Reti di Petri colorate : realizzazione di un traduttore Prolog/Promela per l'analisi con il model checker Spin

Realizzato da

Davide Galeotti

per il corso di

Linguaggi e Modelli Computazionali LM

Università degli Studi di Bologna - Seconda Facoltà di Ingegneria

A.A. 2011-2012

Indice

1 Introduzione alle reti di petri colorate

1.1 Elementi principali.

Le reti di petri colorate (CPN) rappresentano un formalismo grafico che supporta il programmatore nella modellazione degli aspetti relativi alla concorrenza e alla sincronizzazione delle attività all'interno di sistemi anche complessi, come possono essere i moderni sistemi distribuiti o concorrenti. Le CPN, inoltre, ben si prestano alla verifica formale sul modello di proprietà, come quelle di safety e liveness, tramite le logiche LTL(Linear Temporal Logic) (1), CTL (Computational Tree Logic) e affini.

In questo formalismo i sistemi vengono rappresentati in termini di *place* e *transizioni* connessi fra loro tramite *archi* bidirezionali (2). Ciascun place può contenere un insieme di *markers* chiamati anche *token,* ed ognuno di essi porta con sè un "colore", cioè un'informazione dello stesso tipo indicato dal place in cui si trovano.

In certo istante lo stato del sistema è rappresentato dal *marking* cioè dalla disposizione di tutti i token nei diversi place. Questo stato può cambiare a seguito dell'esecuzione di un'azione modellata come una transizione. Lo scatenarsi di una transizione *t*, ad esempio, provoca la rimozione dei token da tutti i place con un arco entrante in *t* ed il posizionamento di un nuovo token in tutti i place connessi a *t* con un arco uscente da quest'ultima. Le informazioni dei token in output a *t* derivano da una qualsiasi elaborazione di quelli in input purchè il tipo di dato sia lo stesso del place di destinazione.

Nell'esempio che segue (Figura 1) sono presenti tutti gli elementi precedentemente citati. I place e le transizioni sono rappresentati rispettivamente da ellissi e rettangoli, ciascun elemento è identificato univocamente dal nome indicato sopra. Detto ciò individuiamo una transizione *t* e tre place, *p0*,*p1* e *p2*. Sotto ciascun place è riportato inoltre il tipo di informazione che i token presenti possono contenere: in *p0* possiamo trovare numeri interi, in *p1* le stringhe ed in *p2* l'insieme costituito da tutte le possibili coppie tra interi e stringhe. Il marking iniziale è stato indicato all'interno dei rettangoli con gli angoli stondati, la cui dicitura va interpretata come #token ' informazione associata + etc.. . Ad esempio in *p0* troviamo un token a cui è associata l'informazione 2, un token con 7 e uno con 4.

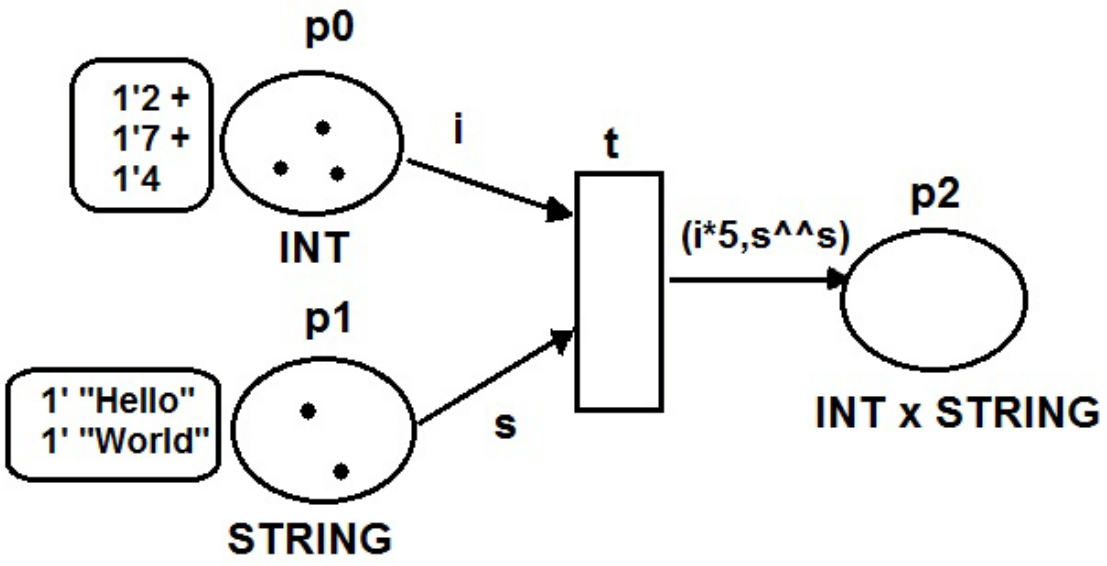

Figura 1 : Un semplice esempio di CPN.

1.2 Comportamento di una CPN.

Al verificarsi dell'azione t, un token viene prelevato da *p0* e un'altro da *p1*. Supponiamo quindi che i valori dei token rimossi siano rispettivamente 2 ed "Hello" allora questi, a seguito di un'operazione di binding, verrano assegnati alle variabili di supporto i ed s, definendo il *binding element* <t, i=2, s="Hello">. Un nuovo token verrà ora inserito in p2, le cui informazioni derivano da un'elaborazione di quelle contenute nei token in input, in questo caso sono state utilizzate le operazioni di moltiplicazione tra interi (operatore *) e di concatenazione tra stringhe (operatore ^^) .

L'informazione del token risultante sarà quindi rappresentata dalla coppia (10,"HelloHello"). Il nuovo stato della rete sarà definito dal marking osservabile in Figura 2.

Osservando reti più complesse, come quelle che vedremo in seguito(es:Figura 1Figura 3), ci si può rendere conto del fatto che più transizioni possono essere abilitate a partire da un determinato marking, perchè tutti i place di input presentano token, ma soltanto una si verifica, cioè verrà scelta come azione. Per essere abilitata una transizione deve inoltre soddisfare anche le guardie, condizioni sui valori dei token.

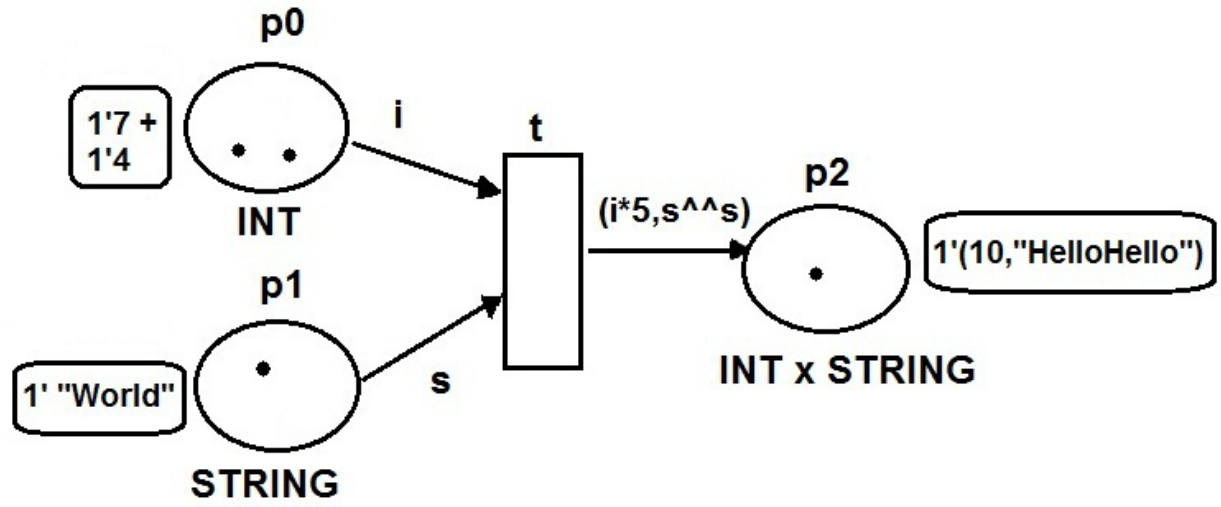

Figura 2: Nuovo marking a seguito dell'esecuzione di t.

2 Definizione formale di una CPN.

La definizione formale (3) delle reti di Petri colorate (non gerarchiche), viene qui presentata con il solo scopo di rendere non ambigui i concetti precedentemente espressi in maniera informale. A tale scopo adottiamo una notazione a tuple che, nonostante sia equivalente alla notazione grafica, meglio si presta alla formulazione di definizioni generali e alla dimostrazione di teoremi validi per qualsiasi CPN.

Per iniziare viene presentata la definizione di multi-set:

Un *multi-set* m, di un insieme non vuoto S, è una funzione m ϵ [S $\rightarrow \mathbb{N}$] che può essere rappresentata come la sommatoria:

$$\sum_{s \in S} m(s)\,'s.$$

Con S_{MS} si denota il set di tutti i multi-set costruibili partendo da S. Gli interi non negativi {m(s) | s ϵ S} sono i coefficienti del multi-set. s ϵ m iff m(s)≠0.

N inoltre denota l'insieme dei numeri naturali mentre "iff" significa "se e solo se". Il multi-set vuoto invece viene indicato con il simbolo $\emptyset$.

Vengono ora definite alcune operazioni possibili sui multi-set.

$\forall\, m, m1, m2 \in S_{MS}, \forall n \in \mathbb{N}$:

i. $m1 + m2 = \sum_{s \in S}\big(m1(s) + m2(s)\big)'s$. (Addizione)

ii. $n * m = \sum_{s \in S}(n * m(s))'s$. (Moltiplicazione scalare)

iii. $m1 \neq m2 = \exists s \in S: m1(s) \neq m2(s)$. (Disuguaglianza, in modo analogo si definisce "=")

iv. $m1 \leq m2 = \forall s \in S: m1(s) \leq m2(s)$. (Minore uguale, in modo analogo si definsce "≤")

v. $|m| = \sum_{s \in S} m(s)$. (Dimensione di un multi-set)

Quando $|m| = \infty$ si dice che m è infinito. Altrimenti m è finito. Quando $m1 \leq m2$ è possibile definire anche la sottrazione:

vi. $m2 - m1 = \sum_{s \in S}(m2(s) - m1(s))'s$. (Sottrazione)

Le operazioni sui multi-set hanno diverse proprietà algebriche standard (commutativa,distributiva, associativa, etc).

Prima di mostrare la definizione più importante, cioè quella di una CPN, è necessario assumere che la sintassi concreta, con cui il programmatore specifica le espressioni

delle reti, esista e che abbia una semantica ben definita. E' perciò necessario definire in modo non ambiguo anche i concetti di:

- Il tipo di un elemento, T. Il set di tutti gli elementi in T è denotato dal nome T stesso.
- Il tipo di una variabile v, indicato con Type(v).
- Il tipo di una espressione expr, indicato con Type(expr).
- L'insieme delle variabili in una espressione expr, indicato con Var(expr).
- Un assegnamento (*binding*) di un set di variabili V, cioè l'associazione ad ogni variabile v ϵ V di un elemento b(v) ϵ Type(v).
- Il valore ottenuto dalla valutazione di una espressione expr, in a binding b, indicato con expr<b>. L'insieme Var(expr) deve essere un sottoinsieme delle variabili di b e la valutazione viene effettuata sostituendo ad ogni variabile v ϵ Var(expr) il valore b(v) ϵ Type(v) determinato dal binding.

Date tali premesse è ora possibile definire una rete di Petri colorata:

Una rete di Petri colorata è una tupla CPN=(∑ ,P, T, A, N, C, G, E, I) in cui:

i. ∑ è un insieme finito non vuoto di tipi, anche chiamato "colour set" o insieme dei colori.
ii. P è un insieme finito di place.
iii. T è un insieme finito di transition.
iv. A è un insieme finito di archi tale che:
    - $P \cap T = P \cap A = T \cap A = \emptyset$.
v. N, C, G sono funzioni denominate rispettivamente fz nodo, fz colore, fz guardia.
vi. E è la funzione che rappresenta l'espressione relativa ad un arco.
vii. I è la funzione di inizializzazione.

(i) Il set di tipi determina i valori dei dati, le operazioni e le funzioni che possono essere impiegate nelle espressioni della rete (espressioni degli archi, guardie ed espressioni di inizializzazione).

(ii,iii,iv) Place, transizioni ed archi sono definiti dai set P,T e A i quali devono essere finiti e disgiunti a due a due. Facendo in modo che i diversi set siano finiti si evitano numerosi problemi come la possibilità di avere un numero infinito di archi tra due nodi.

(v) La funzione nodo mappa ogni arco in una coppia in cui il primo elemento è il nodo sorgente mentre il secondo è il nodo destinazione. I due nodi devono essere di tipo differente (uno deve essere un place mentre l'altro deve essere una transizione) ma è permesso avere archi multipli tra la stessa coppia di nodi. La funzione colore C mappa

ogni place p in un tipo C(p). Intuitivamente questo significa che ogni token in p deve avere un valore che appartiene a C(p). La funzione guardia G mappa ogni transizione t in un'espressione booleana dove tutte le variabili hanno tipi che appartengono a $\sum$. Quando si progetta una CPN si omettono le espressioni delle guardie che hanno sempre il valore "true".

(vi) La funzione E mappa ciascun arco a in un'espressione del tipo $C(p)_{MS}$. Ciò significa che ciascuna espressione sugli archi va valutata in un multi-set con dello stesso tipo del place p adiacente. E' permesso avere solamente una espressione per ciascun arco.

(vii) La funzione di inizializzazione I mappa ogni place p in un'espressione ("closed") che deve essere del tipo $C(p)_{MS}$. Quando si progetta una CPN si omettono le espressioni di inizializzazione la cui valutazione è il set vuoto $\emptyset$.

Ora che è stata definita la struttura degli elementi in rete di Petri colorata è più semplice comprenderne il comportamento ,ma non prima di aver introdotto le seguenti notazioni valide $\forall\, t \in T$ e $\forall\, (x_1, x_2) \in (P \times T \cup T \times P)$:

- $A(t) = \{a \in A | N(a) \in P \times \{t\} \cup \{t\} \times P\}$
- $Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t): v \in Var(E(a))\}$
- $A(x_1, x_2) = \{a \in A | N(a) = (x_1, x_2)\}$
- $E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$

Di seguito viene presentata la definizione di *binding* e di *elemento del binding*:

Il binding di una transizione t è una funzione b definita su Var(t) tale che:

(i) $\forall v \in Var(t): b(v) \in Type(v)$.
(ii) $G(t) < b >$.

Con B(t) viene indicato il set di tutti i possibili binding per t.

Va notato che la (ii) implica il fatto che tutti i binding debbano soddisfare la guardia corrispondente. G(t)<b> denota infatti la valutazione dell'espressione relativa alla guardia G(t) in un determinato binding b.

Un elemento token è una coppia (p,c) dove p ϵ P e c ϵ C(p), diversamente dall'elemento del binding che è una coppia (t,b) dove t ϵ T e b ϵ B(t). Il set di tutti gli elementi token viene indicato con TE mentre il set di tutti i binding element viene denotato con BE.

Un marking è un multi-set di TE mentre uno step è un multi-set non vuoto e finito di BE . Il marking iniziale M_0 è il marking ottenuto dalla valutazione dell'espressione di inizializzazione:

$$\forall (p, c) \in TE: M_0(p, c) = \big(I(p)\big)(c).$$

Ogni marking Mϵ TE_{MS} determina una funzione univoca M* definita su P tale che M*(p)ϵ$C(p)_{MS}$:

$$\forall p \in P, \forall c \in C(P): \big(M^*(P)\big)(c) = M(p,c).$$

D'altro canto, ogni funzione M*, definita su P tale che M*(p)ϵ$C(p)_{MS}$ $\forall\, p \in P$, determina un marking univoco M:

$$\forall (p,c) \in TE: M(p,c) = (M^*(p)(c).$$

Detto ciò si è pronti a comprendere il concetto di *attivazione* e di *occorrenze.*

Uno step Y si dice *attivo* in un marking M iff la seguente proprietà soddisfatta:

$$\forall p \in P: \sum_{(t,b)\in Y} E(p,t) < b > \leq M(p)$$

Quindi anche la coppia (t,b) si dice che è attiva così come t stessa. Gli elementi di Y sono concorrentemente attivi quando $|Y| \geq 1$.

Quando uno step Y è attivo in un marking M_1, esso può verificarsi (occorrenza) modificando il marking M_1 in un nuovo marking M_2 così definito:

$$\forall p \in P: M_2(p) = (M_1(p) - \sum_{(t,b)\in Y} E(p,t) < b >) + \sum_{(t,b)\in Y} E(t,p) < b >.$$

M_2 è raggiungibile direttamente da M_1 . E' possibile denotarlo anche così : $M_1[Y \succ M_2$.

La valutazione dell'espressione E(p,t)<b> restituisce i token che verranno rimossi da p quando t si verifica con il binding b. Effettuando la somma su tutti gli elementi del binding (t,b)ϵY si ottengono tutti i token che verranno rimossi da p quando Y si verifica. Questo multi-set dovrà quindi essere minore o uguale al marking di p. Ciò significa che ciascun elemento del binding (t,b)ϵY deve essere in grado di ottenere i token specificati da E(p,t)<b>, senza che essi siano condivisi con altri elementi del binding di Y. Va ricordato che tutti i binding in certo step, soddisfano automaticamente la guardia corrispondente. Quando un binding appare più di una volta in Y, si ottiene un contributo per ciascuna "apparizione".

Il verificarsi di uno step è un evento indivisibile, atomico. Analizzando la formula sopra, non si riconosce quindi come marking valido il marking intermedio in cui i token sono stati rimossi ma devono ancora essere aggiunti.

Quando un certo numero di binding sono concorrentemente attivi, è possibile che si verifichi uno step che contiene solo alcuni di essi.

Una sequenza finita di occorrenze è una sequenza di marking e step:

$$M_1[Y_1 \succ M_2[Y_2 \succ M_3 \dots M_n[Y_n \succ M_{n+1}$$

tale che n $\in \mathbb{N}$ e $M_i[Y_i \succ M_{i+1}$ $\forall i \in \{1,2,\dots,n\}$ M_1 è il marking iniziale, M_{n+1} è il marking finale e n è la lunghezza.

Analogamente una sequenza infinita di occorrenze è una sequenza di marking e step:

$$M_1[Y_1 \succ M_2[Y_2 \succ M_3 \dots$$

tale che $M_i[Y_i \succ M_{i+1}$ $\forall i \geq 1$.

Un marking M'' è raggiungibile da un marking M' iff esiste un sequenza finita di occorrenze che inizia in M' e termina in M''. Il set dei marking reaggiungibili da M' sono denotati con $[M' \succ$. Un marking è raggiungibile iff appartiene a $[M'_0 \succ$.

Sono permesse sequenze di occorrenze di lunghezza zero. Ciò significa che $M \in [M \succ$ per tutti i marking M.

Con la notazione $M[t,b \succ$ e $M[t \succ$ si indica che il binding (t,b) e la transizione t sono attive nel marking M.

3 Proprietà verificabili sulle rete di Petri colorate.

Le principali proprietà di una CPN sono la *boundedness,* la proprietà di *home state*,quella di *liveness* ed infine quella di *fairness*.

La proprietà di boundedness individua il numero di token che si possono trovare in uno specifico place.

Sia p un place tale che p ϵ P, n un numero intero non negativo tale che n ϵ N ed m un multi-set tale che m ϵ $C(P)_{MS}$ allora:

I. n è un bound intero per p iff:
$\forall M \in [M_0\rangle : |M(p)| \leq n$.

II. M è un multi-set bound per p iff:
$\forall M \in [M_0\rangle : M(p) \leq m$.

La proprietà di home state individua un marking a cui è sempre possibile ritornare.

Sia M un marking tale che M ϵ $\mathbb{M}$ e X sia un set di marking tali che X ⊆ $\mathbb{M}$ allora:

I. M è un home marking iff :
$\forall M' \in [M_0 \succ : M \in [M' \succ$.

II. X è un home space iff:
$\forall M' \in [M_0 \succ X \cap [M' \succ \neq \emptyset$.

E' facile notare che M è un home marking iff {M} è un home space. Si noti che l'esistenza di un home marking indica solamente che è possibile raggiungere l'home marking, ma non è detto che la rete dovrà raggiungerlo per forza. In altre parole possono esistere infiniti percorsi nello spazio degli stati che non contengono l'home marking.

La proprietà di liveness indica che certo set di binding element X rimane attivo. Questo significa che è possibile, per ogni marking raggiungibile M', trovare una sequenza di stati partendo da M ' che contengono un elemento di X.

Sia M un marking tale che M ϵ M e X sia un set di binding element tali che X ⊆ BE allora:

I. M è "dead" iff nessun binding element è attivo, cioè iff:
$\forall x \in BE: \neg M[x \succ$.

II. X è "dead" in M iff nessun elemento di X può diventare attivo, cioè iff :
$\forall M' \in [M \succ, \forall x \in X : \neg M'[x \succ$.

III. X è "live" iff non esiste un marking raggiungibile in cui X è "dead", cioè iff:
$\forall M' \in \lceil M_0 \succ \exists M'' \in \lceil M' \succ \exists x \in X: M''\lceil x \succ$.

La proprietà di liveness richiede che solamente che un elemento di X possa diventare attivo. In questo modo possono esistere sequenze infinite di stati con marking iniziale M' e che non contengono elementi di X. Si dovrebbe notare che "live" non è la negazione di "dead" in quanto ogni set di binding elements "live" non è "dead", ma non è vero il contrario. Per ogni transizione t ϵ T, è possibile utilizzare la notazione $BE(t) \subseteq BE$ per denotare il set di tutti i binding elements che contengono t. Si dice anche che t è strettamente "live" iff {x} è "live" $\forall\, x \in BE(t)$.

La proprietà di fairness indica quanto spesso diversi binding elements vengono scelti nelle varie transizioni. Dato un set di binding element $X \subseteq BE$ ed un sequenza infinita di stati σ nella forma:

$$\sigma = M_1[Y_1 \succ M_2[Y_2 \succ M_3 \ldots.$$

è possibile indicare con $EN_{X,i}(\sigma)$ il numero di elementi di X che sono attivi nel marking M_i . Analogamente con $OC_{X,i}(\sigma)$ è possibile denotare il numero di elementi di X scelti nello step Y_1. Con $EN_X(\sigma)$ e $OC_X(\sigma)$ si indica invece il numero complessivo rispettivamente attivazioni e di occorrenze degli elementi di X cioè:

$$EN_X(\sigma) = \sum_{i=1}^{\infty} EN_{X,i}(\sigma) \qquad\qquad OC_X(\sigma) = \sum_{i=1}^{\infty} OC_{X,i}(\sigma).$$

Fintanto che tutti gli elementi di entrambe le sommatorie sono interi non negativi, è facile verificare che le somme devono essere convergenti o ad ∞ oppure ad un elemento di N.

Sia $X \subseteq BE$ un set di binding elements e sia σ un sequenza infinita di stati.

I. X è imparziale per σ iff si presenta infinite volte in σ, cioè iff:
$OC_X(\sigma) = \infty$.

II. X è "fair" per σ iff una sequenza infinita di attivazioni implica una sequenza infinita di occorrenze, cioè iff:
$EN_X(\sigma) = \infty \Rightarrow OC_X(\sigma) = \infty$.

III. X è "just" per σ iff un'attivazione persistente implica una occorrenza, cioè iff:
$$\forall i \geq 1: [EN_{X,i}(\sigma) \neq 0 \implies \exists k \geq i: \left[EN_{X,k}(\sigma) = 0 \vee OC_{X,k}(\sigma) \neq 0\right]]$$

Si dice che una transizione t ϵ T è imparziale,"fair","just" iff BE(t) possiede la proprietà corrispondente. Si dice anche che t è strettamente imparziale (strettamente "fair"/strettamente "just") iff {x} è imparziale ("fair"/ "just") per tutti gli x ϵ BE(t).

4 Un esempio di CPN : il database distribuito.

Un database "distribuito" in modo tale da garantisce la consistenza dei dati nei diversi nodi a seguito di modifiche dei dati stessi. Questo esempio non è particolarmente complesso ma lo è a sufficienza per mostrare gli elementi e le proprietà descritte in precendenza ed evidenziarne di nuove.

Il database ha n nodi (n>3) posizionati in luoghi geografici differenti. Ciascun nodo contiene una copia dell'intero database e questa copia viene gestita da un manager locale. Quando un manager d_i effettua un aggiornamento alla sua copia del database, deve mandare un messaggio a tutti gli altri manager in modo da garantire la consistenza dei dati tra le n copie del database. In questo esempio non è stato posto l'accento sul contenuto del messaggio ma, più che altro , sulle informazioni contenute nell'header come il nodo sorgente e quello di destinazione. Ciascun messaggio è perciò rappresentato come un coppia (s,r) dove s identifica il nodo sorgente ed r il nodo ricevente.

In Figura 3 : CPN del database distribuitoFigura 3 viene mostrata la rete che modella il database.

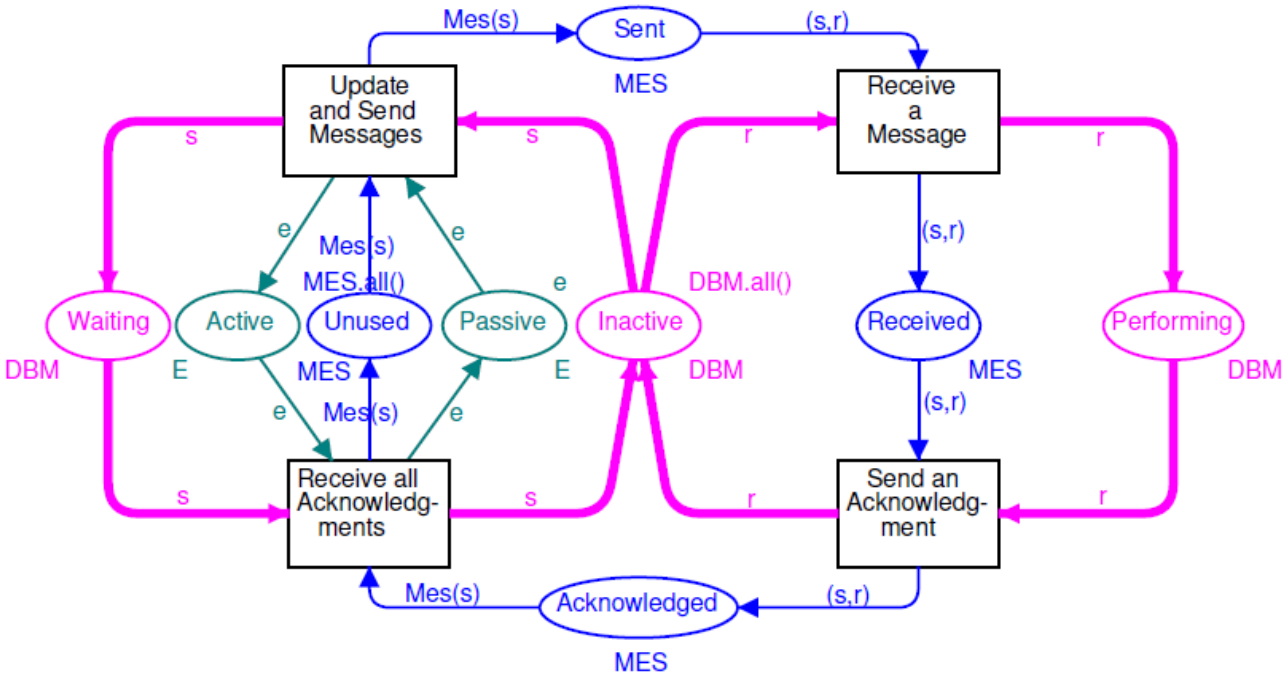

Figura 3 : CPN del database distribuito

La rete è composta da nove diversi place, tre dei quali rappresentano i possibili stati in cui un manager si può trovare, cioè *Inactive*, *Waiting* e *Performing*. Altri quattro rappresentano i possibili stati dei messaggi: *Unused*, *Sent*, *Received* e *Acknowledged.* I due place rimanenti, *Active* e *Passive*, indicano se sta avvenendo o meno un aggiornamento.

Come già ampiamente spiegato, ad ogni place è associato un "colore", o meglio un tipo di dato che il place può contenere. Nell'esempio i colori utilizzati sono :

- l' insieme costituito da tutti i manager:
$$DBM = \{d_1, d_2, \ldots . d_n\}.$$
- l'insieme dei possibili messaggi da inviare agli altri manager:
$$MES=\{(s,r)|s,r \in DBM \wedge s \neq r\}.$$
- l'insieme E costituito dall'unico elemento e:
$$E=\{ e \}.$$

Osservando la Figura 3, si può notare l'indicazione del colore del place a lato del place stesso.

Il marking iniziale della rete (M_0) prevede che tutti i manager si trovano nello stato *Inactive* mentre i messaggi in *Unused.* Ciò è esprimibile in maniera non ambigua con le seguenti espressioni:

- $M_0(\text{Inactive}) = DBM = 1'd_1 + 1'd_2 + \ldots . + 1'd_n$.
- $M_0(\text{Performing}) = M_0(\text{Waiting}) = \emptyset$, dove con $\emptyset$ si intende il multi-set vuoto.
- $M_0(\text{Unused}) = MES = \{(s,r) \in DBM \times DBM | \, s \neq r\}$.
- $M_0(\text{Sent}) = M_0(\text{Received}) = M_0(\text{Acknowledged}) = \emptyset$.

Inoltre un token è presente nel place *Passive*:

- $M_0(\text{Passive}) = E = 1'e$.
- $M_0(\text{Active}) = \emptyset$.

Al fine di comprendere il comportamento della rete è bene sapere che con Mes(s) si denota una funzione che restituisce tutti i messaggi, apparteneti all'insieme MES, aventi come nodo sorgente il nodo s:

$$\sum_{r \in DBM - \{s\}} 1'(s,r)$$

Inoltre con Mes.all() si denota una funzione che genera il multi-set contenente esattamente un token per ogni valore dell'insieme MES. Analogamente DBM.all() genera il multi-set contenente esattamente un token per ogni valore dell'insieme DBM.

Dato il marking iniziale, soltanto la transizione *Update and Send Messages* (SM)può scattare, quindi, quando un manager s intende effettuare una modifica al database, la transizione cambia lo stato di s da *Inactive* a *Waiting* , mentre lo stato dei messaggi Mes(s) cambia da *Unused* a *Sent.* Ora il manager s attende finchè tutti gli altri non hanno inviato una conferma dell'avvenuta modifica nelle rispettive copie. Quando uno degli altri manager r, esegue l'azione *Receives a Message* (RM), il suo stato cambia da *Inactive* a *Performing* (stato in cui sta aggiornando il proprio database), mentre lo

stato del messaggio corrispondente (s,r) cambia da *Sent* a *Received*. Successivamente il manager r può inviare una conferma dell'aggiornamento. Questa azione è rappresentata dalla transizione *Send Acknoledgment* (SA), la quale causa il cambiamento di stato del manager da *Performing* a *Inactive*, mentre il messaggio (s,r) passa allo stato *Acknowledged.* Infine, quando tutti i messaggi Mes(s) si trovano nello stato *Acknowledged*, la transizione *Receive all Acknowledgements* (RA) può scattare riportando il sistema allo stato iniziale M_0.

Allo scopo di garantire la consistenza delle diverse copie del database, questo semplice schema per la sincronizzazione ammette solo un aggiornamento alla volta. In altre parole, quando una manager ha effettuato un aggiornamento, tutti gli altri manager devono anch'essi effettuare la modifica prima che un nuovo aggiornamento possa essere fatto. La mutua esclusione è garantita dal place *Passive* , il quale contiene un solo token e.

Anche se il sistema modellato è assai distante da un sistema realistico, mette in luce alcuni aspetti tipici delle reti di Petri colorate come i conflitti e la concorrenza dei binding nelle transizioni. Nel marking iniziale la transizione *Update and Send Messages* è abilitata per gli n binding, contenenti gli n manager, ma solo uno verrà scelto in modo non deterministico. Una volta che la transizione si verifica con un certo mansager s, i rimanenti binding non sono più abilitati perchè non ci sono più token nel place *Passive.* Questa situazione viene chiamata conflitto e si dice che la transizione *Update and Send Messages* è in conflitto con se stessa.

Si assuma che n=5 e che si sia verificata la transizione SM con il binding $<s = d_2>$. Il sistema raggiunge quindi il marking seguente:

M_1(Inactive) = DBM -1'd_2 = 1'd_1 + 1'd_3 + 1'd_4 + 1'd_5
M_1(Waiting) = 1'd_2
M_1(Sent) = Mes(d_2) =1'(d_2, d_1) + 1'(d_2, d_3) + 1'(d_2, d_4) + 1'(d_2, d_5)
M_1(Unused) = MES – Mes(d_2)
M_1(Active) = 1'e

Nel marking M_1 la transizione RM è abilitata per tutti i quattro binding in cui s vale d_2 mentre r è pari ad un qualsiasi valore diverso da d_2. Intuitivamente questo significa che tutti gli altri manager sono pronti a ricevere il messaggio che d_2 gli ha mandato. Il binding element (RM, $<s = d_2$, r= $d_3>$) sposta un token con valore (d_2,d_3) dal place *Sent* a *Received* e sposta un token con valore d_3 da *Inactive* a *Performing.* Analogamente il binding element (RM, $<s = d_2$, r= $d_4>$) sposta un token con valore (d_2,d_4) dal place *Sent* a *Received* e sposta un token con valore d_4 da *Inactive* a *Performing.* I due binding element hanno a che fare con token diversi che non influenzano gli altri. Questo significa che possono avvenire concorrentemente con tutti gli altri nello stesso passo computazionale (step). E' facile verificare come tutti i quattro binding element abilitati possono verificarsi concorrentemente agli altri. Nel marking successivo il

place *Received* potrà contenerli tutti oppure soltanto un sottoinsieme. La scelta è non deterministica.

Le relazioni di conflitto e di concorrenza tra due binding element sono legate al marking. Aggiungendo dei token, una situazione di conflitto si può tramutare in una di concorrenza. Ad esempio si potrebbe cambiare il marking iniziale del place *Passive* in 3'e, così potrebbero verificarsi concorrentemente fino a tre binding element nella transizione SM. In generale ciò significa che un passo computazionale è un formato da un multi-set di binding element. Si supponga che da M_1, in un passo, avvengano le seguenti azioni con i relativi binding:

$$1'(RM, <s = d_2, r= d_4>) + 1'(RM, <s = d_2, r= d_4>)$$

La rete raggiunge il marking M_2 :

$M_2(Inactive) = M_1(Inactive) – (1'd_3 + 1'd_4) = 1'd_1 + 1'd_5$
$M_2(Waiting) = 1' d_2$
$M_2(Performing) = 1'd_3 + 1'd_4$
$M_2(Sent) = 1'(d_2,d_1) + 1'(d_2,d_5)$
$M_2(Received) =1'(d_2,d_3) + 1'(d_2,d_4)$
$M_2(Unused) = MES – Mes(d_2)$
$M_2(Active) = 1' e$

In M_2 sono presenti quattro binding element abilitati:

$(RM, <s = d_2, r= d_1>)$
$(RM, <s = d_2, r= d_5>)$
$(SA, <s = d_2, r= d_3>)$
$(SA, <s = d_2, r= d_4>)$

i quali possono verificarsi concorrentemente agli altri perchè utilizzano token diversi. Si assuma che si verifichino solo i primi tre, cioè che lo step sia costituito dai seguenti binding element:

$$1'(RM, <s = d_2, r= d_1>) + 1'(RM, <s = d_2, r= d_5>) + 1'(SA, <s = d_2, r= d_3>)$$

Allora lo stato della rete è dato dal marking M_3:

$M_3(Inactive) =1' d_3$
$M_3(Waiting) =1' d_2$
$M_3(Performing) =1' d_1 + 1' d_4 + 1' d_5$
$M_3(Sent) = M_2(Sent) – (1'(d_2,d_1) + 1'(d_2,d_5)) = \emptyset$
$M_3(Received) = 1'(d_2,d_1) + 1'(d_2,d_4) + 1'(d_2,d_5)$
$M_3(Acknowledged) = 1'(d_2,d_3)$
$M_3(Unused) = MES – Mes(d_2)$
$M_3(Active) = 1'e$

In M_3 ci sono tre binding elements attivi:

(SA, <s = d_2, r= d_1>)
(SA, <s = d_2, r= d_4>)
(SA, <s = d_2, r= d_5>)

L'esecuzione concorrente dei tre binding element porta al marking M_4 :

M_4(Inactive) =1' d_1 + 1' d_3 + 1' d_4 + 1' d_5
M_4(Waiting) =1' d_2
M_4(Performing) = M_4(Sent) = M_4(Received) = ∅
M_4(Acknowledged) = Mes(d_2)
M_4(Unused) = MES – Mes(d_2)
M_4(Active) = 1'e

Nel marking M_4 soltanto il binding element (RA,<s= d_2>) è attivo. La sua esecuzione riporta la rete nel marking iniziale M_0. L'insieme dei marking e degli step visti formano una successione di eventi/occorrenze:

$$M_0[Y_0 \succ M_1[Y_1 \succ M_2[Y_2 \succ M_3[Y_3 \succ M_4[Y_4 \succ M_0.$$

La sequenza di eventi considerata è soltanto una delle infinite possibili. Questa in particolare sembra essere ciclica visto che inizia e finisce nello stesso marking M_0.

Di seguito verranno analizzate, una ad una, le proprietà mostrate nel paragrafo 3 sull'esempio appena mostrato.

Per quanto riguarda la proprietà di boundedness nel database distribuito (con n manager) risultano i bound nella Tabella 1.

	Multi-set	Intero
Inactive	DBM	n
Waiting	DBM	1
Performing	DBM	n – 1
Unused	MES	n^2 – n
Sent, Received, Acknowledged	MES	n – 1
Passive, Active	E	1

Tabella 1 : Bound nei vari place dell'esempio

I bound sono tutti ottimi, cioè sono i più piccoli possibili. Si noti come i bound dei multi-set e degli interi si completano tra loro. Da ciasuno di essi è spesso possibile dedurre informazioni che non possono essere dedotte dagli altri e viceversa. Questo è il caso del place *Waiting* e *Inactive*. Per il place *Waiting* il bound intero è più preciso

rispetto a quello al bound del multi-set, mentre per *Inactive* vale esattamente l'opposto.

La proprietà di home state, invece, è verificata in maniera ovvia per il marking iniziale a cui, prima o poi, si torna sempre, mentre la proprietà di liveness è valida per tutte e quattro le transizioni in maniera "strictly". Quest'affermazione può essere verificata dimostrando che il marking iniziale M_0 è un home marking e che esiste una sequenza di step che inizia in M_0 e contiene tutti i binding element BE.

Per quanto riguarda la proprietà di fairness, si ha che le quattro transizioni sono tutte imparziali. SM è strettamente "just" mentre le rimanenti tre sono strettamente "fair".

5 Il software CPN Tools.

Attualmente i software diffusi in rete per la creazione di reti di Petri colorate sono pochissimi. Il più conosciuto in letteratura, ed anche il più completo, è il software CPNTools. Sviluppato dalla Aarhus University, il tool permette di creare, simulare ed analizzare le reti di Petri colorate per via grafica, verificandone inoltre proprietà e performance.

5.1 Installazione.

Il software è reperibile all'indizzo web (4) dove è possibile scaricarne di differenti versioni in base al sistema operativo in uso: Windows, Linux o Mac. La versione più aggiornata è quella per Windows giunta fino alla 3.2.2 mentre la distribuzione per Linux/Mac è ferma alla 2.3.5. Si consiglia pertanto l'uso della prima. Una volta scaricato l'eseguibile, avviarlo. L'installazione è semplice e veloce, non dovrebbe quindi creare problemi.

5.2 Interfaccia e principali comandi.

L'interfaccia grafica di CPNTools (Figura 4) risulta essere molto pulita, in quanto è possibile estrarre man mano dal Menù *Tool box* le palette che contengono i pulsanti per azioni specifiche. Esistono infatti palette per la creazione di place, archi e transizioni, palette per il caricamento o salvataggio della rete, palette per la simulazione della rete e per l'analisi dello spazio degli stati, e queste sono soltanto le principali. Altre azioni come la dichiarazione dei colori o delle variabili non sono eseguibili tramite pulsanti ma nella sezione *declarations* e secondo una precisa sintassi definita nella documentazione. La semplice associazione dei colori ai marking, se non si è consultato la documentazione(5) (comunque non particolarmente chiara), risulta poco intuitiva (tasto TAB).

Nel complesso, una volta familiarizzato con esso, il programma risulta facile da utilizzare per quanto riguarda la creazione e la simulazione delle reti, ma non si può dire lo stesso per la verifica formale di proprietà (o model checking) o l'uso delle reti gerarchiche.

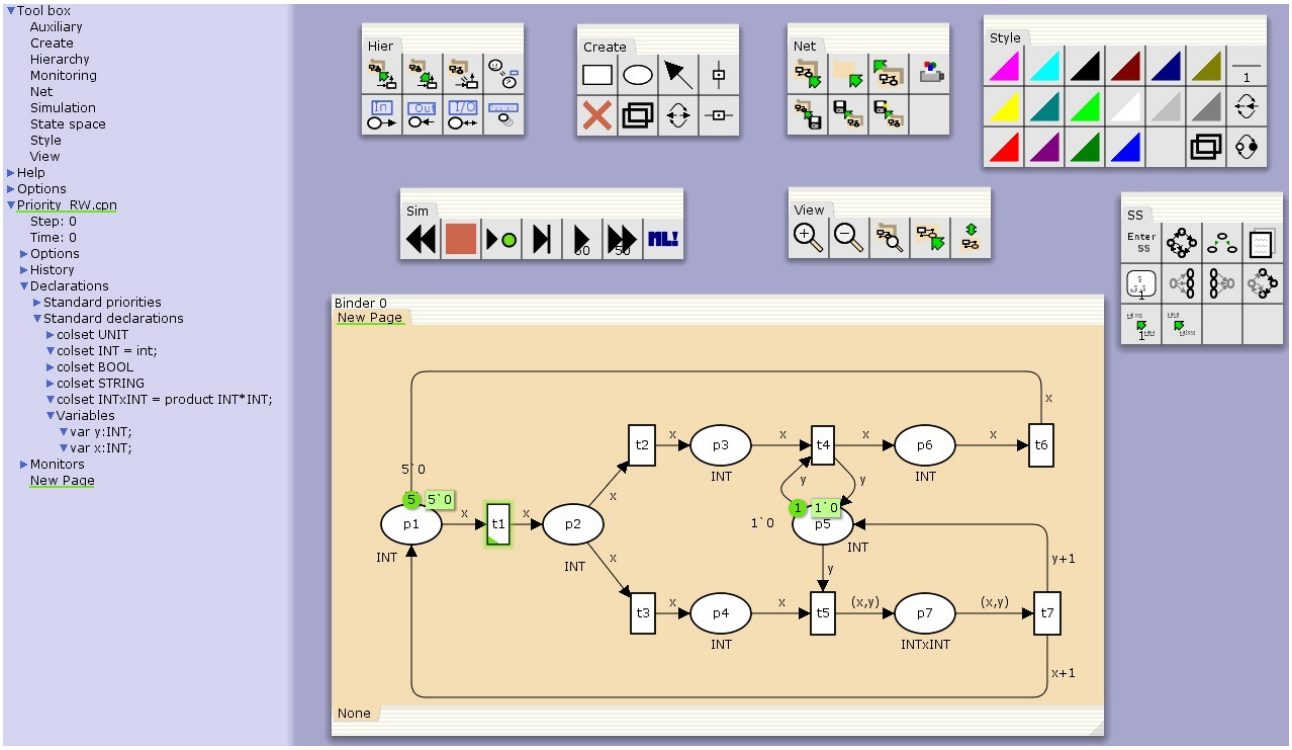

Figura 4:Interfaccia grafica e comandi di CPNTools

6 Spin e Promela.

Spin è un software open-source utilizzato da migliaia di persone nel mondo per l'analisi e la verifica formale di sistemi software distribuiti. Il tool fu sviluppato dal 1980 dal gruppo Unix del Computing Sciences Research Center ai Bell Labs. Il software viene rilasciato gratuitamente dal 1991, e continua ad evolvere per stare al passo con i nuovi sviluppi nel campo. Nell'aprile 2002 il tool è stato premiato con il prestigioso System Software Award per il 2001 da ACM. (6)

Promela(PROcess MEta LAnguage), invece, è il linguaggio di alto livello con cui i sistemi software, che si vogliono verificare con Spin, devono essere scritti. Promela è un linguaggio non deterministico, basato sul concetto di guardia introdotto da Dijkstra ma che non presenta la notazione di operazioni di I/O tipiche del linguaggio CSP di Hoare.(7)

6.1 Spin: descrizione generale (6).

Ecco alcune funzionalità che hanno reso Spin così popolare nella verifica formale dei sistemi software:

- L'obiettivo di Spin è quello di realizzare la verifica formale dei software in maniera efficiente. Spin è stato utilizzato per individuare errori nella progettazione di sistemi distribuiti, come i sistemi operativi, protocolli per lo scambio di dati, algoritmi concorrenti, protocolli per le segnalazioni in ambiente ferroviario ecc.. Il tool controlla la correttezza logica di una specifica, segnalando eventuali deadlock, ricezioni non specificate, corse critiche, assunzioni ingiustificate sulla velocità relativa dei processi.
- Spin fornisce il supporto diretto per l'utilizzo di codice C embedded come parte del modello di specifica. Ciò rende possibile la verifica del sistema direttamente a livello di implementazione della specifiche software, utilizzando Spin come motore logico e verificandone quindi le proprietà temporali di alto livello.
- Spin fornisce il supporto diretto per il suo utilizzo su computer multi-core supportando sia verifiche di proprietà di safety che di liveness.
- Spin funziona in modalità *on-the-fly*, il chè significa che evita di precostruire un grafo di stato globale, o un struttura di Kripke, come prerequisito alla verifica delle proprietà del sistema.
- Spin è un sistema completo per la verifica di proprietà LTL, supporta infatti tutte le espressioni esprimibili nella logica temporal lineare, ma può anche essere utilizzato per la verifica on-the-fly di proprità di safety e di liveness più semplici. La proprietà da verificare possono essere specificate come un sistema o un processo di invarianti (utlizzando le asserzioni), come una formula LTL, come un automa di Büchi, o più comunemente come una proprietà generale nella sintassi delle formule "never claims".
- Il tool supporta la variazione dinamica del numero di processi, utilizzando una tecnica "rubber state vector".

- Il tool supporta sia rendezvous, scambio di messaggi da buffer e comunicazione tramite memoria condivisa. Anche sistemi misti, cioè che utilizzano sia la comunicazione sincrona che quella asincrona, sono supportati. Gli identificatori dei canali, sia per il rendezvous che per i canali bufferizzati, possono essere scambiati tra i processi tramite messaggi.
- Il tool supporta diversi tipi di simulazioni: random, interattiva e guidata. Ognuna di esse può essere fatta in maniera esaustiva o utilizzando tecniche per l'analisi parziale basate sia sulla ricerca depth-first che breadth-first. Il tool è specificamente progettato per scalare efficientemente fino a problemi di dimensioni molto ampie.
- Per ottimizzare l'esecuzione delle verifiche, il tool utilizza la tecnica "partial order reduction" e opzionalmente tecniche per la memorizzazione simili a BDD.

6.2 Installazione

L'installazione di Spin può avvenire in due modi:

1. Su Eclipse installando il plugin dall' indirizzo *http://matrix.uni-mb.si/fileadmin/datoteke/znanost/orodja/ep4s/update-site/*
2. Su un sistema Linux-based o Windows (sconsigliato) assieme ai software tcl/tk e iSpin seguendo la guida sul sito ufficiale (6).

6.3 Interfaccia grafica e principali features

Riferendoci a iSpin, l'interfaccia grafica è suddivisa in schede. Le più importanti sono : *Edit/View*, *Simulation* e *Verification*. Nella prima, osservabile in Figura 5, è possibile caricare, salvare e redigere un programma in Promela; è presente anche un pulsante per la verifica sintattica e una particolare schermata (a destra) che mostra il sistema come un automa a stati finiti (se Graphviz è installato). Nella seconda (Figura 6) si trovano tutti i comandi e le impostazioni per poter eseguire un programma Promela, sono infatti presenti i pulsanti per l'avvio, per il riavvolgimento dell'esecuzione, per l'avanzamento *step by step*. Le principali impostazioni riguardano il numero massimo di passi computazionali che il simulatore deve effettuare prima di arrestarsi e il seed utilizzato per l'avanzamento in punti di non determinismo. Una volta terminata l'esecuzione, è possibili selezionare un qualsiasi passo computazionale dalla schermata nera centrale e verificare lo stato di una qualsiasi variabile del sistema nella schermata nera di sinistra. L'ultima, ma non la meno importante, è la scheda *Verification* (Figura 7) in cui è possibile configurare il model checker per la verifica di una qualsiasi proprietà sul sistema descritto con il codice Promela. La verifica può essere fatta soltanto per una proprietà alla volta specificandone l'identificativo nella sezione *Claims*. La proprietà sarà verificata se, nel riquadro nero di destra, il model checker non riporta errori o *unreachable code* durante l'esecuzione. Nel caso vi siano errori viene anche generato un file di estensione *.trail*, che è possibile rieseguire dalla scheda *Simulate*, e mostra la sequenza di passi che ha provocato l'errore.

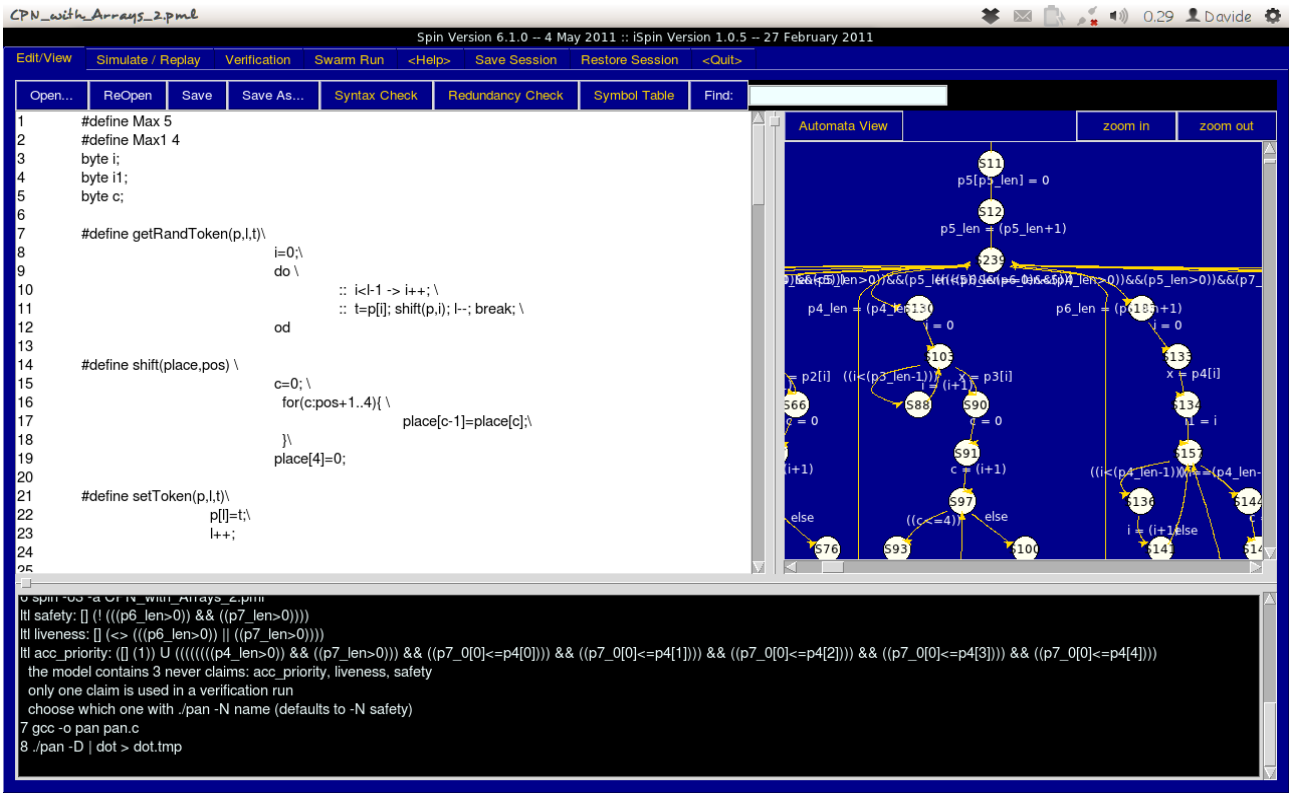

Figura 5: iSpin e la scheda *Edit/View*.

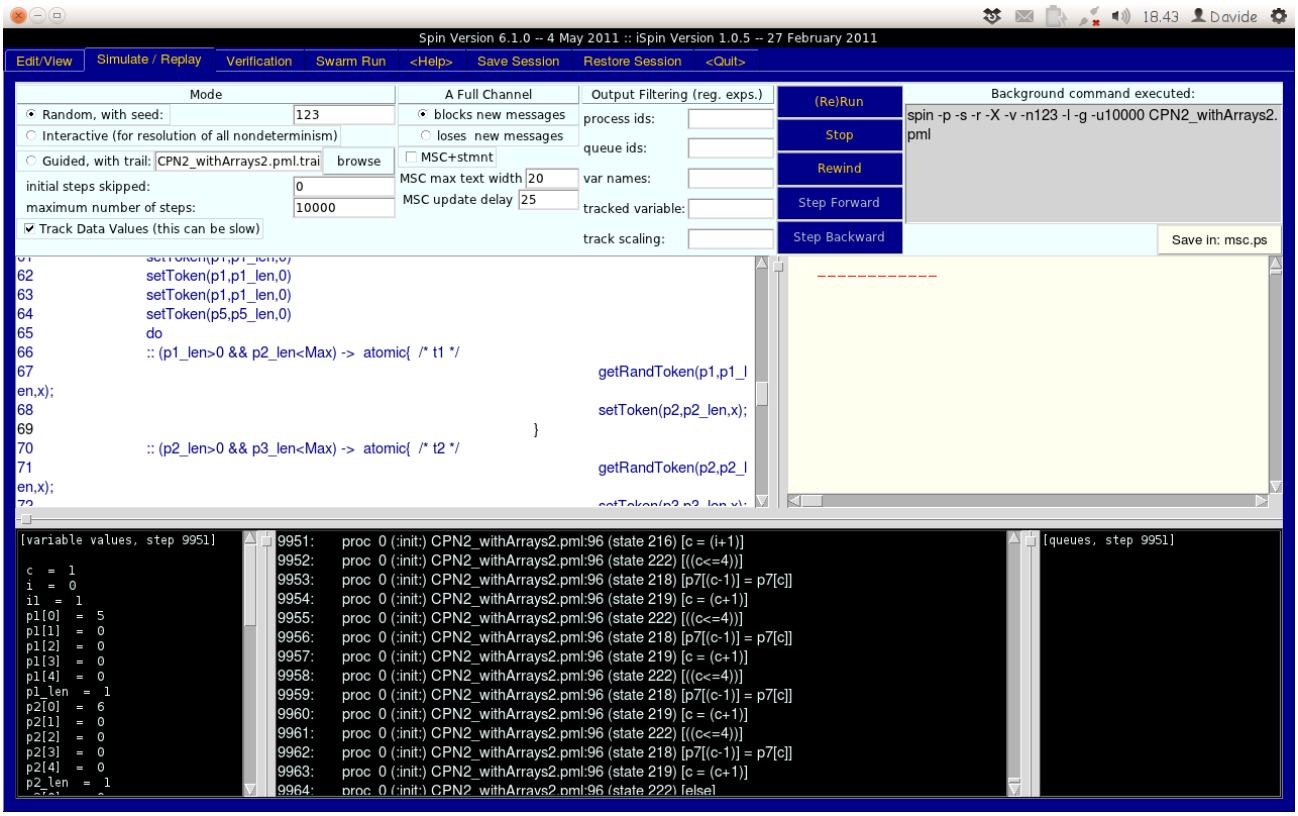

Figura 6: iSpin e la schermata per l'esecuzione del codice.

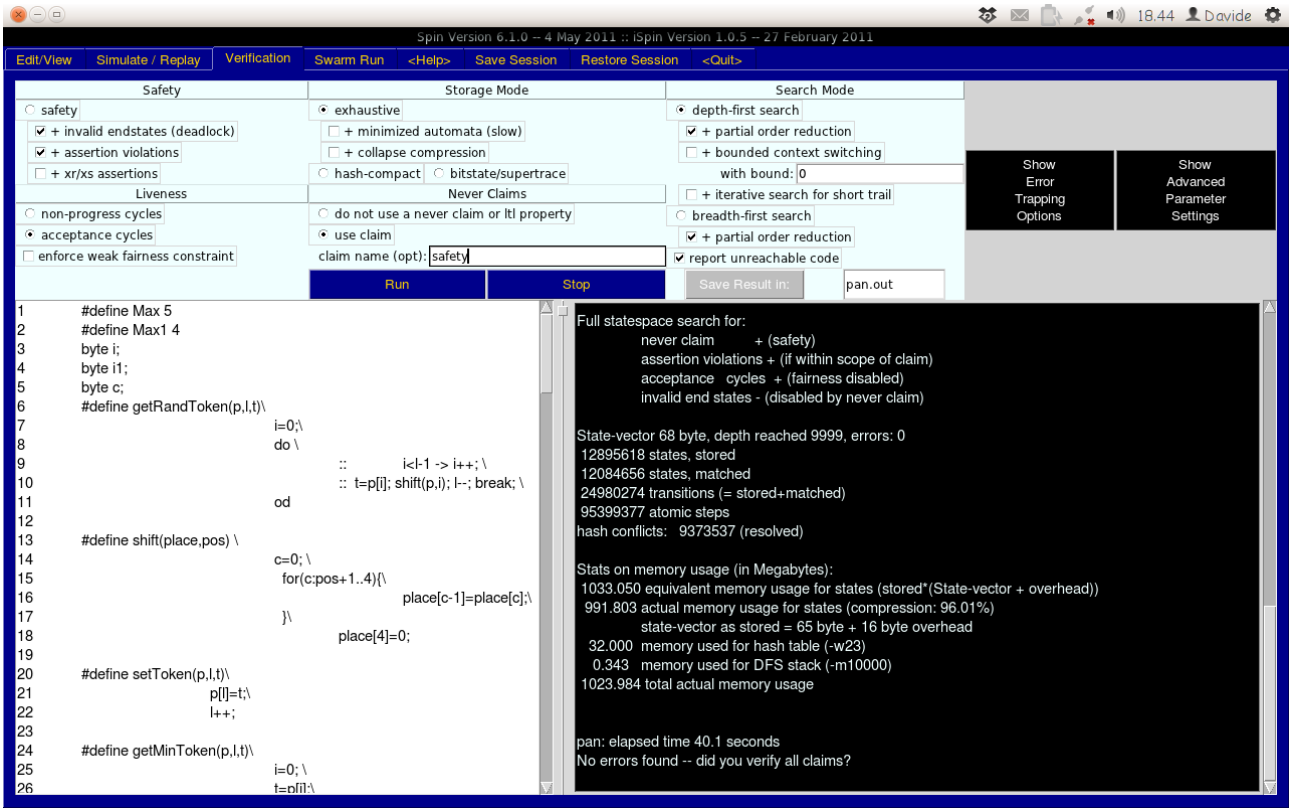

Figura 7: iSpin e la schermata *Verification*

7 Da Prolog a Promela.

Al fine di sfruttare le prestazioni di Spin sia come simulatore che come model checker, si è scelto di realizzare un modello delle reti di Petri colorate in Prolog (8) ed un traduttore model-to-code che produce, dal modello, il codice nel linguaggio Promela, eseguibile da Spin. Fondamentale è l'individuazione di una rappresentazione delle CPN in Promela che rende efficiente, lineare e semplice il codice ma allo stesso tempo permette una traduzione dal modello Prolog non eccessivamente complessa. Visto che le reti di Petri classiche (9), nonostante la capacità espressiva sia la stessa, sono un formalismo più semplice da rappresentare, ma che conserva numerosi elementi in comune con le CPN, sono state scelte come punto di partenza per il progetto con l'idea di estenderle in seguito per aggiungere le caratteristiche tipiche delle CPN.

7.1 Realizzazione del traduttore per le reti di Petri classiche

L'esempio, che è stato adottato come riferimento nelle considerazioni di questo paragrafo, è il classico problema dei Readers-Writer la cui PTN è osservabile in Figura 8. In tale esempio emergono infatti tutte le caratteristiche delle reti Place-Transition (Turing complete) tra cui l'abilitazione multipla delle transizioni e l'uso degli archi inibitori.

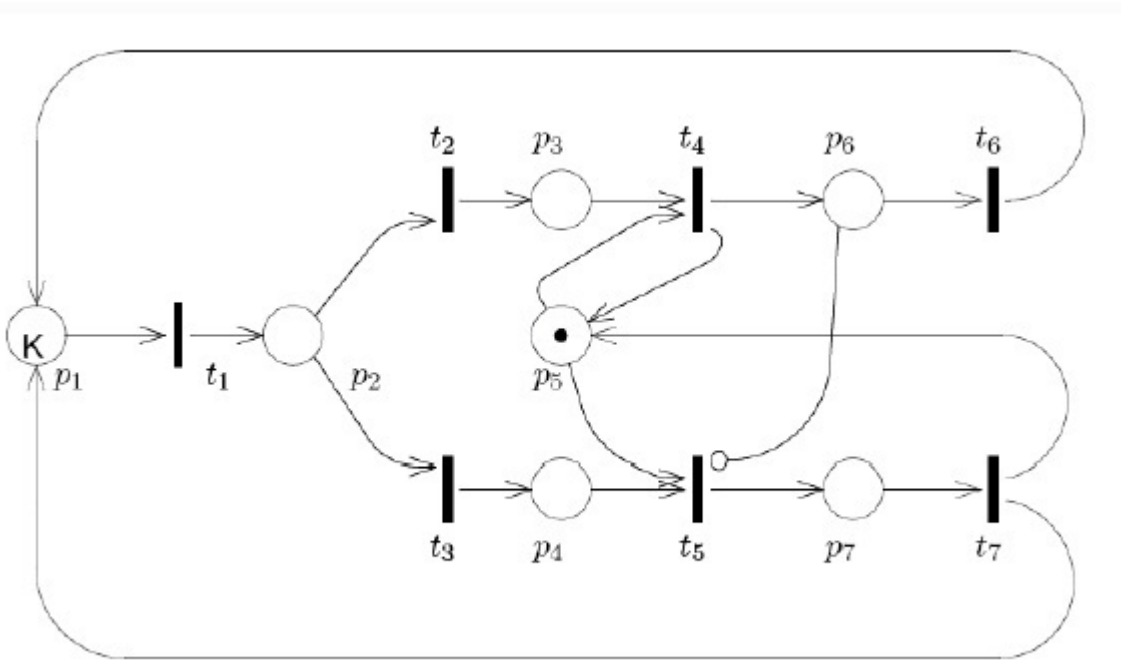

Figura 8 : Rappresentazione del problema dei Readers - Writer con una PTN (10)

7.1.1 Rappresentazione Prolog

La scelta fatta per rappresentare gli elementi tipici di una PTN, cioè place e transizioni, è stata quella di metterli in relazione uno a uno con i fatti Prolog. In particolare è stato adottato un predicato place/2 per i place dove gli argomenti indicano rispettivamente:

- l'identificativo del place;
- il numero di token nel marking iniziale;

ed un predicato transition/3 per le transizioni i cui argomenti rappresentano:

- una lista di place da cui vengono consumati i token ;
- una lista di place in cui i token vengono inseriti;
- una lista di place che inibiscono la transizione;

In riferimento all'esempio di Figura 8, il codice Prolog risulta essere il seguente:

```
/*Readers-writer Petri net Prolog rappresentation*/
place('p1',5).
place('p2',0).
place('p3',0).
place('p4',0).
place('p5',1).
place('p6',0).
place('p7',0).

%transition(InputPlaces,OutputPlaces,InhibitoryArcsSourcePlaces)
transition(['p1'],['p2'],[]).
transition(['p2'],['p3'],[]).
transition(['p2'],['p4'],[]).
transition(['p3','p5'],['p5','p6'],[]).
transition(['p4','p5','p2'],['p7'],['p6']).
transition(['p6'],['p1'],[]).
transition(['p7'],['p5','p1'],[]).
```

Figura 9: Rappresentazione in Prolog della PTN per il problema dei Readers - Writer

Dove *k* è pari a cinque.

7.1.2 Il codice Promela

La ricerca di una corretta rappresentazione degli elementi di una PTN in Promela, sia dal punto di vista funzionale che di semplicità del codice, si è spesso spesso scontrata con alcune lacune che questo linguaggio presenta. In un primo momento l'idea è stata quella di implementare i place come singoli *byte* e le transizioni come un unico processo generale avente due parametri: un'array di place in ingresso ed un array di place in uscita (Figura 10: Una prima rappresentazione per place e transizioni in PromelaFigura 10).

```
#define Place   byte

Place p1, p2, p3, p4, p5, p6, p7

 proctype transition(Place[] input, out){
      atomic{
            int i,o;
            for(i in input){
                  input[i]--;
            }
            for(o in output){
                  output[o]++;
            }
      }
 }
```

Figura 10: Una prima rappresentazione per place e transizioni in Promela.

In questo modo il flusso di controllo principale si riduce, oltre all' inizializzazione del marking iniziale, ad un loop "do" inifinito in cui vengono eseguiti i processi, opportunamente configurati, in base alle guardie soddisfatte in un certo ciclo e al non determinismo di Promela (Figura 11).

```
init {
      p1=3; p5=1;
      do
      :: p1>0 && p2<Max -> run transition({p1},{p2})
      :: p2>0 && p3<Max -> run transition({p2},{p3})
      :: p2>0 && p4<Max -> run transition({p2},{p4})
      :: p3>0 && p5>0 && p5<Max && p6<Max -> run transition({p3,p5},{p5,p6})
      :: p6==0 && p4>0 && p5>0 && p7<Max -> run transition({p4,p5},{p7})
      :: p6>0 && p1<Max -> run transition({p6},{p1})
      :: p7>0 && p1<Max && p5<Max -> run transition({p7},{p1,p5})
      od
}
```

Figura 11: Main loop per l'approccio iniziale.

Questa idea è stata presto abbandonata per il semplice fatto che Promela non supporta il passaggio di array come parametri dei processi e si pensato di definire un processo specifico per ogni transizione che non aveva quindi bisogno di alcun parametro per essere eseguito visto che i place erano definiti globalmente. Quest'approccio, però, ha come ovvia conseguenza l'aumento notevole delle righe di codice riducendo la leggibilità del codice stesso che risulta essere molto dispersivo. Nella soluzione finale si è quindi preferito eliminare il concetto di processo inserendo anche il codice specifico delle transizioni nel loop principale. Il risultato è osservabile in Figura 12. Data la definizione dei place, la stessa di Figura 10, il codice relativo alle transizioni si riduce al solo decremento ed incremento rispettivamente dei place in ingresso ed in uscita alla transizione. Anche l'inibizione di una transizione risulta semplice da realizzare, inserendo la sola condizione *"p6==0"* all'interno della guardia della transizione *t5*.

```
init {
      p1=3; p5=1;
      do
      :: (p1>0 && p2<Max) -> atomic{
                              p1--;
                              p2++;
                        }
      :: (p2>0 && p3<Max) -> atomic{
                              p2--;
                              p3++;
                        }
      :: (p2>0 && p4<Max) -> atomic{
                              p2--;
                              p4++;
                        }
      :: (p3>0 && p5>0 && p5<Max && p6<Max) -> atomic{
                              p3--;
                              p5--;
                              p6++;
                              p5++;
                        }
      :: (p6==0 && p4>0 && p5>0 && p7<Max) -> atomic{
                        p4--;
                        p5--;
                        p7++;
                  }
      :: (p6>0 && p1<Max)-> atomic{
                  p6--;
                  p1++;
            }
      :: (p7>0 && p5<Max && p1<Max)-> atomic{
                  p7--;
                  p5++;
                  p1++;
            }
      od
}
```

Figura 12: Main loop della soluzione finale

7.1.3 Il traduttore

Il traduttore è stato realizzato con un approccio *divide et impera*; è stato scomposto il codice Promela da generare in categorie e sottocategorie e per ciascuna di esse è stato implementato un predicato Prolog che, recuperando le informazioni dai fatti che descrivono la rete di Petri, genera il codice relativo a una specifica categoria. Escludendo il codice indipendente dalla rete specifica come la definizioni delle macro, lo statement *init* e il loop principale che non variano mai, l'attenzione è stata posta sulla scomposizione del codice relativo alle variabili dei place, all'inizializzazione del marking iniziale, alle condizioni delle guardie ed infine all'effetto della transizione. Tutti i predicati, che generano queste parti di codice, vengono eseguiti uno per volta dal predicato principale, cioè il predicato *translate* che si occupa inoltre di creare il file in cui verrà generato il codice e di chiuderlo al termine del procedimento. In Figura 13 è mostrato tale predicato le cui parti, evidenziate in rosso, generano il codice specifico della rete da tradurre.

```
%translate(+FilePath)
translate(X):-  tell(X),
            init_declarations,findall(Y,place(Y, ),P),print_places(P),
            open_main_proc,
            findall(place(ID,V),place(ID,V),L),print_assignments(L),
            open_do,nl,
            findall(transition(I,O,H),transition(I,O,H),L1),
            print_transitions(L1),
            close_do,nl,
            end_main_proc,
            print_ltl_clauses,
            told.
```

Figura 13 : Predicato principale per la traduzione delle reti di Petri classiche

Nel primo riquadro rosso di Figura 13, partendo dall'alto, vi è il codice che genera le strutture dati contenenti i place. Grazie al predicato *findall* vengono recuperati e inseriti in una lista tutti gli identificativi dei place. Questa lista viene passata al predicato ricorsivo *print_places* che si occupa della produzione del codice opportunamente formattato, in particolare delle variabili di tipo *byte* che conterranno il numero di token per ciasun place (Figura 14).

```
%prints the declarations of variables rappresenting Places
print_places([H|T]):-print(H),print_rest(T).
print_rest([]).
print_rest([H|T]):-print(','),print(H),print_rest(T).
```

Figura 14 : Implementazione del predicato *print_places*

Il codice indicato nel riquadro rosso centrale (Figura 13) , invece, genera il codice per l'inizializzazione del marking iniziale. Anche qui il procedimento implica la creazione di una lista contenenti tutti i predicati *place* per mezzo della *findall*. Questa lista viene verrà poi analizzata ricorsivamente dal predicato *print_assignments* che si occupa del codice per l'inizializzazione del marking. Il codice altro non è che un semplice assegnamento del valore del numero di token alla variabile che rappresenta il relativo place (Figura 15).

```
%prints assignments that defines the initial marking.
print_assignments([]).
print_assignments([place(ID,V)|T]):-(V=\=0,print(ID),print('='),
                    print(V),print(';'),print_assignments(T),!;print_assignments(T)).
```

Figura 15 : Inizializzazione del marking nella traduzione delle PTN

Osservando il codice si può osservare che l'assegnamento viene fatto se e solo se il marking è non nullo in quanto in Promela le variabili, alla loro creazione, vengono automaticamente inizializzate con il valore zero.

Infine nell'ultimo riquadro rosso di Figura 13, partendo dall'alto, vi è il codice che genera la rappresentazione Promela delle transizioni. Come in precedenza, tutti i predicati *transition* vengono prelevati dalla teoria Prolog tramite la *findall*, la lista viene poi elaborata dal predicato *print_transitions* che in realtà realizza soltanto la ricorsione sulla lista, delegando al predicato *print_transition* (senza "s" finale) il compito di stampare il codice.

```
%prints the Promela rappresentation of transitions
print_transitions([]).
print_transitions([H|T]):-print_transition(H),print_transitions(T).

print_transition(transition(I,O,H)):-tab(5),print(':: ('),
                print_conditions(I,H),
                print(') -> atomic{'),nl,
                print_all_dec(I),
                print_all_inc(O),
                tab(5),print('}'),nl.
```

Figura 16 : Il predicato *print_transitions* per la traduzione delle PTN

Essendo il codice relativo ad una transizione più complesso, per la traduzione si è scelto di scomporre ulteriormente il codice in sottocategorie come le condizioni, i decrementi dei place in input e gli incrementi di quelli in output. Come è possibile osservare in Figura 16, per ciascuna sottocategoria è stato realizzato il rispettivo predicato (*print_conditions*, *print_all_dec*, *print_all_inc*). Tutti gli altri predicati si occupano di formattare correttamete il codice o di generare codice indipendente dalla rete.

7.1.4 Le proprietà LTL e la loro traduzione in Promela

Dedicato all'utente che, oltre alla simulazione della rete, necessita anche di fare model checking verificando proprietà ltl, è stato pensato il predicato Prolog *ltl/2* che verrà anch'esso tradotto nel codice Promela corrispondente . I due argomenti rappresentano rispettivamente un identificativo univoco e la formula LTL da verificare. Riprendendo l'esempio dei Readers-Writer di Figura 8, sono stati definiti due fatti: uno per la proprietà di safety ed uno per quella di liveness (Figura 17).

```
%ltl(id,formula). 'formula' is a valid ltl formula for Spin
%operators [],<>,U,!,&&,||,V,W,->,<-> can be used
ltl(safety,'[] !(p6>0 && p7 >0)').
ltl(liveness,'[]<> (p6>0 || p7 >0)').
```

Figura 17 : Proprietà ltl del problema Readers-Writer

La traduzione è stata realizzata con lo stesso procedimento basato sulla *findall* visto in precedenza in cui, però, la generazione del codice viene affidata al predicato *print_clauses* (Figura 18).

```
print_ltl_clauses:-findall(ltl(ID,V),ltl(ID,V),L),print_clauses(L).
print_clauses([]).
print_clauses([H|T]):-print_clause(H),print_clauses(T).
print_clause(ltl(ID,V)):-nl,print(ID),print('{'),print(V),print('}').
```

Figura 18 : Codice per la traduzione delle formule LTL

Il codice Promela risultante è osservabile in (Figura 19).

```
safety{[] !(p6>0 && p7 >0)}
liveness{[]<> (p6>0 || p7 >0)}
```

Figura 19 : Codice Promela per le formule LTL

7.2 Realizzazione del traduttore per le reti di Petri colorate

Il passaggio dalle reti di Petri classiche a quelle colorate implica la modifica della rappresentazione Prolog e del modello Promela, per il supporto alle nuove funzionalità in particolare i tipi, il binding, le condizioni sui valori dei token e l'elaborazione dei token stessi. I limiti di Promela unita alla limitata durata del progetto non hanno permesso di realizzare tutte le funzionalità presentate nel Capitolo 0. Si è deciso quindi di implementare prima una soluzione minimale a cui sono state aggiunte man mano diverse funzionalità ed estensioni. Dato che il problema dei Readers-Writer di Figura 8 : Rappresentazione del problema dei Readers - Writer con una PTNFigura 8 non metteva in luce alcune features delle CPN è stato deciso di modificarlo e di adottarlo per la verifica di correttezza del codice Promela generato. Il nuovo problema denominato *Readers-Writer with priority* estende il precedente associando ai token un valore positivo che rappresenta il numero di accessi alla risorsa effettuati per la scrittura e regolando l'accesso a tale risorsa, solo per la scrittura, in base a tale valore; in particolare se due o più token sono in attesa di scrivere sulla risorsa accederà quello che ha precedentemente effettuato meno scritture.

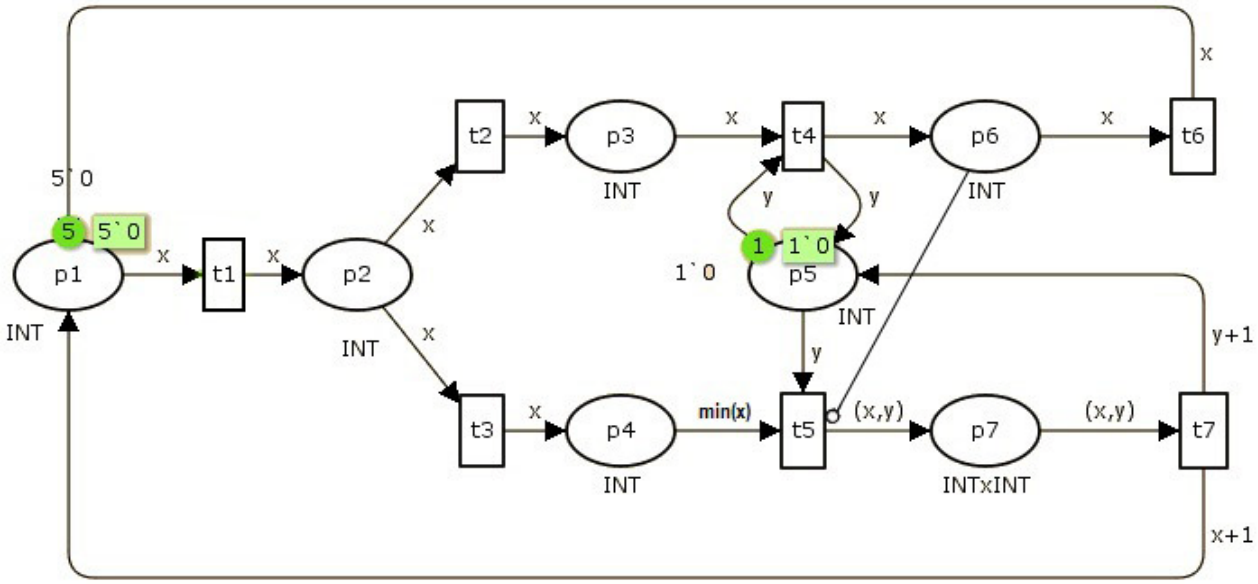

Figura 20: CPN per il problema dei *Readers-Writer with priority*

7.2.1 Rappresentazione Prolog

Come nel Paragrafo 7.1.1, anche per le CPN si è scelto di realizzare un'associazione uno a uno con i fatti prolog sia per i place che per le transizioni. I place vengono rappresentati da un predicato place/3 dove gli argomenti indicano rispettivamente:

1. l'identificativo del place;
2. una lista di tipi che rappresentano il colore del place;
3. una lista di token che rappresentano il marking iniziale;

mentre le transizioni vengono rappresentate da un predicato transition\7 dove gli argomenti indicano rispettivamente:

1. un identificativo della transizione;
2. una lista di place da cui vengono consumati i token ;
3. una lista di variabili in cui vengono memorizzati temporaneamente i valori dei token;
4. una lista di place in cui i token vengono inseriti;
5. una lista di precondizioni che i token devono soddisfare;
6. una lista di elaborazioni fatte sulle variabili;
7. una lista di place che inibiscono la transizione;

Ovviamente, per quanto riguarda i place, la novità principale sono i colori che possono essere scelti tra i tipi di base di Promela come *int*, *bool*, *byte*. In realtà i colori a disposizione sono superiori in quanto è consentito combinare fino a due tipi base. Si noti infatti come sia possibile indicare una lista di tipi come secondo parametro del predicato place. I token del marking iniziale andranno indicati di conseguenza come liste di due valori.

Il predicato transition, come si può notare dall'aumento del numero di argomenti passato da tre a sette, ha subito un notevoli cambiamenti. In primo luogo è stato

introdotto il predicato variable/2 per la definizione di variabili di supporto, i cui parametri indicano rispettivamente il tipo e il nome della variabile. Le variabili così dichiarate possono essere utilizzate all'interno delle transizioni per la memorizzazione dei valori dei token se specificate nel parametro 3. In tal caso è fondamentale l'ordine con cui vengono indicate all'interno della lista in quanto, il valore che tale variabile assumerà, sarà quello del token prelevato dal place in input corrispondente alla stessa posizione nel parametro 2. Ad esempio se si considera il fatto :

transition(t,[p1,p2],[x,y],[ ],[ ],[ ],[ ]),

supponendo che sia p1 che p2 siano due place di tipo int, allora il token prelevato da p1 verrà messo in x mentre quello di p2 in y. Lo stesso ragionamento sta alla base dell'associazione delle variabili ai place in output alla transizione se infatti si condidera il fatto

transition(t,[p1,p2],[x,y],[p3,p4],[ ],[ ],[ ]),

con p1,p2,p3,p4 place di tipo int, allora il token prelevato da p1 andrà in x ed infine verrà inserito in p3, mentre il token preso da p2 andrà in y e poi in p4.
Un'altra novità riguarda la possibilità di trasformare i valori dei token in input prima di inviarli ai place in output. Le trasformazioni vanno specificate come stringhe di codice Promela nella lista del parametro 6.
Allo scopo di porre delle limitazioni sui binding è stato adottato un ulteriore parametro, una lista di condizioni esprimibili o tramite le funzioni max/min oppure tramite gli operatori di confronto >,<,>=,<=. Se nessuna limitazione viene specificata la scelta del binding viene fatta in modo non deterministico.
Detto ciò è possibile osservare in Figura 21 la rappresentazione Prolog della CPN per il problema dei *Readers-Writer with priority*.

```
$place(ID,type/color,TokenList)
place(p1,[int],[0,0,0,0,0]).
place(p2,[int],[]).
place(p3,[int],[]).
place(p4,[int],[]).
place(p5,[int],[0]).
place(p6,[int],[]).
place(p7,[int,int],[]).

$variable(type, name).
$can be used in more different transitions. Names i,i1,c are reserved.
variable(int,x).
variable(int,y).

$transition(ID,in places,variables,out places,pre-conditions,trasformations,inhibitory arc source place)
transition(t1,[p1],[x],[p2],[],[],[]).
transition(t2,[p2],[x],[p3],[],[],[]).
transition(t3,[p2],[x],[p4],[max(x)],[],[]).
transition(t4,[p3,p5],[x,y],[p6,p5],[],[],[]).
transition(t5,[p4,p5],[x,y],[p7],[min(x)],[],[p6]).
transition(t6,[p6],[x],[p1],[],[],[] ).
transition(t7,[p7],[x,y],[p1,p5],[],['x=x+1','y=y+1'],[]).
```

Figura 21: Rappresentazione Prolog del problema *Readers-Writer with priority*

7.2.2 Il codice Promela

Le novità presentate nel Paragrafo 7.2.1 si riflettono anche sul codice Promela. Al fine di supportare le nuove funzionalità è stata cambiata la struttura dati rappresentativa di un place che ora è definita da un array e da una variabile di tipo *byte* la quale mantiene il conteggio dei token nell'array. Se il place è stato definito per contenere elementi di due tipi allora, per il tipo aggiuntivo, sarà definito un secondo array. Inoltre sono state definite una serie di funzioni per vari scopi:

- getRandToken, per ottenere un binding random da un place con singolo colore;
- getRandToken2, per ottenere un binding random da un place con due colori;
- setToken, per inserire un valore in un place con singolo colore;
- setToken2, per inserire un valore in un place con due colori;
- getMinToken, per ottenere il binding con valore minimo da un place con singolo colore;
- getMaxToken, per ottenere il binding con valore massimo da un place con singolo colore;
- shift, per shiftare gli elementi di un array a seguito di una rimozione di un token da un place con singolo colore;
- shift2, per shiftare gli elementi di un array a seguito di una rimozione di un token da un place con due colori;

L'implementazione di tali funzioni è osservabile in Figura 22.

```
#define getRandToken(p,l,t)\
                i=0;\
                do \
                     ::  i<l-1 -> i++; \
                     ::  t=p[i]; shift(p,i); l--; break; \
                od

#define shift(place,pos) \
                c=0; \
                  for(c:pos+1..4){ \
                          place[c-1]=place[c];\
                  }\
                place[4]=0;

#define setToken(p,l,t)\
          p[l]=t;\
          l++;


#define getRandToken2(p0,p1,l,t0,t1)\
                i=0;\
                do \
                     ::   i<l-1 -> i++; \
                     ::  t0=p0[i]; t1=p1[i]; shift2(p0,p1,i); l--; break;\
                od
```

```
#define shift2(p0,p1,pos) \
                c=0; \
                  for(c:pos+1..4){\
                          p0[c-1]=p0[c];\
                          p1[c-1]=p1[c];\
                  }\
                p0[4]=0;\
                p1[4]=0;

#define setToken2(p0,p1,l,t0,t1)\
          p0[l]=t0;\
          p1[l]=t1;\
          l++;

#define getMinToken(p,l,t)\
                i=0; \
                t=p[i];\
                i1=i; \
                do \
                     ::   i<(l-1) -> i++;\
                                  if \
                                      :: p[i]<t ->t=p[i]; i1=i;
                                      :: else  \
                                  fi; \
                     ::  i==(l-1) -> shift(p,i1); l--; break; \
                od
```

Figura 22: Implementazione delle funzioni per il supporto alle CPN

E' opportuno fare alcune considerazioni: le funzioni in questione si appoggiano alle variabili i, c, i1 per comportarsi correttamente,ne è perciò vietato l'utilizzo nelle transizioni; inoltre per variare la capienza dei place non è sufficiente variare le sole costanti Max e Max1 ma bisogna anche correggere i cicli *for* delle funzioni *shift* e *shift2.*

La rappresentazione di una transizione in Promela per le CPN, grazie all'utilizzo di queste funzioni, varia di poco rispetto al caso delle PTN se non sono presenti condizioni di >,<,>=,<= (Figura 23).

```
:: (p6_len==0 && p4_len>0 && p5_len>0  && p7_len<Max) -> atomic{/* t5 */
                                        getMinToken(p4,p4_len,x);
                                        getRandToken(p5,p5_len,y);
                                        setToken2(p7_0,p7_1,p7_len,x,y);
                                    }
```

Figura 23: Esempio di transizione in una CPN

La gestione di questi operatori costringe ad adottare un strategia per cui si preleva comunque un token dai place in ingresso e se il token non soddisfa la condizione allora la transizione fallisce reinserendo tutti i token prelevati nel place di origine. Nel caso invece che tutte le condizioni siano soddisfatte allora si procede normalmente(Figura 24).

```
::(p6_len>0 && p1_len<Max ) -> atomic{
      getRandToken(p6,p6_len,x);
      if
            :: (x<5) ->
                  setToken(p1,p1_len,x);
            :: else ->
                  setToken(p6,p6_len,x);
                  goto LOOP;
      fi;
}
```

Figura 24: Esempio di transizione con gestione degli operatori di confronto

7.2.3 Il traduttore

A maggior ragione, data la maggior complessità delle CPN, lo sviluppo del traduttore è stato basato su un approccio *divide et impera* come nel Paragrafo 7.1.3. Il quantitativo di codice invariante rispetto alla rete specifica è superiore in quanto comprende anche l'implementazione delle funzioni, oltre alla definizione delle costanti e degli statement principali. D'altro canto la traduzione delle transizioni risulta essere più complessa in quanto ogni funzione utilizzata al suo interno va correttamente configurata, vanno opportunamente gestiti i nuovi operatori, in particolare nelle precondizioni, e devono essere fatti dei controlli sull'utilizzo corretto delle variabili il cui colore deve combaciare con quello del token. Il predicato *translate* non mette in luce perciò le novità sopracitate, che emergono nei predicati più specifici.

```
%generate Promela code.
translate(X):-  tell(X),
                print_defines,
                print_user_var_decl,
                print_places_decl,
                print_places_len,
                open_main_proc,
                init_marking,
                translate_transitions,
                close_main_proc,
                print_ltl_clauses,
                told.
```

Figura 25: Predicato *translate* per la traduzione delle CPN

I predicati che generano il codice invariante, escludendo *tell* e *told*, sono :

- *print_defines*, si occupa di generare il codice per la definizione delle costanti Max e Max1, per la dichiarazione delle variabili riservate alle funzioni e per la definizione delle funzioni stesse. Per ciascun elemento è stato definito un ulteriore predicato (Figura 26);

```
%useful defines
print_defines:- print_costants,
                print_reserved_var_decl,
                print_getRandToken,
                print_getRandToken2,
                print_shift,
                print_shift2,
                print_setToken,
                print_setToken2,
                print_getMinToken,
                print_getMaxToken.
```

Figura 26: Predicato per la traduzione di funzioni, variabili e costanti

- *open_main_proc* e *close_main_proc* che si occupano di aprire e chiudere lo statement del processo principale, il processo *init.*

Tutti gli altri predicati hanno bisogno di informazioni dalla rappresentazione Prolog della rete di Petri colorata:

- *print_user_var_decl*, genera la dichiarazione delle variabili indicate con il predicato *variable/2* (Figura 27);

```
%prints variables defined by user
print_user_var_decl:-nl,print_comment('User defined vars'),print_vars.
print_vars:-findall(X,variable(int,X),L0),(L0\=[],print_type(int),print_vars(L0),!;true),
     findall(Y,variable(byte,Y),L1),(L1\=[],print_type(byte),print_vars(L1),!;true),
     findall(Z,variable(bool,Z),L2),(L2\=[],print_type(bool),print_vars(L2),!;true).
print_type(X):-nl,print(X),tab(2).
print_vars([]).
print_vars([X|Y]):-(X\=c,X\=i,X\=i1; told,nl,print('Error: variables i,c,i1 are reserved'),fail),!,
          print(X),print_vars_rest(Y).
print_vars_rest([]):-print(';').
print_vars_rest([X|Y]):-(X\=c,X\=i,X\=i1; told,nl,print('Error: variables i,c,i1 are reserved'),fail),!,
               print(','),print(X),print_vars_rest(Y).
```

Figura 27: Predicato *print_user_var_decl* per la traduzione delle variabili definite dall'utente

- *print_places_decl*, genera la dichiarazione degli array che dovranno contenere i token di un certo place. Il tipo dell'array deve corrispondere con il colore del place (Figura 28);

```
%prints places data structures
print_places_decl:-nl,print_comment('Places are buffers which contains token of the same type of buffer'),
          print_places_1type,
          print_places_2type.
print_places_1type:- findall(X,place(X,[int],_),L0),(L0\=[],print_type(int),print_places(L0),!;true),
               findall(Y,place(Y,[byte],_),L1),(L1\=[],print_type(byte),print_places(L1),!;true),
               findall(Z,place(Z,[bool],_),L2),(L2\=[],print_type(bool),print_places(L2),!;true).

print_places_2type:-findall([X,Y,Z],place(X,[Y,Z],_),L),print_places_2(L).

print_places([]).
print_places([X|Y]):-(X\=c,X\=i,X\=i1,!; told,nl,print('Error: names i,c,i1 cannot be used'),fail),
          print(X),print('[Max]'),print_places_rest(Y).
print_places_rest([]):-print(';').
print_places_rest([X|Y]):-(X\=c,X\=i,X\=i1,!; told,nl,print('Error: names i,c,i1 cannot be used'),fail),
               print_place_name(X),print_places_rest(Y).
print_place_name(X):-print(','),print(X),print('[Max]').

print_places_2([]).
print_places_2([[X,Y,Z]|W]):-(X\=c,X\=i,X\=i1,!; told,nl,print('Error: names i,c,i1 cannot be used'),fail),
          print_type(Y),print(X),print('_0'),print('[Max]'),print(';'),
          print_type(Z),print(X),print('_1'),print('[Max]'),print(';'),
          print places 2(W).
```

Figura 28: Predicato *print_places_decl* per la definizione degli array

- *print_places_len*, genera la dichiarazione delle variabili che terminano con *_len* utilizzate per memorizzare il numero di token nel place (Figura 29);

```
print_places_len:-findall(X,place(X,_,_),L),print_len(L).
print_len([H|T]):-print_type('byte'), print(H),print('_len'),print_len_rest(T).
print_len_rest([]):-print(';').
print_len_rest([H|T]):-print(','),print(H),print('_len'),print_len_rest(T).
```

Figura 29: Predicato *print_places_len* per la dichiarazione dell variabili *_len*

- *init_marking*, configura le funzioni *setToken* e *setToken2* per l'inizializzazione del marking iniziale, nei place è specificato (Figura 30);

```
%fill place with initial marking
init_marking:-findall(Y,place(Y,_,_),L),fill_places(L).
fill_places([]).
fill_places([N|T]):-place(N,TY,TO),(TO==[];fill_place(N,TY,TO)),
            !,fill_places(T).
fill_place(_,_,[]).
fill_place(N,[T],[H|R]):-nl,tab(5),print('setToken('),
            print(N),print(','),
            print(N),print('_len,'),
            print(H),print(');'),
            fill_place(N,[T],R).

fill_place(N,[T0,T1],[[H0,H1]|T]):-nl,tab(5),print('setToken2('),
                        print(N),print('_0,'),
                        print(N),print('_1,'),
                        print(N),print('_len,'),
                        print(H0),print(','),print(H1),print(');'),
                        fill_place(N,[T0,T1],T).
```

Figura 30: Predicato *init_marking* per l'inizializzazione del marking.

- *translate_transitions*, la parte del traduttore più complessa in cui vanno definite le guardie e il corpo della transizione tenendo conto delle precondizioni e delle trasformazioni (Figura 31).

```
translate_transitions:-nl,print('LOOP: do'),
                findall(X,transition(X,_,_,_,_,_,_),L),
                print_transitions(L).

%just make recursion on transitions list
print_transitions([]):-nl,print('od').
print_transitions([H|T]):-print_transition(H),print_transitions(T).


print_transition(N):-nl,tab(5),print('::('),
           print_guard(N),print(') -> atomic{'),
           tab(5),print_comment(N),print_body(N),
           nl,tab(5),print('}').
```

Figura 31: Predicato *translate_transitions* per la generazione del codice relativo alle transizioni.

7.2.4 Le proprietà LTL

Come nel Paragrafo 7.1.4, anche per il problema dei *Readers-Writer with Priority* devono valere le stesse proprietà ltl di safety e liveness opportunamente rivisitate in modo che siano adattate alle nuove strutture dati. Ne va però aggiunta una nuova che esprima il concetto di priorità caratterizzante il nuovo problema. Questa formula è stata denominata *acc_priority* ed è osservabile in Figura 32 con le già note proprietà *safety* e *liveness.* La formula in Promela non è per niente concisa, ma si intende esprimere il fatto che *in tutti i percorsi, ad un certo punto, p4_len e p7_len sono positivi e tutti gli elementi di p4 sono maggiori o uguali a quello in p7_0.*

```
ltl safety {[] !(p6_len>0 && p7_len >0)}
ltl liveness{[]<> (p6_len>0 || p7_len >0)}
ltl acc_priority{[] true U (p4_len>0 && p7_len>0 && p7_0[0]<=p4[0] &&
p7_0[0]<=p4[1] && p7_0[0]<=p4[2] && p7_0[0]<=p4[3] && p7_0[0]<=p4[4])}
```

Figura 32: Proprietà ltl per il problema *Readers-Writer with Priority*

8 Conclusioni e possibili sviluppi

Nonostante rappresenti una versione ridotta, il modello costruito in Promela delle reti di Petri colorate si è rivelata sufficiente per la sperimentazione e la verifica con Spin di CPN per problemi classici della letteratura scientifica e di loro estensioni. La fusione di Prolog con Spin, grazie al traduttore, non riesce certo a produrre un tool paragonabile a CPNTools nè per capacità nè per usabilità, ma è solo un punto di partenza che può essere ancora ampiamente sviluppato in particolare per quanto riguarda la scelta e la combinazione dei colori, le gestione delle gerarchie, le funzioni utilizzabili nelle precondizioni e magari l'aggiunta di un'interfaccia grafica per rendere trasparenti i dettagli implementativi. Purtroppo i limiti di Promela/Spin, come la limitazione della capacità dei place a poche decine di token, impediscono la simulazione ed il model checking di CPN per problemi complessi a meno che non si decida di modificarne autonomamente il codice.

Indice delle figure

Bibliografia

1. **Emerson, E.A.** Temporal and modal logic. *Handbook of Theoretical Computer Science.* s.l. : Elsevier Science Publishers, 1990.

2. **Jensen, Kurt.** *A Brief Introduction to Coloured Petri Nets.* Aarhus : s.n.

3. —. *An Introduction to the Theoretical Aspects of Coloured Petri Nets.* Aarhus : s.n.

4. Download page. *CPNTools.org.* [Online] http://cpntools.org/download.

5. CPNTools Documentation. *CPNTools.org.* [Online] http://cpntools.org/documentation/start.

6. *The Spin model checker.* [Online] http://spinroot.com/spin/whatispin.html.

7. Promela Language Reference. *The Spin model checker.* [Online] http://spinroot.com/spin/Man/promela.html.

8. tuProlog Download Page. *tuProlog.* [Online] APICe. tuprolog.alice.unibo.it/.

9. **Murata, Tadao.** Petri nets: Properties, analysis and applications. Chicago : IEEE.

10. **Viroli, Mirko.** *Petri nets.* Cesena : s.n., 2009.