

Reti di Petri Algebriche

Cristiana Casanova
Linguaggi e Modelli Computazionali

February 27, 2012

Indice

1	Introduzione	3
2	Le Reti di Petri	4
2.1	Definizione formale delle reti di Petri	4
2.2	Reti Posti/Transizioni	5
2.3	Evoluzione di una rete di Petri	5
2.4	Proprietà delle reti di Petri	6
2.4.1	Reachability	6
2.4.2	Boundedness	6
2.4.3	Safeness	6
2.4.4	Liveness	6
2.4.5	Reversibility	7
2.4.6	Conservativeness	7
2.5	Limiti delle reti di Petri	7
3	Estensione delle reti di Petri	9
3.1	Reti di Petri High-Level	9
3.1.1	Reti di Petri Predicati/Transizioni	10
3.1.2	Reti di Petri Colorate	11
3.2	Reti di Petri Algebriche	13
3.2.1	Definizione	14
3.2.2	Specifiche algebriche	14
3.2.3	Produttori-Consumatori con reti di Petri algebriche . . .	16
3.2.4	Caso di studio: il problema della georeferenziazione in assenza di segnale Gps	20
3.3	Reti Concurrent Object-Oriented Petri Nets	25
4	Model Checking per reti di Petri algebriche	26
4.1	LTL e CTL	27
4.2	Problemi del model checking	28
4.3	AlPiNA	29
4.3.1	Framework Overview	29
4.3.2	Modellazione di una rete	29
4.3.3	Property checker	32
4.3.4	Clustering algebrico	35
4.3.5	Algebraic Net Unfolding	36
4.3.6	Caso di studio: il problema della georeferenziazione mo- dellato con AlPiNA	36
4.4	Ulteriori tools	44
5	Conclusioni	46

1 Introduzione

Ideate nel 1962 da Carl Adam Petri, le reti di Petri rappresentano un valido strumento per la modellazione delle comunicazioni e delle interazioni fra processi paralleli. Esse permettono infatti di superare i limiti dei diagrammi di flusso (dove le attività sono rigidamente serializzate), permettendo di rappresentare e descrivere globalmente i processi, consentendo anche di seguirne l'evoluzione e visualizzarne lo stato in cui ci si trova in un dato istante.

Il formalismo delle reti di Petri ha conosciuto una rapida diffusione grazie alla sua potenza espressiva, portando anche alla creazione di numerose varianti pensate per analizzare alcuni aspetti di sistemi reali.

Nel seguito ci si concentrerà su una delle estensioni del formalismo standard delle reti di Petri: le reti di Petri algebriche. Queste nascono dall'esigenza di voler modellare non solo un flusso di informazioni, ma anche il contenuto di queste ultime; a tale proposito vengono quindi introdotti gli *Algebraic Abstract Data Type* e le specifiche algebriche, concetti alla base di tale formalismo.

Nonostante da un lato si aumenti il potere espressivo rispetto alle reti di Petri standard, dall'altro si ha una maggior difficoltà nell'analisi e nella verifica del corretto funzionamento di questo tipo di rete. Vengono quindi introdotti alcuni dei tools presenti in letteratura per la loro modellazione, introducendo i concetti principali e le proprietà verificabili attraverso tali *model checkers*.

2 Le Reti di Petri

Le reti di Petri rappresentano un metodo per risolvere una serie molto vasta e complessa di problemi. Inizialmente utilizzate per la progettazione di sistemi informatici, hanno avuto negli ultimi anni un notevole sviluppo ed un ampliamento del loro campo di applicazione: esse sono, infatti, efficaci per rappresentare comportamenti complessi come la sincronizzazione, il succedersi asincrono di eventi, il verificarsi di operazioni concorrenti, di conflitti e di condivisione di risorse. Si possono definire come dei modelli astratti e formali atti a rappresentare la dinamica di sistemi che esibiscono attività asincrone e concorrenti.

I motivi fondamentali della loro utilità sono principalmente due: da un lato consentono di dare al modello di un sistema il rigore formale necessario per eliminare ambiguità nella rappresentazione e consentono l'analisi e la verifica sul comportamento del sistema stesso. Dall'altro tale formalismo è suscettibile di una rappresentazione grafica piuttosto semplice che ne facilita l'utilizzo.

In una rete di Petri gli elementi fondamentali sono due:

- i *places* che definiscono lo stato del sistema
- le *transizioni* che invece permettono di modificarne lo stato

Lo stato di un sistema è dato dall'unione di più stati parziali e indipendenti, mentre la transizione non riguarda lo stato globale del sistema, ma si limita a variarne solo una parte.

2.1 Definizione formale delle reti di Petri

Una rete di Petri può essere definita da una tripla

$$PN=(P,T,A)$$

dove P è l'insieme (finito) dei places, T è l'insieme (finito) delle transizioni ed A è l'insieme degli archi. Devono inoltre valere le seguenti proprietà:

1. $P \cap T = \emptyset$
2. $P \cup T \neq \emptyset$
3. $A \subseteq (P \times T) \cup (T \times P)$

dove si afferma che l'insieme dei places P e l'insieme delle transizioni sono disgiunti (1), si impone alla rete di non essere vuota (ci devono essere almeno un place ed una transizione) (2) e si afferma che un arco A connette soltanto nodi di tipo diverso, ossia collega soltanto posti a transizioni e transizioni a posti, ma non posti a posti o transizioni a transizioni (3).

2.2 Reti Posti/Transizioni

Una prima estensione delle reti di Petri è data dalle cosiddette reti Posti/-Transizioni (o reti P/T). Una rete P/T è costituita da una quintupla definita come

$$P/T = (P, T, A, W, M_0)$$

dove P, T ed A definiscono una rete generale e W e M_0 sono due funzioni:

- $W : A \rightarrow \mathbb{N}^+$
- $M_0 : P \rightarrow \mathbb{N}$

La prima formula permette di associare ad ogni arco un numero intero positivo detto *peso* o *molteplicità*, mentre la seconda formula definisce il marking iniziale della rete, ovvero associa ad ogni place un numero intero non negativo di *tokens*.

2.3 Evoluzione di una rete di Petri

Il sistema modellato da una rete P/T evolve tramite il verificarsi di uno o più tra gli eventi che in un determinato stato del sistema hanno la possibilità di avere luogo. Per modellare il verificarsi di un evento bisogna perciò considerare due aspetti:

- la possibilità che l'evento si verifichi, detto *abilitazione* di una transizione
- l'effetto che l'evento ha sullo stato del sistema, detto *regola di scatto* di una transizione

Si dice che una transizione t è *abilitata* in M quando ogni place p di ingresso a t contiene un numero di tokens almeno uguale al peso dell'arco che collega p a t . Lo scatto di una transizione t può avvenire solo se essa è abilitata a scattare nel marking M e produce, a partire da questo marking, un nuovo marking M' secondo la seguente regola di scatto: da ogni place p che sia in ingresso a t viene rimosso un numero di token uguale al peso dell'arco che collega p a t , ed in ogni place q in uscita a t viene depositato un numero di tokens uguale al peso dell'arco che collega t a q . Il marking dei places che non siano né di ingresso né di uscita a t rimane inalterato.

Lo scatto di una transizione t è quindi un evento locale della rete, cioè la cui possibilità di avvenire può essere decisa in base al marking dei soli places che sono direttamente collegati in ingresso a t ed i cui effetti riguardano solo i places direttamente collegati in ingresso ed in uscita a t .

2.4 Proprietà delle reti di Petri

Le proprietà di una rete di Petri sono:

- Reachability
- Boundedness e Safeness
- Liveness
- Reversibility
- Conservativeness

Tutte queste proprietà dipendono dal marking iniziale della rete: è quindi necessario verificare una proprietà per tutti i possibili marking della rete presa in esame.

2.4.1 Reachability

Un marking M_k è **raggiungibile** dal marking iniziale M_0 se esiste una sequenza di esecuzioni di transizioni che trasforma M_0 in M_k .

(E' stato dimostrato che il problema della raggiungibilità è risolubile con algoritmi di complessità almeno esponenziale).

2.4.2 Boundedness

Una rete di Petri si dice **k-limitata** se il numero di tokens in ciascun place della rete è uguale o minore al numero intero positivo k per ogni marking raggiungibile da M_0 , cioè:

$$M_k(p_i) \leq k$$

per ciascun posto p_i e per ogni marking M_k raggiungibile da M_0 (incluso M_0).

2.4.3 Safeness

Una rete di Petri si dice **safe** se è *1-bounded*.

Dal momento che, in un sistema modellato tramite reti di Petri, lo stato è rappresentato dal marking dei places, le proprietà di boundedness e safeness implicano uno spazio degli stati finito.

2.4.4 Liveness

Una transizione t_j è **live** se è potenzialmente eseguibile in tutti i marking raggiungibili.

Una rete di Petri si dice **live** se, indipendentemente dal marking raggiunto da M_0 , è potenzialmente possibile eseguire una qualsiasi transizione della rete

di Petri attraverso una opportuna sequenza di esecuzioni. In particolare una rete di Petri si dice *live* se tutte le transizioni della rete sono *live*.

La proprietà di liveness è strettamente connessa all'assenza di possibilità di deadlock.

2.4.5 Reversibility

Una rete di Petri è **reversibile** se, in ogni marking M_k raggiungibile dal marking iniziale M_0 , M_0 è raggiungibile da M_k (ovvero se è sempre possibile “tornare indietro” al marking iniziale).

Un marking M' è un **home state** se, in ogni marking M_k raggiungibile dal marking iniziale M_0 , M' è raggiungibile da M_k (ovvero è sempre possibile “andare” in tale marking).

2.4.6 Conservativeness

La definizione matriciale di una rete di Petri consente di individuare alcune proprietà che non mutano a seconda del marking raggiungibile: tali proprietà prendono il nome di “invarianti”.

Esistono due tipi di invarianti:

- **P-invarianti**: riguardano il marking dei places
- **T-invarianti**: riguardano le transizioni

P-invarianti: una rete di Petri è **strettamente conservativa** se, per ogni marking M raggiungibile da M_0 , il numero di tokens nella rete non varia, cioè

$$\sum_{i=1}^n M(p_i) = \sum_{i=1}^n M_0(p_i)$$

Una rete di Petri si dice **conservativa** se esiste un vettore di interi positivi $x \in \mathbb{N}^+$ tale che, per ogni marking M raggiungibile da M_0 , vale

$$x^T \cdot M = x^T \cdot M_0$$

cioè il numero di tokens pesato tramite x non varia.

Se una rete di Petri è conservativa allora è anche limitata.

T-Invarianti: i T-invarianti individuano sequenze di scatti cicliche, che possono avere luogo ripetutamente, riportando la rete esattamente al marking di partenza.

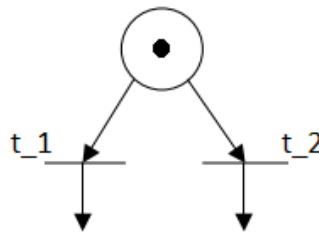
2.5 Limiti delle reti di Petri

Nonostante le reti di Petri modellino piuttosto bene certi aspetti dei sistemi, il loro uso ha rivelato alcune debolezze nell'utilizzo per la specifica di software. Innanzitutto, come le FSM (Macchine a Stati Finiti), sono un modello orientato al controllo. I tokens rappresentano in genere il flusso di controllo dell'esecuzione di diverse azioni. Essi però sono “anonimi”: ad esempio la presenza di un token

in un place può indicare solo la presenza di un messaggio in un buffer, non il suo contenuto.

Questa semplicità può tuttavia rivelarsi utile. Spesso (ad esempio quando siamo interessati ad analizzare il flusso di messaggi all'interno di una rete) la questione importante è se un messaggio, prodotto da qualche parte, giunga o meno a destinazione. In questi casi, il contenuto del messaggio può essere considerato irrilevante.

Si supponga, ad esempio, di volere specificare un sistema in cui un messaggio debba essere spedito lungo uno tra i due differenti canali:



viene ad esempio scelto t_1 se il messaggio è ben formato (ovvero se contiene un numero pari di 1), mentre viene scelto t_2 se il messaggio è scorretto.

La rete mostrata in figura fa sì che quando un messaggio è pronto per essere spedito (ovvero con un token nel place come raffigurato), la scelta tra i due canali sia non deterministica. Pertanto la figura non è una descrizione adeguata del sistema che si vuole modellare, dal momento che la scelta della transizione dipende fortemente dal contenuto del messaggio.

Nell'esempio, lo scatto di una transizione dovrebbe infatti dipendere dal valore del messaggio, ma siccome esso è rappresentato da un token anonimo, ciò è chiaramente impossibile: il token indica infatti solo la sua in quel dato place.

E' quindi necessario associare ai token i *valori* dei messaggi ed essere in grado di *calcolare* il valore di ciascun token.

Un'altra debolezza delle reti di Petri è dovuta al fatto che, in generale, non è possibile specificare una *politica di scelta* tra diverse transizioni abilitate. Per esempio vi possono essere casi in cui si potrebbe imporre una politica per alternare l'abilitazione di diverse transizioni al fine di evitare situazioni di starvation.

La *temporizzazione* è, infine, un altro aspetto critico di alcuni sistemi: la maggior parte dei modelli di sistemi, incluse le reti di Petri, non prende esplicitamente in considerazione il tempo.

Fortunatamente la flessibilità del modello delle reti di Petri ha consentito la sua estensione in diverse direzioni, mantenendo le caratteristiche originali.

3 Estensione delle reti di Petri

Le reti di Petri classiche o P/T fin qui presentate costituiscono uno dei primi modelli di rete definiti in letteratura. Tuttavia, nonostante il loro elevato potere espressivo e la loro immediatezza rappresentativa, le reti P/T hanno delle carenze nella rappresentazione di un cospicuo numero di situazioni e nella modellazione di grandezze come il tempo. Per ampliare il loro potere di modellazione, le reti P/T sono state variamente estese. Esistono infatti molte tipologie/estensioni di tale formalismo: alcune di esse introducono dei dettagli che in realtà non aumentano la potenza del modello di base, altre aggiungono invece proprietà che non possono essere modellate con le reti di Petri d'origine. Inoltre con questi formalismi ci si avvicina a dei veri e propri linguaggi di programmazione ad alto livello: in questo modo è come se un modello corrispondesse ad un programma. Si osserva però che, mentre da un lato tali estensioni permettono di aumentare il potere espressivo dei modelli delle reti di Petri, dall'altro ne compromettono o complicano le possibilità di analisi: diventa infatti difficile provare proprietà di un modello esteso, limitando l'analisi alla sola esecuzione diretta della rete.

3.1 Reti di Petri High-Level

Nel complesso insieme delle reti di Petri è possibile annoverarne una tipologia nota come High-Level Petri Nets. Tale formalismo si differenzia profondamente dalle reti di Petri standard in quanto ha un potere di espressività di più alto livello.

Esse possono essere definite come:

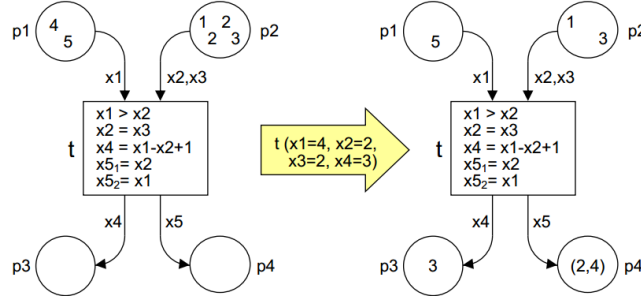
$$HLPN=(P,T,C,F,Pre,Post,M_0)$$

dove:

- P rappresenta un insieme finito di places
- T è un insieme finito di transizioni disgiunto da P ($P \cap T = \emptyset$)
- C è un insieme non vuoto di tipi
- $F : P \cup T \rightarrow C$ è una funzione usata per definire i places e determinare il *modo della transizione* (assegnamento o sostituzione di valori per le variabili della transizione che soddisfano una data condizione)
- Pre e $Post$ rappresentano le pre-condizioni e post-condizioni
- M_0 è il marking della rete

In una rete modellata in questo modo, ogni singolo token è definito da un tipo (*token strutturato*) e sui places si andrà a definire l'insieme ammissibile di tipi che possono ammettere.

Viene fornito di seguito un primo esempio di reti di Petri High-Level:



I places $p1$, $p2$ e $p3$ possono contenere dati di tipo INT; $p4$ invece conterrà valori di tipo INTxINT.

Il marking iniziale della rete è dato dai valori numerici posti all'interno dei places $p1$ e $p2$. La transizione è caratterizzata, invece, da assegnamenti di variabili e condizioni che devono essere soddisfatte. Essa scatterà nel caso in cui vengano assegnati dei valori ammissibili alle variabili poste sugli archi, che permetteranno perciò la validazione delle espressioni specificate sulla transizione stessa.

Di seguito verranno introdotte alcuni tipi di rete di Petri High-Level, focalizzandosi principalmente sulle reti di Petri algebriche.

3.1.1 Reti di Petri Predicati/Transizioni

Una prima estensione delle reti di Petri è stata quella proposta da Genrich e Lautenbach nel 1981 con lo scopo di fornire una nuova dimensione di astrazione al modello P/T precedentemente introdotto: questo tipo di rete prende il nome di rete Predicati/Transizioni (o reti Pr/T).

Il paradigma a cui si fa riferimento è di tipo logico e si basa sul calcolo dei predicati del primo ordine. Il vantaggio che se ne trae è quello di una rappresentazione più immediata e naturale dei vincoli che governano l'evoluzione della rete: inoltre si ha una maggiore leggibilità degli oggetti che costituiscono il marking della rete stessa. I vantaggi sono però controbilanciati da una maggiore difficoltà dell'analisi formale delle proprietà strutturali di quest'ultima.

Il marking di un place è costituito, in questo caso, da un insieme di n -ple la cui arità è costante e dipende dal place in esame. Analogamente sugli archi si hanno etichette che saranno n -ple di variabili di arità uguale a quella definita dal predicato da cui l'arco entra od esce: una sostituzione di tali variabili con delle costanti rappresenta il punto di partenza per verificare se una transizione è abilitata o meno. Ogni transizione ha quindi un predicato denominato *guardiano* ad essa associato, che dovrà essere verificato in base alla sostituzione delle variabili con le costanti che arrivano dai places ad essa connessi.

Possiamo ad esempio pensare di avere un place A da interpretare come una relazione tra un abbonato telefonico e il numero di scatti associato: questa coppia rappresenta, in termini di logica, un predicato, in quanto definisce la

relazione fra due oggetti. Intuitivamente si può capire che ciascun place avrà un'arità legata all'arità della relazione che identifica.

3.1.2 Reti di Petri Colorate

Un approccio più generale rispetto al precedente è quello introdotto dalle reti di Petri colorate. In questa estensione ad ogni place è associato un tipo che determina il tipo di dato (*colorset*) che il place può contenere. I valori che possono assumere i dati associati ai places vengono dichiarati al di fuori della rete, vicino ai places stessi: le dichiarazioni quindi, implicitamente, specificano le operazioni che possono essere eseguite sui dati coinvolti.

In ogni place è ammesso che vi siano più tokens dello stesso colore (*multi-set*) e l'ordine, in generale, non è importante.

Si descrive di seguito come evolve il marking della rete dal momento che, rispetto alle reti finora introdotte, presenta alcune novità. Il problema principale è quello concernente il concetto di transizione abilitata e di come specificare la regola di scatto. Dal momento che per queste reti i tokens sono delle n -ple, bisogna tenere conto di:

- quali tokens sottrarre dai places in ingresso ad una transizione
- quali tokens aggiungere ai places di uscita

La scelta dei tokens da eliminare è realizzata utilizzando delle funzioni *guardia* in corrispondenza delle transizioni. Tali funzioni determinano, insieme alle condizioni di abilitazione delle reti di Petri, la nuova condizione di abilitazione. La guardia rappresenta una condizione logica da verificare e può valutare se, nei places cui fa riferimento, esiste un insieme di tokens aventi determinate caratteristiche, espresse dalla guardia stessa. Affinché una transizione risulti abilitata è quindi necessario che siano soddisfatte le condizioni di abilitazione valide per le reti di Petri standard e, in aggiunta, sia soddisfatta la guardia.

I tokens da eliminare in seguito allo scatto della transizione saranno, naturalmente, tra quelli che soddisfano la guardia.

Nel caso in cui una transizione non abbia nessuna guardia, la sua condizione di abilitazione è quella valida per le reti di Petri standard.

I tokens che saranno accodati nei places saranno indicati dalle relazioni poste sugli archi, del tutto analoghe a quelle viste nelle reti precedenti.

Riconducendoci all'esempio dei produttori-consumatori, una possibile soluzione al problema è la seguente:

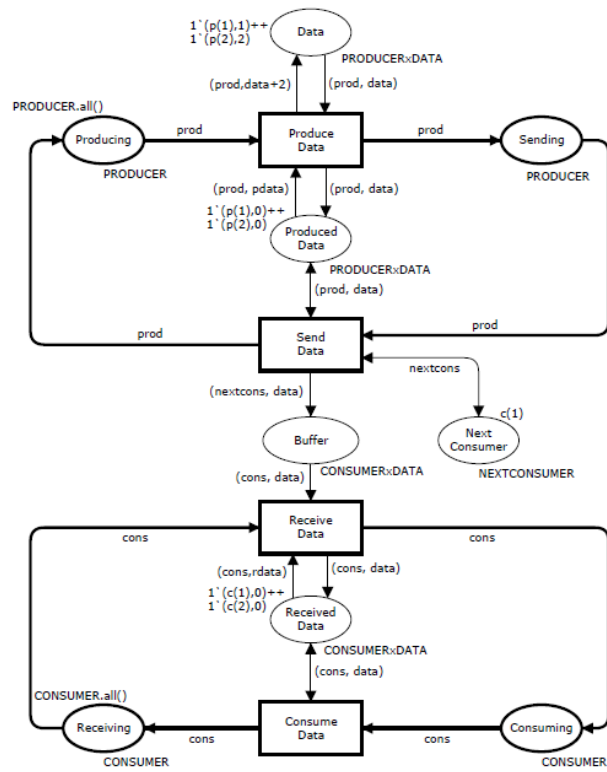


Figura 1: Produttori-Consumatori con Reti di Petri colorate

In particolare si possono notare alcune variazioni rispetto al modello standard delle reti di Petri: sotto ogni place sono presenti delle iscrizioni in maiuscolo che indicano il tipo dell'insieme dei colori possibili (ad esempio il place *Buffer* ha colourset $CONSUMER \times DATA$, definito come il prodotto tra il tipo *CONSUMER* ed il tipo *DATA*, ovvero contiene tutte le coppie dove il primo elemento è un consumatore ed il secondo elemento è il messaggio da consumare). L'iscrizione posta in alto a sinistra del place indica, invece, il marking iniziale dei places: ad esempio nel caso del place *Data*, l'iscrizione denota la presenza di due tokens: uno costituito dalla coppia $(p1(1),1)$ e l'altro dalla coppia $(p2(2),2)$. I simboli $++$ sono invece operatori usati per costruire il multi-set che consiste, in questo caso, di due colori. L'assenza di iscrizioni di questo tipo indica l'assenza di tokens in quei places.

Analogamente al caso precedente, i colori rimossi dai places di input e inseriti nei places di output, sono determinati per mezzo delle espressioni poste sugli archi: solo quando tutte le variabili delle espressioni sono vincolate ai valori del tipo corretto, allora l'espressione può essere valutata. Per abilitare una transizione è quindi necessario trovare un binding per le variabili che compaiono nelle espressioni in input ad essa.

3.2 Reti di Petri Algebriche

Inventate da Jacques Vautherin nel 1985 e approfondite successivamente da Wolfgang Reisig, le reti di Petri algebriche rappresentano un'evoluzione delle reti di Petri standard dove i tokens vengono sostituiti dai cosiddetti *Algebraic Abstract Data Types*.

Il formalismo è caratterizzato da due aspetti:

- il controllo della rete viene gestito dalla rete di Petri
- i dati vengono invece gestiti dagli AADTs creati

La definizione si basa sui concetti standard delle reti di Petri (places, transizioni pre e post-condizioni) e sulle nozioni classiche di *specifiche algebriche*: esse permettono di assegnare una determinata interpretazione ai tokens presenti nella rete.

I tokens sono infatti modificati per avere un valore di un tipo appropriato: un intero, un array di byte o un *ambiente*; anche le transizioni, come già visto prima, vengono modificate in modo da essere associate a *predicati* e *funzioni*. La regola di scatto è quindi basata, anche in questo caso, sui valori delle variabili, oltre che alla presenza o meno dei tokens.

Il vantaggio di questo formalismo è la capacità di descrivere in modo conciso problemi piuttosto complessi.

3.2.1 Definizione

Una rete di Petri algebrica può essere definita come:

$$A = (Spec, N, \phi, \xi, \lambda, m_0)$$

dove:

- $Spec$ è una specifica algebrica
- $N = (P, T, A)$ è una rete come definita precedentemente
- $\phi : P \rightarrow S$ è l'assegnamento di un *sort*
- ξ è una funzione che permette di assegnare ad ogni transizione $t \in T$ un insieme finito di variabili $\xi(t)$
- λ è una funzione che assegna ad ogni coppia $[p, t]$ o $[t, p]$ un multi-set di termini costruiti in base alle operazioni (OP) ammissibili e rispettando il tipo $\lambda(P)$
- m_0 è il marking iniziale della rete

3.2.2 Specifiche algebriche

Le specifiche algebriche definiscono un sistema come un'algebra eterogenea, ovvero come una collezione di diversi insiemi su cui vengono definite diverse operazioni.

Formalmente è definita come:

$$Spec = (S, OP, E)$$

dove:

- S rappresenta l'insieme di sorts con associati dominio e codominio
- OP rappresenta l'insieme delle operazioni
- E rappresenta l'insieme di *assiomi*
- X (non presente nella definizione formale) rappresenta invece l'insieme di variabili all'interno di S

Di seguito verrà proposto un esempio al fine di comprendere appieno il significato di tali specifiche.

Supponiamo di voler specificare un sistema per la gestione delle stringhe e di voler descrivere le seguenti operazioni:

- creazione di nuove stringhe vuote (operazione *new*)
- concatenamento di stringhe (operazione *append*)

- aggiunta di un nuovo carattere alla fine di una stringa (operazione *add*)
- controllo della lunghezza di una data stringa (operazione *length*)
- controllo se una stringa è vuota (operazione *isEmpty*)
- controllo se due stringhe sono uguali (operazione *equal*)

Una breve osservazione della lista indica che gli insiemi coinvolti, oltre all'insieme delle stringhe chiamato **String**, sono:

- **Char** che rappresenta l'insieme dei caratteri alfabetici
- **Nat** che rappresenta l'insieme dei numeri naturali
- **Bool** che rappresenta l'insieme dei valori logici **true** e **false**

La collezione di insiemi che formano l'algebra eterogenea è definita *signature*, che formalmente può essere descritta come:

$$\Sigma = (S, OP)$$

dove S è un insieme di sorts (letteralmente definiti come “tipi”) $S = S_1, \dots, S_k$, e OP è un insieme di operazioni definite come

$$OP = \bigcup_{a \in S_f^*, s \in S} Op_a^s$$

dove $\forall a \in S_f^*$ (dove S_f^* denota l'insieme delle sequenze finite di elementi di S) e $\forall a \in S$, gli Op_a^s sono gli insiemi, a due a due disgiunti, dei “simboli di operatore” con “sorta argomento” a e “sorta risultato” s .

Intuitivamente si può comprendere che i tipi sono considerati come un insieme di valori sui quali vengono applicate determinate operazioni. Un'operazione non è altro che una funzione di arità n ($n \geq 0$) definita su S . Le operazioni con arità 0 vengono chiamate *costanti*.

Per definire un'algebra eterogenea è quindi necessario specificare la sua signature, le operazioni coinvolte e i loro domini e codomini.

Questa definizione prende il nome di *sintassi* dell'algebra e può essere espressa mediante diverse notazioni. Nel seguito verrà proposta una notazione basata sul linguaggio di specifica utilizzato dal tool ALPiNA studiato per la modellazione di sistemi con reti di Petri algebriche.

Adt String

Sorts String; Char; Nat; Bool;

Operations

```

new : -> String;
append : String , String -> String;
add : String , Char -> String;
length : String -> Nat;
isEmpty : String -> Bool;
equal : String , String -> Bool;
```

Il significato delle singole operazioni è in questo caso molto semplice; la *semantica* dell'algebra deve invece essere specificata in maniera precisa, usando *equazioni* che sono intese a definire le *proprietà* essenziali che devono verificarsi quando sono eseguite le operazioni. Per questo motivo tali equazioni prendono il nome di *assiomi* dell'algebra.

La formalizzazione di alcune proprietà è definita quindi come:

```

Adt String
Sorts String; Char; Nat; Bool;
Generators
    zero: Nat;
    suc  : Nat;
Operations
    new  : -> String;
    append : String , String -> String;
    add  : String , Char -> String;
    length : String -> Nat;
    isEmpty : String -> Bool;
    equal : String , String -> Bool;
Axioms
    isEmpty(new) = true;
    length(new) = zero;
    equal(new,new) = true;
    ....

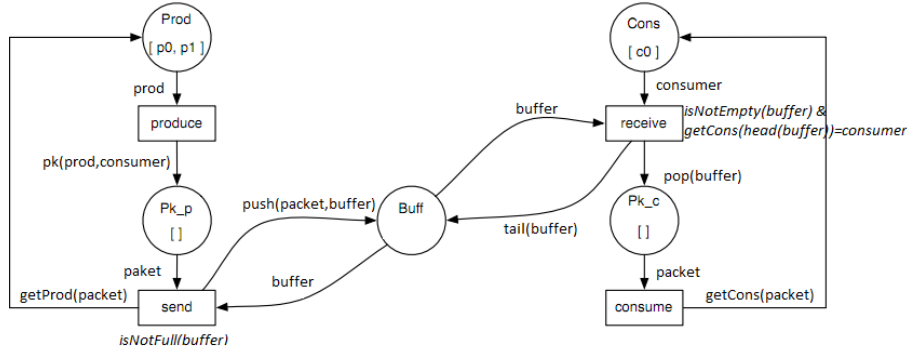
```

Le equazioni vengono dette assiomi in quanto, a partire dalla descrizione delle “verità” sopra definite, è possibile derivare ulteriori proprietà.

E’ infine possibile introdurre anche caratteri costanti come “a”, “b”, ecc. mediante la dicitura **Variables** seguita dalla dichiarazione delle variabili stesse.

3.2.3 Produttori-Consumatori con reti di Petri algebriche

Supponiamo di prendere come esempio, anche in questo caso, il problema dei produttori-consumatori:



Come si può notare, la modellazione del problema mediante questo formalismo è del tutto analoga a quella mostrata dalle reti di Petri colorate. I tipi di dati ammessi dai vari places non vengono però dichiarati in maiuscolo al di sotto dei places stessi ed il marking è deducibile a partire dagli elementi presenti all'interno dei places.

Nell'esempio mostrato si parte da un marking iniziale con due tokens nel place *Prod* e da un token nel place *Cons*. I tipi di dati ammessi dai places sono quelli dichiarati per le variabili contenute al loro interno: nel caso in questione *p0* e *p1* hanno tipo *Producer*, mentre *c0* ha tipo *Consumer*.

La transizione *produce*, l'unica abilitata inizialmente, potrà scattare solamente se il tipo di token estratto dal place *Prod* matcherà con il tipo espresso dall'etichetta posta sull'arco in esame. L'output della transizione sarà dato, invece, da una tupla costituita da due elementi: un elemento di tipo *Producer* e un elemento di tipo *Consumer* (la tupla sarà anch'essa associata ad un tipo che nel nostro caso prende il nome di *Packet*). Con un ragionamento del tutto analogo a Prolog, si può notare che la variabile *consumer* sarà inizialmente una variabile *free*, in quanto non vi è nessun binding possibile con i parametri di ingresso alla transizione.

Successivamente potrà scattare la transizione *send*, la quale prenderà in ingresso un token di tipo *Packet* e darà in output due tokens:

- uno sarà di tipo *Producer* e tornerà nello stato iniziale della rete
- l'altro sarà costituito da un ulteriore tupla di tipo *Buffer*

E' da notare come su questa transizione sia però presente un guardiano, che se non soddisfatto, non ne permetterà l'abilitazione. La condizione da soddisfare è quella relativa al controllo del riempimento del Buffer: in caso di capacità esaurita non sarà infatti possibile inserire ulteriori dati all'interno del contenitore.

Il ragionamento fin qui descritto è da applicare all'intera rete: e' necessario, però, definire gli ADT necessari alla modellazione.

Nel caso in questione possiamo pensare di descrivere i seguenti tipi:

- Packet: è il tipo associato alla tupla contenente le informazioni relative al produttore e al consumatore
- Consumer: definisce il tipo caratterizzante i tokens rappresentanti i consumatori
- Producer: analogo al Consumer, permette di definire il tipo rappresentante i produttori del sistema
- Buffer: permette di modellare il Buffer condiviso
- Naturals e Boolean: rappresentano i tipi di base necessari

Di seguito vengono perciò mostrati gli ADT necessari alla modellazione, utilizzando il linguaggio di specifica adottato dal tool AlPiNA descritto nel capitolo successivo:

Adt Producers

Sorts

producer;

Generators

p0 : producer;

p : producer $\rightarrow$ producer;

Axioms

$p^2(p0) = p0$;

Adt Buffer

Sorts

buffer;

Generators

empty : buffer;

push : packet, buffer $\rightarrow$ buffer;

Operations

size : buffer $\rightarrow$ nat;

head : buffer $\rightarrow$ packet;

tail : buffer $\rightarrow$ buffer;

isNotFull : buffer $\rightarrow$ bool;

isNotEmpty : buffer $\rightarrow$ bool;

Axioms

size(empty) = zero;

size(push(\$p, \$b)) = suc(size(\$b));

head(push(\$p, \$b)) = \$p;

tail(push(\$p, \$b)) = \$b;

isNotFull(\$b) = le(size(\$b), suc³(zero));

```

isNotEmpty(empty) = false;
isNotEmpty(push($p, $b)) = true;

Variables
  p : packet;
  b : buffer;

Adt Consumers
Sorts
  consumer;
Generators
  c0 : consumer;
  c : consumer -> consumer;

Axioms
  c^1(c0) = c0;

Adt Packet
Sorts packet;
Generators
  pk : producer, consumer -> packet;
Operations
  getProd : packet -> producer;
  getCons : packet -> consumer;

Axioms
  getProd(pk($p, $c)) = $p;
  getCons(pk($p, $c)) = $c;

Variables
  p : producer;
  c : consumer;

```

3.2.4 Caso di studio: il problema della georeferenziazione in assenza di segnale Gps

Il principio del Gps si basa sul calcolo della distanza del ricevitore (ovvero della nostra unità portatile) dai satelliti in orbita che si riescono a rilevare. Per effettuare il calcolo della posizione, lo strumento deve ricevere il segnale di almeno tre satelliti, anche se si necessiterebbe di un quarto satellite atto alla verifica e calcolo dell'altitudine.

Il ricevitore Gps utilizza il processo di trilaterazione per determinare la propria posizione sulla superficie terrestre utilizzando le informazioni concernenti i segnali orari presenti a bordo dei tre satelliti. Oltre a tali informazioni, ogni satellite della costellazione Gps emette segnali a intervalli regolari che vengono ricevuti dai dispositivi Gps in modo tale da poter calcolare la distanza tra il dispositivo e ciascun satellite preso in considerazione (la distanza viene calcolata come ritardo tra il segnale di invio e quello di ricezione del segnale).

Successivamente alla ricezione di tali informazioni, il Gps può eseguire i calcoli in base al processo di trilaterazione.

In assenza di segnali satellitari, l'idea è quella di risalire alle informazioni relative alla posizione del nostro Gps grazie alla comunicazione con altre entità Gps e sfruttandone le informazioni ricevute: si provvederà perciò a contattare le entità Gps più vicine, ed utilizzare tali informazioni nel calcolo previsto. I parametri ottenuti in questo scenario sono però differenti: i Gps forniranno, infatti, informazioni relative alle proprie coordinate terrestri e non una distanza calcolata in base al ritardo. E' per questo motivo che il processo di trilaterazione varierà a seconda dei parametri ricevuti in input.

L'idea di utilizzare le reti di Petri algebriche per la formalizzazione di tale modello risulta essere particolarmente conveniente rispetto alle reti di Petri standard:

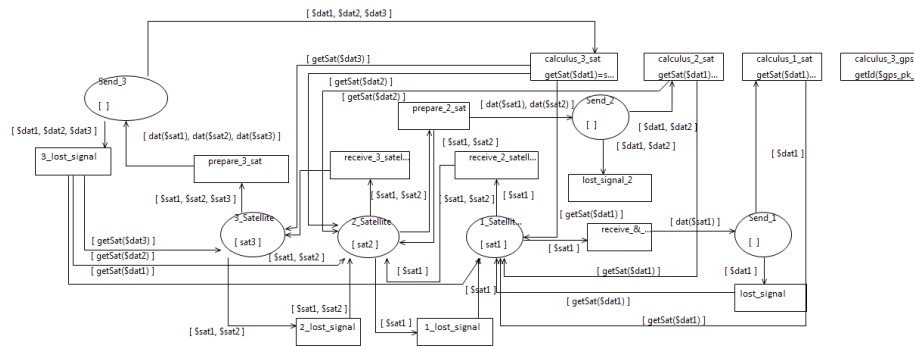
- il calcolo della trilaterazione cambia in base ai dati in ingresso: in particolare a seconda delle quattro combinazioni ottenibili da tre satelliti e tre Gps, i calcoli cambieranno profondamente. L'uso delle reti di Petri algebriche consente il controllo dei tipi di dati in entrata al fine di evitare errori
- la rappresentazione delle entità da modellare (Gps e satelliti) come fossero tuple di informazioni contenenti parametri come distanza, coordinate, ecc. risulta in questo caso agevolata
- l'utilizzo di operazioni matematiche esprimibili mediante ADT risulta anch'essa essere fondamentale per l'utilizzo del modello

Il modello proposto consente di gestire le principali eccezioni e comportamenti che il sistema assumerebbe:

- rimozione di un segnale Gps precedentemente disponibile
- rimozione di un segnale satellitare precedentemente disponibile

- in presenza di N segnali Gps, il modello provvederà a “prelevare” i tre segnali con distanza inferiore. In particolare ogni Gps è rappresentato da una tupla contenente un Id ed una distanza (un intero), che sarà appunto parametro di confronto con gli altri segnali disponibili
- controllo dei valori di input per il processo di trilaterazione

Per quanto riguarda la parte relativa alla ricezione satellitare, viene di seguito mostrata la modellazione proposta:



I places **3.Satellite**, **2.Satellite**, **1.Satellite** rappresentano gli stati contenenti i tre satelliti necessari al calcolo, mentre i places **Send_3**, **Send_2** e **Send_1** rappresentano gli stati in cui transitiamo a seconda del numero di segnali satellitari ricevuti. A seconda di tale numero, saranno abilitate le transizioni che permetteranno di incapsulare i dati relativi alle informazioni ricevute ed eseguire il calcolo di triangolazione. In particolare le transizioni

- **calculus_3_sat**
- **calculus_2_sat**
- **calculus_1_sat**
- **calculus_3_gps**

consentono tale processo in base ai tipi di dato ricevuti in ingresso: il primo esegue il calcolo avendo a disposizione solo tre satelliti, il secondo due satelliti ed un Gps, il terzo un satellite e due Gps, mentre l'ultimo solo tre segnali Gps. In questa prima modellazione, come si può notare, l'ultima transizione non è associata ad alcun place. Questo poiché essa sarà abilitata solamente nel caso in cui si abbiano tre Gps in entrata.

E' stata inoltre modellata la possibilità di rimuovere un segnale che in precedenza risultava disponibile: questo evento viene simulato attraverso le transizioni

- `lost_signal_3`: nel caso in cui si siano ricevuti tre segnali satellitari, simula la perdita di tutti e tre riportando il modello allo stato iniziale
- `lost_signal_2`: nel caso in cui si siano ricevuti tre segnali satellitari, consente di simulare la perdita dell'ultimo segnale ricevuto riportando il sistema allo stato in cui si hanno solamente due segnali satellitari disponibili
- `lost_signal_1`: analogo al precedente, ma con solo due segnali disponibili, riportando il sistema ad uno stato con un solo segnale visibile
- `lost_signal`: consente di simulare la perdita dell'unico segnale satellitare a disposizione

Al termine del calcolo, le entità rappresentanti i segnali satellitari torneranno nello stato iniziale del sistema.

La parte relativa ai Gps è stata invece modellata in questo modo:

- si è utilizzato un contatore, `Gps_counter` per tenere traccia dei Gps disponibili: esso risulta utile sia per il conteggio di eventuali rimozioni dei segnali, che per il controllo delle risorse presenti
- a seconda dei Gps visibili, il sistema provvederà a inserire ciascuna risorsa nei places `First.Gps`, `Second.Gps`, `Third.Gps`, rappresentanti le entità che la nostra antenna portatile potrà effettivamente contattare
- nel caso in cui il numero di Gps sia maggiore di tre, si effettuerà un calcolo per inserire nei relativi places i tre Gps con distanza inferiore. La distanza, come già accennato precedentemente, è rappresentata da un intero inizializzato all'inizio del processo, di conseguenza verranno presi in considerazione i tre con numero inferiore.
- in ogni momento è possibile rimuovere un Gps dai relativi place con le seguenti transizioni:
 - `remove_third_gps`: consente di simulare la perdita del terzo segnale Gps ricevuto (nel caso in cui i Gps iniziali risultassero essere maggiori di tre, allora eliminerà quello più distante)
 - `remove_second_gps`: consente di simulare la perdita del secondo segnale Gps ricevuto
 - `remove_first_gps`: analogo al precedente, simula la perdita dell'unico segnale Gps disponibile

Di seguito è mostrata la modellazione della parte relativa alla ricezione di segnali Gps.

3.3 Reti Concurrent Object-Oriented Petri Nets

Le reti CO-OPN rappresentano un formalismo per la descrizione di aspetti concorrenti di un sistema, che utilizza le reti di Petri algebriche. In particolare una rete CO-OPN può essere vista come un insieme di oggetti ciascuno dei quali rappresentato da una rete di questo tipo.

Formalmente una specifica CO-OPN viene definita come:

$$\langle Spec_O = Spec, O, T_{ext}T_{int}, P, X, AX \rangle$$

dove:

- O è l'insieme degli oggetti
- T_{int} è l'insieme delle transizioni *locali*
- T_{ext} è l'insieme delle transizioni *esterne*
- P è l'insieme dei places
- X è l'insieme delle variabili
- AX è l'insieme degli assiomi

Come già detto, un oggetto è quindi costituito da una rete di Petri algebrica dove si possono distinguere due tipi di transizioni: la prima è chiamata transizione esterna o più semplicemente *metodo*; la seconda viene invece definita come transizione locale o *interna*.

Le transizioni interne vengono utilizzate per modellare l'evoluzione di un oggetto indipendentemente dagli altri oggetti costituenti la CO-OPN: inoltre tali transizioni non sono visibili dagli altri oggetti, ma solo dalla rete che li contiene.

Le transizioni esterne sono, invece, quelle che permettono la comunicazione tra i vari oggetti del sistema: in particolare la comunicazione tra oggetti avviene mediante meccanismi di sincronizzazione del tutto analoghi ai linguaggi di programmazione Object-Oriented. Per ogni transizione esterna si possono indicare i seguenti quattro costrutti:

- *sim* indica l'esecuzione simultanea di due o più azioni
- *seq* indica l'esecuzione sequenziale di due azioni
- *empty* indica l'assenza di sincronizzazione
- *id* indica il nome di una determinata azione che deve essere eseguita

Ad esempio la dicitura “a with sim(b,c)”, dove “a”, “b”, e “c” sono metodi, significa che il metodo “a” può scattare solamente se i metodi “b” e “c” possono scattare simultaneamente.

Analogo ragionamento viene fatto per gli altri costrutti.

4 Model Checking per reti di Petri algebriche

Il model checking è una tecnica di verifica automatica per i sistemi concorrenti a stati finiti che permette di definire un modello del sistema con una notazione formale, specificare secondo una logica temporale le proprietà che tale sistema deve avere e infine consente di verificare le specifiche definite. Il processo di model checking si basa su tre punti fondamentali:

1. definizione di un modello del sistema
2. specifica di un insieme di proprietà mediante logiche temporali CTL (Computation Tree Logic) e LTL (Linear Temporal Logic)
3. verifica delle specifiche mediante appositi tool

Il primo punto richiede una rappresentazione formale del modello: in particolare si richiede che vengano determinate tutte le caratteristiche che il modello deve avere per ottenere da esso un funzionamento corretto. Il sistema viene quindi definito in termini di stati, dove uno stato rappresenta una descrizione del sistema in un determinato istante di tempo. E' inoltre necessario capire come cambia il sistema al verificarsi di particolari eventi, descrivendo tali evoluzioni in termini di transizioni tra due stati. Prima di verificare il modello del sistema definito, è necessario capire quali proprietà esso deve soddisfare. Quest'ultime vengono espresse secondo una logica che, solitamente, è quella temporale: esse permettono, infatti, di esprimere proprietà relative all'evoluzione del sistema nel tempo.

Le proprietà di maggiore interesse in questo tipo di verifica sono le seguenti:

- **assenza di deadlock:** afferma che il sistema non si troverà in una condizione in cui non è possibile alcun progresso (spesso si pensa ad una proprietà di assenza di deadlock come ad una proprietà di safety, dove si descrivono esplicitamente gli stati di deadlock e si prova che non possano essere raggiunti)
- **fairness:** afferma che, in certe condizioni, un evento accadrà infinitamente spesso
- **liveness:** afferma che, sotto certe condizioni, un qualche evento prima o poi si verificherà

La parte di verifica, infine, permette di constatare se il modello soddisfa o meno le specifiche considerate. Questa parte di model checking è svolta in automatico da un apposito tool: nel caso in questione, sono stati trovati tre tools che supportano la verifica di proprietà di reti di Petri algebriche.

4.1 LTL e CTL

La logica LTL è un tipo di logica che non considera l'evoluzione del sistema dallo stato iniziale a quello attuale: il futuro viene visto come una sequenza di stati.

Date due proposizioni φ e ψ , la semantica degli operatori è definita nel seguente modo:

- $X\varphi$ (next): φ deve verificarsi nel prossimo stato
- $G\varphi$ (globally): φ deve essere vera per ogni stato futuro
- $F\varphi$ (eventually): φ deve verificarsi in almeno uno stato futuro
- $\psi U \varphi$ (until): ψ deve essere vera almeno finché non si verifica φ
- $\psi R \varphi$ (release): φ deve essere vera finché non diventa vera ψ . Nel caso in cui ψ non si verifichi mai, allora φ deve rimanere vera sempre

Con la logica CTL è possibile, invece, esprimere formule logiche su cammini a partire da un determinato stato iniziale; il nome deriva dal fatto che il modello temporale è schematizzato da una struttura ad albero. Il futuro non è invece determinato, ovvero a partire da uno stato definito come quello di partenza, vi possono essere più percorsi percorribili.

Gli operatori utilizzabili in questo tipo di logica sono i seguenti:

- $X\varphi$ (next): nel prossimo stato deve verificarsi φ
- $G\varphi$ (globally): φ deve essere vera per ogni stato futuro
- $F\varphi$ (finally): φ deve verificarsi in almeno uno stato futuro
- $\psi U \varphi$ (until): ψ deve essere vera almeno finché non si verifica φ
- $\psi W \varphi$ (weak until o unless): φ deve essere vera finché non diventa vera ψ . Se ψ non si verifica, allora φ deve rimanere vera per sempre.

Inoltre esistono due ulteriori operatori che permettono di identificare i cammini dell'albero e che possono essere combinati con gli operatori precedentemente definiti:

- $E\varphi$ (exists): esiste almeno un cammino in cui si verifica φ
- $A\varphi$ (all): in tutti i cammini φ è vera

Come si può notare, la distinzione principale tra le due logiche è proprio il riferimento ai cammini, che nella logica LTL non compare.

4.2 Problemi del model checking

Il principale problema pratico nel model checking è l'esplosione degli stati causata dal fatto che la dimensione di una *struttura di Kripke*¹ è esponenziale nella dimensione della descrizione del sistema. Anche per quanto riguarda sistemi relativamente piccoli, è spesso impossibile computare la loro struttura di Kripke.

Due metodi per gestire tale problema sono:

- la **verifica simbolica**: metodo conservativo dove la struttura di Kripke è rappresentata da strutture dati BDD (Binary Decision Diagram) senza perdite di informazione
- le **tecniche di astrazione**: sfruttano la conoscenza sulla struttura e sulla specifica in modo da modellare solo le caratteristiche rilevanti della struttura di Kripke

Nella verifica simbolica, utilizzata anche dal tool studiato, la relazione di transizione della struttura di Kripke non è costruita esplicitamente, ma viene computata una funzione booleana che rappresenta tale relazione di transizione; analogamente, gli insiemi di stati vengono rappresentati come funzioni booleane.

Poiché in molte situazioni lo spazio richiesto per le funzioni booleane è esponenzialmente più piccolo della rappresentazione esplicita, si ha che la rappresentazione simbolica permette di alleviare e, in alcuni casi evitare, il problema dell'esplosione degli stati. Il componente fondamentale di questo tipo di verifica è il *diagramma binario di decisione* (BDD).

L'idea dell'astrazione si basa invece sul risolvere il problema dell'esplosione degli stati rimpiazzando una struttura complessa con una struttura più piccola che soddisfa le stesse proprietà di interesse.

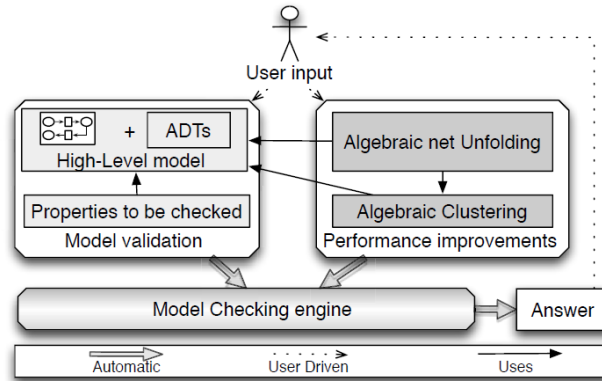
¹Una struttura di Kripke è una struttura che consiste in un insieme di stati, un insieme di transizioni tra stati e una funzione di etichettatura che associa ad ogni stato l'insieme delle proprietà che sono vere in quello stato. Essa rappresenta, perciò, lo spazio degli stati del sistema preso in considerazione

4.3 ALPiNA

ALPiNA (Algebraic Petri Nets Analyzer) è un model-checker simbolico per le reti di Petri algebriche sviluppato dal gruppo SVM dell'Università di Genova, scritto totalmente in Java e scaricabile dal sito: <http://alpina.unige.ch>. L'idea alla base del tool è quella di eseguire un'analisi dei modelli in modo user-friendly utilizzando uno strumento sufficientemente efficiente; inoltre non sono necessarie conoscenze, relative alle tecniche di model checking utilizzate da ALPiNA, per la comprensione dei risultati.

4.3.1 Framework Overview

La modellazione con ALPiNA richiede la trattazione di due aspetti differenti.



La prima concerne la modellazione di un sistema e la specifica delle proprietà che si vogliono verificare. Le specifiche del modello sono costituite dagli ADT algebrici e dalla rete di Petri che permettono, insieme, di descrivere il comportamento del sistema. E' da sottolineare che nel tool in questione, una proprietà si dice verificata se viene valutata *true* in tutti gli stati del modello (tale proprietà prende il nome di *invariante*).

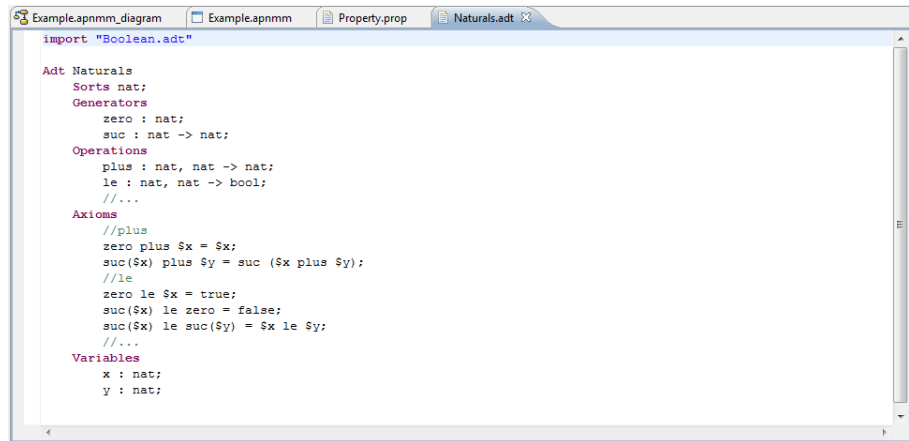
Purtroppo però, per alcuni tipi di problemi, il numero degli stati generato potrebbe assumere dimensioni enormi. Il secondo aspetto da considerare riguarda, infatti, questa problematica: in particolare il supporto del tool all'utilizzo di cluster di specifiche algebriche e di *unfolding*, può aumentare le performance del model checking.

4.3.2 Modellazione di una rete

Il tool è dotato di un editor grafico e di un editor di testo: per prima cosa, per poter modellare un sistema, è necessario definire gli ADT in gioco.

Il codice seguente mostra uno degli ADT forniti dal tool: si dichiara il nome dell'ADT e il tipo da associargli. Vengono poi definiti due *generators*, *zero* e *suc* che, se combinati, permettono di rappresentare tutti i valori dell'algebra; viene

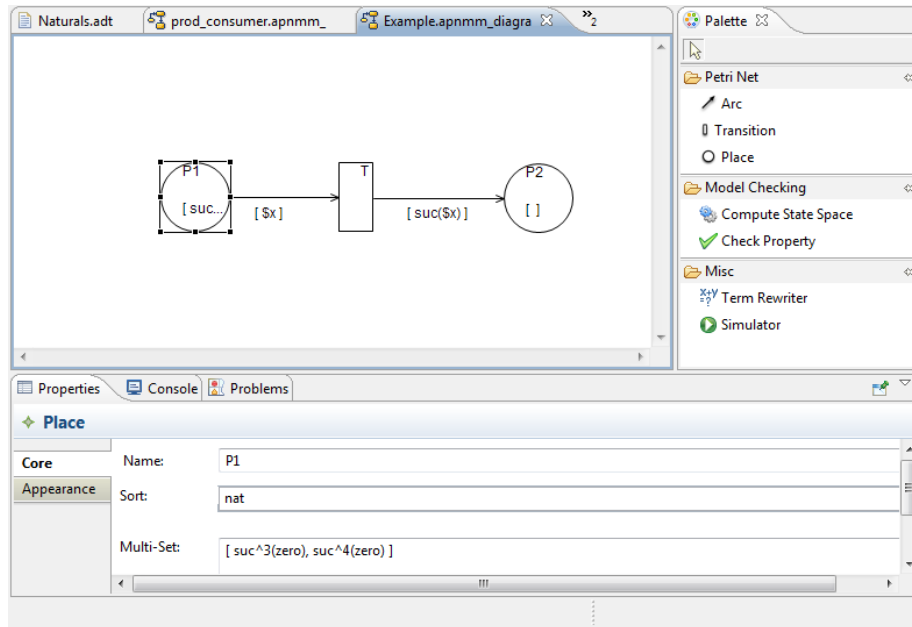
infine definita l'operazione di addizione e di lower-equal, insieme agli assiomi che ne definiscono il comportamento.



```
import "Boolean.adt"

Adt Naturals
  Sorts nat;
  Generators
    zero : nat;
    suc : nat -> nat;
  Operations
    plus : nat, nat -> nat;
    le : nat, nat -> bool;
    //...
  Axioms
    //plus
    zero plus $x = $x;
    suc($x) plus $y = suc ($x plus $y);
    //le
    zero le $x = true;
    suc($x) le zero = false;
    suc($x) le suc($y) = $x le $y;
    //...
  Variables
    x : nat;
    y : nat;
```

Una volta definiti i tipi di dati, è necessario modellare il comportamento del sistema: la presenza di un editor grafico, come mostrato di seguito, facilita notevolmente la creazione della rete.



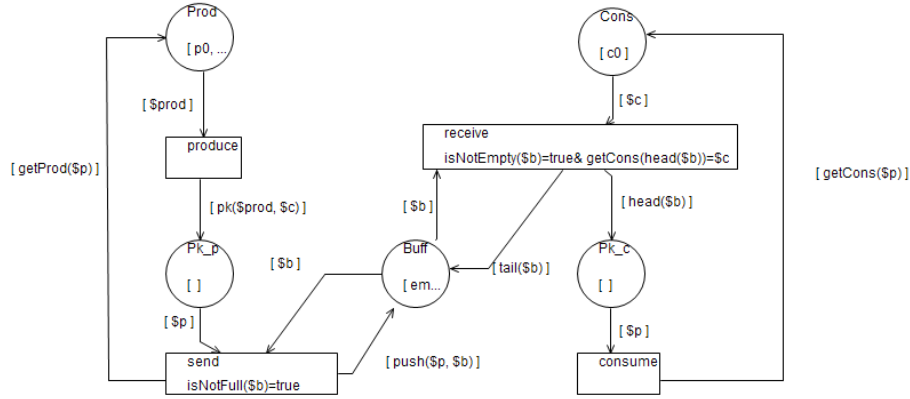
4.3.3 Property checker

Una volta terminato il modello, è possibile eseguire il model checking. ALPiNA mette a disposizione un linguaggio testuale dedicato per la descrizione delle proprietà, ispirato al model checker Helena. Le proprietà in ALPiNA sono delle espressioni booleane che devono valere *true* per ogni stato del modello. Se esiste uno stato raggiungibile in cui la proprietà non è verificata, il tool provvederà a restituire un *counterexample*.

Purtroppo le uniche proprietà verificabili con questo tipo di tool sono due:

- il controllo concernente la cardinalità di un certo multiset
- il controllo concernente il contenuto di un multiset

Di seguito viene mostrato il problema dei produttori-consumatori modellato mediante il tool:



Supponiamo di volere controllare la presenza di tokens all'interno del place Buffer: per esprimere il concetto usando la semantica e sintassi del tool è necessario scrivere:

```
exists($b in Buff)
```

che controlla appunto l'esistenza di una variabile b di sorta Buffer all'interno del place indicato.

Un modo più conciso ed elegante per esprimere tale condizione è nel modo seguente:

```
card(Buff)=1
```

Supponendo ora di voler esprimere la condizione che controlla se il Buffer risulterà essere mai pieno, possiamo scrivere:

```
forall($b in Buff : isNotFull($b)=true)
```

Nel caso in cui la proprietà da validare sia verificata, il tool provvederà a restituire come output dell'analisi il seguente messaggio:

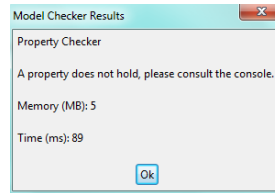
```

AIPiNA Model Checker Engine; [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (14/feb/2012 14.52.38)
*****
Check the properties...
Check property : [Exists(x in Buff:TRUE)]
Property holds : OK
*****
Property Check is finished.
Total time                : 232 ms
State Space fully generated : 0
#Decision Diagrams        : 3.019
Memory (KB)               : 6.993
Check Complete: The property holds.

```

dove vengono fornite anche informazioni relative al tempo impiegato per la verifica della proprietà e alla memoria occupata.

In caso contrario, il risultato sarà dato da un messaggio d'errore seguito da un *counterexample* così espresso:



```

AlPiNA Model Checker Engine: [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (14/feb/2012 14.52.38)
Check the properties...
Check property : [ForAll(x in Buff:((isNotFull(x)) EQUALS (true))))]
Property does not hold ! Here a counter-example :
  < Buff : push(pk(p(p0), c0), push(pk(p(p0), c0), push(pk(p(p0), c0), push(pk(p(p0), c0), empty))))>
*****
Property Check is finished.
Total time           : 89 ms
State Space fully generated : 0
#Decision Diagrams    : 3.284
Memory (KB)          : 5.125
Check Complete: A property does not hold, please see the logs for more details.

```

Data la mancanza di supporto di LTL e CTL all'interno del tool, mediante gli unici operatori disponibili si è cercato di valutare alcune proprietà quali boundedness, assenza di deadlock e liveness.

Esse sono state formulate nel modo seguente:

```

import "prod_consumer.apnmm"
import "Boolean.adt"
import "Buffer.adt"
import "Naturals.adt"
import "Packet.adt"
import "Consumers.adt"
import "Producers.adt"

```

Expressions

```

//Assenza di deadlock
packet_p : card( $ pk in Pk_p)>0 ;
packet_c : card( $ pk in Pk_c)>0 ;
prod0 : card( $p in Prod)>0;
cons0 : card( $c in Cons)>0;
deadlock_freedom : ( @prod0 | @packet_p ) & ( @cons0 | @packet_c ) ;

//Boundedness
b_packet_p : card (Pk_p)=<2;
b_packet_c : card (Pk_c)=<2;
b_prod : card(Prod)=<2;
b_cons : card(Cons)=<2;
b_buffer : card( Buff)=<2;
boundedness : @b_packet_c & @b_packet_p & @b_cons & @b_prod & @b_buffer ;

//Deadlock
dead: card( $p in Prod) + card( $c in Cons) = 0;

```

Check

```

@deadlock_freedom ;
@boundedness ;
@dead ;

```

Variables

```

b : buffer ;
p : producer ;
c : consumer ;
pk : packet ;

```

4.3.4 Clustering algebrico

La concorrenza e il non-determinismo rappresentano le maggiori cause che portano ad un'esplosione degli stati. A questo scopo viene introdotto il concetto di *clustering algebrico*: lo scopo è quello di calcolare lo spazio degli stati di ogni componente del modello in modo indipendente l'uno dall'altro e, successivamente, combinare gli stati così generati.

Per utilizzare il clustering, è necessario assegnare ad ogni termine e ad ogni place del modello un determinato cluster: i criteri di scelta del cluster da attribuire, sono strettamente correlati alla struttura del modello. Generalmente i risultati migliori si ottengono inserendo in clusters differenti gli elementi che non condividono risorse tra di loro.

4.3.5 Algebraic Net Unfolding

Un'altra funzionalità messa a disposizione dal tool per il miglioramento delle performance è l'*algebraic net unfolding*.

Alcuni tipi di dato come i naturali o le liste hanno un dominio infinito; altri invece, come i valori booleani, sono finiti (il loro dominio è infatti costituito dai soli valori **true** e **false**). L'algebraic unfolding consente di numerare i valori degli ADT prima di calcolare lo spazio degli stati. L'aspetto innovativo sta nel fatto che l'utente può decidere se utilizzare o meno l'unfolding e, in caso affermativo, decidere il limite (*bound*) numerico entro il quale rimanere.

4.3.6 Caso di studio: il problema della georeferenziazione modellato con AIPiNA

Nel problema della georeferenziazione introdotto nel capitolo precedente, si possono delineare due entità principali:

- i segnali satellitari
- i GPS

Essi sono stati modellati come ADT nel seguente modo:

```
Adt Satellite
Sorts
    satellite ;
Generators
    sat3 : satellite ;
    sat2 : satellite ;
    sat1 : satellite ;
```

```

import "nat.adt"
import "Distance.adt"
import "boolean.adt"

Adt Gps
Sorts gps;
Generators
    gps0 : gps;
    p : gps -> gps;
    gps-pk : gps, distance -> gps;
Operations
    number_of_gps : nat;
    getId : gps -> gps;
    getDist : gps -> distance;
    getFarest : bool -> gps;
    getNearest : bool -> gps;
Axioms
    number_of_gps = (zero);
    getId(gps-pk($gps-id, $distance)) = $gps-id;
    getDist(gps-pk($gps-id, $distance)) = $distance;
    getFarest(le(getDist($gps-pk), getDist($gps-pk1))) = $gps-pk1;
    getNearest(le(getDist($gps-pk), getDist($gps-pk1))) = $gps-pk;
Variables
    gps-id : gps;
    distance : distance;
    gps-pk : gps;
    gps-pk1 : gps;

```

Tramite le operazioni **getId** e **getDist** è possibile risalire alle informazioni contenute nella tupla rappresentante l'entità Gps. Le operazioni **getFarest** e **getNearest** sono state invece definite, ma non utilizzate all'interno del modello. Questo poiché per semplicità si è supposto un numero massimo di Gps iniziali pari a 4: il confronto tra questi per la scelta dei Gps più vicini alla nostra unità è quindi semplice e non necessita di operazioni particolari. Nel caso in cui, però, il numero dei Gps fosse maggiore, sarebbe agevole utilizzare un approccio differente: si comparano a due a due i segnali Gps in ingresso, copiando in un place temporaneo solamente quelli che risultano avere distanza inferiore e che verranno riesaminati iterativamente in un secondo momento. Da qui l'esigenza dei metodi sopra introdotti.

Come si può notare, nella modellazione del tipo di dato Gps compare anche l'entità "distance": in particolare **gps-pk** rappresenta la tupla contenente l'id del Gps e la distanza dalla nostra unità portatile. L'ADT relativo alla distanza è mostrato di seguito:

```

import "boolean.adt"

Adt Distance
Sorts
    distance;
Generators
    dist0 : distance;
    p : distance -> distance;
Operations
    dec : distance -> distance;
    plus : distance , distance -> distance;
    minus : distance , distance -> distance;
    times : distance , distance -> distance;
    divide : distance , distance -> distance;
    mod : distance , distance -> distance;
    gt : distance , distance -> bool;
    lt : distance , distance -> bool;
    ge : distance , distance -> bool;
    le : distance , distance -> bool;
    max : distance , distance -> distance;
    min : distance , distance -> distance;

Axioms
    p^4(dist0) = dist0; //variare tale parametro per
    impostare un numero maggiore di distanze
    dec(p($dist1)) = $dist1;
    plus (dist0 , $dist1) = $dist1;
    plus (p($dist1) , $dist2) = p (plus ($dist1 , $dist2));

    minus($dist1 , dist0) = $dist1;
    minus(p($dist1) , p($dist2)) = minus($dist1 , $dist2);

    times($dist1 , dist0) = dist0;
    times($dist1 , p($dist2)) = plus($dist1 , times($dist1 , $dist2));

    if ge($dist1 , $dist2) = true & gt($dist2 , dist0) = true
    then divide($dist1 , $dist2) = p(divide(minus($dist1 , $dist2) , $dist2));
    if ge($dist1 , $dist2) = false & gt($dist2 , dist0) = true
    then divide($dist1 , $dist2) = dist0;

    if gt($dist2 , dist0) = true then mod($dist1 , $dist2) =
    minus($dist1 , times($dist2 , divide($dist1 , $dist2)));

    gt(dist0 , $dist1) = false;
    gt(p($dist1) , dist0) = true;
    gt(p($dist2) , p($dist1)) = gt ($dist2 , $dist1);

```

```

lt(dist0 , p($dist1)) = true;
lt($dist1 , dist0) = false;
lt(p($dist2), p($dist1)) = lt($dist2 , $dist1);

ge($dist1 , $dist2) = not(lt($dist1 , $dist2));

le($dist1 , $dist2) = not(gt($dist1 , $dist2));

if ge($dist1 , $dist2) = true then max($dist1 , $dist2) = $dist1;
if ge($dist1 , $dist2) = false then max($dist1 , $dist2) = $dist2;

if le($dist1 , $dist2) = true then max($dist1 , $dist2) = $dist1;
if le($dist1 , $dist2) = false then max($dist1 , $dist2) = $dist2;

```

Variables

```

dist0 : distance;
dist1 : distance;
dist2 : distance;

```

Una volta ricevuti i segnali necessari al calcolo, questi vengono trasformati in un tipo di dato chiamato “Dati” che rappresenta appunto i dati pervenuti. Tale tipo è stato modellato come di seguito

```

import "Satellite.adt"
import "Gps.adt"

```

Adt Dati

```

Sorts dati;

```

Generators

```

dat : satellite -> dati;
dat : gps -> dati;

```

Operations

```

getSat : dati -> satellite;
getGps : dati -> gps;

```

Axioms

```

getSat(dat($sat1)) = $sat1;
//...
getGps(dat($gps1)) = $gps1;
//...

```

Variables

```

sat1 : satellite;
sat2 : satellite;
sat3 : satellite;
gps1 : gps;
gps2 : gps;
gps3 : gps;

```

dove tramite le operazioni di `getGps` e `getSat` è possibile risalire alle entità iniziali e, al termine del calcolo, riportarle allo stato iniziale del sistema.

A causa della notevole complessità, non è stato però modellato il calcolo della trilaterazione.

Di seguito è mostrata l'evoluzione della rete in un ipotetico scenario dato dal susseguirsi delle seguenti transizioni:

- ricezione di un segnale satellitare (`receive_&_prepaire_1`)
- perdita del segnale acquisito (`signal_lost_1`)
- ricezione di un nuovo segnale satellitare (`receive_&_prepaire_1`)
- ricezione di un secondo segnale satellitare (`receive_2_satellites`)
- attesa di un segnale Gps (`prepare_2_satellites`)
- ricezione di un segnale Gps (`insert_gps`)
- controllo del numero di segnali Gps presenti prima di procedere col calcolo (`only_1_gps`)
- calcolo delle coordinate Gps con due satelliti a disposizione ed un solo Gps (`calculus_2_sat`)

In particolare sulle ascisse sono riportate le transizioni che vengono abilitate di volta in volta mentre, sulle ordinate, i places modellati all'interno della rete.

	Initial State	receive_&_prepaire_1
Check_distance	{}	{}
Gps_signal	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))
3_Satellites	sat3	sat3
2_Satellites	sat2	sat2
1_Satellites	sat1	sat1
Send_1	{}	dat(sat1)
Send_2	{}	{}
Send_3	{}	{}
N-Gps	{}	{}
Gps_counter	zero	zero
Third_Gps	{}	{}
Second_Gps	{}	{}
First_Gps	{}	{}

	signal_lost_1	receive_&_prepaire_1
Check_distance	[]	[]
Gps_signal	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))
3_Satellites	sat3	sat3
2_Satellites	sat2	sat2
1_Satellites	sat1	sat1
Send_1	[]	dat(sat1)
Send_2	[]	[]
Send_3	[]	[]
N-Gps	[]	[]
Gps_counter	zero	zero
Third_Gps	[]	[]
Second_Gps	[]	[]
First_Gps	[]	[]

	receive_2_satellites	prepare_2_sat
Check_distance	[]	[]
Gps_signal	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))
3_Satellites	sat3	sat3
2_Satellites	sat1 + sat2	[]
1_Satellites	sat1	sat1
Send_1	dat(sat1)	dat(sat1)
Send_2	[]	dat(sat1) + dat(sat2)
Send_3	[]	[]
N-Gps	[]	[]
Gps_counter	zero	zero
Third_Gps	[]	[]
Second_Gps	[]	[]
First_Gps	[]	[]

	insert_Gps	only_1_gps
Check_distance	[]	[]
Gps_signal	gps_pk(gps0, p(dist0)) + gps_pk(gps1, p^2(dist0)) + gps_pk(gps2, p^3(dist0))	gps_pk(gps0, p(dist0)) + gps_pk(gps1, p^2(dist0)) + gps_pk(gps2, p^3(dist0))
3_Satellites	sat3	sat3
2_Satellites	[]	[]
1_Satellites	sat1	sat1
Send_1	dat(sat1)	dat(sat1)
Send_2	dat(sat1) + dat(sat2)	dat(sat1) + dat(sat2)
Send_3	[]	[]
N-Gps	gps_pk(gps0, dist0)	[]
Gps_counter	suc(zero)	suc(zero)
Third_Gps	[]	[]
Second_Gps	[]	[]
First_Gps	[]	gps_pk(gps0, dist0)

	calculus_2_sat
Check_distance	[]
Gps_signal	gps_pk(gps0, dist0) + gps_pk(gps1, p(dist0)) + gps_pk(gps2, p^2(dist0)) + gps_pk(gps3, p^3(dist0))
3_Satellites	sat3
2_Satellites	sat2
1_Satellites	sat1
Send_1	dat(sat1)
Send_2	[]
Send_3	[]
N-Gps	[]
Gps_counter	zero
Third_Gps	[]
Second_Gps	[]
First_Gps	[]

Per quanto riguarda infine la verifica di proprietà LTL, a causa della presenza di numerosi places e transizioni, e della complessità del modello, AIPiNA presenta delle problematiche: in particolare la generazione dello spazio degli stati richiede un tempo elevato e, molto spesso, porta al crash del tool stesso. Sono state perciò effettuate delle analisi semplificando notevolmente il modello e riducendo gli scenari possibili. Si è provveduto quindi a testare le parti relative ai Gps e ai satelliti in modo separato, controllando che il comportamento fosse quello desiderato. Uno dei test relativi alla parte dei Gps è mostrato di seguito:

Expressions

```
test1: card ($gps in N-Gps)=0;
test2: $n=suc^1(zero);
test3: card ($gps in First_Gps)=1;
test: @test1 & @test2 & @test3;
```

Check

```
@test;
```

Variables

```
s : satellite;
gps : gps;
n : nat;
```

dove si controlla che ogni volta che il contatore dei Gps risulta essere pari ad 1 e il place **N-Gps** atto a contenere le informazioni relative allo stesso risulta essere pari a 0, **First_Gps** contenga l'effettiva informazione prelevata. Lo stesso analogo ragionamento può essere esteso ad un numero maggiore di Gps.

Per quanto riguarda la parte relativa ai satelliti è stata verificata la seguente proprietà, estesa poi alle casistiche possibili

Expressions

```
test1: card($s in Send_3)=3;
```

```

        test2: card($s in 3_Satellite)=0;
        test: @test & @test2;
Check
        @test;
Variables
        s : satellite;

```

dove si controlla che, giunti ad uno stato di attesa prima di poter procedere col calcolo, siano state rimossi correttamente i tokens rappresentanti i satelliti dai relativi places.

4.4 Ulteriori tools

Due ulteriori tools per l'analisi di reti di Petri algebriche risultano essere:

- MARIA (Modular Reachability Analyzer): basato interamente su Prod (un reachability analyzer per reti di Petri Pr/T)
- Helena

Il primo consente la verifica di sistemi concorrenti ed è descritto da un linguaggio le cui espressioni, tipi di dati e operazioni sono gli stessi del linguaggio Java. Inoltre consente anche la costruzione di strutture e unioni tipiche del linguaggio C, oltre al supporto di alcuni tipi di dati (**unsigned**, **enum**, ecc.) sempre tipici di quest'ultimo.

Le proprietà verificabili con l'utilizzo di questo tool sono due:

- proprietà di safety
- proprietà di liveness

La proprietà di safety può essere espressa mediante le parole chiave:

- **deadlock**
- **reject**

Durante l'esplorazione dello spazio degli stati, MARIA indicherà tutte le violazioni di condizione di safety e terminerà l'analisi del sistema nel caso in cui incontri una violazione "fatale".

La sintassi per esprimere tale verifica è la seguente:

```
deadlock fatal;
```

la quale causa l'interruzione dell'analisi nel caso in cui si arrivi ad uno stato di deadlock.

Un altro modo per esprimere la condizione di deadlock, è la seguente:

```
reject place Buf equals empty && fatal;
```

la quale permette di interrompere l'esplorazione dello spazio degli stati nel caso in cui si arrivi ad una condizione in cui il Buffer sia vuoto.

La proprietà di liveness viene invece espressa mediante il comando **eval** seguito dalle formule LTL secondo la seguente semantica:

- **<> formula** indica l'operatore eventually
- **[] formula** indica l'operatore globally
- **() formula** indica l'operatore next
- **formula UNTIL formula** indica l'operatore until

- `formula RELEASE formula` indica l'operatore release

Il secondo tool, Helena, è un High Level Net Analyzer: scritto interamente in Ada, consente la descrizione di reti di Petri High-Level e mette a disposizione un linguaggio di specifica per la verifica di proprietà on-the fly. Purtroppo le uniche proprietà verificabili sono:

- assenza di deadlock (anche qui, come in MARIA, tale proprietà è verificabile mediante l'uso delle keyword `reject` e `deadlock`)
- controllo degli stati (con semantica analoga a quella utilizzata in ALPiNA, mediante l'utilizzo delle keyword `Card` e `forall`)

5 Conclusioni

Le reti di Petri algebriche consentono di superare i limiti imposti dalle reti di Petri standard. Attraverso questo formalismo, infatti, si ha una maggiore espressività ed una maggiore flessibilità nella rappresentazione di sistemi complessi.

D'altra parte la scarsa presenza di tools per la loro modellazione e la limitata valutazione di questi formalismi, rende il loro utilizzo più problematico rispetto alle reti di Petri standard.

Gli studi e le estensioni di queste reti, oltre ad un improvement dei tools fin qui studiati, è però elevato: l'utilizzo di Algebraic Abstract Data Type dinamici, l'introduzione del concetto di tempo all'interno delle reti, nonché l'introduzione delle reti CO-OPN nei sistemi multi-agenti, dimostrano l'importanza e le potenzialità che questi formalismi hanno introdotto.

Riferimenti bibliografici

- [1] DIDIER BUCHS, STEVE HOSTETTLER: *Managing Complexity in Model Checking with Decision Diagrams for Algebraic Petri Net*.
- [2] KARSTEN SCHMIDT: *Parameterized Reachability Trees for Algebraic Petri Nets*.
- [3] JEAN BAILLIE: *An Introduction to the Algebraic Specification of Abstract Data Types*.
- [4] KARSTEN SCHMIDT: *On the Computation of Place Invariants for Algebraic Petri Nets*.
- [5] EKKART KINDLER, HAGEN VOLZER: *Flexibility in Algebraic Nets*.
- [6] KURT JENSEN: *High-level Petri Nets Model-based system development*.
- [7] PAOLO ROMANO: *Studio basato su reti di Petri della semantica formale di modelli per applicazioni Web*.
- [8] BERND BAUMGARTEN: *High Level Petri Nets*.
- [9] LUIGI FILANNINO: *Reti di petri: classi e principali estensioni*.
- [10] ALEXIS MARECHAL: *Properties specification language for algebraic Petri nets*.
- [11] MICHEL DIAZ: *Petri Nets: Fundamental models, Verification and Applications*.
- [12] MARKO MÄKELÄ: *Introduction to MARIA and High-Level Petri Nets*.
- [13] CRISTIAN DIMITROVICI, UDO HUMMERT AND LAURE PETRUCCI: *Semantics, composition and net properties of algebraic high-level nets*.