Traduzione di modelli di sistemi multi-agente AUML in reti di Petri utilizzando tuProlog e analisi delle proprietà del sistema tramite Maude

Marco Alberti
Seconda Facoltà di Ingegneria – Cesena, Italia
Corso di Linguaggi e Modelli Computazionali L-S del prof. Mirko Viroli

1. Introduzione

Modellare correttamente un sistema multi-agente esula dalla semplice modellazione del comportamento dei singoli agenti. Il fatto che il singolo agente abbia o possa comunque essere, in prima approssimazione, considerato un singolo processo computazionale, rende possibile l'applicazione di strutture e tecniche non specificatamente pensate alla descrizione o all'analisi di tali sistemi, che però ormai risultano ben consolidate e comprovate, con minime variazioni. L'evoluzione del ben noto UML nella sua controparte per la modellazione di agenti AUML ne è un esempio.

I modelli scritti in AUML, così come quelli in UML, risultano di facile comprensione ad un individuo umano e sono relativamente semplici da capire e utilizzare in generale, ma sono di difficile lettura ed analisi da parte di programmi automatici. Questa operazione potrebbe essere molto utile per la verifica di certe proprietà del sistema, così da identificarne le eventuali problematiche in fase di analisi del modello. Un sistema di modellazione, al contrario, di espressività meno vicina al linguaggio naturale, ma agilmente analizzabile da una macchina, sono le reti di Petri. Le reti di Petri esprimono intrinsecamente l'interazione tra più processi computazionali e allo stato dell'arte esistono innumerevoli tecnologie che ne permettono l'analisi, in particolar modo di quelle proprietà che condizionerebbero il sistema multi-agente in questione.

Saranno discusse di seguito le tecniche di traduzione da un modello AUML di un sistema multi-agente ad una rappresentazione tramite reti di Petri utilizzando le caratteristiche peculiari del linguaggio *tuProlog* e infine tali reti verranno analizzate con l'ausilio del sistema *Maude* per estrapolare le caratteristiche del sistema multi-agente.

2. Rappresentazione testuale dei modelli AUML

Il primo passo da eseguire è l'analisi dei diagrammi AUML ed identificare un modo per traslare tali diagrammi utilizzando una rappresentazione testuale comprensibile ad una macchina. I diagrammi che interessa maggiormente analizzare sono i diagrammi di sequenza, i diagrammi cioè che descrivono il comportamento delle entità coinvolte nel sistema lungo la loro linea temporale, che caratterizzano lo schema interazionale dell'intero sistema multi-agente.

Un diagramma di sequenza può essere ridotto essenzialmente ad un insieme di entità (gli agenti)

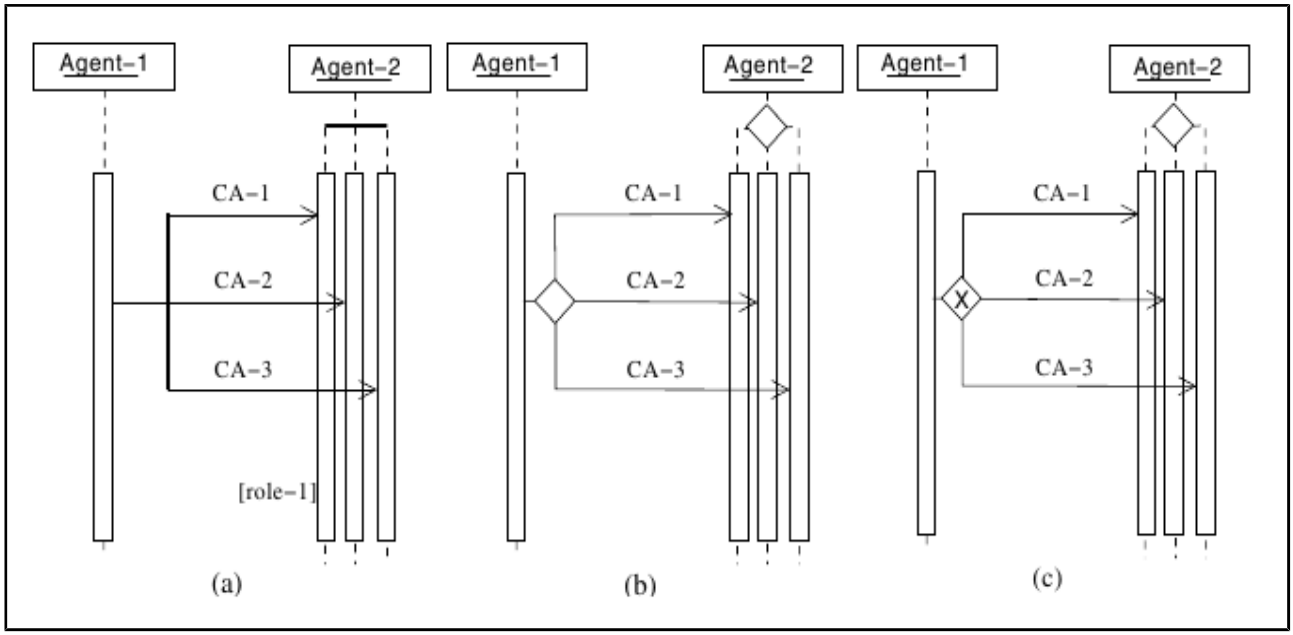
ad ognuna delle quali possono essere associate una o più lifeline (i possibili comportamenti dell'agente) che comunicano le une con le altre tramite un *atto di comunicazione* (abbreviato in CA) di tipo *message passing asincrono*, come prevedono le linee guida sulla interazione tra agenti.

Un CA può quindi essere modellato tramite una tripla

ca (SRC , TGT , MSG)

dove con **SRC** si identifica l'agente che emette l'atto comunicativo, con **TGT** l'agente ricevente l'atto comunicativo e con **MSG** si identifica il particolare atto comunicativo.

Un diagramma di sequenza può inoltre contenere dei simboli che rappresentano 3 particolari diramazioni nel comportamento delle entità:



(a) AND – Agent-1 invia contemporaneamente CA-1, CA-2 e CA-3
(b) OR – Agent-1 può inviare o meno CA-1, inviare o meno CA-2 e inviare o meno CA-3
(c) XOR – Agent-1 invia uno e uno solo tra CA-1, CA-2 e CA-3

L'entità ricevente non deve essere necessariamente la stessa, bensì i CA possono essere diretti a più entità che concorrono alla realizzazione della stessa macro-interazione.

Per una corretta rappresentazione testuale del diagramma di interazione è inoltre utile introdurre un ulteriore operatore → che sarà chiamato *operatore di sequenza*, che indicherà una relazione temporale: ciò che si trova a sinistra dell'operatore, avviene necessariamente prima di ciò che si trova a destra.

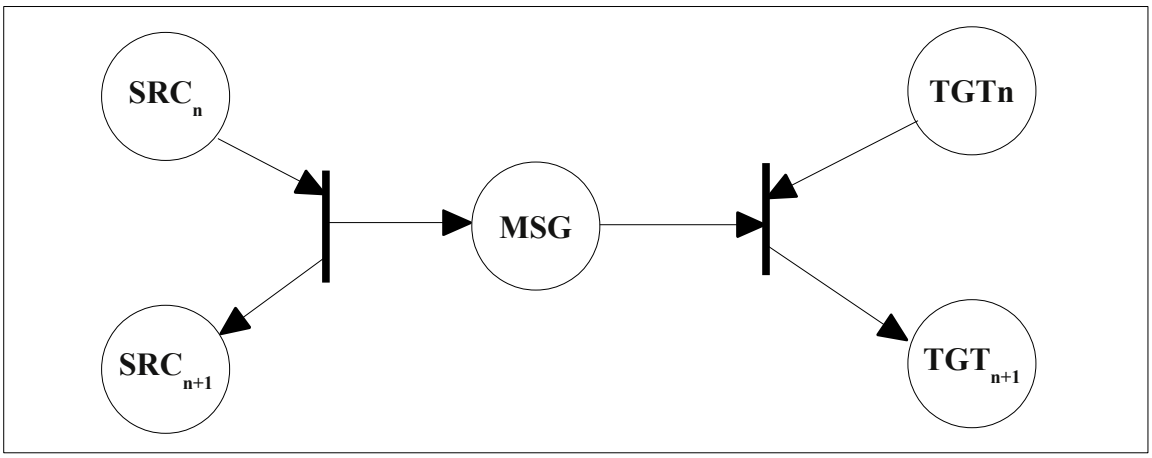
3. Approccio alla traduzione da AUML alla rete di Petri

Il passo successivo è quello di riuscire a definire una corretta mappatura dello schema interazionale scaturito dal diagramma AUML in una rete di Petri. La più logica correlazione risulta quella di identificare i *place* della rete di Petri con gli stati delle entità coinvolte e le *transizioni*

come le azioni che permettono di passare da uno stato all'altro, identificate con i CA del protocollo di interazione. La corretta traduzione del CA identificato da una tripla del tipo

ca (SRC , TGT , MSG)

risulterebbe quindi nella seguente porzione di rete di Petri:



Supponendo di trovarsi nello stato dato dal marking **<SRC$_n$, TGT$_n$>** si transita nello stato **<SRC$_{n+1}$, MSG, TGT$_n$>** che rappresenta l'emissione di **MSG** da parte di **SRC**; dopodiché si transita nello stato **<SRC$_{n+1}$, TGT$_{n+1}$>** che rappresenta la ricezione di **MSG** da parte di **TGT**.

Utilizzando questo metodo è facile tradurre il comportamento definito dall'operatore → e dall'operatore XOR, assegnando opportunamente gli stati delle entità tra le quali avvengono gli scambi di CA; in particolare, nel caso in cui i CA siano legati dall'operatore → si genererà una singola transizione verso lo stato successivo, mentre nel caso in cui i CA siano legati dall'operatore XOR, verranno generate due transizioni verso due possibili stati futuri associati alle due alternative dell'operatore.

Non è altrettanto semplice modellare con le reti di Petri i comportamenti dei due operatori AND e OR in quanto introducono un fattore di incertezza sull'effettivo comportamento dell'entità: nel caso di un AND si può assistere a due comportamenti diversi a seconda dell'argomento dell'operatore che verrà eseguito per primo, mentre nel caso di un OR si possono addirittura verificare casi in cui uno degli argomenti non venga affatto eseguito.

È possibile, tuttavia, esprimere i due operatori di AND e OR utilizzando gli altri due operatori di → e XOR operando delle trasformazioni analoghe alle operazioni di trasformazione in algebra booleana:

X and Y = (X → Y) xor (Y → X)
X or Y = (X → Y) xor (Y → X) xor Y xor X

Una trasformazione alternativa nel caso di un OR sarebbe stata:

X or Y = (X → (Y xor Ø)) xor (Y → (X xor Ø))

Tale trasformazione però introdurrebbe la necessità di definire un operatore Ø che identifichi un'azione vuota.

L'algoritmo di traduzione seguirà quindi la seguente sequenza di passi:

1. eliminare dal modello testuale l'operatore OR applicando la trasformazione di cui sopra
2. eliminare dal modello così ottenuto l'operatore AND usando un processo analogo
3. tradurre il modello finale contenente solo → e XOR in una rete di Petri

Trattandosi della rappresentazione del comportamento di agenti, il modello AUML sottintende una reiterazione ciclica delle azioni descritte; la rete di Petri risultante andrà quindi completata con una serie di transizioni che permettano di tornare dagli stati finali, opportunamente identificati, a quello iniziale.

4. Rappresentazione della rete di Petri in Maude

Per rappresentare una rete di Petri in Maude ci sono innumerevoli possibilità, ma una delle più semplici è quella di rappresentare la rete tramite l'insieme delle sue *transizioni*; una *transizione* può essere rappresentata infatti tramite tre campi: un identificativo, i *place* con archi entranti nella *transizione*, i *place* con archi uscenti dalla *transizione*.

```
fmod PETRI-NET is

  pr QID .

  sorts P T MT MP NetState .

  subsort Qid < P .
  subsort P < MP .
  subsort T < MT .

  op nil : -> MP [ctor] .
  op __ : MP MP -> MP [ctor comm assoc id: nil] .
  op [_@_,_] : Qid MP MP -> T [ctor] .
  op nil : -> MT [ctor] .
  op __ : MT MT -> MT [ctor comm assoc id: nil] .
  op <_,_> : MP MT -> NetState .

  op net : -> MT .

endfm
```

In questo modello, una *transizione* è definita tramite un ***Qid*** che ne rappresenta il nome e due set di tipo ***MP*** che rappresentano rispettivamente il set di *place* con archi entranti nella *transizione* e il set di *place* con archi uscenti dalla *transizione*.

L'operatore ***net*** rappresenta la rete di Petri ed è infatti definito di tipo ***MT*** cioè un set di *transizioni*.

L'operatore per la definizione di ***NetState*** è pensato per contenere nel primo parametro l'insieme dei token della rete (*marking*), identificati ognuno tramite un ***Qid*** con il nome dello stato in cui si trovano e nel secondo parametro, l'insieme delle transizioni scattate dallo stato iniziale fino a quel momento.

Dopo aver definito la rete di Petri in questo modo sono necessarie delle *rewriting rules* che permettano alla rete di evolversi correttamente modificando il *marking* nel ***NetState*** coerentemente

con le transizioni che compongono la rete e tenere traccia delle transizioni eseguite.

```
mod PETRI-NET-ENGINE is

  pr PETRI-NET .

  var T : T .
  var TS MT : MT .
  var TName : Qid .
  var MP MP1 MP2 : MP .

  crl [fire] : < MP1 MP , MT > => < MP2 MP , [ TName @ MP1 , MP2 ] MT >
    if [ TName @ MP1 , MP2 ] TS := net /\ new([ TName @ MP1 , MP2 ] , MT) .

  crl [fire] : < MP1 MP , MT > => < MP2 MP , MT >
    if [ TName @ MP1 , MP2 ] TS := net /\ new([ TName @ MP1 , MP2 ] , MT) ==
    false .

  op new(_,_) : T MT -> Bool .
  eq new(T , T MT) = false .
  eq new(T , MT) = true [owise] .

endm
```

Le r*ewriting rules* permettono alla rete di passare da uno stato ad un altro nel caso in cui un sottoinsieme di token del *marking* coincida con le precondizioni di una data *transizione*, in quel caso, il *marking* viene modificato sostituendo tali token con i rispettivi nelle post-condizioni e qualora la *transizione* in questione non fosse mai stata ancora eseguita, viene aggiunta all'elenco contenuto nel secondo parametro.

Perché una rete di Petri descrivente un protocollo di interazione tra agenti sia *sound*, occorre che rispetti due proprietà: *boundedness* e *liveness*.

Per quanto riguarda l'analisi della *boundedness* è interessante distinguere inoltre due casi: la semplice *k-boundedness* e la più stringente *1-boundedness*.

- 1-boundedness – Perché una rete di Petri sia *1-bounded* (*safe*) occorre che in ogni possibile stato, raggiungibile a partire da quello di partenza, ci sia al massimo 1 token in ogni *place* della rete. Per verificare tale proprietà è stato definito un modulo Maude che definisce la proprietà ***1-bounded*** nel modo seguente:

```
mod 1-BOUNDEDNESS is

  pr PETRI-NET .
  pr SATISFACTION .
  pr MODEL-CHECKER .
  pr LTL-SIMPLIFIER .

  subsort NetState < State .

  var MT : MT .
  var MP : MP .
  var P : P .

  op 1-bounded : -> Prop .
  eq < MP , MT > |= 1-bounded = check(MP) .
```

```
    op check(_) : MP -> Bool .
    eq check(P P MP) = false .
    eq check(MP) = true [owise] .

endm
```

La funzione così definita verrà poi verificata per ogni possibile stato raggiungibile dalla rete tramite l'utilizzo delle funzioni ***reduce*** e ***modelCheck*** di Maude in questo modo:

reduce modelCheck(< *InitState* , nil > , [] 1-bounded)

- k-boundedness – La verifica della *k-boundedness*, cioè della presenza in ogni possibile stato di al massimo **k** token in ogni place, è differente: non conoscendo a priori il numero **k** non è possibile definire una proprietà simile al caso precedente. La proprietà può però essere inferita seguendo tale ragionamento:

  la *k-boundedness*, se rispettata, garantisce che il numero degli stati raggiungibili dalla rete a partire dal marking iniziale $\mathbf{M_0}$, sia finito; dunque, ipotizzando per assurdo che sia possibile, a partire dal marking iniziale $\mathbf{M_0}$, caratterizzato dall'insieme di token $\mathbf{T_0}$, raggiungere un marking $\mathbf{M_1}$ caratterizzato dall'insieme di token $\mathbf{T_1}$, tale che $\mathbf{T_0 \subset T_1}$, allora $\mathbf{T_1 = T_0 \cup T_x}$ con $\mathbf{T_x \neq \emptyset}$. Reiterando l'operazione è possibile produrre un numero infinito di marking, cioè un numero infinito di possibili stati della rete, condizione invalidante la proprietà di *k-boundedness*.

  Utilizzando la funzione ***search*** di Maude, è possibile quindi verificare che una rete sia *k-bounded* (per qualsiasi **k**) in questo modo:

  search [2] < *InitState* , nil > =>+ < *InitState* Token:MP , Transitions:MT >

  Se la ricerca produrrà un assegnamento per la variabile ***Token*** diverso da ***nil*** allora la rete di Petri non sarà *k-bounded*, con all'interno della variabile ***Transitions*** le transizioni utilizzate per produrre quel particolare stato. La limitazione della ricerca alle prime 2 soluzioni, garantisce che, nel caso in cui la rete non fosse *k-bounded*, cioè si avesse una serie infinita di soluzioni, la ricerca abbia termine, una volta trovata almeno una soluzione invalidante la proprietà, oltre all'assegnamento vuoto.

- Liveness – In una rete di Petri, una transizione **T** è detta *live* se, per ogni marking **M** raggiungibile a partire dallo stato iniziale, è possibile raggiungere un altro marking **M'**, attraverso una serie di transizioni tra cui **T**. Se nella rete di Petri tutte le transizioni sono *live*, allora anche la rete è *live*.
  Conseguenza di questo è che in una rete *live*, per ogni marking **M** raggiungibile a partire dallo stato iniziale, è possibile raggiungere di nuovo il marking **M**, utilizzando almeno 1 volta ciascuna transizione della rete; inoltre, se questo risulta vero per un dato marking, allora è vero per tutti gli altri possibili marking.
  Sapendo questo e che lo stato iniziale della rete è, in effetti, un marking, la proprietà di *liveness* può essere verificata eseguendo la seguente ricerca tramite la funzione ***search*** di Maude:

  search [1] < *InitState* , nil > =>+ < *InitState* , *Net* >

In questo caso la variabile ***Net*** rappresenta l'elenco di tutte le transizioni della rete, che in questa particolare rappresentazione, coincide con la rete stessa.

In realtà la proprietà di *liveness* è sempre garantita nel caso in cui la rete di Petri sia stata creata (in modo corretto) a partire da un diagramma di sequenza AUML, per cui la verifica di tale proprietà è opzionale.

Un'altra proprietà di cui potrebbe interessare la verifica è quella di *safety*, cioè il non raggiungimento di particolari stati indesiderati. Tale proprietà risulta però legata fortemente al protocollo di interazione specifico e impossibile da codificare in un algoritmo di analisi generico; inoltre, nel caso in cui la rete di Petri generasse degli stati indesiderati, qualora fosse soddisfatta comunque la *k-boundedness*, il protocollo di interazione risulterebbe analiticamente corretto e in grado di essere realizzato. Il fatto che il protocollo non rispecchi le aspettative del progettista significherebbe semplicemente un diagramma AUML di partenza inidoneo.

5. Il traduttore in tuProlog

Una volta delineati il punto di partenza e quello di arrivo è possibile procedere alla realizzazione dei passi intermedi. Avendo a disposizione una rappresentazione testuale del diagramma AUML occorre costruire un parser che identifichi la struttura del protocollo ed esegua le trasformazioni necessarie. Verrà tralasciata per brevità la presentazione delle clausole di supporto che non concorrono alla realizzazione vera e propria dell'algoritmo di trasformazione.

Ad ogni frase di un linguaggio è associato un albero che la rappresenta; la rappresentazione testuale di un diagramma AUML non fa eccezione, per cui occorre stabilire una serie di operatori cosicché sia possibile identificare tale albero. In questo caso gli operatori sono i seguenti:

```
:- op(100, fx, protocol).
:- op(95, xfx, '->').
:- op(90, xfx, xor).
:- op(90, xfx, and).
:- op(90, xfx, or).
```

L'operatore ***protocol*** serve solamente ad identificare la porzione di testo che identifica il protocollo di interazione, mentre gli altri sono gli operatori veri e propri utilizzati all'interno delle frasi. L'*operatore di sequenza* ha priorità più alta rispetto agli operatori logici, l'idea infatti è quella di utilizzare gli operatori logici per combinare varie sequenze di CA, rappresentati tramite le triple **ca (SRC , TGT , MSG)** esposte in precedenza.

Stabiliti gli operatori del linguaggio è possibile analizzarne le frasi. Le prime due operazioni saranno l'eliminazione da queste frasi degli operatori OR e AND riscrivendoli in termini degli altri due operatori → e XOR utilizzando le seguenti clausole:

```
exchangeOR('->'(A1,B1),'->'(A2,B2)):- exchangeOR(A1,A2), exchangeOR(B1,B2).
exchangeOR(xor(A1,B1),xor(A2,B2)):- exchangeOR(A1,A2), exchangeOR(B1,B2).
exchangeOR(and(A1,B1),and(A2,B2)):- exchangeOR(A1,A2), exchangeOR(B1,B2).
exchangeOR(or(A1,B1),xor('->'(A2,B2),xor('->'(B2,A2),xor(A2,B2)))):-
      exchangeOR(A1,A2), exchangeOR(B1,B2).
exchangeOR(CA,CA).
```

```
exchangeAND('->'(A1,B1),'->'(A2,B2)):- exchangeAND(A1,A2), exchangeAND(B1,B2).
exchangeAND(xor(A1,B1),xor(A2,B2)):- exchangeAND(A1,A2), exchangeAND(B1,B2).
exchangeAND(and(A1,B1),xor('->'(A2,B2),'->'(B2,A2)):-
      exchangeAND(A1,A2), exchangeAND(B1,B2).
exchangeAND(CA,CA).
```

Questi due gruppi di clausole navigano l'albero della frase e tramite un'analisi ricorsiva dei sotto-alberi riscrivono le sequenze che coinvolgono gli operatori indesiderati di OR e AND utilizzando le relazioni descritte in precedenza.

Dopo aver eseguito queste operazioni, la frase viene sottoposta ad un'ulteriore navigazione per identificare tutte le entità (agenti) coinvolti nell'interazione (la clausola ***noPairs*** serve ad eliminare le ripetizioni della medesima entità):

```
actors(P,AA):-
      findall(state(SRC,0),search(P,ca(SRC,TGT,MSG)),L1),
      findall(state(TGT,0),search(P,ca(SRC,TGT,MSG)),L2),
      append(L1,L2,LL), noPairs(LL,AA).

search(ca(SRC,TGT,MSG)):- clause(protocol({P}), true),
      search(P,ca(TYPE,SRC,TGT,MSG)).
search('->'(A,_),ca(SRC,TGT,MSG)):- search(A,ca(SRC,TGT,MSG)).
search('->'(_,B),ca(SRC,TGT,MSG)):- search(B,ca(SRC,TGT,MSG)).
search(xor(A,_),ca(SRC,TGT,MSG)):- search(A,ca(SRC,TGT,MSG)).
search(xor(_,B),ca(SRC,TGT,MSG)):- search(B,ca(SRC,TGT,MSG)).
search(CA,CA).
```

Per ognuna delle entità individuate verrà creato un fatto del tipo **state (A , N)** che verrà memorizzato in un elenco, che verrà usato durante la creazione della rete di Petri, per identificare i *place* associati all'entità **A** utilizzando l'indice **N** ed associare alle transizioni i corretti *place* con archi in ingresso o uscita.

Durante le operazioni di traduzione verranno utilizzate, per rappresentare le transizioni della rete, tre triple:

transition (Name , [state(A,X)] , [state(A,Y) , Msg])
transition (Name , [state(A,X) , Msg] , [state(A,Y)])
transition (Name , [state(A,Y)] , [state(A,X)])

Tutte e tre le triple hanno un campo ***Name*** contenente l'identificativo della transizione e due liste, rispettivamente per i *place* con archi in ingresso e per i *place* con archi in uscita.
La prima tripla rappresenta una transizione associata all'emissione di un CA da parte dell'entità ***A*** che transita dal proprio *place* ***X*** al proprio *place* ***Y*** generando un token nel *place* ***Msg***, che rappresenta il CA in questione.
La seconda tripla rappresenta una transizione associata alla ricezione di un CA da parte dell'entità ***A***, che transitando dal proprio *place* ***X*** al proprio *place* ***Y***, consuma il token nel *place* ***Msg***.
La terza tripla invece rappresenta le transizioni utilizzate per completare la rete così da generare un comportamento ciclico, permettendo ad uno stato finale dell'entità ***A***, qui identificato dal *place* con indice ***Y***, di ritornare allo stato iniziale, identificato dal *place* con indice ***X***.

La generazione dell'elenco di transizioni che rappresenteranno la rete avviene utilizzando le

seguenti clausole che, per motivi di chiarezza, verranno analizzate una per una.

```
netGen(Front,Prot,Net):- netGen(Front,Prot,Net,[],Front,_,_).
      …
      …
      …
netGen(Front,[],Net,Net,Finals,Front,Finals).
```

Occorre spiegare il significato delle variabili che sono state introdotte e che saranno identificate nelle prossime clausole nel modo seguente: ***Front*** contiene l'elenco dei *place* che costituiscono la frontiera del sotto-albero in esame, ***Prot*** contiene i CA del protocollo che sono rimasti da tradurre, ***Net*** è la variabile che verrà unificata con il risultato finale, ***TS*** contiene l'elenco delle transizioni generate fino al momento presente che alla fine verrà unificato con ***Net***, ***Assigned*** contiene per ogni entità l'indice dello stato raggiunto, ***Next*** e ***Finals*** vengono unificati al termine della traduzione di un sotto-albero rispettivamente con la frontiera e l'indice di stato raggiunti nel sotto-albero.

```
netGen(Front,'->'(xor(A,B),C),Net,TS,Assigned,Next,Finals):- !,
      netGen(Front,xor('->'(A,C),'->'(B,C)),Net,TS,Assigned,Next,Finals).
```

Questa clausola serve a trattare il caso spiacevole in cui il protocollo presenti la situazione in cui il sotto-albero ***C*** debba essere eseguito sia che si abbia intrapreso il cammino denotato da ***A***, sia che si abbia intrapreso quello denotato da ***B***; ciò implicherebbe che due sotto-alberi confluiscano nello stesso cammino. Questa situazione non è intrinsecamente dannosa, ma risulta poco gestibile a livello algoritmico, si preferisce quindi applicare una trasformazione che generi due rami distinti. Il ***cut*** serve a tagliare eventuali unificazioni ulteriori mentre le variabili ausiliare vengono inoltrate senza subire cambiamenti.

```
netGen(Front,'->'(A,B),Net,TS,Assigned,Next,Finals):-
      netGen(Front,A,FirstHalf,TS,Assigned,NewFront,NewAssign),
      netGen(NewFront,B,Net,FirstHalf,NewAssign,Next,Finals).
```

Questa clausola gestisce il caso in cui il protocollo presenti una sequenza di due sotto-alberi (escludendo che il sotto-albero ***A*** sia uno XOR, avendolo gestito tramite la clausola precedente). I due sotto-alberi vengono gestiti separatamente; i valori attuali di ***Front*** e ***Assigned*** vengono utilizzati per elaborare il sotto-albero ***A***, mentre per elaborare il sotto-albero ***B*** verranno utilizzati i valori ***NewFront*** e ***NewAssign*** raggiunti dopo l'elaborazione di ***A***, i valori raggiunti dopo l'elaborazione di ***B*** saranno quelli che verranno unificati con le due variabili ***Next*** e ***Finals*** nella testa della clausola.

```
netGen(Front,xor(A,B),Net,TS,Assigned,Next,Finals):-
      netGen(Front,A,FirstHalf,TS,Assigned,NewFront,NewAssign),
      netGen(Front,B,Net,FirstHalf,NewAssign,Next,Finals).
```

Questa clausola gestisce il caso in cui il protocollo presenti una ramificazione in due sotto-alberi. Anche in questo caso i due sotto-alberi vengono gestiti separatamente e l'elaborazione del sotto-albero ***A*** è identica al caso precedente, mentre, per l'elaborazione del sotto-albero ***B***, anziché la nuova frontiera generata dall'elaborazione di ***A***, sarà usata la stessa frontiera utilizzata per ***A***. Per l'assegnamento degli indici ai *place* generati invece sarà utilizzata la variabile NewAssign generata dall'elaborazione di ***A***, poiché, nell'eventualità che il primo sotto-albero abbia coinvolto alcune entità presenti anche in ***B***, i *place* che sono stati generati avranno consumato degli indici che non

potranno più essere utilizzati durante l'elaborazione di ***B***. Anche qui i valori raggiunti dopo l'elaborazione di ***B*** saranno unificati con ***Next*** e ***Finals*** nella testa della clausola.

```
netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals):-
      atom_chars(MsgName,[SRC,"_",MSG,"_",TGT]), member(state(TGT,Y0),Front),
      member(transition(_,[state(TGT,Y0),MsgName],[state(TGT,OldY)]),TS), !,
      member(state(SRC,X0),Front), inc(Assigned,SRC,NewAssign),
      member(state(SRC,X1),NewAssign), setState(Front,SRC,X1,TempFront),
      setState(TempFront,TGT,OldY,NewFront),
      atom_chars(TName,[SRC,X0,"_",out,"_",MSG,"_",TGT,Y0]),
      netGen(NewFront,[],Net,
            [transition(TName,[state(SRC,X0)],[state(SRC,X1),MsgName])|TS],
      NewAssign,Next,Finals).
netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals):-
      atom_chars(MsgName,[SRC,"_",MSG,"_",TGT]), member(state(SRC,X0),Front),
      member(transition(_,[state(SRC,X0)],[state(SRC,OldX),MsgName]),TS), !,
      member(state(TGT,Y0),Front), inc(Assigned,TGT,NewAssign),
      member(state(TGT,Y1),NewAssign), setState(Front,SRC,OldX,TempFront),
      setState(TempFront,TGT,Y1,NewFront),
      atom_chars(TName,[TGT,Y0,"_",in,"_",MSG,"_",SRC,X0]),
      netGen(NewFront,[],Net,
            [transition(TName,[state(TGT,Y0),MsgName],[state(TGT,Y1)])|TS],
      NewAssign,Next,Finals).
netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals):-
      atom_chars(MsgName,[SRC,"_",MSG,"_",TGT]), member(state(SRC,X0),Front),
      member(state(TGT,Y0),Front), inc(Assigned,SRC,TempAssign),
      inc(TempAssign,TGT,NewAssign),
      member(state(SRC,X1),NewAssign), member(state(TGT,Y1),NewAssign),
      setState(Front,SRC,X1,TempFront), setState(TempFront,TGT,Y1,NewFront),
      atom_chars(TName1,[SRC,X0,"_",out,"_",MSG,"_",TGT,Y0]),
      atom_chars(TName2,[TGT,Y0,"_",in,"_",MSG,"_",SRC,X0]),
      netGen(NewFront,[],Net,[
            transition(TName2,[state(TGT,Y0),MsgName],[state(TGT,Y1)]),
            transition(TName1,[state(SRC,X0)],[state(SRC,X1),MsgName])|TS
      ],NewAssign,Next,Finals).
```

Queste sono le clausole che generano le transizioni arrivati alla fine di un sotto-albero, sono quindi le clausole che modificano le variabili ***Front*** e ***Assigned*** generando la nuova frontiera e aggiornando gli indici da utilizzare per generare i *place* associati alle varie entità.

È possibile, in teoria, non utilizzare le prime due clausole. La rete di Petri generata sarebbe comunque corretta e in grado di essere analizzata tramite gli strumenti sopra presentati, si assisterebbe però al proliferare di sotto-alberi inutili che andrebbero ad accrescere enormemente la pesantezza computazionale della rete generata: come side-effect della trasformazione degli operatori di OR e AND si ha, infatti, la duplicazione di parti della rete in più sotto-alberi.

Le prime due clausole servono dunque ad identificare i casi in cui si stia cercando di creare un sotto-albero identico ad una parte di rete già creata. In questi due casi, anziché creare una coppia di transizioni che permettono di far transitare le due entità coinvolte in 2 nuovi *place*, viene creata solo una transizione che permette all'entità emittente o ricevente, secondo il caso, di transitare in un nuovo *place*, mentre per l'altra entità viene riutilizzato il sotto-albero precedentemente creato. Questo avviene, anziché incrementando entrambi gli indici dei *place* delle due entità, incrementando solamente quello dell'entità che non può riutilizzare il sotto-albero e assegnando al valore di frontiera dell'altra, quello del *place* nel sotto-albero in questione.

La rete così generata risulta ancora incompleta poiché necessita delle transizioni che le permettono di realizzare un comportamento ciclico. Tali transizioni vengono generate utilizzando le

seguenti clausole:

```
completeNet(OpenNet,ClosedNet):- completeNet(OpenNet,OpenNet,ClosedNet).
completeNet([transition(_,_,[S])|L],OpenNet,ClosedNet):-
      endingState(S,OpenNet), !, closingTransition(S,CT),
      completeNet(L,[CT|OpenNet],ClosedNet).
completeNet([transition(_,_,[S,_])|L],OpenNet,ClosedNet):-
      endingState(S,OpenNet), !, closingTransition(S,CT),
      completeNet(L,[CT|OpenNet],ClosedNet).
completeNet([_|L],OpenNet,ClosedNet):- completeNet(L,OpenNet,ClosedNet).
completeNet([],Net,Net).
closingTransition(state(A,N),transition(TName,[state(A,N)],[state(A,0)])):-
      atom_chars(TName,[A,N,"_",to,"_",A,0]).
```

```
endingState(S,[transition(_,[S|_],_)|L]):- !, fail.
endingState(S,[_|L]):- endingState(S,L).
endingState(_,[]).
```

Se una transizione, ha nelle sue post-condizioni un *place* che non è presente nelle precondizioni di nessun'altra transizione, allora tale *place* è uno stato finale; viene aggiunta quindi alla rete una transizione che permette da questo *place* ti transitare nel *place* con indice **0** (stato iniziale) associato alla medesima entità.

Una volta generata questa rete finale, tutti i dati vengono trasposti in un file Maude pronto per essere lanciato senza nessuna ulteriore modifica. Durante l'operazione vengono generati due ulteriori file contenenti i due passi intermedi.

6. Conclusioni e possibili sviluppi futuri

In campo informatico, specialmente nell'area più filosofica della modellazione, abbiamo sicuramente l'imbarazzo della scelta tra metodologie e tecnologie a disposizione. La scelta dell'abbinamento di AUML e delle reti di Petri nasce principalmente dal vasto impiego che tali tecnologie hanno avuto singolarmente con risultati positivi e dalle caratteristiche complementari che esse possiedono.

Da una parte AUML, non dissimile dal suo predecessore UML, agile da usare e intuitivo, dall'altra, le reti di Petri, dotate di una esatta definizione matematica delle loro semantiche di esecuzione e di una ben sviluppata teoria matematica per quanto riguarda la loro analisi.

Il sistema così composto permette al modellatore/progettista di sfruttare il meglio di entrambe le tecnologie, curandosi minimamente, se non affatto, di ciò che accade sullo sfondo, permettendogli di concentrare i propri sforzi sulle questioni che più gli premono.

I sistemi multi-agente così modellati e analizzati, prescindono inoltre dalle tecnologie utilizzate per la loro effettiva realizzazione; i risultati ottenuti tramite questo sistema sono quindi sempre validi.

Considerato ciò, sarebbe un'aggiunta molto proficua, un sistema di implementazione automatico dei modelli che soddisfino i requisiti richiesti, in grado di realizzare tali modelli su differenti piattaforme, anche molto eterogenee l'una dall'altra, tutte in grado però di fornire un'infrastruttura adatta a supportare un sistema multi-agente.

Con questa funzionalità, il sistema finale realizzerebbe automaticamente tutti i passi necessari

alla costruzione di un sistema multi-agente perfettamente in grado di funzionare senza ulteriori modifiche sostanziali su, teoricamente, qualsiasi piattaforma, richiedendo solamente un singolo modello dell'interazione che si desidera realizzare.

Bibliografia

Modeling Agent Interactions: Agent UML and Multiagent Interaction Protocol Nets
Marco Alberti
Seconda Facoltà di Ingegneria – Cesena, Italia
marco.alberti3@studio.unibo.it

Generating Machine Processable Representations of Textual Representations of AUML
Jean-Luc Koning and Ivan Romero-Hernandez
CoSy-Inpg, 50 rue Laffemas, BP 54, 26902 Valence cedex 9, France
Jean-Luc.Koning@esisar.inpg.fr, Ivan.Romero@imag.fr
http://www-leibniz.imag.fr/CoSy