UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

LAUREA IN INFORMATICA

Traduzioni di Formato e Unfolding Parziale di Reti di Petri Simmetriche

Relatore:
prof. Lorenzo Capra

Correlatore:
prof. Massimiliano De Pierro

Autore:
Abdelrahman Sobh

Matricola: 958506

Milano, a.a 2020/2021

Abdelrahman Sobh

Traduzioni di Formato e Unfolding Parziale di Reti di Petri Simmetriche
Tesi di Laurea, 16 luglio 2021
Relatore:
prof. Lorenzo Capra

Facoltà di Scienze e Tecnologie
Dipartimento di Informatica "Giovanni degli Antoni" Via Celoria 18 - 20133 Milano

Ringraziamenti

Ringrazio prof. Lorenzo Capra e i suoi colleghi (prof. Massimiliano De pierro & prof.ssa Giuliana Franceschinis) per tutto lo sforzo che è stato svolto durante il tirocinio e la tesi di laurea aiutandomi per arrivare a questo punto di raggiungimento. Ringrazio inoltre tutti i professori dell'ateneo che ci hanno insegnato le materie in modo efficace per capire gli argomenti in modo abbastanza chiaro.

Indice

Elenco delle figure

1. Introduzione

L'argomento principale della tesi ha riguardato lo sviluppo di una libreria software Java per la traduzione di formato di modelli di sistemi concorrenti specificati mediante un formalismo standard, una classe di reti di Petri di alto livello dette Reti Simmetriche, o SN.

Il formalismo SN è caratterizzato da una sintassi compatta e strutturata che permette di evidenziare attraverso le annotazioni di un modello le simmetrie presenti in sistemi distribuiti formati da gruppi di componenti dal comportamento simile. Tali simmetrie sono sfruttate in fase di analisi, sia attraverso la generazione di uno spazio di stati aggregato/simbolico (cui è associato un processo stocastico nella versione temporizzata delle SN), sia mediante tecniche strutturali efficienti.

L'obiettivo della tesi è favorire e migliorare l'interoperabilità fra i diversi strumenti automatici di supporto disponibili per l'analisi dei modelli SN, al fine di contribuire ad estendere l'usabilità stessa del formalismo delle reti SN.

Per garantire la più ampia usabilità, la descrizione iniziale di un modello SN è espressa nel formato XML standard per le reti di Petri, noto come PNML. Tale formato costituisce l'input per il modulo principale di traduzione, che poi opera secondo due modalità. Nella prima modalità viene effettuata la traduzione di una rete SN in formato PNML in oggetti software compatibili con un pacchetto (composto da una libreria con un API ben definita ed una GUI) per l'analisi strutturale simbolica di modelli SN (`SNExpression`). Nella seconda, il modello SN iniziale viene trasformato in uno equivalente (sempre espresso in forma PNML) operando una procedura di "unfolding" parziale/simbolico, che rende ad esempio possibile analizzare una vasta classe di modelli SN stocastici mediante la risoluzione di sistemi equazioni differenziali ordinarie (espresse in forma simbolica/compatta).

La duplice funzionalità del traduttore segue un approccio modulare. La rappresentazione ad oggetti di una rete SN che si ottiene come risultato della prima fase di traduzione, può essere infatti passata in modo indipendente al modulo che implementa l'unfolding parziale, il quale genera un file PNML che rappresenta la rete SN corrispondente all'unfolding.

Il software è stato sviluppato per permettere una agevole integrazione con `GreatSPN`, l'editor/simulatore grafico nativo per reti SN. Uno dei contributi del lavoro di tesi è stato l'uso del formato PNML standard, che supporta la definizione strutturata di tutti gli elementi della sintassi di una rete SN, alcuni dei quali invece, come il partizionamento e/o l'ordinamento circolare delle classi di colori, sono descritti in modo puramente testuale dall'interfaccia grafica di `GreatSPN`. Un modulo della libreria permette quindi il passaggio dal formato standard PNML per reti SN al formato proprietario di `GreatSPN` (pnpro) in modo del tutto automatico.

Il processo di traduzione dal formato PNML di una rete simmetrica ad una rappresentazione ad oggetti di alto livello è complesso e si compone di diverse fasi, ognuna delle quali produce in output un formato intermedio che viene passato come input alla fase successiva.

L'intero processo è descritto in Figura 1.1, dove sono evidenziate le singole fasi e l'output prodotto da ciascuna. La libreria software è stata sviluppata in modo altamente modulare, per consentirne un agevole uso e manutenzione. Nel progetto di tesi sono state implementate le quattro fasi principali di un classico traduttore/compilatore. Ognuna di tali fasi, corrispondente a un modulo del traduttore, è stata adattata per rispondere adeguatamente ai requisiti e alle caratteristiche particolari del dominio applicativo.

La prima fase della traduzione consiste nel parsing (lessicale) di un file PNML e produce in output delle liste contenenti delle descrizioni in forma pseudo-testuale dei diversi elementi di una rete SN (i nodi della rete, cioè i posti e le transizioni, le classi di colori, i domini dei nodi, le annotazioni presenti sugli archi della rete) corrispondenti ai diversi tag PNML. In questa fase è stata usata una libreria Java esistente (org.w3c) per scansionare i tags del file PNML.

Durante la seconda fase i dati presenti in forma testuale nelle liste generate durante la prima fase sono analizzati lessicalmente, per ottenere una rappresentazione ad oggetti degli elementi base (o token) della definizione di una rete SN, quali le classi di colori e le variabili che compaiono nelle annotazioni della rete (le guardie delle transizioni, e le funzioni sugli archi). Ad esempio, le variabili sono tradotte in oggetti di tipo `Projection` della libreria `SNExpression`. Questi oggetti sono passati alla terza fase per una successiva e più articolata analisi semantica. Da un punto di vista concettuale, l'output di questa seconda fase è un albero sintattico astratto (AST).

La fase successiva di analisi semantica produce gli oggetti software più complessi, corrispondenti ai domini e alle guardie delle transizioni, ed alle annotazioni (funzioni) sugli archi, attraverso una manipolazione degli oggetti di primo livello (AST) generati dalla fase precedente. Come risultato finale di questa fase, si ha la descrizione finale dell'intera rete SN come oggetto composito.

Una volta ottenuta la descrizione a oggetti di una rete SN, è possibile applicarvi l'algoritmo di unfolding parziale che genera un'altra rete simmetrica del tutto equivalente (come comportamento) a quella di partenza. L'output, sia del modulo principale di traduzione sia del modulo di unfolding, può essere esportato in formato PNML.

Come già ricordato, è inoltre possibile passare dal formato PNML al formato proprietario del tool grafico `GreatSPN`. Questo dà la possibilità di visualizzare la rete ottenuta in output e confrontarla con la rete iniziale descritta nello stesso formato.

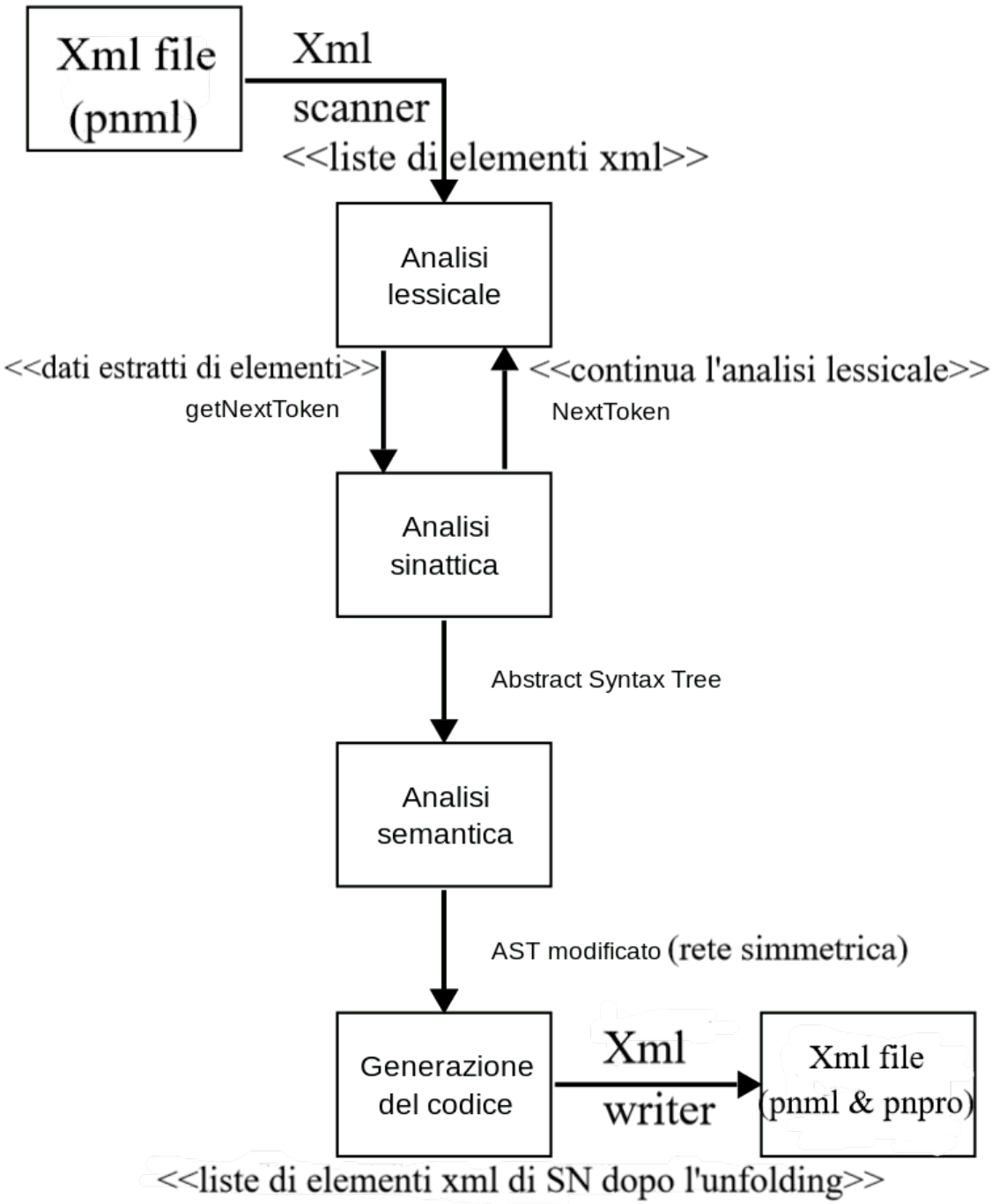

Figura 1.1: Fasi del traduttore SN.

2. Formalismi e strumenti utilizzati

In questo Capitolo sono richiamate le principali nozioni che riguardano i formalismi alla base del lavoro di tesi. Questi sono le Reti di Petri (PN), le reti di Petri Colorate Simmetriche (SN) e il linguaggio di descrizione Petri Net Markup Language (PNML).

2.1 Reti di Petri

Le Reti di Petri [5] appartengono alla famiglia dei *grafi orientati bipartiti*. I due sottoinsiemi dei vertici del grafo indicati rispettivamente con P e T che costituiscono la bipartizione sono chiamati insieme dei *posti* ed insieme delle *transizioni* della rete. Graficamente il nodo *posto* è rappresentato con la forma di "cerchio" mentre il nodo *transizione* è rappresentato con la forma di "rettangolo".

Posti e transizioni sono connessi da archi orientati pesati. Nelle PN si identificano tre tipi di archi: archi detti di *input*, escono da un posto ed entrano in una transizione; archi detti di *output*, escono da una transizione ed entrano in un posto; archi detti di *inibizione* escono da un posto ed entrano in una transizione. I rispettivi insiemi sono indicati secondo l'ordine con le lettere I, O, H.

I posti di una PN possono contenere dei *token*. Il mapping che associa ad ogni posto il numero di token contenuti è chiamato *marcatura* della rete.

Formalmente, una PN è una tupla così formata:

$$N = \langle P, T, I, O, H, \mathbf{m}_0 \rangle$$

dove $\mathbf{m}_0$ è la marcatura *iniziale* della rete.

In Fig. 2.1 è mostrato un esempio di PN [13]. Essa è un modello preso dall'area della *reliability*, la sua semantica verrà descritta in seguito. Tale rete ha 3 posti e 4 transizioni. In questa versione delle PN le transizioni possono essere effettivamente di due tipi, temporizzate (rettangolo più spesso nel disegno) ed immediate (rettangolo più sottile nel disegno). La rete descrive un sistema composto da due parti funzionali entrambe necessarie alla operatività del sistema. Ogni parte funzionale può guastarsi ed andare in riparazione, in questo frangente la parte rimasta funzionante entra in stand-by in attesa che la parte guastatasi sia riparata.

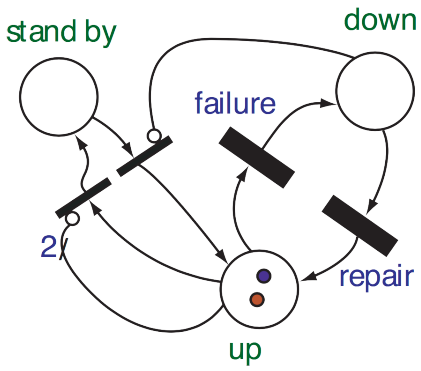

Figura 2.1: Esempio di Rete di Petri ordinaria.

Semantica di una PN

Le PN sono utilizzate come linguaggio per descrivere la struttura e il *comportamento* dei sistemi distribuiti.

In un modello PN la marcatura di tutti i posti descrive uno stato del sistema. Ad esempio, nella rete di Fig. 2.1 la marcatura del posto up descrive il numero di *parti funzionali* del sistema attive e funzionanti al tempo corrente. Essa costituisce una parte dello stato globale perché riguarda solo il posto up. Nella marcatura iniziale, rappresentata nel disegno, le 2 parti sono entrambe attive e funzionanti.

Le transizioni descrivono *eventi* la cui occorrenza, detta *scatto* nella terminologia PN, modifica lo stato del sistema. Una transizione connessa con archi in I o in O può modificare solo la marcatura dei posti ad essa adiacenti e quindi operare una modifica localizzata dello stato globale. Il compor-

tamento di un modello PN nel tempo è descritto secondo la semantica di *scatto* delle transizioni e di come queste modificano lo stato del sistema, quindi la marcatura:

- una transizione t è *abilitata* se i suoi posti di input contengono un numero di token che soddisfa ($\geq$) il peso riportato sui rispettivi archi;
- t può scattare se e solo se è abilitata;
- lo scatto di t produce un cambiamento di stato: **1.** rimuovendo da ogni posto di input un numero di token pari al peso del rispettivo arco in I; **2.** aggiungendo in ogni posto di output un numero di token pari al peso del rispettivo arco in O;
- gli archi inibitori in H inibiscono lo scatto di una transizione abilitata. Un arco inibitore tra il posto p e la transizione t con peso n inibisce lo scatto di t finquando la marcatura di p ha un numero di token maggiore o uguale a n.

Tornando all'esempio di Fig. 2.1 nella marcatura iniziale solo la transizione $failure$ è abilitata, infatti la transizione verso lo stato $stand\text{-}by$ è inibita. Lo scatto di $failure$ rimuove un token da up e produce un token in $down$ a modellare il guasto di una delle due unità funzionali del sistema. Il nuovo stato può essere rappresentato nel seguente modo: $\mathbf{M}_1 = 1.up + 1.down$. Da $\mathbf{M}_1$ il sistema evolve attraverso lo scatto della transizione immediata $sleep$ nello stato $\mathbf{M}_2 = 1.stand\text{-}by + 1.down$. L'arco inibitore tra $down$ e $wake\text{-}up$ inibisce lo scatto di $wake\text{-}up$ in $\mathbf{M}_2$ finché l'unità funzionale guasta è in riparazione.

2.2 Le Reti Simmetriche

L'introduzione delle reti di Petri di alto livello (HLPN), come le Colored Petri Net (CPN) e le Symmetric Net (SN), è stata cruciale se si tiene presente la forza espressiva di questa nuova classe di formalismi di modellazione. La possibilità di associare ai token delle informazioni e di parametrizzare lo scatto delle transizioni ha reso possibile una rappresentazione concisa di sistemi che avrebbero richiesto altriemnti reti ordinarie molto estese.

Le reti colorate Simmetriche, precedentemente nominate reti ben formate (WN) ([3]) sono sostanzialmente identiche alle CPN dal punto di vista della potenza espressiva. Tuttavia, la definizione sintattica del formalismo SN porta a nuovi algoritmi di analisi più efficienti sulla base del concetto di marcatura simbolica.

Nel seguito di questa sezione diamo una descrizione informale del formalismo SN. La sintassi utilizzata nella spiegazione e negli esempi è quella accettata dallo strumento di analisi prototipo che è stato sviluppato presso il Dipartimento di Informatica dell'Università di Torino ed integrato nel tool GreatSPN.

Come nelle PN, i posti delle SN insieme alla loro marcatura svolgono il ruolo di descrivere lo stato del sistema mentre le transizioni rappresentano gli eventi che causano i cambiamenti di stato. Nelle SN ai token si possono associare alcune informazioni, anzi un token può essere considerato come una istanza di una struttura dati con un certo numero di campi la cui semantica dipende dal *posto* al quale il token appartiene.

La definizione del "tipo di dato" associato ad ogni "posto" si chiama "dominio di colore di posto". I tipi di dati dei campi sono selezionati da un insieme di tipi di base e si chiamano "classi di colore di base". La specifica delle classi di colore di base fa parte della definizione della rete. Nel nostro strumento, le classi di colore di base hanno sempre dimensione finita e sono specificate per enumerazione degli elementi.

Spesso può essere utile partizionare una classe di colore di base in sottoclassi disgiunte di oggetti che posseggono proprietà comuni. Ad esempio, la classe di processi potrebbe essere partizionata in due sottoclassi: la sottoclasse dei processi a bassa priorità e la sottoclasse di processi ad alta priorità. Nella terminologia SN queste sono chiamate sottoclassi statiche.

Una classe base può essere ordinata in modo da definire una funzione successore sui suoi elementi. Si assume che l'ordinamento sia circolare.

In SN, un posto può essere usato per rappresentare il valore di una variabile di tipo dato purché non contenga mai più di un token (il posto vuoto potrebbe rappresentare sia una variabile non inizializzata sia un valore predefinito fisso nel dominio variabile) e il dominio del colore del posto è uguale al tipo di dati variabili. Un altro uso dei posti nei modelli SN è per la rappresentazione dello stato di un multi-insieme di possibili distinguibili oggetti. La notazione M(p) denota la marcatura di posto "p", il multi-insieme di C(p) contenuto in "p" secondo alla marcatura M. Le transizioni nelle SN possono essere considerate procedure con parametri formali. I parametri formali si chiamano "il dominio del colore di transizione"; la loro dichiarazione fa parte della descrizione di rete e il tipo associato a ciascun parametro deve essere una classe di colore di base. Un dominio di colore di transizione è definito nello stesso modo di un dominio del colore di posto. L'elenco delle classi nel dominio del colore definisce il tipo associato alla transizione parametri. Il dominio del colore di una transizione t (indicato con C(t)) è vincolato dai domini di colore del suo input, inibitore e posti di uscita. Vedremo più avanti che la relazione tra i domini di colore di transizione

e posto sono definiti attraverso l'arco espressioni. Una transizione i cui parametri formali sono stati istanziati ai valori effettivi si chiama istanza di transizione. Noi usiamo il notazione [t,c] per un'istanza di transizione t ,dove c rappresenta l'assegnazione dei valori reali ai parametri di transizione. Si osservi che un assegnamento sia in realtà un elemento dell'insieme C(t) e per questo motivo è spesso indicato come una " istanza di colore" di t.
Per attivare una transizione, è necessario specificare valori effettivi per i suoi parametri formali (è simile alla esecuzione di una chiamata di procedura), cioè possiamo solo attivare la transizione di istanze. Il controllo di abilitazione di un'istanza di transizione e il cambiamento di stato causato dalla sua accensione dipende (di nuovo) dalle espressioni di archi che etichettano gli archi collegati alla transizione. Si osservi che molte istanze della stessa transizione possano essere contemporaneamente abilitate. sono considerate indipendente, eventi che si verificano contemporaneamente (a meno che non siano in conflitto un altro). Le espressioni di arco sono somme (formali) di tuple; ogni elemento di una tupla a sua volta è una somma pesata di termini che denotano multi-insiemi delle classi di colore di base. Le espressioni dell'arco sono strutturate secondo il corrispondente il dominio del colore di posto. Se il dominio del colore di posto contiene k classi di colore di base (cioè k "campi"), quindi l'espressione dell'arco corrispondente è una somma pesata di "k-tuple". L'intera espressione denota un multi-insieme nel prodotto cartesiano delle classi colorate di base che compongono il corrispondente il dominio del colore di posto.

Definizione delle SN

Le Reti Simmetriche (SN) [2] sono un tipo di reti di Petri Colorate caratterizzate da una sintassi che permette di descrivere in maniera compatta e funzionale le componenti simmetriche di un sistema, cioè gli elementi di un sistema che hanno un comportamento equivalente.

Nella descrizione degli aspetti principali della definizione del formalismo si farà riferimento alla rete SN disegnata in Fig. 2.2. Essa modella un semplice protocollo di comunicazione broadcast tra i nodi di un network. Un nodo può appartenere ad una di due sotto-network e trasmette messaggi agli altri nodi. Alla ricezione di un messaggio di dati, un nodo trasmette a sua volta un ACK agli altri. Un ACK viene conservato dal mittente iniziale e scartato dai nodi rimanenti. Una sessione termina correttamente se il mittente iniziale riceve risposte di conferma da tutti gli altri. I messaggi inviati tra le sotto-reti possono andare persi ma l'estratto qui mostrato non gestisce questo evento.

Una rete simmetrica (SN) è una tupla:

$$(P, T, \mathcal{C} \cup \{E\}, \mathcal{D}, g, \mathrm{I}, \mathrm{O}, \mathrm{H}, \mathbf{m}_0)$$

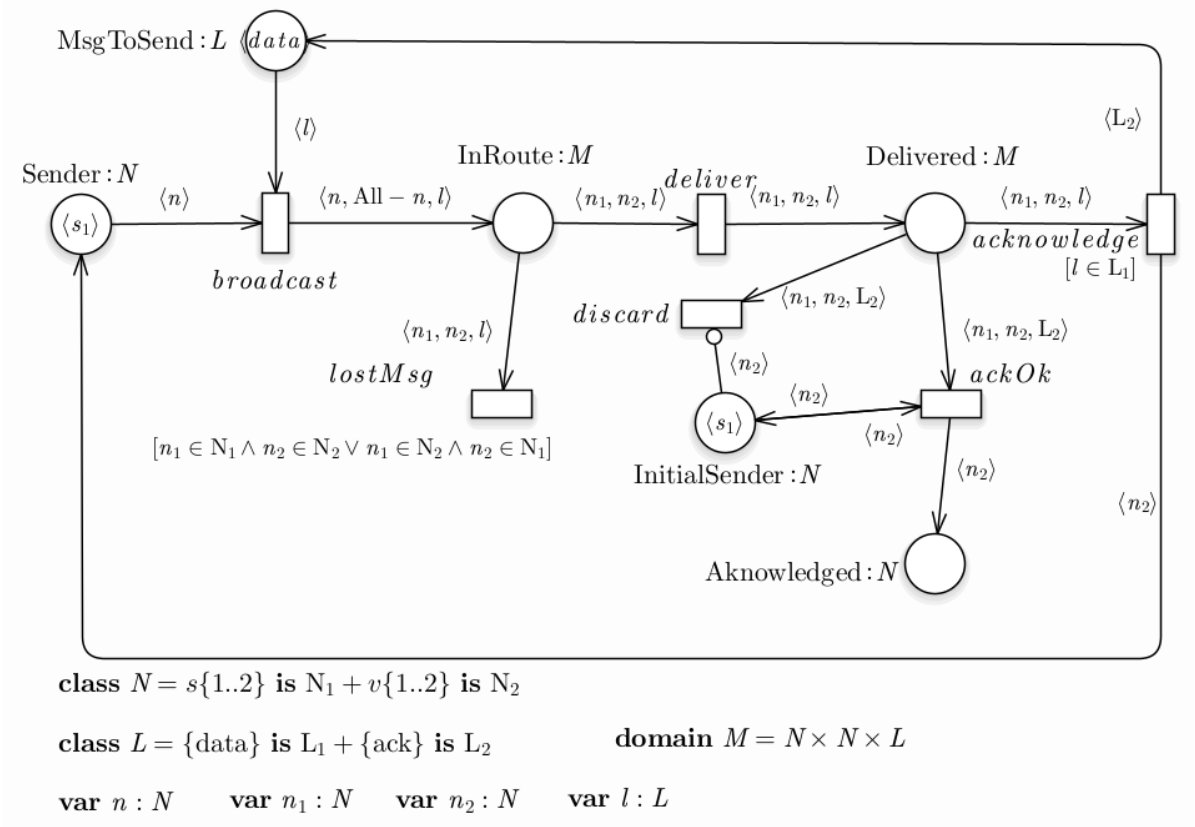

Figura 2.2: Rete SN che rappresenta un protocollo broadcast in un network.

dove:

- P e T sono gli insiemi finiti e disgiunti dei posti e delle transizioni della rete;
- $\mathcal{C} = \{C_i, i = 1 \dots, \mathrm{n}\}$, $\mathrm{n} \in \mathbb{N}^+$, è l'insieme delle *classi di colore*. Una classe di colore C_i può essere *circolarmente ordinata* o *partizionata* in *sottoclassi statiche* $C_{i,j}, j = 1 \dots, k_i$. Il partizionamento statico di una classe determina la simmetria di un modello SN, infatti, tutti e solo i colori di una sottoclasse denotano le componenti omogenee con un comportamento equivalente.

  Nel modello modello di Fig. 2.2 la rete ha due classi di colore: N, L. Esse rappresentano rispettivamente i nodi del network e i tipi dei messaggi che possono essere scambiati. N è partizionata in due sottoclassi statiche ciascuna di dimensione 2. Esse rappresentano due tipi diversi di nodi appartenenti a sotto-network diversi. L è partizionata in due sottoclassi statiche, ciascuna di dimensione 1, che rappresentano i messaggio di *ack* e i messaggi contenenti *dati*.

Ogni modello SN è ulteriormente fornito della classe di colore *neutra* $\mathrm{E} = \{\bullet\}$.

- $\mathcal{D}$ mappa ogni $v \in P \cup T$ a un *dominio di colore*, generalmente definito come un prodotto cartesiano di classi di colore: $\mathcal{D}(v) = \prod_{C_i \in \mathcal{C}} C_i^{e_i}$ dove $e_i \in \mathbb{N}$ è il numero delle ripetizioni di C_i nel dominio. Ci possono essere anche nodi col dominio neutro $\mathcal{D}(v) = \mathrm{E}$. Un dominio di colore del posto p, $\mathcal{D}(p)$, definisce i colori possibili (tuple degli elementi di colore da $\mathcal{D}(p)$) di tokens che p può contenere.

  Nella rete di Fig. 2.2, il dominio di colore del posto `InRoute` è $\mathrm{N}^2 \times \mathrm{L}$.

  Un dominio di colore di una transizione $\mathcal{D}(t)$ definisce le possibili *istanze di scatto* di t. Ad ogni elemento C_i del dominio di t è possibile associare una variabile che assume valori nella classe di colore: un'istanza di transizione può essere vista come una assegnazione alle variabili della transizione.

  Nella rete di Fig. 2.2 la transizione `broadcast` ha dominio di colore $\mathcal{D}(\texttt{broadcast}) = \mathrm{N} \times \mathrm{L}$. Se si indica con $n \in \mathrm{N}$ e con $l \in \mathrm{L}$ le variabili che assumono valori rispettivamente nelle due componenti del dominio allora una istanza di t si ottiene associando un colore di N a n e un colore di L a l. In tale caso un legame possibile per `broadcast` è ($n = s_1$, $l = data$).

- $\mathbf{m}_0$ è la marcatura colorata iniziale del modello. Nella rete di Fig. 2.2 i posti `Sender` e `InitialSender` contengono ciascuno un token di colore $\langle s_1 \rangle \in \mathrm{N}_1$, mentre il posto `MsgToSend` è maracto con il token $\langle data \rangle \in \mathrm{L}_1$.

- g è un mapping che associa ad ogni $t \in T$ una *guardia* $g(t) : \mathcal{D}(t) \to \{true, false\}$. La definizione completa della loro forma sintattica è data in Sez. 2.2. Una guardia *restringe* le istanze valide di una transizione. Una istanza di transizione $b \in \mathcal{D}(t)$, anche denotata da (t, b), è detta *valida* se solo se $g(t)(b) = true$;

- I, O e H sono le annotazioni rispettivamente sugli archi di input, di output e di inibizione della rete. Per ogni coppia $(p, t) \in P \times T$ $\mathrm{I}[p, t]$, $\mathrm{O}[p, t]$ e $\mathrm{H}[p, t]$ sono funzioni tali che $\mathcal{D}(t) \to Bag[\mathcal{D}(p)]$. La sintassi delle funzioni d'arco è formalmente definita nella Sez. 2.2.

Sintassi delle funzioni d'arco nelle SN

In questa sezione è descritta la sintassi delle funzioni SN, saranno usate le seguenti convenzioni: Le classi di colore sono denotate da lettere maiuscole singole, e.g., V. La i-esima sottoclasse statica di V è denotata da V_i. L'insieme $Var(t)$, $t \in T$, è l'insieme delle variabili associate alla transizione

t. Una variabile è denotata da una lettera minuscola singola che implicitamente riferisce al suo tipo, e.g. v_i è una variabile di classe V. Il numero di variabili di tipo V in $Var(t)$ è pari al numero di ripetizioni della classe V in $\mathcal{D}(t)$: l'indice i, che cade tra $1 \dots e^{\mathrm{V}}$, è usato per differenziare le variabili della stessa classe.

Definizione 1 (Proiezione) *Una proiezione $v_i \in Var(t)$ è una funzione $\mathcal{D}(t) \to Bag[\mathrm{V}]$ che mappa una tupla di colore b alla i-esima occorrenza di colore* V *in b.*

Definizione 2 (Guardie) *Una* guardia $g(t) : \mathcal{D}(t) \to \{true, false\}$ *è definita in termini di predicati base e di connettivi logici. Riferendo alla classe generica* V*, un predicato di base è uno dei seguenti:*

- $v_1 = (\neq)\ v_2$ $\quad true$ *(per $b \in \mathcal{D}(t)$) quando $v_1(b) = (\neq)\ v_2(b)$*
- $v_i \in (\notin)\ \mathrm{V}_j$ $\quad true$ *quando $v_1(b)$ appartiene (non appartiene) alla sottoclasse* V_j
- $d(v_1) = (\neq) d(v_2)$ $\quad true$ *quando $v_1(b)$, $v_2(b)$ appartiene alla stessa (differente) sottoclasse(i) (attualmente non supportato dal tool GreatSPN).*

Il secondo ed il terzo predicato sono ammessi se la classe V *è partizionata. Se* V *è ordinata, quindi nel primo tipo di predicato, le proiezioni possono avere un suffisso di $++$, $--$, denotando il* $\mathrm{mod}_{|\mathrm{V}|}$ *successore/predecessore del colore selezionato.*

Ad esempio, la guardia della transizione `lostMsg` *in Fig. 2.2, dove* $\mathcal{D}(\texttt{lostMsg}) = \mathrm{N}^2 \times \mathrm{L}$*, indica che solo i messaggi che sono inviati da una sottorete ad una altra possono essere persi.*

Le funzioni di arco. Le *annotazioni di arco* delle SN sono costruite a partire dalle *funzioni di classe*. Consideriamo un arco che è collegato ad un posto p e ad una transizione t. Una funzione f_i di classe V è una mappa $\mathcal{D}(t) \to Bag[\mathrm{V}]$, che è definita come una combinazione lineare di *funzioni elementari*:

$$f_i = \sum_h \alpha_h . e_h,\ \alpha_h \in \mathbb{Z}, \tag{2.1}$$

dove e_h può essere una tra $\{v_j, \mathrm{V}_q, \mathrm{V}\}$[1]:

- v_j è una *proiezione*, possibilmente seguita se V è ordinata da $++$ o $--$;

[1] Il simbolo All è usualmente utilizzato invece dell'insieme V

- V_q e V sono funzioni *constanti* rispettivamente definite come $\sum_{x \in \mathrm{V}_q} 1 \cdot x$ e $\sum_{x \in \mathrm{V}} 1 \cdot x$.

Nell'esempio di Fig. 2.2 la funzione $All - n$ di classe N sull'arco di output tra `broadcast` e `InRoute` mappa il colore $\langle c_1, c_2 \rangle \in \mathrm{N} \times \mathrm{L} = \mathcal{D}(\texttt{broadcast})$ a $\sum_{x \in \mathrm{N}, x \neq c_1} 1 \cdot x$, che rappresenta tutto l'insieme dei nodi della rete tranne c_1.

Una funzione d'arco $F[p,t] : \mathcal{D}(t) \rightarrow Bag[\mathcal{D}(p)]$ è definita come una combinazione lineare di tuple di funzioni di classe:

$$F[p,t] = \sum_k \lambda_k . T_k[g_k], \ \lambda_k \in \mathbb{Z}, \tag{2.2}$$

dove T_k è un prodotto cartesiano $\langle f_1, \ldots, f_{\mathrm{n}} \rangle$ di funzioni di classe[2], e g_k è una guardia opzionale definita su $\mathcal{D}(t)$, con la stessa sintassi delle guardie di transizione: $T_k[g_k](b) = T_k(b)$ se $g_k(b) = true$, altrimenti $T_k[g_k](b) = \emptyset$, $\forall b \in \mathcal{D}(t)$.

Ad esempio nel modello di Fig. 2.2 la funzione d'arco $\mathrm{O}[\texttt{InRoute}, \texttt{broadcast}] = \langle n, All - n, l \rangle$ quando è valuta su $\langle c_1, c_2 \rangle \in \mathrm{N} \times \mathrm{L}$ risulta in $\langle c_1, \sum_{x \in \mathrm{N}, x \neq c_1} 1 \cdot x, c_2 \rangle$, un multiset che corrisponde a tutte terzine con c_1 come primo elemento, un nodo tranne c_1 come secondo elemento, e c_2 come terzo elemento. Quindi, un messaggio broadcast tipo-c_2 inviato dal nodo c_1..

Semantica delle reti SN

Una *marcatura* di una rete SN corrisponde ad una nozione di stato distribuito. Una marcatura $\mathbf{m}$ è un P-vettore tale che $\mathbf{m}[p] \in Bag[\mathcal{D}(p)]$, $\forall p \in P$. Chiamiamo $\mathbf{m}[p]$ la marcatura del posto p, e gli elementi di $\mathbf{m}[p]$ *tokens*. La dinamica di una SN è definita dalla*regola di scatto*. Assumiamo nel seguito che gli archi mancanti siano annotati da funzioni *nulle*. Una istanza di una transizione (t,b) *ha connessione* in una marcatura $\mathbf{m}$ se e solo se:

- $\forall p \in P$: $\mathrm{I}[p,t](b) \leq \mathbf{m}[p]$
- $\forall p \in P, x \in \mathrm{H}[p,t](b)$: $\mathrm{H}[p,t](b)(x) > \mathbf{m}[p](x)$

(t,b) è *abilitata* in $\mathbf{m}$ se e solo se (t,b) ha concessione in $\mathbf{m}$ e non esiste nessuna istanza (t',b') che abbia concessione in $\mathbf{m}$ tale che $\pi(t') > \pi(t)$. Nella tesi consideriamo reti SN dove le transizioni hanno tutte la stessa priorità e rappresentano eventi osservabili. Per queste retei i concetti di

[2] la classe di colore f_i

concessione e abilitazione coincidono. I moduli di traduzione sviluppati nella tesi sono facilmente estendibili al caso generale.

Assumendo che in Fig. 2.2 la marcatura iniziale del posto `Sender` (i.e., un token singolo $\langle s_1, data \rangle$), l'istanza $(n = s_1, l = data)$ di `broadcast` è abilitata. se sia abilitata, (t, b) potrebbe*scattare*, causando la marcatura $\mathbf{m}'$ formalmente definita come:

$$\forall p : \mathbf{m}'[p] = \mathbf{m}[p] - \mathrm{I}[p,t](b) + \mathrm{O}[p,t](b)$$

è detto che $\mathbf{m}'$ *raggiungibile* da $\mathbf{m}$ tramite (t, b), e denota $\mathbf{m}[t, b\rangle\mathbf{m}'$. Lo scatto di $(\texttt{broadcast}, n = s_1, l = data)$, preleva il token $\langle s_1, data \rangle$ dal posto di input `Sender` ed assegna (insieme di tipo) "bag" di funzioni $\sum_{x \in \mathrm{N}, x \neq s_1} 1 \cdot \langle s_1, x, data \rangle$ nel posto di output `InRoute`, rappresentando un messaggio-dato broadcast inviato da s_1.

è detto che la marcatura $\mathbf{m}$ è *svanito* se esistono qualche istanze di transizioni immediate in $\mathbf{m}$, *tangibili* altrimenti. Un *model* SN una SN con *marcatura iniziale tangibile* $\mathbf{m}_0$. Assumendo che non ci sono sequenze infinite delle istanze immediate di transizione, è possibile definire la *tangible reachability graph* (TRG) di un modello SN, spigoli etichettati, multi-grafo diretto (V, E) i cui nodi sono le marcature tangibili: $\mathbf{m}_0 \in V$; se $\mathbf{m} \in V$ ed esiste una sequenza (possibilmente *vuota*) $\{(t_i, b_i), \pi(t_i) > 0\}, i : 1 \ldots \mathrm{n} \in \mathbb{N}$, tale che $\mathbf{m}[t, b\rangle\mathbf{m}_1[t_1, b_1\rangle \ldots \mathbf{m}_\mathrm{n}[t_\mathrm{n}, b_\mathrm{n}\rangle\mathbf{m}'$, con $\mathbf{m}'$ tangibile, $\mathbf{m}' \neq \mathbf{m}$, quindi $\mathbf{m}' \in V$ e $\mathbf{m} \xrightarrow{t,b} \mathbf{m}' \in E$.
Per quanto riguarda la marcatura "iniziale" che sia un componente indispensabile nella rete, la marcatura iniziale di un posto è una combinazione tra tuple di colori/tokens colorati che seguono lo stesso formato del tipo di posto sia una classe di colore che dominio di classi di colore; durante il momento idle iniziale nella rete si potrebbe cominciare il trasferimento di tuple di tokens dagli archi collegati al posto marcato come archi uscenti da esso. Inoltre, le tuple di tokens nella marcatura iniziale possono seguire anche una forma simmetrica come il seguente:

$$\forall p \in P, \mathbf{m}[p] = \sum_{\tilde{c} \in \bigotimes_{C_i \in \mathcal{C}} \tilde{C}_i^{e_i}} \alpha_{\tilde{c}} \tilde{c}$$

dove $\tilde{c}$ è una tupla di sottoclassi statiche e $\alpha_{\tilde{c}}$ è il coefficiente (molteplicità) di $\tilde{c}$ che deve essere un numero positivo.

2.3 Il formato PNML per reti SN

In questa sezione è stato implementato il formato PNML completo che non è completamente supportato dal tool ”GreatSPN”. I tags e attributi che costituiscono la parte aggiuntiva sono:

- il tag di classe di colore partizionata <partition> e ovviamente anche il tag di sottoclasse <partitionelement>
- l'attributo ”ordered” che segnala che la classe di colore è ordinata/circolare.
- il tag completo di una espressione d'arco <hlinscription> tale che è stata data l'abilità di leggere le guardie ed i filtri delle funzioni d'arco; inoltre, è stata data l'abilità di leggere gli elementi di tuple che possono essere costituiti da certe combinazioni lineari.
- il tag completo di una guardia di transizione <condition>.

Ora descriviamo il formato PNML (http://www.pnml.org) usando la rete in Fig. 2.2.

```
<?xml version="1.0"?>
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
<!-- Written by GreatSPN Editor. -->
<net id="SN" type="http://www.pnml.org/version-2009/grammar/symmetricnet">
<name>
<text>SN</text>
</name>
<declaration>
<structure>
<declarations>
<!-- Declaration of user-defined color classes (sorts) -->
<partition id="N" name="N" ordered="false">
             <partitionelement id="N1" name="N1">
    <finiteintrange start="1" end="2"/>
     </partitionelement>
     <partitionelement id="N2" name="N2">
    <finiteintrange start="1" end="2"/>
     </partitionelement>
    </partition>
<partition id="L" name="L" ordered="false">
            <partitionelement id="L1" name="L1">
     <useroperator declaration="data"/>
     </partitionelement>
     <partitionelement id="L2" name="L2">
     <useroperator declaration="ack"/>
     </partitionelement>
```

```
    </partition>
<namedsort id="M" name="M">
<productsort>
<usersort declaration="N"/>
<usersort declaration="N"/>
<usersort declaration="L"/>
</productsort>
</namedsort>
<!-- Declaration of user-defined color variables -->
<variabledecl id="n" name="n">
<usersort declaration="N"/>
</variabledecl>
<variabledecl id="n1" name="n1">
<usersort declaration="N"/>
</variabledecl>
<variabledecl id="n2" name="n2">
<usersort declaration="N"/>
</variabledecl>
<variabledecl id="l" name="l">
<usersort declaration="L"/>
</variabledecl>
</declarations>
</structure>
</declaration>
<page id="page0">
<name>
<text>DefaultPage</text>
</name>
<!-- List of places -->
<place id="Sender">
<name>
<text>Sender</text>
</name>
<type>
<text>N</text>
</type>
<hlinitialMarking>
<text>&lt;s1&gt;</text>
</hlinitialMarking>
</place>
<place id="InRoute">
<name>
<text>InRoute</text>
</name>
<type>
<text>M</text>
</type>
</place>
<place id="MsgToSend">
<name>
<text>MsgToSend</text>
</name>
```

```
<type>
<text>L</text>
</type>
<hlinitialMarking>
<text>&lt;data&gt;</text>
</hlinitialMarking>
</place>
<place id="Delivered">
<name>
<text>Delivered</text>
</name>
<type>
<text>M</text>
</type>
</place>
<place id="Aknowledged">
<name>
<text>Aknowledged</text>
</name>
<type>
<text>N</text>
</type>
</place>
<place id="InitialSender">
<name>
<text>InitialSender</text>
</name>
<type>
<text>N</text>
</type>
<hlinitialMarking>
<text>&lt;s1&gt;</text>
</hlinitialMarking>
</place>
<!-- List of transitions -->
<transition id="broadcast">
<name>
<text>broadcast</text>
</name>
</transition>
<transition id="lostMsg">
<name>
<text>lostMsg</text>
</name>
<condition>
<text>n1 in N1  &amp;&amp;  n2 in N2 || n1 in N2  &amp;&amp; n2 in N1</text>
</condition>
</transition>
<transition id="deliver">
<name>
<text>deliver</text>
</name>
```

```
</transition>
<transition id="discard">
<name>
<text>discard</text>
</name>
</transition>
<transition id="ackOk">
<name>
<text>ackOk</text>
</name>
</transition>
<transition id="acknowledge">
<name>
<text>acknowledge</text>
</name>
<condition>
<text>l in L1</text>
</condition>
</transition>
<!-- List of arcs -->
<arc id="id1" source="Sender" target="broadcast">
<hlinscription>
<text>&lt;n&gt;</text>
</hlinscription>
</arc>
<arc id="id2" source="broadcast" target="InRoute">
<hlinscription>
<text>&lt;n,All-n,l&gt;</text>
</hlinscription>
</arc>
<arc id="id3" source="InRoute" target="lostMsg">
<hlinscription>
<text>&lt;n1,n2,l&gt;</text>
</hlinscription>
</arc>
<arc id="id4" source="InRoute" target="deliver">
<hlinscription>
<text>&lt;n1,n2,l&gt;</text>
</hlinscription>
</arc>
<arc id="id5" source="MsgToSend" target="broadcast">
<hlinscription>
<text>&lt;l&gt;</text>
</hlinscription>
</arc>
<arc id="id6" source="acknowledge" target="Sender">
<hlinscription>
<text>&lt;n2&gt;</text>
</hlinscription>
</arc>
<arc id="id7" source="acknowledge" target="MsgToSend">
<hlinscription>
```

```
<text>&lt;L2&gt;</text>
</hlinscription>
</arc>
<arc id="id8" source="deliver" target="Delivered">
<hlinscription>
<text>&lt;n1,n2,l&gt;</text>
</hlinscription>
</arc>
<arc id="id9" source="Delivered" target="acknowledge">
<hlinscription>
<text>&lt;n1,n2,l&gt;</text>
</hlinscription>
</arc>
<arc id="id10" source="Delivered" target="ackOk">
<hlinscription>
<text>&lt;n1,n2,L2&gt;</text>
</hlinscription>
</arc>
<arc id="id11" source="ackOk" target="Aknowledged">
<hlinscription>
<text>&lt;n2&gt;</text>
</hlinscription>
</arc>
<arc id="id12" source="InitialSender" target="ackOk">
<hlinscription>
<text>&lt;n2&gt;</text>
</hlinscription>
</arc>
<arc id="id13" source="ackOk" target="InitialSender">
<hlinscription>
<text>&lt;n2&gt;</text>
</hlinscription>
</arc>
<arc id="id14" source="Delivered" target="discard">
<hlinscription>
<text>&lt;n1,n2,L2&gt;</text>
</hlinscription>
</arc>
<arc id="id15" source="InitialSender" target="discard">
<hlinscription>
<text>&lt;n2&gt;</text>
</hlinscription>
<type value="inhibitor"/>
</arc>
</page>
</net>
</pnml>
```

3. Realizzazione di un traduttore dal formato *pnml* per reti simmetriche a un formato ad oggetti per l'analisi simbolica

In questo capitolo discuteremo la prima parte intera del traduttore in cui costruiamo gli oggetti [12] fondamentali della rete simmetrica proveniente dal file "pnml" iniziale; inoltre, questa prima parte del traduttore include tutte le tre fasi iniziali (Analisi lessicale, Analisi sintattica/sinattica, Analisi semantica) partendo da un file xml scritto nel formato" pnml "e finendo alla generazione di un albero sintattico astratto modificato (AST modificato/rete simmetrica), tale che si spiegherà ciascuna fase di loro con i suoi input ed output come il seguente:

3.1 Analisi lessicale (scanner)

l'input di questa fase è le liste che contengono tutti i tags delle componenti della rete simmetrica entrante che sono (posti con cui la marcatura iniziale è inclusa se esista, transizioni con cui la guardia è inclusa se esista, archi con cui il tipo è incluso se sia inibitore, classi di colori, classi di colori partizionate, domini di classi di colore, variabili).

Gli attribuiti significativi di elementi alla prima fase (analisi lessicale)

Ora potremo pensarci per un momento agli attributi correlati a ciascun componente e sono considerabili nella nostra rete:

- caso di una classe di colore sono (il nome della classe, un campo che determina se la classe è circolare/ordinata oppure no, la dimensione della classe oppure i colori interni).
- caso di una classe partizionata sono gli stessi attributi di una classe di colore non partizionata più i nomi delle sottoclassi e le loro dimensioni oppure i loro colori interni.
- caso di un dominio di classi di colori sono (il nome del dominio, il prodotto cartesiano ordinato delle classi contenute nel dominio).
- caso di una variabile sono (il nome della variabile, il tipo della variabile che sia una classe di colore).
- caso di un posto sono (il nome del posto, il tipo del posto "può essere classe di colore oppure dominio di classi di colore", la marcatura iniziale se esista).
- caso di una transizione sono (il nome della transizione, la guardia se esista).
- caso di un arco sono (il nome del nodo di partenza, il nome del nodo d'arrivo, il tipo dell'arco, i dati dell'espressione d'arco "combinazione di tuple collegate con guardie e filtri se esistono in fianco di qualche tupla", il nome d'arco può essere utilizzato come un identificatore).

La relazione tra lo scansionatore xml e l'analizzatore lessicale

Lo scansionatore xml legge e sistema tutti i dati necessari in certe liste, tali che ciascuna lista contiene soltanto gli elementi che appartengono allo stesso tipo (es. lista di elementi posti, lista di elementi classi di colori, …); poi tali liste saranno elaborate per analizzarle lessicalmente nello scanner.

L'analizzatore lessicale/scanner ha differenti modi per analizzare ed elaborare le liste di elementi in modo dipendente dal tipo di ciascuna; si tratta con ciascuna lista come il seguente:

- *lista di elementi di classi di colore*:
  per ciascun elemento, si dovrà sapere se l'elemento è inizialmente partizionato oppure rappresentato come enumerazione finita o intervallo finito su prefisso; se l'elemento fosse partizionato, allora si dovrebbe estrarre anche tutti i dati di ciascuna partizione interna, poi si passeranno questi dati alla prossima fase "analizzatore sintattico". Si ricordi che un elemento partizionato possa contenere diversi tags di enumerazione finita o intervallo finito su prefisso.
  È possibile esprimere i colori/tokens di colori di ciascuna classe di colore/sottoclasse di colore in due modi diversi:

  – enumerazione finita: in questo tipo di espressione, si scrivono esplicitamente i colori di classe di colore/ sottoclasse di colore come il seguente esempio nel caso di classe di colore: `enum {c1, c2, c3, c4, c5}`. Mentre nel caso di sottoclasse di colore sarebbe come il seguente: `enum {c1, c2, c3, c4, c5}` is subclass; si noti che `c1, c2, c3, ...` siano i colori/tokens di colori di una classe di colore oppure una sottoclasse di colore;

  – intervallo finito su prefisso: in questo tipo di espressione, si scrivono implicitamente i colori di classe di colore/ sottoclasse di colore tale che si considera una stringa come un prefisso e poi si lega tal prefisso direttamente con un intervallo solitamente limitato come il seguente: esempio nel caso di classe di colore: "c1..5" che significa "c[1:5]" , in questo caso il prefisso è "c" e l'intervallo è "[1:5]"ed ha lo stesso significato del esempio precitato nella parte di enumerazione finita. Mentre nel caso di sottoclasse di colore sarebbe come il seguente: "c1..5 is subclass".

    Inoltre, nella sintassi implicita di "prefisso{n1..n2}" se 'n1' e 'n2' sono uguali tali che n1,n2 $\in \mathbb{N} - \{0\}$; allora questo significa che la sottoclasse non è parametrica e l'ampiezza di tale sottoclasse è (n2 – n1 + 1); mentre se non lo sono, allora la sottoclasse è parametrica e l'ampiezza di tale sottoclasse è semplicemente n1 o n2

- *lista di elementi di domini*:
  per ciascun elemento, si dovrà estrarre direttamente i suoi dati come (il nome, il prodotto cartesiano di classi di colori), poi si passeranno questi dati alla prossima fase "analizzatore sintattico".

- *lista di elementi di variabili*:
  per ciascun elemento, si dovrà estrarre direttamente i suoi dati come (il nome, il tipo “il nome di una classe di colore”), poi si passeranno questi dati alla prossima fase ”analizzatore sintattico”.

- *lista di elementi di posti*:
  per ciascun elemento, si dovrà estrarre inizialmente i dati essenziali dell’elemento (il nome, il tipo “il nome di una classe di colore oppure di un dominio”), poi si dovrà ulteriormente sapere se l’elemento della marcatura iniziale è incluso in questo elemento di posto oppure no; se lo sia, allora si dovrebbe passare tal elemento di marcatura iniziale al suo scansionatore per estrarre le tuple di tokens in modo strutturato, dopo tale parte di elaborazione si passeranno questi dati alla prossima fase ”analizzatore sintattico”.

- *lista di elementi di transizione*:
  per ciascun elemento, si dovrà estrarre inizialmente il dato essenziale dell’elemento (il nome), poi si dovrà ulteriormente sapere se l’elemento della guardia è incluso in questo elemento di transizione oppure no; se lo sia, allora si dovrebbe passare l’elemento di guardia al suo scansionatore per estrarre le informazioni scritte nella guardia, dopo tale parte di elaborazione si passeranno questi dati estratti alla prossima fase ”analizzatore sintattico”.
  La lista di elementi delle transizioni scansionate verrà analizzata lessicalmente per ogni elemento di transizione in tale lista usando l’analizzatore di transizione, il sotto-analizzatore che analizza le guardie e il sotto-analizzatore che analizza i predicati. Il risultato incapsulato restituito dopo tal tipo d’analisi è come il seguente: il nome di transizione + la guardia di transizione + flag che inverta tale guardia associata. Prendiamo la transizione ”acknowledge” in Fig.2.2 come un esempio, [nome di transizione = ”acknowledge”, guardia = [predicato1 = (”l in L1”), flag predicato1 = ”false”], flag guardia = ”false”].

- *lista di elementi di arco*:
  per ciascun elemento, si dovrà estrarre inizialmente i dati essenziali dell’elemento (il nome del nodo di partenza, il nome di nodo di destinazione, il tipo dell’elemento “arco di transito/inibitore”, il nome che verrà utilizzato come identificatore, le tuple scritte nell’espressione caricata con la loro molteplicità), poi si dovrà ulteriormente sapere se l’elemento della guardia/filtro è associato a qualche tupla oppure no; se lo sia, allora si dovrebbe passare l’elemento di guardia/filtro al suo scansionatore per estrarre le informazioni scritte in tale guardia o tal filtro.

Osservazioni

Si noti che ciascuna tupla trovata nell'espressione esistente su un elemento di arco, dovrà essere passata al suo scansionatore per estrarre i dati necessari dei membri di tupla.

Inoltre, si deve notare che la guardia è suddivisa in un certo numero di predicati che sono combinati usando le operazioni booleane logiche(And, Or, Not); quindi è possibile passare ciascun predicato al suo scansionatore per estrarre i dati necessari di qualsiasi predicato (la negazione del predicato se esista, il primo membro dell'elemento di predicato, l'operazione usata "sia di equivalenza che di appartenenza", il secondo membro dell'elemento di predicato) oppure semplicemente un predicato costante (True o False), poi si restituiscono le informazioni estratte all'elemento antenato che contenga tale guardia che sarebbe "un elemento di transizione oppure un elemento d'arco".

Nella figura 3.1 si trova il diagramma di attività sintetizzato che descrive la nostra prima fase ("analizzatore lessicale/scanner").

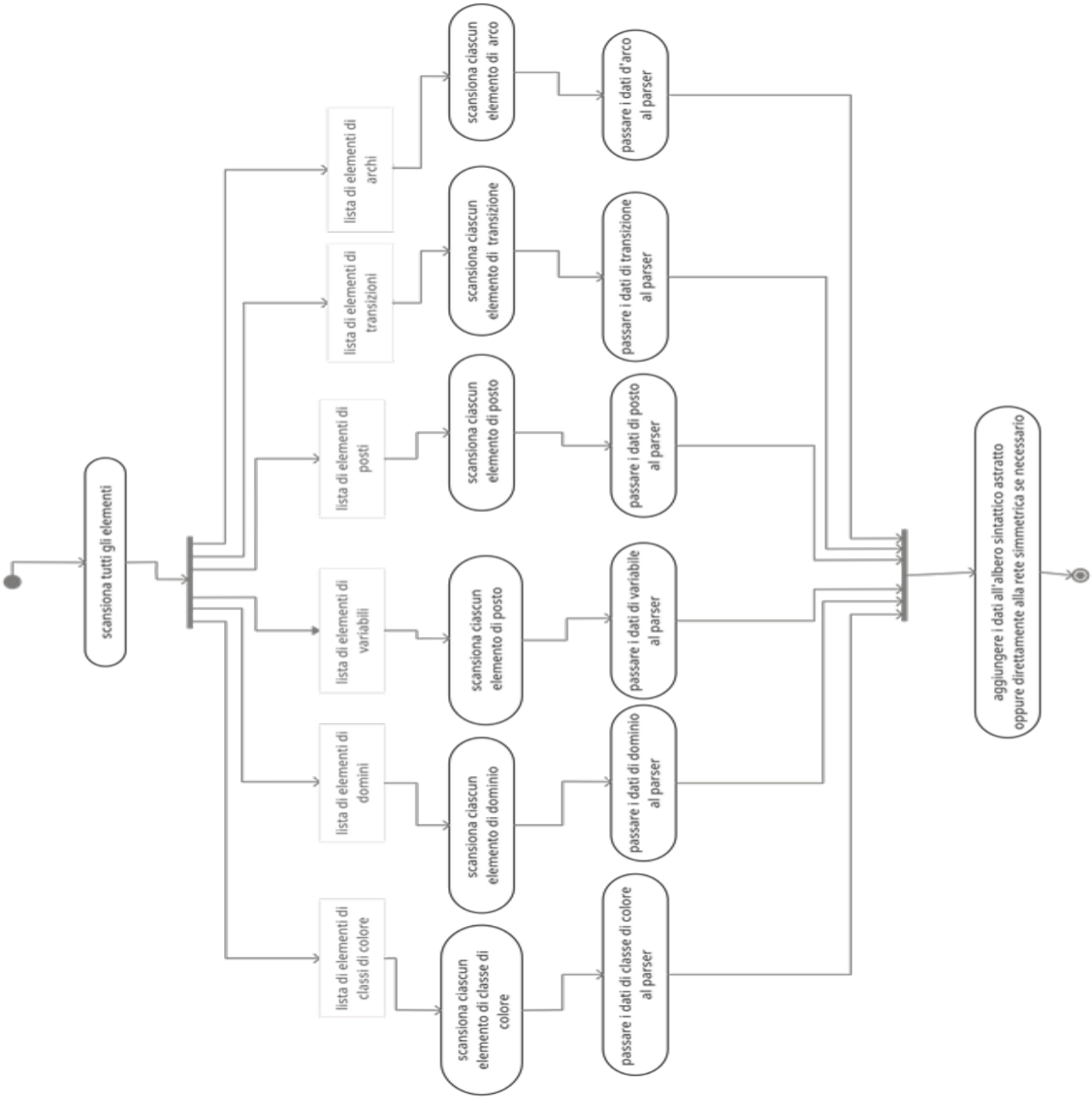

Figura 3.1: Diagramma d'attività dell'analizzatore lessicale.

3.2 Analisi sintattica (Data Parser)

L'input di questa fase è passato su diverse volte in modo dipendente dai dati provenienti dalla prima fase "analisi lessicale", tale che una volta è scansionato un elemento essenziale, verrà passato

direttamente a questa fase per elaborarlo in forma ad oggetti.

L'analizzatore sintattico riceve i dati tramite certi metodi dedicati agli elementi essenziali (posti, transizioni, archi, classi di colori, variabili, domini di classi di colore, marcatura iniziale); tali elementi essenziali possano contenere elementi secondari (es. una transizione abbia l'elemento di guardia come un elemento secondario, un arco abbia l'elemento di tupla come un elemento secondario, …); quindi, i dati di elementi secondari vengono passati con i dati degli elementi essenziali a cui sono collegati.

Elaborazione di dati provenienti alla fase di analisi sintattica

Siccome i dati sono già analizzati lessicalmente dalla fase precedente, allora l'analizzatore sintattico decide cosa dovrà fare con i dati di ciascun elemento essenziale ed i dati di ciascun elemento secondario collegati ad essi; perché saranno incapsulati in oggetti diversi (siano oggetti intermediari dell'albero sintattico astratto 3.2, siano oggetti finale di SN).

Ora si discute l'elaborazione dei dati di ciascun elemento essenziale ed i dati dei suoi elementi secondari:

- *i dati di classe di colore*:
  sono i dati provenienti dallo scansionatore di classe di colore, e si elaborano in tre metodi diversi (elaborazione di classe di colore di tipo enumerazione finita, elaborazione di classe di colore di tipo intervallo finito su prefisso, elaborazione di classe di colore partizionata).

  In ogni caso di elaborazione si salvano i dati direttamente come una classe di colore nell'oggetto finale della rete simmetrica senza attraversare l'albero sintattico astratto; inoltre, si dovrà controllare se il nome della classe sia vietato oppure no, tale che esiste sempre un nome riservato della classe neutra (token di unico colore "nero").

  Se i colori/tokens di colore della classe di colore/sottoclasse di colore sono esplicitamente espressi, quindi verranno conservati in una tabella come tokens espressi; se non lo siano, allora verrà indicato che i tokens sono implicitamente espressi per stimarli dopo, usando lo stimatore di tokens in modo basato sulla marcatura iniziale.

  Inoltre, non esistono dati di nessun elemento secondario sotto questo tipo di dati di una classe di colore, perché si considerano le sottoclassi di colore (se la classe è partizionata) come membri fondamentali nella dichiarazione dell'elemento essenziale "classe di colore".

- *i dati di dominio di classi di colore*:
  sono i dati provenienti dallo scansionatore di dominio di classi di colore, e si elaborano in un unico metodo che salva i dati direttamente come un dominio nell'oggetto finale della rete simmetrica senza attraversare l'albero sintattico astratto; inoltre, si dovrà aggiungere ad una tabella l'ordine del prodotto cartesiano di classi di colore che sono membri di tal dominio, quindi può essere confrontato con la sintassi delle tuple collegate al posto che sia di tal tipo di dominio.

  È possibile notare che durante la creazione di un oggetto di tipo dominio, si dovrà controllare se una classe di colore membro sia moltiplicata più di una volta, in tal caso, si dovrebbe ricordare la molteplicità di ciascuna classe per salvarla nell'oggetto di dominio.

- *i dati di variabile*:
  sono i dati provenienti dallo scansionatore di variabile, e si elaborano in un unico metodo che salva i dati direttamente come una variabile nell'oggetto finale della rete simmetrica senza attraversare l'albero sintattico astratto.

- *i dati di posto*:
  sono i dati provenienti dallo scansionatore di posto, e si elaborano in un unico metodo che salva i dati direttamente come un posto nell'oggetto finale della rete simmetrica; si dovrà passare ulterioremente i dati di tal posto come un nodo componente iniziale nell'albero sintattico astratto, tale che si potrebbe collegarlo con i nodi diversi (transizioni sintattiche) per completare tutte i collegamenti dell'albero.

- *i dati di marcatura iniziale*:
  sono i dati provenienti dallo scansionatore di marcatura iniziale, e si elaborano in due metodi diversi (elaborazione di marcatura di posto appartenente ad un tipo di classe di colore, elaborazione di marcatura di posto appartenente ad un tipo di dominio di classi di colore).

  Nel caso di un tipo di classe di colore, si dovrà passare i dati di tuple di tokens all'analizzatore semantico di tupla per analizzare la combinazione lineare scritta in ciascun membro di tupla, tale che una tupla di tokens di questo tipo potrebbe contenere le informazioni seguenti <combinazione lineare tra colori, costanti, funzioni di classe>.

  Nel caso di un tipo di dominio di classi di colore, si dovrà passare i dati di tuple di tokens all'analizzatore semantico di tupla per analizzare la combinazione lineare scritta in ciascun membro di tupla, tale che una tupla di tokens di questo tipo ha la forma seguente <combinazione lineare1, combinazione lineare2, …>; si deduca che una tupla di marcatura di questo tipo abbia lo stesso ordine di membri di tal dominio.

Inoltre, le tuple di tokens/colori sono combinate linearmente ed hanno le loro molteplicità indipendenti.

- *i dati di transizione*:
  sono i dati provenienti dallo scansionatore di transizione, e si elaborano in un unico metodo, tale che verranno passati questi dati direttamente all'albero sintattico astratto come una transizione sintattica; poiché verrà calcolato il dominio di un nodo transizione quando la prima fase (analizzatore lessicale) finisca il suo svolgimento e tutti gli archi collegati a questa transizione sarebbero già letti, e sono conservati nell'albero sintattico astratto. (si discute l'albero sintattico astratto nella prossima pagina).
  Spiegando la stessa transizione "acknowledge" prediscussa nella sezione di analisi lessicale 3.1, si trova che l'analizzatore sintattico che crea gli oggetti necessari per l'albero sintattico astratto "AST", creerà gli seguenti oggetti relativi all'oggetto di transizione sintattica "acknowledge". Ad esempio, transizione sintattica= [nome= "acknowledge", guardia sintattica [[predicato sintattico[valore di predicato = "l in L1"], flag di predicato sintattico], flag = "false"].

- *i dati di arco*:
  sono i dati provenienti dallo scansionatore d'arco, e si elaborano in un unico metodo, tale che verranno passati questi dati direttamente all'albero sintattico astratto come un'arco sintattico; poiché verrà calcolato il dominio di ciascuna tupla sintattica nell'espressione appartenente a quest'arco, quando la prima fase (analizzatore lessicale) finisca il suo svolgimento e tutti gli archi collegati alla stessa transizione alla quale quest'arco è collegato sarebbero già letti, e sono conservati nell'albero sintattico astratto. (si discute l'albero sintattico astratto nella prossima pagina).

Albero sintattico astratto:
È la struttura di dati che sarebbe trattata come l'output della seconda fase di "analizzatore sintattica" e l'input della terza fase "analizzatore semantico".

L'albero sintattico astratto contiene i nodi sintattici della rete (posti sintattici/transizioni sintattiche), tali che i nodi sono collegati implicitamente tra loro usando archi sintattici, tale che ciascun nodo ha una mappa di nodi prossimi "i nodi a cui è possibile arrivare usando certi archi partendo dal nodo attuale".

Nella prossima pagina figura 3.2, si descriva la struttura parzialmente interna dell'albero sintattico astratto.

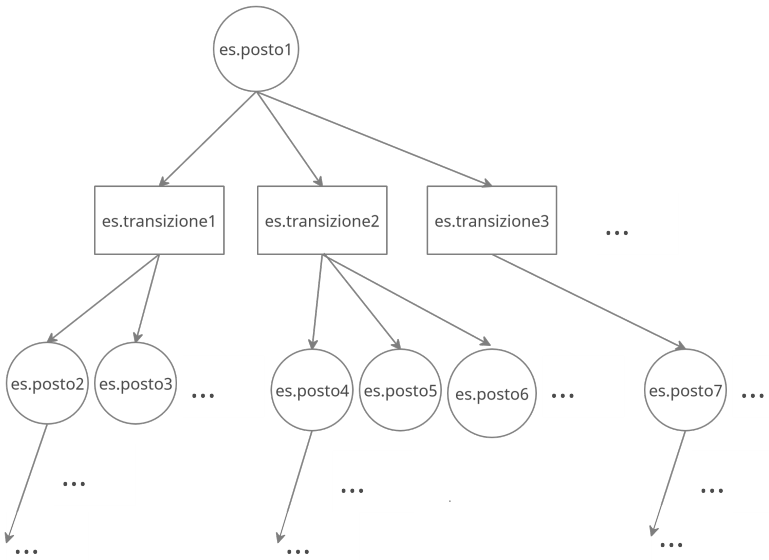

Figura 3.2: esempio di albero sintattico astratto

Tale che la struttura esterna dell'albero ha soltanto due mappe (posti e transizioni), e gli archi prossimi di ciascun nodo sono espressi implicitamente in ciascun nodo e sono associati al nodo d'arrivo appartenente a ciascun arco.

Comunque, ogni transizione sintattica è incapsulata con la sua guardia sintattica; ed ogni arco sintattico è incapsulato con una mappa ordinata che contenga tutte le tuple/funzioni d'arco "sintattiche" associate alla loro molteplicità.

Inoltre, ogni funzione d'arco è incapsulata con la sua guardia sintattica ed il suo filtro sintattico.

Osservazioni

Si noti che la transizione sintattica e la tupla/funzione sintattica contengono le guardie sintattiche (nel caso della tupla sintattica, si aggiunge anche il filtro sintattico); dunque, la guardia sintattica è incapsulata con una mappa ordinata dei predicati sintattici associati ai segni di negazione (operazione logica "Not").

Poi al basso livello di incapsulazione, un predicato sintattico contiene semplicemente una lista dei suoi elementi, includendo l'operazione usata (operazione di appartenenza oppure operazione di eguaglianza).

Si noti che tutti i dati contenuti nell'albero sintattico astratto sono dati preliminari e sono fondamentalmente utilizzabili per calcolare i domini di transizioni, i domini o codomini di ciascuna tupla/funzione d'arco, e sono ulteriormente utilizzabili per analizzare le variabili.

Quando si trova l'albero sintattico completa, allora la seconda fase "l'analizzatore sintattico" ha finito il suo compito si dovrà passarlo alla prossima fase per analizzarlo semanticamente.

Nella figura 3.3 (alla fine capitolo) si trova il diagramma di attività sintetizzato che descrive la nostra seconda fase ("analizzatore sintattico/DataParser").

3.3 Analisi semantica (Semantic analyzer)

L'input di questa fase è l'albero sintattico astratto, su cui sono basate tutte le analisi dell'analizzatore semantico.

Siccome l'albero sintattico assegnato a questa fase è completo e contiene tutti i componenti strutturali della rete simmetrica, allora è possibile analizzare inizialmente il dominio di ciascun nodo transizione come il seguente:

Suppone che abbiamo un nodo transizione 'T', tale che 'T' abbia un certo numero di variabili nella sua guardia; inoltre, vi siano gli archi connessi attorno ad essa che contengano anche un certo numero di variabili nelle loro tuple/guardie; quindi, il dominio della transizione 'T' sarebbe il prodotto cartesiano di tutti i tipi di variabili "non ripetute" (classi di colori) che sono attorno a 'T' oppure dentro la sua guardia.

Inoltre, l'ordine del prodotto cartesiano di una cd(transizione) non è molto interessante, perché è basato sull'ordine degli archi collegati ad una transizione durante la sua elaborazione oppure è ulteriormente basato sull'ordine tali archi scritti nel file pnml.

In questa fase si deve analizzare ciascuna tupla d'arco usando l'analizzatore di tupla che utilizza due tipi di analizzatori diversi:

1. analizzatore di proiezioni/variabili:
   analizza la stringa assegnata per creare uno da 3 tipi di proiezioni con un certo indice assegnato in modo basato sul dominio della transizione collegata, i tre tipi di proiezioni sono come il seguente:

   - variabile con successore (es. x++)
   - variabile con predecessore (es. x–)
   - variabile normale (es. x).

   Per motivi di efficienza l'indice di una proiezione verrà restituito da una certa tabella se la variabile è ripetuta attorno la stessa transizione; altrimenti, verrà generato e salvato nella tabella precitata per utilizzarlo nel futuro nel caso di ripetizione, invece di generare uno nuovo che potrebbe confliggere la molteplicità di suo tipo di classe di colore.

   Per quanto riguarda l'analisi di proiezioni/variabili di filtro di tupla; il filtro di una tuple d'arco non ha nè lo stesso dominio della guardia della stessa tupla nè il dominio di tale

tupla, perché il dominio del filtro d'arco è il codominio della transizione collegata a tal arco; quindi, le proiezioni/variabili di un certo filtro potrebbero avere diversi indici da quelli riservati nelle tuple e/o le guardie della stessa transizione collegata.

2. analizzatore di costanti:
   analizza la stringa assegnata per creare uno da 2 tipi di constanti con un certo indice prefissato, i due tipi di costanti sono come il seguente:

   - costante di una sottoclasse di colore
   - costante di una classe di colore, in ogni caso l'indice di una classe di colore e l'indice di una sottoclasse di colore sono indipendenti dalla traduzione.

Esempio dell'analisi semantica della transizione "acknowledge" spiegata in 3.1:
L'analizzatore semantica dovrà estrarre le parti che è necessario analizzarle prima della generazione dell'oggetto finale di tale transizione come il seguente:
Inizialmente il dominio della transizione sintattica verrà analizzata in modo dipendente dagli archi attorno ad esso. Ad esempio; il dominio di transizione "acknowledge" = $N^2 * L^1$. (la modalità usata è spiegata in questa sezione sopra 3.3).
Inoltre, la guardia sintattica di tale transizione verrà passata all'analizzatore di guardia per separare i predicati sintattici ed analizzare ciascun predicato da solo. Ad esempio, il predicato sintattico unico [l in L1] della guardia associata alla transizione "acknowledge", verrà suddiviso in due parti (la proiezione "l" + la costante "L1") che sarebbero analizzate semanticamente dai loro sotto-analizzatori (analizzatore di proiezioni/variabili + analizzatore di constanti) rispettivamente, quindi la guardia risultata dall'analizzatore di guardia sarebbe (guardia[predicato di appartenenza = "variabile=l" + costante= "L1"]). Poi otterremo l'oggetto finale di tale transizione nel nostro caso come il seguente: transizione finale[nome, guardia, dominio].

L'uso dei sotto-analizzatori semantici (analizz. di tupla, analizz. di guardia, analizz. di proiezione/variabile, analizz. di costante)

L'analizzatore di tupla possa generare un elemento di tipo "funzione di classe" come la funzione "All" che implichi tutti i tokens di colore di una certa classe di colore.

In questa fase si deve analizzare ciascuna guardia di tuple/transizione usando l'analizzatore di guardia che utilizza sempre gli due sotto-analizzatori (analizzatore di proiezioni, analizzatore di proiezioni costanti) che sono precitati sopra.

Inoltre, si utilizza l'analizzatore di guardia per analizzare anche i filtri, siccome hanno la stessa sintassi usata dalle guardie. Nel caso in cui si utilizza una sintassi differente per esprimere gli operatori di un filtro, si dovrà convertire tali operatori in certi proiezioni/costanti corrispondenti ad essi e ripassarle ai sotto-analizzatori appropriati {es. il tool [1] utilizza per esprimere le proiezioni/variabili dei filtri una sintassi parzialmente differente da quella usata per esprimere le proiezioni/variabili delle guardia. In tal caso è stata aggiunta all'analizzatore di tupla l'abilità di analizzare ulteriormente tale sintassi differente}.

Si noti che l'analizzatore di tupla possa anche analizzare le tuple di tokens della marchiatura iniziale siccome possono utilizzare anche le costanti e/o le funzioni di classe, e possono utilizzare la molteplicità di tupla.

L'unica differenza tra le tuple d'arco e le tuple di marcatura è "le tuple d'arco utilizzano le proiezione/variabile, mentre le tuple di tokens utilizzano i colori/tokens di colore", e tale situazione è risolvibile quando sarebbe data all'analizzatore di tuple la possibilità di creare ulteriormente oggetti di tokens di colore analizzati ed assegnarli ad una tabella per riutilizzarli ancore nel caso di ridondanza.

Collegare i nodi finali dopo l'analisi semantica dei loro dati associati

L'analizzatore semantico deve ulteriormente generare oggetti di archi analizzati ed assegnarli ai nodi corrispondenti ad essi, tali che si collegono implicitamente due nodi (posto, transizione) usando un arco analizzato come il seguente:

- collegare da un nodo posto ad un nodo transizione usando un arco analizzato (può essere un arco di transito oppure un arco inibitore): Si aggiungono l'arco e la transizione come due oggetti associati alla mappa di nodi prossimi o nodi inibitori del posto prescelto.

  Si aggiungono inoltre l'arco ed il posto come due oggetti associati tra loro alla mappa di nodi precedenti o nodi inibitori della transizione prescelta.

- collegare da un nodo transizione ad un nodo posto usando un arco analizzato (deve essere un arco di transito): Si aggiungono l'arco ed il posto come due oggetti associati tra loro alla mappa di nodi prossimi della transizione prescelta.

  Si aggiungono inoltre l'arco e la transizione come due oggetti associati tra loro alla mappa di nodi precedenti del posto prescelto.

Tal modo di collegamento tra nodi raggiunge la possibiltà di arrivare a qualsiasi nodo, partendo da qualsiasi altro nodo nella stessa rete.

Osservazioni

Si noti che la maggior parte delle tabelle realizzate nel nostro traduttore siano state usate per supportare questa fase "analisi semantica" e siano citate nel settimo capitolo, sezione componenti 6. Tali tabelle aiutino gli sotto-analizzatori semantici ad analizzare (la sintassi di posti, le proiezioni, i tokens di colore, i tokens marcati, ecc.).
Nella figura 3.4, si trova il diagramma di attività sintetizzato che descrive la nostra terza fase ("analizzatore semantico/semantic analyzer").

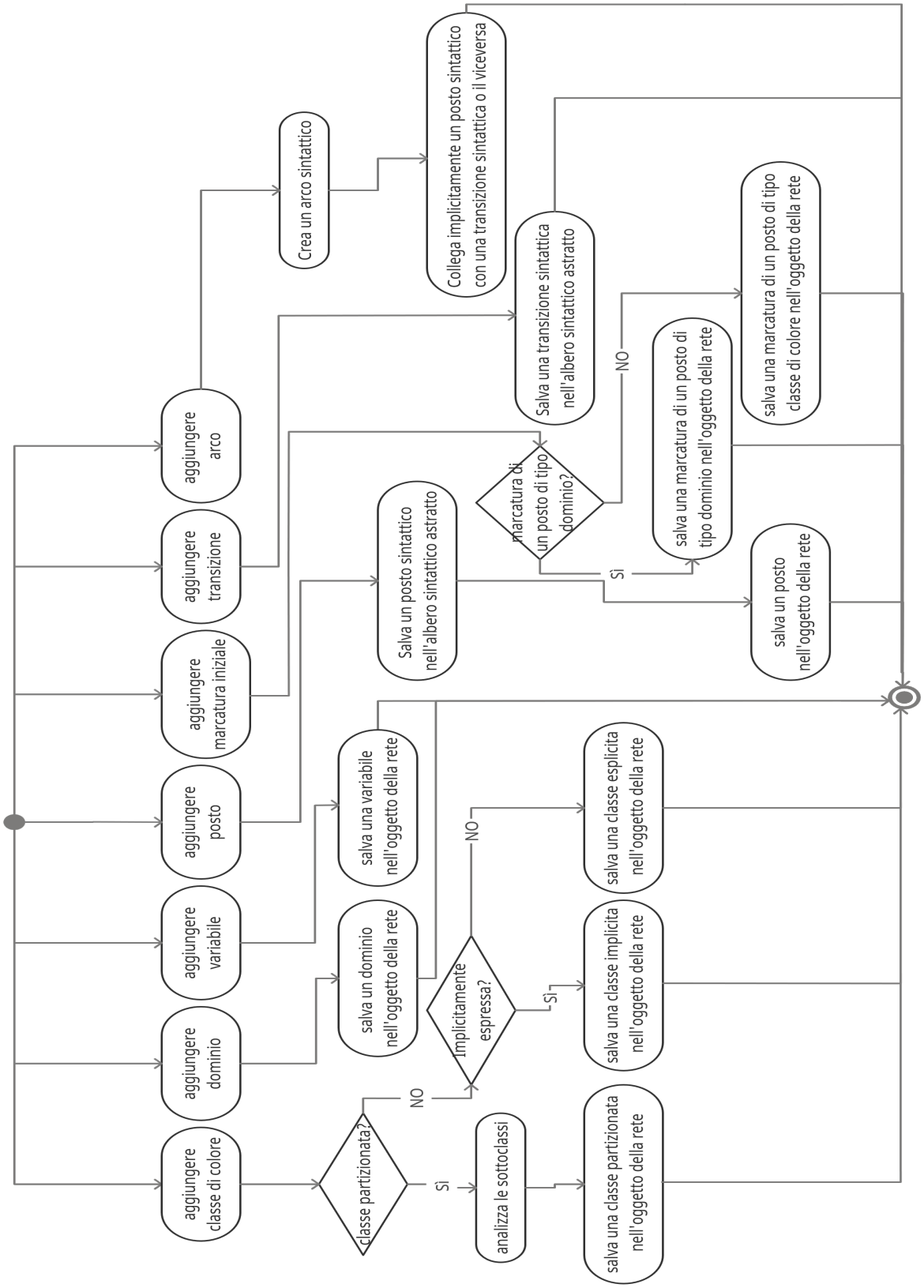

Figura 3.3: Diagramma d'attività dell'analizzatore sintattico.

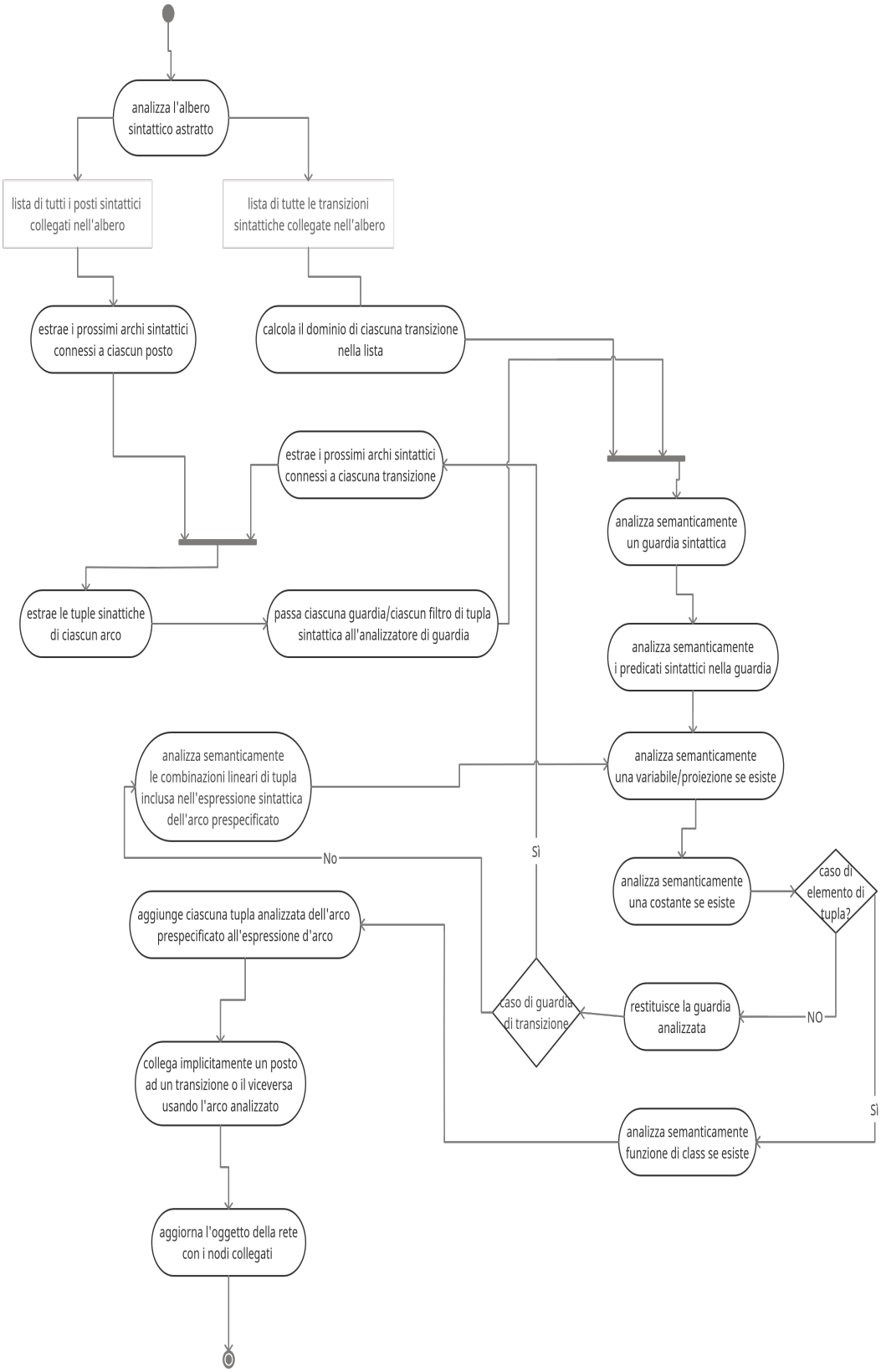

Figura 3.4: Diagramma d'attività dell'analizzatore semantico.

4. Implementazione di un algoritmo di unfolding “parziale (simbolico)” per reti simmetriche

In questa parte discuteremo l’ultima fase del nostro traduttore che sarebbe “generazione del codice”, l’input di questa fase è l’albero sintattico modificato (AST modificato) ovvero una rete simmetrica completa nel nostro caso. Nell’ultima fase di traduzione si applicherebbe un algoritmo [11] ”unfolding parziale” basato sui domini di posti “cd(p), per ciascun posto ‘p’ nella rete”, tale che cd(p) può essere una classe di colore singola oppure un dominio di classi di colori predefinito; l’algoritmo di unfolding parziale che useremmo, mira trovare tutte le combinazioni possibili di un cd(p) per ciascun posto ‘p’ tale che ogni combinazione possibile di cd(p) corrisponde ad un nuovo posto dopo l’unfoldind di ”p”.

4.1 Esempi di dominio di colore cd(p)

Supponiamo che cd(p) abbia una classe di colore singola elevata ad una molteplicità di 2 in tal cd(p), con tre sottoclassi incluse in essa, quindi le combinazioni essenziali possibili di tale classe di colore sarebbero ‘9’ che abbia la seguente forma (n*e) tale che ‘n’ sia il numero delle sottoclassi incluse in una certa classe di colori, mentre ‘e’ sia la molteplicità di tale classe di colore in cd(p). Mentre se cd(p) fosse un dominio, supponeremmo che cd(p) contenga classi di colore differenti,

quindi le combinazioni essenziali possibili di cd(p) avrebbero la forma seguente:

$$\prod_{i=1}^{k} n_i^{e_i} \tag{4.1}$$

che è il prodotto cartesiano tra le combinazioni essenziali di tutte le classi di colore "i : k" coinvolte in cd(p), tale che 'n_i' è il numero di sottoclassi di colore incluse nella i-esima classe di colore, mentre 'e_i' è la molteplicità della i-esima classe di colore in cd(p). Si noti che una combinazione essenziale possibile possa essere scomposta in più di una combinazione possibile, dipendentemente dalla forma della combinazione stessa; per esempio, se abbiamo una combinazione essenziale possibile di una classe di colore in cui più di una cifra di sottoclasse sia ripetuta (es. 11, 1..1, …); in tal caso si dovrebbe suddividere la combinazione essenziale in un certo numero di combinazioni secondarie possibili in modo dipendente dalle ripetizioni della stessa cifra in tale combinazione essenziale; poiché siano combinazioni suddivise, si dovrebbero trattarle come combinazioni essenziali indipendenti nel prodotto cartesiano; quindi le combinazioni possibili sarebbero più di (4.1). Per raggiungere il caso precedente, cioè suddividere le combinazioni essenziali possibili in altre combinazioni, si dovrà avere almeno una classe di colore elevata ad una molteplicità >1 (supponendo che la classe di colore non partizionata contenga sola una sottoclasse di colore).

4.2 I passi implementati dell'algoritmo unfolding parziale (gli ultimi quattro passi riguardano la generazione della rete in formato xml sia PNML sia PNPRO)

1. estrae il dominio di colore di "p": cd(p), per ogni posto "p" nella rete.

2. se cd(p) non è stato visito precedentemente, allora prosegue dal punto (3); invece se cd(p) è stato visito precedentemente, allora prosegue dal punto (11).

3. calcola le combinazioni possibili di base di ogni classe di colore "cc" esistente in cd(p) che sia elevata ad una certa molteplicità e_i; se "cc^{e_i}" è stata visita precedentemente, allora prosegue dal punto (10) per tale classe di colore "cc" in cd(p) poi riesegue l'algoritmo da questo punto per un'altra classe di colore in cd(p).

4. se la molteplicità $e_i > 1$, allora calcola inoltre le combinazione complementari dalle sottoclassi statiche in "cc", ed applica il prodotto cartesiano tra le combinazioni di base di cc e le combinazioni complementari per ottenere le combinazioni possibili complete di cc in cd(p); altrimenti, si considerano le combinazioni possibili di base come combinazioni possibili complete di cc in cd(p).

5. calcolare i filtri di base di ogni classi di colore "cc" esistente in cd(p) che sia elevata ad una certa molteplicità e_i, tale che il filtro di base viene calcolato in modo dipendente dalla combinazione di base corrispondentemente generata.

6. se la molteplicità $e_i > 1$, allora calcola inoltre i filtri complementari dalle sottoclassi statiche in "cc", tale che il filtro complementare viene calcolato in modo dipendente dalla combinazione complementare corrispondentemente generata, ed applica il prodotto cartesiano tra i filtri di base di cc ed i filtri complementari per ottenere i filtri completi di cc in cd(p); altrimenti, si considerano i filtri di base come filtri completi di cc in cd(p).

7. applica il prodotto cartesiano tra le combinazioni complete di cc in cd(p), per ottenere i nuovi nomi di posti ottenuti da tal prodotto che siano corrispondenti alle loro combinazioni.

8. applica il prodotto cartesiano tra i filtri completi (usando l'operazione logica AND) di cc in cd(p), per ottenere i nuovi archi connessi alla stessa transizione di "p", tale che i nuovi filtri saranno combinati (usando l'operazione logica AND) con i filtri dell'espressione d'arco originale che c'era prima dell'unfolding.

9. salva i risultati dell'unfolding parziale di ciascuna classe di colore "cc" in cd(p) e salva i risultati dell'unfolding parziale di cd(p) per utilizzarli nel futuro quando si affronta cd(altro p) oppure un'altra "cc" della stessa molteplicità e_i.

10. restituisce i dati dell'unfolding di ogni classe di colore elevata alla molteplicità e_i in cd(p) e continua l'unfolding dal punto(3) se ci sono altre classi di colore "cc" che non sono state viste precedentemente.

11. restituisce i dati dell'unfolding di cd(p) con i nomi dei posti ed archi generati.

12. collega ogni nuovo arco col posto corrispondente ad una certa combinazione finale di cd(p).

13. crea le liste di elementi XML/PNML generati da SN dopo l'unfolding

14. crea le liste di elementi XML/PNPRO generati da SN dopo l'unfolding

15. scrive i dati della SN generata nel formato PNML.

16. scrive i dati della SN generata ulteriormente nel formato ”PNPRO” assieme alla SN originale.

4.3 Elaborare le combinazioni possibili generate per cd(p)

Si noti che le combinazioni possibili di cd(p) possano essere già calcolate prima, siccome sia possibile aggiungere un tipo di classi di colore/dominio di classi di colore a più di un posto ‘p’; quindi, si dovrebbe salvare le combinazioni possibili calcolati di tutti “cd” in una tabella per restituirle quando si affronti lo stesso “cd” ancora.

Per ciascuna combinazione possibile di cd(p), si dovrebbe generare un filtro che simula la combinazione possibile, tale che il filtro generato dovrà impedire il flusso di qualsiasi altro elemento ovvero qualsiasi altra sottoclasse che non appartiene alla combinazione possibile; ad esempio, se abbiamo una combinazione possibile di “sottoclasse1, sottoclasse2, …”; in tal caso il filtro dovrà ammettere solo il flusso dei tokens di colore/colori che appartengono a “sottoclasse1” come un primo membro nella tupla a cui verrà associato tal filtro, dovrà inoltre ammettere solo il flusso dei tokens di colore/colori che appartengono a “sottoclasse2” come un secondo membro nella tupla a cui verrà associato tal filtro, etc.

Inoltre, la marcatura iniziale di un posto “p” dovrà essere suddivisa in tutti i posti che appartengono alle combinazioni possibili di cd(p) in modo dipendente dalle sottoclassi ammissibili in ciascun posto generato.

4.4 Osservazioni

Si dovrà ulteriormente duplicare le espressioni di archi collegati ad un posto “p” in tutti gli archi collegati ai posti generati dalle combinazioni possibili rispetto all’altro nodo di collegamento; cioè si intende applicare la duplicazione delle espressioni di archi su tutte le tuple con le sue guardie degli archi collegati ad un posto “p”.

Si noti che nel caso di esistenza di un filtro già associato ad una tupla scritta in espressione d'arco di un posto basato su una combinazione possibile, si richiederebbe comporre il filtro già associato col filtro generato dall'algoritmo unfolding usando un'operazione logica "And", poi verrebbe riassociato il filtro composto alla tupla prespecificata.

Ora vediamo un esempio di applicazione l'algoritmo unfolding su una certa rete simmetrica come la seguente: 4.1

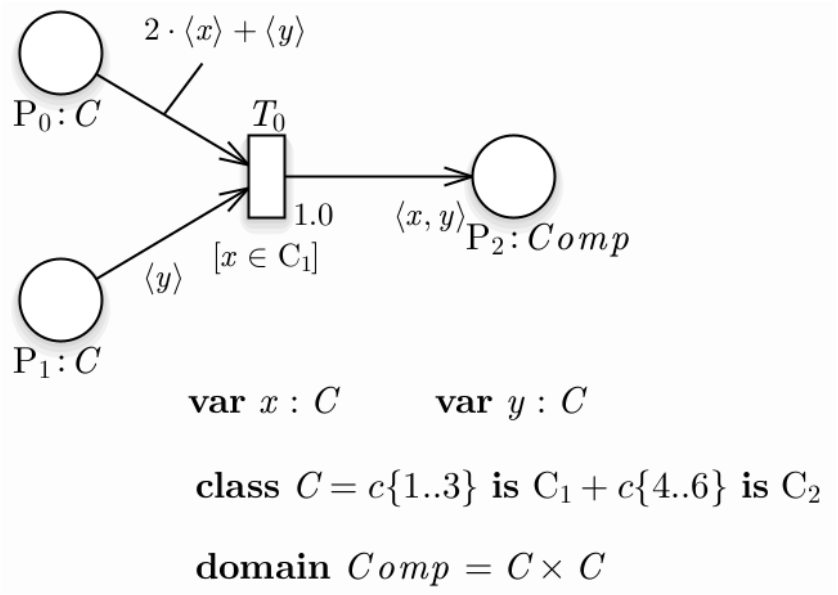

Figura 4.1: Esempio di rete simmetrica iniziale semplice

Quindi la rete uscente dall'unfolding dovrebbe essere la rete seguente nella prossima pagina:4.2

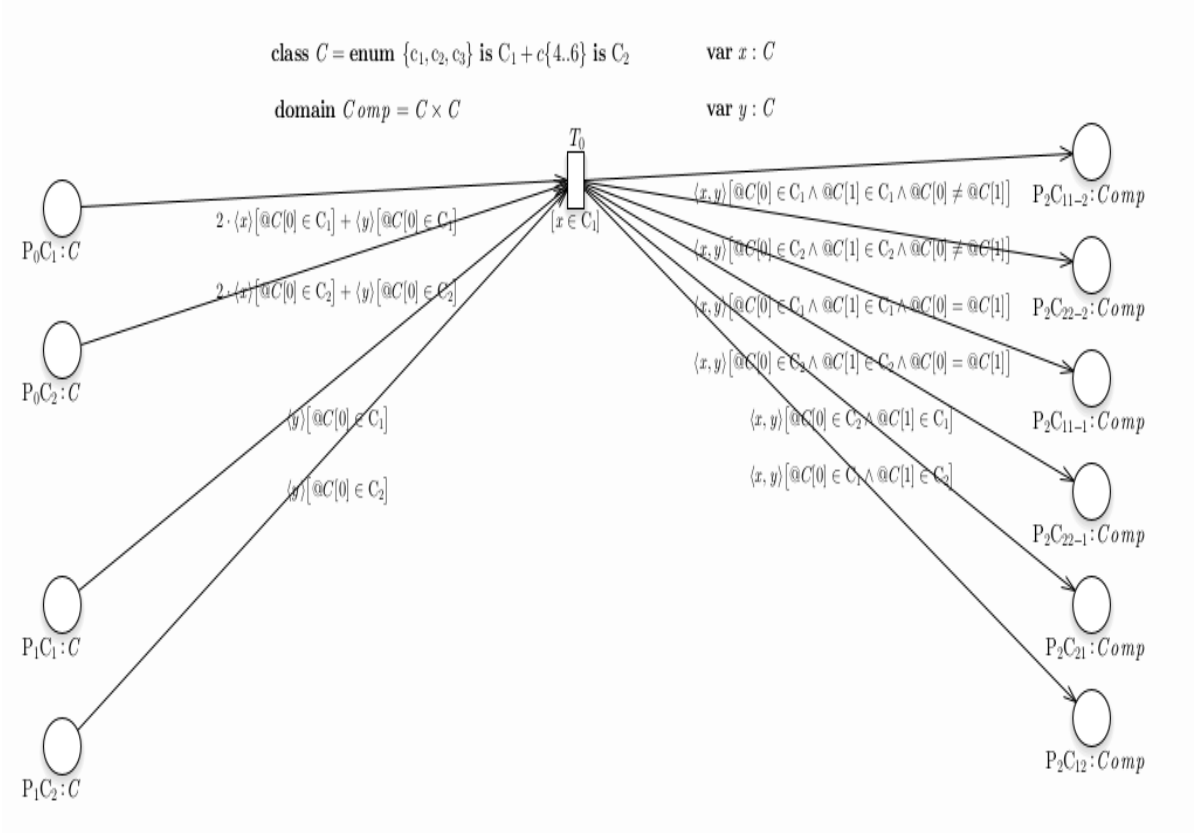

Figura 4.2: Unfolding della rete simmetrica 4.1

Si noti che la sintassi di variabili di filtri nella rete 4.2 è semplicemente la stessa sintassi consciuta; l'unica differenza è quest'operatore "@nome di classe [indice di istanza di una certa classe nella tupla associata]" che implica una variabile ma usando la sintassi del tool grafico "GreatSpn" per non confondere tra le proiezioni/variabili di guardie o tuple e le proiezioni/variabili di filtri.

Nella prossima pagina si trova il diagramma di attività sintetizzato che descrive la nostra quarta fase ("generatore del codice/code generator"): 4.3

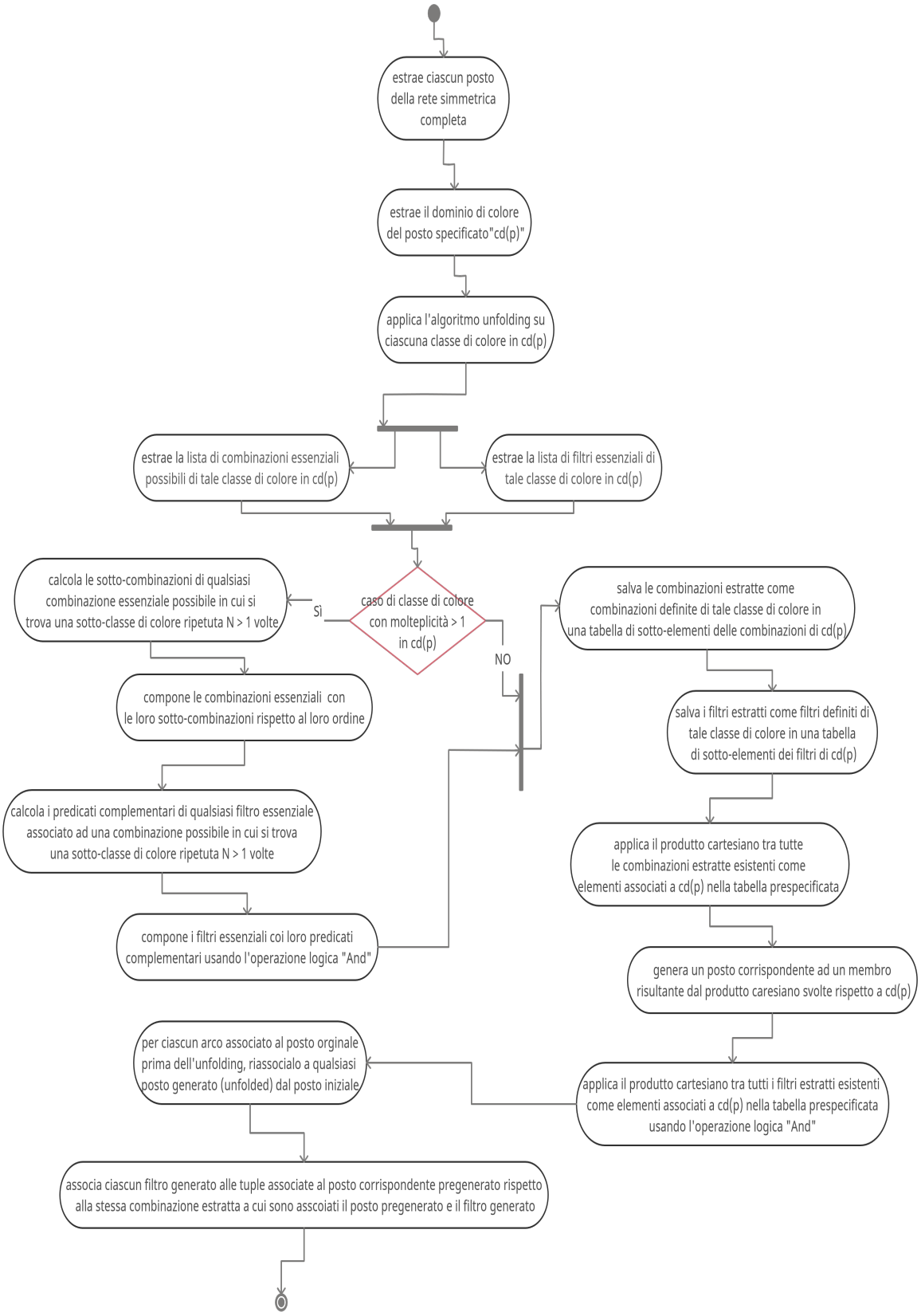

Figura 4.3: Diagramma d'attività del generatore del codice

5. Tecnologie coinvolte

In questo capitolo saranno discusse le tecnologie e le tecniche utilizzate in ciascuna fase di traduttore effettuata. Innanzitutto, nella struttura della rete simmetrica è stato implementato un modo di collegamento implicito tra i nodi diversi della rete; tale che da qualsiasi nodo nella rete è possibile arrivare a qualsiasi altro nodo seguendo un certo percorso di collegamenti differenti; e per renderlo possibile ciascun nodo in tale rete contiene tre mappe che contengono i suoi nodi adiacenti (input, output, inibitori). Comunque, è stata utilizzata la programmazione orientata ad oggetti usando "Java" per costruire interamente il nostro traduttore di reti di petri simmetriche; inoltre, sono state implementate certe parti dalla programmazione funzionale usando "Java streams" per elaborare dati complessi; sono stati ulteriormente usati modi diversi di validazione su dati usando "la gestione di eccezioni" che sia supportata in Java. Sono state largamente usate strutture di dati come complesse "mappe, liste, insiemi", tali che ciascuna struttura di dati può contenere altre strutture di dati in modo dipendente dalla complessità di dati pre-analizzati che sono da contenere. Inoltre, sono stati largamente usati certi tipi di algoritmi respetto alle complessità spaziale e temporale.

5.1 Le tecnologie utilizzate dalle fasi di traduzione diverse

1. Le tecnologie utilizzate nella fase di analisi lessicale

    - Lettura di documenti "xml": il file proveniente deve essere di tipo xml e deve seguire il formato di "pnml". Il documento letto verrà analizzato lessicalmente per estrarre i

tags che hanno certe parole chiave già determinate dal formato usato. Verrà trattato indipendentemente ciascun tag principale che appartiene ad una certa categoria che descrive un componente della rete; tale che ciascun tag sarà considerato come un elemento indipendente da scansionare; poiché tutti i tags principali della stessa parola chiave verranno raccolti in una lista per elaborarli similmente.

- Le espressioni regolari "Regex (ref.[4])" durante l'elaborazioni di elementi simili, si utilizzano le espressioni regolari per elaborare ed ordinare gli sotto-elementi che contengono le stringhe complesse.
- Design patterns (ref.[6])
  - Singleton pattern: è stato usato questo tipo di pattern perché abbiamo bisogno di solo un singolo oggetto dello scansionatore di xml, abbiamo bisogno di solo un singolo oggetto di qualsiasi sotto-scansionatore di elementi (scansionatore di classe di colore, scansionatore di dominio di classi di colore, scansionatore di variabile, scansionatore di posto, scansionatore di transizione, scansionatore d'arco); perché semplicemente il formato di dati che analizziamo è fisso "xml/pnml".
  - Immutable pattern: gli oggetti di classi precitati sopra nella parte di "singleton pattern", dovrebbero essere ulteriormente immutabli/immodificabili, perciò applicheremmo questo tipo di pattern.
  - Façade pattern: questo tipo di pattern implica nascondere la complessità della dipendenza tra lo scansionatore xml e gli scansionatori degli elementi principali della rete.

  Nel nostro caso lo scansionatore xml utilizza i sotto-scansionatori per scansionare i componenti principali della rete simmetrica senza far intervenire la classe del traduttore di rete, siccome la classe del traduttore sia alla cima della nostra gerarchia implementata.

2. Le tecnologie utilizzate nella fase di analisi sintattica

   - Diverse associazioni con le tabelle necessarie per l'analisi sintattica con l'analizzatore sintattico sono state associate certe tabelle completamente necessarie come "tabella di sintassi di posti" che si utilizza per mantenere l'ordine di classi di colore di cd(p), poiché possa essere usata per determinare le funzioni di classi (es. All); è stata ulteriormente usata "tabella di colori/tokens di colore di classi di colore" che si utilizza per mantenere le classi di colori esplicitamente espresse con i loro tokens di colore, poiché non ci sia bisogno di stimare i tokens di colore espliciti. Inoltre, sono stati associati

l'albero sintattico astratto e l'oggetto di tabella di marcatura per compilarli con i dati appropriati.

- Utilizzare la libreria "wncalculus" Partendo da questa fase di traduttore si utilizza la libreria software precitata per creare oggetti esistibili nel nostro caso di reti simmetriche.

  L'analizzatore sintattico utilizza questa libreria per creare diverse classi di colore (partizionate, esplicitamente espresse o implicitamente espresse), tali classi di colori sono già analizzate lessicalmente e sintattica-mente, si utilizza inoltre per creare i domini di classi di colore.

- Design patterns
  - Singleton pattern: è stato usato questo tipo di pattern perché abbiamo bisogno di solo un singolo oggetto dell'analizzatore sintattico, perché semplicemente il compito di tal analizzatore non si cambia durante la traduzione intera. Si noti che l'albero sintattico astratto utilizza e le tabelle associate usufruiscano ulteriormente dal singleton pattern.

3. Le tecnologie utilizzate nella fase di analisi semantica

   - Un'associazione con la tabella necessaria per l'analisi semantica con l'analizzatore semantico è stata associata una certa tabella completamente necessaria "tabella di indici di variabili" che si utilizza per mantenere gli indici delle variabili/proiezioni che sono attorno una certa transizione ed appartengono allo stesso tipo di classe di colore, poiché possa essere usata per retituire semplicemente gli indici di variabili/proiezioni ripetute attorno una transizione senza aver bisogno di rigenerare nu altro indice.
   - Utilizzare la libreria "wncalculus" l'analizzatore semantico utilizza questa libreria per creare proiezioni, costanti, diverse guardie (combinazione tra predicati di tipo "equivalenza/appartenenza"), tuple d'arco (combinazioni lineari tra proiezioni/costanti /funzioni di classe), combinazioni lineari delle espressioni di archi, combinazioni lineari delle tuple di marcatura iniziale.
   - Design patterns
     - Singleton pattern: è stato usato questo tipo di pattern perché abbiamo bisogno di solo un singolo oggetto dell'analizzatore semantico, perché semplicemente il compito di tal analizzatore non si cambia durante la traduzione intera.
     - Façade pattern: nel nostro caso l'analizzatore semantico utilizza i sotto-analizzatori per analizzare i componenti principali della rete simmetrica senza far intervenire

la classe del traduttore di rete, siccome la classe del traduttore sia alla cima della nostra gerarchia implementata.
Inoltre, l'analizzatore semantico costituisce una parte nel facade pattern con l'albero sintattico astratto.

4. Le tecnologie utilizzate nella fase di generazione di codice

   - Utilizzare la libreria "wncalculus" Il generatore del codice utilizza questa libreria per elaborare i dati recentemente salvati dalle fasi precedenti di traduttore come i dati di proiezioni, costanti, diverse guardie (combinazione tra predicati di tipo "equivalenza/ appartenenza"), tuple d'arco (combinazioni lineari tra proiezioni/costanti/funzioni di classe), combinazioni lineari delle espressioni di archi, combinazioni lineari delle tuple di marcatura iniziale.
   - Design patterns
     - Singleton pattern: è stato usato questo tipo di pattern perché abbiamo bisogno di solo un singolo oggetto del generatore del codice, perché semplicemente il compito di tal generatore non si cambia durante la traduzione intera.
     - Façade pattern: nel nostro caso il generatore del codice è associato con lo scrittore xml, per scrivere i dati di componenti diversi (classi di colore, variabili, domini di classi, posti, transizioni, archi). quindi lo scrittore xml ha diversi sottoscrittori per immagazzinare i dati di elementi di ciascun componente nella rete.
     - Delegator pattern: Il pattern delegatore implica creare certi metodi visibili da una certa class mirata, tali che i metodi creati possono dare la classe mirate l'abiltà ad eseguire altri metodi che non sono visibili a tale classe per motivi di incapsulazione.
       Nel nostro caso di generatore del codice, la fase complementare (scrittore xml) dà l'abilità al generatore del codice per utilizzare certi metodi che non erano raggiungibili prima di eseguire questo tipo di pattern.
   - Programmazione dinamica (ref.[7]): inizialmente, si applica il processo del prodotto cartesiano tra gli elementi di cd(p) sui filtri e combinazioni diversi di classi di colore rispettivamente che siano membri in cd(p). (si vede il diagramma seguente **??**):

     durante tal processo, si scompone il processo di prodotto in sotto-processi di prodotto, come l'immagine seguente:

     infine, si trova che i sottoalberi sono ridondanti nei livelli di approfondità > 2; quindi si decide di salvare i risultati precalcolati nel primo sotto-processo, tali che possono essere usati nei prossimi sotto-processi di prodotto.

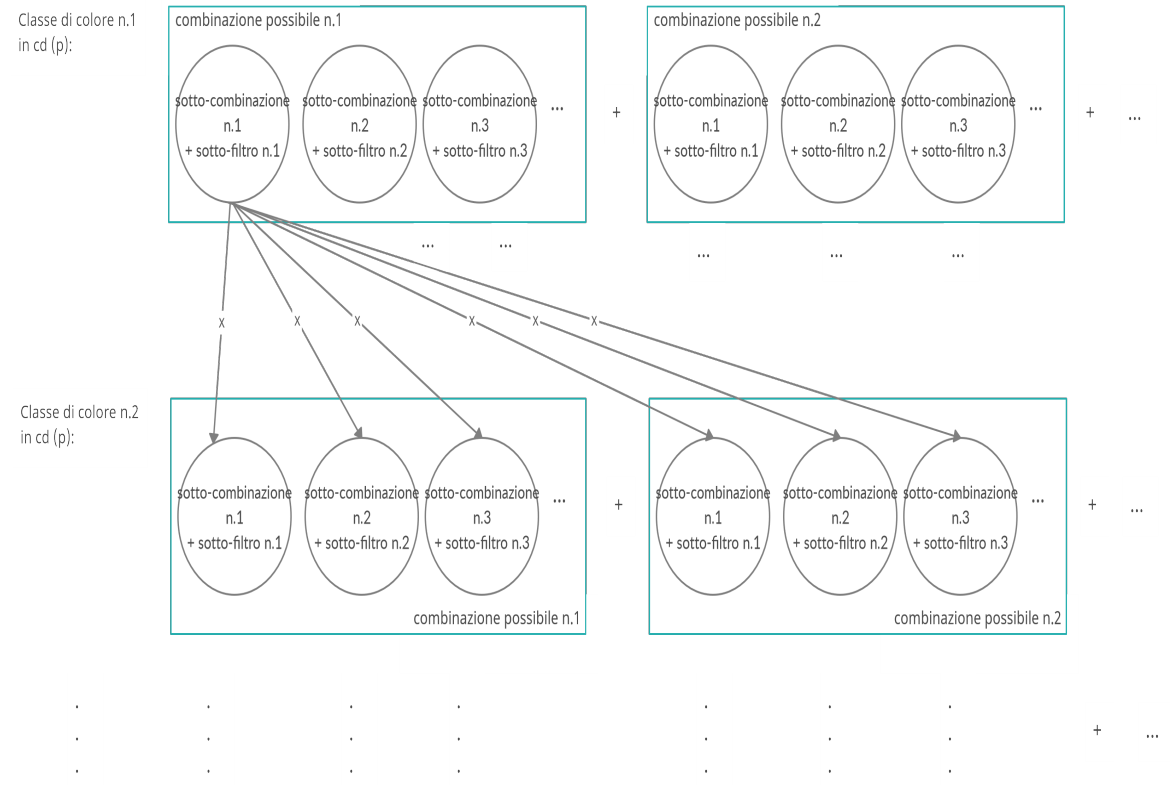

Figura 5.1: Diagramma di sotto-alberi ridondanti in cd(P) e sono risolvibili efficientemente usando la programmazione dinamica

5.2 Osservazioni

Si noti che durante la scrittura del file “pnml”, gli elementi della dichiarazione della rete simmetrica (classi di colore, variabili, domini di classi di colore), verrano copiate dal file “pnml” passato inizialmente alla prima fase di traduzione, poiché non siano stati cambiati. Inoltre, è stato implementato un pacchetto java per stampare/testare le informazioni conservate negli oggetti finali e nelle strutture di dati complesse, poiché si testino le informazioni di (lo scansionatore xml, l’albero sintattico astratto, l’analizzatore semantico, l’oggetto finale di rete simmetrica, generatore del codice “i dati dell’algoritmo unfolding”).
Punto aggiuntivo: Ci possiamo referire anche all’approccio dell’ingegneria software (ref.[8] - ref.[10]) utilizzato per costruire il nostro traduttore, che sarebbe ovviamente l’approccio ”Plan driven”

6. Conclusione

Il principale risultato della tesi è stato lo sviluppo di un pacchetto software per la traduzione del formato PNML delle reti Symmetric Nets (SN) in oggetti analizzabili e manipolabili con la libreria ”SNexpression” che implementa una specie di calcolatore simbolico delle annotazioni delle SN. Si è inoltre sviluppato un modulo che, sfruttando il traduttore, implementa un algoritmo di unfolding parziale per le SN. Questa trasformazione è necessaria per l'uso di tecniche di analisi di modelli di SN stocastiche basate su equazioni differenziali. Il formato in output può essere sia PNML sia il formato proprietario del tool grafico GreatSPN (PNPRO).
I punti che sono stati soddisfatti dal nostro traduttore sono:

- capire ed estrarre le informazioni utili che siano scritte nel file di input.
- classificare le informazioni estratte nelle categorie a cui appartengono tali informazioni.
- trovare le relazioni tra le informazioni provenienti e collegare i dati tra loro.
- creare oggetti connessi tra loro tramite certe associazioni implicite.
- applicare un algoritmo ”unfolding parziale” che elabora tali oggetti connessi in modo efficiente per estrarre altre informazioni utili per la generazione di un'altra rete simmetrica basata su quella iniziale.
- creare un'altra rete simmetrica ed avere l'abilità di scriverla in due formati (PNML,PNPRO).

6.1 Sviluppi futuri

È previsto aggiungere la possibilità di scegliere un file xml in formato "PNPRO" come un file iniziale da cui saranno letti i dati necessari ed i componenti della rete simmetrica, tale che il formato PNPRO è il formato principale del tool grafico "GreatSPN" quindi ci fornirà l'abilità di visualizzare la rete intera. È ulteriormente previsto collegare il nostro traduttore col tool grafico "GreatSPN" per poter usufruire l'interfaccia ad utente e visualizzare direttamente le reti input/output senza aprire nessun file xml. Siccome abbiamo più di un tipo di polimorfismo tra gli oggetti di ("nodo: posto, transizione", "nodo sintattico: posto sintattico, transizione sintattica", …); sarebbe semplicemente possibile applicare il "template" pattern. Per quanto riguarda l'unfolding parziale svolto, è prevista una fase aggiuntiva di ottimizzazione, tale che non sarebbe più necessario generare i posti che sono classificati sotto certe combinazioni che violano transizioni specificate come nodi di livelli precedenti di tali posti; quindi, tali posti ed i loro archi connessi verranno ignorati. Ora si trova la seguente rete simmetrica ottimizzata dopo l'unfolding precitato sopra in "4.2".

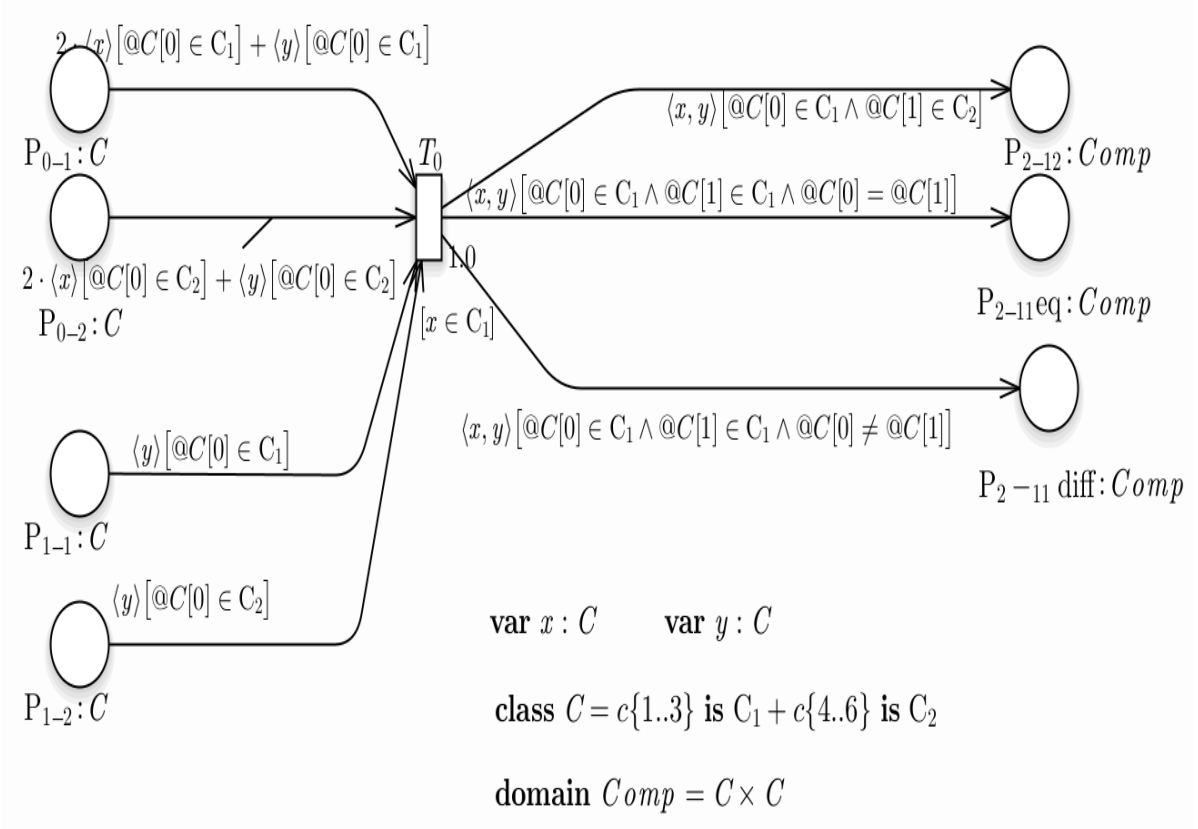

Figura 6.1: optimized Unfolding della rete simmetrica 4.2

La rete simmetrica precedente è stata considerata come una ottimizzazione della rete esibita (4.2); perché siccome la guardia della transizione associata come un livello precedente di “P2” vale a “[x appartiene a C1]”; quindi, la prima variabile “x” non può essere associata a qualsiasi altra costante (es. C2) nelle tuple associate “<x, y>”; poiché non sarà necessario generare i posti (P2-21, P2-22 diff, P2-22 eq) che implichino associare la variabile “x” alla sottoclasse “C2”. Infine, è pianificato fornire un’opzione per poter estrarre solo il file ”PNPRO” equivalente alla rete proveniente dal file ”PNML” di input.

7. Strutture pacchetti e metodi costruiti nel traduttore SN

si discute l'utilizzo delle classi non astratte di ciascun pacchetto tranne il pacchetto "traduttoreSN_wncalculus", mentre i pacchetti secondari "componenti, test, eccezioni" hanno un ruolo ausiliario durante la traduzione e non si devono essere utilizzati fuori il processo di traduzione:

1. struttura_sn (contiene le componenti della rete simmetrica non presenti nella libreria wncalculus):
   - Place: contiene le transizioni connesse a un posto mediante archi di input, output e inibitori; le connessioni sono espresse da mappe che associano a ogni transizione la corrispondente annotazione sull'arco.
     Metodi accessibili essenziali di (Place):
     - Place(name, ColorClass): crea un oggetto posto di tipo classe di colore come cd(p).
     - Place(name, Domain): crea un oggetto posto di tipo dominio come cd(p).
   - Marking: contiene una mappa che associa ai posti inizialmente marcati delle tuple di combinazioni lineari (LinearComb di wncalculus) di "token" corrispondenti ai domini dei posti.
     Metodi accessibili essenziali di (Marking):
     - get_instance(): crea un oggetto unico di marcatura oppure restituisce l'oggetto creato precedentemente.
   - Transition: analogamente a "Place", contiene tre mappe duali che descrivono I posti adiacenti.
     Metodi accessibili essenziali di (Transition):

  - Transition(name, Guard): crea un oggetto transizione che contiene una certa guardia.

- ArcAnnotation: contiene l'identificatore e l'espressione d'arco (TupleBag di wncalculus).
  Metodi accessibili essenziali di (ArcAnnotatio):
  - ArcAnnotation(name, TupleBag): crea un oggetto d'arco ed associare la sua espressione ad esso.

- Token: contiene una stringa del suo valore; inoltre, ha una relazione di eredità con la classe (ElementaryFunction di wncalculus).
  Metodi accessibili essenziali di (Token):
  - Token(value, ColorClass): crea un oggetto di color token ed associa il suo tipo (classe di colore) ad esso.

- Variable: contiene il nome dichiarato, il suo tipo di classe di colore e una lista di tutte le proiezioni / variabili locali che saranno definite sulle tuple d'arco o le guardie della rete.
  Metodi accessibili essenziali di (Variabile):
  - Variable(name, ColorClass): crea un oggetto di variabile ed associa il suo tipo (classe di colore) ad esso.
  - get_available_projection(index, int successor_flag): restituisce l'istanza di variabile (projection) già creata precedentemente in mode dipendente dai parametri passati.

- SN: l'oggetto finale della rete simmetrica scansionata ed analizzata che contiene tutti i dati della rete (lista di posti, lista di transizioni, lista di classi di colore, lista di domini, lista di variabili, l'oggetto unico della marcatura iniziale); inoltre ci sono i metodi necessari per modificare tali liste o aggiornare gli elementi in esse.
  Metodi accessibili essenziali di (SN):
  - get_instance(): crea un oggetto unico di rete simmetrica "sn" oppure restituisce l'oggetto creato precedentemente.
  - update_place(place): aggiorna il posto che ha lo stesso nome di tal parametro.
  - update_transition(transition): aggiorna la transizione che ha lo stesso nome di tal parmetro.
  - update_variable_via_projection(variable): aggiorna la variabile tramite lo stesso nome, a causa di recenti creazioni di nuove istanze di variabile appartenenti a tale variabile.

2. fasi_traduttore (contiene le fasi principali del nostro traduttore):

   - XMLScanner: scansiona il file specificato come un documento xml e suddivide gli elementi in diverse liste dipendentemente dalle loro categorie; poi chiama ciascun sotto-scansionatore per elaborare la sua lista di elementi.
     Metodi accessibili essenziali di (XMLScanner):
     - get_instance(file_address): crea un oggetto unico di XMLScanner oppure restituisce l'oggetto creato precedentemente; inoltre, si associa l'indirizzo del file che si vuole importarlo.
     - scan_file_data(): scansiona in ordine gli elementi di (classi di colore, domini, variabili, posti, transizioni, archi).
   - DataParser: scompone le strutture di dati provenienti dai sotto-scansionatori, poiche' i dati estratti verranno inseriti in nuovi oggetti che appartengono all'albero sintattico astratto, oppure alla rete simmetrica "SN" se non è necessario analizzarli semanticamente.
     Metodi accessibili essenziali di (DataParser):
     - get_instance(): crea un oggetto unico di DataParser oppure restituisce l'oggetto creato precedentemente.
     - add_ColorClass(class_name, start, end, circular): aggiunge direttamente alla SN un oggetto "ColorClass" che sia implicitamente espressa tramite un intervallo finito su prefisso; inoltre, si associa un flag di che dimostra se la classe sia circulare o non lo è.
     - add_ColorClass(class_name, token_names, circular): aggiunge direttamente alla SN un oggetto "ColorClass" che sia esplicitamente espressa tramite un intervallo finito su prefisso; inoltre, si associa un flag di che dimostra se la classe sia circulare o non lo è.
     - add_ColorClass(class_name, subclasses, circular): aggiunge direttamente alla SN un oggetto "ColorClass" che sia partizionata in un certo numero di sottoclassi; inoltre, si associa un flag di che dimostra se la classe sia circulare o non lo è.
     - add_Variable(variable_name, variable_type): aggiunge direttamente alla SN un oggetto "Variable" associando il suo tipo "classe di colore" ad essa.
     - add_Domain(domain_name,colorclasses): aggiunge direttamente alla SN un oggetto "Domain" associando il suo tipo le classi di colore appartenenti ad esso.
     - add_Place(place_name, place_type): aggiunge all'albero sintattico astratto "AST" un nodo posto; inoltre, si associa il suo tipo "classe di colore/dominio" ad esso.

  - add_Marking(place_name, tokens): registra una marcatura iniziale su un posto, utilizzando i tokens passati come parametro.
  - add_Transition(name, guard, invert_guard): aggiunge all'albero sintattico astratto "AST" un nodo transizione; inoltre, si associa la sua guardia (se esiste) ad esso.
  - add_Arc(arc_name, arc_type, from, to, guards, invert_guards, tuples_elements, tuples_mult, filters, invert_filters): aggiunge all'albero sintattico astratto un arco (normale/inibitore); inoltre, si associa la sua espressione ad esso rappresentata nel seguente formato ordinato "(inverted)guard, (multiplied)tuple, (inverted)filter".

- SemanticAnalyzer: analizza il dominio di ciascun nodo nell'albero sintattico astratto; poi passa le "guardie, tuple" ai loro sotto-analizzatori; inoltre, costituisce le espressioni di archi e connette i nodi nella rete finale "SN" usando gli archi analizzati.
  Metodi accessibili essenziali di (SemanticaAnalyzer):
  - get_instance(): crea un oggetto unico di SemanticaAnalyzer oppure restituisce l'oggetto creato precedentemente.
  - analyze_syntax_tree(): analizza tutti i nodi dell'albero sintattica astratto "AST" estraendo i domini di nodi e tuple; inoltre, si analizzano le istanze di variabile (projection), le costanti, le funzioni di classe, i collegamenti tra nodi creando gli oggetti necessari di essi.
- PartialGenerator: applica l'algoritmo unfolding su ciascun posto nella rete simmetrica "SN" ed usa la programmazione dinamica per applicare il prodotto cartesiano sui filtri/combinazioni di diverse classi di cd(p); poi utilizza "XMLWriter" per creare le liste necessari di ciascun componente della rete dopo l'unfolding.
  Metodi accessibili essenziali di (PartialGenerator):
  - get_instance(): crea un oggetto unico di Semantic PartialGenerator oppure restituisce l'oggetto creato precedentemente.
  - unfold_all_places(): applica l'algoritmo unfolding su tutti i posti esistenti nell'oggetto unico "SN".
- XMLWriter: scrive i dati conservati nelle liste diverse in elementi xml in due files diversi (pnml/PNPRO); il formato "PNPRO" è utilizzabile per visualizzare la rete output sul tool "GreatSPN".
  Metodi accessibili essenziali di (XMLWriter):
  - get_instance(): crea un oggetto unico di Semantic XMLWriter oppure restituisce l'oggetto creato precedentemente.
  - write_all_data(): scrive tutti gli elementi creati dall'algoritmo unfolding in file pnml.

  - write_all_data_pnpro(write_in_pnpro): salva la rete simmetriche output dall'algoritmo unfolding in file pnpro; inoltre, salve la rete orginale nello stesso file pnpro; perciò si utilizza un flag come un parametro per determinare il momento di creazione del file.

3. scanner (contiene i sotto-scansionatori che scansionano i componenti diversi della rete simmetrica):

   - ColorClass_scanner: scansiona tutti gli elementi di classi di colore esistenti nella lista riempita dal XMLScanner.
     Metodi accessibili essenziali di (ColorClass_scanner):
     - get_instance(): crea un oggetto unico di Semantic ColorClass_scanner oppure restituisce l'oggetto creato precedentemente.
     - scan_info(ColorClass_element): scanizza tutti i dati di un elemento di classe di colore scritto in formato xml/pnml.
   - Domain_scanner: scansiona tutti gli elementi di domini di classi di colore esistenti nella lista riempita dal XMLScanner.
     Metodi accessibili essenziali di (Domain_scanner):
     - get_instance(): crea un oggetto unico di Semantic Domain_scanner oppure restituisce l'oggetto creato precedentemente.
     - scan_info(Domain_element): scanizza tutti i dati di un elemento di dominio scritto in formato xml/pnml.
   - Variable_scanner: scansiona tutti gli elementi di variabili esistenti nella lista riempita dal XMLScanner.
     Metodi accessibili essenziali di (Variable_scanner):
     - get_instance(): crea un oggetto unico di Semantic Variable_scanner oppure restituisce l'oggetto creato precedentemente.
     - scan_info(Variable_element): scanizza tutti i dati di un elemento di variabile scritto in formato xml/pnml.
   - Place_scanner: scansiona tutti gli elementi di posti esistenti nella lista riempita dal XMLScanner.
     Metodi accessibili essenziali di (Place_scanner):
     - get_instance(): crea un oggetto unico di Semantic Place_scanner oppure restituisce l'oggetto creato precedentemente.

  - scan_info(Place_element): scanizza tutti i dati di un elemento di place scritto in formato xml/pnml.

- Marking_scanner: viene chiamato dal (Place_scanner) per scansionare la marcatura iniziale collegata ad un certo posto.
  Metodi accessibili essenziali di (Marking_scanner):
  - get_instance(): crea un oggetto unico di Semantic Marking_scanner oppure restituisce l'oggetto creato precedentemente.
  - scan_info(Marking_element): scanizza tutti i dati di un elemento di marcatura scritto in formato xml/pnml.
- Transition_scanner: scansiona tutti gli elementi di transizioni esistenti nella lista riempita dal XMLScanner.
  Metodi accessibili essenziali di (Transition_scanner):
  - get_instance(): crea un oggetto unico di Semantic Transition_scanner oppure restituisce l'oggetto creato precedentemente.
  - scan_info(Transition_element): scanizza tutti i dati di un elemento di transizione scritto in formato xml/pnml.
- Guard_scanner: viene chiamato dal (Transition_scanner/Tuple_scanner) per scansionare la guardia collegata ad una certa transizione oppure ad una certa tupla d'arco.
  Metodi accessibili essenziali di (Guard_scanner):
  - get_instance(): crea un oggetto unico di Semantic Guard_scanner oppure restituisce l'oggetto creato precedentemente.
  - scan_info(Guard_element): scanizza un elemento guard scritto in formato xml/pnml.
- Predicate_scanner: viene chiamato dal (Guard_scanner) per scansionare un predicato esistente in una certa guardia.
  Metodi accessibili essenziali di (Predicate_scanner):
  - get_instance(): crea un oggetto unico di Semantic Predicate_scanner oppure restituisce l'oggetto creato precedentemente.
  - scan_predicate(predicate): scanizza tutti i dati di predicato scritto in una stringa estratta da un elemento di guardia dal file pnml.
- Arc_scanner: scansiona tutti gli elementi di archi esistenti nella lista riempita dal XMLScanner.
  Metodi accessibili essenziali di (Arc_scanner):

  - get_instance(): crea un oggetto unico di Semantic Arc_scanner oppure restituisce l'oggetto creato precedentemente.
  - scan_info(Arc_element): scanizza tutti i dati di un elemento d'arco scritto in formato xml/pnml.
- Tuple_scanner: viene chaimato dal (Arc_scanner) per scansionare una tupla esistente in una certa espressione d'arco.
  Metodi accessibili essenziali di (Tuple_scanner):
  - get_instance(): crea un oggetto unico di Semantic Tuple_scanner oppure restituisce l'oggetto creato precedentemente.
  - scan_tuple(tuple): scanizza tutti i dati di tupla scritta in una stringa estratta da un elemento d'arco dal file pnml.

4. albero sintattico (contiene i componenti della rete intermedia che sarebbe necessario analizzarli):

- Syntactic_place: è il posto intermedio nell'albero sintattico astratto che viene creato per completare il collegamento della rete intermedia.
  Metodi accessibili essenziali di (Syntactic_place):
  - Syntactic_place(name): crea un oggetto di posto sintattico per l'albero sintattico astratto "AST".
- Syntactic_transition: è la transizione intermedia nell'albero sintattico astratto che viene creato per completare il collegamento della rete intermedia.
  Metodi accessibili essenziali di (Syntactic_transition):
  - Syntactic_transition(name): crea un oggetto di transizione sintattica per l'albero sintattico astratto "AST"; inoltre, si utilizzano i metodo set/get per assegnare/restituire la guardia se esiste.
- Syntactic_guard: è la guardia intermedia di una (transizione sintattica/tupla sintattica) che verrà analizzata dal suo analizzatore specificato.
  Metodi accessibili essenziali di (Syntactic_guard):
  - Syntactic_guard(invert_guard, separated_predicates): crea un oggetto di guardia sintattica per l'albero sintattico astratto "AST"; inoltre, i predicati separati sono utilizzabili per creare i predicati sinattici appartenenti a tale guardia sintattica.
- Syntactic_predicate: è il predicato intermedio di una (guardia sintattico) che verrà analizzato dal suo analizzatore specificato.
  Metodi accessibili essenziali di (Syntactic_place):

  - Syntactic_predicate(invert_guard, predicate_elements): crea un oggetto di posto sintattico per l'albero sintattico astratto "AST".

- Syntactic_arc: è l'arco intermedio che collega due nodi intermedi (posto sintattico/ transizione sintattica), tali che verranno collegati semanticamente dall'analizzatore semantico.
  Metodi accessibili essenziali di (Syntactic_arc):
  - Syntactic_arc(name): crea un oggetto d'arco sintattico per l'albero sintattico astratto "AST"; inoltre, si utilizzano i metodi set/get per assegnare/restituire l'espressione d'arco, e il tipo d'arco.
- Syntactic_tuple: è la tuple intermedia di un arco, cioè l'espressione caricata da tal arco, tale tuple sintattica verra' analizzata dal suo analizzatore specificato.
  Metodi accessibili essenziali di (Syntactic_tuple):
  - Syntactic_tuple(tuple_elements): crea un oggetto di tupla sintattica per l'albero sintattico astratto "AST".
- SyntaxTree: è l'albero sintattico astratto che sarà analizzato per sapere i domini di transizioni e di tuple di archi.
  Metodi accessibili essenziali di (SyntaxTree):
  - get_instance(): crea un oggetto unico di SyntaxTree oppure restituisce l'oggetto creato precedentemente.

5. analyzer (contiene i sotto-analizzatori che analizzano certi elementi della rete "le guardie, le proiezioni/variabili, le costanti, le tuple"):

   - Guard_analyzer: è l'analizzatore delle guardie sintattiche (intermedie) che analizza inoltre i predicati sintattici esistenti in esse, tal analizzatore suddivide i dati sintattici (intermedi) di una guardia sintattica ai loro sotto-analizzatori di (tupla, costante, proiezione).
     Metodi accessibili essenziali di (Guard_analyzer):
     - get_instance(): crea un oggetto unico di Guard_analyzer oppure restituisce l'oggetto creato precedentemente.
     - analyze_guard_of_predicates(syntactic_guard, transition_name, tuple_vars_names, transition_domain): analizza la guardia collegata ad una certa transizione tramite un arco oppure tramite la transizione stessa.
   - Tuple_analyzer: analizza le tuple sintattiche esistenti nell'espressione d'arco oppure nelle tuple di marcatura iniziale.
     Metodi accessibili essenziali di (Tuple_analyzer):

  - get_instance(): crea un oggetto unico di Tuple_analyzer oppure restituisce l'oggetto creato precedentemente.
  - analyze_arc_tuple(guard, filter, tuple_elements, transition_name, place_name, dominio_transizione): analizza una funzione di espressione d'arco che contiene queste informazioni "guardia,tupla,filtro"; inoltre, il dominio è necessario per creare la tupla desiderata usando la libreria "Wncalculus".

- Constant_analyzer: analizza le costanti esistenti nelle tuple sintattiche o nelle guardie sintattiche.
  Metodi accessibili essenziali di (Constant_analyzer):
  - get_instance(): crea un oggetto unico di Constant_analyzer oppure restituisce l'oggetto creato precedentemente.
  - analyze_constant_element(constant_name): analizza ogni costante (classe di colore/sottoclasse di colore) che esiste in una tupla/guardia.
- Projection_analyzer: analizza le proiezioni esistenti nelle tuple sintattiche o nelle guardie sintattiche.
  Metodi accessibili essenziali di (Projection_analyzer):
  - get_instance(): crea un oggetto unico di Projection_analyzer oppure restituisce l'oggetto creato precedentemente.
  - analyze_projection_element(proj_name, transition_name, tuple_vars_names, transition_domain): analizza ogni proiezione/istanza di variabile che esiste in una tupla/guardia.

6. componenti (contiene le tabelle necessarie per l'analisi generale o specialmente l'analisi semantica):

   - ColorClass_tokens: ha una mappa che contiene le classi di colori esplicite a cui i loro tokens sono associati
   - Marking_tokens: contiene tutti i tokens marcati che verranno utilizzati invece di creare nuovi tokens simili.
   - Place_syntax_table: ha due mappe, la prima contiene tutti i posti con i loro tipi (classe di colore o domini), la secondo contiene tutti tali tipi con la loro sintassi di scrittura; inolte, si usa questa tabella per sapere la sintassi di un posto durante la creazione di una funzione di class (es. All).
   - Variable_index_table: contiene tutti gli indici di variabili creati attorno una certa transizione, tale che viene restituito l'indice precreate nel caso di trovare la stessa variabile attorno la stessa transizione.

- Subcc_token_index_table: contiene I limiti di ciascuna sottoclasse implicitamente espressa e non sia parmetrica.
- Token_estimator: stima I tokens di colore di una certa classe o sottoclasse di colore, solo se sia esplicitamente espressa usando un intervallo finito.

7. test (contiene i testers diversi che testano i dati passanti su ciascuna fase di traduzione):

   - XML_DataTester: si utilizza per stampare tutti di dati scansionati dal file xml (classi di colore, domini, variabili, posti, transizioni, archi, marcatura iniziale).
   - SyntaxTree_DataTester: si utilizza per stampare tutti i componenti della rete inizialmente scansionata con i collegamenti tra loro.
   - Semantic_DataTester: si utilizza per stampare i dati semantici analizzati (es. guardia analizzata, tupla analizzata, proiezione /variabile analizzata, costante analizzata).
   - SN_DataTester: si utilizza per stampare tutti i componenti della rete simmetrica dopo la scansione e l'analisi sintattica e semantica.
   - PartialGenertor_DataTester: si utilizza per stampare le procedure dell'algoritmo unfolding.

8. eccezioni (contiene le eccezioni espressive di certe violazioni o dati inappropriati):

   - BreakConditionException: si lancia quando si vuole terminare un loop funzionale.
   - UnsupportedElementNameException: si lancia quando il nome di un elemento di input non è desiderato (es. Nuova classe di colore col nome "Neutral" mentre tal nome è già riservato).
   - UnsupportedElementdataException: si lancia quando certi dati di un elemento sono sono accettabili, tale elemento dovrebbe essere scritto in file xml dopo l'applicazione di unfolding.
   - UnsupportedFileException: si lancia quando il file di input non è un file "pnml".
   - UnsupportedLinearCombException: si lancia quando un elemento di combinazione lineare non è desiderato (es. aggiungere una variabile in un token di marcatura iniziale).
   - UnsupportedPrediateOperationException: si lancia quando l'operazione di un predicato non è desiderato (es. >).
   - traduttoreSN_wncalculus (contiene il programma principale che convalida il nome del file che si vuole utilizzarlo e chiama in modo ordinato le fasi diverse con i loro testers se sia necessario):

- traduttoreSN_wncalculus (funzione essenziale main).

Nella prossima pagina si trova il diagramma di pacchetti, per dimostrare le dipendenze tra i pacchetti diversi utilizzati 7.1. *Inoltre, si potrebbe rivolgere alla struttura reale del codice su github (ref.[9]).*

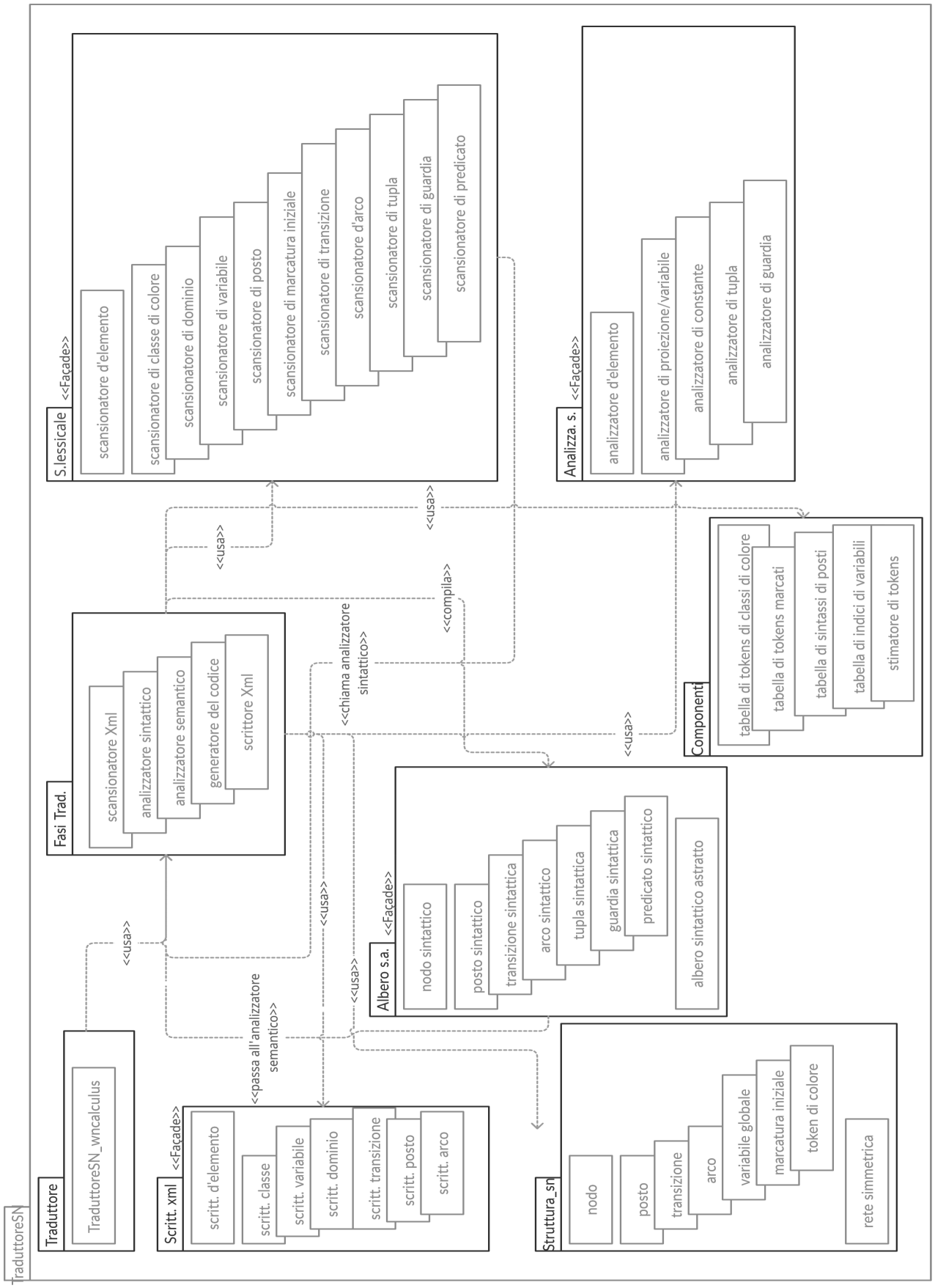

Figura 7.1: Diagramma di pacchetti del nostro traduttore SN

8. Manuale utente

Per esguire il traduttore, è necessario seguire le seguenti istruzioni usando il shell.

1. Scrivere il seguente comando per compilare il file “jar” del traduttore: java -jar Traduttore-SN_wncalculus.jar
2. Dare l’indirizzo del file “pnml” che contiene la rete simmetrica che si desidera importarla: xml-example.pnml
3. Ora si possono vedere i messaggi del debugging durante la creazione di oggetti necessari usando le informazioni contenenti nel file pnml.
4. Quando l’oggetto principale “SN” della rete simmetrica è correttamente creato, si richiede di digitare “1” per applicare l’algoritmo unfolding su tale rete; altrimenti, si crea un file “pnpro” della rete precedentemente importata.
5. Quando si inizia l’applicazione dell’algoritmo unfolding, si possono vedere i messaggi del debugging per esporre i posti originali diversi ed i posti corrispondentemente generati e correlati ad essi.
6. Dopo l’applicazione dell’algoritmo unfolding, si genera un file “pnml” che contiene la rete output; inoltre, si genera un file “pnpro” che contiene la rete output e la rete originale per confrontarle insiemi se è necessario.

Bibliografia

[1] Elvio Amparore. «GreatSPN». In: *Github* (2021). URL: https://github.com/greatspn/SOURCES.

[2] G. Chiola et al. «Stochastic well-formed colored nets and symmetric modeling applications». In: *IEEE Transactions on Computers* 42.11 (1993), pp. 1343–1360. DOI: 10.1109/12.247838.

[3] G. Chiola; C. Dutheillet; G. Franceschinis; e S. Haddad. *"Stochastic well-formed coloured nets and multiprocessor modeling applications," in High-Level Petri Nets; Theory and Application; K. Jensen and G.Rozenberg*. illustrated. Springer Verlag, 1993.

[4] Jeffrey Friedl. *Mastering regular expressions*. illustrated. O'reilly, 2006.

[5] G. Balbo; L. Capra; S. Donatelli; G. Franceschinis; M. Ribaudo; M. Sereno; A. Ferscha; J. Campos; J. M. Colom; M. Silva; E. Teruel G. Chiola; B. Baynat; Y. Dallery; C. Dutheillet; S. Haddad. *Performance Models for Discrete Event Systems with Synchronisations: formalisms and analysis techniques (Match)*. illustrated. Springer, 1998.

[6] Steven john metsker. *Design patterns Java workbook*. illustrated. Addison-Wesley, 2002.

[7] Warren B. Powell. *Approximate Dynamic Programming: solving the curses of dimensionality*. illustrated. Princeton, 2008.

[8] Roger s. Pressman. *Software engineering: a practitioner's approach*. 7ª ed. McGraw-Hill, 2009.

[9] Abdelrahman Sobh. «Traduttore$\mathrm{SN}_{w}ncalculus$». In: *Github* (2021). URL: https://github.com/Abudo-S/TraduttoreSN_wncalculus.

[10] Lan sommerville. *Software engineering*. 10ª ed. Pearson, 2016.

[11] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein. *MIT: Introduction to algorithms*. 3ª ed. McGraw-Hill, 2009.

[12] Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides. *Elements of reusable object-oriented software*. illustrated. Addison-Wesley, 1995.

[13] V. Volovoi et al. «Modeling the reliability of distribution systems using Petri nets». In: *2004 11th International Conference on Harmonics and Quality of Power (IEEE Cat. No.04EX951)*. 2004, pp. 567–572. DOI: 10.1109/ICHQP.2004.1409416.

Se desideri citare questo lavoro, la annotazione completa in BibTeX è la seguente:

```
@thesis{citation_key,
author="Abdelrahman Sobh"
title="Traduzioni di Formato e Unfolding Parziale di Reti di Petri Simmetriche",
school="UNIMI, Facoltà di Scienze e Tecnologie, Dipartimento di informatica",
year=2021
}
```