Implementazione automatica su piattaforma Jason di protocolli di interazione AUML mediante l'ausilio di reti di Petri

Marco Alberti
Seconda Facoltà di Ingegneria – Cesena, Italia
Corso di Progetto di Sistemi Informatici L-S del prof. Andrea Omicini

Abstract. Un modello di un sistema multi-agente descritto in *Agent UML* (AUML) può essere rappresentato attraverso una rete di Petri particolare che prende il nome di *Multiagent Interaction Protocol Net* (MIPNet). La rete di Petri può essere quindi analizzata per identificare le proprietà che garantiscono l'effettiva realizzabilità del protocollo. I place di una MIPNet possono essere suddivisi in sotto-gruppi ognuno dei quali associato ad una delle entità che concorrono alla realizzazione del protocollo e da questi sotto-gruppi è possibile estrapolare il comportamento al quale le singole entità (gli agenti) devono attenersi al fine di realizzare correttamente l'interazione. L'implementazione effettiva del comportamento di tali agenti viene realizzata poi tramite la piattaforma Jason.

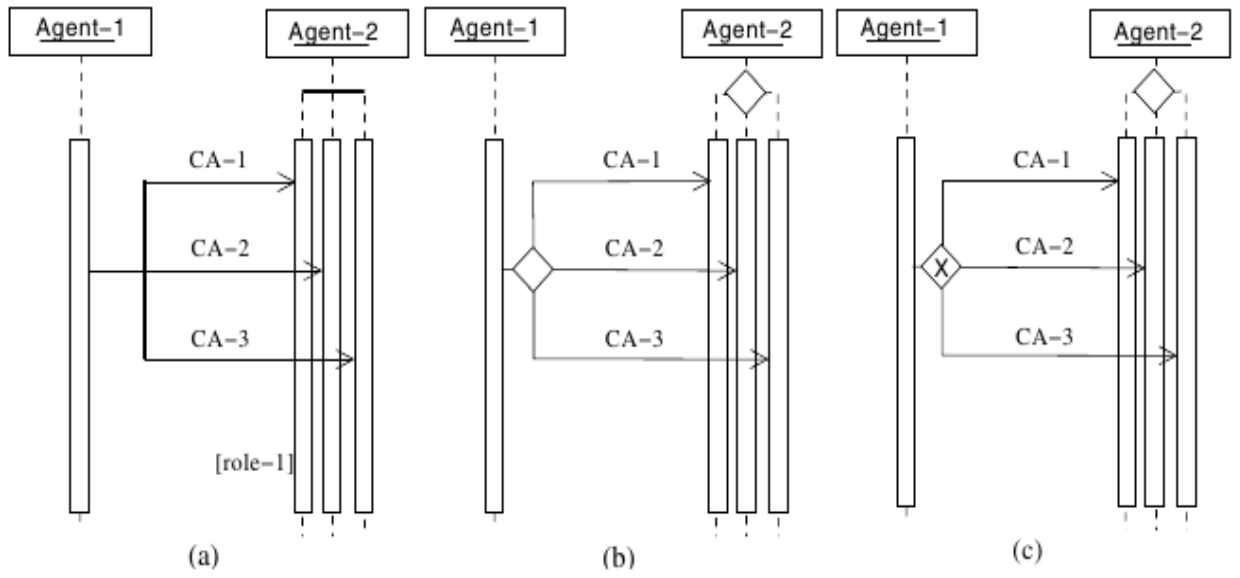
1. Richiami su AUML e le MIPNets

Esprimere un protocollo di interazione in AUML significa fondamentalmente utilizzare una estensione dei classici diagrammi di sequenza UML. Questi sono dotati di particolari costrutti atti a rappresentare con maggiore efficacia il comportamento di un agente rispetto a quello di un oggetto.

Le estensioni riguardano la possibilità per le singole entità di avere più di una *lifeline* e le interazioni attraverso le singole lifeline di ciascuna entità avvengono attraverso l'utilizzo di quelli che vengono chiamati *communication acts* (CA); inoltre sono stati introdotti degli operatori che servono a guidare il comportamento dell'entità analoghi ai classici operatori booleani:

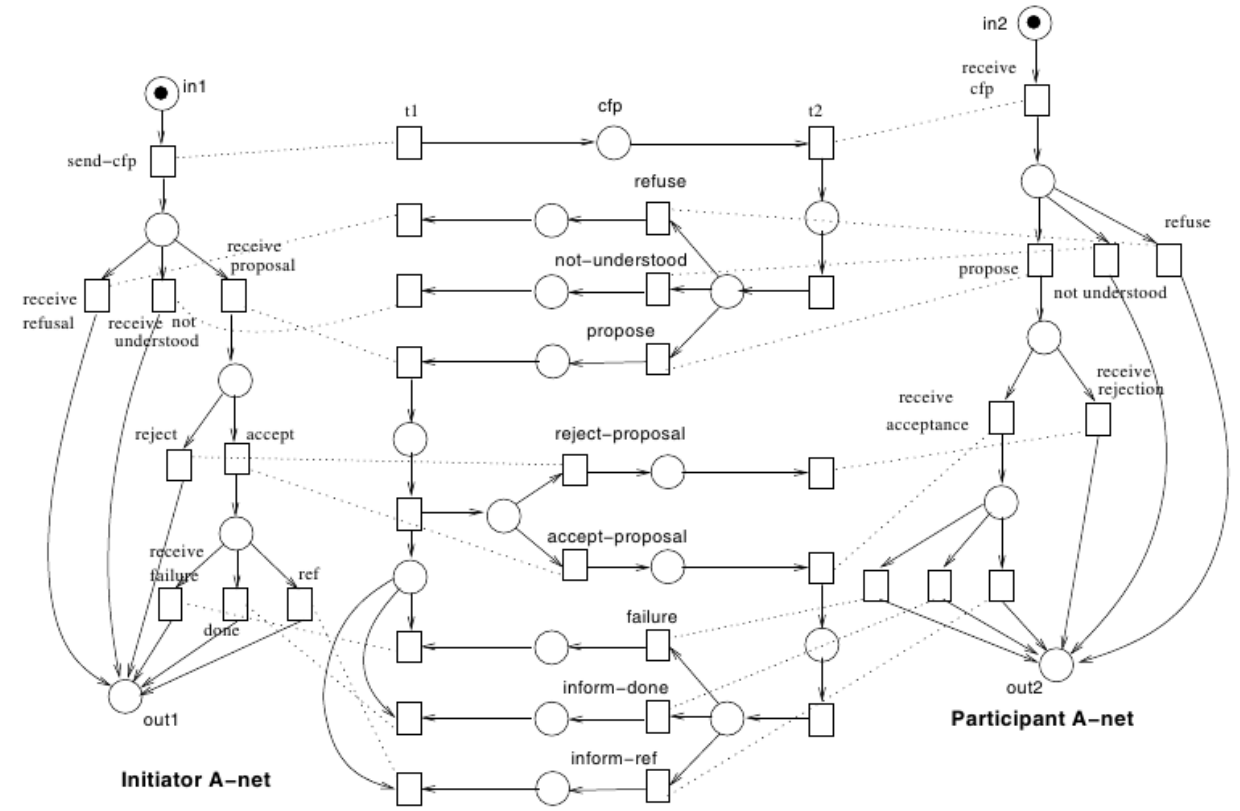
- **AND** – l'operatore AND non è identificato da alcun simbolo ed ha il significato di far eseguire tutti i CA associati parallelamente
- **OR** – l'operatore OR è identificato da un rombo ed ha il significato di poter eseguire uno o più dei CA associati; se viene eseguito più di un CA allora questi sono eseguiti parallelamente
- **XOR** – l'operatore XOR è identificato attraverso lo stesso simbolo dell'operatore OR ma con una X al centro ed ha il significato di far eseguire uno e uno solo tra i CA associati

Bisogna precisare che i CA messi in correlazione tramite un operatore, non necessariamente sono diretti alle *lifeline* della medesima entità, ma ne possono invece coinvolgere un numero arbitrario.



Different notations for inter-agent communication

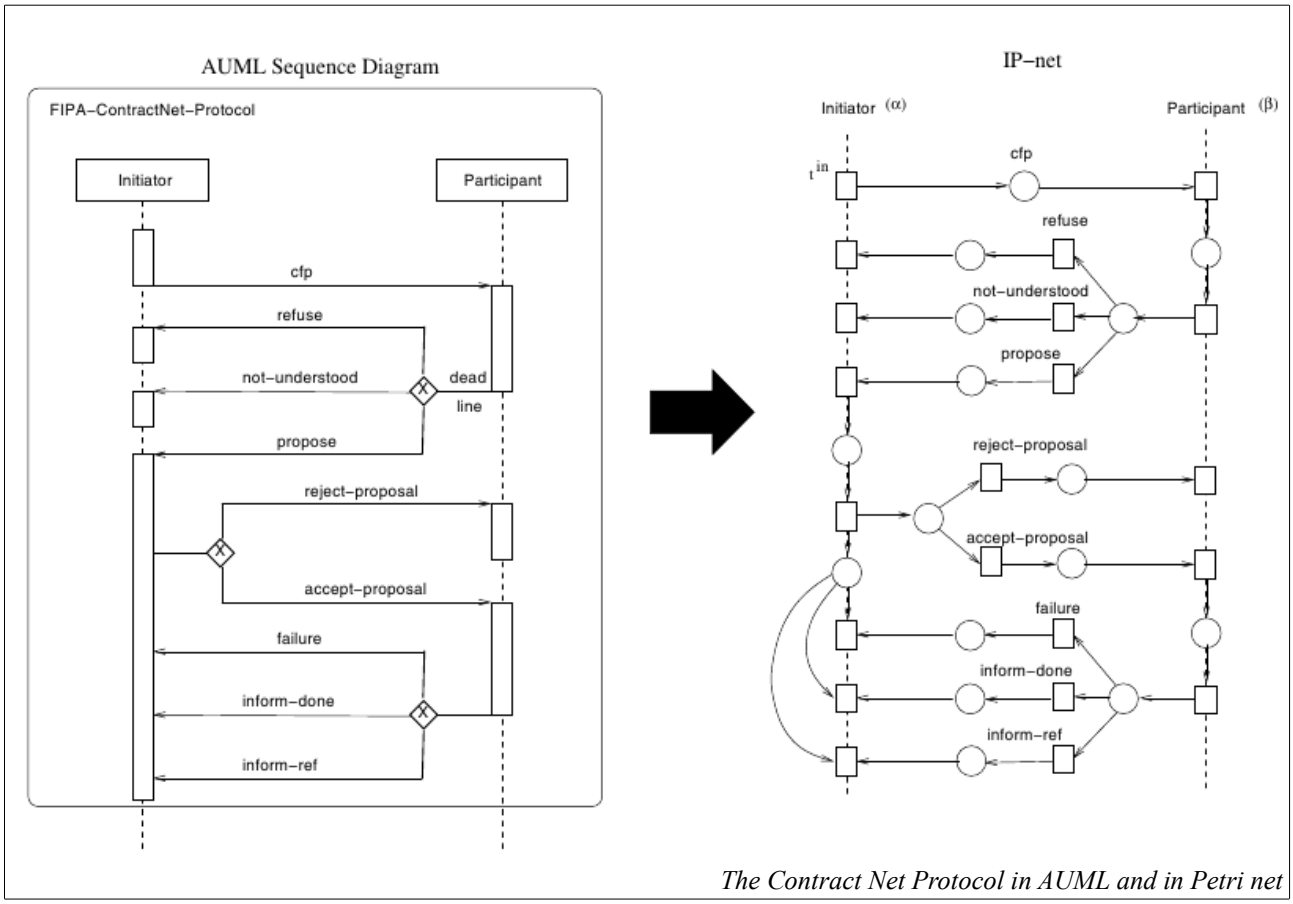
Un protocollo di interazione può essere rappresentato, oltre che tramite un diagramma di sequenza AUML, anche attraverso l'utilizzo di quella che viene chiamata MIPNet.



Una MIPNet non è altro che una rete di Petri formata dall'unione di quelle che vengono definite

Agent Nets (ANets) e *Interaction Protocol Nets* (IPNets), che a loro volta non sono altro che le reti di Petri corrispondenti al comportamento delle singole entità e del protocollo di interazione.

Le due rappresentazioni sono equivalenti, è dunque possibile derivare da un diagramma di sequenza AUML la IPNet corrispondente al protocollo di interazione descritto e le ANet delle entità che lo realizzano.



The Contract Net Protocol in AUML and in Petri net

Le ANet che vengono derivate sono dette *minimali*, cioè col minimo numero di stati necessari al fine di realizzare il protocollo di interazione espresso dalla IPNet. Unendo poi le ANet e la IPNet si ottiene la MIPNet desiderata.

2. Richiami sulla traduzione da AUML a rete di Petri

L'idea di partenza è quella di far eseguire questa traduzione automaticamente, occorre quindi per prima cosa fornire un'adeguata rappresentazione testuale dei diagrammi di sequenza AUML comprensibile ad una macchina. Per fare ciò è stato identificato come componente elementare da utilizzare nella nostra rappresentazione il *communication act*. Ogni CA infatti è identificabile tramite una tripla

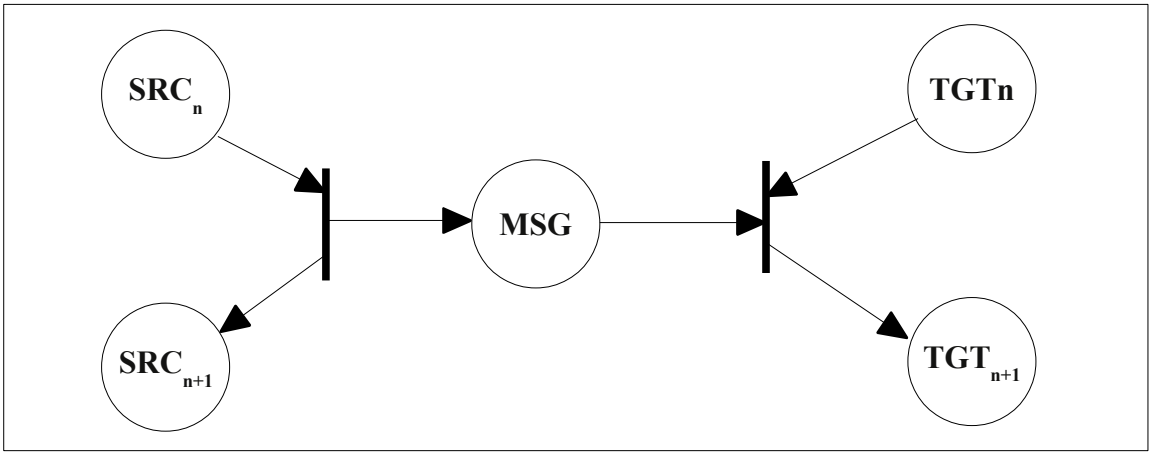
ca (SRC , TGT , MSG)

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

Oltre agli operatori di AND, OR e XOR è necessario introdurre nella rappresentazione testuale un ulteriore operatore → che sarà chiamato operatore di *sequenza*. Il significato dell'operatore di sequenza è quella di fornire una logica temporale ai CA: tutto ciò che sta a sinistra dell'operatore avviene prima di ciò che si trova a destra.

Tutti i protocolli di interazione rappresentabili tramite AUML sono quindi riducibili ad una sequenza di triple messe in relazione attraverso i quattro operatori.

Il passo successivo è quello di riuscire a definire una corretta mappatura 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 di un CA 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, ma gli operatori di AND e OR risultano problematici. È possibile, tuttavia, esprimere i due operatori di AND e OR utilizzando gli altri due operatori → 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

La sequenza di CA così ottenuta, contenente solamente gli operatori → e XOR, può essere finalmente trasformata in una rete di Petri, da cui, aggiungendo delle transizioni che riconducano dai place terminali ai place iniziali delle singole ANet, si ottiene la MIPNet finale.

3. Richiami sull'analisi delle proprietà tramite Maude

Perché una MIPNet generica rappresenti correttamente un protocollo di interazione, cioè sia *sound*, occorre che la rete soddisfi due proprietà fondamentali: la *boundedness* e la *liveness*.

L'analizzatore della rete di Petri sviluppato in Maude è però pensato per essere in grado di

analizzare esclusivamente MIPNet che sono state generate a partire da un diagramma AUML e che appartengono quindi ad un sotto-dominio che garantisce determinate proprietà strutturali della rete; inoltre per garantire la veridicità delle analisi, occorre assumere che la rete analizzata sia connessa, cioè che tutte le entità modellate dalla rete concorrano alla realizzazione di un unico protocollo. Sotto queste ipotesi l'analisi delle proprietà può avvenire secondo quanto indicato di seguito:

- boundedness – l'analisi della *boundedness* viene eseguita ricercando una configurazione indesiderata della rete in cui, a partire dallo stato iniziale, si possa essere in grado di arrivare ad uno stato di *covering* di quello iniziale: una rete di questo tipo è in grado di produrre un numero infinito di stati e quindi non può rispettare la proprietà di *boundedness.*

- 1-boundedness – nell'analizzatore Maude è presente anche un algoritmo per verificare che la MIPNet sia *1-bounded* (*safe*); la *1-boundedness* non è necessaria per determinare la *soundness* della MIPNet, ma può comunque risultare utile identificare le reti *safe.*

- liveness – l'analisi della *liveness* viene eseguita anche qui ricercando una configurazione indesiderata, infatti, una delle proprietà di queste reti risulta essere che, sotto le ipotesi di una rete *bounded* e *connessa*, anche se una sola delle transizioni non dovesse risultare *live*, ciò comporterebbe inevitabilmente, prima o poi, il raggiungimento di uno stato di *deadlock* totale della rete; controllando che in ogni stato ammissibile della rete ci sia sempre una transizione che può scattare, ne garantisce quindi la *liveness*.

4. Jason

Jason è un interprete scritto in Java per AgentSpeak che ne implementa la semantica operazionale e fornisce una piattaforma per lo sviluppo di sistemi multi-agente. AgentSpeak(L) è una estensione alla programmazione logica delle architetture ad agenti BDI e fornisce un supporto alla programmazione di tali agenti.

Un agente AgentSpeak(L) è creato dalla specifica di un set di *belief* e un set di *plan*. Un *belief* è semplicemente un predicato di primo ordine nella usuale notazione e il set di *belief* iniziale non è altro che una collezione di atomi *ground*.

AgentSpeak(L) distingue due tipi di *goal*: *goal obiettivo* e *goal di test*. I *goal* sono predicati (come i *belief*) preceduti dagli operatori "!" e "?" rispettivamente. I *goal obiettivo* definiscono che l'agente vuole raggiungere uno stato del mondo dove il predicato associato risulti vero mentre un *goal di test* restituisce una unificazione per il predicato associato con uno dei *belief* dell'agente.

Un *evento scatenante* definisce quali *eventi* possono causare l'inizio dell'esecuzione di un *plan*. Un *evento* può essere interno, quando occorre raggiungere un *sotto-goal*, oppure esterno, quando causa un aggiornamento dei *belief* in seguito alla percezione dell'ambiente. Ci sono due tipi di *eventi scatenanti*: quelli relativi all'aggiunta (+) e alla cancellazione (-) di *belief* o *goal*.

I *plan* si riferiscono alle azioni base che un agente è in grado di eseguire sull'ambiente. Queste azioni sono anche loro definite tramite predicati del primo ordine ma con degli speciali simboli utilizzati per distinguerle dagli altri predicati. Un *plan* è formato da un *evento scatenante* (che identifica lo scopo di quel *plan*) seguito da una congiunzione di *belief* che rappresentano un contesto. Il contesto dev'essere una conseguenza logica dei *belief* attuali dell'agente affinché il *plan* possa essere applicabile. Il resto del *plan* è una sequenza di azioni o (*sotto-*)*goal* che l'agente deve eseguire o raggiungere quando il *plan*, se applicabile, viene scelto per l'esecuzione.

Jason fornisce inoltre una serie di librerie contenenti delle azioni interne standard a cui ogni agente Jason ha accesso, che gli permettono di modificare la propria base di conoscenza, di agire sull'ambiente e di comunicare con gli altri agenti nell'ambiente.

Per esemplificare meglio la struttura di un agente Jason di seguito viene proposto un esempio tratto dal manuale tecnico di Jason chiamato "*Collecting Garbage*" dove due robot si trovano sulla superficie di Marte. Il robot **r1** cerca della spazzatura e quando ne trova, il robot la raccoglie, la porta alla posizione del robot **r2**, lascia la spazzatura, ritorna alla posizione dove era stata trovata e continua la ricerca da quella posizione. Il robot **r2** è posizionato ad un inceneritore e ogni volta che viene portata della spazzatura alla sua posizione da **r1**, **r2** la mette nell'inceneritore. Uno o due componenti di spazzatura sono sparpagliate casualmente nella griglia. L'azione di raccogliere la spazzatura può fallire, ma si presume che il meccanismo sia abbastanza buono che, nel peggiore dei casi, il robot **r1** debba provare tre volte prima di riuscire definitivamente a raccogliere la spazzatura.

```
  Agent r1
     Beliefs
           pos(r2,2,2).
           checking(slots).
     Plans
[p1]       +pos(r1,X1,Y1) : checking(slots) ← next(slot).
[p2]       +garbage(r1) : checking(slots) ← !stop(check); !take(garb,r2);
                 !continue(check).
[p3]       +!stop(check) : true ← ?pos(r1,X1,Y1); +pos(back,X1,Y1);
                 -checking(slots).
[p4]       +!take(S,L) : true ← !ensure_pick(S); !go(L); drop(S).
[p5]       +!ensure_pick(S) : garbage(r1) ← pick(garb); !ensure_pick(S).
[p6]       +!ensure_pick(S) : true ← true.
[p7]       +!continue(check) : true ← !go(back); -pos(back,X1,Y1);
                 +checking(slots); next(slot).
[p8]       +!go(L) : pos(L,Xl,Yl) & pos(r1,Xl,Yl) ← true.
[p9]       +!go(L) : true ← ?pos(L,Xl,Yl); moveTowards(Xl,Yl); !go(L).
```

Gli unici *belief* iniziali di cui l'agente **r1** ha bisogno sono la posizione dell'agente **r2** nella griglia nella quale è stato diviso il territorio e ciò che sta facendo, cioè cercare la spazzatura all'interno della griglia.

Il piano p1 è utilizzato quando l'agente è in una nuova posizione e sta controllando in cerca di spazzatura; se non ce n'è, allora passa alla prossima posizione (**next(slot)**).

Il piano p2 è utilizzato nel momento in cui l'agente percepisce spazzatura nella sua posizione (**garbage(r1)**) .

L'azione di portare la spazzatura all'inceneritore e tornare alla posizione attuale viene eseguito attraverso i *sotto-goal* identificati dai piani p3, p4 e p7 che rispettivamente memorizzano la posizione attuale (**pos(back,X1,Y1)**), raccolgono la spazzatura (anche questo eseguito attraverso i due ulteriori *sotto-goal* p5 e p6), ritornano alla posizione precedentemente salvata e ricominciano a controllare.

Gli ultimi due piani sono utilizzati per raggiungere il *goal* di portarsi in una posizione specifica nella griglia: p9 controlla la posizione che si vuole raggiungere ed esegue un azione di avvicinamento (**moveTowards(Xl,Yl)**) mentre p8 fornisce la condizione di terminazione della ricorsione.

```
  Agent r2
     Plans
           +garbage(r2) : true ← burn(garb).
```

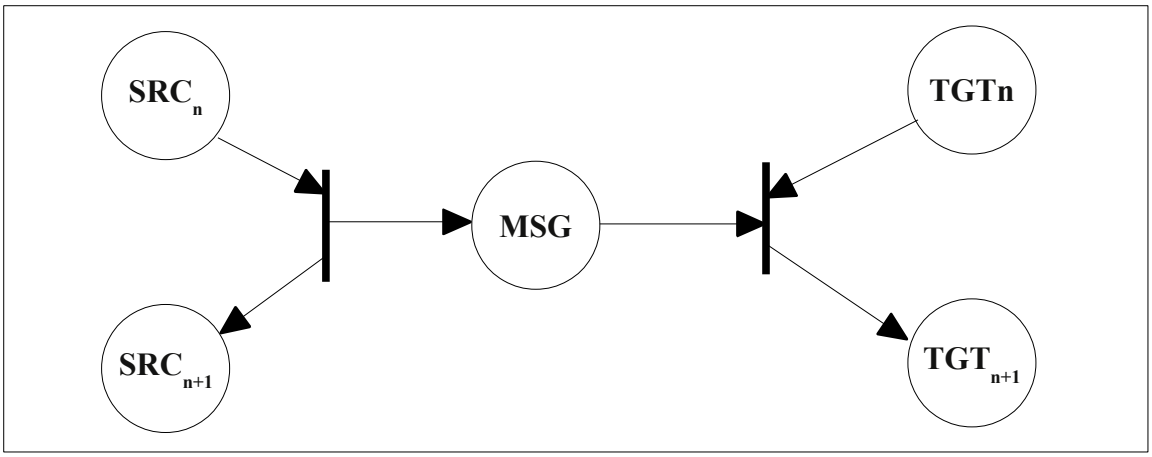
Tutto ciò che **r2** fa è bruciare la spazzatura (**burn(garb)**) quando percepisce che c'è della spazzatura nella propria posizione (**garbage(r2)**).

In questo esempio si presume che i due agenti abbiano entrambi delle azioni interne che permettano loro di spostarsi all'interno della griglia o di raccogliere la spazzatura.

5. Dalla MIPNet agli agenti Jason

Analogamente a quanto fatto in precedenza durante il processo di traduzione del modello AUML in una rete di Petri, è possibile identificare delle analogie tra la MIPNet e il modello di un agente Jason che implementi il protocollo espresso da questa rete.

Per prima cosa occorre suddividere idealmente la MIPNet nelle sue *sotto-reti* componenti, cioè la IPNet e le varie ANet; mentre nel caso della traduzione dal modello AUML ci si focalizza sul rapporto tra lo schema AUML e la IPNet, adesso occorre evidenziare le caratteristiche delle ANet, poiché queste costituiscono uno scheletro vero e proprio utilizzabile per costruire l'agente Jason che sarà dunque l'esatta trasposizione della ANet corrispondente.



Tutta la MIPNet è costituita da *sotto-reti* che rispecchiano il modello in figura, queste *sotto-reti*, dal punto di vista degli agenti coinvolti, rappresentano una emissione di un CA per SRC e una ricezione di un CA per TGT; possiamo quindi ridurre il modello ad una serie di azioni di emissione o di ricezione di CA che modificano lo stato interno degli agenti.

Dal punto di vista della ANet dell'agente **SRC** si ha un *plan* costruito secondo il seguente schema:

+state(N) : true ← .send(TGT,tell,MSG); -state(N); +state(N+1).

La precondizione per l'esecuzione del *plan* è di trovarsi nello stato **N** (identificato qui da un *belief* **state(N)**). La guardia è stata impostata a **true** poiché non ci sono ulteriori condizioni per l'esecuzione del *plan* (il caso in cui il *plan* sia una delle possibili alternative verrà discusso in seguito). I *sotto-goal* eseguiti dal *plan* inviano **MSG** all'agente **TGT** (qui utilizzando l'azione interna standard **send**) e modificano di conseguenza la base di conoscenza, passando dallo stato **N** allo stato **N+1**.

Dal punto di vista della ANet dell'agente **TGT** si ha un *plan* costruito secondo il seguente schema:

+MSG[source(SRC)] : state(N) ← -MSG[source(SRC)]; -state(N); +state(N+1).

La precondizione per l'esecuzione del *plan* è di aver ricevuto dall'agente **SRC** il messaggio **MSG**. La guardia, in questo caso, è di trovarsi effettivamente nello stato in cui il protocollo prevede la ricezione di **MSG**. I *sotto-goal* modificano la base di conoscenza eliminando l'avvenuta ricezione di **MSG** e spostandosi dallo stato **N** allo stato **N+1**.

Una volta eseguito questo processo di traduzione non rimane altro che aggiungere ai vari agenti i *plan* che permettono dagli stati finali di ritornare a quello iniziale, analogamente al processo eseguito sulla rete di Petri:

+state(N) : true ← -state(N); +state(0).

La precondizione del *plan* è quella di trovarsi nello stato **N**, qui ipoteticamente identificato con uno degli stati finali dell'agente. La guardia è **true** poiché non ci sono altre condizioni per l'esecuzione del *plan*. I sotto-*goal* eseguiti modificano la base di conoscenza interna tornando dallo stato **N** allo stato **0**.

Nel caso in cui in un particolare stato si abbiano a disposizione differenti *plan* alternativi tra cui scegliere, modellando in questa maniere l'agente Jason, i *plan* saranno tutti potenzialmente eseguibili ma verrà sempre scelto il primo risultante dal matching dell'algoritmo di selezione interno. Questo non è sbagliato, poiché l'informazione sulla scelta di un *plan* rispetto a un altro non scaturisce dal modello ma deve essere fornita dall'implementatore finale, che, al momento del completamento dell'agente con le funzionalità specifiche, aggiungerà allo scheletro generato automaticamente le condizioni di scelta dei *plan* alternativi secondo i propri desideri.

6. Caso di studio esemplificativo

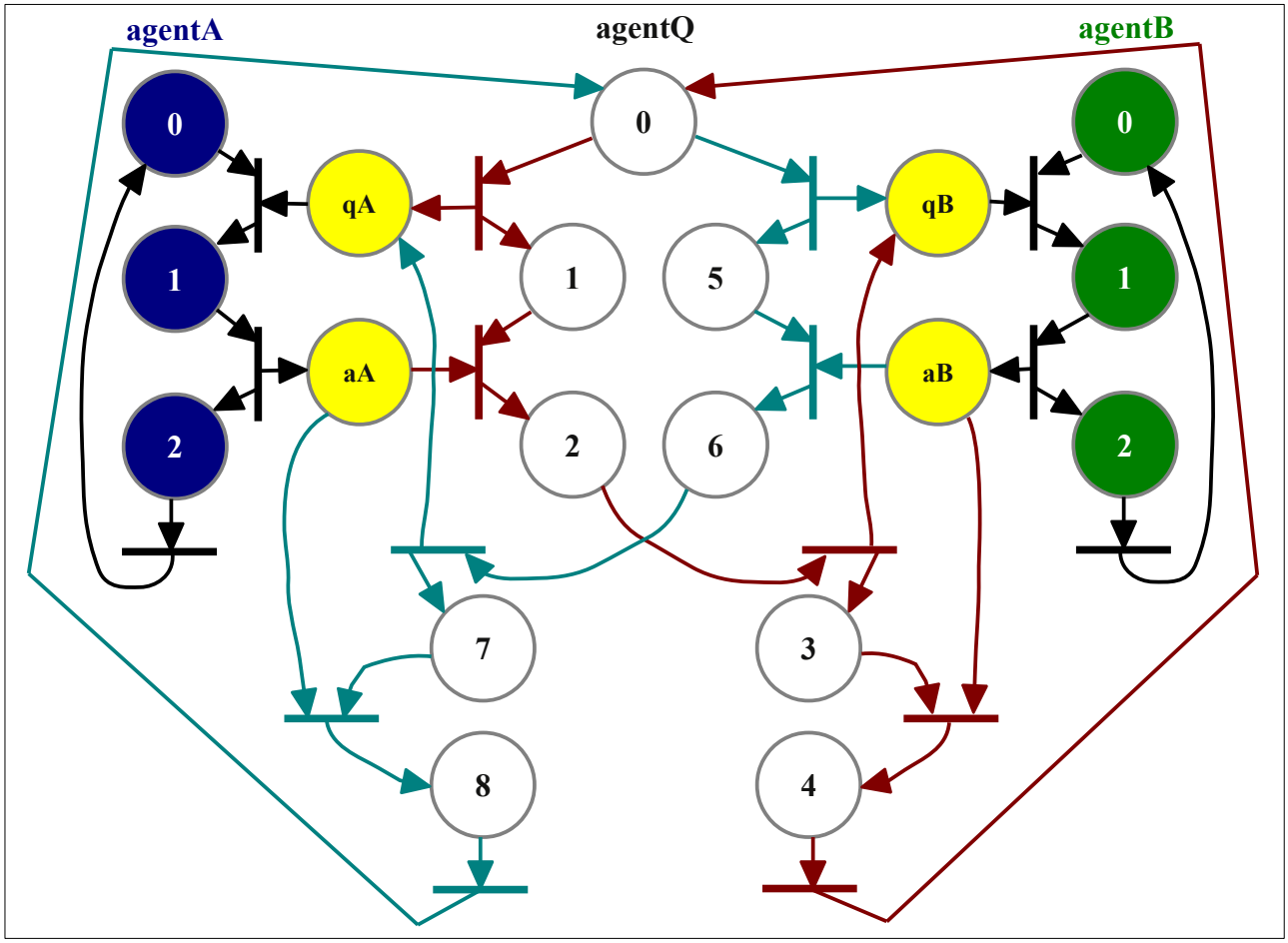
Di seguito viene riportato un esempio dove il processo di traduzione viene applicato ad un semplice protocollo di interazione dato dal seguente modello:

```
(ca(agentQ,agentA,question) → ca(agentA,agentQ,answer))
      and
(ca(agentQ,agentB,question) → ca(agentB,agentQ,answer))
```

Il protocollo non viene tradotto direttamente ma, analogamente a quanto fatto nel caso di creazione della MIPNet, il modello viene raffinato fino ad ottenere la forma contenente solo operatori → e **xor**:

```
xor(
      →(
            →(ca(agentQ,agentA,question),ca(agentA,agentQ,answer)),
            →(ca(agentQ,agentB,question),ca(agentB,agentQ,answer))
      ),
      →(
            →(ca(agentQ,agentB,question),ca(agentB,agentQ,answer)),
            →(ca(agentQ,agentA,question),ca(agentA,agentQ,answer))
      )
)
```

A questo punto si procede come nel caso di creazione della MIPNet, sostituendo alle coppie di *transizioni* associate ad ogni CA le coppie di *plan* corrispondenti, come descritto nel paragrafo precedente. Per mostrare più in dettaglio la correlazione tra la MIPNet e gli agenti Jason che verranno creati, di seguito è presentato lo schema grafico della MIPNet associata:



L'agente Jason corrispondente ad **agentA** (e anche ad **agentB**, essendo il loro comportamento identico) è il seguente:

```
/* Initial beliefs and rules */
     state(0).
/* Plans */
[p0]  +question[source(agentQ)] : state(0) ←
           -question[source(agentQ)]; -state(0); +state(1).
[p1]  +state(1) : true ←
           .send(agentQ,tell,answer); -state(1); +state(2).
[p2]  +state(2) : true ←
           -state(2); +state(0).
```

Osservando la rappresentazione testuale delle *transizioni* corrispondenti della MIPNet la correlazione fra *transizioni* e *plan* è ancora più evidente:

- **P0** = [agentA(0) + qA → agentA(1)]
- **P1** = [agentA(1) → agentA(2) + aA]

- **P2** = [agentA(2) → agentA(0)]

I *place* con archi in ingresso alla *transizione* diventano le precondizioni e le eventuali guardie del *plan* dell'agente Jason. I *place* con archi in uscita dalla *transizione* rappresentano invece i valori aggiornati della base di conoscenza interna dell'agente. I *token* associati ai *place* che non corrispondono ad alcuna entità, cioè che rappresentano lo scambio di informazione, sono rimpiazzati all'interno dell'agente dalla ricezione, nel caso di precondizione, o all'invio, nel caso di post-condizione, di una istruzione **send** da/verso l'agente interlocutore.

L'agente Jason corrispondente ad **agentQ** invece risulta più articolato:

```
/* Initial beliefs and rules */
     state(0).
/* Plans */
[p0]  +state(0) : true ←
           .send(agentA,tell,question); -state(0); +state(1).
[p1]  +answer[source(agentA)] : state(1) ←
           -answer[source(agentA)]; -state(1); +state(2).
[p2]  +state(2) : true ←
           .send(agentB,tell,question); -state(2); +state(3).
[p3]  +answer[source(agentB)] : state(3) ←
           -answer[source(agentB)]; -state(3); +state(4).
[p4]  +state(0) : true ←
           .send(agentB,tell,question); -state(0); +state(5).
[p5]  +answer[source(agentB)] : state(5) ←
           -answer[source(agentB)]; -state(5); +state(6).
[p6]  +state(6) : true ←
           .send(agentA,tell,question); -state(6); +state(7).
[p7]  +answer[source(agentA)] : state(7) ←
           -answer[source(agentA)]; -state(7); +state(8).
[p8]  +state(8) : true ←
           -state(8); +state(0).
[p9]  +state(4) : true ←
           -state(4); +state(0).
```

Anche qui è possibile osservare la correlazione diretta tra i *plan* e le *transizioni* della MIPNet:

- **P0** = [agentQ(0) → agentQ(1) + qA]
- **P1** = [agentQ(1) + aA → agentQ(2)]
- **P2** = [agentQ(2) → agentQ(3) + qB]
- **P3** = [agentQ(3) + aB → agentQ(4)]
- **P4** = [agentQ(0) → agentQ(5) + qB]
- **P5** = [agentQ(5) + aB → agentQ(6)]
- **P6** = [agentQ(6) → agentQ(7) + qA]
- **P7** = [agentQ(7) + aA → agentQ(8)]
- **P8** = [agentQ(8) → agentQ(0)]
- **P9** = [agentQ(4) → agentQ(0)]

Si può notare come il caso in cui l'agente abbia la possibilità di eseguire uno tra più *plan*, in questo particolare caso **P0** e **P4**, entrambi eseguibili in corrispondenza del *belief* **state(0)**, corrisponda nella MIPNet allo stato **agentQ(0)**, in cui è possibile far scattare una delle due *transizioni* che porterebbero negli stati **agentQ(1)** o **agentQ(5)**, come in effetti ci si aspetterebbe di trovare.

7. Conclusione e possibili sviluppi futuri

Durante la realizzazione di un sistema informatico, gli sforzi degli sviluppatori si devono concentrare principalmente su tre fasi distinte: l'analisi, la progettazione e l'implementazione.

Nel caso di un sistema multi-agente, gli sforzi richiesti durante le prime due fasi sono sicuramente maggiori rispetto a quelli richiesti da un normale sistema informatico.

Grazie all'utilizzo combinato di differenti tecnologie, tra le quali le reti di Petri, è stato possibile realizzare un sistema di implementazione automatica di modelli AUML utilizzando agenti Jason, che permette al progettista di curarsi solo minimamente della parte implementativa.

Il progetto ha anche messo in evidenza come le reti di Petri siano una rappresentazione trasversale che si adatta particolarmente bene alla modellazione di sistemi multi-agente, facilitando la rappresentazione delle interazioni.

Gli agenti Jason sono solo una delle tante tecnologie utilizzabili per la realizzazione concreta di sistemi multi-agente, per cui un ampliamento decisamente significativo del progetto sarà quello di fornire meccanismi di implementazione automatica anche verso altre piattaforme, con particolare interesso per il linguaggio di coordinazione ReSpecT.

Bibliografia

MODELING AGENT INTERACTIONS: AGENT UML AND MULTIAGENT INTERACTION PROTOCOL NETS
Marco Alberti
Seconda Facoltà di Ingegneria – Cesena, Italia
marco.alberti3@studio.unibo.it