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

1 Introduzione **2**

2 Requisiti **3**
2.1 Requisiti Funzionali 3
2.2 Requisiti Non Funzionali 3

3 Design **5**
3.1 Architettura 5
3.2 Funzionamento 6

4 Scelte Tecnologiche **9**
4.1 Java 9
4.2 Prolog 9
4.3 Graphivz e PlantUML 9

5 Implementazione **10**
5.1 Model 10
5.2 ViewModel 10
5.3 View 11
5.4 Prolog configuration 12
5.5 Prolog 12

6 Retrospettiva **15**
6.1 Sviluppi Futuri 15
6.1.1 Commenti Finali 15

A Guida utente **16**

B Regole e Predicati Prolog **19**

Capitolo 1

Introduzione

L'elaborato tratta uno degli argomenti visti a lezione nell'anno accademico 2018/2019: Process Algebra.

L'algebra dei processi è un formalismo che consente di modellare sistemi concorrenti e distribuiti che eventualmente interagiscono tra loro. Gli elementi base sono le *azioni* e gli *operatori* che permettono di costruire espressioni che simulano il comportamento del sistema considerato. In particolare gli operatori base comprendono:

- **Operatore di sequenze**: $X\ ;\ Y$ rappresenta un processo che prima esegue X poi Y.
- **Scelta non deterministica**: $X + Y$ rappresenta un processo che può eseguire o X o Y.
- **Composizione parallela**: $X \parallel Y$ rappresenta un processo che può eseguire X e Y concorrentemente e in modo parallelo.

L'algebra dei processi offre la possibilità di:

1. rappresentare la struttura e il comportamento di un sistema concorrente;
2. fare "verification" del funzionamento del sistema.

In questo progetto, andremo a considerare solo il primo caso.

Esistono vari approcci per descrivere la semantica dei processi. Dal punto di vista operazionale, possiamo utilizzare dei grafi di transizione che descrivono i comportamenti del sistema da modellare. In particolare il *Labelled Transition Systems* (LTS), nel quale le transizioni sono etichettate con azioni che causano il passaggio da uno stato all'altro. Formalmente, LTS è una quadrupla $S = (Q, A, \rightarrow, q_0)$ dove:

- Q è l'insieme degli stati;
- A è l'insieme finito delle azioni;
- $\rightarrow \subseteq Q \times A \times Q$ è la relazione ternaria chiamata *Transition Relation* che viene speso scritta: $q \xrightarrow{a} q'$ invece di $(q, a, q') \in \rightarrow$;
- $q_0 \in Q$ è lo stato iniziale.

Nello specifico un LTS può essere rappresentato tramite un albero la cui radice è lo stato iniziale q_0, le relazioni di transizione sono rappresentate dagli archi fra i nodi e i nodi infine rappresentano gli stati appartenenti a Q.

Il nostro progetto quindi è stato ideato per realizzare un sistema capace di produrre un intero grafo degli stati, partendo dalla descrizione di un sistema concorrente/distribuito mediante l' algebra dei processi. In questo modo è possibile creare facilmente un modello formale e testarlo, che altrimenti manualmente richiederebbe più tempo e risulterebbe più complesso.

In seguito andremo a definire le varie fasi dello sviluppo del sistema, partendo dall'individuazione dei requisiti. Procedendo poi andremo ad analizzare e sviluppare l'architettura e il design del sistema, fino ad arrivare all'implementazione vera e propria. In conclusione si troveranno i possibili sviluppi futuri, i commenti relativi allo sviluppo del sistema e le appendici per riuscire a comprendere al meglio il sistema.

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: deve quindi fornite una rappresentazione delle regole che governano un particolare sistema, espresse tramite una qualche logica formale e una rappresentazione dello stato iniziale del sistema espresso con lo stesso linguaggio di cui sopra.
- L'utente deve avere la possibilità di visualizzare l'output graficamente.
- L'utente deve avere la possibilità di modificare le regole di transizione partendo da quelle base già presenti nel sistema.
- L'utente deve avere la possibilità di visualizzare le regole di transizione già esistenti e quelle 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.
- Bisogna fornire all'utente un linguaggio concreto per rappresentare regole e stati.

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.

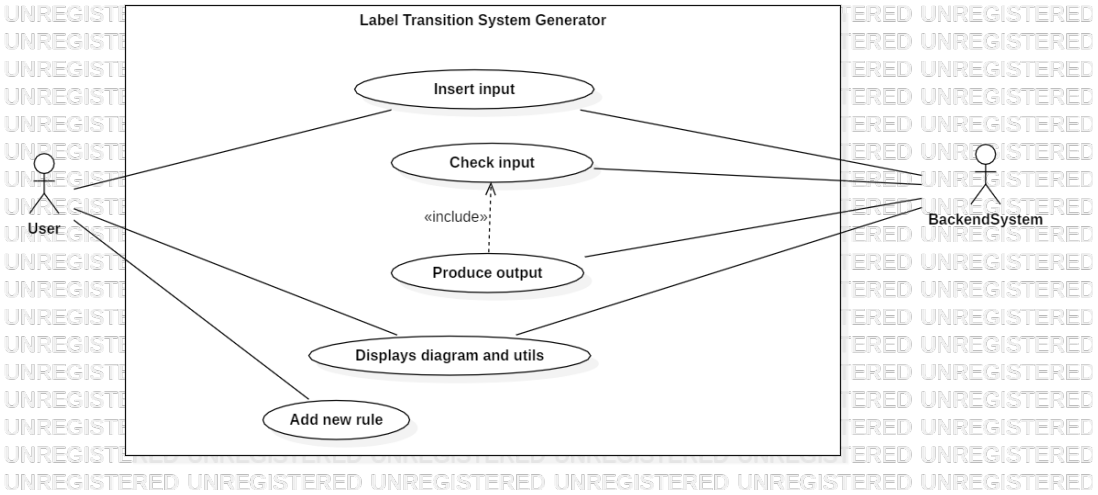

Figura 3.1: Diagramma UML dei casi d'uso rappresentante le iterazioni dell'utente con il sistema

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.

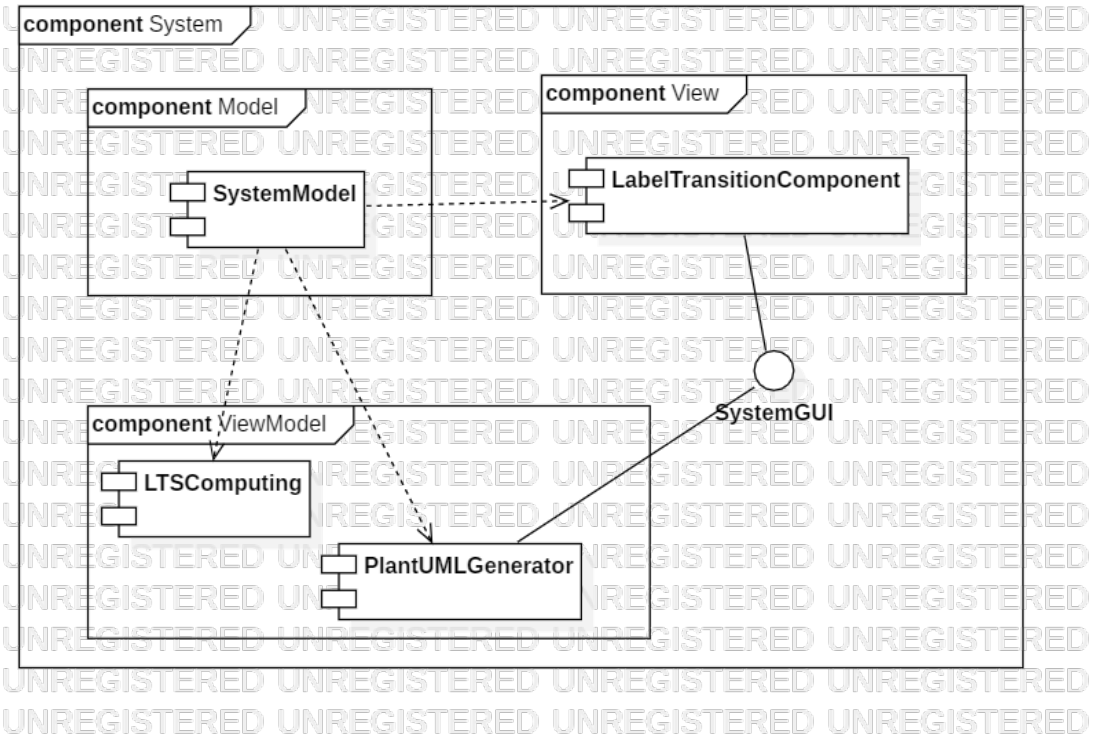

Figura 3.2: Diagramma UML dei componenti - rappresentazione del sistema

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

Il sistema è composto da tre entità principali di interazione:

- **User**: rappresenta la persona fisica che può interagire con l'applicazione.
- **OOP Application**: rappresenta il core del sistema che grazie all'utilizzo di un ambiente orientato agli oggetti, permette di definire oggetti software in grado di interagire gli uni con gli altri, fornendo così un supporto naturale alla modellazione software degli oggetti del modello astratto da riprodurre.
- **Semantic Reasoner**: rappresenta un pezzo di codice capace di inferire conseguenze logiche da un insieme di fatti o assiomi. L'impiego di un semantic reasoner potrebbe enormemente ridurre l'abstraction gap tra l'OOP, piattaforma di partenza e gli LTS in quanto, supportando nativamente backtracking e unificazione, un reasoner siffatto renderebbe molto semplice:
  - fare pattern matching sulla struttura di un sistema espresso mediante Algebra dei Processi
  - sondare tutti gli stati possibili di un sistema espresso mediante Algebra dei Processi

.

LTSGenerator

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'"OOP Application" e rimane in attesa dell'output;
- quando l' "OOP Application" riceve il flusso di controllo inizia la computazione e generando un secondo flusso di controllo verso "semantic reasoner", rimane in attesa finché non riceve la giusta risposta. A fine computazione si occuperà di generare il diagramma di output e lo presenterà all'utente;
- quando il "semantic reasoner" riceve il flusso di controllo da "OOP Application", si occuperà di computare le giuste conseguenze logiche da rimandare al "OOP Application".

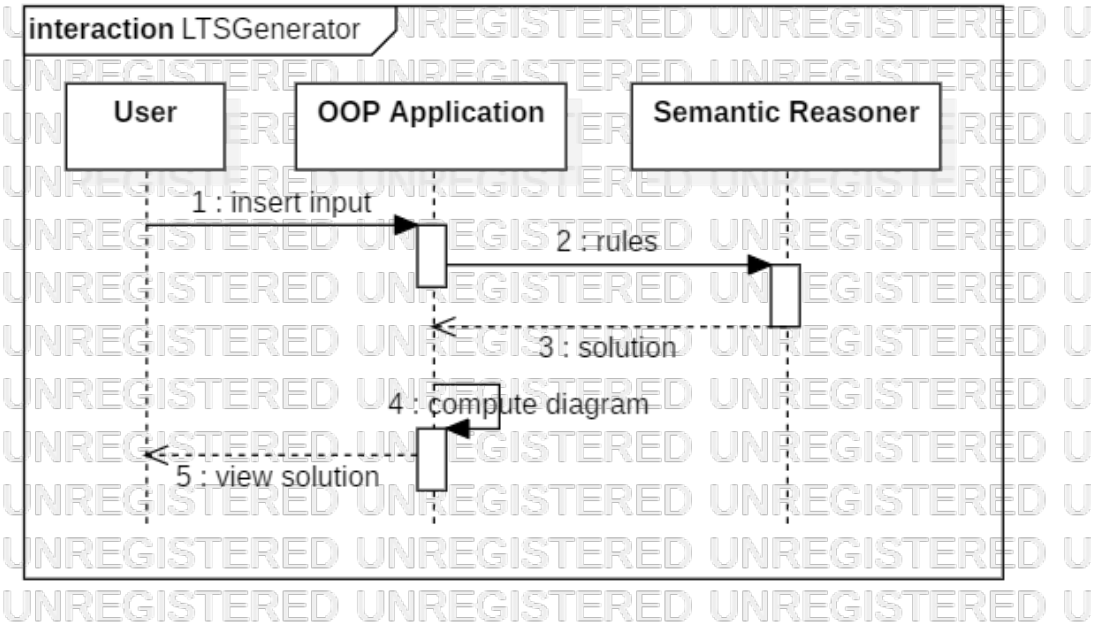

Figura 3.3: Diagramma UML di sequenza - rappresentazione del funzionamento del sistema "step-by-step"

Inserimento nuova regola di transizione

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

- quando l'utente inserisce una regola il flusso di controllo passa al "OOP Application";
- quando "OOP Application" riceve il flusso, crea un nuovo flusso i controllo che passo al "semantic reasoner", in quale inserire il nuovo assioma/regola di transizione;

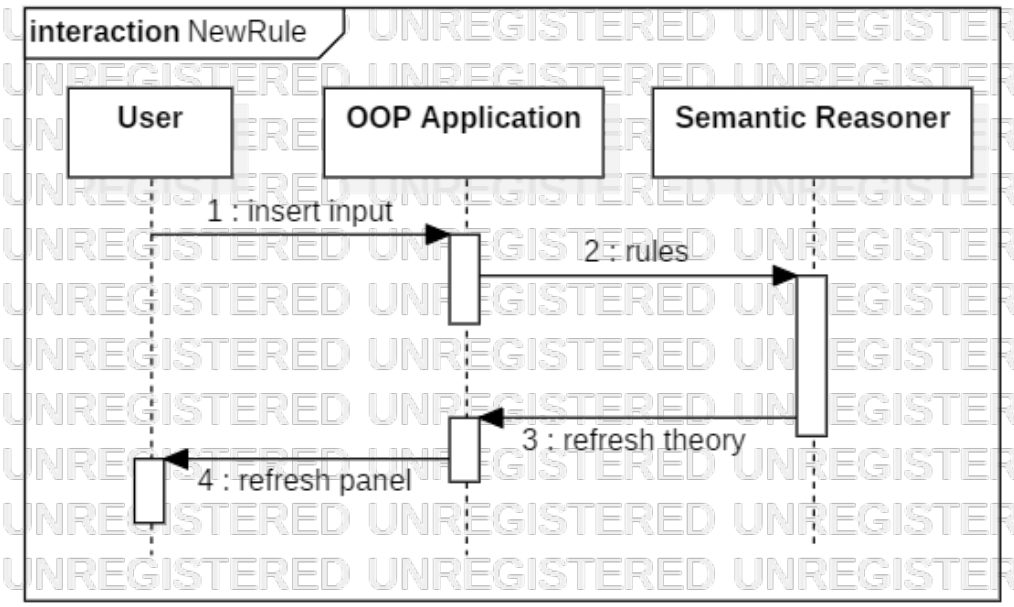

Figura 3.4: Diagramma UML di sequenza - rappresentazione del funzionamento “step-by-step” per l’inserimento di una nuova regola di transazione

Capitolo 4

Scelte Tecnologiche

In questo capitolo si andranno a presentare le scelte tecnologiche cruciali per lo sviluppo del progetto.

4.1 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.

4.2 Prolog

Il Prolog è un linguaggio di programmazione logica che 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. Questa scelta deriva dal fatto che è possibile implementare una potentissima "intelligenza", che ci permetterà, con poche righe di codice, di sviluppare le regole di transizione e gli assiomi necessari. In particolare, abbiamo utilizzato come tecnologia **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.

4.3 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 5

Implementazione

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

5.1 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.

5.2 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.

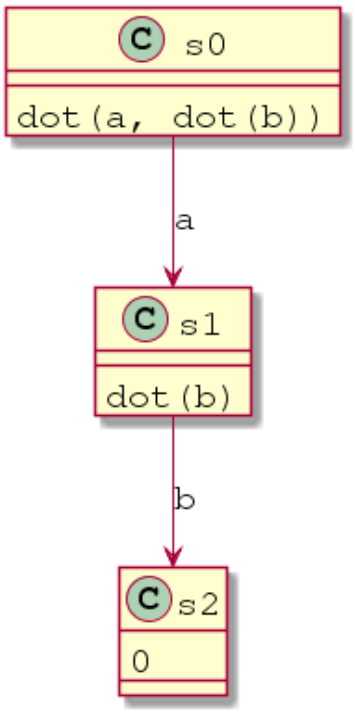

Figura 5.1: Diagramma PlantUML

5.3 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.

5.4 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.

5.5 Prolog

Questa sezione contiene il vero contributo del nostro progetto. Ciò che è stato sviluppato in Prolog consiste nella computazione della process algebra per arrivare al LTS finale del sistema d'input.

L'idea di base da cui siamo partite è quella di riuscire a sviluppare teoria Prolog che permettesse di processare l'intera algebra dei processi in modo del tutto autonomo, senza necessità di intervento da parte dell'utente e/o dell'applicazione Java. Così facendo sarà possibile modificare o sostituire successivamente il file Prolog, a patto di mantenere la medesima struttura di base, non alterando il funzionamento dell'applicazione Java.

In un primo momento è stato necessario la definizione dei tre operatori specificati nei requisiti, nel seguente modo:

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

In riferimento al codice, il predicato *:-op(...)* richiede come parametri corrispettivamente: la priorità, la regola d'associatività dell'operatore e il nome dell'operatore stesso. La priorità maggiore viene assegnata all'operatore che ha il valore più basso. Nel nostro caso l'ordine di priorità è il seguente: plus, dot e par.

Entrando nel merito della regola principale, la definizione della regola di computazione dell'albero deve attenersi al seguente pattern:

`rule(+InitialState, -Event, -FinalState)`

La regola prende in input l'algebra dei processi da computare e, essendo una regola sviluppata in modo ricorsivo, automaticamente svilupperà tutto l'input, dando di volta in volta come output l'evento della transizione e lo stato finale. Come si può notare nell'intero codice nell'appendice B, ogni operatore ha una regola a se stante che ne implementa il proprio comportamento, in quanto abbiamo sfruttato la caratteristica essenziale di Prolog dell'unificazione.

Nel dettaglio l'input che l'utente deve fornire segue in modo fedele la notazione prefissa degli operatori Prolog. Ad esempio, riuscire a prendere una semplice rappresentazione del sistema

$$\texttt{in} \cdot 0 \parallel \texttt{rd} \cdot 0 \parallel \texttt{out} \cdot 0 \parallel \emptyset$$

si trasformerà nel seguente input dell'applicazione:

$$par(in, par(rd, par(out)))$$

Mediante l'interfaccia tuProlog, si può vedere la vera e propria rappresentazione delle regole Prolog e il vero e proprio funzionamento:

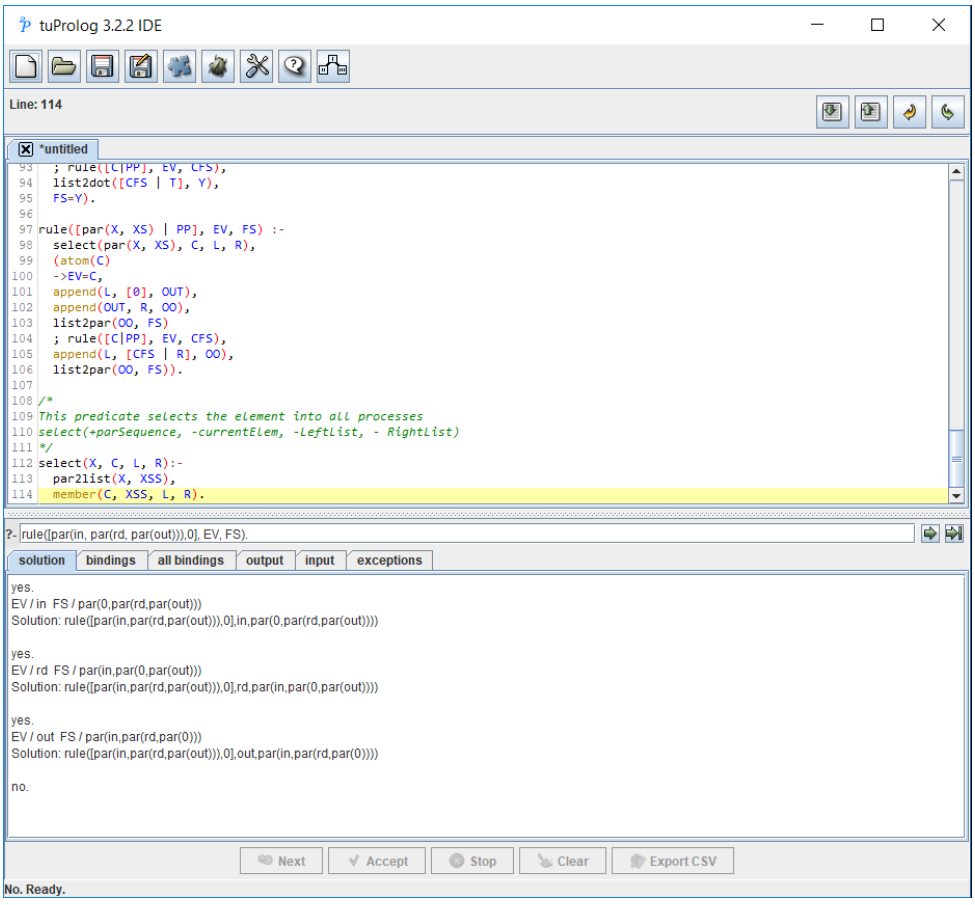

Figura 5.2: Esecuzione regole Prolog

Il goal visibile in figura 5.2 è quello che l'applicazione Java produce una volta che l'utente inserisce l'input nel formato sopra descritto. La regola prende l'algebra dei processi e esegue l'operazione di parallelo, l' output prodotto del termine *FS* viene poi memorizzato dall'applicazione Java che a sua volta riesegue in modo ricorsivo il predicato con l'output emesso finché non giungerà al termine della computazione del grafo. Questo varrà per tutte le operazioni rese disponibili nel nostro sistema.
Il grafico prodotto da questa esecuzione risulterà il seguente [Figura 5.3]:

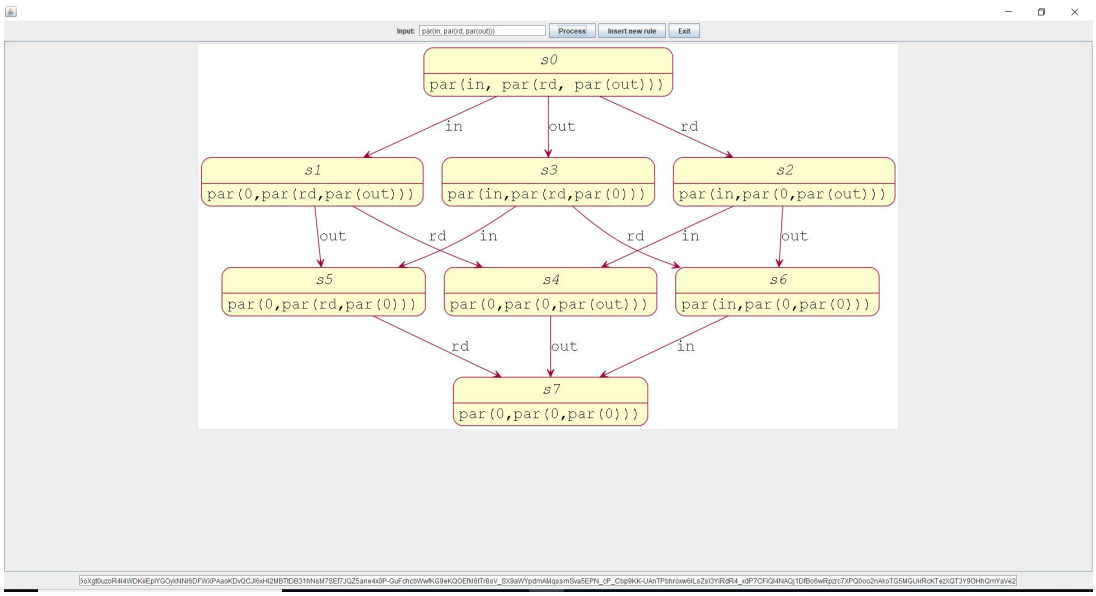

Figura 5.3: Schermata del sistema a computazione terminata

Capitolo 6

Retrospettiva

6.1 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.
- Estensione del sistema in merito alla verification.

6.1.1 Commenti Finali

In conclusione, a lavoro terminato, possiamo dichiarare che il sistema rispecchia completamente i requisiti posti all'inizio dello sviluppo. Questo si può vedere grazie all'esempio inserito alla fine del capitolo d'implementazione, in quanto si vedono esplicitamente tutti i requisiti funzionali dati.
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 maggior difficoltà riscontrata consiste nel implementare un codice Prolog con regole che potessero permettere la creazione di predicati che vadano a rendere le regole più stringenti. Questa caratteristica è limitata nel nostro sistema in quanto probabilmente il codice così definito non permette di creare semplici predicati, da fare utilizzare esternamente all'utente senza modificare le regole base già esistenti. Questo non nega il fatto di creare nuove regole e operatori che possano integrarsi con il codice già esistente, nella sessione di utilizzo.

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.

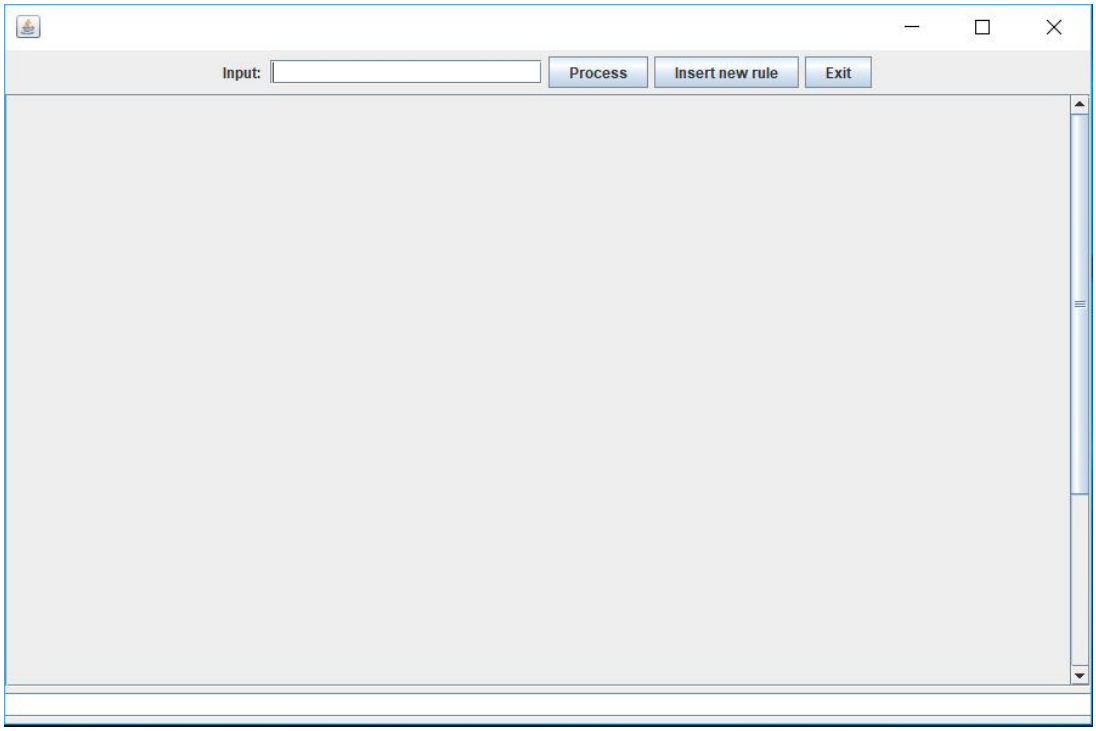

Figura A.1: Schermata iniziale sistema

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

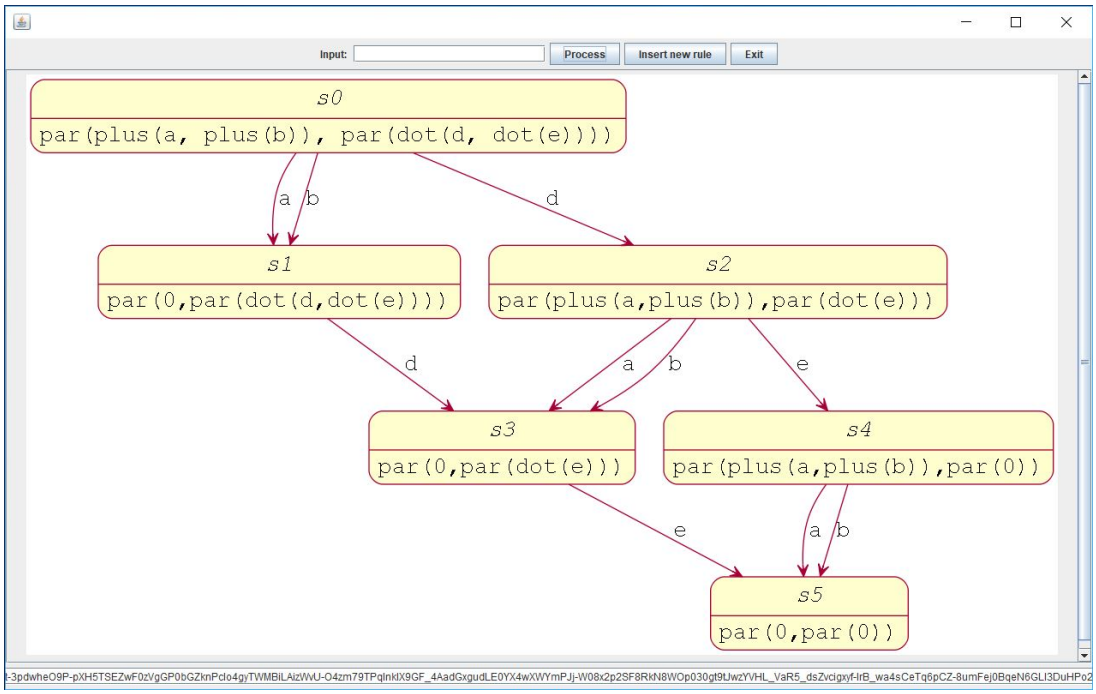

Figura A.2: Schermata del sistema a computazione terminata

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. L'input per inserire una nuova regola deve seguire fedelmente il linguaggio Prolog, in particolare con tecnologia tuProlog, seguendo e utilizzando i predicati già presenti nel file visionati nella parte destra della schermata.
Ad esempio, se si vuole definire una nuova *rule(+initialState, -Event, -FinalState)* è indispensabile attenersi a questo pattern. Stessa cosa se si vuole definire un nuovo operatore, bisogna attenersi al predicato base di tuProlog *:- op(+Priority, +AssociativeRule, +nameOperator).*

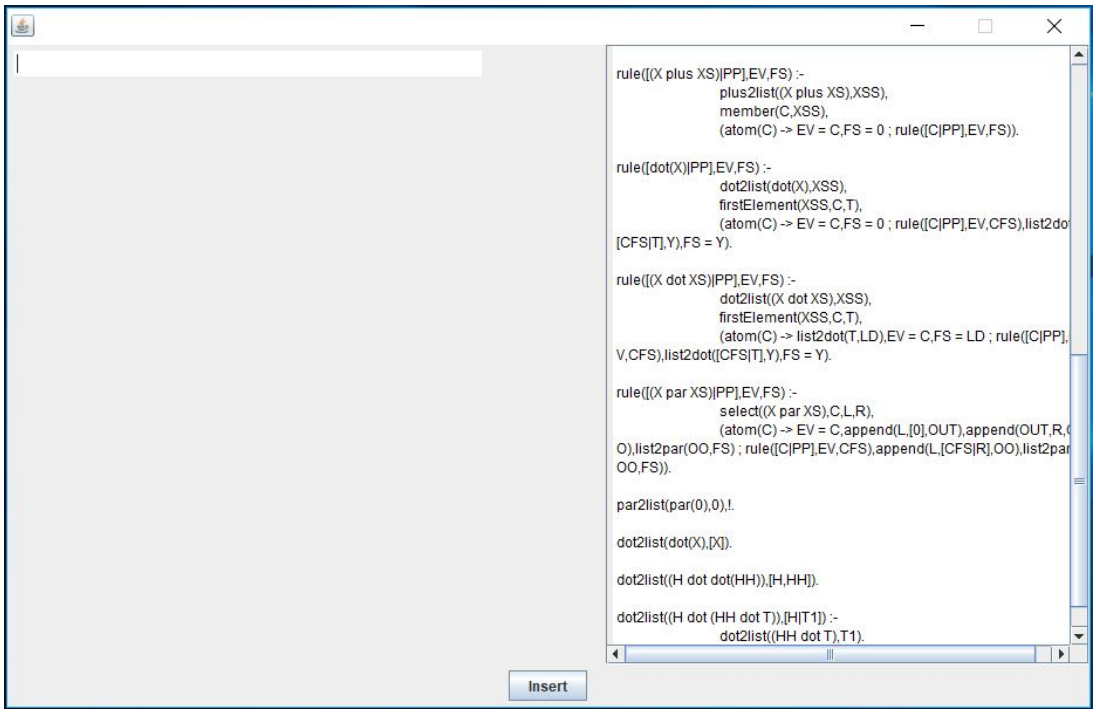

Figura A.3: Schermata iniziale sistema

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).
```