Modeling Agent Interactions: Agent UML and Multiagent Interaction Protocol Nets

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

Abstract. Verranno discusse due delle tecniche di modellazione delle interazioni in contesti *agent oriented* che hanno dimostrato particolare efficacia: Agent UML, l'estensione del ben noto linguaggio UML, già usato estensivamente in ambienti *object oriented* e le MIP-nets, acronimo di Multiagent Interaction Protocol Nets. Inoltre saranno presentate tecniche di trasposizione di modelli scritti in AUML in strutture MIP-net al fine sia di specificare il comportamento di sistemi multi-agente, sia di verificarne la correttezza.

1. Introduzione

Come dal mondo dei linguaggi funzionali sono emersi i linguaggi ad oggetti, così negli ultimi anni si sonno affacciati e stanno prendendo rapidamente piede una serie di paradigmi, tecnologie e approcci alla progettazione e programmazione che vengono riassunti col termine di tecnologie ad agenti.

Le tecnologie ad agenti permettono la progettazione di sistemi caratterizzati da comportamenti intelligenti, distribuzione spaziale e/o temporale e supporto alla mobilità. Sicuramente queste tecnologie giocheranno un ruolo fondamentale per lo sviluppo dei sistemi software futuri. Uno dei punti di forza, ma anche uno degli scogli più alti, dei sistemi multi-agente o, più in generale, dei sistemi ad agenti, è la possibilità/necessità di modellare un comportamento, uno schema interazionale, un fine al quale gli agenti tenderanno, che caratterizzerà il sistema in questione.

Il motivo per cui i paradigmi ad agenti stentano ad affermarsi in ambiti industriali è la difficoltà di estendere le metodologie e le tecniche utilizzate a supporto della modellazione e progettazione di sistemi ad oggetti, ampiamente accettate e fidate anche in ambito industriale, a questa nuova tecnologia che, però, discende direttamente da quella precedente.

La strada che è stata maggiormente percorsa è che promette alquanto bene è quella di estendere il 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 ad agenti.

Il passo successivo è quello di fornire una serie di regole e metodologie per tradurre un modello scritto in AUML in una MIP-net, per poter usufruire della capacità espressiva e semplicità di modellazione di questa estensione di UML e contemporaneamente usare le già ampiamente sviluppate e trattate tecniche di *model checking* applicabili ad una rete di Petri. Utilizzando poi una rappresentazione testuale *machine processable* di un modello AUML, cioè che ne permette la

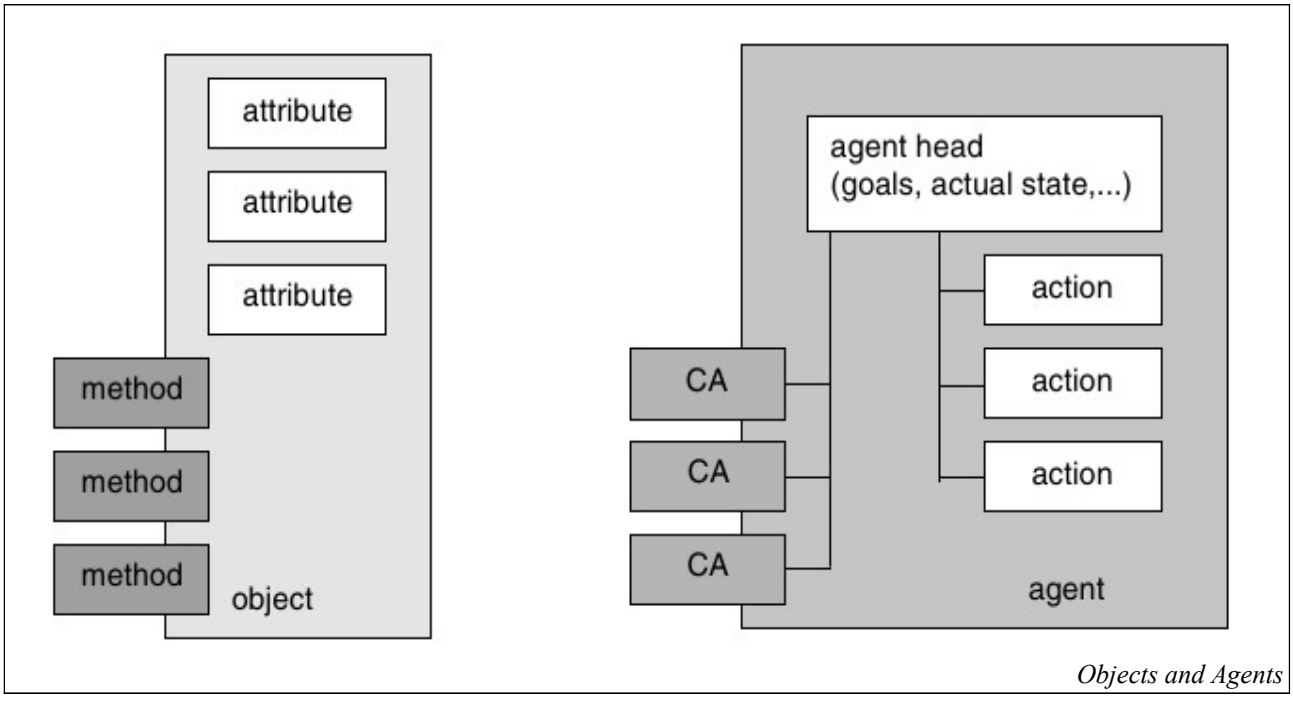
lettura e la comprensione da parte di un compilatore, potremo automatizzare il processo di conversione da un modello all'altro, avendo così a disposizione delle tecniche di modellazione di sistemi ad agenti particolarmente efficaci e, fiduciosamente, efficienti.

2. Relazionare *oggetti* e *agenti*

Un *oggetto* è composto da un insieme di variabili, chiamati attributi o 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 di una *classe* 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.

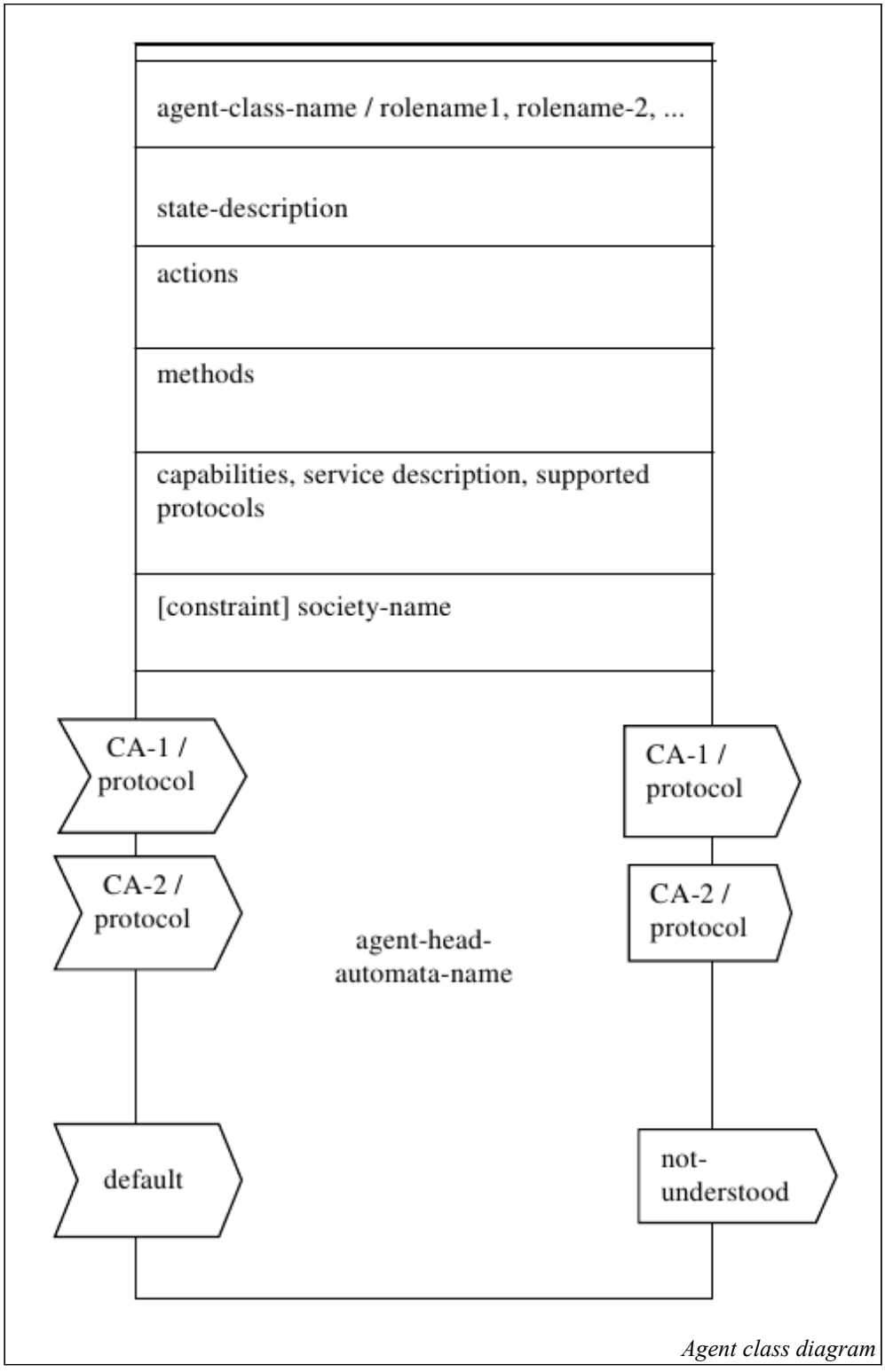
Delle proprietà specificano la visibilità dei metodi o delle variabili da parte dell'utilizzatore. Nella maggior parte dei linguaggi le *classi* definiscono dei tipi. Alcuni linguaggi permettono la definizione di variabili di classe condivise da tutte le istanze. Oltre alle variabili di classe, i metodi 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.



Objects and Agents

Contrariamente ad un *oggetto* che invoca dei metodi, un *agente* interagisce tramite atti comunicativi (**CA** in breve) ed è in grado di valutarli rispettando i propri obiettivi, piani, task o preferenze, in accordo con il suo modello interno del mondo. 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



Agent class diagram

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 multi-agente 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.

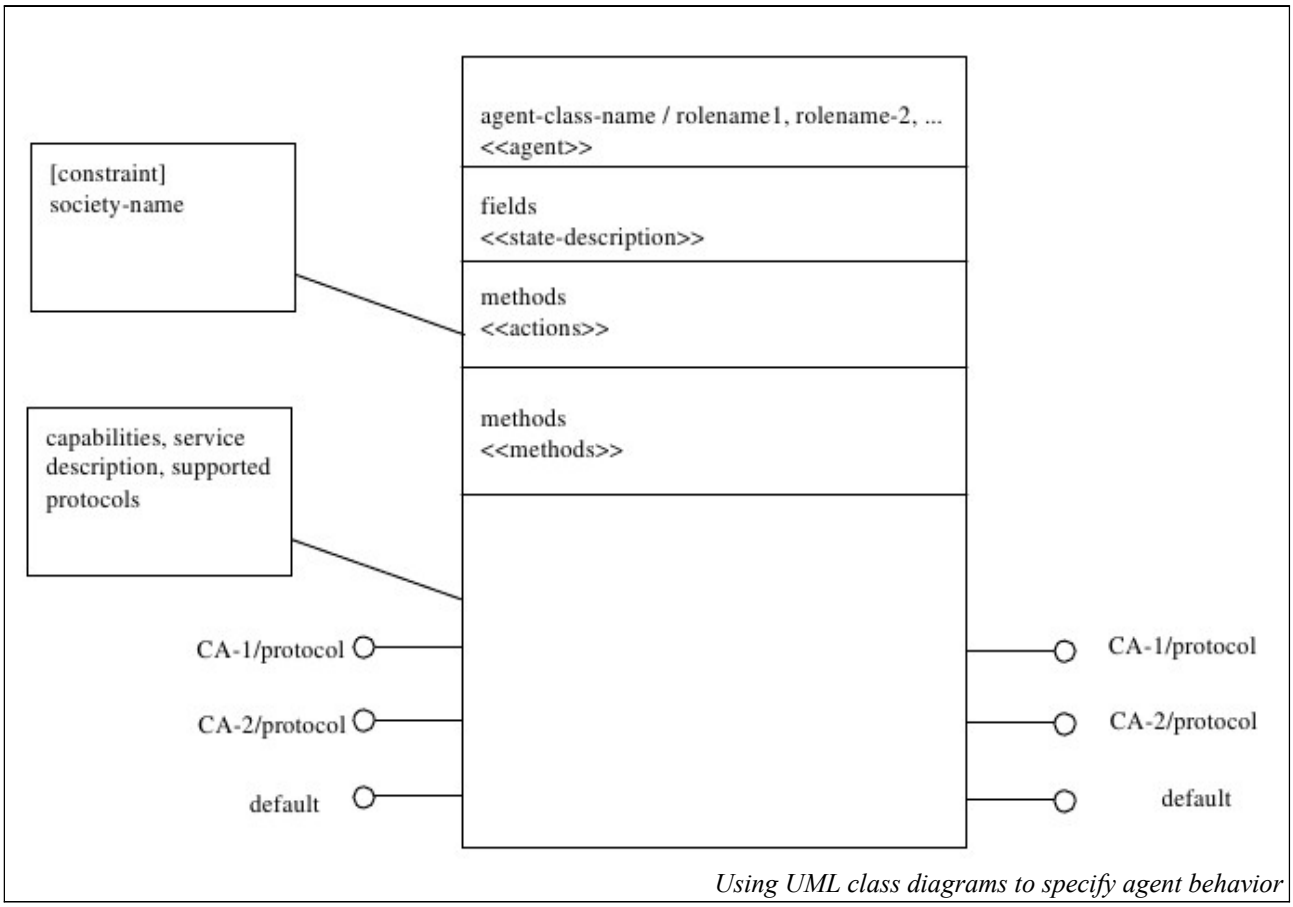
In più agli *agenti* è possibile applicare tecniche proprie del mondo *object oriented* quali **ereditarietà**, **tipi astratti** e **interfaccia**, **generici**, **associazione**, **aggregazione** e **composizione**. I componenti possono essere sia *classi agent oriented* sia comuni *classi object oriented*. Un *agente* può essere infatti costruito includendo *oggetti* come parte del suo stato interno.

Nel paradigma *agent oriented* dobbiamo distinguere *classi* di *agenti*, che definiscono la struttura e il tipo di un *agente*, e *agenti individuali*, istanze di una *classe*. UML permette di distinguere tre livelli: **concettuale**, **specifica** e **implementazione**.

Dal punto di vista *agent oriented* 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 istanze di *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 l'implementazione.



Using UML class diagrams to specify agent behavior

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 multi-agente, 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*. La forma generale per descrivere un *ruolo* in AUML è

instance-1 ... instance-n / role-1 ... role-m : class

intendendo che **instance-1 ... instance-n** con ruoli **role-1 ... role-m** appartengono a **class**.

Una descrizione *degli stati* è simile alla descrizione dei campi di un diagramma delle classi con la differenza che viene introdotta una classe **wff (well-formed-formula)** per tutti i tipi di descrizione logica degli *stati*, indipendente dalla logica sottostante. Questa estensione permette la definizione, per esempio, di *agenti BDI*. Oltre all'estensione del tipo per i campi, è possibile aggiungere attributi come *visibilità* e *persistenza*.

Nel caso di una semantica *BDI* possono essere definite tre variabili di istanza chiamate *convinzioni (beliefs)*, *desideri (desires)* e *intenzioni (intentions)* di tipo *wff*, che descrivono le componenti omonime dell'*agente BDI*.

Nel caso invece di una semantica puramente *goal-oriented* possono essere definite due variabili di istanza *wff* chiamate *goal-permanenti* e *goal-attuali*.

Possono essere specificati due tipi di *azioni*:

- **pro-attive**: attivate dall'*agente* stesso se sono soddisfatte le pre-condizioni
- **re-attive**: attivate da un altro *agente*

La descrizione di un'*azione* di un *agente* consiste nella *signature* dell'azione con l'attributo di visibilità, il nome dell'*azione* e una lista di parametri con i tipi associati. Pre-condizioni, post-condizioni, effetti e invarianti, come in UML, definiscono la semantica dell'*azione*. I *metodi* sono definiti come in UML, eventualmente con pre-condizioni, post-condizioni, effetti e invarianti. Le *capacità* di un *agente* possono essere definite sia in modo informale sia usando i diagrammi delle classi.

Inviare e ricevere *atti comunicativi* caratterizza l'interfaccia di un agente nel suo ambiente. Viene assunto che classi e oggetti rappresentino l'informazione circa gli *atti comunicativi*. L'*atto comunicativo* ricevuto o inviato può essere sia una classe che una istanza concreta.

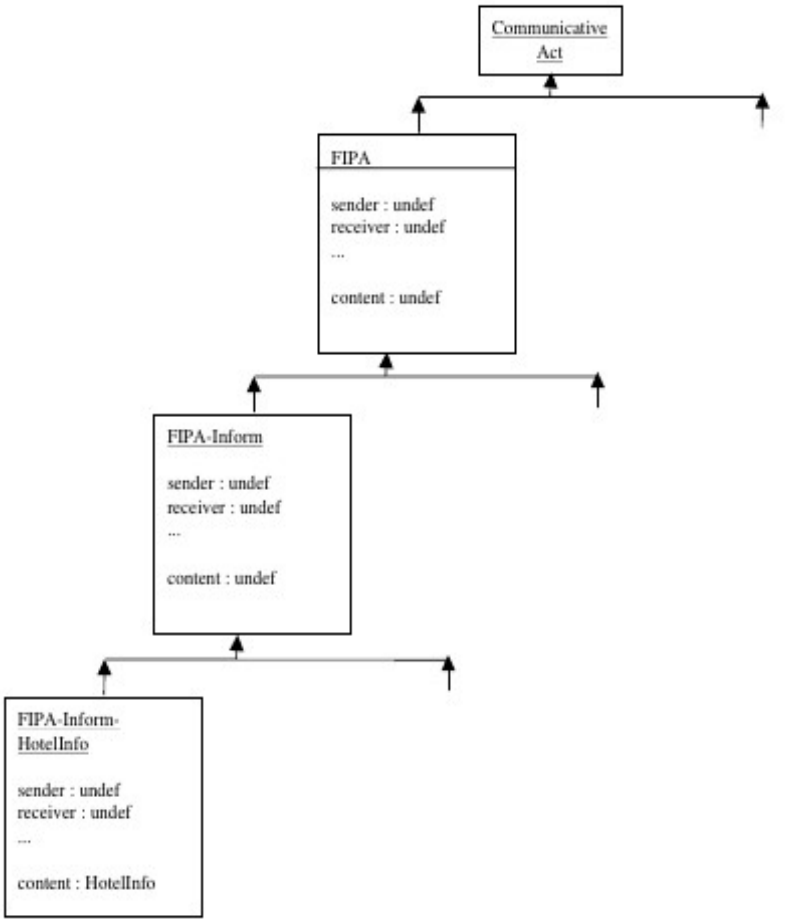
La notazione **CA-1 / protocol** viene utilizzata per descrivere che l'*atto comunicativo* della classe **CA-1** viene ricevuto nel contesto del protocollo di interazione **protocol**. Nel caso di una istanza di un *atto comunicativo* viene applicata la notazione **CA-1 / protocol** con significato analogo. Come notazioni alternativa possiamo utilizzare rispettivamente **protocol[CA-1]** e **protocol[CA-1]**. Il contesto può essere omesso se l'*atto comunicativo* viene interpretato indipendentemente dal contesto.

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.

Il caso più semplice è il caso **default**; **default** può corrispondere a qualsiasi *atto comunicativo* e **not-understood** è la risposta ad *atti comunicativi* non riconosciuti dall'*agente*. Visto che vengono sottoposte alla procedura di *matching* sia *istanze* che classi di *atti comunicativi*, possono essere

presenti variabili libere all'interno di *atti comunicativi* istanziati. Classi senza metodi definiscono *atti comunicativi*.



Instance hierarchy on communicative acts

Un *atto comunicativo* **CA** risolve il *matching* con uno **CA',** se:

- **CA** è una classe
- **CA'** deve essere una istanza della classe **CA**
- **CA'** deve essere una istanza di una sottoclasse di **CA**
- **CA** è una istanza di una classe
- **CA'** è una istanza della stessa classe di cui è istanza **CA**
- **CA.field** risolve il *matching* con **CA'.field** per tutti i campi **field** della classe **CA**

Un campo **CA.field** risolve il *matching* con un campo **CA'.field**, se:

- **CA.field** ha il valore **undef**
- **CA.field** è uguale a **CA'.field** ed è un tipo base
- **CA.field** è una istanza della stessa classe **C**
- **CA.field.cfield** risolve il *matching* con **CA'.field.cfield** per tutti i campi **cfield** di **C**

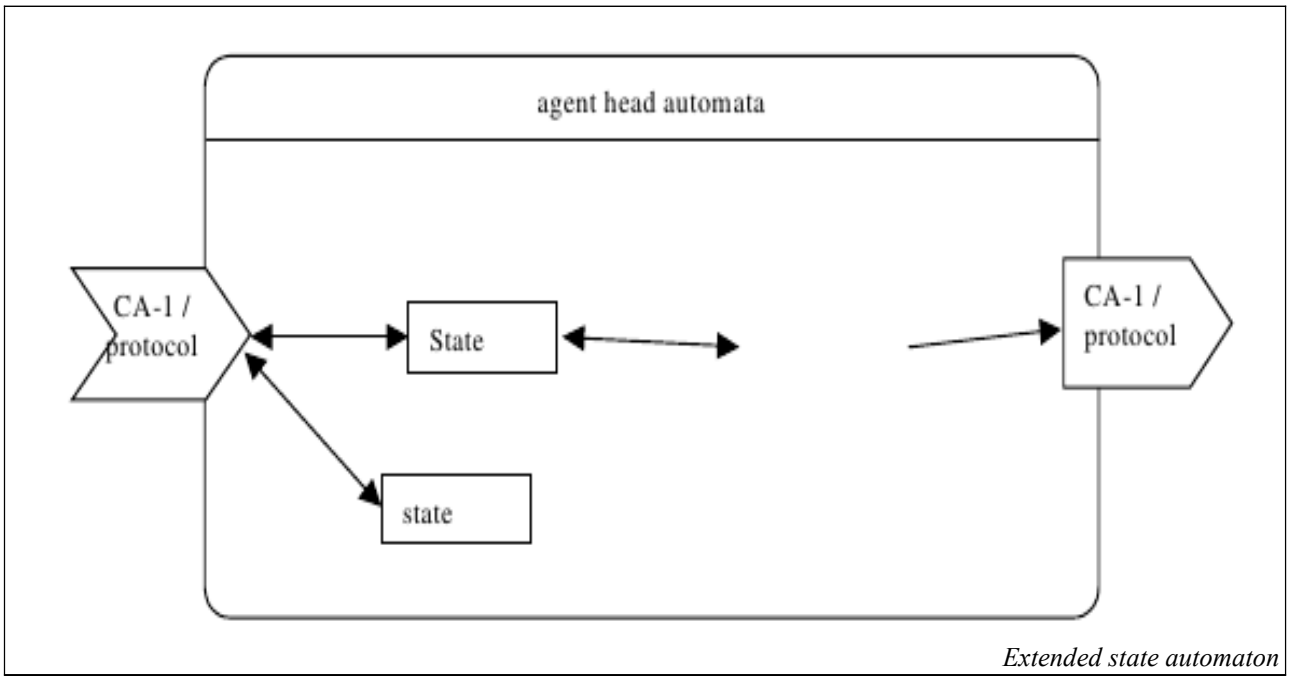
Nel caso di *atti comunicativi* in un preciso contesto, **protocol[CA]** risolve il *matching* con **protocol'[CA']** se e solo se:

- **CA** risolve il matching con **CA'**
- **protocol** è uguale a **protocol'**

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*.

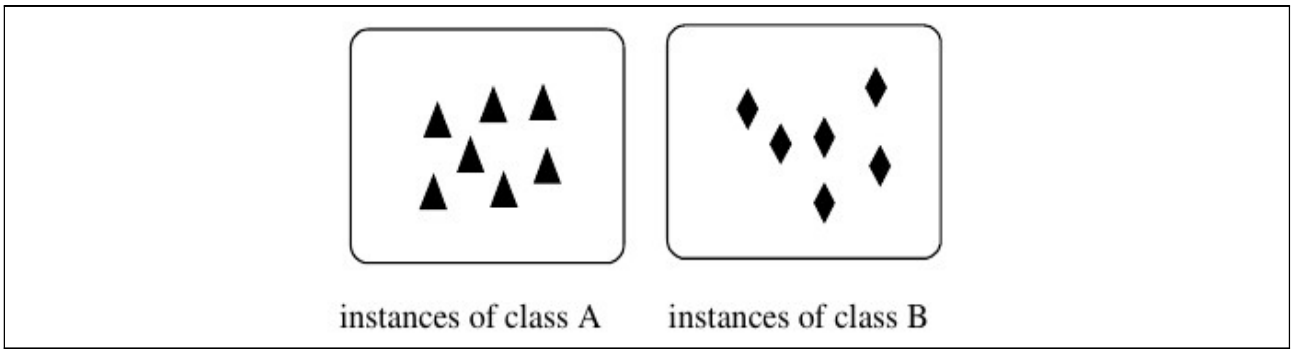
UML supporta quattro tipi di diagrammi per la definizione di comportamenti dinamici: **diagrammi di sequenza** e **diagrammi di collaborazione** in contesti ad *oggetti*, **diagrammi di stato** e **diagrammi di attività** per il resto. I primi due sono adatti alla definizione del comportamento della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



Extended state automaton

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)

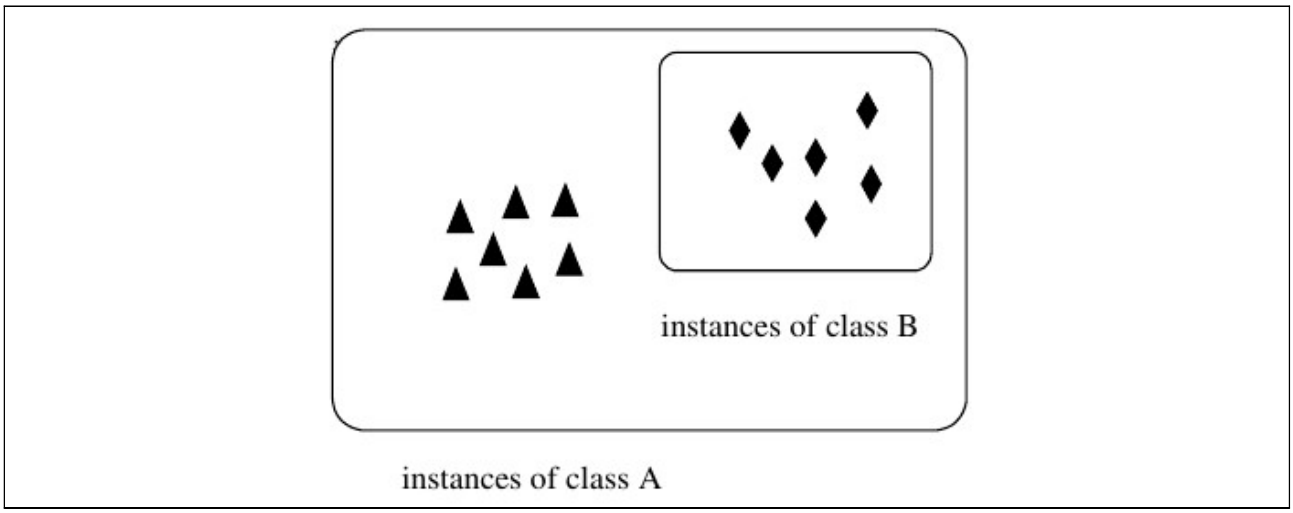
Il comportamento pro-attivo non è scatenato da CA in ingresso ma dipende dallo stato dei vincoli e delle condizioni con le quali vengono marcati gli stati iniziali dell'automa.



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 e l'interscambiabilità, supporta intrinsecamente la condivisione di un comportamento comune. Di solito un oggetto appartiene alla classe A o alla classe B ma non è possibile che un oggetto appartenga sia alla classe A che alla classe B.

Talvolta questo ci limita notevolmente: le istanze della classe B potrebbero essere istanze anche della classe A.



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 **superclasse** di B. L'**ereditarietà** *singola* permette di esprimete 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 superclassi. 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:

- una sottoclasse eredita tutti i ruoli della superclasse

- stato interno:
- possono essere definiti campi addizionali, con nuovi nomi e tipi
- possono essere inizializzati i campi definiti nella superclasse
- l'inizializzazione può essere ridefinita

- azioni:
- sono ereditate dalla superclasse
- 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 superclasse 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 dall'**agent head automaton**

- Agent Head Automaton:
- sono ereditati dalla superclasse
- possono essere ridefiniti con lo stesso nome
- può essere aggiunto comportamento addizionale

3. Modellare l'interazione in 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. 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.

Ogni diagramma 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 e condizioni opzionali. Le istanze possono essere

create o distrutte, come si può intuire dal fatto che le *lifeline* inizino e terminino. Possono essere inseriti nel modello anche vincoli o note.

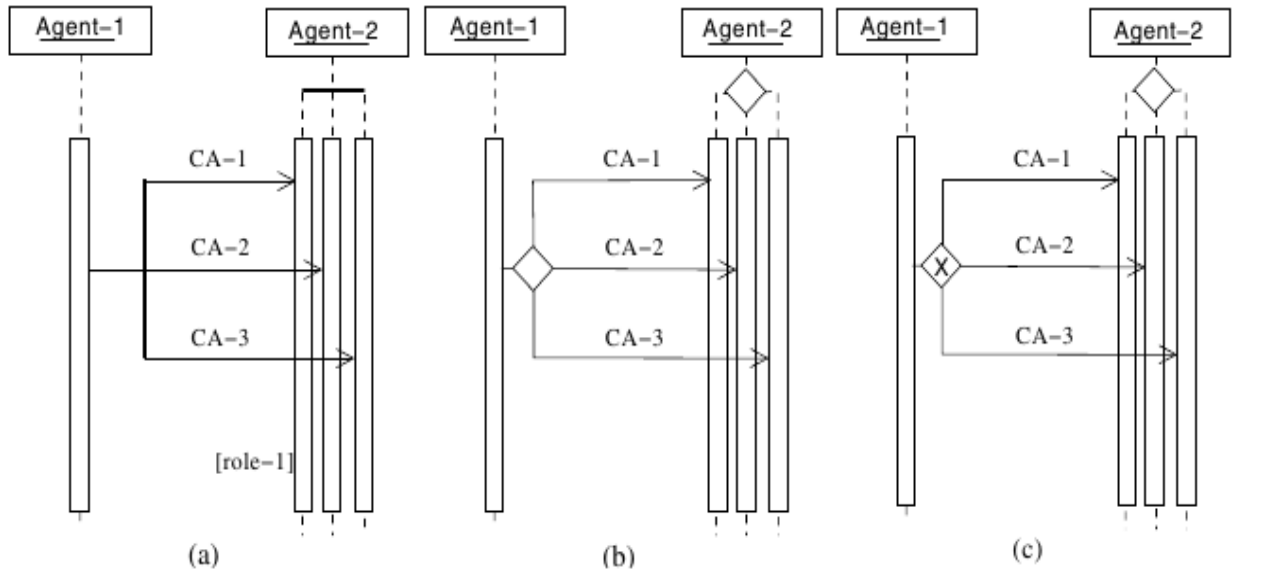
Ad un livello di astrazione molto alto i **diagrammi di sequenza** possono essere utilizzati per modellare degli *use case*, cioè intendendo l'interazione come uno *use case* tra gli attori e il sistema stesso. I diagrammi di sequenza possono essere raffinati ulteriormente durante il processo di sviluppo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

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.



Different notations for inter-agent communication

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.

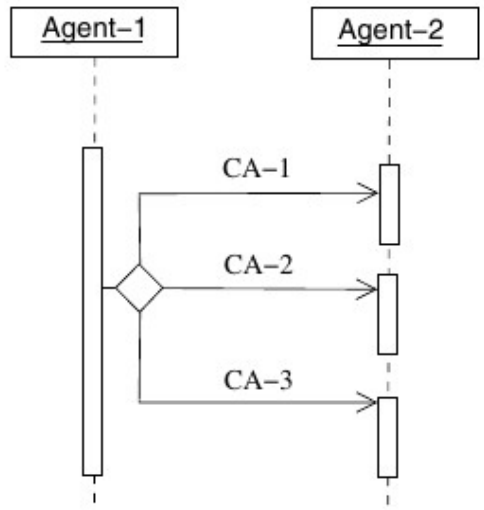
(a) **Agent-1** che invia messaggi multipli concorrenti ad **Agent-2**. I messaggi inviati denotano gli atti comunicativi **CA-1**, **CA-2** e **CA-3**. **Agent-2** elabora questi messaggi parallelamente, infatti **Agent-2** potrebbe ricoprire ruoli differenti contemporaneamente, ognuno dei quali elaborare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

(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.

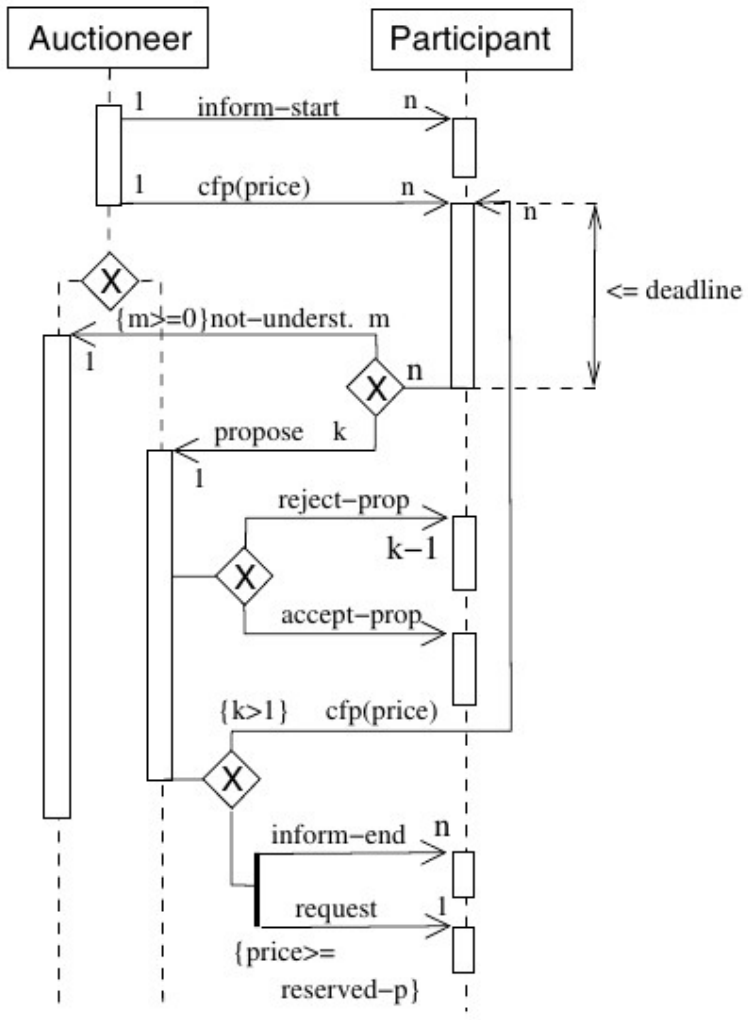


Alternative notation to the inclusive or case

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.

Le istanze hanno il loro nome sottolineato. Per esempio **o/Role:C** rappresenta un oggetto **o** della classe **C** che ricopre il ruolo **Role**. Questo è facilmente adattato agli agenti: **a/Role** può essere usato per rappresentare un agente **a** che ricopre il ruolo **Role**. Se indichiamo solo il nome del ruolo anziché dell'istanza, non dobbiamo sottolinearlo. Visto che possiamo avere diversi agenti che ricoprono diversi ruoli, è comuni in AUML aggiungere una molteplicità ai messaggi per indicare quando questi sono esplicitamente inviati a più di un agente.

Di seguito viene riportato il diagramma di quello che viene definito **protocollo di interazione dell'asta all'inglese**. In un'asta all'inglese, il banditore prova a vendere un oggetto proponendo inizialmente un prezzo al di sotto del valore di mercato e quindi aumentandolo gradualmente. Ogni volta che viene annunciato il prezzo, il banditore aspetta che i compratori segnalino la loro intenzione di offrire il prezzo proposto. Quando un compratore segnala la sua intenzione di comprare l'oggetto, il banditore aumenta il prezzo di vendita e aspetta una nuova segnalazione da parte di un compratore. Se nessun partecipante è disposto a pagare di più, l'oggetto viene venduto al prezzo dell'ultima offerta ricevuta, fintanto che il prezzo offerto è maggiore del prezzo limite precedentemente fissato. Il banditore informa quindi i partecipanti che l'asta è terminata.



English Auction Interaction Protocol

Il **banditore** informa i **partecipanti** che l'asta è cominciata (**inform-start**) e annuncia il prezzo iniziale (**cfp – call for price**). Entrambi i messaggi hanno una molteplicità di tipo *one-to-many*, dove **n** è un numero finito di agenti col ruolo di **partecipanti** all'asta. Il problema di usare **n** è doppio: primo, non è conforme allo standard di UML per indicare molteplicità; secondo, impone che il numero di agenti sia lo stesso in ogni momento. Non permette ad un agente di unirsi all'asta dopo **inform-start**. Prima della *deadline* ogni **partecipante** decide di rispondere o che non ha capito o che accetta l'offerta (**propose**). A questo punto le molteplicità sembrano indicare che tutti gli agenti rispondono o che non hanno capito o che accettano. Ipoteticamente **m** indica il numero di agenti che decide di non aver capito e **k** il numero che invece accetta l'offerta, conseguentemente **k** + **m** = **n**. Il **banditore** quindi decide o di accettare o di rifiutare l'offerta. Di nuovo la molteplicità indica che il banditore accetta 1 offerta e rifiuta tutte le altre; non è possibile però identificare il **partecipante** che corrisponde all'offerta accettata. La *decision box* permette al **banditore** di sceglierne uno qualsiasi. Dopodiché il **banditore** decide se continuare con l'asta inviando un nuovo **cfp** (se più di un **partecipante** si era mostrato interessato all'offerta precedente) o se chiudere l'asta. In quest'ultimo caso, il **banditore** invia un messaggio a tutti i **partecipanti** parallelamente e sempre parallelamente invia un messaggio all'ultimo offerente se il prezzo è maggiore del prezzo limite. Anche qui non è possibile identificare l'agente che ha inviato l'ultima offerta accettata.

Questo diagramma però presenta due possibili scenari che risultano incongruenti con la logica

del modello: un agente potrebbe ricevere un messaggio che lo informa che l'oggetto in questione è stato venduto a lui anche se non aveva fatto offerte e il banditore potrebbe decidere di chiudere l'asta anche nel caso **k** > 1. Inoltre ci sono altre questioni da analizzare:

1. Il diagramma rappresenta un possibile scenario o tutti quelli possibili?
2. Nella prima *decision box* c'è una molteplicità **n**. Significa che **n** agenti devono, o possono, rispondere in una delle due maniere? Il fatto che gli agenti sono autonomi suggerirebbe un "possono", ma le norme di comportamento dei partecipanti ad un'asta impongono un "devono". Dovrebbe essere possibile discriminare le due alternative.
3. I messaggi possono andare perduti?
4. Come possiamo inviare un messaggio ad un preciso agente nel set?
5. Come possiamo garantire che alla fine il banditore deciderà effettivamente di chiudere l'asta?
6. Come possiamo esprimere che un comportamento sia vietato?

4. A-nets, IP-nets e MIP-nets

Le reti di Petri sono ampiamente utilizzate per la specifica di processi e la loro verifica. Una rete di Petri standard consiste in un grafo a rete composto da *posti*, rappresentati da cerchi, *transizioni*, rappresentate da rettangoli e *archi*, rappresentati da frecce. Dato un set di identificatori **U**, una struttura di reti di Petri o, semplicemente, una rete di Petri, è data dalla tripla **(P, T, F)**, dove $\mathbf{P \subseteq U}$ e $\mathbf{T \subseteq U}$ sono insiemi disgiunti non vuoti di *posti* e *transizioni* rispettivamente e $\mathbf{F \subseteq (P \times T) \cup (T \times P)}$ è il set degli *archi*. I componenti di una rete **N** sono anche denotati da $\mathbf{P_N}$, $\mathbf{T_N}$ e $\mathbf{F_N}$. Un *cammino* di una rete è una sequenza non vuota $\mathbf{x_1\ x_2 \ldots x_k\ (k \in \mathbb{N})}$ di elementi della rete che soddisfa

$$\mathbf{(x_1,x_2), (x_2,x_3), \ldots, (x_{k-1},x_k) \in F}$$

Una rete è *fortemente connessa* se e solo se per ogni coppia di elementi **x** e **y** esiste un cammino che conduce da **x** ad **y**. Per gli $\mathbf{x \in P \cup T}$ si definisce $\mathbf{{}^\bullet x = \{y \mid (y,x) \in F\}}$ *preset* di **x** e $\mathbf{x^\bullet = \{y \mid (x,y) \in F\}}$ *postset* di **x**.

Si può identificare nei *posti* l'insieme dei possibili stati del sistema, nelle *transizioni* gli eventi che modificano lo stato del sistema e negli *archi* i collegamenti da un uno all'altro o viceversa. In altre parole un *arco* che collega un *posto* ad una *transizione* indica da quale stato del sistema si origina l'evento, mentre un *arco* che collega una *transizione* ad un *posto* indica verso quale stato transita il sistema in risposta all'evento. In ogni momento, un *posto* contiene 0 o più *token* identificati da pallini neri. Lo stato **M**, chiamato *marking*, è la distribuzione dei *token* all'interno dei vari *posti*. $\mathbf{M: P \rightarrow N}$ viene definito un *mapping*; per esempio **2p1 + p2 + 3p3** identifica lo stato in cui abbiamo 2 *token* nel *posto* **p1**, 1 *token* nel *posto* **p2** e 3 *token* nel *posto* **p3**. Se un *posto* è marcato da un *token* si considera la condizione associata come verificata. Un sistema di reti di Petri è la coppia $\mathbf{(N, M_0)}$ dove **N** è una struttura a rete e $\mathbf{M_0}$ è un *marking* definito *marking iniziale* (stato iniziale) del sistema.

La dinamica di una rete di Petri è governata da quella che viene definita *firing rule*. Una *transizione* può scattare se c'è almeno 1 *token* in ciascuno dei *posti* in ingresso, interpretabile come il verificarsi di tutte le precondizioni; questa *transizione* viene quindi detta *abilitata*. Una *transizione abilitata* scatta rimuovendo 1 *token* da ognuno dei *posti* in input e depositando 1 *token* in ognuno dei *posti* in output, attualizzando cioè le post-condizioni. Questo movimento dei *token* da un *posto* all'altro indica un cambiamento dello stato del sistema in risposta all'evento. La notazione $\mathbf{M1 \xrightarrow{t} M2}$ definisce una *transizione* **t** abilitata in **M1** e che scattando porta il sistema in **M2**. Se

esistono delle *transizioni* $\mathbf{M}^{t1}\rightarrow\mathbf{M}_1{}^{t2}\rightarrow \ldots {}^{tn}\rightarrow\mathbf{M}_n$ allora si può definire $\boldsymbol{\sigma}$ = **t1 t2 … tn** una *sequenza di occorrenze* che porta da **M** ad $\mathbf{M}_n$ e si abbrevia come $\mathbf{M}^{\sigma}\rightarrow\mathbf{M}_n$. $\mathbf{M}_n$ è raggiungibile da **M** <u>se e solo se</u> esiste almeno una sequenza $\boldsymbol{\sigma}$ tale che $\mathbf{M}^{\sigma}\rightarrow\mathbf{M}_n$ e si rappresenta come $\mathbf{M}\rightarrow^*\mathbf{M}_n$. **[M⟩** rappresenta il set di tutti i possibili stati (*marking*) raggiungibili da **M**.

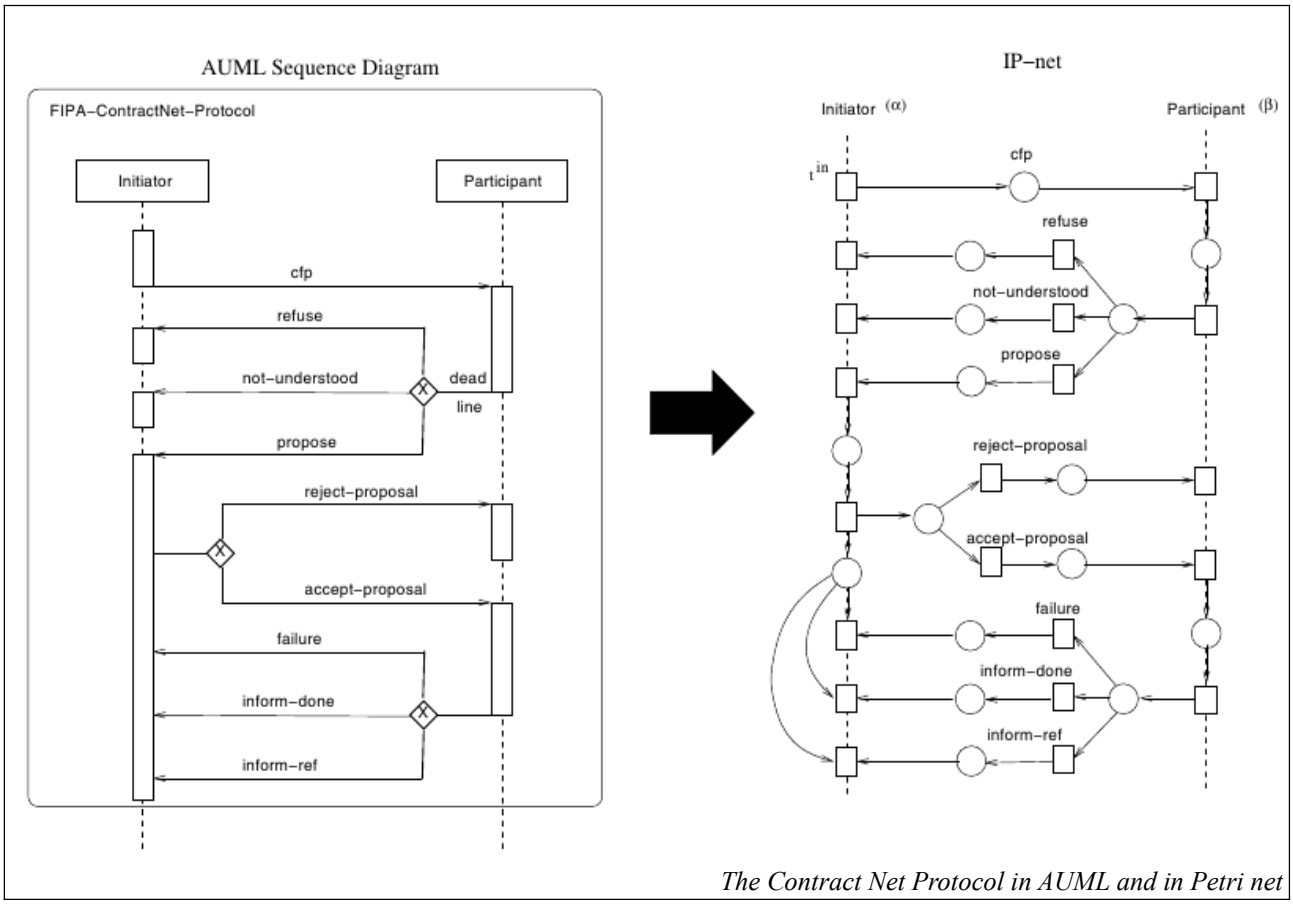
Sono state definite un gran numero di proprietà per le reti di Petri. Un sistema di reti di Petri è *vivo* se, per ogni *marking* **M** e ogni *transizione* **t**, esiste un *marking* $\mathbf{M'}\in\mathbf{[M\rangle}$ che abilita **t**. Un sistema è *bounded* se per ogni *posto* **p**, c'è un numero naturale **n** tale che $\mathbf{M(p)}\leq\mathbf{n}$ per ogni *marking* raggiungibile **M**. Il sistema è *1-bounded* o *safe* se **n=1**.

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 nets* (*WF-nets*), è stata adattata per modellare il comportamento di agenti. Una rete di Petri che modella il comportamento di un agente viene chiamata a*gent net* (*A-net*).

Una rete di Petri **A=(P,T,F)** è una *agent net* <u>se e solo se</u>:

1. ha due *posti* speciali **in** e **out**, dove $\bullet\mathbf{in}=\varnothing$ e $\mathbf{out}\bullet=\varnothing$
2. aggiungendo una transizione t* ad A, la rete cortocircuitata $(P,T\cup\{t^*\},F\cup\{(out,t^*),(t^*,in)\})$ 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 *posti* e le *transizioni* contribuiscano al comportamento generale dell'agente.



The Contract Net Protocol in AUML and in Petri net

Il diagramma rappresenta la specifica del protocollo di interazione nello scenario di una proposta

di contratto. L'attore identificato come **initiator** invia una **call-for-proposal** (**cfp**) al **participant**. Prima di raggiungere una deadline, il **participant** può o rifiutare, inviando **not-understood** o accettare generando un **propose**. Se **initiator** riceve un **propose** può o rifiutare inviando un **reject-proposal** o accettare con un **accept-proposal**. Dopo l'accettazione, il **participant** esegue il task fino alla fine e invia uno dei tre messaggi **failure**, **inform-done** e **inform-ref**. Per ogni agente sono presenti diverse barre verticali, ognuna delle quali rappresenta potenzialmente un ruolo, o più in generale, un processo, diverso. La separazione di queste barre permette all'agente di avere un altrettanto numero di *lifeline*.

La descrizione di un protocollo 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 $\mathbf{t^{in} \in T}$ chiamata *protocol input transition* tale che $\mathbf{\bullet t^{in} = \varnothing}$
2. **IP** ha uno speciale set di *transizioni* $\mathbf{T^{out}}$ tale che per ogni $\mathbf{t \in T^{out}}$, $\mathbf{t\bullet = \varnothing}$ e $\mathbf{t^{in} \notin T^{out}}$
3. ci sono due classi disgiunte di *transizioni* $\mathbf{T^{\alpha} \subset T}$ e $\mathbf{T^{\beta} \subset T}$ tali che $\mathbf{t^{in} \in T^{\alpha}}$ e $\mathbf{T^{out} \in T^{\alpha} \cup T^{\beta}}$

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 necessariamente di α, le $\mathbf{T^{out}}$ possono appartenere anche a β. Nell'esempio, le transizioni $\mathbf{T^{out}}$ sono quelle senza *archi* in uscita.

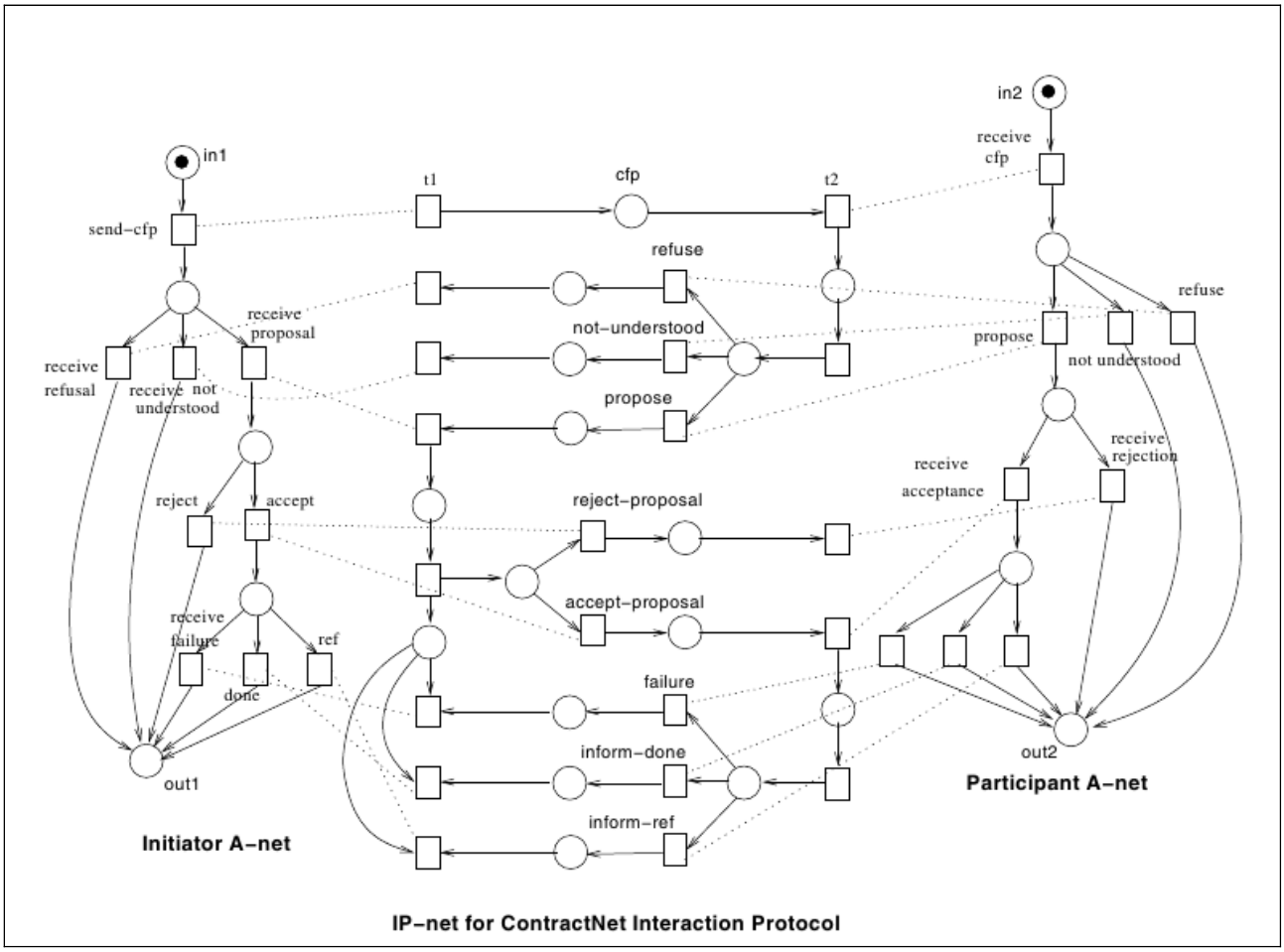
Nel protocollo un agente può avere *lifeline* multiple, denotate da barre verticali separate. I messaggi possono essere inviati e ricevuti all'interno dello stesso thread o in thread indipendenti in accordo con la specifica. Questa caratteristica viene modellata avendo differenti *transizioni* di invio e ricezione nella *IP-net*. L'agente **initiator** invia **cfp** tramite $\mathbf{t^{in}}$ indipendentemente da altri eventi, però, se viene ricevuto un **propose** da **participant**, deve necessariamente inviare **reject-proposal** o **accept-proposal** e aspettare di ricevere un messaggio di **failure**, **inform-done** o **inform-ref** all'interno dello stesso thread.

Un sistema multi-agente 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-nets* e *IP-nets* utilizzando alcuni elementi di sincronizzazione.

Una *MIP-net* è una tupla $\mathbf{MIP=(A_1,A_2, \dots ,A_n,IP_1,IP_2, \dots ,Ip_k,T_{SC},SC)}$ tale che:
1. $\mathbf{n,k \in \mathbb{N}}$, dove **n** è il numero di *A-nets* e **k** è il numero di *IP-nets* e $\mathbf{n-1 \leq k \leq n(n-1)/2}$
2. per ogni $\mathbf{i \in \{1, \dots ,n\}}$, $\mathbf{A_i}$ è una *A-net* con un *posto* iniziale $\mathbf{in_{Ai}}$ e uno finale $\mathbf{out_{Ai}}$
3. per ogni $\mathbf{i \in \{1, \dots ,n\}}$, $\mathbf{IP_i}$ 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. $\mathbf{SC \subseteq T_{SC} \times T^{\Diamond} \times T^{\Box}}$ è la legge di sincronizzazione:
   $\mathbf{T^{\Box} = U_{i \in \{1, \dots ,n\}} T_{Ai}}$, $\mathbf{T^{A} = U_{i \in \{1, \dots ,k\}} T^{\alpha}_{IPi}}$, $\mathbf{T^{B} = U_{i \in \{1, \dots ,k\}} T^{\beta}_{IPi}}$, $\mathbf{T^{\Diamond} = T^{A \cup} T^{B}}$
6. per ogni $\mathbf{t \in T_{SC}}$, $\mathbf{\{(t',x,y) \in SC | t' = t\}}$ è un *singleton*

7. per ogni $(t_1,x_1,y_1),(t_2,x_2,y_2) \in \mathbf{SC}$, se $t_1 \neq t_2$ allora $x_1 \neq x_2 \wedge y_1 \neq y_2$
8. per ogni $(t_1,x_1,y_1),(t_2,x_2,y_2) \in \mathbf{SC}$, se y_1 e y_2 sono della stessa *A-net* allora $x_1 \in T^{\alpha}_{IP} \Rightarrow x_2 \notin T^{\beta}_{IP}$ per una particolare *IP-net* **IP**

Il requisito (1) definisce la relazione tra il numero di agenti e di protocolli nel sistema multi-agente, 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 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*.

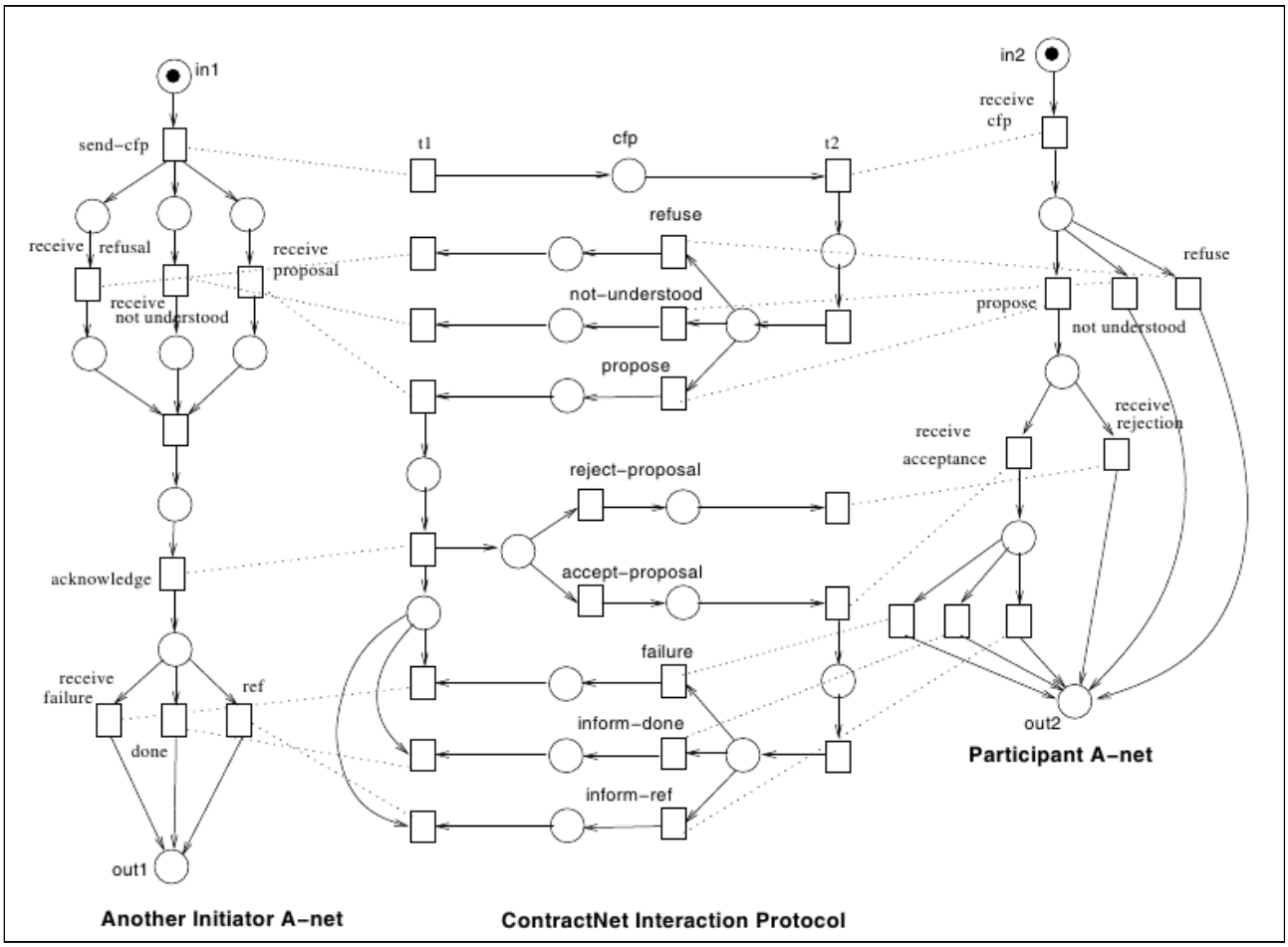


Uno dei motivi per modellare i sistemi multi-agente attraverso delle reti di Petri è il poter utilizzare tutti i metodi già ben ampiamente affermati per l'analisi di queste. Tali metodi sono comunemente utilizzati per identificare le proprietà di *liveness* e *boundedness* dei sistemi modellati. Nei sistemi multi-agente, 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-nets* sono essenzialmente *WF-nets* 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** <u>se e solo se</u>:

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 *posto* **out** al *posto* **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*. Il teorema stabilisce che un agente, o una *A-net* **A**, è **sound**, <u>se e solo se</u> 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".



Sia **MIP=(A_1,A_2, … ,An,IP_1,IP_2, … ,Ip_k,T_{SC},SC)** una *MIP-net*. **U(MIP)=(P^U,T^U,F^U)** è il **dispiegamento** (**unfolding**) di **MIP**, dove:

1. $P^U=P_{A1}\cup P_{A2}\cup \ldots \cup P_{An}\cup P_{IP1}\cup P_{IP2}\cup \ldots \cup P_{IPk}\cup\{i,o\}$
2. $T^U=r(T^{\square}\cup T^{\Diamond})\cup T_{SC}\cup(U_{j\in\{1,\ldots,k\}}T_{IPj}\setminus T^{\Diamond})\cup\{t_i,t_o\}$
3. $\{i,o,t_i,t_o\}\cap\{P_{A1}\cup P_{A2}\cup \ldots \cup P_{An}\cup P_{IP1}\cup P_{IP2}\cup \ldots \cup P_{IPk}\cup T_{sc}\cup(U_{j\in\{1,\ldots,k\}}T_{IPj}\setminus T^{\Diamond})\}=\varnothing$
4. **r** è la funzione di rinomina: $r(x)=t_{sc}$ se c'è un $t_{sc}\in T_{SC}$ e $x,y\in\{T^{\square}\cup T^{\Diamond}\}$ in modo tale da avere o $(t_{sc},x,y)\in SC$ o $(t_{sc},y,x)\in SC$; altrimenti $r(x)=x$.
5. $F^{\square}=F_{A1}\cup F_{A2}\cup \ldots \cup F_{An}\cup F_{IP1}\cup F_{IP2}\cup \ldots \cup F_{IPk}$
6. $F'=F^{\square}\cup\{(i,t_i),(t_o,o)\}\cup\{(t_i.in_{Aj})|j\in\{1,\ldots,n\}\}\cup\{(out_{Aj},t_o)|j\in\{1,\ldots,n\}\}$
7. $F^U=\{(r(x),r(y))|(x,y)\in F'\}$

Il **dispiegamento** mira a creare una singola e globale *A-net* dalla *MIP-net* introducendo due nuove *transizioni* (**t_i** e **t_o**) e due nuovi *posti* (**i** e **o**). Intuitivamente, i *posti* **i** e **o** diventano rispettivamente i *posti* **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.

Una *MIP-net* è **sound** <u>se e solo</u> 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

Nell'ultima figura viene illustrata la *MIP-net* derivante dal particolare scenario in cui l'agente **initiator** decida di abbandonare l'interazione e un altro agente intenda occupare tale ruolo. Si può verificare che anche se le *A-net* risultino **sound**, il dispiegamento della nuova *MIP-net* non lo sia, ne consegue che la stessa *MIP-net* non è **sound**.
La ragione di questo è data dal fatto che il nuovo **initiator** (sulla sinistra) implementa il suo comportamento generando 3 thread paralleli per gestire **receive refusal**, **receive not understood** e **receive proposal**, possibilmente per poter interagire con altri sistemi, prima di unirsi al protocollo corrente. Il sistema corrente non risulta quindi **sound** poiché il protocollo di interazione richiede che l'agente **initiator** accetti solamente 1 dei 3 messaggi. Per rendere il sistema **sound,** il nuovo agente dovrebbe cambiare il proprio comportamento per adattarsi al protocollo e quindi necessita di essere riprogrammato.

Dato un protocollo di interazione, possiamo costruire delle *A-net* minimali che soddisfino il protocollo. Le *A-net* nella penultima figura sono minimali, cioè utilizzano il minor numero di *transizioni receive/send* per sincronizzarsi correttamente con la *IP-net*. Tali *A-net* sono state derivate osservando direttamente la struttura della *IP-net* e garantiscono la *soundness* del sistema. Dalla *IP-net*, l'agente **initiator** deve avere una *transizione* (**send-cfp**) per sincronizzarsi con **t^{in}**, seguita da 3 *transizioni* a scelta libera (**receive refusal**, **receive not understood** e **receive proposal**) per accettare uno dei 3 possibili messaggi. Se viene ricevuta una proposta (la terza opzione), dev'essere seguita da una conferma (**aknowledge**) e da una di altre 3 transizioni a scelta libera (**receive failure**, **done** e **ref**) sulla stessa *lifeline*.
Questa è esattamente l'implementazione della *A-net* per l'agente **initiator** e può essere considerato lo scheletro del codice dell'agente stesso. La *A-net* può essere quindi identificata come rappresentante del ruolo dell'agente più che dell'agente individuale. Ci potrebbero essere più agenti partecipanti alla realizzazione concreta di un sistema multi-agente che utilizza il protocollo a contratto presentato e la A-net può essere intesa rappresentare la classe degli agenti partecipanti.

Il comportamento dell'agente, quindi anche lo scheletro del codice, può essere modificato cambiando o migliorando la *A-net* aggiungendo *transizioni* e *posti*; occorre però seguire regole di trasformazioni appropriate per garantire che i cambiamenti non inficino la soundness dell'agente o

dell'intero sistema; inoltre, nelle *A-nets* e nelle *MIP-nets*, occorre preservare il corretto ordinamento delle attività, come non è avvenuto nell'ultimo diagramma, alterando quindi il comportamento inteso nella *A-net* originale e intaccando la *soundness* della *MIP-net*.

5. Conclusioni e sviluppi futuri

Data la velocità con la quale le tecnologie informatiche si evolvono, è un'impresa alquanto ardua starne al passo e risulta praticamente impossibile riuscire a prevedere quali tecnologie risulteranno vincenti per poter indirizzare gli sforzi a supporto. Il fatto di potersi appoggiare su qualcosa di già ben consolidato, come possono essere il linguaggio di modellazione UML o le reti di Petri, garantisce perlomeno una certa sicurezza nella validità dei risultati raggiunti.

Per quanto riguarda Agent UML ci sono principalmente due vantaggi nel suo utilizzo:

- permettono a coloro che conoscono bene l'ambiente *object-oriented*, ma che sono ancora nuovi all'approccio *agent-oriented,* di comprendere le meccaniche di un sistema multi-agente e di riconoscere in questi un sistema realizzato tramite una società di agenti piuttosto che come una collezione di oggetti
- il tempo impiegato nella fase di design può essere ridotto, con una prospettiva di un'ulteriore vantaggio nel momento in cui vengano forniti dei componenti che supportino l'implementazione basata sulle specifiche di Agent UML

L'utilizzo delle reti di Petri si focalizza sul modellare le interazioni principali del comportamento degli agenti, fornendo quello che viene definito uno scheletro dell'agente, per poi poter applicare tecniche di *model checking* e validare quindi il sistema.

Tali modelli però necessitano di essere migliorati per poter includere concetti quali *intelligenza* e *autonomia*, peculiari appunto dei sistemi multi-agente; sorgono quindi diverse caratteristiche che potrebbero migliorare il modello:

- la possibilità di identificare più approfonditamente le caratteristiche dei singoli agenti, utilizzando meccanismi simili a quelli delle *coloured Petri nets*
- la capacità di valutare proprietà non funzionali come le *performance* sfruttando la nozione di tempo presenti nelle *timed Petri nets*
- la possibilità di utilizzo di protocolli aperti con la conseguente capacità degli agenti di verificare a runtime la propria capacità di uniformarsi ad un certo protocollo
- la capacità degli agenti di intervenire contemporaneamente in più protocolli, separando maggiormente il concetto di *agente* e *ruolo*

Un ulteriore arricchimento potrebbe essere quello di utilizzare tecniche di trasformazione *model-to-model* per automatizzare completamente la traduzione dei modelli AUML in MIP-nets; sia (A)UML che le reti di Petri, infatti, supportano una rappresentazione testuale che si adatterebbe perfettamente ad un tale approccio.

Una volta validato il modello, inoltre, sarebbe possibile realizzare gli scheletri sia degli agenti che dei protocolli realizzanti il sistema multi-agente, tramite l'utilizzo di tecniche di generazione automatica *model-to*-code (eventualmente, qualora ci si appoggi su piattaforme situate su layer superiori, si tratterebbe di utilizzare di nuovo una trasformazione *model-to-model*).

Così facendo sarà possibile dedicare la maggior parte degli sforzi concettuali alla modellazione e alla progettazione delle interazioni del sistema, che ne costituiscono di fatto la componente principale e caratterizzante, avendo oltretutto a disposizione un (potenzialmente) veloce metodo di

generazione di eventuali prototipi del sistema stesso.

Anche se l'approccio ad agenti risulta relativamente nuovo, tramite l'utilizzo di queste metodologie, sebbene i modelli presentati non siano omni-comprensivi e sicuramente richiedano ulteriori raffinamenti, è disponibile un buon supporto alla modellazione e alla progettazione di sistemi ad agenti.

Bibliografia

GIVING LIFE TO AGENT INTERACTIONS
Juliana Küster Filipe
Laboratory for Foundations of Computer Science, School of Informatics
The University of Edinburgh, King's Buildings
Edinburgh EH9 3JZ, United Kingdom
jkfilipe@inf.ed.ac.uk

MIP-NETS: A COMPOSITIONAL MODEL OF MULTIAGENT INTERACTION
Sea Ling and Seng Wai Loke
School of Computer Science and Software Engineering, Monash University
P O Box 197, VIC 3145, Australia
sling@csse.monash.edu.au, swlok@csse.monash.edu.au

UML CLASS DIAGRAMS REVISITED IN THE CONTEXT OF AGENT-BASED SYSTEMS
Bernhard Bauer
Siemens AG, Corporate Technology, Information and Communications
Otto-Hahn-Ring 6
81739 Munich, Germany
bernhard.bauer@mchp.siemens.de