ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

IMPLEMENTAZIONE AUTOMATICA DI PROTOCOLLI DI INTERAZIONE AUML MEDIANTE RETI DI PETRI

Elaborata nel corso di: Sistemi Multi-Agente L-S

Tesi di Laurea Specialistica di:
MARCO ALBERTI

Relatore:
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

Sistemi Multiagente [MAS]
AUML
Reti di Petri
Jason
ReSpecT
TuCSoN

Ci sono due modi di scrivere un programma senza errori.
Solo il terzo funziona.

Indice

Introduzione **ix**

1 AUML e le MIP-Net **1**

1.1 *Oggetti* e *agenti* 1
1.1.1 Classi di *oggetti* 1
1.1.2 Classi di *agenti* 2
1.1.3 Comunicazione 3
1.1.4 Agent head automaton 4
1.1.5 Ereditarietà 5
1.2 AUML 7
1.2.1 Interazione in AUML 7
1.2.2 Diagrammi di sequenza 8
1.3 MIP-Net 10
1.3.1 A-Net 10
1.3.2 IP-Net 11
1.3.3 Sincronizzazione tra A-Net e IP-Net 12
1.3.4 *Soundness* dei sistemi 13
1.3.5 *Dispiegamento* di una MIP-Net 14
1.3.6 A-Net *minimali* 15

2 Traduzione e analisi **17**

2.1 Algoritmo di traduzione 17
2.1.1 Rappresentazione testuale di AUML 17
2.1.2 Da AUML alla rete di Petri 18
2.1.3 Algoritmo in `tuProlog` 20
2.2 Analisi delle proprietà 27
2.2.1 Analisi tramite LTL 27
2.2.2 Rappresentazione della rete in `Maude` 29
2.2.3 Model checking 30

3 Implementazione su `Jason` **33**

3.1 La piattaforma `Jason` 33

3.2 MIP-Net e agenti `Jason` 35
3.3 Caso di studio 37

4 Implementazione su `TuCSoN` 41
4.1 Il meta-modello A&A 41
4.2 Il linguaggio `ReSpecT` 42
4.2.1 A&A in `ReSpecT` 42
4.3 Primo modello: agenti complessi 45
4.3.1 Gli agenti 45
4.3.2 Gli artefatti 47
4.3.3 Caso di studio 48
4.4 Secondo modello: artefatti complessi 51
4.4.1 Gli agenti 51
4.4.2 Gli artefatti 52
4.4.3 Caso di studio 54

5 Conclusioni 61

Introduzione

Il cambio di paradigma negli ambienti informatici, è spesso rallentato dall'inerzia con la quale certe tecnologie continuano a protrarsi nel tempo, non per una qualche loro caratteristica particolare che le rende preferibili alle altre, ma per la familiarità che ormai i loro utilizzatori hanno maturato nel corso del tempo. Augurando ogni bene alle tecnologie che si sono ormai affermate, è innegabile che, dapprima in ambito accademico, ma ultimamente anche in ambito industriale, abbiano cominciato a fare capolino quelli che ormai sono universalmente noti come sistemi *agent based*.

In ambiente accademico e di ricerca, i sistemi *agent based* non sono qualcosa di nuovo, dimostrato dal fatto che la quasi totalità delle infrastrutture in grado di supportarli siano state sviluppate, appunto, in ambito accademico, ma negli ultimi anni, questi sistemi hanno cominciato ad acquistare popolarità anche al di fuori degli atenei. Le applicazioni principali per le quali sono stati utilizzati riguardano: le simulazioni di ambienti complessi, dove il gran numero di entità coinvolte rende impossibile la definizione di modelli matematici; sistemi software distribuiti, dove il disaccoppiamento spaziale impedisce un approccio classico; lo sviluppo di applicazioni pensate per l'esecuzione su macchine *multi-core*, per sfruttare al massimo le risorse a disposizione; etc.

Qualsiasi sia l'uso che ne viene fatto, i modelli *agent based* necessitano di un attento studio riguardo al modo in cui gli agenti interagiscono tra di loro, che collaborino a realizzare un task o che concorrano per accedere a delle risorse condivise.

Questa tesi si basa su una serie di lavori precedenti, volti ad analizzare la correlazione tra i modelli AUML e le reti di Petri, per riuscire a fornire una metodologia di traduzione dai primi alle seconde. Questa traduzione permetterà di applicare tecniche di *model checking* alle reti così create, al fine di stabilire le proprietà necessarie al sistema per poter essere realizzato effettivamente. Verrà poi discussa un'implementazione

di tale algoritmo sviluppata in `tuProlog` ed un primo approccio al *model checking* utilizzando il programma `Maude`.

Con piccole modifiche all'algoritmo utilizzato per la conversione dei diagrammi AUML in reti di Petri, è stato possibile, inoltre, realizzare un sistema di implementazione automatica dei protocolli precedentemente analizzati, verso due piattaforme per la realizzazione di sistemi multiagente: `Jason` e `TuCSoN`. Verranno quindi presentate tre implementazioni diverse:

- la prima per la piattaforma `Jason`, che utilizza degli agenti BDI per realizzare il protocollo di interazione
- la seconda per la piattaforma `TuCSoN`, che utilizza il modello A&A per rendersi compatibile ad un ambiente distribuito, ma che ricalca la struttura dell'implementazione precedente
- la terza ancora per `TuCSoN`, che sfrutta gli strumenti forniti dalle reazioni `ReSpecT` per generare degli artefatti in grado di fornire una infrastruttura in grado di garantire la realizzazione del protocollo di interazione agli agenti partecipanti

Infine, verranno discusse le caratteristiche di queste tre differenti implementazioni su un caso di studio reale, analizzandone i punti chiave.

Capitolo 1

AUML e le MIP-Net

1.1 *Oggetti* e *agenti*

Di seguito viene discussa una estensione del ben noto linguaggio UML, largamente impiegato in ambito *object oriented*, fornendolo di capacità espressiva aggiuntiva, per modellare efficacemente e intuitivamente le interazioni che caratterizzano e determinano il comportamento di un sistema multiagente [2, 5].

Il passo successivo sarà quello di fornire una serie di regole e metodologie per tradurre un modello scritto in AUML, in una rete di Petri particolare chiamata MIP-Net [4], per poter usufruire delle tecniche di *model checking* proprie delle reti di Petri.

1.1.1 Classi di *oggetti*

Un oggetto è composto da un insieme di variabili, attributi, campi e dai suoi metodi. Il valore delle variabili può essere un dato primitivo o un riferimento ad un altro oggetto. I metodi sono funzioni, operazioni o procedure che possono coinvolgere le variabili interne dell'oggetto o altri oggetti. Una classe descrive un set di oggetti, definiti istanze di tale classe, che condividono la stessa struttura: hanno le stesse variabili, lo stesso comportamento e gli stessi metodi.

Le classi vengono create tramite l'utilizzo di un qualche tipo di invocazione standard "**new**"; la definizione è data dalla dichiarazione delle variabili interne e dall'implementazione dei metodi; più comunemente da una parte di specifica, o interfaccia, e da una parte di implementazione. La parte di specifica definisce i metodi supportati dalla classe e le loro

funzionalità, tralasciando il modo in cui viene realizzata l'operazione; la parte di implementazione definisce la realizzazione delle operazioni ed è solitamente nascosta all'utilizzatore del metodo. Delle proprietà specificano la visibilità dei metodi o delle variabili da parte dell'utilizzatore. Nella maggior parte dei linguaggi le classi definiscono dei tipi. Alcuni linguaggi permettono la definizione di variabili di classe condivise da tutte le istanze. Oltre alle variabili di classe, i metodi di classe possono essere richiamati indipendentemente dalla creazione di un oggetto. Sia le variabili che i metodi di classe possono essere utilizzati, rispettivamente, come variabili e procedure globali.

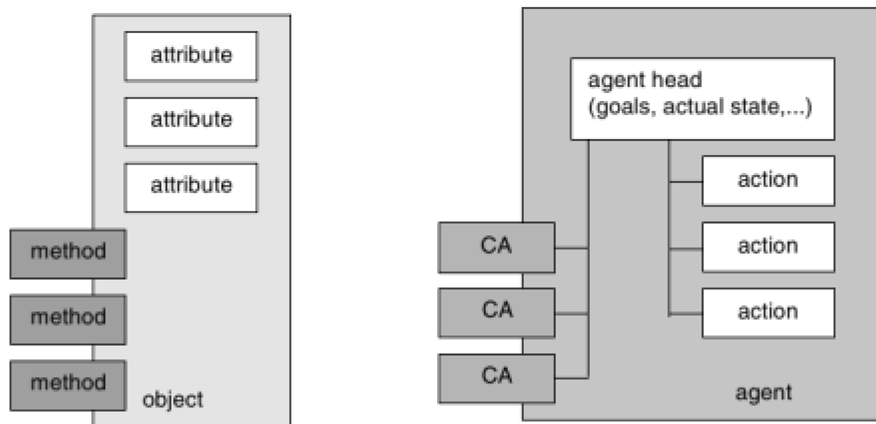

Figura 1.1: Objects and Agents

1.1.2 Classi di *agenti*

A differenza di un oggetto, un agente è caratterizzato da:

- un comportamento non procedurale
- equilibrio tra reattività e pro-attività
- pattern e protocolli di interazione complessi
- uno stato interno costituito da convinzioni, obiettivi e piani

Un agente può essere diviso idealmente in tre parti: interfaccia comunicativo, testa e corpo. L'interfaccia comunicativo è responsabile di gestire gli scambi di informazioni, di solito eseguito tramite protocolli di tipo *message passing* supportati dall'infrastruttura sottostante; la testa contiene le informazioni in possesso dell'agente, obiettivi o convinzioni, lo

stato interno; il corpo invece definisce il comportamento effettivo dell'agente. Deve essere riservata una particolare attenzione alla semantica degli atti comunicativi e alla reazione degli agenti. Può essere stabilita dal progettista del sistema multiagente oppure non specificata in fase di design ma derivata durante l'esecuzione dall'agente stesso tramite un qualche metodo di ragionamento.
Un agente decide autonomamente se eseguire o meno un'azione. Un'azione astratta è caratterizzata da pre-condizioni, effetti e invarianti; in più agli agenti è possibile applicare tecniche proprie del mondo *object oriented* quali ereditarietà, tipi astratti e interfaccia, generici, associazione, aggregazione e composizione. I componenti possono essere sia classi *agent oriented* sia comuni classi *object oriented*. Un agente può essere infatti costruito includendo oggetti come parte del suo stato interno.

Nel paradigma *agent oriented* dobbiamo distinguere tra classi di agenti e agenti individuali. UML permette di distinguere livello concettuale, di specifica e di implementazione:

- nel livello concettuale, alla classe di un agente corrisponde il ruolo di un agente
- nel livello di specifica, la classe di un agente rappresenta la struttura per le istanze degli agenti, ma vengono definiti solo gli stati interni e gli interfaccia e non la loro implementazione
- il livello di implementazione è quello più dettagliato, mostrando come lavorano le istanze degli agenti e la loro implementazione

Il ruolo identifica un set che soddisfa particolari proprietà o interfaccia, che fornisce particolari servizi o che denota un particolare comportamento. UML distingue classificazioni multiple, dove un agente può ricoprire più ruoli contemporaneamente, o classificazioni dinamiche, dove un agente cambia il proprio ruolo durante la sua esistenza. Gli agenti possono ricoprire vari ruoli all'interno un protocollo di interazione, ne consegue che l'implementazione di un agente può soddisfare ruoli differenti. All'interno di un sistema multiagente il ruolo può essere applicato a due livelli: può riferirsi ad istanze concrete oppure ad un set di agenti conformi ad un particolare ruolo o classe.

1.1.3 Comunicazione

Inviare e ricevere atti comunicativi (CA in breve) caratterizza l'interfaccia di un agente nel suo ambiente. Viene assunto che classi e oggetti

rappresentino l'informazione circa gli atti comunicativi [2]. L'atto comunicativo ricevuto o inviato può essere sia una classe che una istanza concreta. Si deve distinguere tra istanze e classi poiché una istanza descrive un atto comunicativo concreto con un contenuto, o altri campi, fissati. In alcuni contesti è utile invece utilizzare classi al fine di ottenere maggiore flessibilità. Un atto comunicativo ricevuto deve essere accoppiato con gli atti comunicativi in ingresso per attivare il comportamento dell'agente corrispondente. Nel caso in cui ci sia più di un matching possibile, viene data la precedenza nella scelta considerando l'ordinamento, nominalmente dall'alto verso il basso.
Un atto comunicativo $\mathbf{CA}^1$ risolve il matching con uno $\mathbf{CA}^2$, se:

- $\mathbf{CA}^1$ è una classe
  - $\mathbf{CA}^2$ deve essere una istanza della classe $\mathbf{CA}^1$
  - $\mathbf{CA}^2$ deve essere una istanza di una sottoclasse di $\mathbf{CA}^1$
- $\mathbf{CA}^1$ è una istanza di una classe
  - $\mathbf{CA}^2$ è una istanza della stessa classe di cui è istanza $\mathbf{CA}^1$
  - $\mathbf{CA}^1$**.field** risolve il matching con $\mathbf{CA}^2$**.field** per tutti i campi **field** della classe $\mathbf{CA}^1$

Un campo $\mathbf{CA}^1$**.field** risolve il matching con un campo $\mathbf{CA}^2$**.field**, se:

- $\mathbf{CA}^1$**.field** non ha un valore associato
- $\mathbf{CA}^1$**.field** è uguale a $\mathbf{CA}^2$**.field** ed è un tipo base
- $\mathbf{CA}^1$**.field** è una istanza della classe **C**
  - $\mathbf{CA}^1$**.field.cfield** risolve il matching con $\mathbf{CA}^2$**.field.cfield** per tutti i campi **cfield** di **C**

1.1.4 Agent head automaton

Un agente è stato definito come composto da tre parti. Le funzionalità principali sono implementate nel corpo, ma la testa è ciò che permette il ragionamento tramite quello che viene definito *agent head automaton*. Qui gli ingressi vengono relazionati allo stato interno e le azioni ed i metodi alle uscite. Queste regole permettono il comportamento reattivo e/o pro-attivo dell'agente [2].
Per il comportamento reattivo viene specificato come l'agente reagisce ai messaggi in ingresso usando un automa a stati esteso. A differenza di

un automa a stati standard, i CA in ingresso attivano un automa (stato iniziale) e i CA in uscita lo chiudono (stato finale).
Il comportamento pro-attivo non è scatenato da CA in ingresso ma dipende dallo stato dei vincoli e delle condizioni con le quali vengono marcati gli stati iniziali dell'automa.

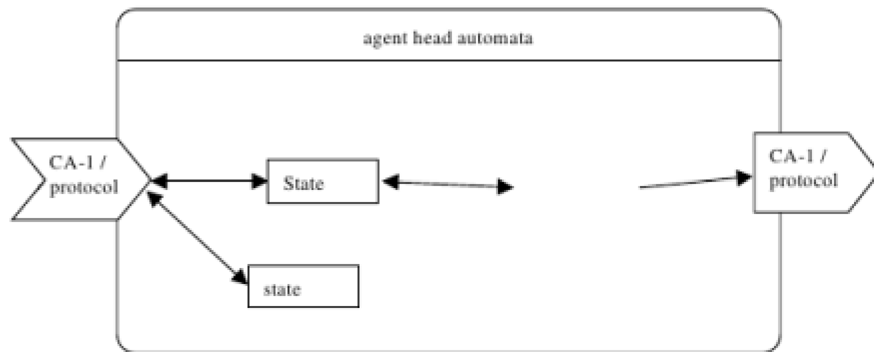

Figura 1.2: Agent head automaton

UML supporta quattro tipi di diagrammi per la definizione di comportamenti dinamici: diagrammi di sequenza e diagrammi di collaborazione in contesti ad oggetti, diagrammi di stato e diagrammi di attività per il resto. I primi due sono adatti alla definizione del comportamento della testa dell'agente, gli ultimi due sono più indicati per una specifica astratta del comportamento dell'agente.

1.1.5 Ereditarietà

Una delle principali caratteristiche e meccanismo per la modularità e la strutturazione dei linguaggi *object oriented* è l'ereditarietà. Tale meccanismo è applicabile anche ai sistemi *agent based.* Solitamente in letteratura troviamo tre tipi di ereditarietà: singola, multipla e dinamica; le ragioni per introdurla e usarla sono principalmente che la gerarchia delle classi definisce una gerarchia dei tipi, favorisce la riusabilità, favorisce l'interscambiabilità, supporta intrinsecamente la condivisione di un comportamento comune.

L'ereditarietà **semplice** definisce alcuni tipi di gerarchia: la classe **B** è un sottotipo della classe **A** e le istanze di **B** sono un subset delle istanze di **A**. Se la classe **B** eredita dalla classe **A**, allora **B** è una sottoclasse di **A** e **A** è una super-classe di **B**. L'ereditarietà singola permette di

esprimere che una istanza della classe **A** e le istanze della classe **B** sono anche istanze di un'ulteriore classe **C**. Per l'implementazione di tale meccanismo sono state percorse due strade: *copy-view* e *search-view*. Nella prima, come per esempio nel C++, se **B** eredita da **A** allora **B** possiede tutti i componenti di **A**. La seconda è possibile solo se l'accesso alle variabili e ai metodi è eseguito nella classe vera e propria, come in Java: se il componente non è presente, viene cercato nelle super-classi. Da un lato il concetto di ereditarietà semplifica l'implementazione e dall'altro rende l'implementazione più sicura. L'implementazione è semplificata visto che il codice può essere riutilizzato, modificato o esteso utilizzando l'ereditarietà senza riscrivere tutto il codice. I metodi possono essere ridefiniti per specializzarli per casi particolari così come possono essere definiti nuovi metodi per estendere le funzionalità di una data classe. L'implementazione è più sicura perché può essere utilizzato codice testato.
L'ereditarietà supporta sia approcci allo sviluppo di tipo *top-down* che *bottom-up*: fissando i tipi dei dati e gli interfaccia, i sistemi possono essere sviluppati *top-down*; usare librerie di classi e combinarle con altre classi risulta in un approccio *bottom-up*.

La generalizzazione e la specializzazione possono essere espresse sempre utilizzando l'ereditarietà, dato che viene definita una gerarchia dei tipi, che permette ad un oggetto di appartenere a più di una classe. Una generalizzazione viene ottenuta estraendo componenti comuni di una classe (variabili e metodi) utilizzati anche in classi differenti e definendoli in una classe propria. Una specializzazione viene ottenuta definendo una nuova classe **B**, che eredita da una classe **A**; dopodiché possono essere aggiunti componenti alla classe **B** non presenti in **A**.

Attualizzando l'ereditarietà in un contesto ad agenti si può ricavare questa prima caratterizzazione [2]:

- una sottoclasse eredita tutti i ruoli della super-classe
- stato interno:
  - possono essere definiti campi addizionali, con nuovi nomi e tipi
  - possono essere inizializzati i campi definiti nella super-classe
  - l'inizializzazione può essere ridefinita
- azioni:
  - sono ereditate dalla super-classe

  - possibilità di ridefinirle senza però la possibilità di distinguere il tipo del risultato

- metodi:
  - come per le azioni, ma capacità di definire il tipo dei risultati
  - non visibili agli altri agenti

- le capacità sono ereditate: tutto ciò che fa la super-classe lo fa anche la sottoclasse

- CA:
  - la ricerca è eseguita come in caso di *method-call* nel linguaggio *object oriented*
  - i CA in uscita sono determinati dal *agent head automaton*

- *agent head automaton*:
  - sono ereditati dalla super-classe
  - possono essere ridefiniti con lo stesso nome
  - può essere aggiunto comportamento addizionale

1.2 AUML

Uno dei principali aspetti su cui si concentra AUML è di estendere UML in modo da modellare protocolli di interazione tra agenti. Questi protocolli descrivono i pattern di comunicazione permessi fra i diversi ruoli ricoperti dagli agenti [5]. In UML sono presenti due diagrammi utili a questo fine: i diagrammi di sequenza e i diagrammi di collaborazione. I diagrammi di interazione (cioè entrambi i diagrammi prima citati) vengono utilizzati per rappresentare come le istanze interagiscono per eseguire determinati task che singolarmente non sarebbero in grado di eseguire.

1.2.1 Interazione in AUML

Ogni diagramma AUML mostra una interazione (per scenario) come una collezione di comunicazioni tra le istanze parzialmente ordinate nel tempo. Un messaggio è una comunicazione tra istanze e può causare l'invocazione di una operazione, l'emergere di un segnale (asincrono), la creazione o la distruzione di una istanza. All'interno dei diagrammi di sequenza ci

sono due dimensioni: una verticale, rappresentate il tempo; una orizzontale, rappresentante le diverse istanze coinvolte nell'interazione.
Ogni istanza coinvolta nell'interazione ha una lifeline verticale (una linea tratteggiata sotto il simbolo dell'istanza che rappresenta l'esistenza di quell'istanza in un certo punto temporale). I messaggi sono rappresentati come frecce orizzontali/oblique, corredate di una etichetta. Le etichette contengono l'operazione o il segnale invocato. Le istanze possono essere create o distrutte, come si può intuire dal fatto che le lifeline inizino e terminino.

Ad un livello di astrazione molto alto i diagrammi di sequenza possono essere utilizzati per modellare degli *use case*, cioè intendendo l'interazione come uno *use case* tra gli attori e il sistema stesso. I diagrammi di sequenza possono essere raffinati ulteriormente durante il processo di sviluppo mano a mano che l'interazione diventa più complessa.

Quando vengono utilizzati in un contesto *object oriented*, le istanze menzionate sopra sono normalmente istanze di classi, cioè gli oggetti che esisteranno a runtime nel sistema. In un contesto ad agenti i diagrammi di sequenza possono essere utilizzati per modellare le interazioni tra gli agenti, oppure, ad un livello di specifica, tra i ruoli.

1.2.2 Diagrammi di sequenza

In AUML i diagrammi di sequenza vengono estesi per catturare due aspetti fondamentali e distintivi degli agenti:

- ruoli: gli agenti possono ricoprire uno o più ruoli durante la loro vita e, in particolare, possono cambiare ruolo dinamicamente

- concorrenza: gli agenti possiedono una concorrenza interna che permette loro di portare avanti più task contemporaneamente, come per esempio partecipare a più conversazioni con diversi agenti distinti

Conseguentemente, i diagrammi di sequenza sono stati estesi per supportare ruoli e processi concorrenti multipli. Tutta la comunicazione è asincrona, come indicato dalle frecce stilizzate. La comunicazione sincrona, indicata da una freccia con la punta piena, tra agenti, è comunemente evitata.

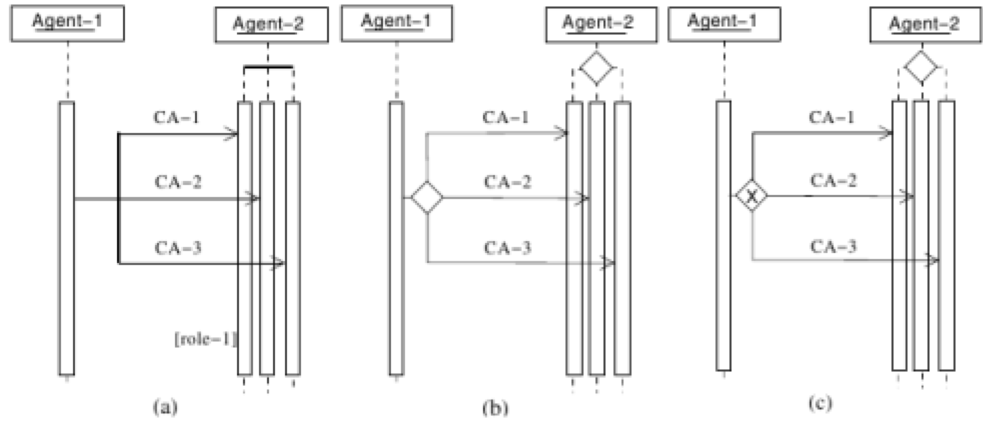

Figura 1.3: Differenti notazioni per la comunicazione tra agenti

(a) **Agent-1** che invia messaggi multipli concorrenti ad **Agent-2**. I messaggi inviati denotano gli atti comunicativi **CA-1**, **CA-2** e **CA-3**. **Agent-2** elabora questi messaggi parallelamente, infatti **Agent-2** potrebbe ricoprire ruoli differenti contemporaneamente, ognuno dei quali potrebbe elaborare un messaggio indipendentemente.

(b) I rombi indicano una *decision box* che permette ad **Agent-1** di decidere se inviare 0 o più (in questo caso 3 al massimo) messaggi ad **Agent-2**. Se viene inviato più di un messaggio allora questi sono elaborati parallelamente. Ancora **Agent-2** potrebbe ricoprire più ruoli contemporaneamente.

(c) I rombi contenenti una X rappresentano scelte esclusive. **Agent-1** decide se inviare solo 1 dei possibili messaggi ad **Agent-2**. La scelta non è deterministica.

C'è una notazione alternativa per ognuno dei casi prima esemplificati che risulta comoda nel caso in cui siano combinate diverse comunicazioni nello steso diagramma. Nella figura viene mostrato il caso (b) precedente in questa notazione alternativa.

Tale notazione però risulta ambigua e, possibilmente, fuorviante. Visto che le barre di attivazione sono poste lungo la lifeline perdiamo l'intuibilità visiva che gli eventi di **Agent-2** corrispondenti alla ricezione di **CA-1** e **CA-2** avvengano parallelamente alla ricezione di **CA-3** e che possano appartenere a ruoli differenti, o che possano essere in conflitto e non appartenere alla medesima esecuzione. Potremmo essere erroneamente portati a presumere che tali eventi accadano sequenzialmente.

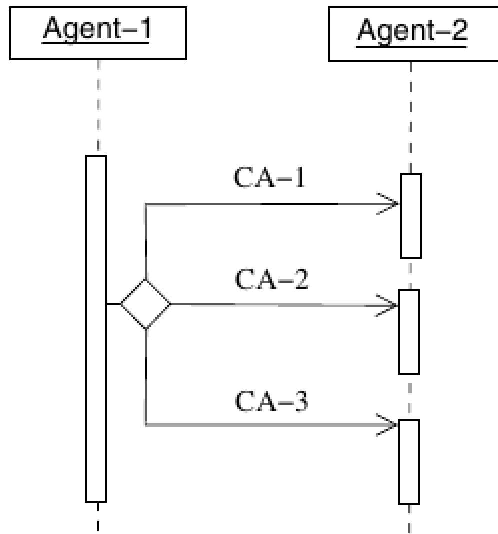

Figura 1.4: Notazione alternativa OR inclusivo

1.3 MIP-Net

Il comportamento di un agente può essere modellato come un processo di lavoro composto da un set di task coordinati da essere eseguiti dall'inizio alla fine. In particolare una classe di reti di Petri, chiamate *workflow net* (WF-Net), è stata adattata per modellare il comportamento di agenti [4].

1.3.1 A-Net

Una rete di Petri che modella il comportamento di un agente viene chiamata *agent net* (A-Net). Una rete di Petri **A=(P, T, F)** è una *agent net* se e solo se:

1. ha due place particolari **in** e **out**, dove **•in**=∅ e **out•**=∅

2. aggiungendo una transizione **t*** ad **A**, la rete cortocircuitata (**P**, **T**∪{**t***}, **F**∪{**(out,t*),(t*,in)**}) risulta *fortemente connessa*

Nel contesto dei sistemi ad agenti, la prima clausola dichiara che un agente ha uno stato iniziale e uno finale; la seconda invece assicura che tutti i place e le transizioni contribuiscano al comportamento generale dell'agente [4].

1.3.2 IP-Net

Un protocollo di interazione in AUML può essere tradotto in un modello che utilizza le reti di Petri chiamato *interaction protocol net* (IP-Net). Una rete di Petri **IP=(P, T, F)** è una *interaction protocol net* se e solo se:

1. **IP** ha una transizione speciale $\mathbf{t}^{in} \in \mathbf{T}$ chiamata *protocol input transition* tale che $\bullet\mathbf{t}^{in}=\varnothing$

2. **IP** ha uno speciale set di transizioni $\mathbf{T}^{out}$ tale che per ogni $\mathbf{t} \in \mathbf{T}^{out}$, $\mathbf{t}\bullet=\varnothing$ e $\mathbf{t}^{in} \notin \mathbf{T}^{out}$

3. ci sono due classi disgiunte di transizioni $\mathbf{T}^{\alpha} \subset \mathbf{T}$ e $\mathbf{T}^{\beta} \subset \mathbf{T}$ tali che $\mathbf{t}^{in} \subset \mathbf{T}^{\alpha}$ e $\mathbf{T}^{out} \subset (\mathbf{T}^{\alpha} \cup \mathbf{T}^{\beta})$

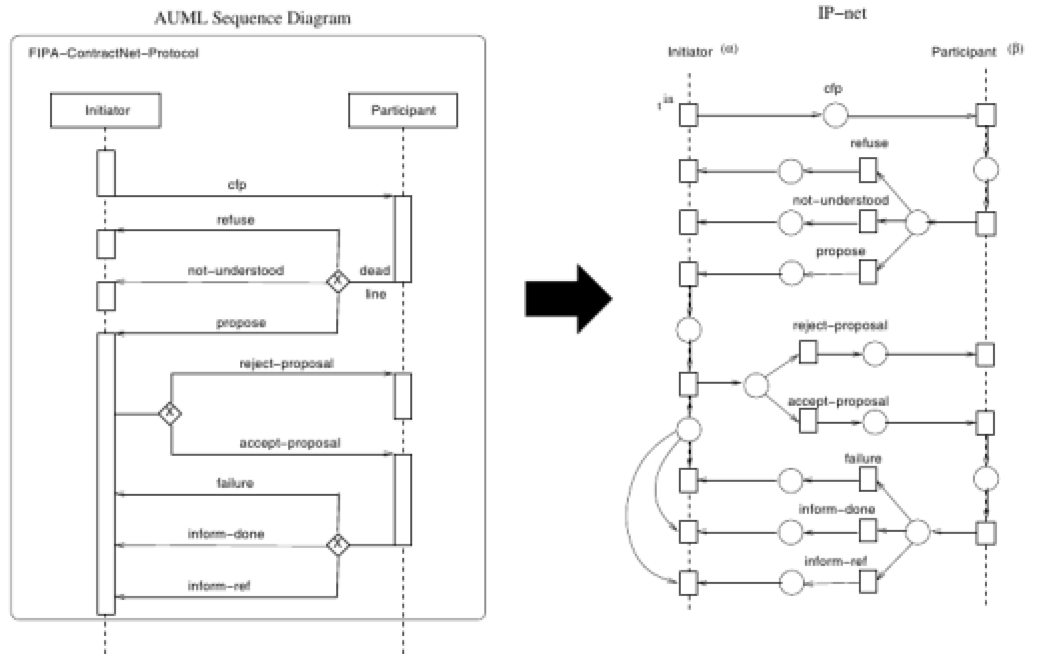

Figura 1.5: Contract Protocol in AUML e in rete di Petri

Nella definizione, la transizione di input $\mathbf{t}^{in}$ effettivamente inizia un protocollo, con l'idea che ogni protocollo sia attivato dallo scattare di una singola transizione $\mathbf{t}^{in}$; non ci sono ulteriori elementi prima di $\mathbf{t}^{in}$. La IP-Net termina con una serie di transizioni $\mathbf{T}^{out}$ con nessun altro elemento dopo ogni **t** nel set. In ogni IP-Net si possono distinguere due set disgiunti di transizioni, ognuno dei quali relativo ad uno dei due agenti (α e β) coinvolti nel protocollo. Intuitivamente le transizioni in $\mathbf{T}^{\alpha}$ e in

$\mathbf{T}^\beta$ sono attivate dai rispettivi agenti attraverso sincronizzazione. Mentre la transizione $\mathbf{t}^{in}$ deve essere necessariamente di α, le $\mathbf{T}^{out}$ possono appartenere anche a β [4].

1.3.3 Sincronizzazione tra A-Net e IP-Net

Un sistema multiagente è composto da un numero di agenti e un numero di protocolli di interazione. Ogni coppia di agenti comunica attraverso un protocollo di interazione. Questa situazione può essere modellata tramite una *multiagent interaction protocol net* (MIP-Net) che è la combinazioni di A-Net e IP-Net utilizzando alcuni elementi di sincronizzazione. Una MIP-Net è una tupla $\mathbf{MIP=(A_1,\ A_2,\ \ldots,\ A_n,\ IP_1,\ IP_2,\ \ldots,\ IP_k,\ T_{SC},\ SC)}$ tale che:

1. $\mathbf{n,k}\in \mathbb{N}$, dove $\mathbf{n}$ è il numero di A-Net e $\mathbf{k}$ è il numero di IP-Net e $\mathbf{n\text{-}1 \leq k \leq n(n\text{-}1)/2}$
2. per ogni $\mathbf{i} \in \{\mathbf{1,\ \ldots,\ n}\}$, $\mathbf{A}_i$ è una A-Net con un place iniziale $\mathbf{in}_{Ai}$ e uno finale $\mathbf{out}_{Ai}$
3. per ogni $\mathbf{i} \in \{\mathbf{1,\ \ldots,\ n}\}$, $\mathbf{IP}_i$ è una IP-Net con una transizione di input iniziale $\mathbf{t}^{in}_{IPi}$ e i set di transizioni $\mathbf{T}^{\alpha}_{IPi}$ e $\mathbf{T}^{\beta}_{IPi}$
4. $\mathbf{T}_{SC}$ è il set degli elementi per la sincronizzazione (*fusion set*)
5. $\mathbf{SC}\subseteq(\mathbf{T}_{SC}\times\mathbf{T}^{\Diamond}\times\mathbf{T}^{\Box})$ è la legge di sincronizzazione:

   $\mathbf{T}^{\Box}=\mathbf{U}_{i\in\{1,\ldots,n\}}\mathbf{T}_{Ai}$ $\qquad$ $\mathbf{T}^{A}=\mathbf{U}_{i\in\{1,\ldots,k\}}\mathbf{T}^{\alpha}_{IPi}$

   $\mathbf{T}^{\Diamond}=\mathbf{T}^{A}\cup\mathbf{T}^{B}$ $\qquad$ $\mathbf{T}^{B}=\mathbf{U}_{i\in\{1,\ldots,k\}}\mathbf{T}^{\beta}_{IPi}$

6. per ogni $\mathbf{t}\in\mathbf{T}_{SC}$, $\{\mathbf{(t',\ x,\ y)\in SC}\ |\ \mathbf{t'=t}\}$ è un *singleton*
7. per ogni $[(\mathbf{t}_1,\ \mathbf{x}_1,\ \mathbf{y}_1),(\mathbf{t}_2,\ \mathbf{x}_2,\ \mathbf{y}_2)]\in\mathbf{SC}$, $\mathbf{t}_1 \neq \mathbf{t}_2 \Rightarrow (\mathbf{x}_1 \neq \mathbf{x}_2) \wedge (\mathbf{y}_1 \neq \mathbf{y}_2)$
8. per ogni $[(\mathbf{t}_1,\ \mathbf{x}_1,\ \mathbf{y}_1),(\mathbf{t}_2,\ \mathbf{x}_2,\ \mathbf{y}_2)]\in\mathbf{SC}$, se $\mathbf{y}_1$ e $\mathbf{y}_2$ sono della stessa A-Net allora $\mathbf{x}_1 \in \mathbf{T}^{\alpha}_{IP} \Rightarrow \mathbf{x}_2 \notin \mathbf{T}^{\beta}_{IP}$ per una particolare IP-Net $\mathbf{IP}$

Il requisito (1) definisce la relazione tra il numero di agenti e di protocolli nel sistema multiagente, ipotizzando che tutti gli agenti debbano prendere parte ad una comunicazione con almeno un altro agente del sistema. I requisiti (2) e (3) definiscono tutte le A-Net e IP-Net, rispettivamente. Una transizione chiamata $\mathbf{t}_{SC}$ nel set $\mathbf{T}_{SC}$ è il risultato della sincronizzazione o "fusione" di due transizioni, una dalla IP-Net e una

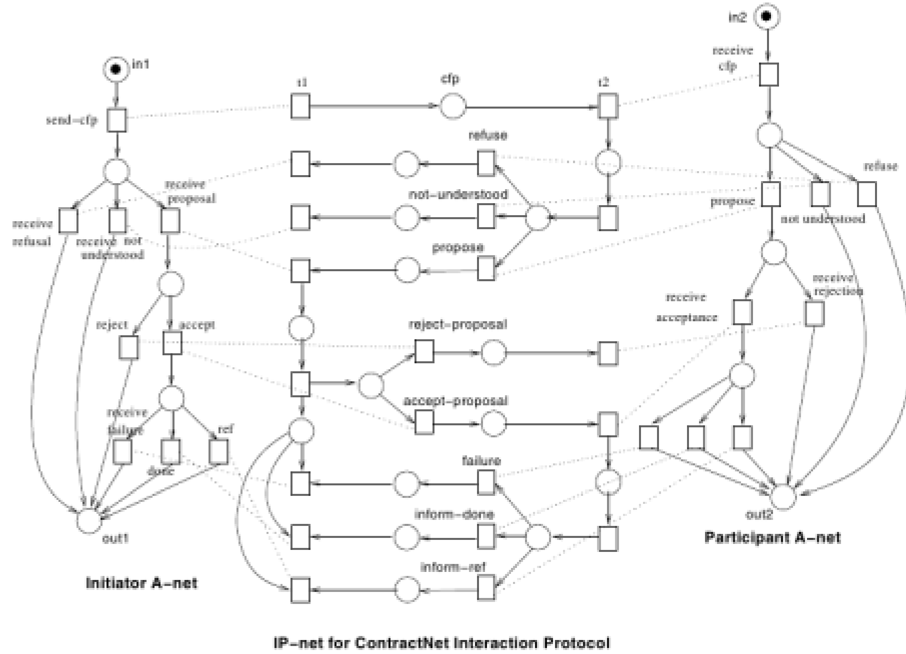

Figura 1.6: Sincronizzazione tra A-Net e IP-Net

dalla A-Net, ed è chiamata *elemento di comunicazione sincrona*. La relazione è definita da **SC** che è un set di triple, costituite dall'elemento di comunicazione sincrona e dalla coppia di transizioni "fuse". I requisiti (6) e (7) stabiliscono che per ogni elemento di comunicazione sincrona c'è un unico e solo elemento in **SC** e che la coppia delle transizioni "fuse" è anch'essa unica, cioè nessuna transizione che sia già stata "fusa" può essere coinvolta in altre relazioni sincrone. Il requisito (8) impone che ogni A-Net si possa sincronizzare in maniera mutualmente esclusiva o con il lato sinistro (α) o con il lato destro (β) di una particolare IP-Net [4].

1.3.4 *Soundness* dei sistemi

Uno dei motivi per modellare i sistemi multiagente attraverso delle reti di Petri è il poter utilizzare tutti i metodi già ben ampiamente affermati per l'analisi di queste. Tali metodi sono utilizzati per identificare le proprietà di *liveness* e *boundedness* dei sistemi modellati. Nei sistemi multiagente, le reti di Petri vengono utilizzate per fornire una qualche proprietà di "correttezza" e in seguito per riuscire ad analizzare i suddetti sistemi. Dal momento che le A-Net sono essenzialmente WF-Net modificate, la correttezza di una A-Net è definita in maniera simile a quella di una WF-

Net, attraverso la cosiddetta proprietà di *soundness*. Il comportamento di un agente modellato utilizzando una A-Net è sound se e solo se:

1. l'agente è in grado di completare il proprio task partendo dallo stato **in** e finendo in **out**

2. dopo il completamento, non c'è più alcun task che aspetta di essere completato nella A-Net

3. ogni task definito nella A-Net deve avere la possibilità di essere attivato e completato

La proprietà di *soundness* è strettamente collegata a quelle di *liveness* e *boundedness*. Per decidere se una rete **A=(P, T, F)** sia *sound*, si estende la rete **A** aggiungendo una ulteriore transizione che collega il place **out** al place **in**. A questo punto, la rete **A** è in grado di essere analizzata utilizzando i metodi classici per l'analisi delle proprietà di *liveness* e *boundedness* [4]. Il teorema stabilisce che un agente, o una A-Net **A**, è *sound*, se e solo se il sistema di reti di Petri **(A, in)** soddisfa le proprietà di *liveness* e *boundedness*. Le due A-Net nella figura precedente sono sound ma la combinazione delle stesse tramite un protocollo di interazione potrebbe incorrere in errori di sincronizzazione del protocollo. La MIP-Net risultante potrebbe quindi non essere "corretta".

1.3.5 *Dispiegamento* di una MIP-Net

Sia $\mathbf{MIP}=(\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n, \mathbf{IP}_1, \mathbf{IP}_2, \ldots, \mathbf{IP}_k, \mathbf{T}_{SC}, \mathbf{SC})$ una MIP-Net. $\mathbf{U(MIP)}=(\mathbf{P}^U, \mathbf{T}^U, \mathbf{F}^U)$ è il *dispiegamento* (*unfolding*) di **MIP**, dove:

1. $\mathbf{P}^U=(\mathbf{P}_{A1}\cup\mathbf{P}_{A2}\cup\cdots\cup\mathbf{P}_{An})\cup(\mathbf{P}_{IP1}\cup\mathbf{P}_{IP2}\cup\cdots\cup\mathbf{P}_{IPk})\cup\{\mathbf{i},\mathbf{o}\}$

2. $\mathbf{T}^U=\mathbf{r}(\mathbf{T}^\square\cup\mathbf{T}^\Diamond)\cup\mathbf{T}_{SC}\cup(\mathbf{U}_{j\in\{1,\ldots,k\}}\mathbf{T}_{IPj}\backslash\mathbf{T}^\Diamond)\cup\{\mathbf{t}_i,\mathbf{t}_o\}$

3. $\{\mathbf{i},\mathbf{o},\mathbf{t}_i,\mathbf{t}_o\}\cap\{(\mathbf{P}_{A1}\cup\mathbf{P}_{A2}\cup\cdots\cup\mathbf{P}_{An})\cup(\mathbf{P}_{IP1}\cup\mathbf{P}_{IP2}\cup\cdots\cup\mathbf{P}_{IPk})\cup\mathbf{T}_{SC}\cup(\mathbf{U}_{j\in\{1,\ldots,k\}}\mathbf{T}_{IPj}\backslash\mathbf{T}^\Diamond)\}=\varnothing$

4. **r** è la funzione di rinomina: $\mathbf{r(x)}=\mathbf{t}_{SC}$ se $\exists(\mathbf{t}_{SC}\in\mathbf{T}_{SC};\ \mathbf{x,y}\in\{\mathbf{T}^\square\cup\mathbf{T}^\Diamond\})\mid((\mathbf{t}_{SC},\mathbf{x},\mathbf{y})\in\mathbf{SC}$ o $(\mathbf{t}_{SC},\mathbf{y},\mathbf{x})\in\mathbf{SC})$; $\mathbf{r(x)}=\mathbf{x}$ altrimenti

5. $\mathbf{F}^\square=\mathbf{F}_{A1}\cup\mathbf{F}_{A2}\cup\cdots\cup\mathbf{F}_{An}\cup\mathbf{F}_{IP1}\cup\mathbf{F}_{IP2}\cup\cdots\cup\mathbf{F}_{IPk}$

6. $\mathbf{F'}=\mathbf{F}^\square\cup\{(\mathbf{i},\mathbf{t}_i),(\mathbf{t}_o,\mathbf{o})\}\cup\{(\mathbf{t}_i,\mathbf{in}_{Aj})\mid\mathbf{j}\in\{\mathbf{1},\ldots,\mathbf{n}\}\}\cup\{(\mathbf{out}_{Aj},\mathbf{t}_o)\mid\mathbf{j}\in\{\mathbf{1},\ldots,\mathbf{n}\}\}$

7. $\mathbf{F}^U$=**{(r(x),r(y)) | (x,y)∈F'}**

Il *dispiegamento* mira a creare una singola e globale A-Net dalla MIP-Net, introducendo due nuove transizioni ($\mathbf{t}_i$ e $\mathbf{t}_o$) e due nuovi place (**i** e **o**). Intuitivamente, i place **i** e **o** diventano rispettivamente i place **in** e **out** della A-Net globale. I requisiti (1), (2) e (3) introducono quattro nuovi elementi di rete. I requisiti (4), (5) e (6) stabiliscono che tutte le coppie di transizioni della IP-Net e della A-Net che subiranno sincronizzazione, saranno fuse e rinominate utilizzando la funzione di rinomina. Il risultato del dispiegamento è una singola rete di Petri che può essere analizzata come già detto in precedenza [4].

Una MIP-Net è *sound* se e solo se ogni A-Net nel sistema e il *dispiegamento* della MIP-Net sono anch'essi *sound*. La proprietà di *soundness* può essere verificata utilizzando i metodi di analisi per *liveness* e *boundedness* delle reti di Petri. Ricapitolando, per verificare che un sistema sia *sound* occorre:

1. costruire una A-Net per ogni agente e una IP-Net per ogni protocollo di interazione

2. combinare le A-Net e le IP-Net per formare una MIP-Net

3. effettuare il *dispiegamento* della MIP-Net

4. controllare la proprietà di *soundness* per ogni A-Net e per la MIP-Net dispiegata

1.3.6 A-Net *minimali*

Dato un protocollo di interazione, possiamo costruire delle A-Net *minimali* che soddisfino tale protocollo. Le A-Net *minimali* utilizzano il minor numero di transizioni *receive/send* per sincronizzarsi correttamente con la IP-Net. Tali A-Net sono derivate osservando direttamente la struttura della IP-Net e ne è garantita la *soundness* [4]. Il comportamento dell'agente, di fatto lo scheletro del codice, può essere quindi modificato cambiando o migliorando la A-Net aggiungendo transizioni e place; occorre però seguire regole di trasformazioni appropriate per garantire che i cambiamenti non intacchino la *soundness* dell'agente o dell'intero sistema, preservando il corretto ordinamento delle attività.

Capitolo 2

Traduzione e analisi

2.1 Algoritmo di traduzione

Il passo successivo da eseguire è l'analisi dei diagrammi AUML per 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 multiagente. Dopodiché un algoritmo apposito utilizzerà le informazioni estrapolate dall'analisi di questa rappresentazione testuale per creare la MIP-Net associata al protocollo.

2.1.1 Rappresentazione testuale di AUML

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 di tipo message passing asincrono. 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:

- **AND** - Un agente invia contemporaneamente tutti i CA coinvolti dall'operatore

- **OR** - Un agente può inviare uno o più dei CA coinvolti dall'operatore
- **XOR** - Un agente può inviare uno e uno solo dei CA coinvolti dall'operatore

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 [3].

Per una corretta rappresentazione testuale del diagramma di interazione è inoltre utile introdurre l'**operatore di sequenza** ($\rightarrow$), che indicherà una relazione temporale: ciò che si trova a sinistra dell'operatore, avviene necessariamente prima di ciò che si trova a destra.

2.1.2 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.

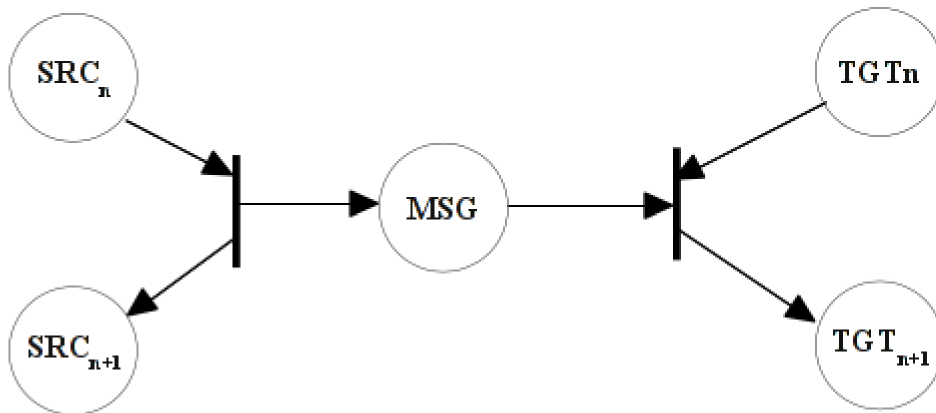

Figura 2.1: Porzione di rete di Petri relativa ad un CA

Supponendo di trovarsi nello stato con marking $< \mathbf{SRC}_n, \mathbf{TGT}_n >$, si transita nello stato $< \mathbf{SRC}_{n+1}, \mathbf{MSG}, \mathbf{TGT}_n >$, che rappresenta l'emissione di **MSG** da parte di **SRC**; dopodiché si può transitare nello stato $< \mathbf{SRC}_{n+1}, \mathbf{TGT}_{n+1} >$, che rappresenta l'avvenuta ricezione di

MSG da parte di **TGT**.

Utilizzando questo metodo è facile tradurre il comportamento definito dall'operatore $\rightarrow$ 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 $\rightarrow$ 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 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 $\rightarrow$ e **XOR** operando delle trasformazioni analoghe alle operazioni di trasformazione in algebra booleana [3]:

$$\mathbf{X\ and\ Y = (X \rightarrow Y)\ xor\ (Y \rightarrow X)}$$
$$\mathbf{X\ or\ Y = X\ xor\ Y\ xor\ (X \rightarrow Y)\ xor\ (Y \rightarrow X)}$$

Una trasformazione alternativa nel caso di un **OR** sarebbe stata:

$$\mathbf{X\ or\ Y = (X \rightarrow (Y\ xor\ \varnothing))\ xor\ (Y \rightarrow (X\ xor\ \varnothing))}$$

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

L'algoritmo di traduzione seguirà quindi i seguenti 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 $\rightarrow$ 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.

2.1.3 Algoritmo in `tuProlog`

Avendo a disposizione una rappresentazione testuale del diagramma AUML occorre costruire un parser che identifichi la struttura del protocollo ed esegua le trasformazioni necessarie. Per realizzare tale parser è stato scelto il linguaggio `tuProlog`, che si presta particolarmente bene alle operazioni di meta-programmazione, come in questo caso. 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.

Le prime due operazioni saranno l'eliminazione da queste frasi degli operatori **OR** e **AND** riscrivendoli in termini degli altri due operatori $\rightarrow$ 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,xor(B2,xor('->'(A2,B2),'->'(B2,A2))))):-
  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/2** 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 viene creato un fatto **state(A, N)** che verrà memorizzato in un elenco 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:

1. **transition (Name, [state(A,X)], [state(A,Y), Msg])**
2. **transition (Name, [state(A,X), Msg], [state(A,Y)])**
3. **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:

1. 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

2. 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**

3. la terza tripla invece rappresenta le transizioni utilizzate per completare la rete cos 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,ES):-
  netGen(Front,Prot,Net,[],Front,_,_,ES,Front).

  ⋮

netGen(Front,[],Net,Net,Finals,Front,Finals,ES,ES).
```

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

- **ES**, inizializzato con i valori della frontiera, conterrà alla fine tutti gli stati finali della rete

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

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, le variabili ausiliare vengono inoltrate senza subire cambiamenti.

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

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**, **Assigned** e **TempES** vengono utilizzati per elaborare il sotto-albero **A**, mentre per elaborare il sotto-albero **B** verranno utilizzati i valori **NewFront**, **NewAssign** e **NewTempES** raggiunti dopo l'elaborazione di **A**; i valori raggiunti dopo l'elaborazione di **B** saranno quelli che verranno unificati con le variabili **Next**, **Finals** e **ES** nella testa della clausola.

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

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**, **Finals** e **ES** nella testa della clausola.

Quelle di seguito sono le clausole che generano le transizioni arrivati alla fine di un sotto-albero, sono quindi le clausole che modificano le variabili **Front**, **Assigned** e **ES** generando la nuova frontiera, aggiornando gli indici da utilizzare per generare i place associati alle varie entità e aggiornando gli stati finali sostituendo quelli da cui ci si è spostati con quelli raggiunti.

```
netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals,ES,TempES):-
  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),
  deleteOne(state(SRC,X0),TempES,Temp),
  deleteOne(state(TGT,Y0),Temp,NewTempES),
  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,ES,
    [state(SRC,X1),state(TGT,Y1)|NewTempES]).
```

Questa clausola esegue, in ordine, le seguenti operazioni:

1. genera un nome univoco per identificare il messaggio scambiato
2. identifica gli stati attuali delle entità coinvolte nell'interazione recuperandoli dalla frontiera
3. aggiorna i valori degli indici di stato utilizzati tramite la clausola **inc/3** che incrementa l'indice dell'entità specificata
4. questi nuovi valori vengono recuperati e utilizzati per aggiornare la frontiera tramite la clausola **setState/4** che sostituisce il valore dell'indice dell'entità specificata con un nuovo valore

5. vengono aggiornati gli stati finali rimuovendo gli stati precedenti tramite la clausola **deleteOne/3** che elimina la prima occorrenza di un elemento da una lista

6. vengono generati i nomi univoci delle due transizioni da generare

7. viene richiamata la clausola **netGen/9** passando i parametri aggiornati, comprese le due nuove transizioni e i due nuovi stati finali

Teoricamente l'algoritmo di generazione della rete di Petri, a questo punto, sarebbe completo; l'algoritmo, in questa forma primitiva tuttavia, non è in grado di distinguere, tra i vari sotto-alberi generati, i casi in cui si stiano percorrendo sotto-alberi analoghi. Questo risulta particolarmente indesiderato nel caso in cui il protocollo in esame faccia uso di molti operatori **AND** e **OR** che causano lo sviluppo di un numero enorme di stati alternativi, che però, nella maggior parte dei casi, risultano solamente ripetizioni del medesimo comportamento. Per ovviare a questa esplosione di stati, sono state aggiunte le seguenti clausole:

```
netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals,ES,TempES):-
  member(state(TGT,Y0),Front), member(state(SRC,X0),Front),
  member(transition(_,[state(TGT,Y0),MsgName],
    [state(TGT,OldY)]),TS),
  member(transition(_,[state(SRC,X0)],
    [state(SRC,OldX),MsgName]),TS), !,
  setState(Front,SRC,OldX,TempFront),
  setState(TempFront,TGT,OldY,NewFront),
  deleteOne(state(SRC,X0),TempES,Temp),
  deleteOne(state(TGT,Y0),Temp,NewTempES),
netGen(NewFront,[],Net,TS,Assigned,Next,Finals,ES,
  [state(SRC,OldX),state(TGT,OldY)|NewTempES]).

netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals,ES,TempES):-
  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]),
  deleteOne(state(SRC,X0),TempES,Temp),
  deleteOne(state(TGT,Y0),Temp,NewTempES),
  netGen(NewFront,[],Net,
    [transition(TName,[state(SRC,X0)],[state(SRC,X1),MsgName])|TS],
    NewAssign,Next,Finals,ES,
    [state(SRC,X1),state(TGT,OldY)|NewTempES]).
```

```
netGen(Front,ca(SRC,TGT,MSG),Net,TS,Assigned,Next,Finals,ES,TempES):-
  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]),
  deleteOne(state(SRC,X0),TempES,Temp),
  deleteOne(state(TGT,Y0),Temp,NewTempES),
  netGen(NewFront,[],Net,
    [transition(TName,[state(TGT,Y0),MsgName],[state(TGT,Y1)])|TS],
    NewAssign,Next,Finals,ES,
    [state(SRC,OldX),state(TGT,Y1)|NewTempES]).
```

Queste ultime clausole hanno un comportamento pressoché analogo alla clausola di generazione delle transizioni illustrata precedentemente; l'unica differenza è che prima di generare effettivamente le transizioni, queste clausole controllano che tali transizioni non esistano già, che si stia percorrendo, cioè, un sotto-albero uguale ad uno già generato. In questo caso vengono generate solo le transizioni necessarie alla modellazione di un comportamento nuovo (in alcuni casi, nessuna) e le variabili vengono aggiornate coi valori delle transizioni già esistenti. Tutto il resto rimane lo stesso. Anche qui il **cut** serve ad evitare ulteriori unificazioni con la clausola di generazione generica.

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([S|SS],Net,[T|TT]):-
  closingTransition(S,T), completeNet(SS,Net,TT).
completeNet([],Net,Net).
closingTransition(state(A,N),
  transition(TName,[state(A,N)],[state(A,0)])):-
    atom_chars(TName,[A,N,"_",to,"_",A,0]).
```

Il primo argomento contiene la lista degli stati finali (identificata nelle clausole precedenti con la variabile **ES**), il secondo argomento contiene invece la rete realizzata utilizzando l'algoritmo di cui sopra (identificata nelle clausole precedenti con la variabile **Net**). Per ogni stato finale viene realizzata una transizione che permette il transito da questo stato allo stato **0** dell'entità specifica.

2.2 Analisi delle proprietà

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

Dovendo verificare tali proprietà, per rappresentare la rete di Petri generata automaticamente, è stato scelto il programma `Maude`. Questo programma, oltre a permettere una facile e veloce rappresentazione di una rete di Petri, fornisce anche una serie di strumenti di *model checking* che è possibile utilizzare per verificare le proprietà sopracitate utilizzando LTL.

2.2.1 Analisi tramite LTL

L'analisi di queste due proprietà non è banale, specialmente volendo utilizzare delle proprietà LTL; per fortuna le reti che devono essere analizzate non sono reti di Petri generiche, ma delle reti di Petri derivate da un diagramma di sequenza AUML, fatto che garantisce determinate proprietà strutturali che rendono il processo di analisi delle proprietà molto più semplice.

Boundedness

La proprietà di *boundedness* stabilisce che per ogni possibile stato, raggiungibile a partire da quello iniziale, in ogni place della rete ci siano al più un certo numero finito di token. Supponendo che questo numero sia $\mathbf{k}$, la proprietà di *k-boundedness*, potrebbe essere verificata tramite un analisi esaustiva di tutti i possibili stati della rete in cerca di stati che abbiano $\mathbf{k}_0 > \mathbf{k}$ token; nel caso questo avvenisse, allora la proprietà sarebbe violata.

In questo caso il numero $\mathbf{k}$ non è noto a priori, la proprietà di *boundedness* garantisce però che la rete abbia un numero finito di stati; la verifica della proprietà di *boundedness* potrebbe quindi essere eseguita controllando che la rete abbia effettivamente un numero finito di configurazioni. Dunque, ipotizzando che sia possibile, a partire da un marking ammissibile $\mathbf{M}_1$, caratterizzato dall'insieme di token $\mathbf{T}_1$, raggiungere un marking $\mathbf{M}_2$, caratterizzato dall'insieme di token $\mathbf{T}_2$, tale che $\mathbf{T}_1 \subset \mathbf{T}_2$, allora $\mathbf{T}_2$=$\mathbf{T}_1 \cup \mathbf{T}_x$ con $\mathbf{T}_x \neq \varnothing$. 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 *boundedness*.

Il fatto che la rete sia stata realizzata a partire da un diagramma di sequenza AUML garantisce inoltre che, per ogni entità, non esistano transizioni con archi in uscita da uno stato $\mathbf{i}$ e in entrata in uno stato $\mathbf{j}$, a meno che $\mathbf{i} < \mathbf{j}$ oppure che $\mathbf{j} = \mathbf{0}$. Questo implica che a livello di rete globale, a partire da un marking ammissibile qualsiasi, esista sempre una sequenza finita di transizioni, che permetta di raggiungere un marking $\mathbf{M}_C$ tale che $\mathbf{M}_0 \subseteq \mathbf{M}_C$.

Fatte queste considerazioni, il controllo sul rispetto della proprietà di *boundedness* può essere eseguito ricercando tra tutti i possibili stati raggiungibili dalla rete a partire dal marking iniziale $\mathbf{M}_0$, lo stato indesiderato dato da un marking $\mathbf{M}_C$ tale che $\mathbf{M}_0 \subset \mathbf{M}_C$, cioè per cui identificando con $\mathbf{T}_0$ l'insieme dei token del marking $\mathbf{M}_0$ e con $\mathbf{T}_C$ l'insieme dei token del marking $\mathbf{M}_C$, risulti che $\mathbf{T}_C$=$\mathbf{T}_0$∪$\mathbf{T}_x$ con $\mathbf{T}_x \neq \varnothing$.

Liveness

Prima di procedere all'analisi della *liveness*, occorre fare due ipotesi:

- la rete è *connessa*:
  detto $\mathbf{O}$ l'insieme di tutti i place della rete, $\mathbf{P}$ e $\mathbf{Q}$ due sottoinsiemi di $\mathbf{O}$ tali che $\mathbf{O}=\mathbf{P}\cup\mathbf{Q}$ e $\mathbf{P} \neq \varnothing$ e $\mathbf{Q} \neq \varnothing$, allora esiste una transizione che ha un arco in uscita da un place $\mathbf{p}\in\mathbf{P}$ e arco in ingresso a un place $\mathbf{q}\in\mathbf{Q}$, per qualsiasi coppia $\mathbf{P}$, $\mathbf{Q}$

- la rete è *bounded*:
  dato $\mathbf{E}=\{\mathbf{E}_1, \mathbf{E}_2, \ldots, \mathbf{E}_{n-1}, \mathbf{E}_n\}$ insieme delle entità della rete, $\mathbf{A}$ e $\mathbf{B}$ due sottoinsiemi di $\mathbf{E}$ tali che $\mathbf{E}=\mathbf{A}\cup\mathbf{B}$ e $\mathbf{A}\cap\mathbf{B}=\varnothing$, se esiste una transizione con un arco in uscita da un place $\mathbf{p}_{Ei} \in \mathbf{A}$ ed un arco in ingresso a un place $\mathbf{q}_{msg} \notin \mathbf{A}$, allora esiste anche una transizione con un arco in uscita da un place $\mathbf{p}_{msg} \notin \mathbf{A}$ ed un arco in ingresso a un place $\mathbf{q}_{Ej} \in \mathbf{A}$, per qualsiasi coppia $\mathbf{A}$, $\mathbf{B}$

Nella rete, escluse le transizioni banali che dagli stati finali permettono di tornare a quelli iniziali, esistono solo due tipi di transizione: quelle di un CA di invio e quelle di un CA di ricezione.

Avere una transizione *non-live* significa che tra tutti i possibili marking della rete, ne esiste almeno uno, da cui non è più possibile raggiungere una configurazione della rete tale da poter utilizzare la transizione. Supponendo di aver già raggiunto tale marking, l'entità che cerchi di utilizzare la transizione *non-live* rimarrà bloccata nello stato in cui si trova e con essa rimarrà bloccata anche qualsiasi altra entità, che cerchi

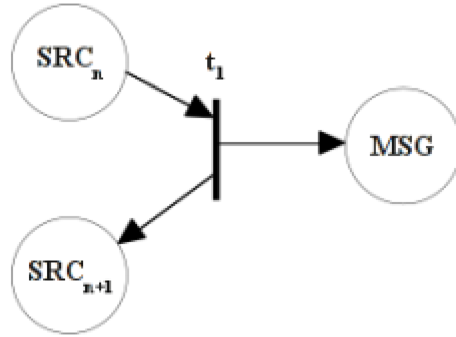

Figura 2.2: CA di Invio

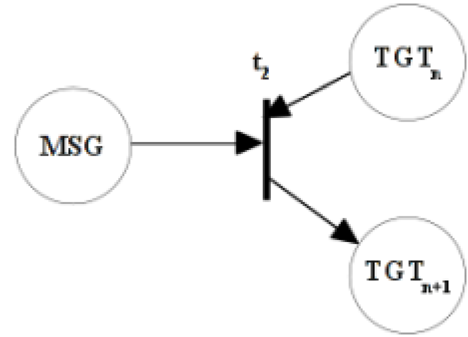

Figura 2.3: CA di ricezione

di usare una transizione di tipo $\mathbf{t}_2$ con la complementare transizione $\mathbf{t}_1$ in possesso dell'entità precedentemente bloccata. Per le ipotesi iniziali di *boundedness* e *connessione*, questo blocco si ripercuoterà su tutte le entità coinvolte nel protocollo, fino al raggiungimento di uno stato di *deadlock* totale di tutte le entità e, quindi, della rete.
Identificando con $\mathbf{T}_{pre}$ l'insieme dei token che devono trovarsi nei place da cui escono gli archi in ingresso ad una transizione generica $\mathbf{t}$ e con $\mathbf{T}_i$ l'insieme dei token corrispondenti al marking generico $\mathbf{M}_i$, allora

$$\textbf{se la rete è } \textbf{\textit{live}} \Rightarrow \forall\ \mathbf{M}_i\ \exists\ \mathbf{t}\ |\ \mathbf{T}_{pre} \subseteq \mathbf{T}_i$$

Fatte queste considerazioni, il controllo sul rispetto della proprietà di *liveness* può essere eseguito verificando che in tutti i possibili stati raggiungibili dalla rete, esista sempre almeno una transizione che può essere utilizzata.

2.2.2 Rappresentazione della rete in `Maude`

Il modo più semplice di rappresentare la rete è tramite l'insieme delle sue transizioni. Una transizione può essere rappresentata tramite un nome, il set di place con archi entranti e il set di place con archi uscenti.

```
fmod PETRI-NET is
  pr QID .
  sorts P T MT MP Marking .
  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 -> Marking .
  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 come set di transizioni. L'operatore per la definizione di **Marking** è pensato per contenere nel parametro l'insieme dei token della rete (marking), identificati ognuno tramite un **Qid** con il nome dello stato in cui si trovano.

A questo punto sono necessarie delle regole che permettano alla rete di evolversi correttamente modificando il marking in **Marking** coerentemente con le transizioni che compongono la rete.

```
mod PETRI-NET-ENGINE is
  pr PETRI-NET .
  var TS : MT .
  var TName : Qid .
  var MP MP1 MP2 : MP .
  crl [fire] : < MP1 MP > => < MP2 MP > if
    [ TName @ MP1 , MP2 ] TS := net .
endm
```

Le *rewriting rules* permettono alla rete di passare da un marking a un altro nel caso in cui un sottoinsieme di token presenti coincida con il set di ingresso di una data transizione, in quel caso, il marking viene modificato sostituendo tali token con i rispettivi nel set di uscita.

2.2.3 Model checking

Di seguito viene mostrato come le considerazioni sulle proprietà della MIP-Net vengano applicate per verificare la *soundness* della rete generata, utilizzando gli strumenti che `Maude` mette a disposizione.

Boundedness

Per identificare gli stati indesiderati invalidanti la *boundedness* è possibile utilizzare una funzione di `Maude` chiamata **search** nel modo seguente:

```
search [2] < InitState > =>+ < InitState Tokens:MP > .
```

La funzione **search** così utilizzata cercherà un marking costruito a partire da quello iniziale, qui identificato da **InitState**, con l'aggiunta ulteriore di un set di token. La ricerca restituirà tutti i possibili assegnamenti di **Tokens** che soddisfano i criteri di ricerca. Nel caso che la rete sia *bounded*, la ricerca restituirà solamente l'assegnamento **nil**. Il parametro [**2**] di **search** obbliga la ricerca a fermarsi dopo aver trovato al massimo due soluzioni, così da terminare nel caso di una rete con un numero infinito di stati e contemporaneamente assicurarsi di eliminare l'assegnamento **nil** che potrebbe verificarsi anche in caso di una rete *unbounded*.

Liveness

Per la verifica della *liveness* si può utilizzare una funzionalità del model checker di `Maude` chiamata **reduce** abbinata alla funzione **modelCheck** nel seguente modo:

```
reduce modelCheck( < InitState > , [] no-deadlock ) .
```

Questa istruzione permette di esaminare tutti i possibili stati della rete raggiungibili a partire da **InitState** e verificare che la proprietà **no-deadlock** sia sempre (□) verificata. La proprietà **no-deadlock** è definita come segue:

```
mod NO-DEADLOCK is
  pr PETRI-NET .
  pr SATISFACTION .
  pr MODEL-CHECKER .
  pr LTL-SIMPLIFIER .
  subsort Marking < State .
  var TName : Qid .
  var MP MP1 MP2 : MP .
  var MT : MT .
  op no-deadlock : -> Prop .
  eq < MP > |= no-deadlock = confront(MP) .
  op confront(_) : MP -> Bool .
  ceq confront(MP1 MP) = true if
    [ TName @ MP1 , MP2 ] MT := net .
endm
```

1-boundedness

Perché una rete di Petri sia *1-bounded* occorre che in ogni possibile stato raggiungibile a partire da quello di partenza, ci sia al massimo 1 token in ogni place. La 1-boundedness non è necessaria per definire la *soundness* di una rete, ma potrebbe essere utile lo stesso poter discriminare tra reti

safe o *unsafe*.

Per eseguire il controllo per la *1-boundedness* si procede analogamente a quello per la *liveness*, utilizzando le funzioni **reduce** e **modelCheck** nel modo seguente:

```
reduce modelCheck( < InitState > , [] 1-bounded ) .
```

Come nel caso precedente, verifichiamo che per ogni stato raggiungibile a partire da **InitState**, sia sempre ($\Box$) verificata la proprietà **1-bounded** definita come segue:

```
mod 1-BOUNDEDNESS is
  pr PETRI-NET .
  pr SATISFACTION .
  pr MODEL-CHECKER .
  pr LTL-SIMPLIFIER .
  subsort Marking < State .
  var MP : MP .
  var P : P .
  op 1-bounded : -> Prop .
  eq < MP > |= 1-bounded = check(MP) .
  op check(_) : MP -> Bool .
  eq check(P P MP) = false .
  eq check(MP) = true [owise] .
endm
```

La proprietà non fa altro che controllare che per ogni marking non vi siano due token uguali, cioè all'interno dello stesso place, qui identificati dalla variabile **P**.

Capitolo 3

Implementazione su `Jason`

3.1 La piattaforma `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 multiagente. `AgentSpeak(L)` è una estensione alla programmazione logica delle architetture ad agenti BDI e fornisce un supporto alla programmazione di tali agenti [7].

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 g*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 dallo stesso manuale tecnico di Jason chiamato "*Collecting Garbage*" dove due robot si trovano sulla superficie di Marte [7]. 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*) e 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
[p1]    +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.

3.2 MIP-Net e 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 MIP-Net e il modello di un agente Jason che implementi il protocollo espresso da questa rete.

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

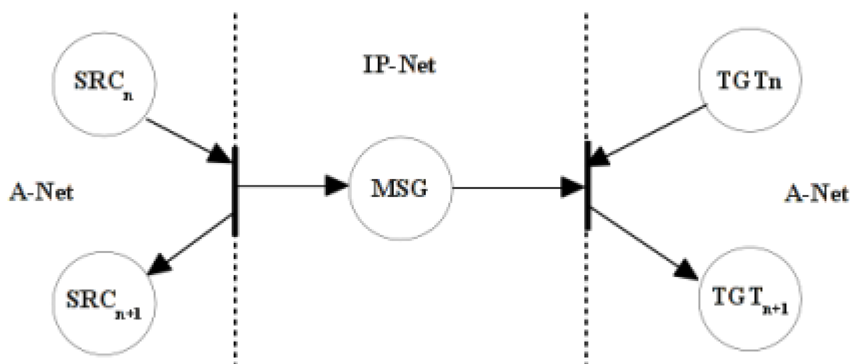

Figura 3.1: Suddivisione della MIP-Net

Tutta la MIP-Net 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 A-Net dell'agente **SRC** si ha un plan costruito secondo il seguente schema:

```
+!living : state(X) <- .send(TGT,tell,MSG);
                       -state(X);
                       +state(Y);
                       !living.
```

La precondizione per l'esecuzione del plan è di aver messo in esecuzione **!living**, plan associato al processo di vita dell'agente. La guardia impone di trovarsi nello stato **X** (identificato qui da un belief **state(X)**) per poter eseguire il plan. I sotto-goal eseguiti dal plan inviano **MSG** all'agente **TGT** (qui utilizzando l'azione interna standard **send/3**) e modificano di conseguenza la base di conoscenza, passando dallo stato **X** allo stato **Y**. Avendo portato a compimento **!living**, il plan viene di nuovo richiamato per la prossima esecuzione.

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

```
+!living : MSG[source(SRC)] & state(X) <- -MSG[source(SRC)];
                                          -state(X);
                                          +state(Y);
                                          !living.
```

La precondizione per l'esecuzione del plan è anche qui di aver messo in esecuzione **!living**. In questo caso la guardia non è costituita dal solo trovarsi nello stato **X**, ma anche di aver ricevuto dall'agente **SRC** il messaggio **MSG**, che ci si aspetta di ricevere proprio nello stato **X**. I sotto-goal modificano la base di conoscenza eliminando l'avvenuta ricezione di **MSG**, spostandosi dallo stato **X** allo stato **Y** e **!living** viene di nuovo messo in esecuzione.

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:

```
+!living : state(X) <- -state(X);
                       +state(0);
                       !living.
```

La precondizione del plan è ancora di aver messo in esecuzione **!living**. La guardia è quella di trovarsi nello stato **X**, qui ipoteticamente identificato con uno degli stati finali dell'agente. I sotto-goal eseguiti modificano la base di conoscenza interna tornando dallo stato **X** allo stato **0**. Ancora una volta viene rimesso in esecuzione **!living**.

Nel caso in cui nessun plan precedentemente definito possa essere messo in esecuzione viene aggiunto un plan ulteriore di default:

```
+!living : true <- !living.
```

Questo plan permette di continuare ad eseguire **!living**, qualora non sia applicabile nessun plan in quel particolare momento di vita dell'agente.

A questo punto il comportamento è completamente definito, occorre però aggiungere altre due piccole cose per completare l'agente `Jason`:

```
state(0).
!living.
```

Questi saranno i belief iniziali dell'agente, cioè di trovarsi nello stato iniziale **0** e di dover eseguire **!living**.

3.3 Caso di studio

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

```
(ca(a,x,q1) -> ca(x,a,r1)) and (ca(b,x,q2) -> ca(x,b,r2))
```

Il protocollo viene raffinato fino ad ottenere la forma contenente solo operatori $\rightarrow$ e **XOR**:

```
xor(
  '->'(
    '->'(ca(a,x,q1),ca(x,a,r1)),
    '->'(ca(b,x,q2),ca(x,b,r2))
    ),
  '->'(
    '->'(ca(b,x,q2),ca(x,b,r2)),
    '->'(ca(a,x,q1),ca(x,a,r1))
  )
)
```

A questo punto si procede come nel caso di creazione della MIP-Net, sostituendo alle coppie di transizioni associate ad ogni CA, le coppie di plan corrispondenti. L'algoritmo utilizzato è identico a quello per la creazione della MIP-Net, fatta eccezione per piccole modifiche non sostanziali per adattarlo alla creazione di sorgenti `Jason`, anziché `Maude`.

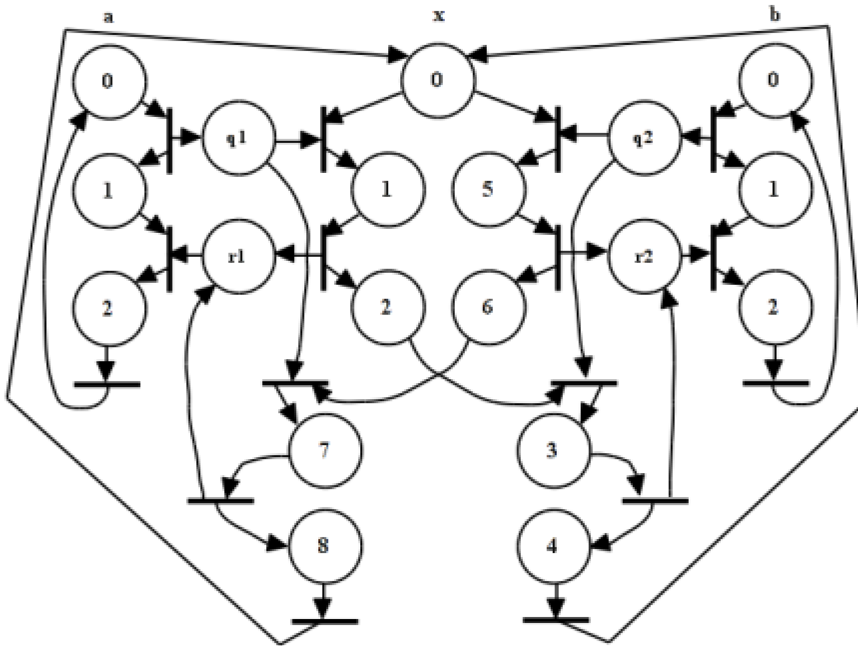

Figura 3.2: MIP-Net caso di studio

Il codice dell'agente `Jason` corrispondente ad **a** (e anche a **b**, essendo i loro comportamenti identici) è riportato di seguito:

```
      /* Initial beliefs and rules */

        state(0).
        !living.

      /* Plans */

[p0]    +!living : state(0) <- .send(x,tell,q1);
          -state(0); +state(1); !living.
[p1]    +!living : r1[source(x)] & state(1) <- -r1[source(x)];
          -state(1); +state(2); !living.
[p2]    +!living : state(2) <- -state(2); +state(0); !living.
        +!living : true <- !living.
```

Osservando la rappresentazione testuale delle transizioni corrispondenti della MIP-Net, la correlazione fra transizioni e plan è ancora più evidente:

- *p0* $\longleftrightarrow$ [a(0) $\longmapsto$ a(1) + q1]
- *p1* $\longleftrightarrow$ [a(1) + r1 $\longmapsto$ a(2)]
- *p2* $\longleftrightarrow$ [a(2) $\longmapsto$ a(0)]

I place con archi in ingresso alla transizione diventano le 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 guardia, o all'invio, nel caso di post-condizione, di una istruzione **send** da/verso l'agente interlocutore.

L'agente Jason corrispondente a **x** invece risulta più articolato, occupandosi della parte più significativa del protocollo:

```
      /* Initial beliefs and rules */

        state(0).
        !living.

      /* Plans */

[p0]    +!living : q1[source(a)] & state(0) <- -q1[source(a)];
          -state(0); +state(1); !living.
[p1]    +!living : state(1) <- .send(a,tell,r1);
          -state(1); +state(2); !living.
[p2]    +!living : q2[source(b)] & state(2) <- -q2[source(b)];
          -state(2); +state(3); !living.
[p3]    +!living : state(3) <- .send(b,tell,r2);
          -state(3); +state(4); !living.
```

```
[p4]    +!living : q2[source(b)] & state(0) <- -q2[source(b)];
          -state(0); +state(5); !living.
[p5]    +!living : state(5) <- .send(b,tell,r2);
          -state(5); +state(6); !living.
[p6]    +!living : q1[source(a)] & state(6) <- -q1[source(a)];
          -state(6); +state(7); !living.
[p7]    +!living : state(7) <- .send(a,tell,r1);
          -state(7); +state(8); !living.

[p8]    +!living : state(4) <- -state(4); +state(0); !living.
[p9]    +!living : state(8) <- -state(8); +state(0); !living.
        +!living : true <- !living.
```

Anche qui è possibile osservare la correlazione diretta tra i plan e le transizioni della MIP-Net:

$p0 \longleftrightarrow [\ \mathrm{x}(0) + \mathrm{q1} \longmapsto \mathrm{x}(1)\ ]$ $\qquad$ $p4 \longleftrightarrow [\ \mathrm{x}(0) + \mathrm{q2} \longmapsto \mathrm{x}(5)\ ]$

$p1 \longleftrightarrow [\ \mathrm{x}(1) \longmapsto \mathrm{x}(2) + \mathrm{r1}\ ]$ $\qquad$ $p5 \longleftrightarrow [\ \mathrm{x}(5) \longmapsto \mathrm{x}(6) + \mathrm{r2}\ ]$

$p2 \longleftrightarrow [\ \mathrm{x}(2) + \mathrm{q2} \longmapsto \mathrm{x}(3)\ ]$ $\qquad$ $p6 \longleftrightarrow [\ \mathrm{x}(6) + \mathrm{q1} \longmapsto \mathrm{x}(7)\ ]$

$p3 \longleftrightarrow [\ \mathrm{x}(3) \longmapsto \mathrm{x}(4) + \mathrm{r2}\ ]$ $\qquad$ $p7 \longleftrightarrow [\ \mathrm{x}(7) \longmapsto \mathrm{x}(8) + \mathrm{r2}\ ]$

$p8 \longleftrightarrow [\ \mathrm{x}(4) \longmapsto \mathrm{x}(0)\ ]$ $\qquad$ $p9 \longleftrightarrow [\ \mathrm{x}(8) \longmapsto \mathrm{x}(0)\ ]$

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*, corrisponda nella MIP-Net allo stato **x(0)**, in cui è possibile utilizzare una delle due transizioni che porterebbero negli stati **x(1)** o **x(5)** consumando rispettivamente il token **q1** o **q2**, come in effetti rispecchiato dalle guardie dell'agente `Jason`.

Capitolo 4

Implementazione su TuCSoN

4.1 Il meta-modello A&A

L'approccio principale per il coordinamento di sistemi multiagente (MAS) si fonda sul meta-modello A&A (agenti e artefatti), che adotta gli *artefatti*, assieme agli *agenti*, come entità di base per la costruzione di sistemi multiagente o, in generale, sistemi software complessi [6].
Nel meta-modello A&A, gli *agenti* sono le astrazioni base per rappresentare entità attive *task/goal oriented*, progettati per essere in grado perseguire pro-attivamente una attività con lo scopo di raggiungere un qualche obiettivo, sfruttando differenti livelli di capacità; gli *artefatti*, invece, sono le astrazioni base per rappresentare entità passive *function oriented* che sono costruite e utilizzate dagli *agenti* sia individualmente che cooperativamente. Al contrario degli *agenti*, gli *artefatti* non sono pensati per avere un comportamento pro-attivo.

Tra le proprietà principali di un *artefatto* vi sono:

- *ispezionabilità* e *controllabilità*: la capacità di osservare e controllare la struttura, lo stato e il comportamento di un artefatto, a runtime
- *malleabilità* (*forgiabilità*): la possibilità di cambiare la funzione di un artefatto in accordo a nuovi requisiti o eventi verificatisi durante l'esecuzione
- *collegabilità* (*linkability*): la possibilità di collegare assieme artefatti distinti in una composizione dinamica per far fronte ad una complessità di sistema scalante e promuovere il riuso
- *localizzazione* (*situation*): la capacità di reagire a eventi o cambiamenti dell'ambiente

La maggior parte di queste caratteristiche, inoltre, non sono caratteristiche di *agente*: un *agente* non è ispezionabile, non può essere modificato e non interagisce con altri *agenti* tramite link operazionali.

4.2 Il linguaggio `ReSpecT`

Il linguaggio `ReSpecT` si prefigge di superare alcune limitazioni intrinseche dell'uso degli *spazi di tuple* come medium di comunicazione, prima tra tutte, l'impossibilità di dividere la rappresentazione dell'informazione dall'informazione stessa.

Il linguaggio `ReSpecT` mantiene l'interfaccia standard di interazione con lo *spazio di tuple* e introduce la possibilità di arricchirne il comportamento con la specifica di particolari transizioni di stato in corrispondenza di eventi standard di comunicazione; da qui l'idea alla base dei *centri di tuple*, cioè *spazi di tuple* il cui comportamento in risposta agli eventi di comunicazione non è più fissato, ma può essere modificato a seconda delle esigenze di coordinazione.

Il comportamento di un centro di tuple è specificato tramite quello che viene definito *reaction specification language* che è in grado di associare a qualsiasi evento di comunicazione dello spazio di tuple, un set di attività computazionali chiamate *reazioni*. Ogni *reazione* può accedere e modificare lo stato corrente del centro di tuple e ha accesso a tutte le informazioni sull'evento di comunicazione che l'ha causata. Questo rende possibile, innanzitutto, disaccoppiare la percezione dell'informazione da parte dell'agente, dal modo in cui l'informazione è rappresentata realmente all'interno del centro di tuple, ma anche di essere in grado di specificare comportamenti impensabili per uno spazio di tuple, in grado di incorporare le leggi di coordinazione del sistema multiagente [1].

4.2.1 A&A in `ReSpecT`

Il linguaggio `ReSpecT` si è evoluto dal nucleo primitivo per essere in grado di perseguire il meta-modello A&A [6].

Alla possibilità di specificare delle *reazioni*, è stata aggiunta la possibilità di corredarle di *guardie*. Una *guardia* è una sequenza di condizioni circa un evento che necessitano di essere verificate prima che la *reazione* sia messa in atto. `ReSpecT` fornisce un set di guardie da utilizzare per discriminare tra la fase di pre-esecuzione o post-esecuzione di una invoca-

zione; tra una invocazione da parte di un agente o da parte del centro di tuple stesso (per mezzo di una *reazione*); verificare che un evento accada prima o dopo un istante temporale; etc.

Il modello A&A introduce inoltre la possibilità che un centro di tuple possa coesistere assieme ad altri *artefatti* nello stesso spazio concettuale. Questo implica, di fatto, la necessità di fornire ai centri di tuple degli strumenti per poter interagire con gli altri centri di tuple, così come ci si aspetterebbe che un *artefatto* sia in grado di interagire con altri *artefatti*. Sono state introdotte, quindi, operazioni utilizzabili dai centri di tuple per eseguire delle invocazioni su altri centri di tuple, sia nel proprio nodo di appartenenza, sia sparsi per la rete in un contesto distribuito, corredate di guardie per gestire le possibili *reazioni* in seguito a queste operazioni di *linking*.

Dining philosophers

Per illustrare meglio la struttura di un modello A&A in `ReSpecT` viene riportare di seguito il classico esempio dei *Dining philosophers* in cui N filosofi condividono N bacchette e un vassoio di spaghetti. Ogni filosofo ha bisogno di due bacchette per mangiare, ma ogni bacchetta è condivisa tra due filosofi adiacenti: le due bacchette devono essere acquisite e rilasciate atomicamente, rispettivamente, per evitare *deadlock* e garantire la *fairness*. In questo esempio vengono sfruttate tutte le capacità di A&A `ReSpecT` per implementarne una versione distribuita [6].

Ogni filosofo possiede un artefatto di coordinazione **seat(i, j)**, associato alle bacchette di indici **i** e **j**, localizzato nel suo stesso nodo. Quando il filosofo intende mangiare o pensare esprime l'intenzione emettendo la tupla **wanna_eat** o **wanna_think** nel proprio **seat(i, j)**. La gestione delle intenzioni e dello stato dell'agente e la disponibilità delle bacchette sono interamente a carico dell'artefatto.

La specifica di comportamento per il generico artefatto **seat(i, j)** è la seguente:

```
reaction( out(wanna_eat), (operation, invocation), (
  in(philosopher(thinking)), out(philosopher(waiting_to_eat)),
  current_target(seat(C1,C2)),
  table@node ? in(chops(C1,C2)) )
).
reaction( out(wanna_eat), (operation, completion),
  in(wanna_eat)
).
```

```
reaction( in(chops(C1,C2)), (link_out, completion), (
  in(philosopher(waiting_to_eat)), out(philosopher(eating)),
  out(chops(C1,C2)) )
).
reaction( out(wanna_think), (operation, invocation), (
  in(philosopher(eating)), out(philosopher(waiting_to_think)),
  current_target(seat(C1,C2)), in(chops(C1,C2)),
  table@node ? out(chops(C1,C2)) )
).
reaction( out(wanna_think), (operation, completion),
  in(wanna_think)
).
reaction( out(chops(C1,C2)), (link_out, completion), (
  in(philosopher(waiting_to_think)), out(philosopher(thinking)) )
).
```

Come si può vedere, l'agente è del tutto all'oscuro del fatto che le bacchette (identificate dalla tupla **chops(C1,C2)**) non siano affatto nel proprio **seat(i, j)**, né tanto meno nel proprio nodo; tutto questo è gestito tramite il meccanismo di *reazioni* dell'artefatto.

Per quanto riguarda le politiche di *fairness* e assenza di *deadlock*, invece, il controllo è rimandato all'artefatto di coordinazione **table**. L'artefatto **table**, contrariamente si **seat(i, j)**, rappresenta le tuple associate alle bacchette singolarmente tramite tuple del tipo **chop(i)**.

La specifica di comportamento per l'artefatto centrale **table** è la seguente:

```
reaction( out(chops(C1,C2)), (link_in, completion), (
  in(chops(C1,C2)),
  out(chop(C1)), out(chop(C2)) )
).
reaction( in(chops(C1,C2)), (link_in, invocation), (
  out(required(C1,C2)) )
).
reaction( in(chops(C1,C2)), (link_in, completion), (
  in(required(C1,C2)) )
).
reaction( out(required(C1,C2)), internal, (
  in(chop(C1)), in(chop(C2)),
  out(chops(C1,C2)) )
).
reaction( out(chop(C)), internal, (
  rd(required(C,C2))
  in(chop(C)), in(chop(C2)),
  out(chops(C,C2)) )
).
```

```
reaction( out(chop(C)), internal, (
  rd(required(C1,C))
  in(chop(C1)), in(chop(C)),
out(chops(C1,C)) )
).
```

L'acquisizione e il rilascio sono gestiti atomicamente sulle singole tuple **chop(i)** e **chop(j)** corrispondenti a **chops(i, j)** tramite le *reazioni* specificate; è l'artefatto **table** che si fa carico, inoltre, di controllare che entrambe le bacchette siano disponibili prima di rilasciarle all'agente che ne ha fatto richiesta ed, eventualmente, controllare quando una bacchetta viene rilasciata, per consegnarla ad un agente in attesa.

Come si nota, gli agenti sono completamente all'oscuro di qualsiasi politica di coordinazione, gestita interamente dagli artefatti da loro utilizzati tramite il meccanismo delle *reazioni*.

4.3 Primo modello: agenti complessi

Come primo approccio è stato realizzato un modello che ricalca la struttura del modello in `Jason`, lasciando incorporato negli agenti il controllo completo sul protocollo di interazione. In questo modello, i centri di tuple vengono utilizzati in maniera molto simile a come si utilizzerebbe un semplice spazio di tuple.

4.3.1 Gli agenti

Gli agenti di questo modello rispecchiano da vicino la struttura degli agenti `Jason` ed è possibile definire una relazione diretta tra le operazioni effettuate dall'agente `TuCSoN` e i *plan* dell'agente `Jason`.

Per prima cosa, l'agente dovrà mantenere traccia dello stato del protocollo. Analogamente al *belief* **state(N)** dell'agente `Jason`, avremo una variabile dove l'agente salverà l'indice dello stato corrente. Il modo più semplice di fare ciò, è utilizzando il seguente codice a livello di classe:

```
private int state;
```

La variabile verrà utilizzata dall'agente per discriminare in quale stato si trovi e quindi decidere quali azioni compiere. Tali azioni modificheranno la variabile portando l'agente in un nuovo stato.

Il comportamento di un agente è ciclico, quindi, il *main plan* dell'agente `TuCSoN` andrà inserito all'interno di un ciclo del seguente tipo:

```
while(true) {

  ⋮

}
```

Si può notare come questo ciclo ricopra la funzione che nell'agente Jason era di **!living**.

A questo punto possiamo modellare tutte le operazioni dell'agente TuCSoN sulla falsa riga dei *plan* dell'agente Jason.

Le transizioni del tipo [SRC(X) $\longmapsto$ SRC(Y) + MSG] assumono la seguente forma:

```
if(state==X) {
  toSend = new LogicTuple("ca", new Value("MSG"), new Value("TGT"));
  cnt.out(tid, toSend, (Long) null);
  state=Y;
}
```

La guardia **state(N)** è diventata la condizione per eseguire il blocco di istruzioni; l'invio del CA, eseguito tramite l'azione interna standard **send/3**, è stato rimpiazzato dalla istruzione di **out** verso il centro di tuple; l'aggiornamento dei *belief* interni, invece, consiste semplicemente nell'aggiornamento della variabile interna **state**; il destinatario del CA è sempre identificato da **TGT**.

Le transizioni del tipo [TGT(X) + MSG $\longmapsto$ TGT(Y)] possono diventare due operazioni diverse a seconda del fatto che lo stato **X**, abbia o meno, un numero di archi in uscita maggiore di **1**.

Nel caso in cui nello stato **X** si possa eseguire solo questa operazione, si ha:

```
if(state==X) {
  template = new LogicTuple("ca", new Value("MSG"), new Value("SRC"));
  received = cnt.in(tid, template, (Long) null);
  state=Y;
}
```

La ricezione del messaggio deve essere eseguita attivamente dall'agente e non è automatica come nell'aggiornamento della base di conoscenza in Jason. In questo caso viene utilizzata una istruzione **in**, bloccante, non potendo nello stato **X** fare altro. Da notare anche che il campo **TGT** è

stato sostituito da **SRC**, essendo in attesa di ricevere il CA.

Nel caso in cui, invece, nello stato **X**, l'agente possa eseguire più di una operazione:

```
if(state==X) {
  template = new LogicTuple("ca", new Value("MSG"), new Value("SRC"));
  received = cnt.inp(tid, template, (Long) null);
  if(received != null) {
    state=Y;
  }
}
```

Il caso è analogo al precedente, salvo il fatto che la ricezione è eseguita tramite una **inp**, non bloccante. Questa differenza è dovuta al fatto che, mentre nel caso precedente qualora il CA identificato da **MSG** non fosse stato presente, occorreva mettersi in attesa, adesso lo stato **X** presenta almeno una alternativa di comportamento che potrebbe essere necessario intraprendere, impedendo la possibilità dell'utilizzo di una **in**.

Le transizioni del tipo [A(X) $\longmapsto$ A(0)], dove con **A** identifichiamo un agente generico, invece sono codificate come segue:

```
if(state==X) {
  state=0;
}
```

L'operazione non fa altro che aggiornare la variabile interna **state** con il valore **0**. Queste operazioni sono posizionate alla fine della specifica di comportamento, per cui verranno eseguite solamente se non c'è nessun'altra operazione possibile, in quel momento, per lo stato **X**.

4.3.2 Gli artefatti

Ad ogni entità che partecipa al protocollo viene associato un artefatto personale che permetterà all'entità di interagire con le altre del protocollo. Gli artefatti verranno utilizzati solamente per lo scambio dei CA, andando a ricoprire la funzione che nel caso di Jason era gestita dall'azione interna **send/3**. Tutte le informazioni sul protocollo, cioè sulle entità con le quali deve avvenire lo scambio di CA, l'ordine con il quale devono avvenire le comunicazioni e le informazioni sullo stato interno, sono ancora gestite dall'agente.

Il comportamento di ogni artefatto sarà costituito dalle seguenti due sole *reazioni*:

```
reaction(
  out(ca(MSG,TGT)), (operation, invocation), TGT ? out(ca(MSG,SRC))
).
reaction(
  out(ca(MSG,TGT)), (operation, completion), in(ca(MSG,TGT))
).
```

Entrambe le *reazioni* vengono attivate dalla emissione da parte di un agente (è stato usato **operation**) della tupla **ca(MSG,TGT)** che identifica il desiderio dell'agente di inviare il CA identificato da **MSG** a **TGT**. La prima differenza dai casi precedenti è che **TGT** non si riferisce all'agente destinatario, ma all'artefatto corrispondente.
La prima *reazione* (attivata in fase di pre-esecuzione grazie a **invocation**) inoltra il CA al centro di tuple corrispondente all'artefatto **TGT**, avendo cura di sostituire **TGT**, ormai inutile come informazione, con **SRC**, permettendo lato ricevente di discriminare la provenienza del CA.
La seconda *reazione* (attivata in fase di post-esecuzione grazie a **completion**), elimina la tupla precedentemente emessa dall'agente, avendo assolto al compito che le spettava.

Le *reazioni* si riferiscono esclusivamente alla transizione di invio del CA; la transizione di ricezione è integrata nella fase di recupero della tupla corrispondente al CA inviato, da parte dell'agente destinatario, dal proprio artefatto personale.

4.3.3 Caso di studio

Come esempio, verrà utilizzato lo stesso protocollo del caso di studio per `Jason`:

```
(ca(a,x,q1) -> ca(x,a,r1)) and (ca(b,x,q2) -> ca(x,b,r2))
```

In questo caso verranno generati sette file in totale: tre saranno i prototipi dei sorgenti per gli agenti **a**, **b** e **x**; tre saranno i rispettivi artefatti, denominati secondo gli agenti; ultimo, sarà generato un file **Main** provvisorio.

Come nel caso `Jason`, anche qui i sorgenti degli agenti **a** e **b** sono identici, per cui verrà proposto solo il codice del *main plan* di **a**:

```
if(state==0) { // [b0]
  toSend = new LogicTuple("ca", new Value("q1"), new Value("x"));
  cnt.out(tid, toSend, (Long) null);
  state=1;
}
if(state==1) { // [b1]
```

```
  template = new LogicTuple("ca", new Value("r1"), new Value("x"));
  received = cnt.in(tid, template, (Long) null);
  state=2;
}
if(state==2) { // [b2]
  state=0;
}
```

Anche qui è possibile ravvisare una corrispondenza univoca tra i blocchi *if* e le transizioni della A-Net associata all'agente:

- *b0* $\longleftrightarrow$ [a(0) $\longmapsto$ a(1) + q1]
- *b1* $\longleftrightarrow$ [a(1) + r1 $\longmapsto$ a(2)]
- *b2* $\longleftrightarrow$ [a(2) $\longmapsto$ a(0)]

Visto che nello stato **1** l'unica operazione possibile è la ricezione del CA identificato da **r1**, viene utilizzata l'operazione bloccante **in**.

Il sorgente dell'agente **x**, a causa dei due comportamenti alternativi possibili, presenta molte più alternative di comportamento:

```
if(state==0) { // [b0]
  template = new LogicTuple("ca", new Value("q1"), new Value("a"));
  received = cnt.inp(tid, template, (Long) null);
  if(received != null) { state=1; }
}
if(state==1) { // [b1]
  toSend = new LogicTuple("ca", new Value("r1"), new Value("a"));
  cnt.out(tid, toSend, (Long) null);
  state=2;
}
if(state==2) { // [b2]
  template = new LogicTuple("ca", new Value("q2"), new Value("b"));
    received = cnt.in(tid, template, (Long) null);
    state=3;
  }
if(state==3) { // [b3]
  toSend = new LogicTuple("ca", new Value("r2"), new Value("b"));
  cnt.out(tid, toSend, (Long) null);
  state=4;
}

if(state==0) { // [b4]
  template = new LogicTuple("ca", new Value("q2"), new Value("b"));
  received = cnt.inp(tid, template, (Long) null);
    if(received != null) { state=5; }
}
```

```
if(state==5) { // [b5]
  toSend = new LogicTuple("ca", new Value("r2"), new Value("b"));
  cnt.out(tid, toSend, (Long) null);
  state=6;
}
if(state==6) { // [b6]
  template = new LogicTuple("ca", new Value("q1"), new Value("a"));
  received = cnt.in(tid, template, (Long) null);
  state=7;
}
if(state==7) { // [b7]
  toSend = new LogicTuple("ca", new Value("r1"), new Value("a"));
  cnt.out(tid, toSend, (Long) null);
  state=8;
}
if(state==4) { state=0; } // [b8]
if(state==8) { state=0; } // [b9]
```

Riprendendo le transizione della A-Net di **x**, osserviamo:

$b0 \longleftrightarrow [\ \mathrm{x}(0) + \mathrm{q1} \longmapsto \mathrm{x}(1)\ ]$ $\quad$ $b4 \longleftrightarrow [\ \mathrm{x}(0) + \mathrm{q2} \longmapsto \mathrm{x}(5)\ ]$

$b1 \longleftrightarrow [\ \mathrm{x}(1) \longmapsto \mathrm{x}(2) + \mathrm{r1}\ ]$ $\quad$ $b5 \longleftrightarrow [\ \mathrm{x}(5) \longmapsto \mathrm{x}(6) + \mathrm{r2}\ ]$

$b2 \longleftrightarrow [\ \mathrm{x}(2) + \mathrm{q2} \longmapsto \mathrm{x}(3)\ ]$ $\quad$ $b6 \longleftrightarrow [\ \mathrm{x}(6) + \mathrm{q1} \longmapsto \mathrm{x}(7)\ ]$

$b3 \longleftrightarrow [\ \mathrm{x}(3) \longmapsto \mathrm{x}(4) + \mathrm{r2}\ ]$ $\quad$ $b7 \longleftrightarrow [\ \mathrm{x}(7) \longmapsto \mathrm{x}(8) + \mathrm{r2}\ ]$

$b8 \longleftrightarrow [\ \mathrm{x}(4) \longmapsto \mathrm{x}(0)\ ]$ $\quad$ $b9 \longleftrightarrow [\ \mathrm{x}(8) \longmapsto \mathrm{x}(0)\ ]$

Nello stato **0**, l'agente può intraprendere due comportamenti diversi a seconda del CA ricevuto, quindi le operazioni di recupero della tupla devono essere non bloccanti e usare quindi una **inp**; una volta nello stato **1** o nello stato **5**, però, il comportamento è stabilito e per la ricezione si utilizza una **in**.

Le specifiche di comportamento dei centri di tuple sono identiche per tutti e tre gli artefatti. Prendendo, ad esempio, l'artefatto di **a**, è la seguente:

```
reaction(
  out(ca(MSG,TGT)), (operation, invocation), TGT ? out(ca(MSG,a))
).
reaction(
  out(ca(MSG,TGT)), (operation, completion), in(ca(MSG,TGT))
).
```

L'artefatto non fa altro che inoltrare i CA all'artefatto **TGT** espresso dall'agente, sostituendo l'informazione **TGT** del CA con se stesso, **a** in questo caso.

4.4 Secondo modello: artefatti complessi

In questo modello, le informazioni sul protocollo di interazione sono state incorporate negli artefatti. In questo modo, la sola struttura data dagli artefatti è in grado di garantire il protocollo, indipendentemente dagli agenti che effettivamente vi prenderanno parte.

4.4.1 Gli agenti

Questo modello si distacca dalla struttura simile a `Jason` del modello precedente. Gli agenti non devono più gestire le informazioni sullo stato del protocollo, mantenuta interamente dagli artefatti personali.

Anche in questo caso dobbiamo presupporre un comportamento ciclico, per cui all'interno avremo un ciclo analogo al caso precedente:

```
while(true) {
  ⋮
}
```

Per quanto riguarda la modellazione delle transizioni abbiamo delle differenze.

Il codice per le transizioni del tipo [SRC(X) $\longmapsto$ SRC(Y) + MSG] diventa:

```
toSend = new LogicTuple("MSG");
cnt.out(tid, toSend, (Long) null);
```

Le transizioni del tipo [TGT(X) + MSG $\longmapsto$ TGT(Y)] hanno ancora due forme alternative a seconda della possibilità di eseguire o meno altre operazioni nello stato **X**.

Nel caso di unica operazione possibile abbiamo:

```
template = new LogicTuple("MSG");
received = cnt.in(tid, template, (Long) null);
```

Nel caso di altre operazioni alternative, invece:

```
template = new LogicTuple("MSG");
received = cnt.inp(tid, template, (Long) null);
```

Non essendo più disponibile l'informazione sullo stato, non è più possibile controllare a priori il flusso di esecuzione delle operazioni, per cui l'agente dovrà occuparsi di definire una logica di esecuzione compatibile con il protocollo di interazione a cui intende partecipare. Dalle operazioni sono anche sparite le informazioni sulla provenienza e sulla destinazione dei CA, in quanto sarà l'artefatto ad occuparsene.

Le transizioni del tipo [A(X) $\longmapsto$ A(0)] spariscono dal comportamento dell'agente.

4.4.2 Gli artefatti

La specifica di comportamento degli artefatti è decisamente più complessa di quella del caso precedente, dovendo, in questo modello, occuparsi della gestione di tutte le informazioni per una corretta esecuzione del protocollo di interazione.

La prima differenza consiste nel fatto che, mentre prima l'informazione sullo stato era gestita dall'agente, adesso è l'artefatto a doversene occupare. Occorre presupporre, quindi, che l'artefatto sia in possesso di una tupla come la seguente:

```
state(X).
```

La tupla indicherà il valore corrente dello stato del protocollo, qui identificato con **X**. La tupla sarà dunque inizializzata con il valore **0** alla creazione dell'artefatto.

Per le transizioni del tipo [SRC(X) $\longmapsto$ SRC(Y) + MSG] abbiamo due *reazioni*:

```
reaction(
  out(MSG), (operation, invocation), (
    inp(state(X)), out(state(X,Y)), TGT ? out(ca(MSG,SRC))
  )
).
reaction(
  out(MSG), (operation, completion), (
    inp(state(X,Y)), out(state(Y)), in(MSG)
  )
).
```

In questo caso, l'agente emette la sola informazione **MSG**, senza specificare il destinatario del CA. La *reazione* identifica il corretto destinatario

del CA, qui identificato da **TGT**, attraverso la tupla **state(X)**. L'operazione **inp(state(X))** eseguita dalla *reazione*, permette di discriminare tra tutte le possibili *reazioni* che potrebbero essere eseguite, quelle che effettivamente devono essere eseguite in corrispondenza dello stato **X**. Supponendo che la *reazione* in fase di pre-esecuzione abbia avuto successo, la tupla **state(X)** viene sostituita dalla tupla **state(X,Y)** ad identificare lo stato intermedio di transizione tra lo stato **X** e lo stato **Y**. L'invocazione della **out** sull'artefatto **TGT** avviene come nel caso precedente. In fase di post-esecuzione, analogamente a quanto detto per la tupla **state(X)**, l'unica *reazione* che verrà valutata positivamente sarà quella che corrisponderà allo stato intermedio **state(X,Y)**, che aggiornerà lo stato al nuovo valore **Y** ed eliminerà la tupla **MSG**, in quanto ormai superflua.

Per le transizioni del tipo [TGT(X) + MSG $\longmapsto$ TGT(Y)], le *reazioni* nei casi di operazione unica e operazione con possibili alternative, sono molto diverse. Nel primo caso abbiamo:

```
reaction(
  in(MSG), (operation, invocation), (
    inp(state(X)), out(state(X,Y))
  )
).
reaction(
  in(MSG), (operation, completion), (
    inp(state(X,Y)), out(state(Y))
  )
).
reaction(
  out(state(X,Y)), internal, (
    inp(ca(MSG,SRC)), out(MSG)
  )
).
reaction(
  out(ca(MSG,SRC)), (link_in, completion), (
    rdp(state(X,Y)), in(ca(MSG,SRC)), out(MSG)
  )
).
```

Le prime due *reazioni* servono, in fase di pre-esecuzione, per aggiornare il valore **state(X)** al valore intermedio **state(X,Y)** e, in fase di post-esecuzione, per aggiornare il valore intermedio **state(X,Y)** al valore definitivo **state(Y)**. La terza *reazione* viene attivata dall'emissione interna (grazie a **internal**) della tupla **state(X,Y)**, che procede a verificare la presenza di **ca(MSG,SRC)** e, in caso affermativo, prelevarlo ed emettere la tupla **MSG** richiesta dall'agente. Nel caso invece che **ca(MSG,SRC)** non sia ancora presente nell'artefatto, al momento della

sua emissione, sarà ancora presente la tupla **state(X,Y)**, che permetterà alla quarta *reazione* (attivata da una operazione di linking verso questo artefatto grazie a **link_in**) di recuperare **ca(MSG,SRC)** appena emesso e sostituirlo con **MSG**, permettendo all'agente di continuare la propria esecuzione.

Il secondo caso è più semplice. L'unica *reazione* associata alla transizione è la seguente:

```
reaction(
  inp(MSG), (operation, invocation), (
    inp(ca(MSG,SRC)), inp(state(X)), out(MSG), out(state(Y))
  )
).
```

La *reazione* verifica la presenza di **ca(MSG,SRC)** e, in caso affermativo, emette **MSG** e aggiorna i valori di stato. Nel caso che, invece, **ca(MSG,SRC)** non sia presente, non succede nulla.

Le transizioni del tipo [A(X) $\longmapsto$ A(0)] vengono gestite tramite una *reazione* come la seguente:

```
reaction(
  out(state(X)), internal, (
    in(state(X)), out(state(0))
  )
).
```

La *reazione* aggiorna lo stato con il valore **0** ogni volta che l'artefatto arriva in uno stato finale, qui identificato dallo stato con valore **X**.

4.4.3 Caso di studio

In questo caso di studio verrà analizzato lo stesso protocollo del modello precedente, così da evidenziare le differenze. Il protocollo in questione è sempre:

```
(ca(a,x,q1) -> ca(x,a,r1)) and (ca(b,x,q2) -> ca(x,b,r2))
```

Anche per questo modello verranno generati sette file: tre per gli agenti **a**, **b** e **x**, tre per gli artefatti e il file **Main** provvisorio.

Di seguito è proposto il *main plan* dell'agente **a** (sempre analogo a quello dell'agente **b**):

```
// [b0]
toSend = new LogicTuple("q1");
cnt.out(tid, toSend, (Long) null);
// [b1]
template = new LogicTuple("r1");
received = cnt.in(tid, template, (Long) null);
```

Il comportamento risulta molto semplificato rispetto a quello del modello precedente, in quanto si è ridotto alla sola emissione del CA identificato da **q1** e alla ricezione del CA identificato da **r1**; tutte le restanti informazioni sono gestite dall'artefatto. La correlazione con le transizioni della A-Net non è più evidente: l'agente diventa, di fatto, un semplice utilizzatore dell'artefatto.

Anche le specifiche del comportamento dei centri di tuple corrispondenti sono pressoché identiche, per cui è riportata solo quella dell'artefatto di **a**:

```
reaction( % [r0.a]
  out(q1), (operation, invocation), (
    inp(state(0)), out(state(0,1)), x ? out(ca(q1,a))
  )
).
reaction( % [r0.b]
  out(q1), (operation, completion), (
    inp(state(0,1)), out(state(1)), in(q1)
  )
).

reaction( % [r1.a]
  in(r1), (operation, invocation), (
    inp(state(1)), out(state(1,2))
  )
).
reaction( % [r1.b]
  in(r1), (operation, completion), (
    inp(state(1,2)), out(state(2))
  )
).
reaction( % [r1.c]
  out(state(1,2)), internal, (
    inp(ca(r1,x)), out(r1)
 )
).
reaction( % [r1.d]
  out(ca(r1,x)), (link_in, completion), (
    rdp(state(1,2)), in(ca(r1,x)), out(r1)
  )
).
```

```
reaction( % [r2]
  out(state(2)), internal, (
    in(state(2)), out(state(0))
  )
).
```

Per evidenziare la correlazione con le transizioni della A-Net, possiamo distinguere tre gruppi di *reazioni*:

- *r0* $\longleftrightarrow$ [a(0) $\longmapsto$ a(1) + q1]
- *r1* $\longleftrightarrow$ [a(1) + r1 $\longmapsto$ a(2)]
- *r2* $\longleftrightarrow$ [a(2) $\longmapsto$ a(0)]

Il caso dell'agente **x** è leggermente più complicato. Il codice generato dell'agente è il seguente:

```
template = new LogicTuple("q1");
received = cnt.inp(tid, template, (Long) null);

if(received != null) { // not auto-generated
  toSend = new LogicTuple("r1");
  cnt.out(tid, toSend, (Long) null);

  template = new LogicTuple("q2");
  received = cnt.in(tid, template, (Long) null);

  toSend = new LogicTuple("r2");
  cnt.out(tid, toSend, (Long) null);
} // not auto-generated

template = new LogicTuple("q2");
received = cnt.inp(tid, template, (Long) null);

if(received != null) { // not auto-generated
  toSend = new LogicTuple("r2");
  cnt.out(tid, toSend, (Long) null);

  template = new LogicTuple("q1");
  received = cnt.in(tid, template, (Long) null);

  toSend = new LogicTuple("r1");
  cnt.out(tid, toSend, (Long) null);
} // not auto-generated
```

L'agente viene generato con tutte le operazioni che gli competono e ad esso rimane soltanto il compito di applicare un criterio di scelta, quando

necessario. In questo caso, l'agente può intraprendere due comportamenti diversi, a seconda dell'arrivo del CA identificato da **q1** o dell'arrivo del CA identificato da **q2**, per questo le prime operazioni da eseguire saranno le **inp** di questi CA. Nel caso in cui una delle **inp** abbia successo, allora l'agente inizierà ad eseguirà le operazioni corrispondenti. Le operazioni di ricezione dopo la prima sono delle **in**, poiché una volta stabilito quale dei due comportamenti tenere, le azioni sono predeterminate. L'ordine con cui l'agente deciderà di eseguire le **inp** iniziali è ininfluente, a patto che non venga alterato l'ordine delle operazioni a seguire.

```
reaction( % [r0]
  inp(q1),
  (operation, invocation),
  (
    inp(ca(q1,a)),
    inp(state(0)),
    out(q1), out(state(1))
  )
).
reaction( % [r1.a]
  out(r1),
  (operation, invocation),
  (
    inp(state(1)),
    out(state(1,2)),
    a ? out(ca(r1,x))
  )
).
reaction( % [r1.b]
  out(r1),
  (operation, completion),
  (
    inp(state(1,2)),
    out(state(2)),
    in(r1)
  )
).
```

```
reaction( % [r2]
  inp(q2),
  (operation, invocation),
  (
    inp(ca(q2,b)),
    inp(state(0)),
    out(q2), out(state(5))
  )
).
reaction( % [r3.a]
  out(r2),
  (operation, invocation),
  (
    inp(state(5)),
    out(state(5,6)),
    b ? out(ca(r2,x))
  )
).
reaction( % [r3.b]
  out(r2),
  (operation, completion),
  (
    inp(state(5,6)),
    out(state(6)),
    in(r2)
  )
).
```

Queste prime *reazioni* permettono di controllare quale tra **ca(q1,a)** e **ca(q2,b)** sia disponibile. Nel caso che uno dei due sia presente, viene rimosso dall'artefatto, viene emesso il messaggio corrispondente e viene aggiornato lo stato interno, concordemente a quale CA sia stato prelevato dall'agente (**state(1)** per **q1**, **state(5)** per **q2**). Le *reazioni* che seguono vengono attivate dall'emissione delle tuple **r1** e **r2**; anche se sono presenti altre *reazioni* attivate dalle medesime azioni, esse non possono essere

valutate positivamente a questo punto, perché non riguardano né lo stato **1**, né lo stato **5**. Le transizioni corrispondenti sono:

r0 ⟷ [x(0) + q1 ⟼ x(1)]   *r2* ⟷ [x(0) + q2 ⟼ x(5)]

r1 ⟷ [x(1) ⟼ x(2) + r1]   *r3* ⟷ [x(5) ⟼ x(6) + r2]

```
reaction( % [r4.a]
  in(q2),
  (operation, invocation),
  (
    inp(state(2)),
    out(state(2,3))
  )
).
reaction( % [r4.b]
  in(q2),
  (operation, completion),
  (
    inp(state(2,3)),
    out(state(3))
  )
).
reaction( % r[r4.c]
  out(state(2,3)),
  internal,
  (
    inp(ca(q2,b)), out(q2)
  )
).
reaction( % r[4.d]
  out(ca(q2,b)),
  (link_in, completion),
  (
    rdp(state(2,3)), in(ca(q2,b)),
    out(q2)
  )
).
```

```
reaction( % r[5.a]
  in(q1),
  (operation, invocation),
  (
    inp(state(6)),
    out(state(6,7))
  )
).
reaction( % r[5.b]
  in(q1),
  (operation, completion),
  (
    inp(state(6,7)),
    out(state(7))
  )
).
reaction( % r[5.c]
  out(state(6,7)),
  internal,
  (
    inp(ca(q1,a)), out(q1)
  )
).
reaction( % r[5.d]
  out(ca(q1,a)),
  (link_in, completion),
  (
    rdp(state(6,7)), in(ca(q1,a)),
    out(q1)
  )
).
```

Queste *reazioni* permettono di eseguire le operazioni di ricezione di **q2** nel primo caso e **q1** nel secondo, bloccanti; a questo punto, infatti, non ci sono altre alternative e le ricezioni possono essere fatte tramite delle **in**. Il secondo gruppo di *reazioni* permette all'agente di ignorare da quale mittente debba provenire il CA: il fatto di recuperare il CA proveniente dal giusto mittente è garantito dall'artefatto, le ultime *reazioni*, infatti, generano la tupla che interessa l'agente solo se questo ne ha fatto già richiesta e ci si trovi, quindi, negli stati intermedi dati da **state(2,3)** e **state(6,7)**. Le transizioni corrispondenti sono:

- *r4* $\longleftrightarrow$ [x(2) + q2 $\longmapsto$ x(3)]
- *r5* $\longleftrightarrow$ [x(6) + q1 $\longmapsto$ x(7)]

```
reaction( % [r6.a]
  out(r2),
  (operation, invocation),
  (
    inp(state(3)),
    out(state(3,4)),
    b ? out(ca(r2,x))
  )
).
reaction( % [r6.b]
  out(r2),
  (operation, completion),
  (
    inp(state(3,4)),
    out(state(4)),
    in(r2)
  )
).
```

```
reaction( % [r7.a]
  out(r1),
  (operation, invocation),
  (
    inp(state(7)),
    out(state(7,8)),
    a ? out(ca(r1,x))
  )
).
reaction( % [r7.b]
  out(r1),
  (operation, completion),
  (
    inp(state(7,8)),
    out(state(8)),
    in(r1)
  )
).
```

Queste *reazioni* sono le corrispettive di *r1* e *r3* viste precedentemente. Le transizioni corrispondenti sono:

- *r6* $\longleftrightarrow$ [x(3) $\longmapsto$ x(4) + r2]
- *r7* $\longleftrightarrow$ [x(7) $\longmapsto$ x(8) + r1]

```
reaction( % [r8]
  out(state(4)),
  internal,
  (
    in(state(4)),
    out(state(0))
  )
).
```

```
reaction( % [r9]
  out(state(8)),
  internal,
  (
    in(state(8)),
    out(state(0))
  )
).
```

Queste ultime due *reazioni* sono quelle che permettono la ciclicità del comportamento, aggiornando i valori degli stati finali, identificati tramite **state(4)** e **state(8)**, con il valore iniziale **state(0)**. Queste due *reazioni* si attivano autonomamente senza alcun intervento dell'agente. Le transizioni corrispondenti sono:

- *r8* $\longleftrightarrow$ [x(4) $\longmapsto$ x(0)]
- *r9* $\longleftrightarrow$ [x(8) $\longmapsto$ x(0)]

Capitolo 5

Conclusioni

Si può affermare, che qualsiasi sia la strada che si segue per la progettazione e la realizzazione di un sistema multiagente, l'interazione gioca sicuramente il ruolo chiave. Modellare l'interazione di entità software è generalmente cosa complessa, specialmente nei casi in cui la numerosità delle entità e delle interazioni che tra esse intercorrono, comincia a crescere.

Lo scopo iniziale di questo lavoro era quello di venire incontro ai progettisti, fornendo loro uno strumento al contempo di analisi e prototipazione, in grado di sollevare dalle loro spalle molta della fatica necessaria alla realizzazione. L'applicazione delle reti di Petri alla modellazione di sistemi multiagente, permette di analizzare i modelli alla ricerca delle proprietà desiderate, che da un'analisi dei modelli AUML sarebbe difficile, se non impossibile, dedurre. D'altra parte, i progettisti umani preferiscono astrazioni di alto livello come quelle fornite da AUML, rispetto a quelle di una rete di Petri. Volendo sfruttare entrambe le caratteristiche, l'unico modo era quello di realizzare un sistema in grado di trasformare il modello *user friendly* AUML, in una più formale rete di Petri. Per quanto concerne questo aspetto, il sistema ha sicuramente soddisfatto le aspettative.

Il processo di realizzazione del sistema modellato, ha richiesto inoltre uno studio approfondito dei paradigmi utilizzati dalle due piattaforme, `Jason` e `TuCSoN`, al fine di fornire una corretta implementazione.

La piattaforma `Jason`, utilizza un sistema di entità base costituito esclusivamente dagli *agenti* e dall'*ambiente*. Una struttura del genere è sicuramente semplice e di facile comprensione, ma limita notevolmente il sistema realizzato. A parte le comunicazioni dirette tra *agenti*, che

sfruttano le azioni interne standard, l'unico altro modo di comunicare è attraverso l'*ambiente*, che però è un'entità unica e indivisibile, condivisa tra tutti gli *agenti*. Utilizzare l'*ambiente* per meccanismi di sincronizzazione è possibile, ma richiede un attento lavoro di specializzazione. Il protocollo di interazione risulta quindi esclusivamente a carico degli *agenti*, rendendo necessario un attento processo di riprogettazione, qualora si dovesse decidere di sostituirne o modificarne uno.

La piattaforma `TuCSoN` realizza una infrastruttura basata sul modello A&A, che sfrutta il linguaggio di specifica `ReSpecT`. La combinazione di queste tecnologie rende possibile supportare nativamente sistemi multiagenti distribuiti. A differenza che nel caso di `Jason`, qui c'è una distinzione netta tra *nodo* e *artefatto*; gli *agenti* sono a conoscenza del fatto di trovarsi in un *nodo*, popolato da *artefatti* e da altri *agenti* e che il loro *nodo* non è l'unico. Questa rappresentazione del mondo ci offre molte più possibilità di quanto non facesse `Jason`.

La prima modellazione in `TuCSoN` è stata fatta più che altro per illustrare le analogie con la modellazione in `Jason`. Gli *agenti* continuano anche qui a incorporare il protocollo e sfruttano solo minimamente il meccanismo delle *reazioni* dei centri di tuple, al solo fine di disaccoppiare spazialmente gli *agenti* e permettere un sistema distribuito.

Nella seconda modellazione, invece, vengono sfruttate completamente tutte le possibilità della piattaforma. Lo schema comportamentale non viene più incorporato negli *agenti*, ma sfruttando il meccanismo delle *reazioni*, è stato possibile inglobare il controllo del protocollo negli *artefatti*, che gli *agenti* poi andranno ad utilizzare. Non solo questo snellisce incredibilmente la specifica di comportamento degli *agenti*, ma permette di spostarsi dalla concezione in cui "gli agenti realizzano un protocollo di interazione" a quella in cui "gli agenti ricoprono un ruolo all'interno del protocollo di interazione".

L'affermazione precedente non è da prendere con leggerezza, in quanto separa il concetto di protocollo di interazione da quello di *agente*. È la struttura realizzata dagli *artefatti*, dunque, che permette di realizzare un protocollo di interazione, indipendentemente da quali entità vogliano prendervi parte, che partecipare a tale protocollo sia la loro funzione primaria, oppure no.

Bibliografia

[1] Andrea Omicini, Enrico Denti, *From Tuple Spaces to Tuple Centres*, Science of Computer Programming, 2001.

[2] Bernhard Bauer, *UML Class Diagrams Revisited in the Context of Agent-Based Systems*, Lecture Notes in Computer Science, 2002.

[3] Jean-Luc Koning, Ivan Romero-Hernandez, *Generating Machine Processable Representations of Textual Representations of AUML*, Lecture Notes in Computer Science, 2003.

[4] Sea Ling, Seng Wai Loke, *MIP-Nets: A Compositional Model of Multiagent Interaction*, Lecture Notes in Computer Science, 2003.

[5] Juliana Küster Filipe, *Giving Life to Agent Interactions*, Lecture Notes in Computer Science, 2004.

[6] Andrea Omicini, *Formal ReSpecT in the A&A Perspective*, Electronic Notes in Theoretical Computer Science, 2007.

[7] Rafael H. Bordini1, Jomi F. Hübner, *Jason - A Java-Based Interpreter for an Extended Version of AgentSpeak*, 2007.