Label Transition System Generator

Elaborato di progetto per Sistemi Distribuiti

`https://gitlab.com/pika-lab/courses/ds/projects/ds-project-aa1718-lucchi-volonnino`

Chiara Volonnino[1] and Giulia Lucchi[2]

Email ids: [1]chiara.volonnino@studio.unibo.it, [2]giulia.lucchi7@studio.unibo.it

A.A. 2017/2018

Indice

Capitolo 1

Introduzione

L'elaborato tratta uno degli argomenti visti a lezione nell'anno accademico 2018/2019: Process Algebra. La scelta di questo elaborato è dovuta dalla voglia di approfondire questo ambito e tutto quello che ne concerne.
Vogliamo, quindi, realizzare un sistema capace di produrre un intero grafo degli stati, partendo dalla descrizione di un sistema concorrente/distribuito mediante l' algebra dei processi.

Capitolo 2

Requisiti

Partendo dal goal citato sopra, abbiamo individuato i nostri requisiti. Quest'ultimi li abbiamo divisi in due grandi categorie:

- requisiti funzionali
- requisiti non funzionali

2.1 Requisiti Funzionali

Abbiamo individuato i seguenti requisiti funzionali:

- Il sistema deve accettare come input una qualsivoglia algebra dei processi.
- Il sistema deve produrre come output una rappresentazione grafica del "Label Transition System".
- Gli operatori da definire sono:
  - **Operatore di sequenza**: non soddisfa la proprietà commutativa;
  - **Composizione parallela**: può essere commutativa o non commutativa, ma nel sistema sarà definito come operatore che soddisfa la proprietà commutativa;
  - **Scelta non deterministica**: soddisfa la proprietà commutativa.
- L'utente deve avere la possibilità di inserire tramite interfaccia grafica l'input richiesto.
- L'utente deve avere la possibilità di visualizzare l'output graficamente.
- L'utente deve avere la possibilità di inserire una nuova regola.
- L'utente deve avere la possibilità di visualizzare le regole già esistenti e le regole da lui inserite.
- Le nuove regole devono essere valide solo per quella computazione.

2.2 Requisiti Non Funzionali

Abbiamo individuato i seguenti requisiti non funzionali:

- Il sistema deve essere usabile e intuitivo: l'utente deve poter utilizzare il sistema in modo facile e chiaro.
- Il sistema deve essere efficace ed efficiente.

Capitolo 3

Design

3.1 Architettura

Da subito abbiamo individuato due entità che compongono il sistema [Figura 3.1]:

- **Utente**: l'entità che interagisce con il sistema. Questo ha il compito di inserire un input, tramite interfaccia grafica, e, a fine computazione, deve poter osservare l'output prodotto. Inoltre deve avere la possibilità di inserire nuove regole.
- **Backend system**: core del sistema. Si deve occupare di catturare l'input inserito dall'utente e produrre il *Label Transitions System* associato all'input, tramite la computazione di regole definite.

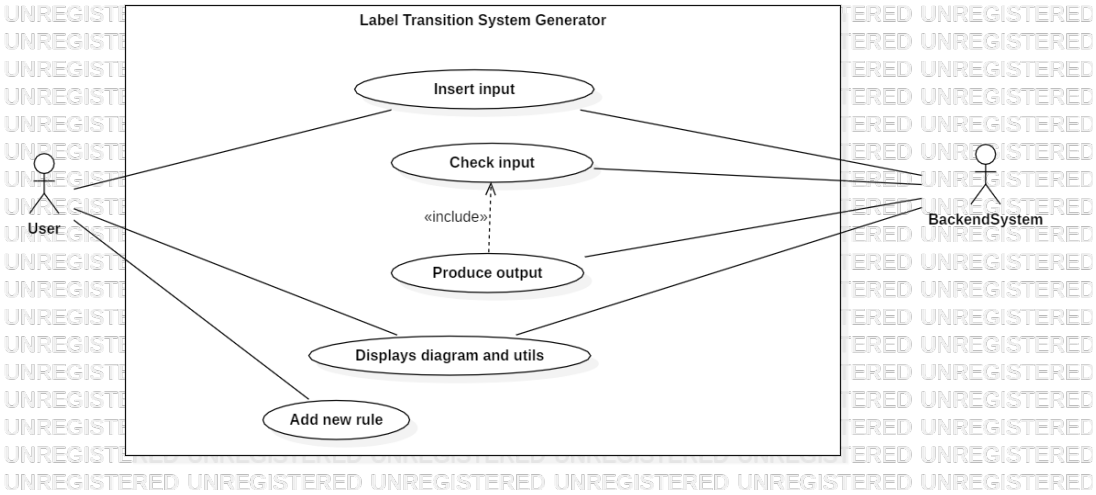

Figura 3.1: Diagramma UML dei casi d'uso rappresentante le iterazioni dell'utente con il sistema

Per lo sviluppo del sistema si è deciso di adottare il pattern architetturale **Model-View-ViewModel** (**MVVM**), variante del pattern *Model-View-Controller*. Nello specifico MVVM astrae lo stato di View e il comportamento; mentre il Model astrae una vista, e quindi crea un ViewModel, in una maniera che non dipende da una specifica piattaforma interfaccia utente.

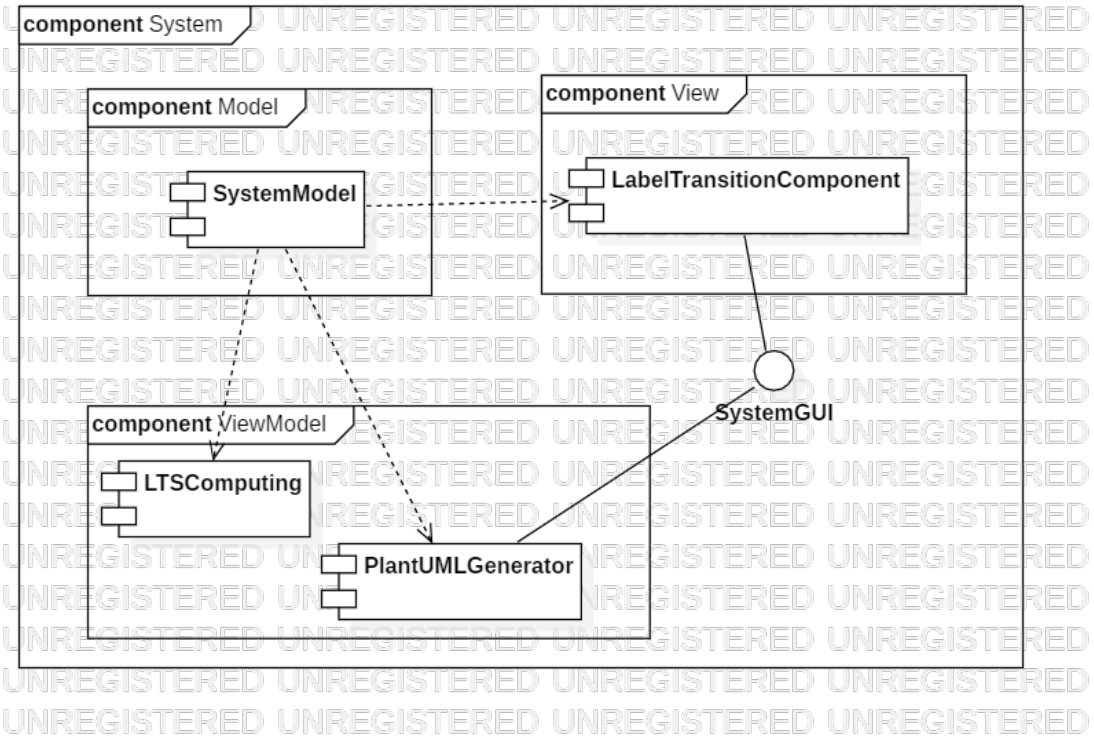

Figura 3.2: Diagramma UML dei componenti - rappresentazione del sistema

In particolare:

- **Model**: implementazione del modello di dominio (domain model) della applicazione che include un modello di dati insieme con la logica di business e di validazione.
- **View**: responsabile della definizione della struttura, il layout e l'aspetto di ciò che l'utente vede sullo schermo
- **ViewModel**: fa da intermediario tra la View e il Model, ed è responsabile per la gestione della logica della View. In genere, fornisce quindi i dati dal Model in una forma che la View può usare facilmente.

3.2 Funzionamento

Sistema

Dopo un'attenta analisi dei requisiti si sono individuati le seguenti iterazioni [Figura 3.3]:

- quando l'utente inserisce un input passa il flusso di controllo all'applicazione Java e rimane in attesa dell'output;
- quando il sistema Java riceve il flusso di controllo inizia la computazione, generando un secondo flusso di controllo verso il file Prolog, richiedendo di risolvere il goal e rimane in attesa finché non riceve una soluzione. A fine computazione si occuperà di generare il diagramma di output e lo presenterà all'utente;
- quando il Prolog riceve il flusso di controllo dal programma Java, si occuperò di attuare la giusta regola e rimandare al sistema Java la soluzione inerente.

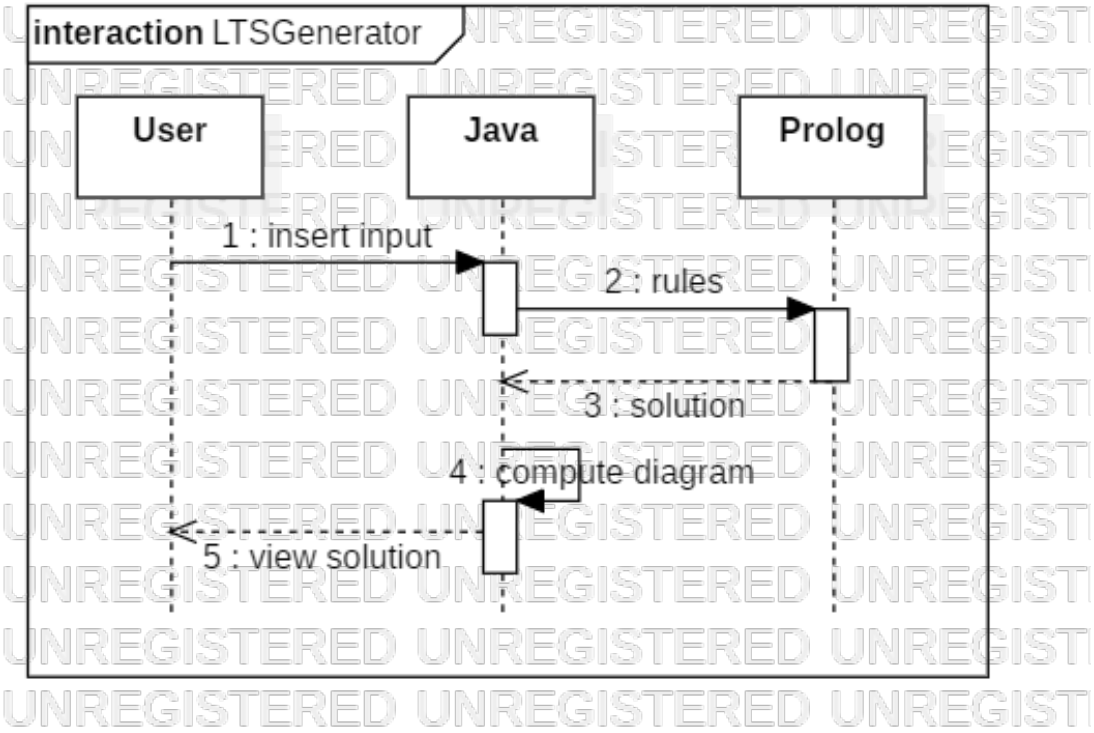

Figura 3.3: Diagramma UML di sequenza - rappresentazione del funzionamento del sistema “step-by-step”

Inserimento nuova regola

L’inserimento di una nuova regola da parte dell’utente comporta le seguenti iterazioni:

- quando l’utente inserisce una regola il flusso di controllo passa all’applicazione Java;
- quando il sistema Java riceve il flusso inserisce il flusso, crea una nuova teoria Prolog che deve essere inserita all’interno della teoria corrente;
- in questo contesto il Prolog come entità che permette di aggiungere nuove regole.

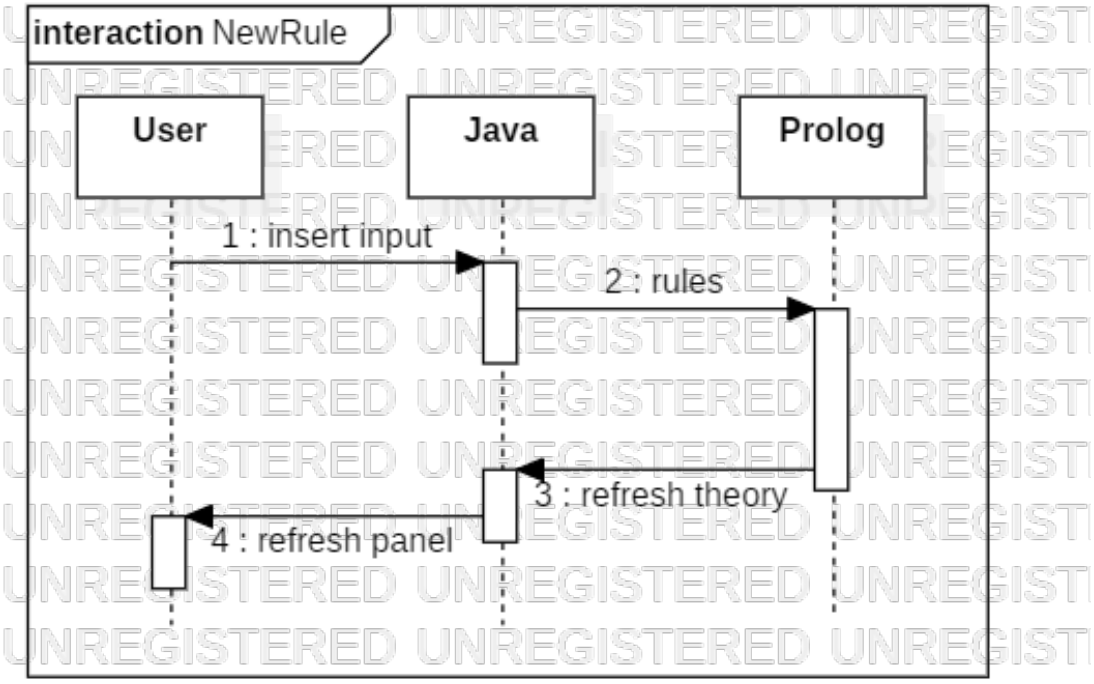

Figura 3.4: Diagramma UML di sequenza - rappresentazione del funzionamento "step-by-step" per l'inserimento di una nuova regola

3.3 Tecnologie

3.3.1 Java

Il sistema è implementato (in massima) in **Java**, un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, specificamente progettato per essere il più possibile indipendente dalla piattaforma di esecuzione. Nello specifico con tale linguaggio si vuole gestire la computazione del programma, l'interfaccia grafica e la "connessione" con il tuProlog.

3.3.2 tuProlog

Per lo sviluppo delle regole adottate dai tre diversi operatori, abbiamo decido di sfruttare la potenza del **tuProlog**, linguaggio che permette di adottare il paradigma di *programmazione logica* e che supporta nativamente la programmazione multi-paradigma, fornendo un modello di integrazione pulito e trasparente tra Prolog e i principali linguaggi orientati agli oggetti. Il Prolog si basa sul calcolo dei predicati del primo ordine e, l'esecuzione è comparabile alla dimostrazione di un teorema mediante la regola di inferenza detta risoluzione.

Così facendo con poche righe di codice è possibile implementare una potentissima "intelligenza".

3.3.3 Graphivz e PlantUML

Per quanto concerne la visualizzazione del grafo si adotterà il toolkit **Graphivz**, che accetta descrizioni di grafici in un semplice linguaggio di testo e creano diagrammi in formati utili, come immagini o PDF, oppure permette di visualizzare nel browser il grafico interattivo.

Per la creazione del digramma, invece, abbiamo sfruttato il toolkit open source **PlantUML** che utilizza Graphviz per disporre i suoi diagrammi. Questo ci permette, appunto, di creare diagrammi UML da un semplice linguaggio di testo.

Capitolo 4

Implementazione

In questo capitolo si andrà a vedere in dettaglio il codice implementato.

4.1 Model

In questo package è possibile trovare le interfacce e le classi che descrivono il modello di dominio. In particolare:

- **State**: rappresenta la struttura cardine del progetto che descrive gli stati individuati e creati a seguito di ogni computazione. Questo è composto da un identificatore univoco ed un valore ad esse associato.
- **TransitionState**: rappresenta le transazioni del grafo che comprendono lo stato iniziale, lo stato finale e l'evento che ha scatenato la transizione. Questa classe è stata pensata anche per rendere facilmente adattabile il sistema alla decisione di utilizzare PlantUML come descritto sopra.
- **LabelTransitionSystem**: gestisce le strutture dati su cui si basa la creazione del grafo degli stati finale. La classe stessa è stata gestita come un *Singleton*, in quanto è necessario assicurarsi di avere una singola istanza di questa struttura dati in modo da assicurare la consistenza in tutte le parti del programma. La struttura dati core della computazione è una mappa:

  ```
  Map <Integer , List<TransitionState>>
  ```

  La chiave della mappa rappresenta il turno di computazione e il valore è la lista di tutte le transizioni computate. Inoltre è presente anche un semplice lista contenente tutti gli stati del grafo da computare.

4.2 ViewModel

In tale package è possibile trovare le classi che rappresentano il core del programma:

- **Initialization**: gestisce il primo passo della creazione del grafo, ovvero permette l'inserimento in radice del "TransitionState", oggetto avente evento e stato iniziale nulli. Quindi da qui parte la computazione della mappa.
- **LTScomputing**: gestisce la computazione ricorsiva dell'intero grafo. Questo è possibile grazie all'utilizzo delle regole descritte nel file Prolog *LTSoperator.pl*. La classe si limita quindi ad eseguire i goal Prolog e salvare i dati all'interno delle strutture dati apposite.

Diagram

In questa sezione sono raggruppate tutte le interfacce e le classi che utilizzano la libreria Java di **PlantUML** descritto nel capitolo precedente. Nello specifico sono presenti due interfacce:

- **PlantUMLInterpreter**: ha il compito di creare il file con il formato specifico di plantUML, come si può vedere nell' esempio sottostante.

```
@startuml
skinparam DefaultFontSize 20
skinparam StateFontStyle italics
skinparam DefaultFontName Courier
hide empty description

s0 : dot(a, dot(b))
s1 : dot(b)
s2 : 0

s0 --> s1: a
s1 --> s2: b
@enduml
```

- **PlantUMLutils**: ha il compito di generare l'immagine dal file prodotto da *PlantUMLInterpreter*. In seguito si troverà l'immagine generata dal file PlantUML sopra definito.

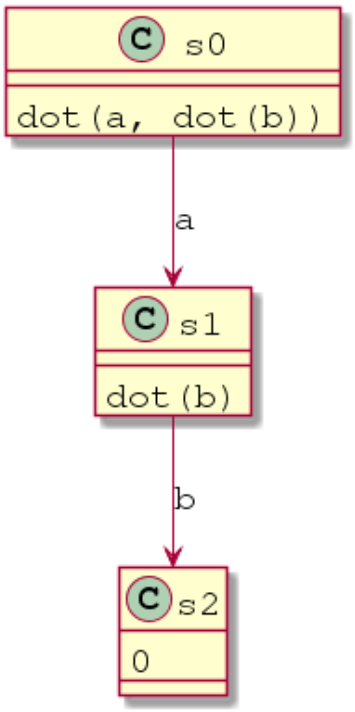

Figura 4.1: Diagramma PlantUML

4.3 View

Questo package contiene due classi utili per la gestione della GUI del sistema. Queste sono state implementate con la libreria *javax.swing*.

In particolare si trovano:

- **View**: è il frame principale nel quale è possibile inserire l'input dell'utente, aggiungere nuove regole Prolog e chiudere il programma, con i rispettivi bottoni. (vedi APPENDICE A)
- **InsertRuleView**: è il frame che compare nel momento in cui l'utente decide di aggiungere nuove regole Prolog. L'aggiunta di queste regole viene fatta tramite la liberia di Prolog per Java, nel seguente modo:

```
Theory theory = new Theory(text);
PrologConfig.engine.addTheory(theory);
prologPane.setText(PrologConfig.engine.getTheory().toString());
```

  In questo caso, la regola aggiunta non andrà a modificare il file Prolog con il quale viene inizializzato il sistema, ma viene inserita all'interno della teoria riferita all' oggetto *Prolog engine* del sistema. Per questo, come si vede nel codice, per visualizzare la teoria modificata all'interno del pannello è necessario utilizzare il metodo di libreria Prolog e non caricarla da file.

4.4 Prolog configuration

Questo package comprende tutte le classi che permettono l'integrazione fra i due linguaggi all'interno del sistema. Esso comprende:

- **Java2Prolog**: gestisce la connessione di Java a Prolog per utilizzare le regole e i predicati nel file *src/main/prolog/LTSOperators.pl*.
- **PrologConfig**: è utile per l'inizializzazione del sistema: inizializziamo la teoria prolog di base caricandola dal file *src/main/prolog/LTSOperators.pl* nel seguente modo:

```
Theory theory = new Theory(new FileInputStream(fileName));
 this.engine = new Prolog();
 this.engine.setTheory(theory);
```

  Come si può notare all'interno della classe compare un campo statico che rappresenta l'oggetto *Prolog engine* fondamentale per l'utilizzo di Prolog in Java. L'utilizzo di un campo statico è dovuto al fatto che in questo modo è visibile da tutte le istanze di quell'oggetto ed il suo valore non cambia da istanza ad istanza, per questo appartiene trasversalmente a tutta la classe.

4.5 Prolog

Qui è possibile trovare il file contenete tutte le regole Prolog che permettono la computazione del diagramma degli stati finale. In seguito si vedrà il codice specifico delle regole ricorsive che permettono questo.

```
    % rule(+InitialState, -Event, -FinalState)
rule([plus(X, XS) | PP], EV, FS) :-
        plus2list(plus(X, XS), XSS),
        member(C, XSS),
        (atom(C)
        ->EV=C, FS = 0
        ;rule([C | PP], EV, FS)).

rule([dot(X) | PP], EV, FS):-
        dot2list(dot(X), XSS),
        firstElemet(XSS, C, T),
        (atom(C)
        -> EV = C, FS = 0
        ; rule([C|PP], EV, CFS),
        list2dot([CFS | T], Y),
```

```
        FS=Y).


rule([dot(X, XS) | PP], EV, FS):-
        dot2list(dot(X, XS), XSS),
        firstElemet(XSS,C,T),
        (atom(C)
        -> list2dot(T, LD),
                EV=C,
        FS=LD
        ; rule([C|PP], EV, CFS),
        list2dot([CFS | T], Y),
        FS=Y).

rule([par(X, XS) | PP], EV, FS) :-
        par2list(par(X, XS), XSS),
        member(C, XSS, L, R),
        (atom(C)
        ->EV=C,
        append(L, [0], OUT),
        append(OUT, R, OO),
        list2par(OO, FS)
        ;       rule([C|PP], EV, CFS),
        append(L, [CFS | R], OO),
        list2par(OO, FS)).
```

Come si può vedere sono regole ricorsive che sfruttano l'unificazione e il backtracking tipico del linguaggio prolog.
Per tutte le regole e i predicati utili anche all'utente per aggiungere nuovi operatori o regole bisogna far riferimento al "Appendice B".

Capitolo 5

Retrospettiva

5.1 Sviluppi Futuri

Dopo una valutazione, abbiamo riscontrato alcune migliorie che potrebbero essere integrate all'interno del sistema per renderlo più completo. Di seguito si elencano alcuni punti:

- aggiunta di un interprete che permetta l'inserimento di un input, da parte dell'utente, più semplice e intuitivo;
- computazione dell'albero con tecniche avanzate di parallelizzazione dei task;
- ottimizzazione delle regole e dei predicati Prolog per permettere una miglior interazione dell'utente;
- sviluppo di un interfaccia grafica più consona e performante
- Aggiunta dell'operatore di "Composizione parallela" nella versione non commutativa.

5.1.1 Commenti Finali

L'elaborato è stato un buon modo per imparare in modo più approfondito degli argomenti teorici trattati a lezione e il linguaggio Prolog di cui non conoscevamo appieno le sue potenzialità.
La difficoltà riscontrata è stata con l'utilizzo di PlantUML, che comprende una libreria Java e una documentazione non completa e nemmeno intuitiva.

Appendice A

Guida utente

Avviando il programma [Figura A.1] si apre una schermata dove troviamo uno spazio dove è possibile inserire l'input. Una volta inserito l'input è necessario premere il bottone *Proccess* per permettere al sistema di iniziare la computazione. Inoltre è possibile cliccare su *InsertNewRule* per poter inserire nuove regole o *Exit* per terminare il programma.

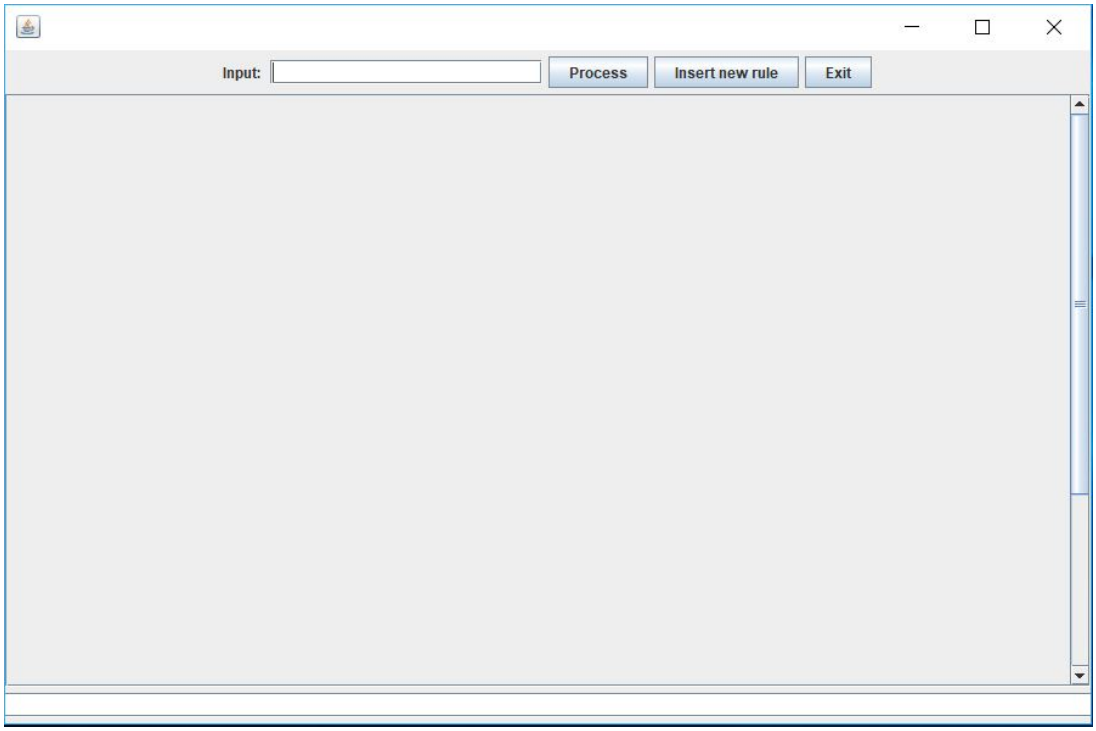

Figura A.1: Schermata iniziale sistema

Una volta che il sistema ha terminato la computazione [Figura A.2] l'utente può osservare il diagramma ottento e può cliccare il link (in basso), che reindirizzerà alla pagina web ufficiale di PlantUML.

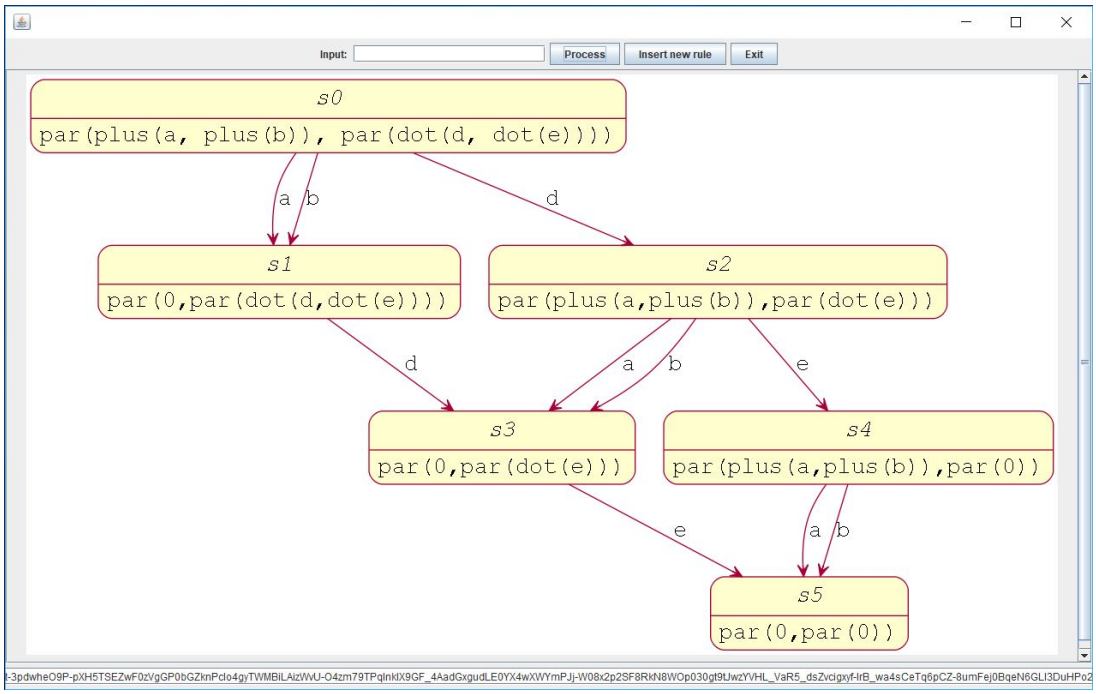

Figura A.2: Schermata del sistema a computazione terminata

Per inserire una nuova regola [Figura A.3] è sufficiente inserire l'input nell'apposita sezione di inserimento e cliccare *Insert* per vedersi apparire la nuova regola nella schermata che visualizza la teoria Prolog.

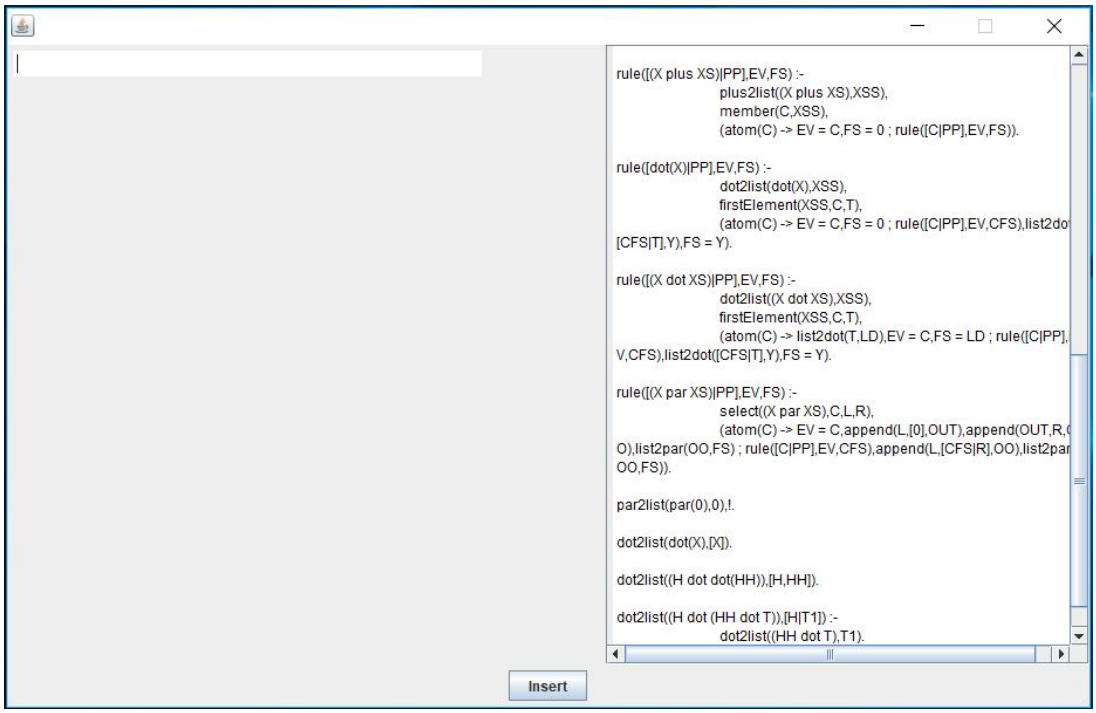

Figura A.3: Schermata iniziale sistema

Appendice B

Regole e Predicati Prolog

```
% Operators definitions
:- op(550, yfx, "par").
:- op(525, yfx, "dot").
:- op(500, yfx, "plus").


/*
This function converts parallel operator's sequence into process's sequence.
par2list(+sequence of parallel operator, -process parallel's list).
*/
par2list(par(0),0),!.
par2list(par(X),[X]).
par2list(par(X,Y), [X|Z]):-
        par2list(Y,Z).


/*
This function converts dot operator's sequence into process's sequence.
dot2list(+sequence of dot operator, -process dot's list).
*/
dot2list(dot(X),[X]).
dot2list(dot(H, dot(HH)),[H,HH]).
dot2list(dot(H, dot(HH,T)),[H|T1]):-
        dot2list(dot(HH,T),T1).


/*
This function converts plus operator's sequence into process's sequence.
plus2lit(?sequence of plus operator, ?process plus's list).
*/
plus2list(plus(H, plus(HH)),[H,HH]).
plus2list(plus(H, plus(HH,T)),[H|T1]):-
        plus2list(plus(HH,T),T1).


/*
This function converts the process's sequence into parallel operator's sequence.
list2par(+process parallel's list, -sequence of parallel operator).
*/
list2par([H], par(H)).
list2par([H,HH], par(H, par(HH))).
```

```
list2par([H|T1], par(H, par(HH,T))):-
 list2par(T1, par(HH,T)).


/*
This function converts the process's sequence into dot operator's sequence.
list2dot(+process dot's list, -sequence of dot operator).
*/
list2dot([H], dot(H)):- !.
list2dot([H,HH], dot(H, dot(HH))):- !.
list2dot([H|T1], dot(H, dot(HH,T))):-
 list2dot(T1, dot(HH,T)).


/*
This function return lists' head
first(+ListInput, -FistElem, -RemainList)
*/
firstElement([X|T],X, T).


/*
This function is abstraction of the member method. Return all element present in a list.
There are:
- right list: contains all the elements that appear before +Elem
- left list: contains all the elements that appear after +Elem
member(-Elem, +List, -RightList, -LeftList).
*/
member(E, [E | Xs], [], Xs).
member(E, [X | Xs], [X | L], R) :-
    member(E, Xs, L, R).


/*
This functions implements all operations' rules
rule(+InitialState, -Event, -FinalState)
*/
rule([plus(X, XS) | PP], EV, FS) :-
        plus2list(plus(X, XS), XSS),
        member(C, XSS),
        (atom(C)
        ->EV=C, FS = 0
        ;rule([C | PP], EV, FS)).

rule([dot(X) | PP], EV, FS):-
        dot2list(dot(X), XSS),
        firstElement(XSS, C, T),
        (atom(C)
        -> EV = C, FS = 0
        ; rule([C|PP], EV, CFS),
        list2dot([CFS | T], Y),
        FS=Y).
rule([dot(X, XS) | PP], EV, FS):-
        dot2list(dot(X, XS), XSS),
        firstElement(XSS,C,T),
```

```
        (atom(C)
        -> list2dot(T, LD),
                EV=C,
        FS=LD
        ; rule([C|PP], EV, CFS),
        list2dot([CFS | T], Y),
        FS=Y).

rule([par(X, XS) | PP], EV, FS) :-
        select(par(X, XS), C, L, R),
        (atom(C)
        ->EV=C,
        append(L, [0], OUT),
        append(OUT, R, OO),
        list2par(OO, FS)
        ;       rule([C|PP], EV, CFS),
        append(L, [CFS | R], OO),
        list2par(OO, FS)).


/*
This predicate selects the element into all processes
select(+parSequence, -currentElem, -LeftList, - RightList)
*/
select(X, C, L, R):-
        par2list(X, XSS),
        member(C, XSS, L, R).
```