Strumenti testuali e grafici per la specifica delle reti di Petri

Marco Previtali

Indice

Capitolo 1

Attuale stato dell'arte

1.1 Linguaggio PetriNet

PetriNet è un linguaggio di specifica testuale per le reti di Petri sviluppato da Sara Faggioni. 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.

1.1.1 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 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 degli stati con i nome di una marcature:

```
marcatura_a [transizione_x> marcatura_b
```

Operazioni sulle reti

net.toDot(nomeFile, ext) crea il file dot della rete su cui viene chiamto 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 su quali posti o transizioni devono essere unite 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 una transizione e un posto, l'unione non li prende in considerazione.

1.2 Strumenti

1.2.1 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 degli 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.

1.2.2 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.

1.3 Utilizzo dei moduli

1.3.1 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 documentazinoe di questo modulo richiamando il comando

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

all'interno dell'interprete.

1.3.2 Parser

Un altro modo per definire una rete di Petri è utilizzando il liguaggio di specifica definito in 1.1.1. 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 el parser che mostrerà un ambiente di questo genere:

```
$ python PetriNet_parser.py

petriNet >
```

Da questo momento il parser accetterà comandi strutturati come percedentemente 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.

1.4 GraphViz

Per poter visualizzare facilmente le reti create, il software PetriNet si basa sulle funzionalità fornite da GraphViz[1].
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.

1.5 PNML

PNML[2] è un formato di documenti basato su XML ed è stato inventato 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 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.

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

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 1.1: Elementi grafici PNML

1.5.1 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 oggetti all'interno di una rete di Petri ha un identificatore univoco.

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

Classe	Elemento XML	Attributi XML
PetriNet Doc	`<pnml>`	
PetriNet	`<net>`	`id:ID` `type:anyURL`
Place	`<place>`	`id:ID`
Transition	`<transition>`	`id:ID`

Arc	`<arc>`	`id:ID`
		`source:IDRef (Nodo)`
		`target:IDRef (Nodo)`

Page	`<page>`	`id:ID`
RefPlace	`<referencePlace>`	`id:ID` `ref:IDRef`
RefTrans	`<referenceTransition>`	`id:ID` `ref:IDRef`
ToolInfo	`<toolspecific>`	`tool:String` `version:String`
Graphics	`<graphics>`	

Tabella 1.2: Elementi base PNML

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.

Elemento Base	Sottoelementi di `<graphics>`

Nodo, Page	`<position>`
	`<dimension>`
	`<fill>`
	`<line>`

Arc	`<position>` (zero o più) `<line>`

Annotation	`<offset>`
	`<fil>`
	`<line>`
	`<font>`

Tabella 1.3: Possibili tag dell'elemento `<graphics>`

1.5.2 Sintassi

Viene ora mostrata una panoramica sui principali elementi del formato PNML. Nella Tabella 1.2 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.

1.5.3 Graphics

Tutti gli oggetti e tutte le etichette possono avere delle informazioni grafiche. La Tabella 1.3 mostra i possibili figli della sezione `<graphics>` a seconda dell'elemento base in cui viene inserita mentre la Tabella 1.1 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 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 [WK] o [Kin06].

Capitolo 2

Analisi di PetriNet

Nel seguente capitolo si analizzerà il linugaggio PetriNet e il funzionamento dei moduli sviluppati precendetemente evidenziandone errori e accenando possibili soluzioni

2.1 Linguaggio PetriNet

Nella seguente sezione viene analizzato il linguaggio PetriNet come sviluppato finora.

2.1.1 Classe di reti trattate

PetriNet permette la specifica di reti di occorrenze 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.

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

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 altrenativo 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`.

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

coseguenza, 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.

2.1.3 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 2.1 il nome del

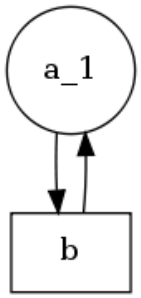

Figura 2.1: Esempio di errore nella generazione della rete

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;
```

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 2.2, un grafo nel quale compaiono due posti con lo stesso identico nome.

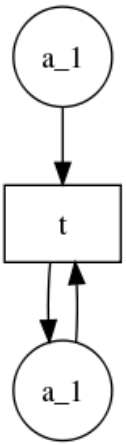

Figura 2.2: Rete con due posti con lo stesso nome

2.1.4 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` a 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 2.3.

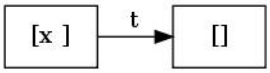

Figura 2.3: Grafo dei casi errato

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.

2.1.5 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.

2.2 Moduli Python

In questa sezione verranno analizzati i moduli python sviluppati.

2.2.1 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.

2.2.2 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.

[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 averà che `eval(repr(x)) == x`.

Obj

La classe `Obj` è una superclasse dalla quale erediteranno sia le classi `State` e `Trans`. Contiene al suo interno siolemente 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.

State

State è la classe rappresentante uno stato all'interno del modulo PetriNet.
State aggiunge alla classe `Obj` una sola variabile `token` e il metod `__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 classse 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 utilizate 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. Prendiamop come esempio i seguenti codici che effettuano una ricerca all'interno di una struttura dati e stamapano 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.

2.2.3 PetriNet_lex.py

Questo modulo si occupa della suddivisione in token del codice basandosi sul modulo `lex.py` di `ply`[3].
Il modulo è sostanzialmente corretto anche se vengono definiti token non utilizzati nel linguaggio (nella fattispecie `in` e `to`).

2.2.4 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

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

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 3

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.

3.1 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 2.1.

3.1.1 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 3.2.3 e 3.2.5 per una trattazione migliore dell'argomento) che collegano i vari elementi della rete.

3.1.2 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 2.1.2.

3.1.3 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
```

3.1.4 Creazione del grafo delle marcature

Nell'implementazione percedente di PetriNet era possibile creare il grafo delle marcature della rete definita. Dato che questo è un aspetto di simulazione mentre l'obiettivo di questo sistema è quello di fornire un linguaggio di specifica per la rete, dato che l'implementazione della creazione di grafi delle marcature era errata e quindi sarebbe stato necessario riscrivere anche questo pezzo di codice è stato deciso di eliminare queste 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.

3.1.5 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 precendetemente 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;
```

3.1.6 Marcatura iniziale

Il modo di definire una marcatura in PetriNet è stato modificato totalmente per poter essere utilizzabile con reti che hanno posti con capacità maggiore di

[1] Si comprenderà meglio la sintassi della dichiarazione dei posti nella sezione 3.2.3
[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/

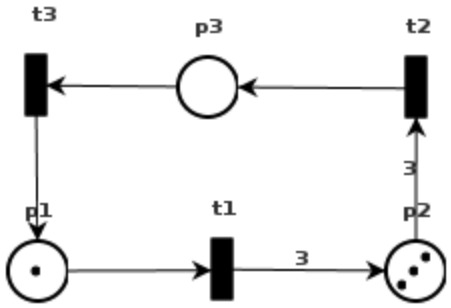

Figura 3.1: Esempio di semplice rete con marche

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 create una rete contenente 3 posti (`p2] con capacità 5` e 3 transizione. Successivamente viene definito il flusso della rete descrivendo una rete circolare e dopodiché viene istanziata la marcatura iniziale della rete con il codice `netp1=1, p2=3` che inserisce una marca in `p1` e 3 marche in `p2`. Una rappresentazione di questa rete è mostrata in figura 3.1. Ovviamente, nel caso si cercasse di inserire una quantità di marche maggiore alla capacità di un posto, il sistema fornirà un errore.

3.2 Nuove funzionalità di PetriNet

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

3.2.1 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 banalmente 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 3.2.6 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 comportament 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 sarà comunque limitata al numero intero massimo gestibile dalla versione di python sottostante al sistema.

3.2.2 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];
```

[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

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

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:

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

Si può fin da ora notare come ci sia già un marcato guadagno in lunghezza e chiarezza del codice derivante da questa nuova funzionalità dovuto anche alla sola dichiarazione di reti (precedentemente sarebbero dovute venir dichiarate una per una tutte le reti).

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 sue reti simili. Per questo è possibile aggiungere posti, transizioni e archi di flusso ad ogni singola rete appartentente 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};
[...]
```

[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

```
philosophers[0]{get_ready -> times};
[...]
```

aggiunge 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 questo componente.

3.2.3 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` contentente 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 appartenti a reti diverse sulla stessa riga nel seguente modo:

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

- Il secondo metodo è dichiarando 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 a 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 3.1.1 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 rappresente una collezione di posti uguali e sarebbe poco logico averne con capacità diverse.

3.2.4 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;
```

3.2.5 Definizione flussi

La sintatssi per la definizione di flussi 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 comportamente di ogni componenete sarà facilmente intuibile. È possibile definire più flussi in due modi:

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

- "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};
```

In entrambi i casi il risultato sarà quello riportato in figura 3.2.

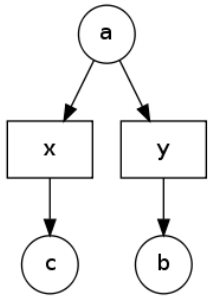

Figura 3.2: Rete semplice generata con doppio inserimento di flusso

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 3.3.

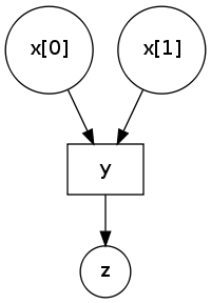

Figura 3.3: Rete semplice generata con doppio inserimento di flusso

Archi pesati

Come detto nella sezione 3.1.1 il sistema è stato portato a definire classi 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)};
Transition net{consuma_token, consuma_2_token};
net{ buffer -> consuma_token};
net{ buffer ->(2) consuma_2_token};
```

crea la rete presentata in figura 3.4 nella quale si vede che l'arco tra il posto `buffer` (di capacità 8) e la transizione `consuma_2_token` ha peso 2.

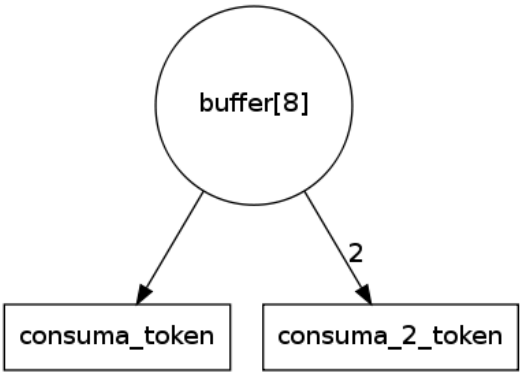

Figura 3.4: Rete con archi pesati

3.2.6 Unione di reti

Per migliorare la comprensibilità di un codice scritto in linguaggio PetriNet è possibile creare delle reti che descrivano 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 = emph(*lista-reti*) on [*lista-elementi-unione-1*,*lista-elementi-unione-2*, ... *lista-elementi-unione-n*,];

dove:

lista-reti = *nome-rete-1* — *nome-rete-2* — ... *nome-rete-n*

lista-elementi-unione = *elemento-rete-1* = *elemento-rete-2* = ... = *elemento-rete-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 appartente 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 esistano elementi con lo stesso nome essi verranno trattati come uno solo nell'unione e, di conseguenza, verranno collegati tutti gli elementi delle reti unite.
É comunque sconsigliato creare reti partendo da modelli nei quali esistano elementi con lo stesso nome che non verranno uniti.
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 prendano 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.

3.2.7 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 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 cilo e *incrementi* sono le istruzioni da eseguire ogni volta che un ciclo termina.

3.3 Moduli Python

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

3.3.1 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.

3.3.2 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

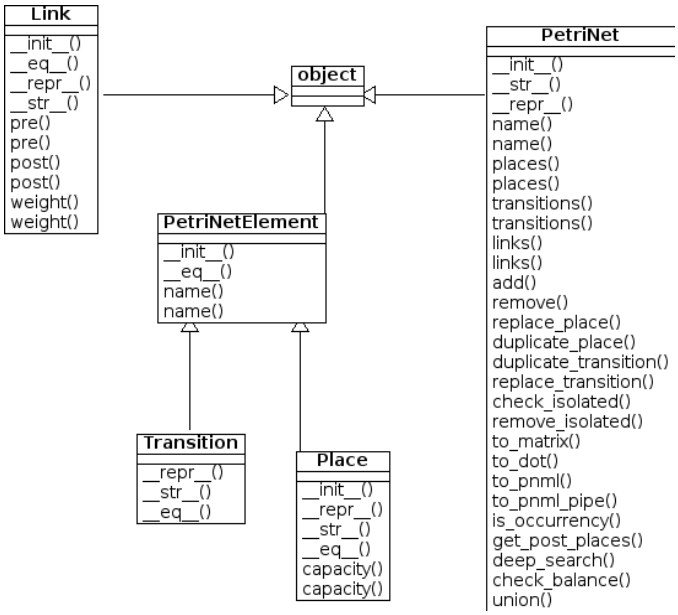

Figura 3.5: Diagramma UML delle classi Python sviluppate

nell'immagine 3.5.
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.

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 gettere 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 3.1.4.

3.3.3 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 seguenti regole:

`'|'` per consentire di riconoscere l'`or` nella descrizione di flussi1

`'COMMENT'` per riconoscere i commenti all'interno del codice (si veda la sezione 3.1.5)

`'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 3.1.2)

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

`'for'` in quanto inutilizzato anche precedentemente e non ancora implementati i cicli

`'to'` in quanto inutilizzato

`'in'` in quanto inutilizzato

3.3.4 PetriNet_parser.py

Questo modulo si occupa della definizione della grammatica formale del linguaggio analizzando la sequenza di token trasmessa del 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[11].
Vengono successivamente dichiarate le regole per la dichiarazione di reti, posti e transizioni nelle modalità viste nelle sezioni precedenti.
Particolamente 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.

3.3.5 pnpre.py

3.3.6 pnc

[11] Esisite 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

Capitolo 4

Esempi di utilizzo

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

4.1 Processo sequenziale ciclico

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

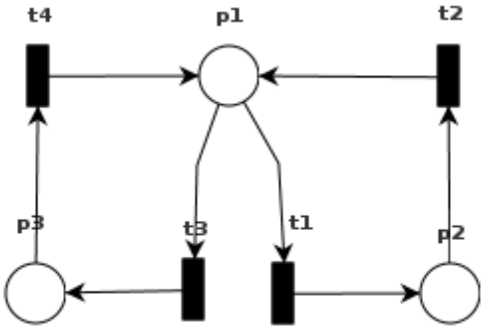

Figura 4.1: Processo sequenziale che presenta concorrenza sul posto p1

```
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 4.1[1].

4.2 Processi concorrenti su di 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 4.2.

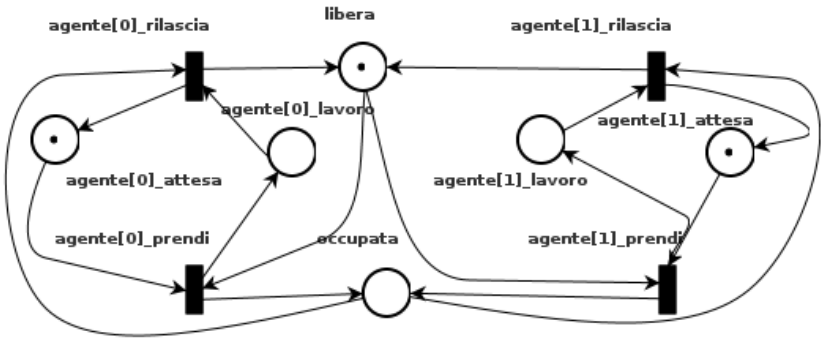

Figura 4.2: Rete che descrive due processi che concorrono per l'utilizzo di una risorsa

```
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};

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

agente[0] = union(agente[0], risorsa) on [prendi = prendi, rilascia = rilascia];
agente[1] = union(agente[1], risorsa) on [prendi = prendi, rilascia = rilascia];

sys = union(agente[0],agente[1]) on [libera = libera, occupata = occupata];

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

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

4.3 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 rieffettuare lo stesso ciclo.
Il processo `receiver` attenderà di ricevere un messaggio, invierà l'ack e consumerà il messaggio per poi effettuare lo stesso ciclo.

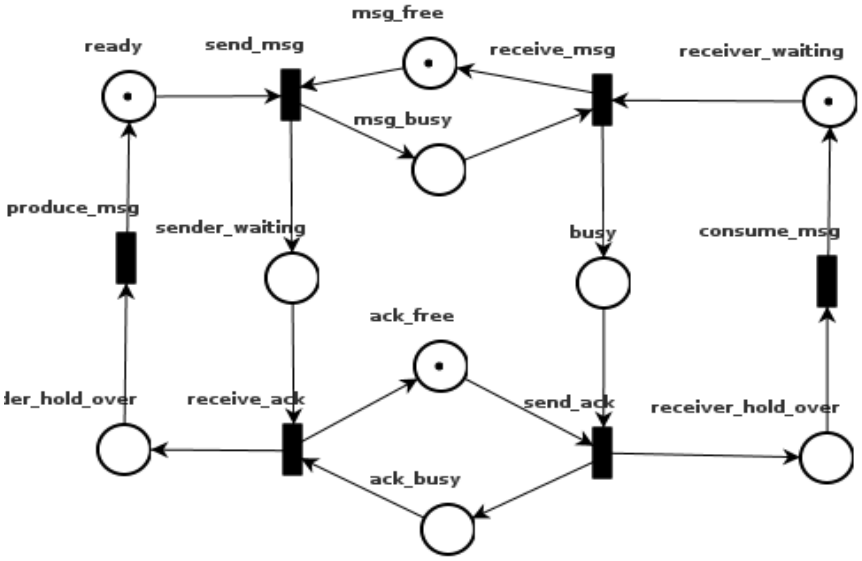

Figura 4.3: Classico esempio di modellazione di un canale di comunicazione

```
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};
```

```
// 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};

s = union(sender, receiver);
s = union(s, buffer) on [send_msg = push_msg, receive_ack = pop_ack,
send_ack = push_ack, receive_msg = pop_msg];

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

Il risultato della compilazione di tale codice è mostrato nell'immagine 4.3.

4.4 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 [Mur88].

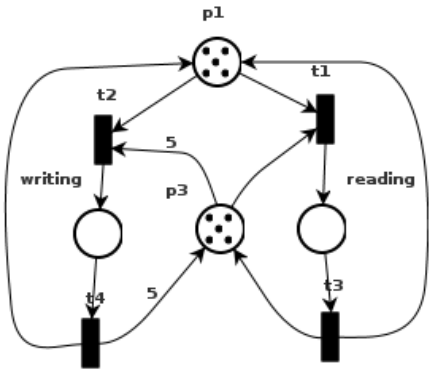

Figura 4.4: Modellazione del problema lettori/scrittori

```
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");
```

Nalla figura 4.4 si può vedere 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`.

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 directionse. 2006.

[Mur88] Tadao Murata. petri nets: Properties, analysis and applications. 1988.

[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.

[WK] Michael Weber and Ekkart Kindler. the petri net markup language.