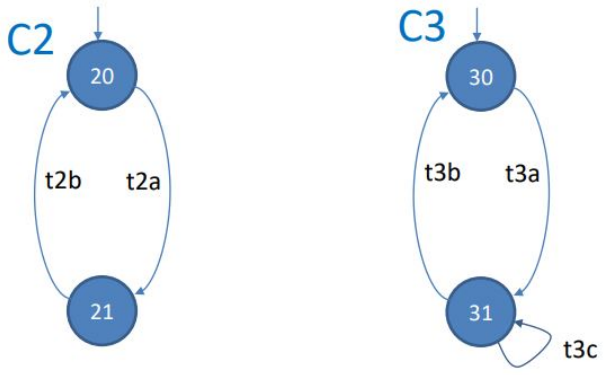
Capitolo 7

Benchmark

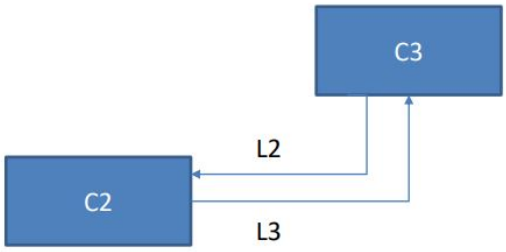
7.1 Rete principale

7.1.1 Composizione

Automi considerati



Topologia



Transizioni

C2

- t2a: $e_2(L_2)/\{e_3(L_3)\}$
- t2b: $/\{e_3(L_3)\}$

C3

- t3a: $/\{e_2(L_2)\}$
- t3b: $e_3(L_3)$
- t3c: $e_3(L_3)$

Osservabilità

C2

- t2a: o_2
- t2b: ϵ

C3

- t3a: o_3
- t3b: ϵ
- t3c: ϵ

Rilevanza

C2

- t2a: ϵ
- t2b: r

C3

- t3a: ϵ
- t3b: ϵ
- t3c: f

7.1.2 Diagnosi

Il calcolo della diagnosi senza diagnosticatore, per l'osservazione lineare [o_3,o_2], impiega un tempo pari a *0,11s*, occupando uno spazio di memoria di *7,49MB*. La diagnosi risultante è $\epsilon\epsilon$(f(r(fϵ|$\epsilon\epsilon$)|ϵ)|$\epsilon\epsilon$)

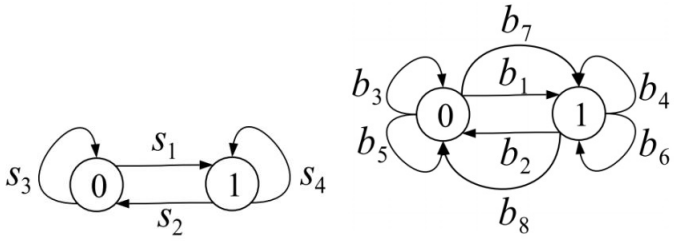
Il calcolo della diagnosi con diagnosticatore, invece, impiega un tempo minore e occupa meno spazio rispetto al calcolo senza diagnosticatore. Il risultato ottenuto è un tempo pari a *0,03s* e uno spazio di memoria occupato di *2,23MB*. La diagnosi risultante è $\epsilon\epsilon\epsilon\epsilon\epsilon(fr\epsilon|frf|f\epsilon|\epsilon\epsilon)$

```
Tempo Impiegato: 0,03s

Spazio Occupato: 2,23MB

Diagnosi Lineare (con Diagnosticatore) trovata per osservazione [o3, o2]: εεεεε(frε|frf|fε|εε)
```

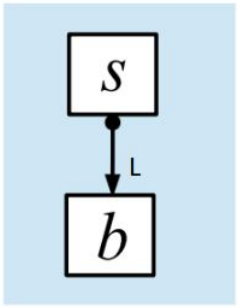
I due diversi algoritmi hanno generato la stessa diagnosi.

7.2 Rete Altra Rete

Automi



Topologia



Transizioni

s

- s1: $/\{op(L)\}$
- s2: $/\{cl(L)\}$
- s3: $/\{cl(L)\}$
- s4: $/\{op(L)\}$

b

- b1: $\{op(L)\}$
- b2: $\{cl(L)\}$
- b3: $\{op(L)\}$
- b4: $\{cl(L)\}$

- b5: $\{cl(L)\}$
- b6: $\{op(L)\}$
- b7: $\{cl(L)\}$
- b8: $\{op(L)\}$

Osservabilità

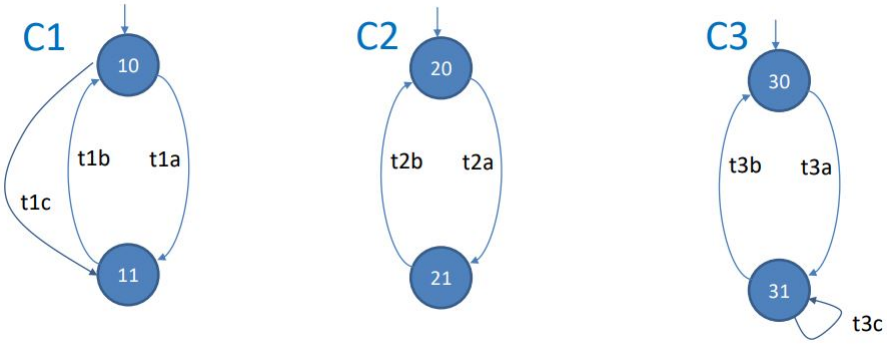
S	r
	• b1: opn
• s1: act	• b2: cls
	• b3: ϵ
• s2: sby	• b4: ϵ
	• b5: nop
• s3: ϵ	• b6: nop
	• b7: opn
• s4: ϵ	• b8: cls

Rilevanza

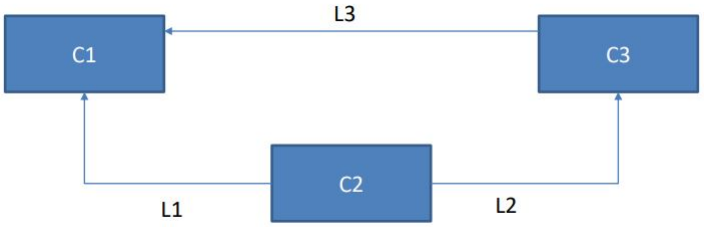
S	r
	• b1: ϵ
• s1: ϵ	• b2: ϵ
	• b3: f_3
• s2: ϵ	• b4: f_4
	• b5: ϵ
• s3: f_1	• b6: ϵ
	• b7: f_5
• s4: f_2	• b8: f_6

7.3 Rete di Benchmark

Automi



Topologia



Transizioni

C1

- t1a: $e_1(L_1)$
- t1b: $e_2(L_3)$
- t1c:

C2

- t2a: $/\{e_1(L_1),e_3(L_2)\}$
- t2b: $/\{e_1(L_1)\}$

C3

- t3a: $/\{e_2(L_3)\}$
- t3b: $e_3(L_2)$
- t3c: $e_3(L_2)$

Osservabilità

C1

- t1a: ϵ
- t1b: ϵ
- t1c: ϵ

C2

- t2a: o_1
- t2b: o_2

C3

- t3a: ϵ
- t3b: ϵ
- t3c: ϵ

Rilevanza

C1	C2	C3
• t1a: ϵ	• t2a: ϵ	• t3a: ϵ
• t1b: ϵ		• t3b: ϵ
• t1c: f_1	• t2b: ϵ	• t3c: f_3

7.3.1 Diagnosi

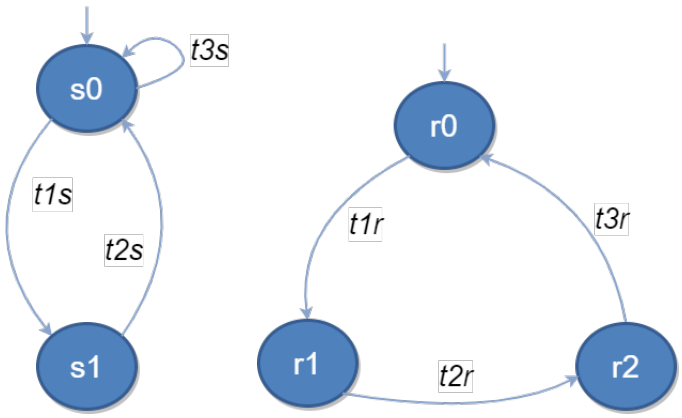
Il calcolo della diagnosi senza diagnosticatore, per l'osservazione lineare $[o_1,o_2]$, impiega un tempo pari a *0,39s*, occupando uno spazio di memoria di *1,54MB*. La diagnosi risultante, già semplificata, è $f_1|f_3|\epsilon|f_1f_1|f_1f_3|f_3f_1$. Il calcolo della diagnosi con diagnosticatore, invece, impiega un tempo minore e occupa meno spazio rispetto al calcolo senza diagnosticatore. Il risultato ottenuto è un tempo pari a *0,05s* e uno spazio di memoria occupato di *0,61MB*. La diagnosi risultante, già semplificata, è $f_1|f_3|\epsilon|f_1f_1|f_1f_3|f_3f_1$. I due diversi algoritmi hanno generato la stessa diagnosi, come da definizione.

7.4 Rete creata da noi

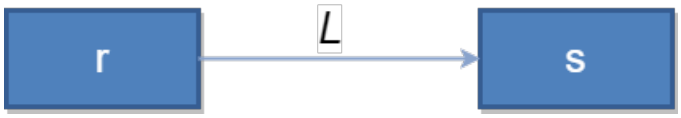
Inizialmente prendiamo in considerazione una rete costituita da solo due MSF Comportamentali. Successivamente integreremo alla rete una MSF Comportamentale aggiuntiva e valuteremo le variazioni dei risultati nella sperimentazione.

7.4.1 Composizione iniziale

Automi



Topologia



Transizioni

S

- t1s: $e_2(L)$
- t2s:
- t3s: $e_1(L)$

r

- t1r: $/\{e_2(L)\}$
- t2r: $/\{e_1(L)\}$
- t3r:

Osservabilità

S

- t1s: o_2
- t2s: ϵ
- t3s: ϵ

r

- t1r: o_1
- t2r: ϵ
- t3r: ϵ

Rilevanza

S

- t1s: ϵ
- t2s: f_1
- t3s: ϵ

r

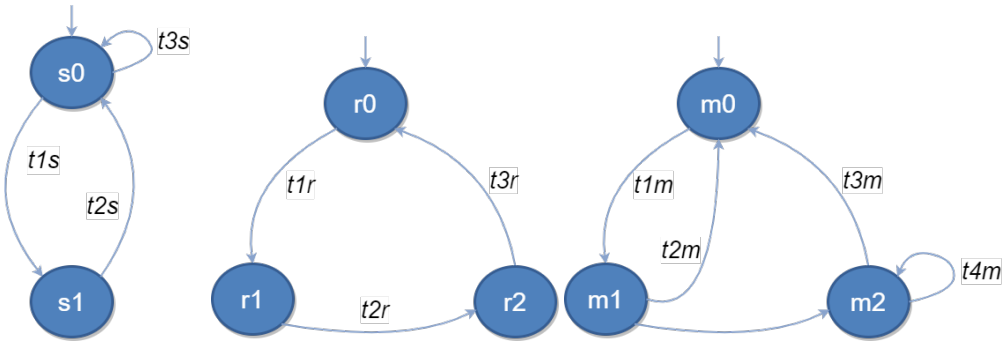
- t1r: ϵ
- t2r: ϵ
- t3r: f_2

7.4.2 Diagnosi rete iniziale

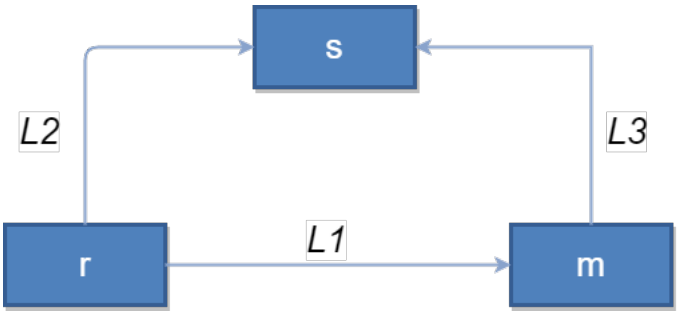
Il calcolo della diagnosi senza diagnosticatore, per l'osservazione lineare [o_1,o_2], impiega un tempo pari a *0,04s*, occupando uno spazio di memoria di *5,96MB*. La diagnosi risultante, già semplificata, è $f_2f_1|f_1f_2|\epsilon|f_1$. Il calcolo della diagnosi con diagnosticatore, invece, impiega un tempo minore e occupa meno spazio rispetto al calcolo senza diagnosticatore. Il risultato ottenuto è un tempo pari a *0,02s* e uno spazio di memoria occupato di *2,60MB*. La diagnosi risultante, già semplificata, è $f_2f_1|f_1f_2|\epsilon|f_1$. I due diversi algoritmi hanno generato la stessa diagnosi, come da definizione.

7.4.3 Composizione a 3 automi

Automi



Topologia



Transizioni

s

- t1s: $e_2(L_1)$
- t2s:
- t3s: $e_3(L_3)$

r

- t1r: $/\{e_3(L_3)\}$
- t2r: $/\{e_1(L_1)\}$
- t3r:

m

- t1m: $e_1(L_1)/\{e_3(L_3)\}$
- t2m: $e_2(L_1)$
- t3m: $/\{e_2(L_3)\}$
- t4m:

Osservabilità

s

- t1s: ϵ
- t2s: ϵ
- t3s: ϵ

r

- t1r: o_1
- t2r: o_2
- t3r: ϵ

m

- t1m: ϵ
- t2m: ϵ
- t3m: o_3

- t4m: ϵ

Rilevanza

s	r	m
• t1s: ϵ	• t1r: ϵ	• t1m: f_3
• t2s: f_1	• t2r: ϵ	• t2m: ϵ
• t3s: ϵ	• t3r: f_2	• t3m: ϵ
		• t4m: f_4

7.4.4 Diagnosi rete a 3 automi

Il calcolo della diagnosi senza diagnosticatore, per l'osservazione lineare $[o_2]$, impiega un tempo pari a *0,06s*, occupando uno spazio di memoria di *10,88MB*. La diagnosi risultante, già semplificata, è $f_2f_1|f_1f_2|\epsilon|f_1$. Il calcolo della diagnosi con diagnosticatore, invece, impiega un tempo minore e occupa meno spazio rispetto al calcolo senza diagnosticatore. Il risultato ottenuto è un tempo pari a *0,02s* e uno spazio di memoria occupato di *2,97MB*. La diagnosi risultante, già semplificata, è $f_2f_1|f_1f_2|\epsilon|f_1$. I due diversi algoritmi hanno generato la stessa diagnosi, come da definizione.

Capitolo 8

Manuale utente

8.1 Interfaccia Utente

8.1.1 Comandi

Il programma presenta una serie di comandi a disposizione dell'utente.

Schermata principale

All'avvio i comandi mostrati sono i seguenti (dove non specificato, i comandi non richiedono parametri):

- *exit*: Esce dal programma
- *readcommands*: Legge ed esegue una serie di comandi da file .txt - Sintassi: *readcommands [fileName].txt*" (verrà presentato nel dettaglio successivamente)
- *newnet*: Crea una nuova rete di *Automa a Stati Finiti Comportamentale*
- *loadnet*: Carica una rete di *Automa a Stati Finiti Comportamentale*da file - Sintassi: *loadnet [fileName]*"
- *shownet*: Mostra una descrizione della rete di *Automa a Stati Finiti Comportamentale* caricata
- *spaceops*: Sottomenù per le operazioni su Spazi Comportamentali

Le operazioni *shownet* e *spaceops* richiedono di aver caricato una rete tramite il comando *loadnet*.

Schermata creazione nuova rete

Il comando *newnet* apre il sottomenù per la creazione di una nuova rete di *Automa a Stati Finiti Comportamentale*, e presenta i seguenti comandi:

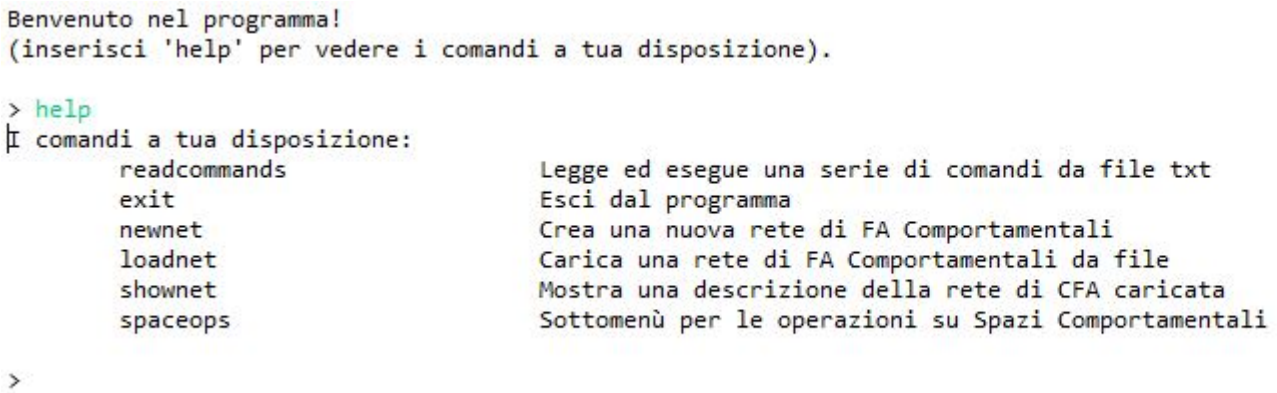


Figura 8.1: help nella schermata principale del programma

- *back*: Torna al menù precedente
- *newcfa*: Crea una nuova MSF Comportamentale - Sintassi: *newcfa [id]*
- *newlink*: Crea un nuovo Link - Sintassi: *newlink [id]*
- *newevent*: Crea un nuovo Evento - Sintassi: *newevent [id]*
- *linkcfas*: Collega due *Automa a Stati Finiti Comportamentale* tramite un link - Sintassi: *linkcfas [linkId]*
- *showcfas*: Mostra le *Automa a Stati Finiti Comportamentale* attualmente creati
- *showlinks*: Mostra i Link attualmente creati
- *showevents*: Mostra gli Event attualmente creati
- *savenet*: Salva la rete su file specificato - Sintassi: *savenet [fileName]*
- *resetnet*: Resetta tutti i dati creati per la rete (*Automa a Stati Finiti Comportamentale*, Eventi, Link...)

Alla creazione di un nuovo *link*, non viene chiesto immediatamente quali sono gli *Automa a Stati Finiti Comportamentale*da collegare, questo perchè molto probabilmente gli *Automa a Stati Finiti Comportamentale*desiderati non sono ancora stati creati. Per collegare due *Automa a Stati Finiti Comportamentale*dopo aver terminato la loro creazione, è sufficiente utilizzare il comando *linkcfas*.

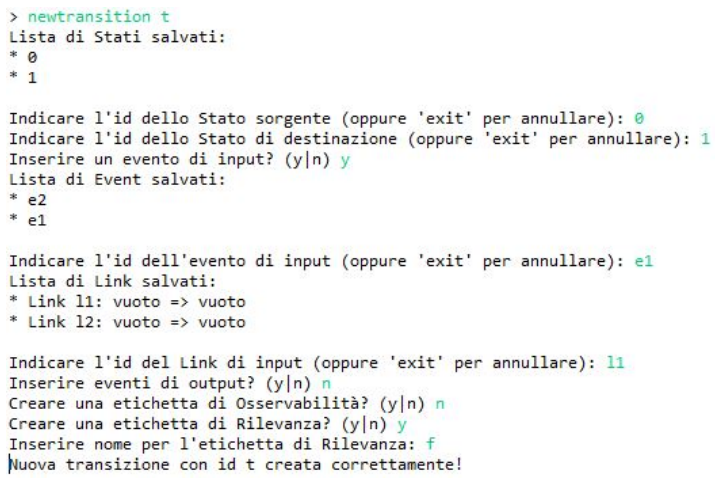


Figura 8.2: esempio di creazione di una transizione

Schermata creazione nuovo *Automa a Stati Finiti Comportamentale*

Attraverso il comando *newcfa* è possibile creare un nuovo *Automa a Stati Finiti Comportamentale*. Il comando apre un apposito sottomenù con i seguenti comandi:

- *newstate*: Crea un nuovo Stato Comportamentale - Sintassi: *newstate [id]*
- *newtransition*: Crea una nuova Transizione Comportamentale - Sintassi: *newtransition [id]*
- *setinitial*: Imposta lo stato con l'id specificato come iniziale per la *Automa a Stati Finiti Comportamentale* - Sintassi: *setinitial [stateId]*
- *showstates*: Mostra gli Stati attualmente creati
- *showtransitions*: Mostra le Transizioni attualmente create
- *savecfa*: Salva l'*Automa a Stati Finiti Comportamentale* su cui si sta lavorando
- *resetcfa*: Resetta tutti i dati creati per l'*Automa a Stati Finiti Comportamentale*(Stati, Transizioni...)

Per creare una nuova transizione tramite il comando *newtransition* è necessario aver creato almeno due stati tramite il comando *newstate*. Per salvare l'*Automa a Stati Finiti Comportamentale*tramite comando *newtransition*, è richiesto di aver creato almeno due stati, almeno una transizione, e di aver specificato lo stato iniziale per l'automa.

Schermata operazioni su Spazio Comportamentale

L'ultimo sottomenù da prendere in considerazione è il sottomenù relativo alle operazioni sugli Spazi Comportamentali. Per accederci, è sufficiente inserire il comando *spaceops* nella home. Come già specificato, è richiesto di aver caricato una rete.

Il sottomenù presenta i seguenti comandi:

- *generatespace*: Genera uno spazio comportamentale da una rete di CFA
- *generatespaceobs*: Genera una Osservazione Lineare da una rete di CFA
- *showobs*: Mostra le Osservazioni Lineari effettuate sulla rete
- *diagnosticatore*: Genera un Diagnosticatore a partire dallo Spazio Comportamentale
- *diagnosiobs*: Computa la diagnosi di un Osservazione Lineare già calcolata
- *diagnosiobswd*: Computa diagnosi di Osservazione Lineare tramite Diagnosticatore
- *updatenet*: Aggiorna il file della rete attualmente caricata

8.1.2 Automazione di comandi tramite file

Attraverso il comando *readcommands* è possibile leggere da un file di testo (in formato *.txt*) una lista di comandi che verranno eseguiti in sequenza e automaticamente. È possibile utilizzare tutti i comandi presenti nel programma, con l'aggiunta di due speciali comandi:

- *readcommands*: è possibile richiamare in modo ricorsivo il comando *readcommands*, permettendo di creare strutture a blocchi, ad esempio salvando Macchine a Stati Finiti su file separati.
- *robot [nomecomando]*: nel caso di comandi che richiedano un input dell'utente, il comando speciale *robot* permette di automatizzare l'inserimento dell'input. Sintassi:

```
robot nomecomando
    testo da digitare
    ...
    testo da digitare
endrobot
```

Capitolo 9

Conclusioni