Progetto Algoritmi e Strutture Dati

De Ramundo Marco, Parolin Yanez Diego

Aprile 2021

1 Introduzione

Il progetto richiesto e trattato in questo report riguarda il programma disegnato e scritto utile per operazioni e analisi su una rete di automi a stati finiti. Obiettivo principale del programma è quello di poter operare su tali automi e generare delle strutture particolari quali dei spazi comportamentali utili per eseguire una successiva analisi e valutazione.

Durante la progettazione e realizzazione del programma sono state adoperate delle scelte utili sia in valutazione della complessità spaziale ma soprattutto temporale del software sia per usabilità del programma e ipotetico sviluppo futuro. Tali scelte verranno manifestate e argomentate nei capitoli successivi.

La struttura di questo report è suddivisa in pochi capitoli. Subito dopo l'introduzione si parlerà dei strumenti utilizzati per la realizzazione del software, sia in termini di linguaggi di programmazione e di librerie che di tipologie di file dati utilizzati per la creazione ma anche per l'esecuzione del programma. Il terzo capitolo affronterà quali sono le strutture dati che sono alla base del progetto e gli algoritmi che operano su tali strutture, commentando brevemente lo pseudocodice delle varie funzioni presenti nel programma; verranno accennate anche alcune delle funzioni di supporto utilizzate nel progetto. Trattato il programma, si passerà con il quarto capitolo all'analisi delle prestazioni, sia a livello teorico valutando la complessità teorica dell'algoritmo, possibili differenze di complessità con lo pseudocodice originario e le effettive prestazioni del programma in esecuzione, sia in termini di tempo che di memoria. Infine nel quinto e ultimo capitolo verranno mostrati alcuni esempi di reti di automi che possono essere elaborate dal programma e quali sono i risultati di una loro possibile analisi.

La presente relazione è pensata per essere esaminata assieme alla documentazione dell'elaborato fornito a inizio progetto. Vi potranno essere riferimenti teorici da tale file che non saranno spiegati esaustivamente in questo documento. Pertanto per qualsiasi precisazione l'invito è quello di esaminare tale documentazione qualora dovessero servire dei chiarimenti.

Il codice del programma e tutta la documentazione è disponibile su `https://github.com/dr-marco/FA_tool`

2 Strumenti utilizzati

2.1 Comando di esempio

Per eseguire il programma è sufficiente lanciare da terminale il comando

`~python project_function.py -h`

Questo comando permette di stampare a video un semplice help che aiuta chi deve eseguire il programma a digitare correttamente il comando.

`~python project_function.py -net net.json -ol o3 o2 -ol2 o3 o2 o3 o2`

Questo è un esempio di possibile comando per esecuzione del programma con input una rete e dei vettori di osservazione. Altre modalità previste dal programma verranno mostrate nel prossimo paragrafo dove verranno descritti i parametri disponibili utili per eseguire le operazioni desiderate.

2.2 Librerie e Parametri

La scelta del linguaggio di programmazione è ricaduta su Python, per via della sua versabilità, adattabilità e per la presenza di una vasta disponibilità di librerie, tra le quali noi abbiamo utilizzato:

1. **copy** per duplicare gli oggetti senza mantenere alcun riferimento all'oggetto d'origine (deep copy)
2. **time** per monitorare tempistiche di esecuzione
3. **memory_profile** per monitorare memoria fisica occupata dalle varie funzioni
4. **json** per salvare/caricare gli oggetti di classe create dall'utente in formato json

5. **jsonpickle** per salvare/caricare i file di tipo json

6. **argparse** per aggiungere opzioni da linea di comando

Come accennato, tramite la libreria argparse, è stato possibile aggiungere delle opzioni per il funzionamento del programma, quando questo viene lanciato tramite linea di commando. Richiede di specificare il file di input

1. **-net** + *pathtofile.json*: se si vogliono applicare tutti gli step dell'algoritmo

2. **-b** + *pathtofile.json*: se si vuole generare una spazio osservabile dato uno spazio comportamentale oppure lo spazio delle chiusure silenziose

3. **-o** + *pathtofile.json*: se si vuole generare la diagnosi relativa a uno spazio comportamentale riferito a una osservazione

4. **-cs** + *pathtofile.json*: se si vuole generare il diagnosticatore partendo dallo spazio delle chiusure silenziose

5. -**dgn** + *pathtofile.json*: se si vuole generare la diagnosi lineare dato il diagnosticatore

Oltre a una serie di opzioni relative agli steps da eseguire, che se omesse (in presenza di -net) equivale a lanciare l'algoritmo con tutti gli steps:

1. **-all** (se è presente -net) per lanciare il programmo comprendendo tutti i passaggi dell'algoritmo

2. **-gb** (se presente -net) per generare lo spazio comportamentale

3. **-go** (se presente -b) per generare lo spazio osservabile

4. **-d** (se presente -o) per generare la diagnosi lineare

5. **-gcs** (se presente -b) per generare lo spazio delle chiusure

6. **-gd** (se presente -cs) per generare il diagnosticatore

7. **-do** (se presente -d) per generare la diagnosi lineare

Infine ci sono altre opzioni indipendenti dalla selezione precedente

1. **-ol** + *'serie di label'* (se presente -net, -all, -o) per specificare il vettore osservazione per generare lo spazio osservabile

2. **-ol2** + *'serie di label'* (se presente -net, -all, -do) per specificare il vettore osservazione per generare la diagnosi lineare

3. **-bk** per lanciare un benchmark

Dunque in input richiede dei file .json, o le specifiche della rete di base, oppure lo stato della rete nelle varie fasi intermedie, questi stati vengono generati e salvati in un file .json in output durante l'esecuzione dei vari steps del programma, dunque non è necessario scriverli manualmente (anche se risulta possibile).

Oltre ai file degli stati intermedi, in output il programma può salvare altri tre file:

1. *summary.txt* contenente il riassunto di tutti gli step eseguiti

2. *diagnosis.txt* contennte il risutlato della diagnosi

3. *lineardiagnosis.txt* contennete il risultato della diagnosi lineare

Per rappresentare e salvare la rete abbiamo deciso di sfruttare il formato json, nel quale abbiamo inventato una nostra struttura, come vedremo successivamente, dove sono espresse in maniera formale tutte le caratteristiche della rete. Per salvare gli stati intermedi è stato preferito salvare gli oggetti in formato json e non pickle (serializzando) per una questione di sicurezza, velocità e facilità nella lettura. Il benchmark non stampa nessun risultato intermedio o il summary della reta, presenta unicamente in output il tempo, in s, di esecuzione del programma.

3 Algortimi e Strutture Dati del programma

3.1 Strutture Dati

Prima di descrivere gli algoritmi alla base del programma è necessario introdurre quali sono gli oggetti che verranno elaborati, come sono strutturati, anche a livello di codice, e come questi oggetti si relazionino tra di loro.

Iniziamo mostrando l'oggetto dato in ingresso all'interno del programma, ovvero la rete di automi a stati finiti.

Brevemente, questa rete è composta da un numero di automi a stati finiti, questi automi sono collegati tra loro tramite link e all'interno degli automi questi sono composti da stati, interconnessi tramite transizioni i quali possono avere degli eventi in ingresso richiesti per l'attivazione oppure degli eventi in uscita, o entrambi.

Quindi abbiamo iniziato partendo con la definizione degli elementi che risiedono alla base degli automi: gli stati, le transizioni e gli eventi. Questi oggetti sono in genere utilizzati per definire un automa a stati finiti in modo completo e dettagliato e potrebbero essere usati più o meno come struttura dati anche in altri contesti in cui non vi è necessariamente bisogno di interagire con rete di automi.

Lo **stato** (`State`) è l'oggetto più semplice, rappresenta un singolo stato di un automa e ha due sole proprietà:

- **id** : stringa che va a identificare univocamente l'oggetto. Potrebbe essere riferito anche come `alias` all'interno del codice
- **isFinal** : variabile booleana che indica se l'oggetto rappresenta uno stato finale

La **transizione** (`Transition`) invece è un collegamento orientato tra due stati e può avere o meno eventi in ingresso/uscita e/o etichette. È composta da:

- **id** : stringa che va a identificare univocamente l'oggetto. Riferito anche come `alias` all'interno del codice
- **start_state** : stato iniziale da cui parte la transizione
- **finish_state** : stato di arrivo della transizione
- **label_oss** : stringa che indica una eventuale etichetta di osservabilità
- **label_rel** : stringa che indica una eventuale etichetta di rilevanza
- **input_event_func** : un oggetto `EventFunction` che indica un possibile evento necessario in ingresso per l'attivazione della transizione
- **output_events_func** : un oggetto `EventFunction` che indica un possibile evento in uscita generato dall'attivazione della transizione

Nel caso di default, in cui non ci sono etichette di osservabilità e di rilevanza, tali etichette vengono istanziate come ϵ mentre nel caso in cui non ci siano eventi in ingresso o in uscita questi vengono settati come `None`.

Gli oggetti `EventFunction` verranno descritti a breve. Prima è necessario introdurre l'oggetto **evento** (`Event`) che rappresenta un generico evento. L'unica proprietà contenuta in evento è:

- **id** : stringa che va a identificare univocamente l'oggetto. Riferito anche come `alias` all'interno del codice

Con questi tre oggetti è quindi possibile rappresentare un'**automa a stati finiti**, in inglese Finite-State Automata o in breve `FA`, come verrà principalmente chiamato da qui in avanti. È formato da:

- **id** : stringa che va a identificare univocamente l'oggetto. Riferito anche come `alias` all'interno del codice
- **list_states** : una lista contenente gli stati che appartengono al `FA`
- **list_trans** : una lista contenente le transizioni tra stati che appartengono al `FA`
- **initial_state** : lo stato iniziale del `FA`
- **actual_state** : uno stato che identifica lo stato attuale durante l'esecuzione dell'automa[1]

In questo modo si descrive un `FA` come l'insieme dei suoi stati, delle sue transizioni e dello stato iniziale. Nel momento in cui si hanno più `FA` che sono interconnessi tra loro si andrà a introdurre il **link** il quale descrive un collegamento tra due automi con un possibile evento caratterizzante. Un oggetto `link` è quindi composto da:

- **id** : stringa che va a identificare univocamente l'oggetto. Riferito anche come `alias` all'interno del codice
- **start_FA** : `FA` iniziale da cui parte il link

[1]Questo attributo non è indispensabile per il compito del programma in senso stretto ma può essere utile in futuro qualora ampliando il programma si necessiti dell'esecuzione dell'automa.

- **finish_FA** : `FA` finale dove termina il link
- **contenuto** : attributo che andrà a contenere un `Evento` che va a caratterizzare il link. Riferito anche come `buffer` all'interno del codice

Come promesso, andiamo a descrivere l'oggetto `EventFunction`. Questo oggetto ha il compito di racchiudere un link e un evento associato. Come oggetto di per sé è ridondante ma ha permesso di garantire la correttezza del programma semplificando non poco la realizzazione del codice in quanto permette di rendere più leggere alcune fasi di controllo all'interno degli algoritmi. Come anticipato, è composto da:

- **event** : l'evento che caratterizza il link tra i due automi
- **link** : il collegamento tra i due automi

A tutti gli effetti `EventFunction` è un oggetto di supporto all'interno del programma.

Infine, per descrivere la **rete** di automi semplicemente si racchiudono gli `FA` e i `link` nell'oggetto `Net`:

- **list_FA** : lista contenente i `FA` che appartengono alla rete
- **list_link** : lista contenente i collegamenti tra i vari `FA` della rete

Questi sono le strutture dati utilizzate all'interno del programma per manipolare le reti di automi e tramite queste classi è possibile applicare degli algoritmi per generare nuove strutture.

Ciò che ora è possibile generare tramite gli oggetti qui sopra presentati con il programma sono gli **spazi comportamentali**, il **diagnosticatore** e gli **spazi delle chiusure**. Tali oggetti saranno fondamentali per eseguire analisi sul comportamento della rete di automi ed effettuare diagnostiche. Andiamo ora a descrivere le loro strutture dati.

Così come si è partiti con gli stati per descrivere le `FA` inizieremo a descrivere gli spazi comportamentali dai **nodi** (`Node`). Tali nodi descrivono un possibile stato della rete di automi. È formato da:

- **id** : stringa che identifica univocamente il nodo

- **alias** : stringa che descrive brevemente il valore degli stati e dei link in quel nodo. Utile per stampare a video il valore del nodo in maniera leggibile
- **final** : variabile booleana che indica se il nodo è un nodo finale
- **list_states_FA** : vettore contenente gli stati che appartengono a tale nodo, ogni stato è riferito a un diverso `FA`
- **list_values_link** : vettore contenente i valori attuali dei link in base agli eventi azionati
- **index_oss** : numero che identifica l'indice di osservazione del nodo. Usato per creare spazi comportamentali relativo a un'osservazione

Per indicare un collegamento tra due nodi useremo una **rotta** `Route`, paragonabile a una transizione per gli automi. È caratterizzato da:

- **id** : stringa che va a identificare univocamente l'oggetto. Riferito anche come `alias` all'interno del codice
- **start_node** : nodo iniziale da cui parte la rotta
- **finish_node** : nodo di arrivo della rotta
- **label_oss** : stringa che indica una eventuale etichetta di osservabilità
- **label_rel** : stringa che indica una eventuale etichetta di rilevanza
- **set_label_rel** : un insieme contenente etichette di rilevanza. Necessario per l'algoritmo `EspressioniRegolari`
- **rif_node** : nodo di riferimento usato per l'algoritmo `EspressioniRegolari`

I due campi label sono di ridondanza ma semplificano il codice evitando di accedere necessariamente alla transizione/link di riferimento mantenendo comunque la correttezza del programma. Gli ultimi due attributi sono utili per la risoluzione delle espressioni regolari con le etichette di rilevanza per mezzo dell'algoritmo `EspressioniRegolari` in quanto permettono di gestire le stringhe in riferimento al rispettivo nodo finale (`rif_node`) e di gestire facilmente le operazioni di concatenazione tra stringhe adoperando

`set_label_rel`.

Le rotte e i nodi sono quindi gli oggetti fondamentali per generare gli **spazi comportamentali** (`Behavior_Space`), semplicemente formati da:

- **initial_node** : nodo iniziale dello spazio comportamentale. Generalmente composto dagli stati iniziali degli `FA`
- **list_nodes** : lista contenente i nodi appartenenti allo spazio comportamentale
- **list_routes** : lista contenente i collegamenti tra i vari nodi dello spazio comportamentale

Descrivono in modo sintetico lo spazio comportamentale e la loro struttura permette di essere sfruttati sia nel caso generale in cui lo spazio è l'insieme di tutti i comportamenti possibili dalla rete di `FA` sia nel caso relativo a una osservazione particolare data.

L'altro spazio che è possibile generare dal programma è il cosiddetto **spazio delle chiusure**. Iniziamo descrivendo cosa le **chiusure** sono.

Innanzitutto un **nodo chiusura** (`Closure_Node`) altro non è che un nodo adoperato nelle chiusure. È semplicemente formato da:

- **node** : un oggetto nodo che comporrà la chiusura
- **label** : una etichetta assegnata al nodo della chiusura qualora prevista

I `Closure_Nodes` aiutano a differenziare i nodi usati negli spazi comportamentali da quelli appartenenti alle chiusure con l'aggiunta di poter contenere una etichetta al loro interno, utile per la successiva diagnosi.

Vari nodi chiusura permettono di comporre una **chiusura** (`Closure`), composta da:

- **id** : stringa che identifica univocamente la chiusura
- **initial_node** : nodo iniziale della chiusura. Generalmente è univoco all'interno di uno spazio delle chiusure
- **list_nodes** : lista contenente i nodi appartenenti alla chiusura
- **list_routes** : lista contenente i collegamenti tra i vari nodi della chiusura

- **delta** : stringa contenente il `delta` della chiusura
- **list_diagnostic_output_route** : lista contenente le rotte in uscita dalla chisura verso nodi esterni alla chiusura stessa

La struttura di una chiusura è simile a quella di uno spazio comportamentale sebbene i loro scopi siano differenti.

L'insieme di più chisure (univoche rispetto al nodo iniziale) può andare a formare lo **spazio delle chiusure** (`Closure_Space`), necessario per il successivo lavoro di diagnostica. È formato banalmente da:

- **list_closures** : lista delle chiusure appartenenti allo spazio
- **list_routes** : lista contenente i collegamenti tra le varie chiusure dello spazio

L'ultimo oggetto da descrivere è il **diagnosticatore**. Questo altro non è che un riassunto dello spazio delle chiusure dove ogni chiusura viene semplificata come uno stato contenente il minimo quantitativo di informazione necessaria per la diagnosi. Pertanto introduciamo lo **stato diagnostico** (`State_Diagnostic`) contenente:

- **id** : stringa identificativa dello stato. Eredita l'id della chiusura relativa
- **alias** : stringa contenente il nome dello stato. Principalmente utile per stampare a video in modo leggibile il diagnosticatore
- **delta** : `delta` della chiusura qualora ne sia provvisto
- **list_routes** : lista delle rotte uscenti dalla chiusura

Le rotte adoperano la classe **Route_Diagnostic**, ereditata da `Route` usata per il contesto del diagnosticatore. Risiedono all'interno degli stati in quanto una singola rotta dello spazio comportamentale potrebbe ripresentarsi più volte nello spazio delle chiusure. In questo ogni stato ha tutti i riferimenti chiave della rispettiva chiusura.

Il **diagnosticatore** (`Diagnosticator_Space`) è infine formato da:

- **list_states** : lista contenente gli stati del diagnosticatore
- **initial_state** : stato iniziale del diagnosticatore

Tramite il `Diagnosticatore` è quindi possibile eseguire la diagnosi attraverso gli algoritmi che verranno mostrati e descritti nel prossimo paragrafo.

3.2 Algoritmi

Come già accennato, il nostro programma si occupa, data in input la definizione di una rete, di calcolare il relativo spazio comportamentale, quello relativo a un'osservazione lineare (che chiameremo spazio osservabile), la diagnosi relativa a quest'ultimo, la chiusura silenziosa e lo spazio delle chiusure, oltre al diagnosticatore e alla diagnosi lineare relativa a una osservazione lineare data. In sintesi:

- Generazione spazio comportamentale
- Generazione spazio comportamentale da osservazione lineare (spazio osservabile)
- Diagnosi relativa a una osservazione lineare
- Generazione spazio delle chiusure silenzione (e decorazione)
- Generazione diagnosticatore
- Diagnosi lineare relativa a una osservazione lineare

All'interno del GitHub del progetto è possibile accedere a tutti i pseudocodici e alle varie funzioni del programma scritto in python citate nelle seguenti sezioni.

3.2.1 Generazione spazio comportamentale

Il primo step dell'algoritmo consisteva nel creare lo spazio compartimentale partendo dalla definizione di una rete di automi. Lo pseudo codice è illustrato

di seguito.

Algorithm 1: Generazione Spazio Comportamentale

```
   Result: BEHAVIOR_SPACE
 1 new list_space_nodes as empty list
 2 new list_routes as empty list
 3 new tail_nodes as empty list
 4
 5 new initial_node as (Net.get_state_list_FAs,
     Net.get_content_list_LINKs)
 6 set initial_node.final as True
 7
 8 add initial_node to list_space_nodes
 9 add initial_node to tail_nodes
10 while tail_nodes is not empty do
11     get tail_nodes.firstElement as node_current
12     forall state_FA in node_current.state_list_FAs do
13         forall transaction in state_FA.output_transaction do
14             if transaction.event couples is present in
                node_current.content_list_LINKs then
15                 get transaction.finish_state as new_state
16                 get transaction.output_events as link_events_to_update
17
18                 new node_to_add as copy of node_current
19                 update state_FA in node.state_list_FAs with new_state
20                 update content_list_LINKs couples with
                    link_events_to_update
21
22                 new route_to_add as (node_current, transaction,
                    node_to_add)
23                 add route_to_add to list_routes
24                 if node_to_add is not in list_space_nodes then
25                     if all node_to_add.content_list_LINKs has empty
                        link.content then
26                         set node_to_add.final as True
27                     end
28                     add node_to_add to list_space_nodes
                        tail_nodes.push(node_to_add)
29                 end
30             end
31         end
32     end
33     tailNodi.pop()
34 end
```


Algorithm 2: Generazione Spazio Comportamentale

```
   Result: BEHAVIOR_SPACE
```

Nel nostro programma, il corrispettivo risulta essere la funzione **generate_behavior_space**. Inizialmente si creava il nodo iniziale contenente gli stati iniziali delle FA e il contenuto iniziale dei link (tutti vuoti). Successivamente si cicla su una coda, costituita inizialmente dal nodo iniziale:

1. Si prende il primo elemento del nodo e si controllano tutti gli stati attuali.

2. Per ogni stato analizzato si itera sulle transazioni in uscita e se l'evento della transazione è attivabile (dunque tale evento in ingresso è attualmente presente nel corrispettivo link del nodo analizzato) si crea il nuovo nodo che differenzia da quello analizzato in base alla transazione attivata:

    - avrà uno stato diverso nella lista degli stati in base allo stato di arrivo della transazione
    - avrà dei contenuti dei link diversi in base agli eventi in uscita della transazione

3. Nel codice, il compito di questo passaggio è affidato alla funzione in *extrafunction*, denominata **change_state**.

4. Se il nodo non è presente nello spazio lo si aggiunge alla spazio e alla coda dei nodi

5. Si aggiunge una nuova transizione tra il nodo appena creato e quello di partenza

6. Se il nodo presenta il contenuti dei suoi link vuoti lo si imposta come nodo finale di accettazione

7. Si rimuove il nodo analizzato

In seguito si esegue una potatura di tutti quei nodi che non portano a un nodo iniziale (con le rispettive transizioni) attraverso un algoritmo di **BFS** (nel nostro programma utilizziamo la funzione omonima presente nel file *extrafunction*, scritta da noi), che per ogni nodo finale dato, restituisce tutti i nodi che portano a quel nodo, alla fine della analizzi avremo dei nodi che non sono mai stati "utilizzati", dunque che sarà necessario rimuovere, perché potrebbero generare dei cicli o punti morti.

BFS altro non è l'algoritmo **Breadth-First Search** leggermente modificato per essere applicato al nostro caso, dove il grafo è il nostro spazio comportamentale, la direzione degli archi è invertita e il nodo iniziale è uno dei nodi finali. Se qualche nodo non viene toccato dopo l'esecuzione dell'algoritmo vuol dire che quel nodo non raggiunge il nodo finale e quindi verrà potato qualora non raggiunga nessun nodo finale presenti nello spazio comportamentale.

Infine, si rinomina i nodi rimanenti con un id crescente

3.2.2 Generazione Spazio Osservabile

Il secondo passaggio richiesto è quello di generare uno spazio comportamentale osservabile, partendo dalla definizione della rete e una lista di osservazioni,

cioè una lista di etichette di osservabilità.

Algorithm 3: Generazione Spazio Comportamentale

```
   Result: OSSERVABLE_SPACE
 1 new list_space_nodes as empty list
 2 new list_routes as empty list
 3 new tail_nodes as empty list
 4
 5 new initial_node as (Net.get_state_list_FAs,
     Net.get_content_list_LINKs)
 6 set initial_node.final as True
 7 add initial_node to list_space_nodes
 8 add initial_node to tail_nodes
 9 while tail_nodes is not empty do
10     get tail_nodes.firstElement as node_current
11     forall state_FA in node_current.state_list_FAs do
12         forall transaction in state_FA.output_transaction do
13             if transaction.event couples is present in
                node_current.content_list_LINKs, index_oss = 0 then
14                 if transaction.osservation_label exist and
                    transaction.osservation_label is not the
                    next(list_osservable) from node.index_oss then
15                     break
16                 end
17                 get transaction.finish_state as new_state
18                 get transaction.output_events as link_events_to_update
19                 new node_to_add as copy of node_current
20                 update state_FA in node.state_list_FAs with new_state
21                 update content_list_LINKs couples with
                    link_events_to_update
22                 new route_to_add as (node_current, transaction,
                    node_to_add)
23                 add route_to_add to list_routes
24                 if node_to_add is not in list_space_nodes then
25                     if all node_to_add.content_list_LINKs has empty
                        link.content and index_oss is equal
                        len(list_osservable) then
26                         set node_to_add.final as True
27                     end
28                     add node_to_add to list_space_nodes
                        tail_nodes.push(node_to_add)
29                 end
30             end
31         end
32     end
33     tailNodi.pop()
34 end
```

Rispetto all'algoritmo precedente, quello di generazione dello spazio comportamentale, si aggiungono unicamente due condizioni in più:

- **STEP 2** $\rightarrow$ Per ogni stato analizzato si itera sulle transazioni in uscita e se l'evento della transazione è attivabile (dunque tale evento in ingresso è attualmente presente nel corrispettivo link del nodo analizzato) e **se la sua etichetta di osservabilità è nulla oppure è il successivo elemento della lista di osservabilità**, si crea il nuovo nodo che differenzia da quello analizzato in base alla transazione attivata:
  - ...
  - **avrà come indice di osservazione incrementato di uno rispetto al nodo analizzato.**
  - **STEP 6** $\rightarrow$ Se il nodo presenta il contenuto dei suoi link come vuoti e l'**indice di osservazione pari alla lunghezza della lista di osservabilità**, lo si imposta come nodo finale di accettazione

Segue BFS e generazione indici.

Riassumendo, vengono aggiunti i nodi in modo da creare percorsi che portano a dei nodi finali nei quali sono presenti transazioni contenenti o etichette di osservabilità nulle oppure, se presenti, coerenti con l'ordine dato dalla lista di osservabilità

3.2.3 Diagnosi relativa a osservazione lineare

La terza fase consiste nel produrre la diagnosi dello spazio osservabile, che risulterà essere un'espressione regolare. Lo pseudo codice è illustrato di

seguito.

Algorithm 5: Diagnosi relativa a osservazione lineare

Result: listRoutes-Diagnosis

```
new listRoutes as copy of OssSpace.listRoutes
new listNodes as coppy of OssSpace.listNodes
new initial_node as copy of OssSpace.initial_node

if initial_node is present in route in listRoutes as finish_node then
    new new_initial_node
    set new_initial_node.isFinal as True

    new new_route as ¡new_initial_node, 'empty', initial_node¿

    add new_initial_node to listNodes
    add new_route to listRoutes

    update initial_node with new_initial_node
end
new final_node
set final_node.isFinal as True

new count_acept as count(node in listNodes where isFinal is True)
if count_acept > 1 or (count_acept is 1 and exist route in listRoutes
  as start_node is node in listNodes where isFinal is True and
  start_node is not finish_node then
    forall node in listNodes where isFinal is True do
        new new_route as ¡node, 'empty', final_node¿
        add new_route to listRoutes
    end
end
```

Algorithm 6: Diagnosi relativa a osservazione lineare

Result: listRoutes-Diagnosis

```
while listRoutes has more than 1 element do
    if exist sequence
      [< n, r_1, n_1 >, < n_1, r_2, n_2 >, ..., < n_(k − 1), r_k, n' >] of
      routes in listRoutes where all node n_i have only the sequence
      routes then
        remove sequence routes in listRoutes
        add new_route as < n, concat(r_i), n' > to listRoutes
    else if exist routes with same n as start_node and n' as
      finish_node then
        remove sequence routes in listRoutes
        add new_route as < n, concat(routes, '|'), n' > to listRoutes
    else
        take n from listNodes with n is not initial_node and n is not
```

La funzione responsabile di questo passaggio è quella presente nel file *extrafunction* con il nome **reg_expr**, la quale implementazione varia leggermente rispetto all'algoritmo proposto.

Per produrre in uscita una espressione regolare è necessario seguire questi steps:

1. Se il nodo iniziale ha transizioni in entrata, allora bisogna creare un nuovo nodo iniziale temporaneo che avrà solamente una transizione vuota in uscita verso il vecchio nodo iniziale

2. Si crea un nodo finale e tutti i nodi di accettazione presenti nello spazio verranno connessi a questo mediante una transizione vuota in uscita.

3. Poi, finchè sono presenti dei nodi che non sono quello iniziale e finale si eseguiranno questi step:

   (a) Si prende un nodo random tra quelli attualmenti presenti

   (b) Se presenta più transizioni in uscita verso gli stessi nodi, queste verranno raggruppate in un'unica transizione verso il rispettivo nodo che avrà come etichetta di rilevanza, le etichette di rilevanza delle transizioni d'origine unite tramite un **OR**

   (c) Se invece presenta solo una transizione in entrata e in uscita, allora il nodo verrà rimosso e le due transazioni concatenate in **AND**

   (d) Se invece le transizioni in entrata o in uscita sono molteplici, si combinano tutte le transizioni in entrata e in uscita, creando delle nuove transizioni che avranno come etichetta di rilevanza **AND** tra l'etichetta di quella in ingresso considerata e quella di uscita. Inoltre se il nodo presenta delle *autotransizioni*, tra le due etichette verrà frapposto in **AND** anche l'etichetta dell'*autotransizione* con il simbolo di molteplicità.

In conclusione, rimarrà un'unica transizione con un'etichetta di rilevanza che risulterà essere il raggruppamento di tutte le transizioni dello spazio.

3.2.4 Generazione dello spazio delle chiusure silenziose

La successiva fase consiste nel generare lo spazio delle chiusure partendo dallo spazio comportamentale. Lo pseudo codice è illustrato di seguito ed è diviso

in due parti, la prima per generare la chiusura silenziosa, l'altra per generare iterativamente lo spazio delle chiusure.

Algorithm 7: Generazione spazio delle chiusure

```
   Result: SILENCE_CLOSING
 1 new list_closing_nodes as empty list
 2 new list_closing_routes as empty list
 3 new list_output_routes as empty list
 4 new tail_nodes as empty list
 5
 6 if node not exist in list_space_nodes then
 7 |   ERROR
 8 end
 9
10 new initial_node as copy of node
11 add initial_node to list_closing_nodes
12 add initial_node to tail_nodes
13 while tail_nodes is not empty do
14 |   get tail_nodes.firstElement as node_current
15 |   forall route in osservable_space.list_routes where route.start_node
   |     is node_current do
16 |   |   if route.label_osservation not exist then
17 |   |   |   get route.finish_node as node_to_add add ruote to
   |   |   |     list_closing_routes
18 |   |   |   if node_to_add is not in list_closing_nodes then
19 |   |   |   |   add node_to_add to list_closing_nodes
   |   |   |   |     tail_nodes.push(node_to_add)
20 |   |   |   end
21 |   |   else
22 |   |   |   add route to list_output_routes
23 |   |   end
24 |   end
25 |   pop tail_node
26 end
```

Le funzioni che si occupano di questo passaggio sono due, ovvero **generator_silence_closure** che viene chiamata iterativamente da **generator_closures_space**. Per creare una chiusura silenziosa è necessario specificare un nodo di partenza e lo spazio comportamentale

1. Si crea il nodo iniziale della chiusura che corrisponderà a quello di partenza dato.
2. Lo si aggiunge a una coda
3. Finchè la coda ha elementi, si prende il primo elemento e lo si analizza
4. Per ogni transizione nello spazio comportamentale che ha quel nodo come nodo di partenza, se l'etichetta di osservabilità è nulla, allora si crea un nuovo nodo per la chiusura silenziosa
5. lo si aggiunge alla coda
6. si aggiunge la transizione alla lista delle transazioni della chiusura
7. Se invece l'etichetta di osservabilità non è nulla, allora non si aggiunge nessun nodo ma si aggiunge la transizione alla lista delle transizioni in uscita dalla chiusura stessa (che sarà diretta a un'altra chiusura)

Algorithm 8: Generazione spazio delle chiusure

```
   Result: List Closing
 1 new list_closing as empty list
 2 new tail_closing as empty list
 3
 4 new initialize_closing as SILENCE_CLOSING(osservable_space,
     osservable_space.initialize_node)
 5 add initialize_closing to list_closing
 6 push to tail_closing
 7 while tail_closing is not empty do
 8 |   get tail_closing.firstElement as closing_current
 9 |   forall output_route in closing_current.list_output_routes do
10 |   |   if output_route.finish_node is not in list_closing as
   |   |    closing.initialize_node then
11 |   |   |   new closing_temp as
   |   |   |    SILENCE_CLOSING(osservable_space,
   |   |   |    output_route.finish_node)
12 |   |   |
13 |   |   |   add closing_temp to list_closing
14 |   |   |   push closing_temp to tail_closing
15 |   |   end
16 |   end
17 |   pop tail_closing
18 end
```

Nel programma corrisponde ai seguenti passi:

1. Viene creata la prima chiusura silenziosa partendo dal nodo iniziale
2. Si aggiunge la chiusura a una coda e finchè la coda ha elementi si itera:
   (a) Si prende la prima chiusura dalla coda e per ogni transizione in uscita si recupera il nodo di arrivo
   (b) Si genera la chiusura silenziosa a partire dal nodo di arrivo e se non presente, lo si aggiunge alla coda.
   (c) Si crea una transizione tra le due chiusure con gli stessi riferimenti della transizione di uscita

Una volta concluso si avrà uno spazio delle chiusure silenziose.

Per ogni chiusura si trova la relativa decorazione per decorare lo spazio delle chiusure, i passaggi sono simili a quelli già visti con il calcolo della diagnosi, tranne che l'algoritmo produce in output una lista di espressioni, una per ogni stato di accettazione.

Algorithm 9: Decorazione di una chiusura

Result: listRoutes-Diagnosis

```
1  new listRoutes as copy of OssSpace.listRoutes
2  new listNodes as coppy of OssSpace.listNodes
3  new list_output_routes as copy of Closing.list_output_routes new
     initial_node as copy of OssSpace.initial_node
4
5  if initial_node is present in route in listRoutes as finish_node then
6  |   new new_initial_node
7  |   set new_initial_node.isFinal as True
8  |
9  |   new new_route as ¡new_initial_node, 'empty', initial_node¿
10 |
11 |   add new_initial_node to listNodes
12 |   add new_route to listRoutes
13 |
14 |   update initial_node with new_initial_node
15 end
16 new final_node
17 set final_node.isFinal as True
18 forall node in listde where isFinal is True do
19 |   new new_route as ¡node, 'empty', final_node¿ add new_route to
   |     listRoutes
20 end
```

E nel ciclo while

Algorithm 10: Decorazione di una chiusura

Result: listRoutes-Diagnosis

if *exist sequence* $[< n, r\_1, n\_1 >, < n\_1, r\_2, n\_2 >, ..., < n\_(k-1), r\_k, n' >]$ *of routes in listRoutes where all node n_i have only the sequence routes* **then**
  if *n' is not final_node and n_(k-1).isFinal is False* **then**
    remove sequence routes in listRoutes
    add new_route as $< n, concat(r\_i), n' >$ to listRoutes
  else
    remove sequence routes in listRoutes
    add new_route as $< n, concat(r\_i), n' >$ to listRoutes
    update new_route.subscript with value n_(k-1)
  end
else if *exist sequence [¡n,r_1, n_1¿, ¡n_1,r_2,n_2¿,...,¡n_(k-1), r_k.subscript is n_p, n'¿] of routes in listRoutes where all node n_i have only the sequence routes* **then**
  remove sequence routes in listRoutes
  add new_route as $< n, concat(r\_i), n' >$ to listRoutes update new_route.subscript with value n_p
else if *exist routes with same n as start_node and n' as finish_node* **then**
  remove sequence routes in listRoutes
  add new_route as $< n, concat(routes, '\,|'), n' >$ to listRoutes
else if *exist routes with same n as start_node and n' as finish_node with routes.subscript is n_p* **then**
  move sequence routes in listRoutes
  add new_route as $< n, concat(routes, '\,|'), n' >$ to listRoutes
  update new_route.subscript with value n_p

Algorithm 11: Decorazione di una chiusura

Result: listRoutes-Diagnosis

else
  take n from listNodes with n is not initial_node and n is not the final_node
  forall *routes as* $< n', r', n >$ *where finish_node is n* **do**
    forall *routes as* $< n, r'', n' >$ *where start_node is n* **do**
      if *n" is final_node and (n is Final or n exist in list_output_routes as start_node)* **then**
        if *exist route as* $< n, r, n >$ *where star_node and finish_node are n* **then**
          add new_route as $< n', r'(r*)r'', n'' >$ to listRoutes
          update new_route.subscript with value n
        else
          add new_route as $< n', r'r'', n'' >$ to listRoutes
          update new_route.subscript with value n

La prima fase è identica a quella della diagnosi, si raggruppano tutte le transazioni dei nodi che non sono iniziali o di accettazione.

Nella seconda fase invece si considerano i nodi di accettazione, escluso quello iniziale e quello finale e si creano tante transizioni parallele che vanno dal nodo iniziale a quello finale, quante il numero di nodi di accettazione. Per far questo, durante la semplificazione si aggiunge anche un riferimento al nodo che è stato eliminato. Si otterrà quindi una serie di transizioni parallele.

Per ogni transizione si recupererà il nodo di riferimento e si aggiungerà una label al nodo della chiusura con il valore della etichetta di rilevanza della transizione

3.2.5 Generazione del diagnosticatore

Il quinto step consiste nel generare il diagnosticatore partendo dallo spazio delle chiusure. Lo pseudo codice è illustrato di seguito.

Algorithm 12: Generazione del diagnosicatore

```
   Result: Diagnosticator
 1 create list_diagnosticated_closing as empty list
 2 forall closing in list_closing do
 3 |   new delta as empty string
 4 |   forall node in closing.list_closing_nodes where node.final is True
   |    do
 5 |   |   update delta adding node.label
 6 |   end
 7 |   forall output_route in closing.list_output_routes do
 8 |   |   update output_route.label with start_node.label
 9 |   end
10 |   new diagnostic_closing as < delta, closing >
11 |   add diagnostic_closing to list_diagnosticated_closing
12 end
```

La funzione responsabile di tale compito è **generator_diagnosticator**. Nel nostro codice, per ottimizzare il calcolo, abbiamo creato la variante integrata nella generazione del closure space, denominata molto semplicemente **generator_closures_space_and_diagnosticator**.

Il ragionamento che sta dietro alla generazione del diagnosticatore è molto semplice, per ogni chiusura dello spazio delle chiusure decorato, si crea uno stato, che avrà come delta il valore delle label degli stati di accettazione.

Ogni transizione in uscita diverrà anche una transizione tra i vari stati, ma che avrà come etichetta l'unione tra la sua etichetta e quella del nodo di partenza della chiusura (se presente).

3.2.6 Generazione diagnosi lineare

L'ultima fase dell'algoritmo consiste nel generare la diagnosi lineare partendo dal diagnosticatore e da un'osservazione lineare.

Lo pseudo codice è mostrato di seguito.

Algoritmo DiagnosiLineare(Γ,O)

$\chi \leftarrow \{(x_0, \varepsilon)\}$
for all evento o entro l'osservazione O (nell'ordine di comparizione) **do**
  $\chi_{\text{new}} \leftarrow \emptyset$
  for all $(x', \rho') \in \chi$ **do**
    for all arco $\langle x', (o, \rho), x'_2 \rangle$ entro il diagnosticatore Γ **do**
      $\rho_2 \leftarrow \rho'\rho$
      if $(x'_2, \rho'_2) \in \chi_{\text{new}}$ **then**
        sostituire (x'_2, ρ'_2) con $(x'_2, (\rho'_2|\rho_2))$ in χ_{new}
      else
        inserire (x'_2, ρ_2) in χ_{new}
      end if
    end for
  end for
  $\chi \leftarrow \chi_{\text{new}}$
end for
rimuovere da χ tutte le coppie (x_i, ρ_i) in cui x_i non è di accettazione
if $\chi = \{(x, \rho)\}$ **then**
  $R \leftarrow \rho\Delta(x)$
else if $\chi = \{(x_1, \rho_1), \ldots, (x_k, \rho_k)\}$ con $k > 1$ **then**
  $R \leftarrow (\rho_1(\Delta(x_1))) \mid \ldots \mid (\rho_k(\Delta(x_k)))$
end if
return R

85

La funzione responsabile è La funzione responsabile è **linear_diagnostic**. I passi per produrre la diagnosi lineare sono i seguenti:. I passi per produrre la diagnosi lineare sono i seguenti:

1. produrre la diagnosi lineare: Viene generato un insieme di valori, il cui primo valore è dato dallo stato iniziale del diagnosticatore e da un'etichetta nulla

2. Successivamente, per ogni etichetta dell'osservazione lineare si recuperano tutti i possibili stati raggiungibili secondo l'etichetta data partendo dal tutti gli stati già raggiunti

3. Salvati i riferimenti a tutti i passaggi eseguiti (stato-etichetta transizione), si esegue l'algoritmo con una nuova etichetta e i nuovi stati raggiunti nel passaggio precedente, concatenando le etichette delle transizioni su cui si transita

4. Terminato questo calcolo, si avrà un insieme di stati "attraversati" con varie etichette di rilevanza, si mantengono solo gli stati finali (dunque con un delta esistente) di accettazione. Eventualmente si raggruppano.

5. Si produce in output una lista di diagnosi concatenando l'etichetta di rilevanza con il delta del relativo stato

6. Infine si concatenano in OR le diagnosi ricavate

3.3 Funzioni di supporto

Le funzioni di supporto, sono state:

1. Nel programma principale denominato **project_function.py**:
   (a) **steps** che raggruppa i singoli passaggi, da *generate_behavior* a *generator_diagnosticator*
   (b) **benchmark** per il calcolo del benchmark riducendo all'osso i caricamenti e i salvataggi di file e stampe in stdoutput

2. Nel programma contenente solo funzioni di supporto denominato **extrafunction.py**:
   (a) **parse_arguments** che contiene la logica di esecuzione del parser degli argomenti da linea di comando per l'esecuzioni del programmo
   (b) **json_to_obj** per leggere la definizione della rete (in formato json) e creare il relativo oggetto python
   (c) **format_net_to_text** per stampare la definizione della rete
   (d) **clear_label** per concatenare le stringhe delle etichette in maniera corretta

4 Istanze di esempio

In questo capitolo mostreremo le istanze prese per verificare la correttezza del codice e per misurare le prestazioni del programma. Tali misurazioni verranno presentate nel prossimo capitolo riguardante appunto le prestazioni.

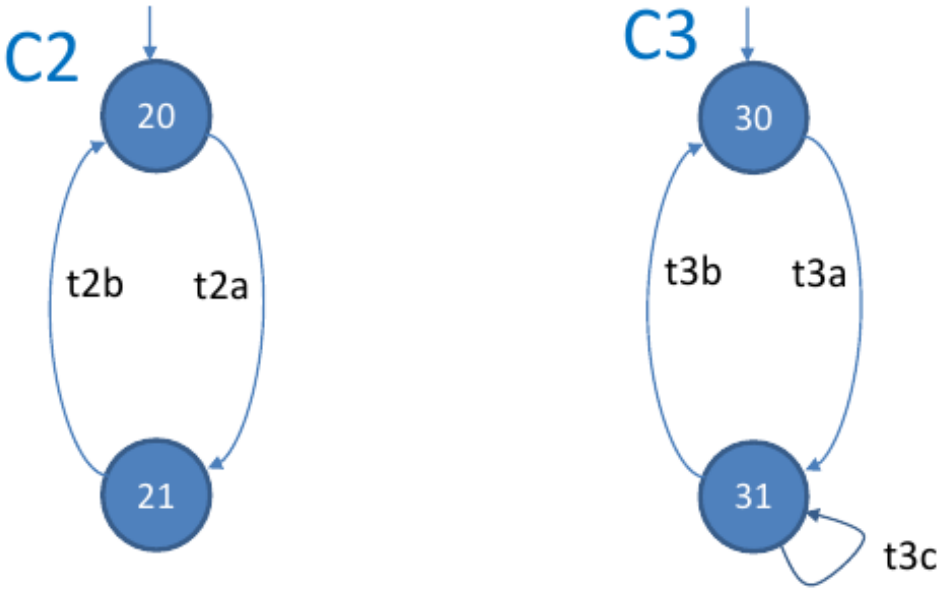
Abbiamo preso in esame quattro istanze di cui tre tratte dalla specifica del progetto fornitaci mentre la quarta è una istanza progettata da noi. Le prime due istanze prese dalla specifica sono servite in particolare per verificare che il codice fosse corretto e ha aiutato la fase di debug nei momenti in cui il programma non aveva un comportamento atteso. La terza istanza e quella da noi creata sono servite invece per misurare le prestazioni, sia in termini temporali, quanto tempo richiedesse per compiere una determinata operazione, sia in termini spaziali, ovvero effettivamente di quanta memoria il programma necessiti per la risoluzione del proprio compito.

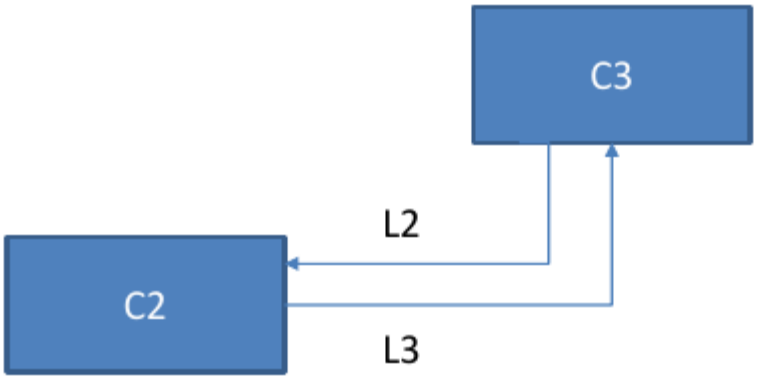
Qui sotto mostriamo brevemente la struttura delle reti di automi che formano le varie istanze. Queste sono le rappresentazioni visive delle istanze date in input al programma.

Da notare che durante la presentazione dei risultati per *Diagnosi Relativa all'osservazione lineare* si intende il risultato uscente dall'analisi dello spazio osservabile mentre per *Diagnosi Lineare sull'osservazione lineare* si intende il risultato dell'esecuzione dell'analisi del diagnosticatore.

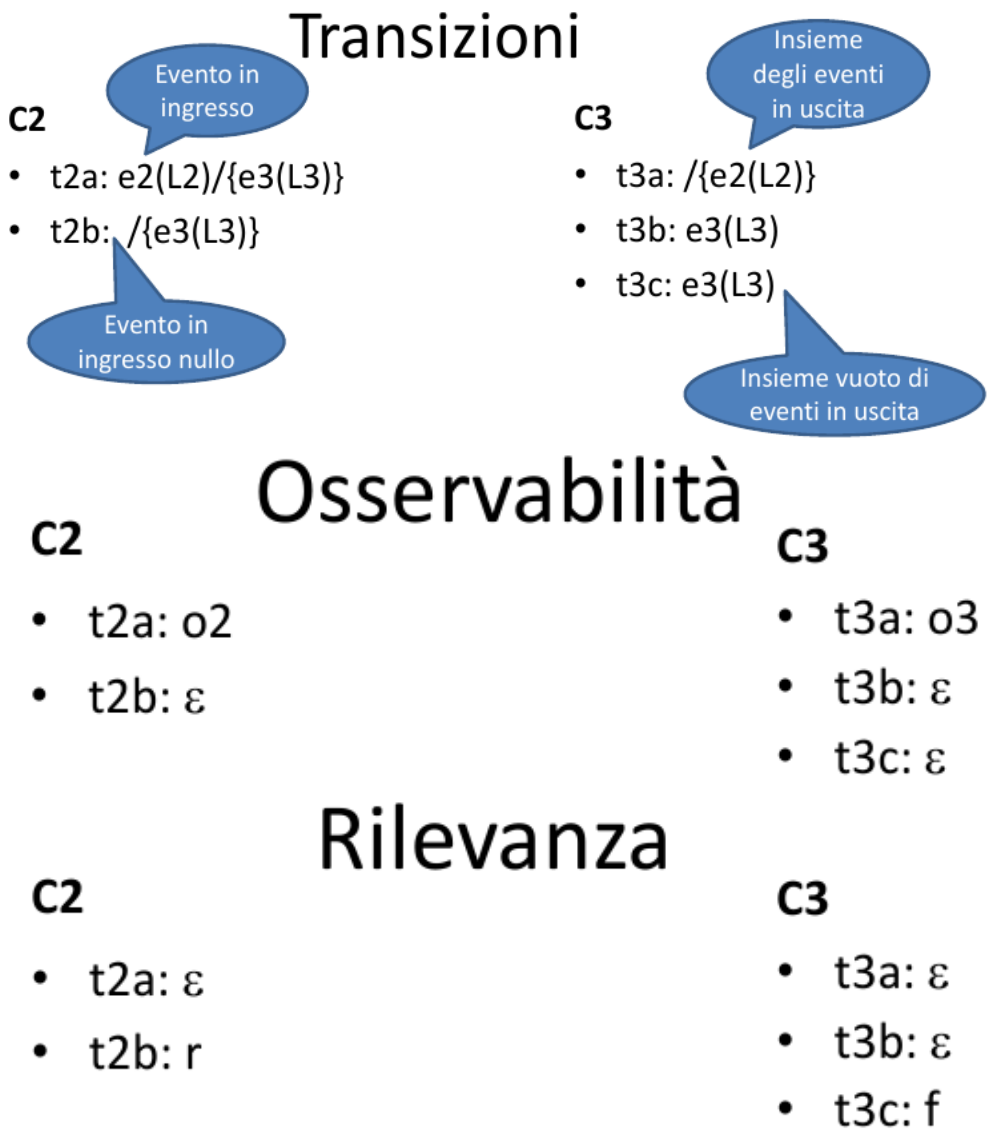
4.1 Istanza di Base

4.1.1 Rappresentazione grafica





4.1.2 Etichette ed Eventi



4.1.3 Risultati

Diagnosi Relativa all'osservazione lineare $[o3, o2]$

- $fr \| frf \| \epsilon \| f$

Diagnosi Lineare sull'osservazione lineare $[o3, o2, o3, o2]$

- $rf \| rffrf \| rffr \| rff$
- $fr \| frfrf \| frfr \| frf$

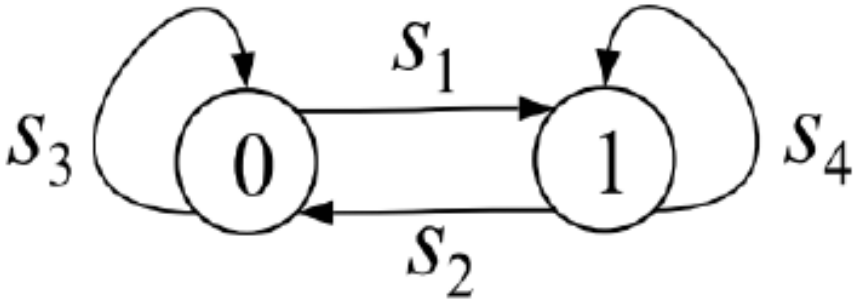
Il risultato completo dell'esecuzione è visionabile come immagine di seguito oltre che all'interno della reposity GitHub del progetto, nella cartella "Risultati Esempio" con il nome di "SummaryI1".

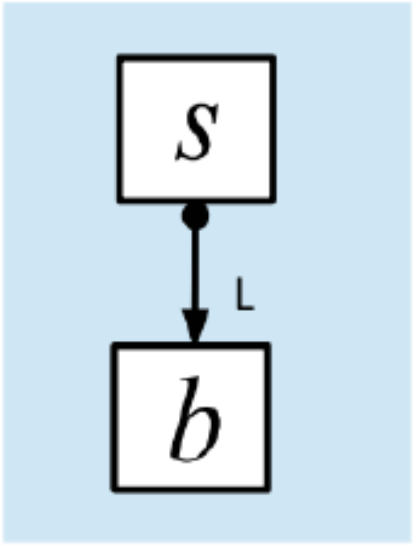
```
Generation Behavior Space
        13 Nodes
         ID: 0 ALIAS: 20:30|:
         ID: 1 ALIAS: 20:31|e2:
         ID: 2 ALIAS: 21:31|:e3
         ID: 3 ALIAS: 21:30|:
         ID: 4 ALIAS: 21:31|:
         ID: 5 ALIAS: 20:30|:e3
         ID: 6 ALIAS: 21:31|e2:
         ID: 7 ALIAS: 20:31|:e3
         ID: 8 ALIAS: 20:31|e2:e3
         ID: 9 ALIAS: 20:31|:
         ID: 10 ALIAS: 20:30|e2:
         ID: 11 ALIAS: 21:30|:e3
         ID: 12 ALIAS: 21:31|e2:e3
        16 Routes
        ROUTE: <0:t3a:1> with oss: o3 and rel:ε
        ROUTE: <1:t2a:2> with oss: o2 and rel:ε
        ROUTE: <2:t3b:3> with oss: ε and rel:ε
        ROUTE: <2:t3c:4> with oss: ε and rel:f
        ROUTE: <3:t2b:5> with oss: ε and rel:r
        ROUTE: <3:t3a:6> with oss: o3 and rel:ε
        ROUTE: <4:t2b:7> with oss: ε and rel:r
        ROUTE: <5:t3a:8> with oss: o3 and rel:ε
        ROUTE: <6:t2b:8> with oss: ε and rel:r
        ROUTE: <7:t3b:0> with oss: ε and rel:ε
        ROUTE: <7:t3c:9> with oss: ε and rel:f
        ROUTE: <8:t3b:10> with oss: ε and rel:ε
        ROUTE: <8:t3c:1> with oss: ε and rel:f
        ROUTE: <10:t2a:11> with oss: o2 and rel:ε
        ROUTE: <11:t3a:12> with oss: o3 and rel:ε
        ROUTE: <12:t3c:6> with oss: ε and rel:f
Generation Behavior Space from osservation
        8 Nodes
         ID: 0 ALIAS: 20:30|: and OSS.INDEX: 0
ID: 1 ALIAS: 20:31|e2: and OSS.INDEX: 1
         ID: 2 ALIAS: 21:31|:e3 and OSS.INDEX: 2
         ID: 3 ALIAS: 21:30|: and OSS.INDEX: 2
         ID: 4 ALIAS: 21:31|: and OSS.INDEX: 2
         ID: 5 ALIAS: 20:31|:e3 and OSS.INDEX: 2
         ID: 6 ALIAS: 20:30|: and OSS.INDEX: 2
         ID: 7 ALIAS: 20:31|: and OSS.INDEX: 2
        7 Routes
        ROUTE: <0:t3a:1> with oss: o3 and rel:ε
        ROUTE: <1:t2a:2> with oss: o2 and rel:ε
        ROUTE: <2:t3b:3> with oss: ε and rel:ε
        ROUTE: <2:t3c:4> with oss: ε and rel:f
        ROUTE: <4:t2b:5> with oss: ε and rel:r
        ROUTE: <5:t3b:6> with oss: ε and rel:ε
        ROUTE: <5:t3c:7> with oss: ε and rel:f
    Generation diagnosis in osservation space
        fr|frf|ε|f
    Generation Closing Space
          x0
                ID: 0
                ROUTEOUT: <0:ε:1> with rel: ε
                DELTA: ε
          x1
                ID: 1
                ROUTEOUT: <1:ε:2> with rel: ε
          x2
                ID: 2
                ID: 3
                ID: 4
                ID: 5
                ID: 7
                ID: 0
          ID: 9
                ROUTE: <2:t3b:3> with rel: ε
                ROUTE: <2:t3c:4> with rel: f
```

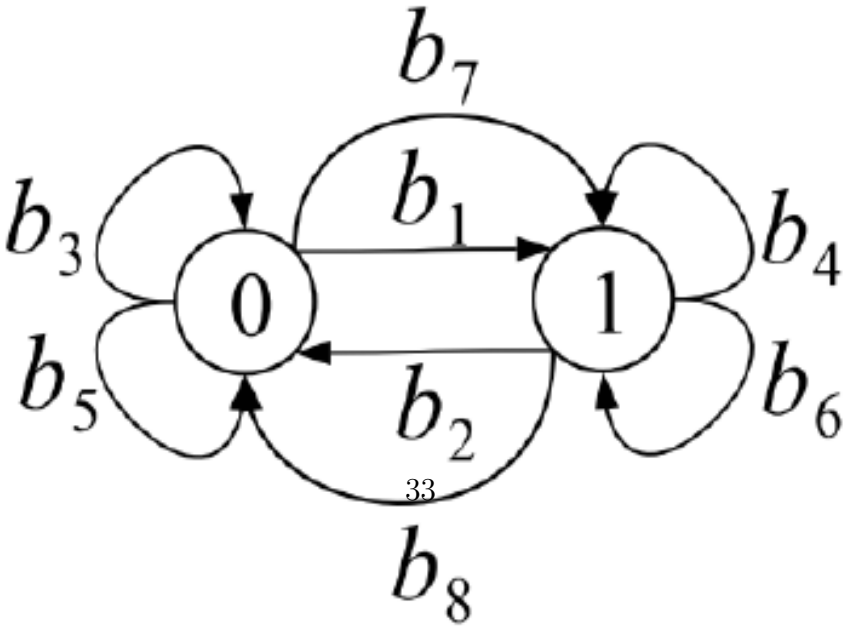
```
ROUTE: <3:t2b:5> with rel: r
        ROUTE: <4:t2b:7> with rel: r
        ROUTE: <7:t3b:0> with rel: ε
        ROUTE: <7:t3c:9> with rel: f
        ROUTEOUT: <0:fr:1> with rel: fr
        ROUTEOUT: <3:ε:6> with rel: ε
        ROUTEOUT: <5:r:8> with rel: r
        DELTA: ε|frf|f|fr
x3
        ID: 6
        ID: 8
        ID: 10
        ID: 1
        ROUTE: <6:t2b:8> with rel: r
        ROUTE: <8:t3b:10> with rel: ε
ROUTE: <8:t3c:1> with rel: f
        ROUTEOUT: <1:rf:2> with rel: rf
ROUTEOUT: <10:r:11> with rel: r
x4
                ID: 8
                ID: 10
                ID: 1
        ROUTE: <8:t3b:10> with rel: ε
                ROUTE: <8:t3c:1> with rel: f
                ROUTEOUT: <1:f:2> with rel: f
        ROUTEOUT: <10:ε:11> with rel: ε
        x5
                ID: 11
                ROUTEOUT: <11:ε:12> with rel: ε
        x6
                ID: 12
                ID: 6
        ID: 8
                ID: 10
                ID: 1
        ROUTE: <12:t3c:6> with rel: f
                ROUTE: <6:t2b:8> with rel: r
                ROUTE: <8:t3b:10> with rel: ε
                ROUTE: <8:t3c:1> with rel: f
                ROUTEOUT: <1:frf:2> with rel: frf
                ROUTEOUT: <10:fr:11> with rel: fr
 Generation Diagnosticator
        STATE: 0
        ROUTE OUT: <0:ε:1> with rel: fr
                DELTA: ε
        STATE: 1
                ROUTE OUT: <1:ε:2> with rel: fr
        STATE: 2
                ROUTE OUT: <3:ε:6> with rel: fr
                ROUTE OUT: <5:r:8> with rel: fr
        ROUTE OUT: <0:fr:1> with rel: fr
                DELTA: ε|frf|f|fr
STATE: 3
        ROUTE OUT: <10:r:11> with rel: fr
                        ROUTE OUT: <1:rf:2> with rel: fr
                STATE: 4
                        ROUTE OUT: <10:ε:11> with rel: fr
                ROUTE OUT: <1:f:2> with rel: fr
                STATE: 5
                        ROUTE OUT: <11:ε:12> with rel: fr
        STATE: 6
                ROUTE OUT: <10:fr:11> with rel: fr
                        ROUTE OUT: <1:frf:2> with rel: fr
  Generation linear diagnostic
       rf|rfrf|rfr|rff
              frfrf|frf|frfr|fr
  --- 0.007228374481201172 seconds ---
```

4.2 Istanza di Test

4.2.1 Rappresentazione grafica

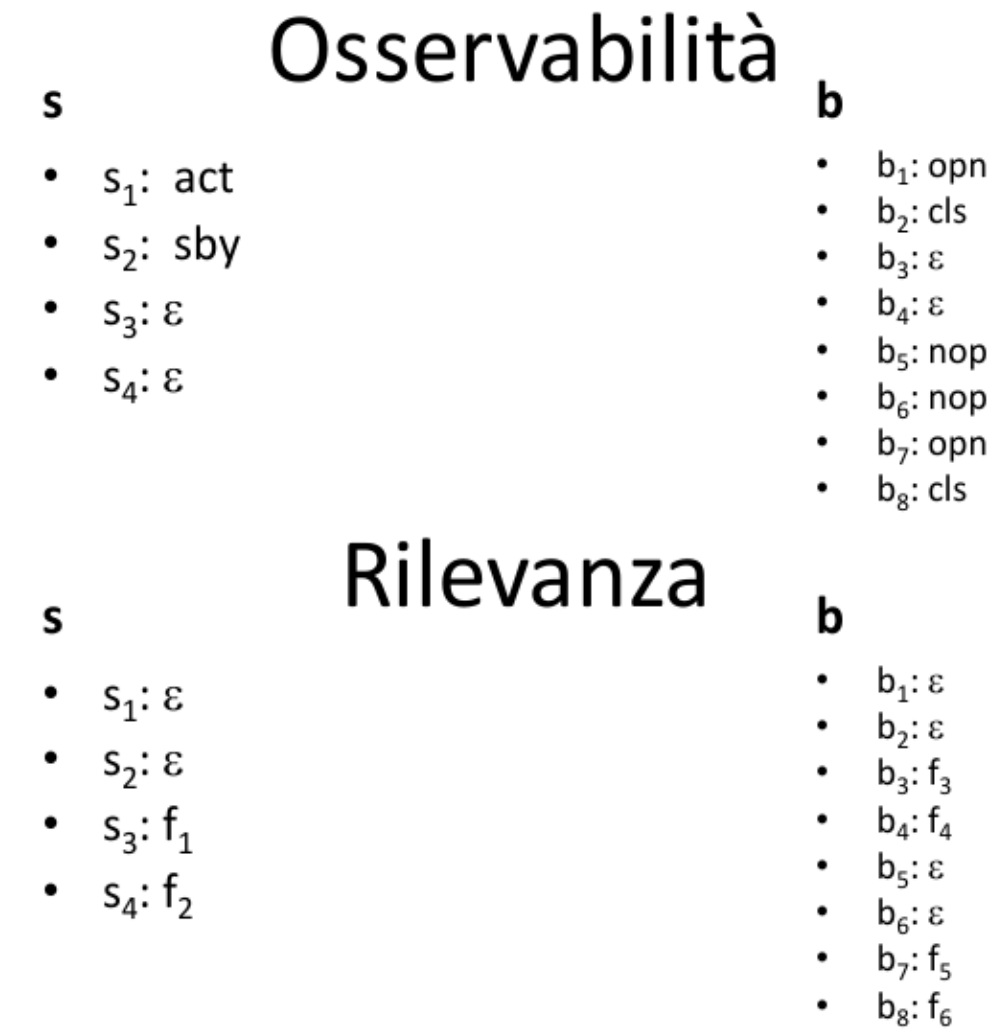


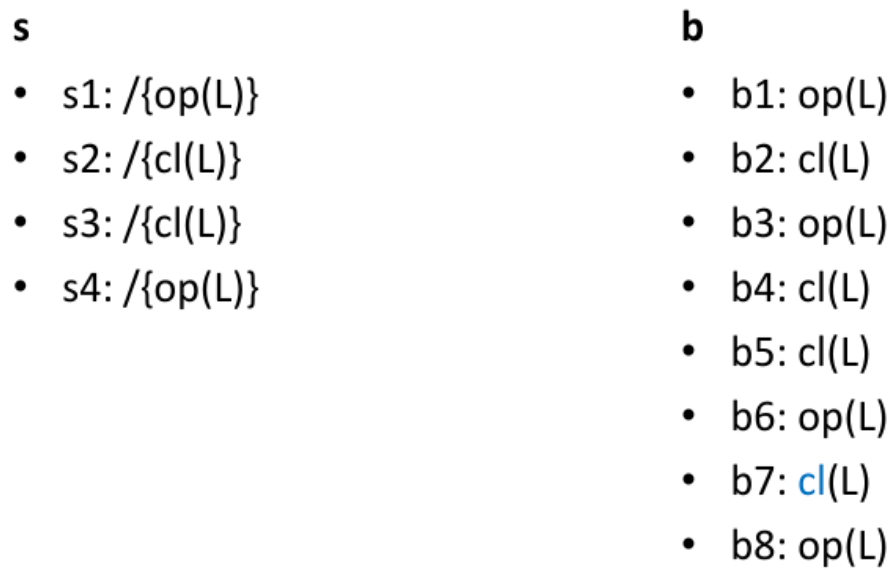




87

4.2.2 Etichette ed Eventi





4.2.3 Risultati

Diagnosi Relativa all'osservazione lineare $[act, sby, nop]$

- $(f3f2)^*f3$

Diagnosi Lineare sull'osservazione lineare $[act, sby, nop]$

- $(f3f2)^*f3$

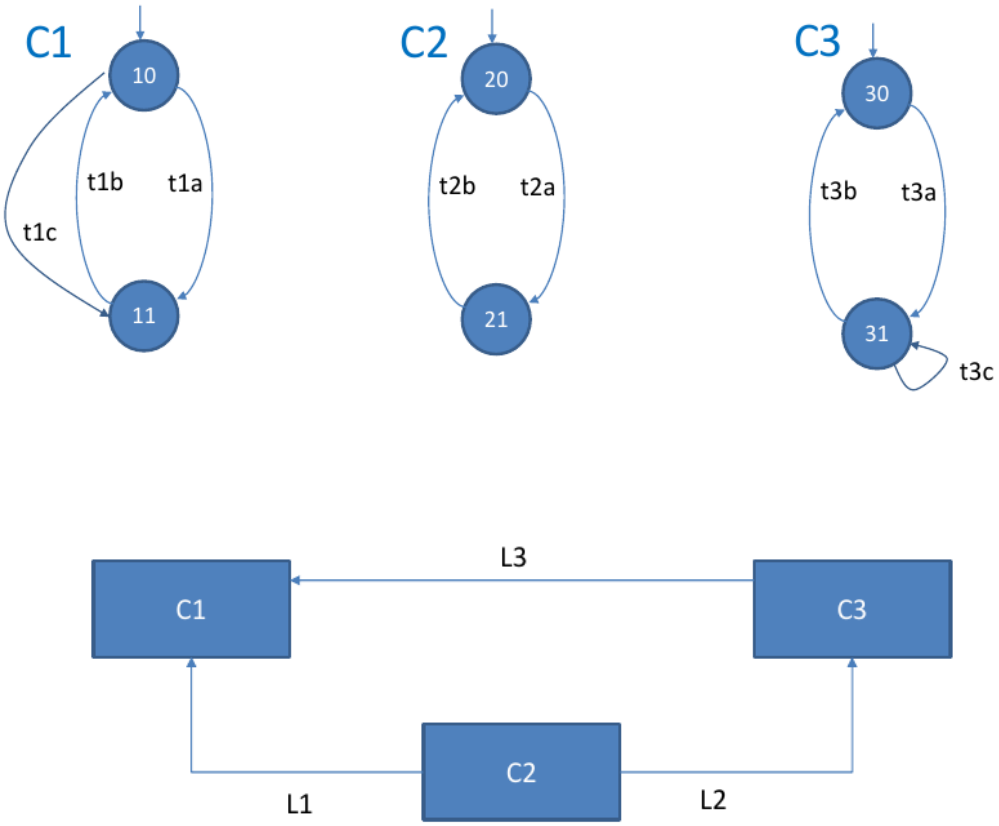
Il risultato completo dell'esecuzione è visionabile come immagine di seguito oltre che all'interno della reposity GitHub del progetto, nella cartella "Risultati Esempi" con il nome di "SummaryI2".

```
Generation Behavior Space
8 Nodes
ID: 0 ALIAS: 0:0|
ID: 1 ALIAS: 1:0|op
ID: 2 ALIAS: 0:0|cl
ID: 3 ALIAS: 1:1|
ID: 4 ALIAS: 1:0|
ID: 5 ALIAS: 0:1|
ID: 6 ALIAS: 0:1|cl
ID: 7 ALIAS: 1:1|op
16 Routes
ROUTE: <0:s1:1> with oss: act and rel:ε
ROUTE: <0:s3:2> with oss: ε and rel:f1
ROUTE: <1:b1:3> with oss: opn and rel:ε
ROUTE: <1:b3:4> with oss: ε and rel:f3
ROUTE: <2:b5:0> with oss: nop and rel:ε
ROUTE: <2:b7:5> with oss: opn and rel:f5
ROUTE: <3:s2:6> with oss: sby and rel:ε
ROUTE: <3:s4:7> with oss: ε and rel:f2
ROUTE: <4:s2:2> with oss: sby and rel:ε
ROUTE: <4:s4:1> with oss: ε and rel:f2
ROUTE: <5:s1:7> with oss: act and rel:ε
ROUTE: <5:s3:6> with oss: ε and rel:f1
ROUTE: <6:b2:0> with oss: cls and rel:ε
ROUTE: <6:b4:5> with oss: ε and rel:f4
ROUTE: <7:b6:3> with oss: nop and rel:ε
ROUTE: <7:b8:4> with oss: cls and rel:f6
Generation Behavior Space from osservation
5 Nodes
ID: 0 ALIAS: 0:0| and OSS.INDEX: 0
ID: 1 ALIAS: 1:0|op and OSS.INDEX: 1
ID: 2 ALIAS: 1:0| and OSS.INDEX: 1
ID: 3 ALIAS: 0:0|cl and OSS.INDEX: 2
ID: 4 ALIAS: 0:0| and OSS.INDEX: 3
5 Routes
ROUTE: <0:s1:1> with oss: act and rel:ε
ROUTE: <1:b3:2> with oss: ε and rel:f3
ROUTE: <2:s2:3> with oss: sby and rel:ε
ROUTE: <2:s4:1> with oss: ε and rel:f2
ROUTE: <3:b5:4> with oss: nop and rel:ε
Generation diagnosis in osservation space
(f3f2)*f3
Generation Closing Space
x0
ID: 0
ID: 2
ROUTE: <0:s3:2> with rel: f1
ROUTEOUT: <0:ε:1> with rel: ε
ROUTEOUT: <2:f1:0> with rel: f1
ROUTEOUT: <2:f5f1:5> with rel: f5f1
DELTA: ε
x1
ID: 1
ID: 4
ROUTE: <1:b3:4> with rel: f3
ROUTE: <4:s4:1> with rel: f2
ROUTEOUT: <1:(f3f2)*:3> with rel: (f3f2)*
ROUTEOUT: <4:(f3f2)*f3:2> with rel: (f3f2)*f3
DELTA: (f3f2)*f3
x2
ID: 5
ID: 6
ROUTE: <5:s3:6> with rel: f1
ROUTE: <6:b4:5> with rel: f4
ROUTEOUT: <5:(f1f4)*:7> with rel: (f1f4)*
ROUTEOUT: <6:(f1f4)*f1:0> with rel: (f1f4)*f1
DELTA: (f1f4)*
x3
```

```
ID: 3
ID: 7
ROUTE: <3:s4:7> with rel: f2
ROUTEOUT: <3:ε:6> with rel: ε
ROUTEOUT: <7:f2:3> with rel: f2
ROUTEOUT: <7:f6f2:4> with rel: f6f2
DELTA: ε
x4
ID: 2
ROUTEOUT: <2:ε:0> with rel: ε
ROUTEOUT: <2:f5:5> with rel: f5
x5
ID: 7
ROUTEOUT: <7:ε:3> with rel: ε
ROUTEOUT: <7:f6:4> with rel: f6
x6
ID: 6
ID: 5
ROUTE: <6:b4:5> with rel: f4
ROUTE: <5:s3:6> with rel: f1
ROUTEOUT: <5:f4(f1f4)*:7> with rel: f4(f1f4)*
ROUTEOUT: <6:f4(f1f4)*f1:0> with rel: f4(f1f4)*f1
DELTA: f4(f1f4)*
x7
ID: 4
ID: 1
ROUTE: <4:s4:1> with rel: f2
ROUTE: <1:b3:4> with rel: f3
ROUTEOUT: <4:(f2f3)*:2> with rel: (f2f3)*
ROUTEOUT: <1:(f2f3)*f2:3> with rel: (f2f3)*f2
DELTA: (f2f3)*
Generation Diagnosticator
STATE: 0
ROUTE OUT: <0:ε:1> with rel: (f2f3)*f2
ROUTE OUT: <2:f1:0> with rel: (f2f3)*f2
ROUTE OUT: <2:f5f1:5> with rel: (f2f3)*f2
DELTA: ε
STATE: 1
ROUTE OUT: <1:(f3f2)*:3> with rel: (f2f3)*f2
ROUTE OUT: <4:(f3f2)*f3:2> with rel: (f2f3)*f2
DELTA: (f3f2)*f3
STATE: 2
ROUTE OUT: <5:(f1f4)*:7> with rel: (f2f3)*f2
ROUTE OUT: <6:(f1f4)*f1:0> with rel: (f2f3)*f2
DELTA: (f1f4)*
STATE: 3
ROUTE OUT: <3:ε:6> with rel: (f2f3)*f2
ROUTE OUT: <7:f2:3> with rel: (f2f3)*f2
ROUTE OUT: <7:f6f2:4> with rel: (f2f3)*f2
DELTA: ε
STATE: 4
ROUTE OUT: <2:ε:0> with rel: (f2f3)*f2
ROUTE OUT: <2:f5:5> with rel: (f2f3)*f2
STATE: 5
ROUTE OUT: <7:ε:3> with rel: (f2f3)*f2
ROUTE OUT: <7:f6:4> with rel: (f2f3)*f2
STATE: 6
ROUTE OUT: <6:f4(f1f4)*f1:0> with rel: (f2f3)*f2
ROUTE OUT: <5:f4(f1f4)*:7> with rel: (f2f3)*f2
DELTA: f4(f1f4)*
STATE: 7
ROUTE OUT: <4:(f2f3)*:2> with rel: (f2f3)*f2
ROUTE OUT: <1:(f2f3)*f2:3> with rel: (f2f3)*f2
DELTA: (f2f3)*
Generation linear diagnostic
(f3f2)*f3
```

4.3 Istanza di Benchmark

4.3.1 Rappresentazione grafica



4.3.2 Etichette ed Eventi

C1

- t1a: e1(L1)
- t1b: e2(L3)
- t1c:

C2

- t2a: /{e1(L1),e3(L2)}
- t2b: /{e1(L1)}

C3

- t3a: /{e2(L3)}
- t3b: e3(L2)
- t3c: e3(L2)

Evento in ingresso nullo e insieme vuoto di eventi in uscita

C1

- t1a: ε
- t1b: ε
- t1c: ε

C2

- t2a: o1
- t2b: o2

C3

- t3a: ε
- t3b: ε
- t3c: ε

Rilevanza

C1

- t1a: ε
- t1b: ε
- t1c: f1

C2

- t2a: ε
- t2b: ε

C3

- t3a: ε
- t3b: ε
- t3c: f3

4.3.3 Risultati

Diagnosi Relativa all'osservazione lineare $[o1, o2]$

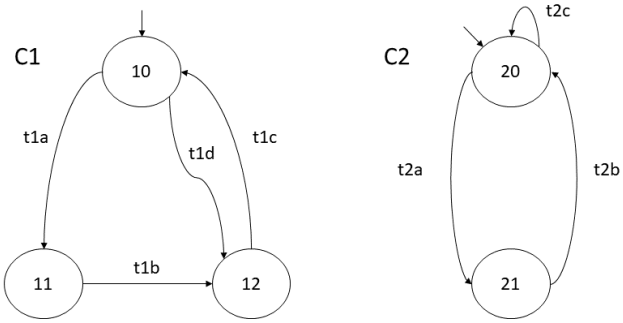
- $f3 \| f1 \| f1 f1 \| \epsilon$

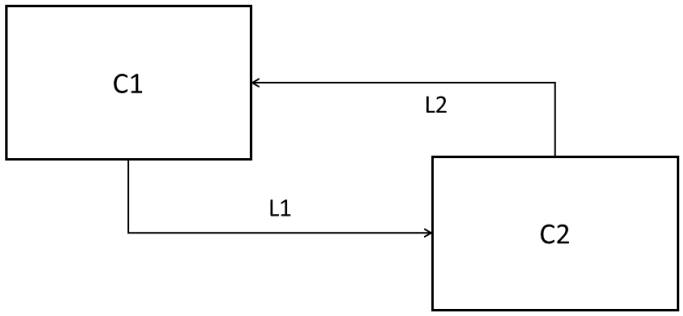
Diagnosi Lineare sull'osservazione lineare $[o1, o2]$

- $f3 \| f1 \| \epsilon$
- $f1 \| \epsilon$
- $f1$
- $f3 \| f1$

4.4 Istanza creata da noi

4.4.1 Rappresentazione grafica





4.4.2 Etichette ed Eventi

Etichette di osservazione:

- t2b → **r**
- t2c → **f**

Etichette di rilevanza:

- t1a → **o1**
- t1d → **o2**

Eventi:

- t1b → __**:L1(e2)**
- t1d → **L2(e1):**__
- t2b → __**:L2(e1)**
- t2c → **L1(e2):**__

4.4.3 Risultati

Diagnosi Relativa all'osservazione lineare $[o1, o2]$

- $rfr \| r \| rf \| fr \| frr \| \epsilon$

Diagnosi Lineare sull'osservazione lineare $[o1, o2, o1, o2]$

- $rfrf \| frrf$
- $rfrf \| rrff$
- $rffr \| rfrf \| rrff$

4.5 Dimensioni istanze

Di seguito inseriamo alcune tabelle riassuntive che raccolgono le dimensioni dei vari spazi generati dal programma. Tali valori possono essere utili anche nel prossimo capitolo dove andiamo a commentare le prestazioni del software.

	I1	I2	Icustom	**Ibenchmark**
Nodi	13	8	23	41
Rotte	16	16	46	85
Totale elementi	29	24	69	126

Tabella 1: Tabella relativa al numero di elementi generati dagli spazi comportamentali delle istanze presentate

	I1	I2	Icustom	**Ibenchmark**
Chiusure	7	8	11	17
Nodi	22	14	136	177
Rotte	15	10	185	256
Rotte di output	12	18	88	64
Totale elementi	56	50	420	514

Tabella 2: Tabella relativa al numero di elementi generati dagli spazi delle chiusure delle istanze presentate

5 Analisi delle Prestazioni

5.1 Complessità Temporale

L'analisi temporale degli algoritmi e di tutte le funzioni presenti è stata eseguita manualmente, inoltre per gli esempi che verranno illustrati nella sezione seguente, verrà proposto il calcolo del tempo effettivo di esecuzione. Nella lista sottostante verranno quindi inseriti sia i calcoli di complessità intermedi risultanti dai vari cicli che compongono le funzioni (per lo meno i cicli più importanti) e successivamente l'effettiva complessità risultante.

L'analisi è stata eseguita sull'effettivo codice che in ogni caso rispecchia lo pseudocodice commentato nel terzo capitolo.

Un file, denominato **time complex analysis** è presente nel GitHub del progetto e contiene tutte le informazioni qui contenute.

Definizione dei simboli per le variabili utilizzate per esprimere la complessità temporale:

- list_FA: **F**
- list_link: **L** $\rightarrow$ $\mathbf{F}^2$
- list_nodes: **V**
- list_routes: **E** $\rightarrow$ $\mathbf{V}^2$
- list_states: **S**
- list_transitions: **T**

Lista delle funzioni principali dell'algoritmo:

- **generate_behavior_space**
  - O(V)*(O(F)*O(T) *(change_state + O(V))+O(V))
  - complessità risultante: O($V^2 * F * T$)
- **generate_behavior_space_from_osservation**
  - O(V)*(O(F)*O(T)*(change_state + O(V))+O(V))
  - complessità risultante: O($V^2 * F * T$)
- **diagnosis_space** $\rightarrow$ O(V^2)

- **generator_silence_closure** → O(V)*(O(E)*O(V)+O(V))
  - complessità risultante: O(V^4)
- **generator_closures_space_and_diagnosticator** → la somma di
  - O(V)*O(E)*(O(V)+O(generate silence closure))
  - O(V)*(O(V)+O(E)+fn.reg_expr_closing+O(E)*O(E))
  - complessità risultante: O(V^7)
- **generator_closures_space** → la somma di
  - O(V)*O(E)*(O(V)+O(generate silence closure))
  - O(V)*(O(V)+O(E)+reg_expr_closing+O(E)*O(E))
  - complessità risultante: O(V^7)
- **generator_diagnosticator** → la somma di
  - O(V)*O(E)*O(V)
  - O(V)
  - complessità risultante: O(V^4)

Funzioni di supporto all'algoritmo:

- **BFS** → O(V+E)
- **change_state** → O(T+F^2)
- **reg_expr_closing** → O(V^4)

La progettazione del programma ha avuto come obiettivo primario quello di avere una complessità temporale polinomiale, evitando di avere così complessità esponenziali o superiori al polinomiale. Tale obiettivo è stato rispettato ottenendo un programma con complessità polinomiale rispetto al numero di nodi presente nello spazio comportamentale.

Da notare come le complessità sono state parametrizzate rispetto al numero di nodi e non rispetto al numero di automi o stati o transizioni degli automi. Questo poiché anche con un numero simile di stati è possibile avere un numero di nodi dello spazio comportamentale molto differente e in particolar modo al numero dei nodi del successivo spazio delle chiusure, dove le

analisi hanno mostrato avere in quest'ultimo un numero di nodi più elevato. Vedasi ad esempio l'istanza creata da noi mostrata nel capitolo precedente. Avendo solo due automi con un totale di soli cinque stati si è arrivati ad avere uno spazio comportamentale comunque molto grande e soprattutto ad avere una dimensione dello spazio delle chiusure pressoché paragonabile all'istanza di benchmark il quale ha un numero di stati e `FA` superiori alla nostra istanza.

Perciò essendo le successive funzioni operanti sullo spazio comportamentale, e quindi dipendenti dalla loro dimensione, e non sulla rete di `FA` è dal nostro punto di vista preferibile giudicare le prestazioni rispetto al numero di nodi.

5.2 Benchmark

5.2.1 Complessità Temporale

Nelle seguenti tabelle mostriamo le misurazioni temporali effettuate con il nostro programma. Ogni valore è misurato in millisecondi se non viene specificato altrimenti. I valori risultanti sono una media di numerose esecuzioni per ottenere un valore di benchmark più affidabile e veritiero (all'incirca una decina di esecuzioni per istanza).

Di seguito è mostrata la tabella che mostra il calcolo sperimentale della complessità temporale delle istanze mostrate nella sezione precedente; **I1** corrisponde all'istanza "Istanza Base", **I2** a "Istanze di Test", **Icustom** a "Istanza creata da noi" e infine **Ibenchmark** a "Istanza per Benchmark".

	I1	I2	Icustom	**Ibenchmark**
spazio comp.	0,21	0,17	0,77	1,56
spazio osservabile	0,16	0,16	0,82	1,13
Diagnosi	0,23	0,24	2,36	3,75
spazio delle chiusure	1,40	1,37	13,33	15,49
diagnosticatore	0,04	0,07	0,40	0,51
Diagnosi lineare	0,08	0,05	1,00	0,30
TOTALE (s)	$2,3 \times 10^{-3}$	$2,1 \times 10^{-3}$	$1,9 \times 10^{-2}$	$2,3 \times 10^{-2}$

Tabella 3: Tabella relativa alle calcolo della complessità temporale sperimentale delle istanze presentate

Come mostrato nella tabella le istanze semplici, usate da noi per i test, hanno un tempo di esecuzione molto simile tra loro, dato dal fatto che con

la loro semplicità il numero di nodi del loro spazio comportamentale non è elevato. Stesso discorso può essere applicato per le due istanze usate come effettivo benchmark, ovvero l'istanza di benchmark e la nostra istanza. Queste due istanze prevedono un numero di nodi di gran lunga superiore e per forza di cose questo implica un aumento nei tempi di esecuzione confrontato ai tempi delle istanze di test. Tra loro le due istanze hanno comunque valori paragonabili.

In nessun caso si è arrivato ad avere tempi delle varie funzioni tra varie istanze discrepanti, ad esempio non risulta che in una qualche istanza una funzione particolare impiegasse un quantitativo di tempo nettamente superiore rispetto a quello di una istanza paragonabile. I tempi misurati risultano quindi in linea ed è stato rispettato il comportamento atteso.

Tra le funzioni notiamo inoltre come la funzione che richiede un maggior quantitativo di tempo risulta **generator_closures_space** che in effetti è la funzione con la peggior complessità temporale ma allo stesso tempo è la funzione che necessità di un maggior numero di operazioni di calcolo per ottenere il risultato desiderato e opera con strutture nettamente più grandi.

5.2.2 Complessità Spaziale

Per quanto riguarda invece la complessità spaziale, abbiamo optato per calcolarla, sperimentalmente, attraverso una libreria di python, memory_profiler, che restituisce in output la memoria occupata dallo script.

Il nostro script occupa di base **43.95 MB**, importando le librerie e caricando la definizione della rete. Lanciando le varie istanze si ottengono i seguenti consumi di memoria:

- Istanza di Base → **+0.07 MB**
- Istanza di Test → **+0.05 MB**
- Istanza creata da noi → **+0.57 MB**
- Istanza di Benchmark → **+0.69 MB**

Così come la complessità temporale, anche la complessità spaziale è dipendente dal numero di nodi che compongono lo spazio comportamentale e lo spazio delle chiusure di ciascuna istanza. Ovviamente qualora una istanza avesse delle dimensioni superiori rispetto a un'altra è abbastanza normale che la quantità di memoria necessaria sia maggiore.

Una possibile misura per migliorare la complessità spaziale sarebbe stata quella di progettare una specifica che vada a salvare i dati in memoria in modo da risparmiare molto più spazio. Tale approccio non è stato intrapreso perché così facendo si sarebbe dovuto implementare delle operazioni custom che andavano a interpretare correttamente le nostre strutture ed eseguire operazioni in modo da mantenere la coerenza con i risultati attesi. Questo innanzitutto avrebbe inutilmente complicato il progetto, avrebbe aumentato la possibilità di incagliarsi in errori e poteva non garantire un miglioramento nella complessità temporale ma al contrario ottenere un peggioramento. Affidandoci a un linguaggio di programmazione e a delle librerie consolidate, a strutture per la memorizzazione standard si è così ottenuto stabilità nel codice e una maggior semplicità a gestire in modo efficace la memoria a disposizione, sfruttando appunto le caratteristiche del linguaggio python e dei file json.

Per questo motivo si è preferito concentrarsi più sulla complessità temporale che non su quella spaziale. Ciononostante i risultati ottenuti sono comunque soddisfacenti con l'attuale implementazione del programma.