Università degli studi di Milano-Bicocca

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica


# Strumenti testuali e grafici per la specifica delle reti di Petri

**Supervisori**
Prof.ssa Lucia Pomello
Dott. Luca Bernardinello

**Relazione della prova finale di**
Marco Previtali
Matricola: 704496

Anno Accademico 2009/2010

# Indice

# Capitolo 1

# Introduzione

L'obiettivo di questo stage è continuare lo sviluppo di un linguaggio di specifica testuale per la definizione delle reti di Petri chiamato PetriNet sviluppato per essere utilizzabile anche da utenti senza nozioni di programmazione.
Le reti di Petri sono uno dei metodi di rappresentazione matematica di sistemi distribuiti inventate dal matematico tedesco Carl Adam Petri durante il suo dottorato nel 1962.
Lo sviluppo di un sistema testuale per la specifica di reti di Petri permette di svincolarsi dall'utilizzo di editor grafici e permette di sviluppare concettualmente il sistema. Sebbene sviluppare una rete di Petri in modo grafico permette di avere immediatamente un riscontro di facile comprensione (grafico) la modellazione di reti di grande dimensione è molto laboriosa e con ogni probabilità il guadagno in termini di tempo nel dichiarare una rete in modo testuale è netto.
L'utilizzo di una definizione delle reti basate su strumenti testuali permette, attraverso l'utilizzo di semplici costrutti quali *for* ed *array*, di semplificare decisamente il processo di creazione soprattutto in casi in cui siano presenti componenti ridondanti all'interno del sistema.
Le reti di Petri sono dei grafi direzionati formati da tre elementi fondamentali: i posti, le transizioni e gli archi.
I posti (il quale insieme è solitamente indicato con la lettera $P$) vengono generalmente rappresentati attraverso dei cerchi e possono o meno contenere degli elementi (chiamati token o marche) che rappresentano il fatto che un dato posto sia considerato attivo all'interno dello stato attuale del sistema.
É possibile inoltre considerare i posti come precondizioni o postcondizioni di un'azione e, in presenza di marche, pensare a queste condizioni come soddi-

sfatte o vere.
Gli archi sono archi direzionati che vanno da un posto ad una transizione o viceversa; le reti di Petri vengono definite come grafi bipartiti e quindi non è possibile collegare posti con posti e transizioni con transizioni. Gli archi vengono detti pesati quando agiscono su di un posto aggiungendo o togliendo più di una marca dai posti una volta attivati.
Le transizioni (insieme $T$) invece vengono generalmente rappresentate graficamente attraverso rettangoli o sbarre e rappresentano azioni compiute dal sistema. Le transizioni agiscono su gruppi di posti togliendo marche da alcuni e aggiungendone ad altri. Si dice che un posto è *di ingresso* rispetto ad una transizione se l'arco che collega i due elementi è direzionato verso la transizione, viceversa si dice che il posto è *di uscita*.
Una transizione si dice abilitata se tutti i posti di ingresso hanno almeno tante marche quanto è il peso dell'arco stesso. Quando una transizione *scatta* vengono tolte marche dai posti in ingresso solidalmente al peso degli archi ed equivalentemente ne vengono aggiunti ai posti in uscita.
Una delle prime modifiche apportate al linguaggio è stata quella di passare dal trattare reti elementari (formate cioè da posti contenenti al più una marca e archi con peso uno) alle più complesse reti posti e transizioni che ha richiesto la riscrittura di gran parte del codice che gestisce le strutture dati basilari al funzionamento del sistema.
La mancanza di un posizionamento manuale degli elementi della rete da una parte diminuisce il tempo necessario all'utente per definire il sistema ma dall'altra fa sorgere un problema non indifferente relativo alla scelta automatica della posizione dei nodi una volta esportato il sistema in un formato grafico. Durante questo stage non è stato possibile sviluppare un algoritmo di posizionamento adatto alle reti di Petri ma la struttura del progetto sviluppato permette facilmente l'implementazione di un metodo all'interno del codice.
La versione attuale del software sviluppato permette di esportare la struttura creata in due diversi modi. Il primo metodo è quello di utilizzare il pacchetto *Graphviz* che, attraverso gli algoritmi forniti all'utente, permette di creare reti con posizioni scelte in modo corretto. Un secondo metodo è quello di esportare la rete secondo le direttive del formato **PNML** che è stato standardizzato negli ultimi anni e sul quale parecchi tool di sviluppo a analisi per reti di Petri si basano.
Altra modifica apportata al linguaggio è stata l'aggiunta della possibilità di gestire array monodimensionali di Posti, Transizioni o Reti. Avere vettori di elementi fornisce la possibilità di utilizzare gli stessi all'interno di cicli po-

tendo in questo modo evitare di dover scrivere comandi molto simili tra di loro compattandoli viceversa in espressioni generiche.
Una delle aggiunte con più ampio impatto sulla capacità espressiva del linguaggio è stata la definizione di un costrutto che ricalca il funzionamento del ciclo `for` dei più utilizzati linguaggi di programmazione. La combinazione di questo costrutto e dei vettori, come precedentemente accennato, permette di aumentare decisamente la potenza espressiva del linguaggio e di diminuire le forze impiegate nella stesura del codice. Si è cercato di mantenere la sintassi di questo costrutto il più possibile simile a quanto presente in altri linguaggi di programmazione per rendere l'apprendimento del linguaggio semplice.
Uno degli aspetti principali del linguaggio è la possibilità di sviluppare un sistema in modo modulare, sviluppando una alla volta le componenti per poi unirle in una nuova rete generica.
La procedura di unione era già presente in PetriNet all'inizio del mio lavoro e permetteva di unire due reti ma, come si mostrerà nei prossimi capitoli, commetteva degli errori in particolari casi. Una parte del mio stage è stata dedicata al ridefinire questo costrutto in modo da avere la possibilità di unire un numero generico di reti e verificarne il corretto funzionamento. Dato che durante lo sviluppo di questa funzionalità sono sorte considerazioni contrastanti riguardo al comportamento di default del sistema è fornita all'utente la possibilità di decidere come il software sviluppato gestisca questi casi. Per fare ciò è stato necessario sviluppare un metodo per fornire direttive al parser sulla falsariga di come i maggiori compilatori C gestiscono le direttive al compilatore.
Per poter assecondare la natura descrittiva di PetriNet è stato inoltre deciso di inserire la possibilità di gestire i commenti all'interno del codice mantenendo il più possibile la similarità di PetriNet con il linguaggio C. Per ora sono permessi solo commenti monoriga o alla fine di ogni istruzione e non commenti dislocati su righe diverse.
Lo strumento software fornito per poter gestire il linguaggio sviluppato è formato da quttro componenti principali.
Il lexer chiamato già nella versione precedente `PetriNet_lex.py` permette di suddividere il codice fornito al modulo in token e di verificarne la correttezza lessicale. Una volta che il suddetto modulo esegue correttamente il suo lavoro passa gli elementi al parser (chiamato `PetriNet_parser.py`) che si occupa di verificare la correttezza sintattica dei comandi gestiti.
Sia parser che lexer sono sviluppati basandosi fortemente sul modulo `ply` che è un'implementazione dei classici strumenti *lex* e *yacc* implementato comple-

tamente in Python e sviluppato in ambito accademico da David Beazley.
Un terzo componente è il modulo `PetriNet.py` che contiene tutte le classi Python di gestione degli elementi di una rete di Petri. Questo modulo, come precedentemente scritto, è stato quasi completamente riscritto per poter funzionare con la nuova classe di reti trattate.
Ultimo modulo Python sviluppato è il modulo `pnpre.py` (Petri Net PREprocessor) che, come facilmente comprensibile dal nome, si occupa di preprocessare il file prima che lexer e parser inizino il loro lavoro. Il compito principale di questo modulo è quello di espandere correttamente i cicli for creando un numero corretto di istruzioni ed sostituendo le variabili utilizzate in modo corretto.
É stato inoltre creato un semplice script bash che permette di richiamare in modo corretto e semplice preprocessore, parser e lexer. Questo comando ha preso il nome di `pnc` (PetriNet complier) e non fa nient'altro che chiamare il preprocessore sul file passato e passare il risultato al parser che si occupa di comprendere ed eseguire il codice.
Alcune funzionalità fornite dalla precedente versione di PetriNet sono state eliminate in quanto implementate incorrettamente o comunque gestibili in modo più corretto, completo e veloce da altri tool (un esempio è la possibilità di creare il grafo delle marcature di una rete che generava grafi errati e può essere facilmente sostituito con il tool `pipe`).

I prossimi capitoli di questo documento sono suddivisi nel seguente modo:

**Capitolo 2** si occupa di presentare il linguaggio sviluppato fino al momento in cui è subentrato il mio lavoro di stage, gli strumenti utilizzati per sviluppare il software e dà una panoramica sulla struttura dei file in formato PNML.

**Capitolo 3** si occupa di analizzare e di portare una critica alle scelte prese nello sviluppo di PetriNet nel formato in cui si presentava all'inizio del mio stage. In questo capitolo vengono presentati anche i problemi che affliggevano il sistema e si è cercato di dare un'idea sulle possibili soluzioni.

**Capitolo 4** si occupa di mostrare come il linguaggio è stato modificato, come è stato scelto di trattare i problemi sorti nell'analisi mostrata nel capitolo 3 e mostra le aggiunte apportate sia al linguaggio che allo strumento software sviluppato.

**Capitolo 5** si occupa di presentare qualche esempio di come PetriNet può essere utilizzato per descrivere problemi tipici.

# Capitolo 2

# Attuale stato dell'arte

## 2.1 Introduzione alle reti di Petri

Le reti di Petri (conosciute anche come reti di Posti/Transizioni) sono uno dei metodi di rappresentazione matematica di sistemi distribuiti che devono il loro nome al matematico ed informatico tedesco Carl Adam Petri (1926 - 2010) che le inventò durante la sua tesi di dottorato nel 1962.
Queste reti sono grafi bipartiti cioè grafi nei quali i vertici possono essere disgiunti in due insiemi $U$ e $V$ tali che ogni arco colleghi un vertice di $U$ con un vertice di $V$. Nel caso delle reti di Petri questi due insiemi sono gli insiemi delle **transizioni** ($T$) e dei **posti** ($P$). Gli elementi di questi due insiemi vengono collegati da archi direzionati che fanno parte di un insieme chiamato **relazione di flusso** ($F$). L'elemento dal quale l'arco parte viene detto precondizione mentre l'elemento nel quale l'arco arriva viene chiamato postcondizione.
Più formalmente è possibile dire che

**Definizione 1.** *Una rete di Petri è una tupla $N = (P, T, F)$ per la quale valgono le seguenti condizioni:*

- $P$ *è un insieme finito di posti*
- $T$ *è un insieme finito di transizioni*
- $P \cap T = \emptyset$
- $F \subseteq P \times T \cup T \times P$

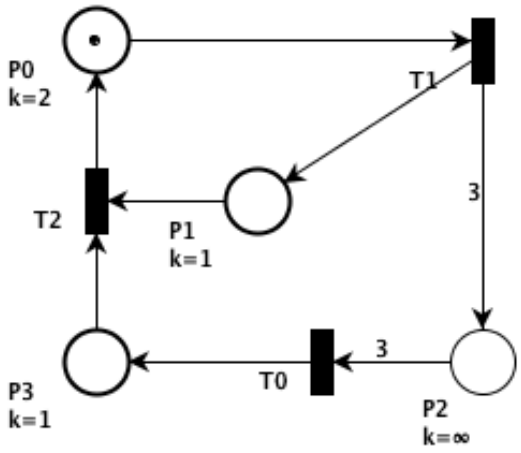



Figura 2.1: Semplice rete di Petri che mostra tutte le componenti principali: posti, capacità dei posti, transizioni, archi, archi pesati e marcatura

Una marcatura di una rete di Petri è un multiinsieme dell'insieme dei Posti ($M$) e viene chiamata marcatura iniziale o $m_0$ la marcatura relativa allo stato iniziale della rete. La marcatura può essere vista come una funzione $M : P \to \mathbb{N}$ che associa ad ogni posto il numero di token contenuti.
Ogni posto può avere una *capacità* associata che assegna ad ogni posto di $P$ un valore in $\mathbb{N}$ che indica quanti token possono essere contenuti al massimo all'interno del posto stesso. La capacità può essere vista come una funzione $K : P \to \mathbb{N}$.
Ogni arco può avere un *peso* associato che assegna ad ogni arco di $F$ un valore in $\mathbb{N}$ che indica quanti token sono consumati in un posto da una transizione o quanti token sono prodotti in un posto da una transizione. Il peso può essere visto come una funzione $W : F \to \mathbb{N}$.
Dopo aver aggiunto queste definizioni è possibile ridefinire una rete di Petri nel seguente modo:

**Definizione 2.** *Una rete di Petri è una sestupla* $N = (P, T, F, m_0, W, K)$ *per la quale valgono le condizioni della definizione 1 e in più:*

- $K : P \to \mathbb{N}$ *è una funzione che associa ad ogni posto la sua capacità*
- $W : F \to \mathbb{N}$ *è una funzione che associa ad ogni arco il suo peso*

- $m_0 : P \to \mathbb{N} : \forall p \in P \Rightarrow 0 \leq m_0(p) \leq K(p)$ *è una funzione che associa ad ogni posto il numero di marche contenute*

**Rappresentazione delle reti di Petri**

Come già mostrato in figura 2.1 le reti di Petri vengono solitamente rappresentate graficamente utilizzando dei cerchi per i posti e dei rettangoli (o delle semplici sbarre) per le transizioni.

### 2.1.1 Grafo delle marcature

Per poter descrivere il comportamento di un sistema modellato da una rete di Petri è possibile creare un grafo di tutti gli stati raggiungibili da una generica rete $N$ partendo da una marcatura iniziale $m_0$. Gli elementi di questo grafo possono essere collegati tra di loro da archi che rappresentano la transizione che venendo eseguita fa passare la rete di Petri dalla marcatura $m_i$ alla marcatura $m_k$ Questo grafo viene chiamato Grafo delle marcature o

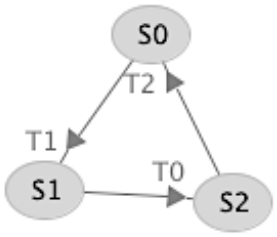



Figura 2.2: Rete di raggiungibilità della rete di Petri mostrata in figura 2.1

grafo di raggiungibilità.

### 2.1.2 Matrice di incidenza

Oltre che alla rappresentazione grafica, le reti di Petri, sono dotate di una semplice rappresentazioni matematica. Attraverso questa rappresentazione è possibile eseguire analisi automatiche della rete.
La matrice di incidenza $C$ ha $|P|$ righe e $|T|$ colonne e la generica casella $(i, k)$ rappresenta il peso dell'arco che collega il posto $p_i$ e la transizione $t_k$ con segno positivo nel caso che entri nel posto e con segno negativo nel caso

esca dal posto.
É possibile notare che esiste la possibilità (in presenza di autoanelli[1]) che la matrice di incidenza indichi il valore zero sia in caso di presenza o assenza di archi tra $p_i$ e $t_k$.

| | T0 | T1 | T2 |
|---|---|---|---|
| P0 | 0 | -1 | 1 |
| P1 | 0 | 1 | -1 |
| P2 | -3 | 3 | 0 |
| P3 | 1 | 0 | -1 |

Figura 2.3: Matrice di incidenza della rete di Petri mostrata in figura 2.1

### 2.1.3 Proprietà di una rete di Petri

#### Liveness

Una rete di Petri viene detta viva se, qualunque marcatura $m_i$ raggiunta a partire da $m_0$, è sempre possibile far scattare qualunque transizione della rete a seguita di una sequenza di scatti.
La vivezza è una proprietà non monotona nelle reti di Petri.

#### Boundness

Una rete viene detta k-limitata se esiste un numero intero positivo $k$ tale che il numero di token in ogni posto della rete è minore o uguale a $k$ per orni marcatura raggiungibile da $m_0$. Una rete di Petri si dice **safe** se è 1-limitata (1-bounded). Come è facilmente intuibile, la limitatezza di una rete dipende dalla sua marcatura iniziale.

#### P-Invarianti

I p-invarianti sono dei vettori di cardinalità $|P|$ nei quali ogni elemento è un numero $p_i \in \mathbb{N}$. Un vettore si dice essere un p-invariante se la sommatoria

[1]Si definisce autoanello una situazione nella quale un arco di peso $k$ va dal posto $i$-esimo alla transizione $k$-esima e un altro arco con lo stesso peso $k$ fa il tragitto inverso

dei prodotti del valore $i$-esimo e il numero di marche del posto $i$-esimo di una rete di Petri è costante.

**Definizione 3.** *Si dice p-invariante un vettore* $p = (p_1, p_2, \ldots, p_n)$ *con cardinalità* $n = |P|$ *tale che:*

$$\forall M \in R(N, m_0), p\prime \times M = p\prime \times m_0$$

*dove* $R(N, m_0)$ *è il grafo delle marcature della reti di Petri* $N$ *partendo dalla marcatura iniziale* $m_0$*.*

### T-Invarianti

In modo duale ai P-invarianti, i T-invarianti si riferiscono alle transizioni. Sono rappresentati da vettori di cardinalità $|T|$, i cui elementi contengono il numero di volte in cui una transizione deve scattare per riprodurre una data marcatura.
I T-invarianti rappresentano **possibili** sequenze di scatti che riportano la rete nella marcatura iniziale.
La presenza di un T-invariante non implica che è davvero possibile ritornare alla marcatura iniziale, qualunque essa sia. Infatti la marcatura iniziale potrebbe non soddisfare la condizione di abilitazione per nessuna transizione, ovvero potrebbe non esistere nessuna sequenza di scatti tale che il relativo vettore delle occorrenze sia proprio il T-invariante.
Più propriamente, quindi, un T-invariante y indica che, se fosse possibile far scattare ogni transizione del supporto di y in un ordine qualunque, allora lo stato della rete potrebbe tornare al valore iniziale al termine della sequenza.

## 2.2 Linguaggio PetriNet

PetriNet è un linguaggio di specifica testuale per le reti di Petri sviluppato da Sara Faggioni[Fag10]. Attualmente il linguaggio permette la creazione di sole reti elementari nelle quali ogni posto può contenere al più una marca e gli archi che connettono posti e transizioni utilizzano una e una sola marca per volta.

### 2.2.1 Linguaggio di specifica

Nel seguito viene fatta una introduzione al linguaggio PetriNet. Per una trattazione approfondita si veda [Fag10].

#### Rete

Esistono tre elementi sintattici che portano alla definizione di una rete

***PetriNet*** *nome*; che definisce una rete

***Place*** $p_1, \ldots, p_n$; che definisce dei posti

***Transition*** $t_1, \ldots, t_m$; che definisce delle transizioni

#### Marche

È possibile creare una marcatura nel seguente modo:
**nomeMarcatura**=$\{p_1, \ldots, p_n\}$ dove nomeMarca è il nome associato alla marcatura e $p_1, \ldots, p_n$ sono i posti marcati nella stessa.
É inoltre possibile creare marcature senza che le stesse vengano associate a nessun nome. Questa possibilità torna utile quando una marcatura viene usata una sola volta ed è inutile salvarne un riferimento.
È presente nel linguaggio la keyword *M0* che descrive la marcatura iniziale di un sistema; è possibile assegnare una qualsiasi marcatura a questo nome e il sistema, in fase di creazione del grafo, aggiungerà le marche adeguate.

#### Relazione di flusso

Esistono due metodi per indicare le relazioni di flusso, il primo è definendo collegamenti singoli tra stati e transizioni mentre il secondo è utilizzando la sintassi della regola di scatto.

Definire singoli collegamenti tra stati e transizioni è possibile nel seguente modo:

1. *posto* `->` *transizione*
2. *transizione* `->` *posto*
3. $posto_1$ `->` $transizione_1$ `->` $posto_2$ `->` $transizione_2$ `->` ...

Un secondo metodo rende possibile descrivere i posti marcati prima dello scatto di una transizione e quelli marcati successivamente allo scatto della stessa.

```
{p_k,...,p_k} [transizione_x> {p_q,...,p_n}
```

Equivalentemente è possibile cambiare le liste dei posti con il nome di una marcatura:

```
marcatura_a [transizione_x> marcatura_b
```

**Operazioni sulle reti**

**net.toDot(nomeFile, ext)** crea il file dot della rete su cui viene chiamato il metodo e un file immagine con estensione ext

**net.isOccurrency()** controlla che la rete sia una rete di occorrenze

**net.matrix()** restituisce la rete di incidenza della rete

***list*** **net.createCaseGraph(mark)** crea il grafo della rete delle marcature a partire dalla marcatura mark

**net.CGtoDot(graph,nomeFile,ext)** crea il file DOT del grafo delle marcature precedentemente creato e il file immagine con estensione ext.

***PetriNet*** **net.union(A,B) on** $p_{1n} = p_{1m}, p_{1k} = p_{2h}, \ldots$ esegue l'unione delle reti A e B, opzionalmente è possibile indicare quali posti o transizioni devono essere uniti all'interno della nuova rete. É necessario prestare attenzione sul fatto che il primo elemento appartenga alla prima rete e il secondo alla seconda. Se i due elementi sono di tipo diverso, l'unione non li prende in considerazione.

## 2.3 Strumenti

### 2.3.1 Lexer

Il lexer si occupa di analizzare un flusso di caratteri di input e di produrre in output uno stream di tokens. I tokens sono delle strutture che hanno un tipo ed un valore e sono gli elementi base su cui il Parser andrà ad operare.
Solitamente gli analizzatori lessicali basano il proprio lavoro su degli automi a stati finiti, partendo da uno stato iniziale si spostano in altri stati in base al carattere letto sullo stream di input ed una volta raggiunto uno stato di accettazione inviano il token riconosciuto al Parser.
L'individuazione di tokens all'interno di uno stream avviene tramite il riconoscimento di patterns.

### 2.3.2 Parser

Il parser si occupa dell'analisi sintattica dei token ricevuti dal lexer. All'interno di questo strumento vengono definite le regole che formano la grammatica formale del linguaggio di specifica.

## 2.4 Utilizzo dei moduli

### 2.4.1 PetriNet.py

Tramite il modulo PetriNet.py è possibile creare una rete di Petri e il relativo grafo delle marcature da riga di comando. Dopo aver aperto una sessione della shell interattiva di Python basterà importare il modulo PetriNet e creare una rete.
È possibile accedere alla documentazione di questo modulo richiamando il comando

```
>>> help(PetriNet)
```

all'interno dell'interprete.

### 2.4.2 Parser

Un altro modo per definire una rete di Petri è quello di utilizzare il linguaggio di specifica definito in 2.2.1. Esistono due metodi per poter creare reti

in questo modo: tramite l'ambiente fornito dal modulo PetriNet_parser.py oppure scrivendo un file di testo in linguaggio *PetriNet* e compilarlo tramite il parser.
Per eseguire il parser basterà fornire il nome del modulo a Python come variabile di esecuzione. Questo comando farà partire l'interprete del parser che mostrerà un ambiente di questo genere:

```
$ python PetriNet_parser.py

petriNet >
```

Da questo momento il parser accetterà comandi strutturati come precedentemente mostrato.
Come già accennato, è possibile compilare un file di testo scritto precedentemente con sintassi *PetriNet* nel seguente modo:

```
$ python petriNet_parser.py -f nomeFile
```

In questo modo l'interprete riceve in input tutti i comandi contenuti nel file passato come parametro, li esegue e termina.

## 2.5 GraphViz

Per poter visualizzare facilmente le reti create, il software PetriNet si basa sulle funzionalità fornite da GraphViz[2].
GraphViz definisce la struttura dei grafi a partire da un linguaggio chiamato dot. GraphViz è in grado di visualizzare il grafo scelto secondo vari algoritmi tra i quali *dot* per le reti gerarchiche e *circo* per le reti circolari.
Per generare le immagini di output di una rete è possibile usare il comando `dot` per il quale si rimanda alla documentazione relativa.

## 2.6 PNML

PNML[3] è un formato di documenti basato su XML ed è stato definito per la rappresentazione di reti di Petri pensando alla possibilità di venire usato come mezzo di scambio tra strumenti differenti. PNML ha subito una fase

[2]Sito internet `http://www.graphviz.org`
[3]Petri Net Markup Language

di standardizzazione ed ora è standard ISO.
La progettazione di PNML è stata fatta seguendo tre principi:

- **Leggibilità**: il formato deve essere leggibile e modificabile utilizzando un comune editor di testo.
- **Universalità**: il formato non deve escludere alcuna forma di rete di Petri.
- **Mutualità**: il formato deve permettere di estrarre la massima quantità di informazioni possibili da una rete di Petri anche se il suo tipo è sconosciuto.

PNML fornisce due meccanismi che permettono di strutturare reti molto vaste:

- **Pagine e riferimenti**: Pagine e riferimenti permettono all'utente di disegnare una rete in diverse pagine e collegare queste reti tramite i riferimenti
- **Moduli**: In molti casi l'utilizzo di Moduli è più conveniente in quanto lo stesso modulo può essere utilizzato svariate volte una volta definito. A differenza delle Pagine e dei Riferimenti, i Moduli supportano in modo migliore l'astrazione.

Pagine e riferimenti sono concetti largamente usati attualmente da molti software.

### 2.6.1 Formato generale

Fondamentalmente il formato generale di una rete di Petri è un grafo marcato asimmetrico con due tipi di nodi: Posti e Transizioni. In seguito vengono mostrati i principali concetti di un file PNML.

**Oggetti** Ogni file conforme allo standard PNML viene chiamato *Petri net file* e può contenere più reti di Petri. Ogni rete è formata da oggetti che possono essere *posti, transizioni, archi, pagine, riferimenti a posti e riferimenti a transizioni.* Ogni oggetto all'interno di una rete di Petri ha un identificatore univoco.

| Elemento XML | Attributo | Dominio |
|---|---|---|
| `<position>` | x | decimale |
| | y | decimale |
| `<offset>` | x | decimale |
| | y | decimale |
| `<dimension>` | x | decimale non negativo |
| | y | decimale non negativo |
| `<fill>` | color | colore CSS2 |
| | image | URI |
| | gradient-color | colore CSS2 |
| | gradient-rotation | {vertical,horizontal,diagonal} |
| `<line>` | shape | {line,curve} |
| | color | colore CSS2 |
| | width | decimale non negativo |
| | style | {solid,dash,dot} |
| `<font>` | family | CSS2-font-family |
| | style | CSS2-font-style |
| | weight | CSS-font-weight |
| | size | CSS2-font-size |
| | decoration | {undeline,overline,line-through} |
| | align | {left,center,right} |
| | rotation | decimale |

Tabella 2.1: Elementi grafici PNML

**Etichette** Per poter assegnare significati ad un oggetto, allo stesso possono venir assegnate delle etichette (label) che possono rappresentare svariate caratteristiche (nome, marcatura, ...). Vengono definite tue tipi di etichette: *annotazioni e attributi*. Le annotazioni sono etichette con un dominio di valori infinito mentre un attributo ha un dominio ristretto di valori.

**Informazioni grafiche** Ogni oggetto ed ogni annotazione può avere qualche informazione grafica. Per un posto e per una transizione sono la loro posizione, per un arco sono i punti per il quale lo stesso deve passare mentre per un'annotazione è la posizione relativa all'oggetto a cui è riferita.

**Informazioni specifiche del tool** Per alcuni tool può essere necessario salvare qualche informazione non necessaria per altri software. Per questo ogni oggetto può avere una sezione relativa ad ogni tool che ci interessa.

**Pagine e riferimenti a nodi** Un riferimento ad un nodo può riferire ad un qualsiasi nodo della rete, collocato in una qualsiasi pagina della rete. Non deve essere possibile creare riferimenti ciclici.

### 2.6.2 Sintassi

Viene ora mostrata una panoramica sui principali elementi del formato PNML. Nella Tabella 2.2 il tipo di dati ID è un insieme di identificatori univoci all'interno del documento PNML. Il tipo di dati IDRef è un riferimento ad un identificatore.

### 2.6.3 Graphics

Tutti gli oggetti e tutte le etichette possono avere delle informazioni grafiche. La Tabella 2.3 mostra i possibili figli della sezione `<graphics>` a seconda dell'elemento base in cui viene inserita mentre la Tabella 2.1 mostra gli elementi grafici di PNML.
Il tag `<position>` definisce la posizione assoluta di un nodo mentre `<offset>` ne definisce la posizione relativa. Per un arco, la sequenza di `<position>` definisce i punti intermedi dello stesso. I punti iniziali e finali dell'arco non vengono forniti in quanto si basano sui nodi di partenza e di arrivo dello

| Classe | Elemento XML | Attributi XML |
|---|---|---|
| PetriNet Doc | `<pnml>` | |
| PetriNet | `<net>` | `id:ID`<br>`type:anyURL` |
| Place | `<place>` | `id:ID` |
| Transition | `<transition>` | `id:ID` |
| Arc | `<arc>` | `id:ID`<br>`source:IDRef (Nodo)`<br>`target:IDRef (Nodo)` |
| Page | `<page>` | `id:ID` |
| RefPlace | `<referencePlace>` | `id:ID`<br>`ref:IDRef` |
| RefTrans | `<referenceTransition>` | `id:ID`<br>`ref:IDRef` |
| ToolInfo | `<toolspecific>` | `tool:String`<br>`version:String` |
| Graphics | `<graphics>` | |

Tabella 2.2: Elementi base PNML

| Elemento Base | Sottoelementi di <graphics> |
|---|---|
| Nodo, Page | <position><br><dimension><br><fill><br><line> |
| Arc | <position> (zero o più)<br><line> |
| Annotation | <offset><br><fil><br><line><br><font> |

Tabella 2.3: Possibili tag dell'elemento <graphics>

stesso.
Il tag <dimension> definisce altezza e larghezza di un nodo. I due elementi <fill> e <line> definiscono il colore interno e quello del bordo di un elemento. I valori di questi attributi devono essere secondo formato RGB. Per le annotazioni il tag <font> definisce il carattere che deve venire usato per visualizzare il testo delle etichette.
Per una trattazione approfondita di PNML si rimanda a [WK03] o [Kin06].

## 2.7 Python

Il software sviluppato durante questo stage è stato scritti nel linguaggio di programmazione Python.
Python è un linguaggio di programmazione di alto livello adatto a venir utilizzato in svariati ambiti applicativi. Lo sviluppo di questo linguaggio è iniziato alla fine degli anni '80 ad opera di Guido van Rossum che ne segue tuttora l'avanzamento.
Python supporta diversi paradigmi di programmazione: ad oggetti, procedurale e, seppur in modo meno esteso, il paradigma funzionale. A differenza di altri linguaggi fornisce una gestione automatica della memoria, un garbage collector e la tipizzazione dinamica e forte delle variabili.
Essendo un linguaggio interpretato necessita della presenza di un program-

ma interprete per poter eseguire il codice evitando in questo modo la fase di linking classica dei linguaggi compilati. É comunque possibile, impostando correttamente il compilatore, ottenere il file eseguibile anziché eseguire il codice.
L'implementazione più utilizzata del linguaggio è CPython sviluppato dalla Python Software Foundation ed è scritta in C ma sono largamente utilizzate anche implementazioni in Java (Jython), implementazioni che si interfacciano con l'architettura .NET di Microsoft (IronPython) nonché implementazioni scritte in Python (PyPy) che si prestano alla modifica da parte degli utenti e permettono facilmente di identificare aree nelle quali il linguaggio può venir migliorato. L'architettura di Python è stata sviluppata sin dalla sua prima implementazione in modo da rendere il linguaggio facilmente estensibile con C o C++.
Una delle caratteristiche più note di questo linguaggio è il fatto che la delimitazione dei blocchi di codice non avviene come nella maggior parte dei linguaggi con parentesi o keyword bensì tramite l'indentazione. Questa caratteristica migliora la leggibilità del codice e obbliga chi sviluppa in questo linguaggio a mantenere un certo standard nello stile di scrittura del codice a differenza di altri linguaggi che lasciano questo aspetto alla discrezionalità dell'utilizzatore.
Uno dei punti di forza di Python è certamente la libreria standard fornita con le implementazioni che dota l'utente di utili funzionalità e strutture dati complesse. Avendo inoltre una comunità molto sviluppata è possibile trovare librerie adatte ad ogni necessità già sviluppate.
Essendo Python un linguaggio interpretato le performance in termini di esecuzione del codice non sono uno dei suoi punti di forza allineandosi sostanzialmente con altri linguaggi di questo genere. L'utilizzo della libreria standard comunque permette di avere ottimi risultati in quanto la macchina virtuale è dotata di una serie di ottimizzazioni interne.
L'ostacolo delle performance inoltre può essere aggirato inserendo direttamente del codice C all'interno dei sorgenti Python sfruttando le prestazioni di un linguaggio compilato solamente nelle parti dove la rapidità di esecuzione è un requisito fondamentale.

# Capitolo 3

# Analisi di PetriNet

Nel seguente capitolo si analizzerà il linguaggio PetriNet e il funzionamento dei moduli sviluppati precedentemente evidenziandone errori e accennando possibili soluzioni

## 3.1 Linguaggio PetriNet

Nella seguente sezione viene analizzato il linguaggio PetriNet come sviluppato durante il lavoro di tesi di Sara Faggioni. Vengono riportate delle considerazioni mostrando alcuni problemi che si sono presentati durante l'utilizzo e accennando a possibili soluzioni applicabili.

### 3.1.1 Classe di reti trattate

PetriNet permette la specifica di reti elementari e non delle più generali reti P/T. Ciò limita l'utilizzo del linguaggio in quanto non permette di definire capacità per i posti e pesi per gli archi ma assume entrambi a valore 1. Questa decisione viene riflessa anche sulle strutture dati create nei moduli Python il che rende necessario una riscrittura del codice o, almeno, una profonda lettura dello stesso per effettuarne le dovute modifiche.

### 3.1.2 Rete corrente

Nello sviluppo del linguaggio un aspetto mal documentato è il concetto di rete corrente.

In PetriNet, è possibile lavorare su una rete alla volta e tutti i posti e le transizioni dichiarati verranno aggiunti automaticamente alla stessa. Il passaggio di focus da una rete all'altra può avvenire nei seguenti modi[1]:

1. **Dichiarazione di una nuova rete**
   Dichiarando una nuova rete si cambia la rete corrente sulla quale si sta lavorando. Per esempio, il seguente codice:

   ```
   PetriNet a;
   Place p1, p2;
   Transition t1, t2;
   PetriNet b;
   Place p3;
   ```

   crea due reti, una di nome `a` contenente i posti `p1` e `p2` e le transizioni `t1` e `t2` ed una di nome `b` contenente il solo posto `p3`. Si può notare come le dichiarazioni dei posti siano dislocate in più punti all'interno del codice e come il linguaggio sia in un certo modo triviale in quanto ad una prima lettura del codice potrebbe non essere ovvio il fatto che `p3` appartenga a `b`.

2. **Utilizzo della direttiva WorkOn**
   Un metodo alternativo per cambiare la rete corrente è quello di utilizzare la funzione `WorkOn(net)`. Dalla chiamata a questa funzione in poi si lavorerà sulla rete di nome `net`. Per esempio, il seguente codice:

   ```
   PetriNet a;
   Place p1, p2;
   Transition t1;
   PetriNet b;
   Place p3;
   WorkOn(a);
   Place p4;
   ```

   crea due reti, uguali a quelle del punto precedente ma aggiungendo un posto in più (di nome `p4`) alla rete `a`.

[1]Non essendo questo aspetto documentato potrebbero esistere altri casi in cui la rete corrente venga cambiata

**Problemi**

Questo aspetto, seppur utile, può portare a casi non completamente chiari e esplicativi.
Prendiamo ad esempio il seguente codice:

```
PetriNet a,b;
Place p1, p2;
```

A quale rete verranno aggiunti `p1` e `p2`?
Secondo una lettura "umana" del codice l'ultima rete dichiarata sarebbe `b` e, di conseguenza, `p1` e `p2` dovrebbero venir aggiunti a questa rete.
Ciò, invece, non avviene in quanto la regola interna al parser del linguaggio si basa su una definizione ricorsiva che vede l'ordine delle reti in modo inverso e, di conseguenza, l'ultima rete dichiarata per il sistema è `a` e i posti `p1` e `p2` vengono aggiunti alla prima rete dichiarata.
L'utilizzo del concetto di rete corrente inoltre non rende il codice corretto in quanto non è sempre chiaro a quale rete ci si vuole riferire.

### 3.1.3 Nomi dei posti

Prendiamo in considerazione il seguente codice:

```
PetriNet net;
Place a,a;
Transition b;
a -> b -> a;
```

come si nota in questa rete vengono creati due posti con lo stesso nome, un normale comportamento del sistema potrebbe essere la notifica di un errore in questo caso in quanto non è logico avere due posti distinti con lo stesso nome. In realtà il modulo modifica i nomi dei posti se già presenti e il risultato è qualcosa di completamente inaspettato; come si può vedere dall'immagine 3.1 il nome del posto a viene modificato in a_1 e viene creato un solo posto.
Un altro codice che crea grafi errati è il seguente:

```
PetriNet n;
Place a,a,a_1;
Transition t;
a_1 -> t -> a -> t;
```

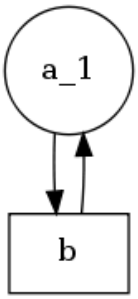



Figura 3.1: Esempio di errore nella generazione della rete

ovviamente ci si potrebbe aspettare un grafo formato da un arco che vada da un posto chiamato `a_1` ad una transizione chiamata `t` e da un cappio tra `t` e `a`. In realtà il grafo creato è quello mostrato nell'immagine 3.2, un grafo

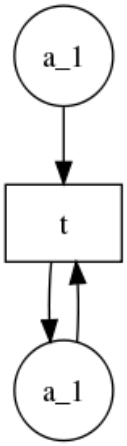



Figura 3.2: Rete con due posti con lo stesso nome

nel quale compaiono due posti con lo stesso identico nome.

### 3.1.4 Creazione grafo delle marcature

In PetriNet è possibile creare il grafo delle marcature. Tale procedura è errata e produce grafi non corretti.
Prendiamo ad esempio il seguente codice:

```
PetriNet a;
```

```
Place x;
Transition t;
x -> t -> x;
M0 = {x};
m={x};
g = a.createCaseGraph(m);
```

sarebbe corretto attendersi un grafo dei casi `g` con un cappio tra un posto relativo alla marcatura `{x}` con arco chiamato `t`. In realtà il grafo ottenuto sarà un grafo come quello mostrato in figura 3.3.

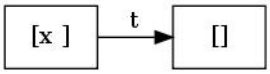



Figura 3.3: Grafo dei casi errato

#### Funzione per creare l'immagine del grafo delle marcature

Nella sezione precedente abbiamo visto la creazione del grafo delle marcature di una semplice rete.
Nel codice non è stata inserita la riga che permette la creazione del file immagine della rete. Per poter effettuare questa operazione il codice utilizzato è stato:

```
a.CGtoDot(g,imgCG,jpg);
```

La funzione viene chiamata sulla rete base (`a`), passando come parametri il grafo delle marcature (`g`), il nome dell'immagine che si vuole creare e l'estensione dello stesso.
Essendo le reti viste come oggetti, sarebbe più logico chiamare la funzione direttamente sul grafo delle marcature e non sulla rete stessa.

### 3.1.5 Commenti

Finora in PetriNet non è stata presa in considerazione la possibilità di inserire commenti nel codice.
Dato il carattere descrittivo del linguaggio sarebbe opportuno prevedere questa possibilità per aiutare il lettore nella comprensione del codice.

## 3.2 Moduli Python

In questa sezione verranno analizzati i moduli Python sviluppati.

### 3.2.1 Stile di programmazione

All'interno di tutti i moduli sviluppati è stato usato uno stile di programmazione simile a quello utilizzato per il linguaggio di programmazione Java. Secondo le linee guida ufficiali per Python (consigliate dallo stesso Guido Von Rossum a [Ros00]) lo stile di programmazione è leggermente diverso. Una lettura del PEP relativo chiarisce in che modo sarà necessario modificare i nomi delle variabili, dei metodi e delle costanti.

### 3.2.2 PetriNet.py

PetriNet.py contiene le classi utili al funzionamento del parser per il linguaggio.
Tutti gli attributi di queste classi hanno nomi comuni (`token`, `nome`, `...`) pur essendo intuitivamente attributi privati. Reimplementando queste classi è stato dunque necessario cambiarne i nomi e inserire i relativi getters e setters.
Le definizioni del metodo `__repr__()` in ogni classe sono scorrette in quanto si cerca di utilizzare questo metodo speciale come se fosse il metodo `__str__()` [2].
Per una trattazione completa degli attributi privati e dei metodi speciali si rimanda a [Py] o [Sum09].
Verranno ora analizzate una per una le classi di questo modulo.

#### Obj

La classe `Obj` è una superclasse dalla quale erediteranno sia le classi `State` e `Trans`. Contiene al suo interno solamente due variabili (`index` e `name`). La scelta di inserire il valore indice all'interno degli oggetti verrà rivista nella reimplementazione dei moduli in quanto potrebbe creare problemi di inconsistenza.

[2] Il metodo `__repr__()` serve a creare una rappresentazione dell'oggetto in modo che la valutazione della stringa ritornata crei una copia dell'oggetto stesso. Praticamente si avrà che `eval(repr(x)) == x`.

### State

State è la classe rappresentante uno stato all'interno del modulo PetriNet.
State aggiunge alla classe `Obj` una sola variabile `token` e il metodo `__repr__`.
Non si capisce se token sia una variabile booleana che informa se il posto sia pieno o vuoto (si ricorda che la classe di reti trattate permette posti di capacità 1) oppure se sia un valore intero. All'istanziazione dell'oggetto non viene fatto alcun controllo sul valore che assumo le variabili passate, di conseguenza `token` potrebbe avere valori negativi.
Questa classe accede pubblicamente alle variabili di `Obj` e non dichiara metodi getter e setter per le sue variabili.

### Trans

Trans è la classe rappresentante una transizione all'interno del modulo PetriNet.
Questa classe non aggiunge nulla alla classe `Obj` ed accede alle variabili della stessa come se fossero pubbliche.

### Link

Link è la classe rappresentante un arco all'interno del modulo PetriNet.
Questa classe ha tre variabili `state`, `trans` e `pre`. `state` rappresenta lo stato relativo all'arco, `trans` la transizione e `pre` il verso dell'arco.
La scelta di inserire una variabile booleana che indichi il verso dell'arco è discutibile. Questa variabile verrà tolta nella reimplementazione della classe e si controllerà che l'arco vada da un posto ad una transizione o viceversa durante l'inserimento.

### PetriNet

PetriNet è la classe rappresentante una rete di Petri all'interno del modulo PetriNet.
In questa classe vengono definiti metodi diversi per aggiungere posti, transizioni e link. Nella reimplementazione del modulo si è trovato più usabile l'unire queste funzioni all'interno di un semplice metodo `add` generico che si occupa di inserire l'oggetto passato nel modo corretto.
Le strutture dati utilizzate per posti, transizioni e archi sono liste; questa

scelta implica il fatto che all'interno del codice si cicli parecchie volte su queste strutture per verificare la presenza di un dato oggetto. Se si fosse scelto di utilizzare strutture quali dizionari (forniti nelle librerie standard di Python) si sarebbe potuto guadagnare espressività e comprensibilità del codice. Prendiamo come esempio i seguenti codici che effettuano una ricerca all'interno di una struttura dati e stampano il contenuto se presente.
Utilizzando le liste il risultato sarebbe qualcosa di simile:

```
x = []
[...]
pos = -1
for i in range(len(x)):
    if x[i] == element:
        pos = i
        break
if pos != -1:
    print x[pos]
else:
    print 'errore'
```

Mentre, usando i dizionari, il risultato è questo:

```
x ={}
[...]
if element in x:
# equivalentemente
# if element in x.keys():
    print x[element]
else:
    print 'errore'
```

Come si può notare l'utilizzo dei dizionari permette di scrivere codice molto più compatto e comprensibile. Per questo motivo queste strutture verranno modificate in dizionari.

### 3.2.3 PetriNet_lex.py

Questo modulo si occupa della suddivisione in token del codice basandosi sul modulo `lex.py` di `ply`[3].

[3] si veda `http://www.dabeaz.com/ply/` per maggiori dettagli su `ply`

Il modulo è sostanzialmente corretto anche se vengono definiti token non utilizzati nel linguaggio (nella fattispecie `in` e `to`).

### 3.2.4 PetriNet_parser.py

Questo modulo si occupa della definizione della grammatica formale del linguaggio analizzando la sequenza di token trasmessa dal lexer basandosi sul modulo `yacc.py` di `ply`.
In questo modulo vengono definite due variabili globali UnionA e UnionB che servono solamente come appoggio alla funzione unione. La loro visibilità a livello globale non è necessaria.
Viene poi definita una classe rete che ricalca sostanzialmente la struttura della classe `PetriNet` del modulo `PetriNet.py`; sarebbe stato più chiaro, secondo il mio punto di vista, utilizzare direttamente la classe `PetriNet.PetriNet`.
Dopo la definizione di queste variabili vengono definite le regole per il parsing del linguaggio in modo sostanzialmente corretto con quanto finora sviluppato.

# Capitolo 4

# Sviluppo di PetriNet

Nel seguente capitolo verranno mostrate le modifiche apportate al formalismo del linguaggio per ovviare ai problemi riportati nel capitolo precedente e le nuove funzionalità dello stesso.

## 4.1 Modifiche vecchie funzionalità di PetriNet

Nella seguente sezione vengono mostrate le modifiche effettuate alla specifica del linguaggio PetriNet con le motivazioni che mi hanno portato ad apportarle. Una trattazione degli aspetti modificati può essere trovata nella sezione 3.1.

### 4.1.1 Classe di reti trattate

Uno degli obiettivi del mio stage è stato portare il linguaggio a specificare reti P/T e non più solo reti di Petri elementari. È ora possibile in PetriNet definire le capacità dei posti e i pesi degli archi (si veda 4.2.3 e 4.2.5 per una trattazione migliore dell'argomento) che collegano i vari elementi della rete.

### 4.1.2 Rete Corrente

Il concetto di rete corrente è stato eliminato nella reimplementazione di PetriNet. Per una migliore comprensibilità del codice è stato scelto di dichiarare sempre la rete sulla quale si sta lavorando e non permettere al sistema di

inferire automaticamente la rete sulla quale si sceglie il focus.
È stata scelta questa modalità per ovviare ai problemi riportati nella sezione 3.1.2.

### 4.1.3 Nomi dei posti

Non è più ora possibile definire posti con lo stesso nome all'interno della stessa rete. Nel caso ciò si verificasse il programma deve terminare mostrando un errore. In questo modo non sarà più possibile avere problemi di ambiguità riguardo ai nomi.
Nel caso ora si cercasse di compilare il codice seguente[1]:

```
PetriNet x;
Place x::a, x::a;
show_places;
```

si otterrebbe un output di questo tipo[2] [3] :

```
$ python PetriNet_parser.py -f ~/temp/001.pn
It seems that an element named a is already in x
```

### 4.1.4 Creazione del grafo delle marcature

Nell'implementazione precedente di PetriNet era possibile creare il grafo delle marcature della rete definita. Come mostrato nella sezione 3.1.4 erano presenti dei casi in cui questo compito veniva svolto in modo scorretto.
Siccome questo è un aspetto di simulazione mentre PetriNet si prefigge di essere un linguaggio di specifica è stato scelto di eliminare questa funzionalità. La decisione presa non limita le possibilità fornite dall'utilizzo di PetriNet in quanto, come si vedrà in seguito, è stata aggiunta la possibilità di esportare le reti in un formato adatto a software di simulazione[4] che creano grafi delle marcature in modo corretto.

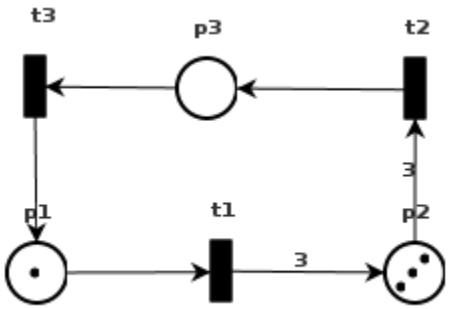



Figura 4.1: Esempio di semplice rete con marche

### 4.1.5 Marcatura iniziale

Il modo di definire una marcatura in PetriNet è stato modificato totalmente per poter essere utilizzato con reti che hanno posti con capacità maggiore di 1. Il modo migliore per poter spiegare questo nuovo metodo è mostrando un esempio.

```
PetriNet net;
Place net{p1, p2(5), p3};
Transition net{t1, t2, t3};

net{p1 -> t1 ->(3) p2 ->(3) t2 -> p3 -> t3 -> p1};

net{p1=1, p2=3};
```

Nell'esempio appena scritto viene creata una rete contenente 3 posti (`p2` con capacità 5) e 3 transizioni. Successivamente viene definito il flusso della rete descrivendo una rete circolare e dopodiché viene istanziata la marcatura iniziale della rete con il codice `net{p1=1, p2=3}` che inserisce una marca in `p1` e 3 marche in `p2`. Una rappresentazione di questa rete è mostrata in figura 4.1.

[1]Si comprenderà meglio la sintassi della dichiarazione dei posti nella sezione 4.2.3

[2]Per comodità sono stati eliminati gli output di debug che vengono comunque stampati del programma

[3]Come si può notare viene usata una funzione show_places, per il suo significato (che può apparire ovvio) si veda il seguito

[4]Vedi programma pipe a http://pipe2.sourceforge.net/

Ovviamente, nel caso si cercasse di inserire una quantità di marche maggiore alla capacità di un posto, il sistema fornirà un errore.

## 4.2 Nuove funzionalità di PetriNet

Nella seguente sezione vengono mostrate le innovazioni apportate al linguaggio PetriNet.

### 4.2.1 Direttive al parser

Durante la progettazione del sistema PetriNet ci si è accorti che non necessariamente tutte le reti definite debbano sottostare ad assunti uguali. Un esempio abbastanza banale è il fatto che alcune reti potrebbero richiedere posti con capacità 1 come default mentre alcune potrebbero assumere una capacità illimitata come comportamento standard.
Per far fronte a questo problema è stato deciso di inserire la possibilità di modificare il comportamento del sistema inserendo direttamente nel codice delle direttive[5].
Per poter passare una direttiva al parser lo schema generale è il seguente:
`# nome_parametro=valore_parametro;` [6]
Attualmente i valori gestiti da PetriNet sono *default_place_capacity*, *union_type* e *union_add_prefix*. Per una trattazione dei due parametri relativi all'unione di reti si rimanda alla sezione dedicata 4.2.7 mentre la prima verrà spiegata ora.
Il parametro *default_place_capacity* influisce sulla dichiarazione di posti nel codice. Come precedentemente accennato potrebbe essere necessario avere reti con capacità di default diversa, grazie a questo parametro è possibile modificare questo comportamento assegnandovi un numero positivo oppure il valore unlimited.
Nel caso si dichiari *default_place_capacity* come unlimited il sistema assegnerà ai posti una capacità di default concettualmente illimitata ma, in realtà, essa

[5] Chiamate per chiarezza direttive al parser in quanto sono valori che vengono considerati da questa componente del sistema

[6] É stata scelta questa sintassi in quanto simile alle direttive al preprocessore del linguaggio c e quindi è possibile utilizzare i sintax highlighting di alcuni ambienti di sviluppo

sarà comunque limitata al numero intero massimo gestibile dalla versione di Python sottostante al sistema.

### 4.2.2 Array di reti

Nella specifica di reti è possibile dover realizzare modelli con la stessa struttura base che definiscono componenti del sistema che si comportano nello stesso modo. Finora per poter modellare una situazione simile era necessario creare una componente alla volta ripetendo svariate volte le stesse righe di codice. Provando a pensare alla modellazione di un numero elevato di processi è facile comprendere come il codice diventasse di una lunghezza difficilmente contenibile.
Per ovviare a questo problema è stato deciso di dare la possibilità di creare array monodimensionali di reti. In questo modo è possibile descrivere solamente una volta il comportamento e applicarlo ad ogni componente interessata.

#### Dichiarazione

`PetriNet` *nomereti*`[`*cardinalità*`], ...;`

Per poter definire array di reti, come prevedibile, sarà necessario dare a priori una grandezza dell'array stesso ponendo la cardinalità all'interno di due parentesi quadre poste successivamente al nome dell'array, come nella maggior parte dei linguaggi di programmazione[7].
Per esempio:

```
PetriNet philosophers[5];
```

creerà 5 reti che potrebbero successivamente venire modellate come nel classico esempio dei filosofi.
È permessa l'istanziazione di array sulla stessa riga, di conseguenza il seguente codice:

```
PetriNet philosophers[5], fork[5];
```

è corretto secondo le specifiche di PetriNet.
È inoltre possibile mischiare le dichiarazioni di array di reti e di reti normali. Il seguente codice quindi è da considerarsi corretto:

[7] L'indice ammissibile per una rete dichiarata di cardinalità n andrà da 0 a n-1

```
PetriNet philosophers[5], monitor, fork[5];
```

Si può fin da ora notare come questa nuova funzionalità permetta di diminuire la quantità necessaria di codice e permetta di usare una sintassi che favorisce il raggruppamento di reti in modo concettuale.

### Modellazione

È possibile, come precedentemente accennato, modellare il comportamento di tutte le reti appartenenti ad un array in modo che siano uguali. Il seguente codice[8] [9]:

```
PetriNet philosophers[5];
[...]
philosophers{think -> get_ready -> ready -> get_left
     -> got_left -> get_right -> got_right ->
      eat -> release -> think};
[...]
```

crea la struttura base di ogni rete di philosophers in modo che ogni componente si comporti esattamente nello stesso modo.
È ragionevole pensare che in alcuni casi qualche componente abbia un comportamento particolare pur basandosi sullo stesso schema delle altre reti. Per questo è possibile aggiungere posti, transizioni e archi di flusso ad ogni singola rete appartenente ad un array.
Il seguente codice:

```
PetriNet philosophers[5];
[...]
philosophers{think -> get_ready -> ready -> get_left
     -> got_left -> get_right -> got_right ->
      eat -> release -> think};
[...]
philosophers[0]{get_ready -> times};
[...]
```

[8] Le linee di codice sono state spezzate su più righe. Il codice del flusso di transizioni, per funzionare, deve essere scritto su una riga sola

[9] Vengono in questo momento tagliate parti di codice (dichiarazione di posti e di transizioni, altri comandi) in quanto non ancora trattati

aggiunge un arco alla prima rete dell'array `philosophers` che va da una (possibile) transizione `get_ready` ad un (possibile) posto `times` che potrebbe funzionare come contatore per questa componente.

### 4.2.3 Dichiarazione di posti

Nella nuova implementazione di PetriNet è possibile definire posti in diverse modalità.

- Il primo metodo è dichiarando un posto come "attributo" della rete nel seguente modo:

  ```
  PetriNet x;
  Place x::a;
  ```

  dopo aver eseguito tale codice si avrà una rete `x` contenente un posto `a`. È ovviamente possibile dichiarare più posti in successione sulla stessa riga separati da virgole. Il codice seguente è corretto secondo la nuova specifica di PetriNet:

  ```
  PetriNet x;
  Place x::a, x::b;
  ```

  È anche possibile dichiarare posti appartenenti a reti diverse sulla stessa riga nel seguente modo:

  ```
  PetriNet x, y;
  Place x::a, y::a, x::b, y::c;
  ```

- Il secondo metodo permette di dichiarare una lista di posti appartenenti ad una rete nel seguente modo:

  ```
  PetriNet x;
  Place x{a, b, c};
  ```

  che creerà una rete contenente i posti `a`, `b` e `c`.

- Esiste una terza possibilità, della quale si scoraggia l'utilizzo, per inserire posti in una rete. Questa modalità sfrutta la capacità del sistema di fornire le funzioni dei moduli Python al linguaggio PetriNet (verrà mostrato in seguito questo aspetto). Di conseguenza è possibile definire posti "generici" slegati da ogni rete e poi andare ad aggiungerli alle reti che ci interessano. Il seguente codice potrà aiutare a chiarire le idee al riguardo:

  ```
  PetriNet x;
  Place y;
  x::add(y);
  ```

  come si può notare viene chiamata la funzione `add` sulla rete `x` passando come parametro `y`. Andando a guardare il codice Python dell'implementazione delle reti di Petri si può vedere come la classe `PetriNet` (che definisce ovviamente il concetto di rete di Petri) ha al suo interno la funzione `add` alla quale è possibile passare un qualsiasi elemento.
  Questo metodo è scoraggiato in quanto poco funzionale e complica inutilmente il codice.

### Posti con pesi

Come detto nella sezione 4.1.1 il sistema è stato portato a definire classi P/T e quindi deve essere possibile definire una capacità per i posti. Ciò è possibile inserendo, dopo il nome del posto, un numero intero e positivo racchiuso all'interno di parentesi tonde. Per esempio

```
PetriNet x;
Place x::p1(3);
Place x{p2(5), p3};
```

sono dichiarazioni valide. Mentre:

```
PetriNet x;
Place x::p1(-6);
Place x{a(0)};
```

non sono dichiarazioni valide.
Nel caso non venga dichiarata alcuna capacità il sistema assume il valore di default selezionato nelle direttive al parser.

**Array di posti**

Come per le reti, è possibile creare array di posti in modo da poter rendere più compatto il codice. Il seguente esempio contiene dichiarazioni di posti ammissibili:

```
PetriNet foo[5];
Place foo{ bar[3], baz}, foo[0]{ tar(8)};
Place foo::zip(7), foo[1]::apt(3);
```

come si può notare le combinazioni possibili per poter dichiarare posti in una rete sono molto varie e sono abbastanza comprensibili.
È stato scelto nell'implementazione che tutti i posti appartenenti ad un array debbano avere la stessa capacità in quanto un array di posti rappresenta una collezione di posti uguali e sarebbe poco logico averne con capacità diverse.

### 4.2.4 Transizioni

Equivalentemente ai posti è possibile dichiarare transizioni in 3 modi diversi:

- Dichiarando una transizione come "attributo" di una rete nel seguente modo:

  ```
  PetriNet x, y;
  Transition x::a, x::b, y::b, x::c;
  ```

- Dichiarando una lista di transizioni appartenenti ad una rete:

  ```
  PetriNet x;
  Transition x{ a,b,c};
  ```

- Aggiungendo una transizione ad una rete tramite il metodo add:

  ```
  PetriNet x;
  Transition a;
  x::add(a);
  ```

  come per i posti l'utilizzo di questo metodo di inserimento è fortemente scoraggiato.

#### Array di transizioni

Anche per le transizioni è possibile definire array per avere transizioni logicamente simili raggruppate sotto lo stesso nome.

Le seguenti dichiarazioni di transizioni sono tutte corrette:

```
PetriNet foo[5];
Transition foo{ bar[3], baz}, foo[0]{ tar}, foo::zip, foo[1]::apt;
```

### 4.2.5 Definizione del flusso

La sintassi per la definizione del flusso in PetriNet è stata rivista completamente ed è stata reimplementata ispirandosi leggermente al linguaggio di specifica FSP[10].
Per poter definire un flusso all'interno di una rete sarà necessario ora specificare su quale rete si stia lavorando in questo momento e successivamente raccogliere tra parentesi graffe una lista di posti e transizioni collegati tra di loro dai caratteri `->`. In questo modo si specificheranno percorsi all'interno della rete e il comportamento di ogni componente sarà facilmente intuibile.
È possibile definire più flussi in due modi:

- "Aprendo" due volte la rete ed inserendo un percorso nel seguente modo:

  ```
  PetriNet net;
  Place net{a,b,c};
  Transition net{x,y};
  net{ a -> x -> c};
  net{ a -> y -> b};
  ```

- Definendo un percorso alternativo con l'inserimento del simbolo di OR `|`:

  ```
  PetriNet net;
  Place net{a,b,c};
  Transition net{x,y};
  net{ a -> x -> c | a -> y -> b};
  ```

[10] Si veda http://www.doc.ic.ac.uk/∼jnm/LTSdocumention/FSP-notation.html per maggiori dettagli

In entrambi i casi il risultato sarà quello riportato in figura 4.2.

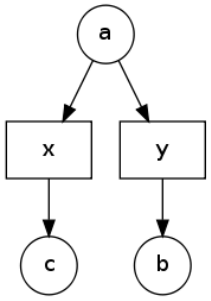



Figura 4.2: Rete semplice generata con doppio inserimento di flusso

Per ora non è ancora possibile collegare array di posti e transizioni ma solo elementi di questi array. Il codice

```
PetriNet net;
Place net{x[2], z};
Transition net{y};
net{ x -> y -> z};
```

di conseguenza non ha alcun risultato ma sarà necessario collegare ogni elemento di `x` alla transizione nel seguente modo:

```
PetriNet net;
Place net{x[2], z};
Transition net{y};
net{ x[0] -> y -> z | x[1] -> y};
```

per ottenere quanto mostrato in figura 4.3.

### Archi pesati

Come detto nella sezione 4.1.1 il sistema è stato portato a definire classi di reti P/T e quindi deve essere possibile definire archi con pesi diversi da 1. Ciò è possibile inserendo, dopo i caratteri `->` un numero intero positivo racchiuso all'interno di parentesi tonde. Per esempio la seguente porzione di codice:

```
PetriNet net;
Place net{ buffer(8)};
```

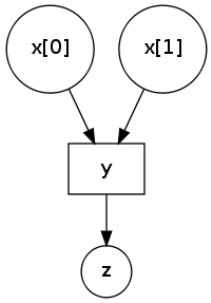



Figura 4.3: Rete semplice generata con doppio inserimento di flusso

```
Transition net{consuma_token, consuma_2_token};
net{ buffer -> consuma_token};
net{ buffer ->(2) consuma_2_token};
```

crea la rete presentata in figura 4.4 nella quale si vede che l'arco tra il posto `buffer` (di capacità 8) e la transizione `consuma_2_token` ha peso 2.

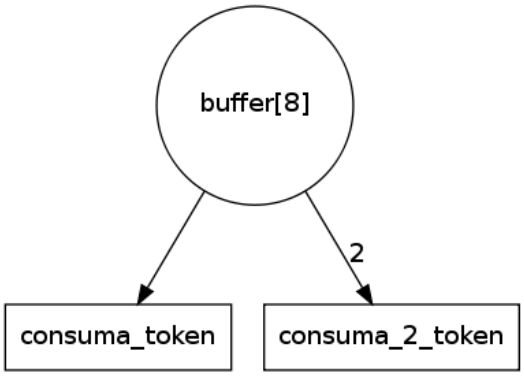



Figura 4.4: Rete con archi pesati

### 4.2.6 Commenti

Nella nuova implementazione di PetriNet è stata aggiunta la possibilità di inserire commenti all'interno del codice. Volendo mantenere uno stile `c`-like (decisione presa precedentemente al mio stage) si è optato per la possibilità di inserire commenti monoriga preceduti dai caratteri `//`. Il seguente esempio mostra dei commenti innestati nel codice:

```
PetriNet net;
Place net::a, net::b; // Commento a fine riga
// Commento che prende tutta la riga
Transition net::x, net::y;
```

### 4.2.7 Unione di reti

Per migliorare la comprensibilità di un codice scritto in linguaggio PetriNet è possibile creare delle reti che descrivono il comportamento di una componente per poi unire tutte queste sottoreti in una rete di Petri.
L'unione di reti era presente anche nella versione precedente del linguaggio PetriNet ma presentava qualche problema sulla gestione della sovrapposizione di posti e transizioni.
È possibile comporre $n$ reti sia per sovrapposizione di posti sia per sovrapposizione di transizioni e lo schema generale dell'unione di $n$ reti è il seguente:

**rete** = (*lista-reti*) **on** [*lista-elem-unione-1*, ..., *lista-elem-unione-n*];

dove:

**listareti** := *rete-1* | *rete-2* | ... | *rete-n*

**lista-elem-unione** := *elemento-1* = ... = *elemento-n* **as** *nuovo-nome*

É possibile lasciare *lista-elementi-unione* vuota.
Il comportamento dell'operazione di unione è diverso a seconda delle direttive che vengono passate al parser. Le direttive che riguardano l'unione sono due ed esattamente quelle chiamate *union_type* e *union_add_prefix*.
La direttiva *union_type* ha due valori possibili:

**only_when_equal**: in questo caso sarà possibile sovrapporre posti solamente nel caso in cui abbiano marcature e capacità uguali [comportamento di default]

**override**: in questo caso sarà possibile sovrapporre i posti sempre e gli elementi avranno la capacità e la marcatura del posto appartenente alla prima rete della lista.

La direttiva *union_add_prefix* è un valore booleano (valori True e False) che permette di decidere se durante l'unione i posti che non vengono uniti esplicitamente nella *lista-elementi-unione* debba venir aggiunto come prefisso il nome della rete alla quale appartengono o meno. Nel caso questo valore venisse impostato a `False` e nel caso in due reti diverse esistano elementi con lo stesso nome essi verranno trattati come uno solo nell'unione.
É comunque sconsigliato creare reti partendo da modelli nei quali esistono elementi con lo stesso nome che non verranno sovrapposti.
La *lista-elementi-unione* per ora deve necessariamente essere un'espressione di `n` elementi dove `n` è il numero di reti che partecipano alla procedura di unione. É molto facile notare che non necessariamente tutte le reti prendono parte con dei propri elementi ad una sovrapposizione, in questo caso, nell'elencare gli elementi, sarà sufficiente inserire la parola null nella posizione della rete che non partecipa all'unione.

### Esempi

Per capire meglio il funzionamento dell'unione portiamo ora alcuni semplici esempi. Per esempi più complessi si rimanda al capitolo 5.

### Esempio 1

Unione di due reti con posti con lo stesso nome e senza direttiva di aggiunta di prefisso.

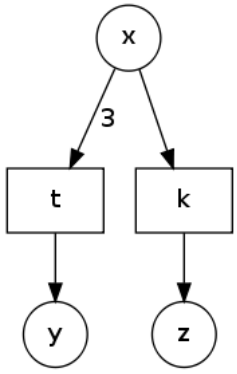



Figura 4.5: Risultato esempio 1

### Codice

```
PetriNet a,b;
Place a{x,y}, b{x,z};
Transition a::t, b::k;

a{x ->(3) t -> y};
b{x -> k -> z};

sys = (a | b) on [];
```

### Esempio 2

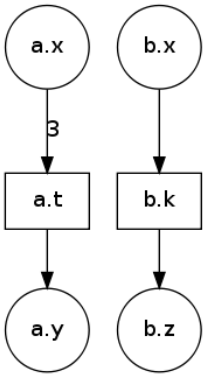



Figura 4.6: Risultato esempio 2

Unione di due reti con posti con lo stesso nome e con direttiva di aggiunta di prefisso.

### Codice

```
PetriNet a,b;
Place a{x,y}, b{x,z};
Transition a::t, b::k;

a{x ->(3) t -> y};
b{x -> k -> z};
```

```
#union_add_prefix=True;
sys = (a | b) on [];
```

### Esempio 3

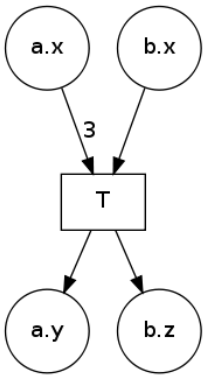



Figura 4.7: Risultato esempio 3

Unione di due reti con posti con lo stesso nome, con direttiva di aggiunta di prefisso e con dichiarazione di unione delle transizioni.

### Codice

```
PetriNet a,b;
Place a{x,y}, b{x,z};
Transition a::t, b::k;

a{x ->(3) t -> y};
b{x -> k -> z};

#union_add_prefix=True;
sys = (a | b) on [t = k as T];
```

### Esempio 4

In questo caso si cerca di sovrapporre due elementi con capacità diverse.

### Codice

```
PetriNet a,b;
Place a{x(75),y}, b{x,z};
Transition a::t, b::k;

a{x ->(3) t -> y};
b{x -> k -> z};

#union_add_prefix=False;
#union_type=only_when_equal;
sys = (a | b) on [t = k as T, x = x as X];
```

### Risultato esempio 4

Errore in quanto x della rete a e x della rete b hanno capacità diverse

### Esempio 5

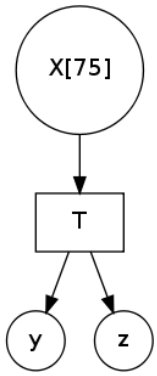



Figura 4.8: Risultato esempio 5

In questo caso si cerca di sovrapporre due elementi con capacità diverse dichiarando correttamente la direttiva union_type.

### Codice

```
PetriNet a,b;
```

```
Place a{x(75),y}, b{x,z};
Transition a::t, b::k;

a{x ->(3) t -> y};
b{x -> k -> z};

#union_add_prefix=False;
#union_type=override;
sys = (a | b) on [t = k as T, x = x as X];
```

### Esempio 6

In questo esempio vengono sovrapposte tre reti su elementi diversi.

### Codice

```
PetriNet a,b,c;
Place a{x,y}, b{x,z}, c{u,s};
Transition a::t, b::k, c::r;

a{x ->(3) t -> y};
b{x -> k -> z};
c{u -> r ->(2) s};

#union_add_prefix=False;
sys = (a | b | c) on [t = k = null as T, null = z = s as Z];
```

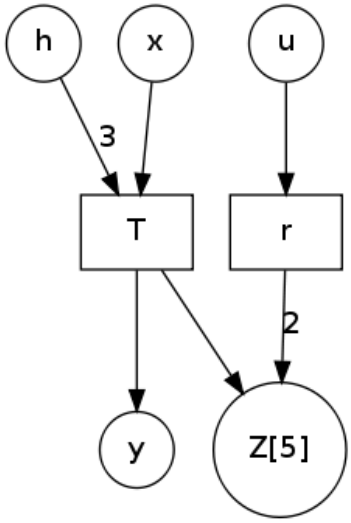



Figura 4.9: Risultato esempio 6

### 4.2.8 Ciclo for

Uno degli obiettivi principali del mio lavoro svolto su PetriNet era quello di implementare una funzione simile ai cicli for presenti in quasi la totalità dei linguaggi di programmazione.
Questa funzionalità è utile perché permette di diminuire lo sforzo nella scrittura del codice, diminuisce la lunghezza dello stesso e rende concettualmente più chiaro il flusso delle istruzioni.
La sintassi dei cicli è simile a quella classica della maggior parte dei linguaggi di programmazione che può essere schematizzata nel seguente modo:

```
for(condizioni_iniziali;condizioni_finali;incrementi){
istruzione_1;
istruzione_2;
...
istruzione_n;
}
```

dove: *condizioni_iniziali* serve ad inizializzare le variabili utilizzate come parametri all'interno del codice, *condizioni_finali* serve ad impostare le condizioni di uscita dal ciclo e *incrementi* sono le istruzioni da eseguire ogni volta che un ciclo termina.

Ogni condizione può essere multipla, in questo caso ogni istruzione deve essere separata dalle altre utilizzando una virgola.
Per ora il ciclo termina quando tutte le condizioni finali sono verificate. Non è stata implementata la possibilità di utilizzare operatori logici `or` e `and` in quanto PetriNet è pensato come linguaggio descrittivo e non come linguaggio di programmazione. Sarà comunque possibile in futuro migliorare questo aspetto.

### Espressioni matematiche

È possibile, all'interno di un blocco for, utilizzare le variabili dichiarate come indici di array di posti, transizioni o reti. È stato inoltre sviluppato un rudimentale sistema che permette di gestire correttamente i calcoli base che vedono coinvolte le variabili. Ciò vuol dire che il seguente estratto di codice è corretto:

```
for(i=0;i<10;i+=1){
net{ p[i] -> t[i] -> p[i+1]};
}
```

In questo caso l'espressione *i+1* viene correttamente interpretata dal sistema PetriNet.
In PetriNet sono gestiti i seguenti operandi tra variabili e valori interi:

\+ per la somma

\- per la sottrazione

* per la moltiplicazione

/ per la divisione

% per calcolare il modulo di due valori

Non è possibile raggruppare elementi in parentesi tonde per gestire la precedenza delle operazioni in quanto non ci si è soffermati troppo sulla gestione delle espressioni matematiche. L'ordine in cui le operazioni vengono effettuate è lo stesso in cui sono state presentate nel precedente elenco.

Anche nelle condizioni di inizio, fine e incremento del ciclo sono presenti delle

espressioni matematiche. In queste espressioni è possibile utilizzare qualsiasi costrutto matematico conosciuto dalla versione Python utilizzata [11].

#### Cicli innestati

È possibile utilizzare i cicli for in modo annidato. In questo caso tutte le variabili dichiarate nel ciclo esterno saranno utilizzabili anche nel ciclo interno. Il seguente codice di conseguenza è corretto:

```
for(i=0; i<5; i+=1){
  for(j=0; j<i; j+=1){
    net{p[i] -> t[j] -> p[i+1]};
  }
}
```

#### Limitazioni sui pesi degli archi

Per ora non è possibile utilizzare le variabili dei cicli come pesi per i posti degli archi.

## 4.3 Moduli Python

All'interno di questa sezione verranno mostrate le modifiche apportate ai moduli Python sviluppati per il linguaggio PetriNet.

### 4.3.1 Stile di programmazione

Durante la riscrittura del codice si è cercato di portare lo stile di programmazione il più vicino possibile a quanto consigliato da Guido Von Rossum a [Ros00]. Il vecchio codice non faceva utilizzo di list comprehension che sono molto utili per scrivere codice compatto, ove possibile si è cercato di utilizzare questa funzionalità del linguaggio.

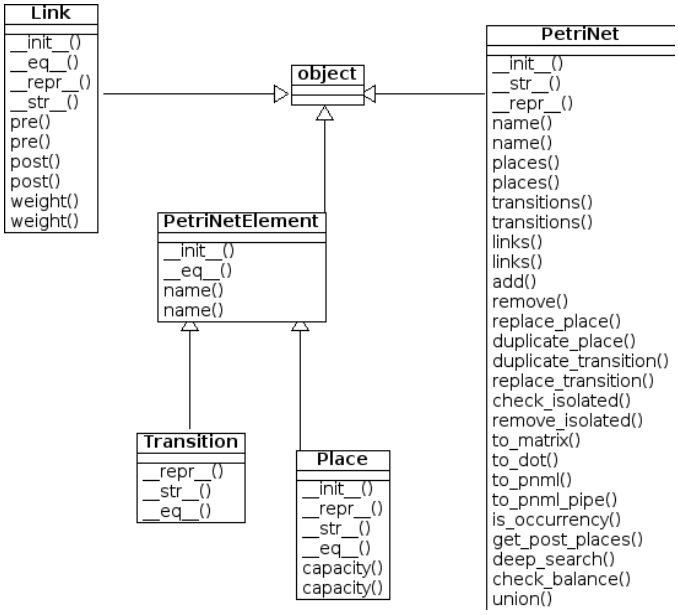



Figura 4.10: Diagramma UML delle classi Python sviluppate

### 4.3.2 PetriNet.py

I nomi degli attributi di ogni classe del modulo PetriNet.py sono stati modificati rendendoli attributi privati e creando i relativi metodi getter e setter. Per quasi ogni classe sono stati creati i metodi speciali `__repr()__`, `__str()__` e `__eq__()` corretti. È possibile vedere uno schema UML del sistema sviluppato nell'immagine 4.10.
Verranno ora analizzate le classi di questo modulo.

#### PetriNetElement

La classe PetriNetElement prende il posto della classe Obj del sistema sviluppato precedentemente. Il cambio di nome è stato effettuato per aumentare la comprensibilità del codice in quanto Obj non dava alcuna informazione sul tipo di dato ed era confondibile con la classe `object` di Python. In questa classe viene definito un attributo di nome name che sta ad indicare il nome dell'oggetto e i relativi getter e setter.

[11] Questo vuol dire che il costrutto `++` non è utilizzabile e che il codice `x++` dovrà essere sostituito dal codice `x+=1`.

### Place

Place è la classe rappresentante un posto all'interno del modulo PetriNet. L'unico attributo aggiunto da questa classe alla classe PetriNetElement (dalla quale deriva) è capacity che sta ad indicare, appunto, la capacità del posto. Vengono definiti i metodi speciali `__repr()__`, `__str()__` e `__eq__()` e il getter e il setter per l'attributo capacity.

### Transition

Transition è la classe rappresentante una transizione all'interno del modulo PetriNet. Deriva dalla classe PetriNetElement e ridefinisce i metodi `__repr()__` e `__str()__` senza aggiungere alcun altro attributo.

### Link

La classe Link rappresenta un arco all'interno del modulo PetriNet. Nel reimplementare la classe è stato eliminato l'attributo pre che indicava il verso dell'arco. Ora questa classe contiene due soli attributi, `pre` che indica lo stato o transizione dal quale l'arco parte e `post` che indica a quale stato o transizione l'arco arriva.
Altra modifica apportata è stata l'aggiunta di un attributo `weight` che indica il peso dell'arco, nell'inizializzazione dell'oggetto questo valore assume come default 1 e non è possibile inserire valori negativi o 0.
Sono stati definiti anche tutti i metodi getters e setters per ogni attributo della classe.

### PetriNet

PetriNet è la classe rappresentante una rete di Petri all'interno del modulo PetriNet.
Questa classe ha quattro attributi: `name`, `places`, `transitions` e `links`. Il primo è il nome della rete mentre gli altri sono liste contenenti rispettivamente posti, transizioni e archi.
In questa classe sono stati raggruppati tutti i metodi per inserire oggetti all'interno della rete in un solo metodo chiamato per l'appunto `add` che si occupa di capire il tipo di oggetto passatogli e di aggiungerlo alla struttura dati adeguata. Sono stati inoltre implementati metodi per la rimozione dei posti e delle transizioni e per il rimpiazzamento di questi elementi.

È stata inoltre inserita la possibilità di esportare una rete in formato `pnml`. Come si può notare dando una rapida lettura al codice sono stati implementati due diversi metodi che permettono di esportare in questo formato. Questa scelta è dovuta al fatto che il programma `pipe` non rispetta completamente lo standard `pnml` e, ad esempio, usa l'attributo `value` invece che `text` per indicare alcuni valori. Purtroppo, non avendo ancora trovato un algoritmo di posizionamento adatto per le reti di Petri tutti gli elementi esportati in `pnml` saranno posizionati automaticamente dal metodo in posizione `(0, 0)` e sarà necessario spostarli manualmente per creare una vista migliore della rete.
In questa classe sono stati rimossi i metodi per la creazione del grafo delle marcature per le motivazioni spiegate nella sezione 4.1.4.

### 4.3.3 PetriNet_lex.py

Come precedentemente detto questo modulo si occupa della suddivisione in token del codice basandosi sul modulo `lex.py` di `ply`.
Nella reimplementazione sono stati eliminati i token inutilizzati precedentemente e sono state inserite le seguenti regole:

`'|'` per consentire di riconoscere l'`or` nella descrizione di flussi

`'COMMENT'` per riconoscere i commenti all'interno del codice (si veda la sezione 4.2.6)

`'ARRAY_ID'` per riconoscere un identificatore di array

`'DDOT'` per riconoscere il doppio due punti (`::`) per permettere di accedere a funzioni di una rete in stile (`C++`)

`'STRING'` per riconoscere stringhe (questa regola è diversa dalla regola ID in quanto in una stringa possono essere presenti caratteri non accettabili per un ID)

Sono inoltre state inseriti i seguenti token utili per il debug:

`'show_places'` che permetterà di stampare a video i posti dichiarati in un certo momento

`'show_transitions'` che permetterà di stampare a video le transizioni dichiarate in un certo momento

`'show_nets'` che permetterà di stampare a video le reti dichiarate in un certo momento

`'show_links'` che permetterà di stampare a video gli archi dichiarati in un certo momento

Sono stati anche eliminati e seguenti token presenti precedentemente:

`'toDot'` in quanto, come si vedrà successivamente è stata implementata una funzione in `PetriNet_parser.py` che è in grado di richiamare ogni funzione della classe `PetriNet` senza dover dichiarare token dedicati

`'isOccurrency'` per lo stesso motivo del token `toDot`

`'matrix'` per lo stesso motivo del token `toDot`

`'createCaseGraph'` in quanto non più presente questa funzionalità

`'CGtoDot'` in quanto non più presente la funzionalità di creare grafi delle marcature

`'WorkOn'` in quanto non più presente il concetto di rete corrente (si veda sezione 4.1.2)

`'ADD'` per lo stesso motivo del token `toDot`

`'to'` in quanto inutilizzato

`'in'` in quanto inutilizzato

### 4.3.4 PetriNet_parser.py

```
$ ./PetriNet_parser.py --help
Usage: PetriNet_parser.py [-f INPUT_FILE] [-i]

Options:
  -h, --help            show this help message and exit
  -f INPUT_FILE, --file=INPUT_FILE
                        File su cui richiamare il parser
  -i, --interactive     Esegue il parser in modo interattivo
                        (in questo caso il file passato non
                        viene preso in considerazione)
```

Questo modulo si occupa della definizione della grammatica formale del linguaggio analizzando la sequenza di token trasmessa dal lexer.
In questo modulo sono state eliminate le variabili globali inutili precedentemente presenti (`UnionA` e `UnionB`).
Essendo stato eliminato il concetto di rete corrente è stata eliminata la variabile `actualNet` e la definizione della classe `rete`. Il dizionario `net` è poi stato rinominato in `nets` e sono stati creati dei dizionari chiamati `places`, `transitions` e `links` che contengono rispettivamente i posti, le transizioni e gli archi dichiarati non appartenenti ad alcuna rete[12].
Vengono successivamente dichiarate le regole per la dichiarazione di reti, posti e transizioni nelle modalità viste nelle sezioni precedenti.
Particolarmente interessante può essere la funzione `p_call_function_on_net` che permette di richiamare tutti metodi della classe `PetriNet`. Questa funzione crea una regola che verifica la presenza del token `DDOT` e del token `ID` successivamente ad un token `ID` o un token `ARRAY_ID`. Successivamente viene dinamicamente valutata l'esecuzione della funzione scelta sulla rete selezionata.
In questo modo è stato possibile eliminare tutti i token e tutte le regole che effettuavano questo lavoro. Un possibile problema di questa funzionalità è che tutti i metodi della classe `PetriNet` sono invocabili dal codice passato al parser. Per evitare questo spiacevole inconveniente sarebbe possibile obbligare a chiamare in qualche particolare modo i metodi "privati" (magari aggiungendo un particolare prefisso) e fare una selezione all'interno della funzione `p_call_function_on_net`.
Al termine di questo file vengono inserite le regole `show_nets`, `show_places` `show_transitions` e `show_links` utili per il debug del programma ma praticamente inutili al fine del linguaggio PetriNet.
Come si può vedere dall'help fornito all'inizio della sezione, per poter richiamare il parser interattivo è ora necessario passare il parametro `-i`.
Il modulo ora ricava i parametri passati tramite il modulo Python `optparse` che permette più flessibilità nella gestione degli stessi.

[12]Esiste attualmente, in PetriNet, la possibilità di dichiarare posti, transizioni e archi non appartenenti ad alcuna rete. Elementi dichiarati in questo modo non hanno molta utilità se non nel caso si volessero poi aggiungere a svariate reti attraverso il metodo `add`. È comunque sconsigliata la dichiarazione di posti e transizioni in questo modo

### 4.3.5 Il preprocessore: pnpre.py

```
$ ./pnpre.py --help
Usage: pnpre.py -i INPUT_FILE [-o OUTPUT_FILE] [-p]

Options:
  -h, --help            show this help message and exit
  -i INPUT_FILE, --input=INPUT_FILE
                        Source file
  -o OUTPUT_FILE, --output=OUTPUT_FILE
                        Destination file
  -p, --print-output    Print the output to console
                        instead output file
```

Per poter far fronte alla necessità di fornire un costrutto for al linguaggio PetriNet è stato necessario implementare un modulo che preprocessasse i file in ingresso interpretando correttamente le condizioni, gli incrementi e le variabili all'interno dei cicli.
Il modulo pnpre.py[13] effettua esattamente questo compito permettendo all'utente di passare il proprio file contenente codice corretto secondo le specifiche di PetriNet ed ottenendo un file espanso correttamente.
Le opzioni che questo modulo accetta sono:

**-i** per fornire il file di input al modulo

**-o** per dire al modulo su quale file salvare l'output

**-p** per dire al modulo di stampare l'output sulla console invece che su di un file

Come precedentemente accennato questo modulo permette di espandere i valori delle variabili presenti all'interno del ciclo for in modo corretto quando le stesse sono utilizzate come indici in array ma non quando vengono utilizzate come pesi degli archi.
Portiamo ora qualche esempio per mostrare come il modulo si comporta con diversi file di input.

[13] pnpre sta per Petri Net PREprocessor

### Esempio 1

Il primo esempio mostra semplicemente come il modulo espanda gli elementi all'interno del codice.
Passando il seguente codice al modulo:

```
for(i=0;i<5;i+=1){
  rete{ posto -> transizione -> posto2};
}
```

si ottiene

```
rete{ posto -> transizione -> posto2};
rete{ posto -> transizione -> posto2};
rete{ posto -> transizione -> posto2};
rete{ posto -> transizione -> posto2};
rete{ posto -> transizione -> posto2};
```

### Esempio 2

Questo esempio mostra come le variabili all'interno di un ciclo vengano espanse correttamente e come la parte del preprocessore che si occupa delle espressioni matematiche gestisca le precedenze fra gli operatori.
Passando il seguente codice al modulo:

```
for(i = 0; i < 5; i+=1){
  rete{p[i] -> t[i] -> p[i+1 % 5]};
}
```

si ottiene

```
rete{p[0] -> t[0] -> p[1]};
rete{p[1] -> t[1] -> p[2]};
rete{p[2] -> t[2] -> p[3]};
rete{p[3] -> t[3] -> p[4]};
rete{p[4] -> t[4] -> p[0]};
```

### Esempio 3

In questo esempio viene mostrato come il preprocessore gestisca correttamente i cicli innestati.
Passando il seguente codice al modulo:

```
for(i=0; i<3; i+=1){
  for(j=i; j<3; j+=1){
    rete[i]{p[j] -> t[j-1 % 3] -> p[j+1]};
  }
}
```

si ottiene:

```
rete[0]{p[0] -> t[2] -> p[1]};
rete[0]{p[1] -> t[0] -> p[2]};
rete[0]{p[2] -> t[1] -> p[3]};

rete[1]{p[1] -> t[0] -> p[2]};
rete[1]{p[2] -> t[1] -> p[3]};

rete[2]{p[2] -> t[1] -> p[3]};
```

### 4.3.6 pnc

Per poter gestire più comodamente i passaggi per poter ottenere il risultato voluto è stato creato uno script bash che permette di preprocessare e fare il parsing del file. Questo script è stato chiamato pnc[14] ed accetta in ingresso un solo argomento (il file su cui fare il parsing). É necessario eseguire questo script dall'interno della cartella che contiene i file `PetriNet_parser.py` e `pnpre.py`.

[14] acronimo di Petri Net Compiler

# Capitolo 5

# Esempi di utilizzo

Nel seguente capitolo verranno mostrati e spiegati esempi di utilizzo didattici del linguaggio PetriNet.

## 5.1 Processo sequenziale ciclico

In questo semplicissimo esempio viene mostrato come modellare un comune processo ciclico che presenta un caso di conflitto.

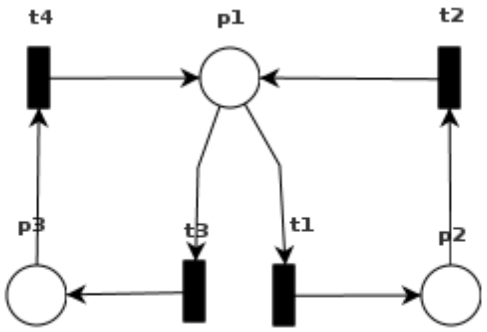



Figura 5.1: Processo sequenziale che presenta concorrenza sul posto p1

```
PetriNet net;
Place net{p1,p2,p3};
Transition net{t1,t2,t3,t4};
```

```
// Modellazione flussi
net{ p1 -> t1 -> p2 -> t2 -> p1 | p1 -> t3 -> p3 -> t4 -> p1};

net::to_pnml_pipe("~/reti/processo_seq_concorrenza.xml");
```

Il risultato della compilazione di tale codice è mostrato nella fugura 5.1[1].

## 5.2 Processi che si contendono una risorsa

In questo esempio si modellano due processi con lo stesso comportamento e li si combina con un processo che descrive una generica risorsa che può essere libera oppure occupata. Il risultato è mostrato in figura 5.2.

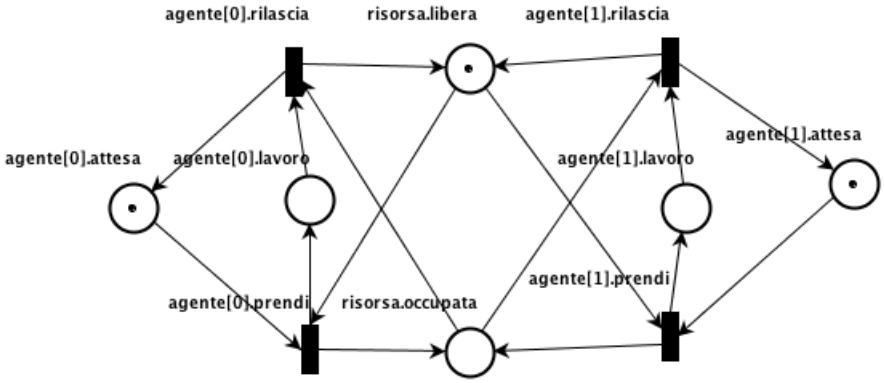



Figura 5.2: Rete che descrive due processi che competono per l'utilizzo di una risorsa

```
PetriNet agente[2], risorsa;
Place agente{attesa, lavoro}, risorsa{libera, occupata};
Transition agente{prendi, rilascia}, risorsa{prendi, rilascia};

agente{attesa -> prendi -> lavoro -> rilascia -> attesa};
risorsa{libera -> prendi -> occupata -> rilascia -> libera};
```

[1]Il file è stato modificato con il software pipe per poter avere un layout corretto

```
agente{attesa=1};
risorsa{libera=1};

#union_add_prefix=True;
for(i=0;i<2;i+=1){
  agente[i] = (agente[i] | risorsa) on
     [rilascia = rilascia as agente[i].rilascia,
      prendi = prendi as agente[i].prendi];
 }
#union_add_prefix=False;
sys = (agente[0] | agente[1]) on [];

sys::to_pnml_pipe("~/Desktop/conc.xml");
```

## 5.3 Comunicazione tra due processi

In questo classico esempio verrà modellato un sistema che simula l'azione di due processi `sender` e `receiver`.
Il processo `sender` produrrà un messaggio, poi lo depositerà all'interno di un buffer e attenderà di ricevere un ack per poi reiterare lo stesso ciclo.
Il processo `receiver` attenderà di ricevere un messaggio, invierà l'ack e consumerà il messaggio per poi effettuare lo stesso ciclo.

```
PetriNet sender, receiver, buffer;
Place sender{ready, waiting, hold_over},
      receiver{waiting, busy, hold_over};
Place buffer{msg_free, msg_busy, ack_free, ack_busy};

Transition sender{produce_msg, send_msg, receive_ack},
           receiver{receive_msg, send_ack, consume_msg};
Transition buffer{push_msg, pop_msg, push_ack, pop_ack};

// Modellazione processo sender
sender{hold_over -> produce_msg -> ready ->
       send_msg -> waiting -> receive_ack ->
       hold_over};
```

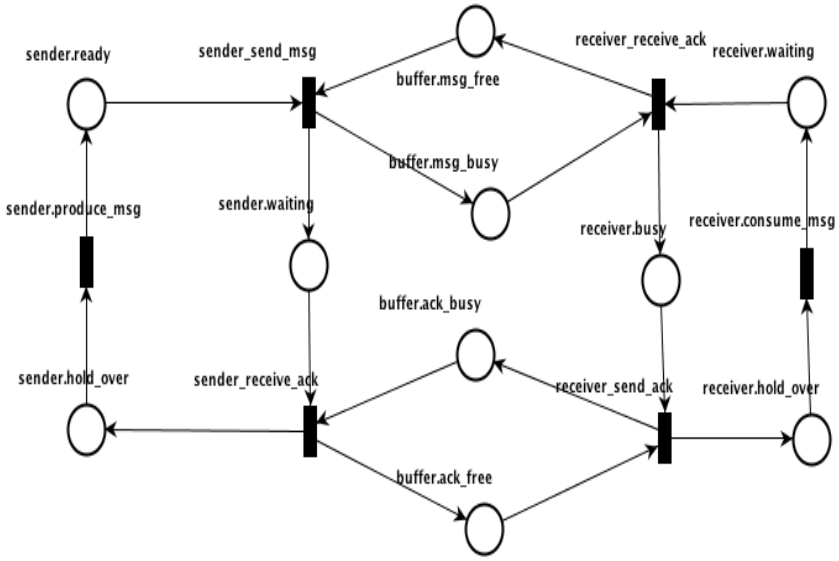



Figura 5.3: Classico esempio di modellazione di un canale di comunicazione

```
// Modellazione processo receiver
receiver{waiting -> receive_msg -> busy ->
         send_ack -> hold_over -> consume_msg ->
         waiting};
// Modellazione buffer
buffer{msg_free -> push_msg -> msg_busy -> pop_msg ->
       msg_free | ack_free -> push_ack -> ack_busy ->
       pop_ack -> ack_free};

#union_add_prefix = True;
s = (sender | receiver | buffer) on [send_msg = null =
         push_msg as sender_send_msg,
         receive_ack = null = pop_ack as sender_receive_ack,
         null = send_ack = push_ack as receiver_send_ack,
         null = receive_msg = pop_msg as receiver_receive_ack];

s::to_pnml_pipe("~/Desktop/send_rec.xml");
```

Il risultato della compilazione di tale codice è mostrato nell'immagine 5.3.

## 5.4 Problema lettori/scrittori

Il seguente esempio mostra come modellare la rete che descrive il problema dei lettori e scrittori. L'esempio rispecchia quanto svolto da Tadao Murata in [Mur89].

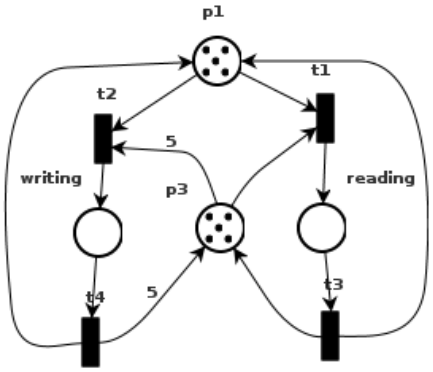



Figura 5.4: Modellazione del problema lettori/scrittori

```
PetriNet net;
Place net{reading(5), writing, p1(5), p3(5)};
Transition net{t1, t2, t3, t4};

net{p3 -> t1 -> reading -> t3 -> p3
   | p3 ->(5) t2 -> writing -> t4 ->(5) p3};
net{p1 -> t1 | p1 -> t2 | t3 -> p1 | t4 -> p1};

net{p1=5, p3=5};

net::to_pnml_pipe("~/reti/murata_writers_readers.xml");
```

Nella figura 5.4 si può vedere che il peso degli archi è stampato a video mentre non si vede la capacità dei posti. Con il software pipe è possibile verificare che la capacità viene inserita correttamente all'interno del file `pnml`.

## 5.5 Rete con contatore di cicli

Il seguente semplice esempio serve a mostare come gestire i posti di capacità illimitata. In questo caso un posto viene utilizzato come contatore dei cicli eseguiti dal sistema.

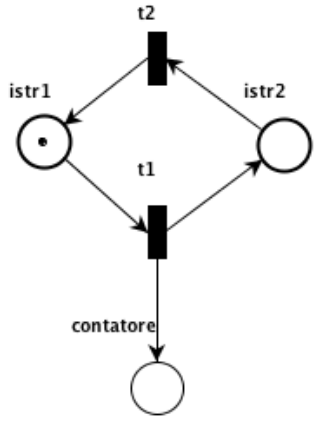



Figura 5.5: Rete con contatore

```
PetriNet net;
Place net{istr1, istr2};
Transition net{t1, t2};
#default_place_capacity=unlimited;
Place net{contatore};
net{istr1 = 1};
net{istr1 -> t1 -> contatore |
    istr1 -> t1 -> istr2 -> t2 -> istr1};

net::to_pnml_pipe("~/Desktop/cont.pnml");
```

## 5.6 Il problema dei cinque filosofi

Il seguente esempio mostra come si può modellare il classico problema dei cinque filosofi proposto da Edsger Dijkstra nel 1965.
Si cercherà di costruire l'esempio passo passo mostrando i risultati intermedi e facendo qualche considerazione sui risultati raggiunti.

La prima istruzione da fornire al sistema sarà quella di definire le reti che verranno utilizzate durante l'esercizio. A livello concettuale all'interno di questo problema esistono due componenti distinte: i filosofi e le forchette. Sarà comodo, come vedremo proseguendo l'esempio, dichiarare queste reti come facenti parte di due array.

```
PetriNet philo[5], fork[5];
```

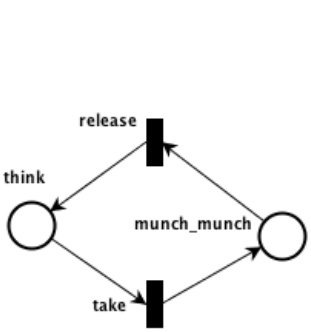
In questo momento il sistema ha creato 10 contenitori pronti a gestire posti e transizioni.
Riflettendo sulla componente che descrive il comportamento dei filosofi è sensato pensare che possa essere definita come una rete circolare che effettua due transizioni: pensa e prendi le forchette. I posti utili in questo momento sono solo quelli che descrivono lo stato temporaneo tra le due transizioni.
Verrà utilizzato il costrutto mostrato nella sezione 4.2.2 che permette di dichiarare i posti e le transizioni come appartenenti all'array di reti per poter aggiungere questi elementi ad ogni rete.

```
Place philo{ munch_munch, think};
Transition philo{ take, release};
```

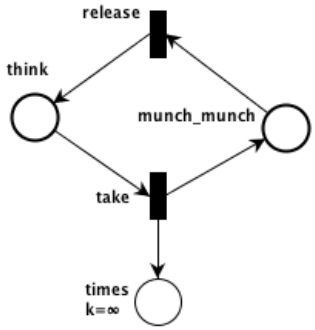
In questo momento *ogni* rete appartenente all'array `philo` contiene due posti e due transizioni slegate tra di loro. É quindi necessario ora creare gli archi che li collegano.
É interessante mostrare il fatto che non viene in questo momento dichiarata alcuna capacità per i posti delle reti `philo` e che quindi viene applicato il valore di default del sistema (capacità unitaria).

```
philo{ think -> take -> munch_munch -> release -> think};
```

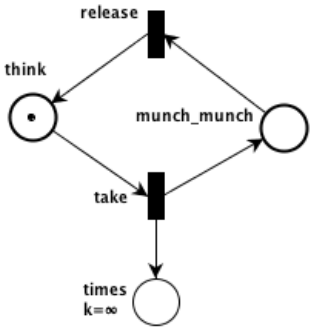
É possibile vedere una rappresentazione grafica di ogni rete appartenente all'array philo nell'immagine 5.6(a).




(a)




(b)




(c)

Figura 5.6: Modelli parziali del problema dei cinque filosofi relativi ai filosofi

Per poter mostrare un'altra funzionalità del linguaggio è stato scelto ora di modificare leggermente il problema da come descritto da Dijkstra.
Supponiamo che si voglia contare il numero di volte in cui ogni filosofo mangia. In questo caso è necessario creare un posto illimitato collegato ad esempio alla transizione take che tenga traccia dei cicli effettuati dalla rete. Aggiungendo il codice seguente:

```
#default_place_capacity=unlimited;
Place philo{times};
philo{ take -> times};
```

Si raggiunge il risultato desiderato che viene mostrato nella figura 5.6(b).

Passiamo ora a considerare le reti che definiscono i comportamenti delle forchette.
Queste reti contengono un solo posto (con capacità unitaria) che descrive la presenza o meno della forchetta sul tavolo. Oltre a questo posto devono essere presenti anche due transizioni che permettono di aggiungere e togliere la marca dal posto.
Prima cosa sarà necessario ripristinare la capacità di default assegnata ai posti dopodiché si passerà a dichiarare i posti e le transizioni delle componenti.

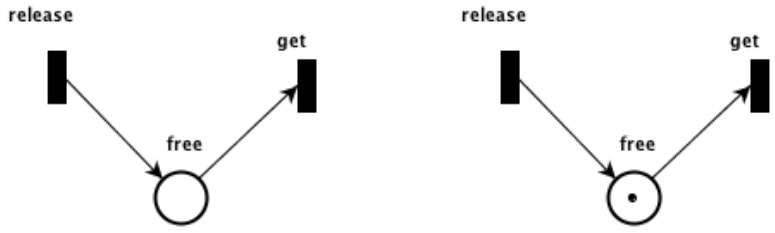
```
#default_place_capacity=1;
Place fork{free};
Transition fork{get, release};
fork{free -> get | release -> free};
```

La rappresentazione grafica di questa rete è mostrata nella figura 5.7(a).

In questo momento sono state definite tutte le reti necessarie, è ora possibile definire la marcatura iniziale di ogni componente. Le rappresentazioni grafiche sono mostrate nelle figure 5.6(c) e 5.7(b).

```
philo{ think=1};
fork{ free=1};
```

É ora necessario unire ogni filosofo con le forchette adiacenti. É stato per comodità deciso che il filosofo di posto `i` ha come forchette vicine quelle di indice `i` e `i+1`. Per poter distinguere le reti `fork` con le quali ogni rete




(a) (b)

Figura 5.7: Modelli parziali del problema dei cinque filosofi relativi alle forchette

`philo` interagisce è necessario aggiungere il prefisso ad ogni elemento che viene aggiunto nella nuova rete altrimenti i posti di nome `free` vengono sovrapposti scorrettamente.
Per comodità ogni rete creata viene riassegnata alle reti che descrivono i filosofi.

```
#union_add_prefix=True;
for(i=0;i<5;i+=1){
  philo[i] = (philo[i] | fork[i] | fork[i+1 % 5] )
   on [take = get = get as philo[i].take,
       release = release = release as philo[i].release ];
 }
```

Come mostrato in figura 5.8 le reti vengono unite correttamente, vengono aggiunti tutti i prefissi e non ci sono sovrapposizioni non volute.
Ultimo passo per raggiungere quanto desiderato è l'unione di tutte le nuove reti. In questo caso non è necessario aggiungere i prefissi in quanto già presenti e non è necessario dichiarare su quali elementi effettuare la sovrapposizione in quanto i posti con nomi uguali sono concettualmente (e correttamente) gli stessi.
Scrivendo quindi le seguenti istruzioni:

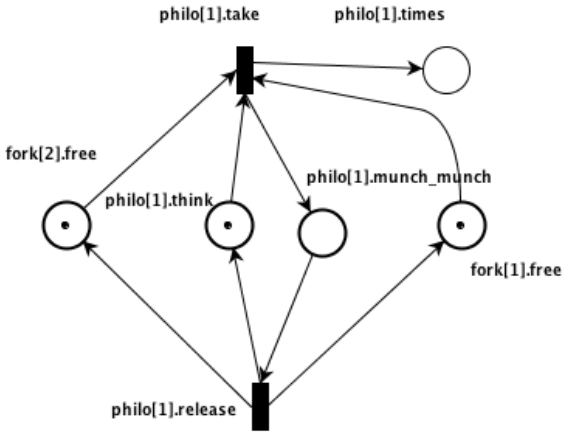



Figura 5.8: Rappresentazione grafica delle interazioni tra il filosofo 1 e le forchette adiacenti

```
#union_add_prefix=False;
sys = (philo[0] | philo[1] | philo[2] | philo[3] | philo[4]) on [ ];
```

Si ottiene quanto mostrato in figura 5.9.

Viene ora mostrato il codice completo e leggermente compattato.

```
PetriNet philo[5], fork[5];
Place philo{munch_munch, think}, fork{free};
Transition philo{take, release}, fork{get, release};

#default_place_capacity=unlimited;
Place philo{times};

philo{think=1};
fork{free=1};

philo{think -> take -> munch_munch -> release -> think
          | take -> times};
fork{free -> get | release -> free};
```

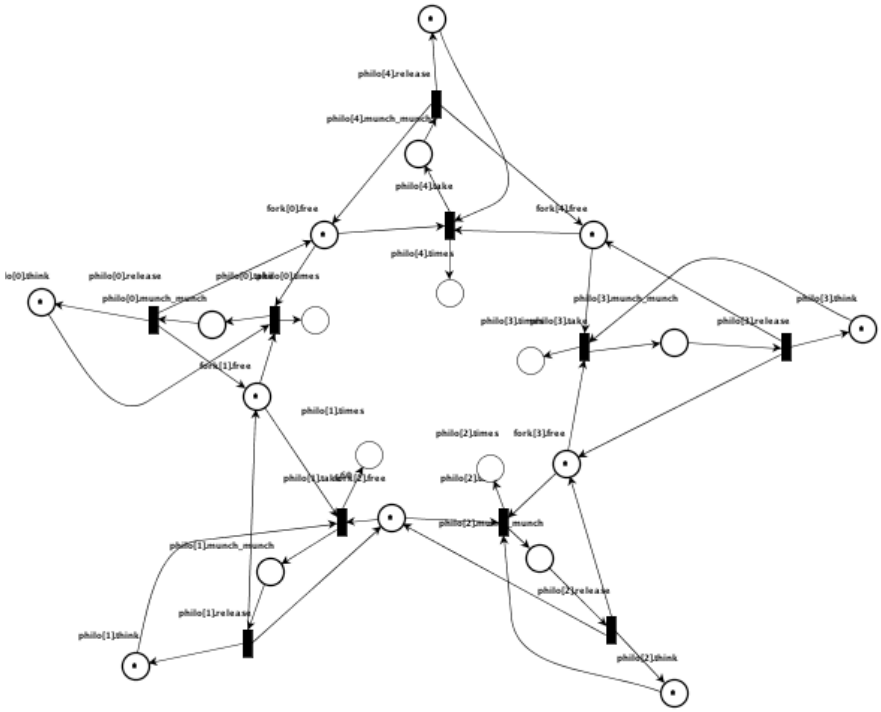



Figura 5.9: Modellazione del problema dei 5 filosofi

```
#union_add_prefix=True;
for(i=0;i<5;i+=1){
  philo[i] = (philo[i] | fork[i] | fork[i+1 % 5] )
   on [take = get = get as philo[i].take,
       release = release = release as philo[i].release ];
 }

#union_add_prefix=False;
sys = (philo[0] | philo[1] | philo[2] | philo[3] | philo[4]) on [ ];
```

# Bibliografia


[Fag10] Sara Faggioni. Strumenti Testuali e grafici per la specifica di reti di Petri. Master's thesis, Universitá degli Studi di Milano Bicocca, 2010.

[Kin06] Ekkart Kindler. Concepts, Status, and Future Directions. *Entwurf Komplexer Automatisierungssysteme (EKA)*, 9:35–55, 2006.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Py] Python. http://www.python.org.

[Ros00] Guido Von Rossum. http://www.python.org/dev/peps/pep-0008, 2000.

[Sum09] Mark Summerfield. *Programming in Python3*. Addison Wesley, 2009.

[WK03] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. *Petri Net Technology for Communication-Based Systems – Advances in Petri Nets*, 2472:122–144, 2003.