ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE - CESENA

A&A in Unity: Primi Esperimenti

Relazione per il corso di Sistemi Autonomi

Alessandro Bagnoli - 763930 - alessandro.bagnoli4@studio.unibo.it

A.A. 2016-2017

INDICE

1 Introduzione

1.1 Multi Agent Systems

I modelli e le tecnologie agent-oriented si sono dimostrati essere utili in tutte quelle situazioni in cui è richiesta la presenza di entità autonome le cui azioni e interazioni determinano l'evoluzione del sistema, e le discipline e campi di ricerca interessati nello studiare gli effetti dell'interazione locale-globale, auto-organizzazione, eterogeneità sono in aumento [1]. Un multi-agent system (MAS) fornisce agli sviluppatori e ai designer tre astrazioni principali:

- *Agenti*: Le entità atomiche che compongono il sistema. Sono autonomi, eterogenei, in grado di comunicare, intelligenti, dinamici, e situati.
- *Società*: Rappresenta un gruppo di entità il cui comportamento emerge dall'interazione tra i singoli elementi.
- *Ambiente*: Il "contenitore" in cui gli agenti sono immersi e con il quale questi ultimi possono interagire, modificandolo. La caratteristica degli agenti di essere situati nell'ambiente in cui si trovano permette loro di percepire e produrre cambiamenti su di esso.

Studi successivi hanno portato ad ideare un nuovo meta-modello per i MAS, in cui le entità immerse nell'ambiente non sono solo gli agenti, ma anche degli oggetti, strumenti che possono influenzare i processi cognitivi degli agenti e che possono essere utilizzati per i loro obiettivi. Questo meta-modello viene chiamato Agents and Artifacts (A&A), in cui l'*artefatto* rappresenta la nuova entità appena introdotta [2].

Come il nome lascia intuire, secondo questo meta-modello i MAS sono modellati sulla base di due fondamentali astrazioni computazionali, gli agenti e gli artefatti. Gli agenti rimangono le entità autonome, pro-attive che incapsulano il controllo e determinano il comportamento dell'intero MAS, come descritto precedentemente. Gli artefatti invece sono le entità passive e reattive che forniscono i servizi e le funzioni che permettono ai singoli agenti di lavorare insieme in un MAS, e danno forma all'ambiente in accordo con le esigenze del MAS. Viene inoltre introdotta una nuova astrazione, chiamata *workspace*, definita come container concettuale degli agenti e degli artefatti, utile per definire la topologia dell'ambiente e che fornisce un modo per specificare una nozione di località [2]. Si può quindi pensare al workspace come ad una porzione dell'ambiente.

1.2 Game Engines

I Game Engines (GE) sono framework resi disponibili per i designer di videogiochi che permettono di programmare e pianificare un gioco senza costruirne uno completamente da zero, ed offrono strumenti per coadiuvare la creazione e la disposizione delle risorse di gioco. Forniscono aspetti avanzati quali rendering di modelli 2D e 3D, motori fisici, rilevamento delle collisioni, etc. [3]. Come punto di riferimento da ora in avanti consideriamo il GE Unity3D, del quale si elencano le astrazioni principali messe a disposizione del designer:

- *GameObject*: La classe base per tutte le entità presenti su una scena di Unity: un personaggio controllabile dall'utente, un personaggio non giocabile, un oggetto. Tutto ciò che è presente sulla scena è un GameObject.
- *Script*: Codice sorgente applicato a un GameObject, grazie al quale è possibile assegnare a quest ultimo un comportamento. Gli script vengono eseguiti dal game loop di Unity, che in maniera sequenziale esegue una volta ogni script, durante ogni frame del gioco renderizzato. Non esiste concorrenza.
- *Co-routine*: Una soluzione alla sequenzialità imposta agli script, grazie al quale è possibile partizionare una computazione e distribuirla su più frame, sospendendo e riprendendo l'esecuzione in precisi punti del codice.

Come si può notare, l'abstraction gap presente tra MAS e GE è piuttosto marcato e l'unica astrazione dei MAS rappresentata e supportata nei GE è quello dell'ambiente, attraverso i game object e le capacità di Unity di supportare molte forme di interazione agente-ambiente, ad esempio il rilevamento delle collisioni o la capacità di evitare gli ostacoli.

2 Integrazione e mapping tra Game Engines e MAS: I primi passi

I primi passi verso l'integrazione tra GE e MAS comprendono l'implementazione in Unity di alcuni modelli tipici dei MAS, che verranno brevemente descritti:

- Un modello di coordinazione degli agenti tramite spazio di tuple e primitive Linda [4].
- Il modello Beliefs, Desires, Intentions (BDI) per la programmazione degli agenti [5].

2.1 Spazio di tuple

Il punto cardine del modello di coordinazione in analisi è il medium di coordinazione, rappresentato dallo spazio di tuple. Il mapping di quest ultimo all'interno di Unity prevede l'utilizzo di un game object con uno script allegato. Questa implementazione ha tre campi fondamentali: uno in cui vengono salvate le tuple, uno per memorizzare le richieste ancora in attesa, e l'ultimo per memorizzare le richieste appena arrivate. Il comportamento viene poi specificato nel metodo Update dello script: ad ogni frame si cercherà prima di gestire tutte le richieste in attesa, e poi quelle appena arrivate.

Gli agenti, anch'essi mappati su game object e script allegato, inviano le richieste e ricevono le risposte dal centro di tuple tramite scambio di messaggi.

2.2 Modello BDI

Il modello BDI proposto per Unity prevede, come prima, un mapping degli agenti tramite game object e script, che avrà alcune proprietà fondamentali: il nome dell'agente, tre liste (una per i belief, una per i desire, e una per i plan) e un reasoner. Il reasoner si occupa della selezione del piano da applicare. Un piano è formato da un desire, una lista di belief e le azioni che dovrà svolgere l'agente nel caso in cui il piano sia applicabile. L'agente è in grado di percepire l'ambiente (formato da game objects) grazie al rilevatore di collisioni già integrato in Unity, e quindi usufruibile "gratuitamente": quando avviene la collisione tra un agente e un oggetto percepibile presente sulla scena, viene aggiunto un belief alla belief base dell'agente. L'architettura del modello descritto può essere osservata in Figura 2.1.

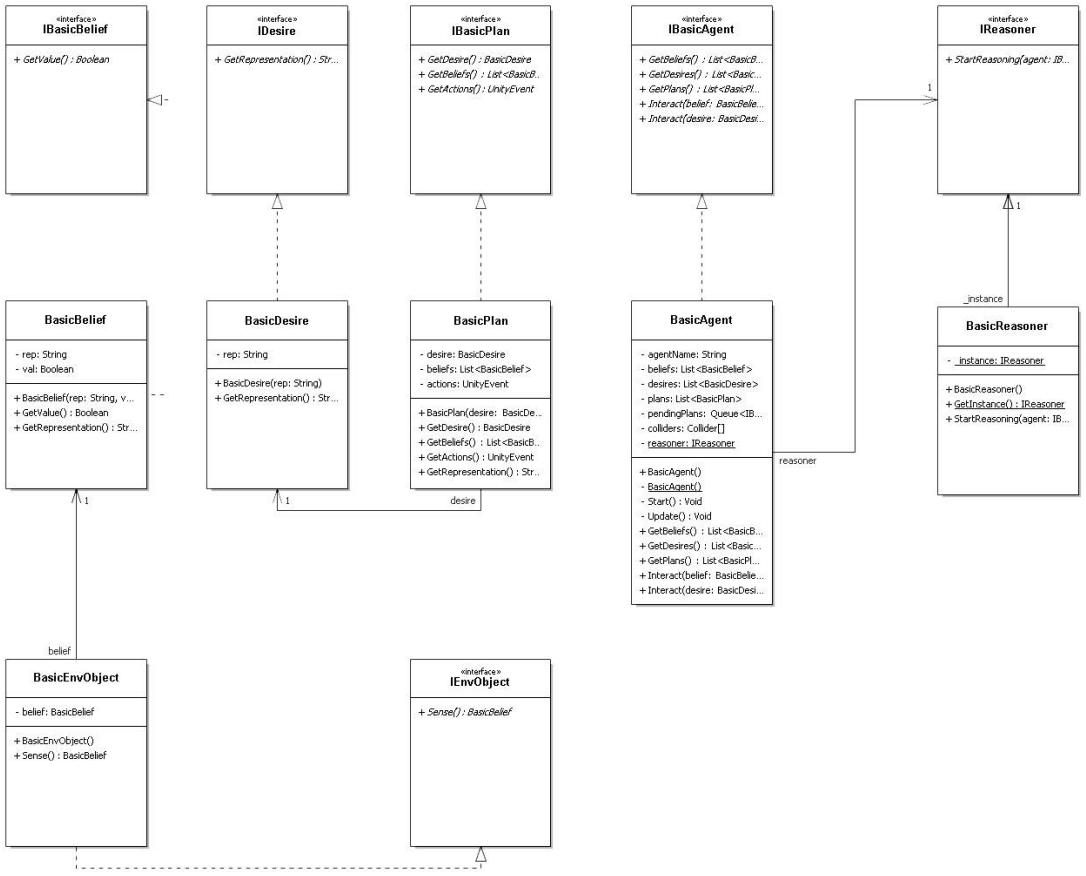

Figura 2.1: Architettura di un agente BDI in Unity [5].

3 MODELLAZIONE DI A&A SU UNITY3D

Come si può notare, i primi esperimenti di integrazione non includono un mapping ben delineato tra le astrazioni di un MAS organizzato secondo il meta-modello A&A descritto nell'introduzione e Unity, si cercherà quindi di formalizzarlo in maniera più esplicita nel corso di questa sezione, avvalendosi del lavoro fino a qua svolto dai pionieri di questa integrazione.

Come affermato nell'introduzione, il meta-modello A&A è caratterizzato da tre astrazioni basilari: gli agenti, gli artefatti e i workspace. Viene ora proposto un primo mapping concettuale per le due entità che non sono ancora state rappresentate su Unity: gli artefatti e i workspace.

3.1 ARTEFATTI

Un artefatto A&A è una entità computazionale destinata ad essere utilizzata da agenti A&A [2]. Secondo il meta-modello A&A, un artefatto deve avere le seguenti proprietà:

- *Function Description*: La funzione che l'artefatto fornisce agli agenti.
- *Operating Instructions*: Le procedure ammissibili per utilizzare un artefatto.
- *Usage interface*: La struttura esterna, osservabile dagli agenti, dell'artefatto.

L'artefatto può quindi essere mappato in ambiente Unity come un game object con uno script che conterrà una lista di proprietà osservabili, eventualmente modificabili, e le funzioni che possono essere eseguite con l'artefatto in questione, le quali potranno modificare sia le proprie proprietà osservabili, sia lo stato dell'ambiente. Le proprietà osservabili dell'artefatto saranno aggiunte alla belief base dell'agente quando quest ultimo inizierà ad osservare lo stato attuale dell'artefatto, come viene già fatto nell'architettura BDI descritta nel Capitolo 2 quando viene percepito un oggetto della scena da parte di un agente, con la differenza che l'agente in questo caso può decidere quando osservare un artefatto, senza la necessità che ci sia una collisione tra di essi. In questo modo, ogniqualvolta che una proprietà osservabile viene aggiornata, anche il belief corrispondente viene aggiornato per tutti gli agenti che stanno osservando l'artefatto.

3.2 WORKSPACE

Data una intera scena di gioco in Unity, un workspace A&A può essere interpretato e quindi mappato fisicamente su tutta la scena di Unity, o su una specifica parte di essa. Una condizione necessaria affinché un agente possa usare un artefatto è che quest ultimo sia presente nel workspace in cui l'agente è situato. Viceversa, gli eventi generati dagli artefatti di un workspace possono essere osservati solo dagli agenti situati nello stesso workspace [6]. Sfruttando le astrazioni Unity, il workspace è quindi rappresentabile come game object in grado di essere calpestato o attraversato tramite collisioni con uno script che tiene traccia a runtime degli agenti e artefatti presenti sulla sua superficie (o al suo interno), in modo tale da permettere agli agenti di osservare ed utilizzare gli artefatti solo quando situati nello stesso workspace. Una modellazione fatta in questo modo permette agilmente anche la creazione di topologie più articolate, come ad esempio l'intersezione e l'annidamento dei workspace.

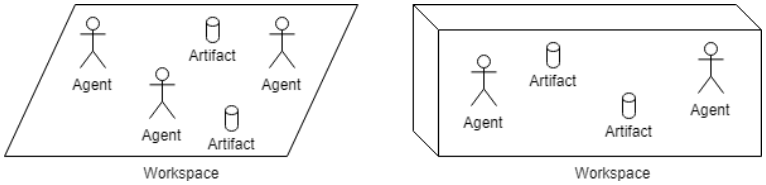

Figura 3.1: Esempi di workspace rappresentati come superficie calpestabile (a sinistra) e volume attraversabile (a destra).

4 GLI ULTERIORI SVILUPPI AVVENUTI NEL CORSO DEL TEMPO

La progettazione ed implementazione dei lavori descritti nella seconda sezione sono state riviste e rifatte dagli stessi creatori al fine di analizzare in maniera più approfondita fornire delle API pronte all'uso. Nello specifico Cerbara si è occupato della coordinazione basata su tuple, mentre Poli dell'architettura BDI degli agenti. Il cuore pulsante di entrambi i lavori risiede nell'uso intensivo di un interprete Prolog fatto ad hoc per Unity, *UnityProlog* [8].

4.1 PERCHÈ IL PROLOG PER LA COORDINAZIONE BASATA SU TUPLE?

Nel lavoro di Cerbara, *UnityProlog* è stato utilizzato come tecnologia abilitante per i modelli di coordinazione ed interazione implementati. Questi modelli sono basati su spazi di tuple, dove piccole porzioni di informazioni sono usate per fornire un supporto di coordinazione basato su informazioni, e sulle primitive *Linda* come le principali azioni che permettono di interagire con gli spazi di tuple aggiungendo, rimuovendo o leggendo tuple. Le primitive *Linda*, le tuple e gli spazi di tuple sono concetti già utilizzati in progetti come TuCSoN e LuCe, in cui questi concetti sono specificati con un modello di comunicazione logic-based e utilizzando la teoria della logica del primo ordine, quindi sfruttando le caratteristiche dell'interprete Prolog, come la definizione parziale di un template utilizzando variabili logiche, il backtracking, la sintassi dichiarativa e l'unificazione come meccanismo di matching.

Una volta ottenuto pieno accesso all'interprete Prolog, Cerbara si è occupato di progettare e sviluppare una libreria *Linda* in Prolog, pienamente accessibile dal meccanismo di scripting C# di Unity 3D, e di fornire meccanismi ad alto livello per l'interazione e coordinazione sfruttabili durante la definizione comportamentale di un agente.

4.2 PERCHÈ IL PROLOG PER L'ARCHITETTURA BDI DEGLI AGENTI?

Per progettare un'architettura BDI per agenti è necessario prima di tutto capire come questi funzionano. Il ciclo di reasoning di un agente è composto da quattro fasi principali:

1. Generazione di opzioni: l'agente restituisce un insieme di opzioni in base a cosa conosce riguardo al mondo e quali sono i suoi desideri;

2. Deliberazione: l'agente seleziona un sottoinsieme delle opzioni precedentemente selezionate;

3. Esecuzione: l'agente esegue, se è presente la relativa intenzione, un'azione tra le opzioni precedentemente selezionate;

4. Percezione: l'agente infine aggiorna la sua conoscenza del mondo (ed eventualmente se stesso).

Le analogie tra il ciclo di reasoning di un agente e il reasoning logico sono evidenti, e per questo motivo Poli ha deciso di usare il paradigma logico come cardine del suo sistema BDI.

L'obiettivo di Poli è quello di ottenere un nuovo layer di astrazione che permette al programmatore finale di dichiarare comportamenti ad alto livello in maniera semplice e senza il bisogno di sincronizzazione esplicita con l'event loop di Unity 3D. Il comportamento di un agente potrà essere così definito interamente in un file prolog, che avrà questa struttura:

```
/* Belief iniziali */
belief yourBelief.
...

/* Desire iniziali */
desire yourDesire.
...

/* Piani definiti */
add yourDesire && (preConditions) => [
   action,
   ...
].

...
```

Sfruttando questo design, Poli si è inoltre occupato di creare un'implementazione per gli artefatti, ottenendo un primo prototipo funzionante per il meta-modello A&A su Unity 3D. In maniera simile agli agenti, le caratteristiche di un artefatto sono descritte all'interno di un file Prolog, e comprendono belief e piani assimilabili da parte di un agente.

5 NUOVE FUNZIONALITÀ PER IL MODELLO BDI

Come si può notare, la struttura interna di un agente BDI progettato con le API di Poli è molto simile a quella di un agente progettato con *Jason* [7], riprendendone la stessa semantica ma con una sintassi leggermente diversa, a causa di alcuni operatori già riservati per l'interprete Prolog: ad esempio, in *Jason* è possibile definire l'aggiunta di un evento che innesca un achievement goal utilizzando la struttura "+!some(goal)", ma l'operatore "!" è già riservato per UnityProlog, e non è quindi possibile ridefinirlo, di conseguenza, la stessa semantica è stata ottenuta con la struttura "add some(goal)".

5.1 ESTENSIONE DELLE API DI POLI CON LA GOAL DELETION

Nel caso del fallimento di un piano, Poli sfrutta il backtracking che Prolog offre per selezionare un altro piano da eseguire. Ad esempio, definiamo un agente con due piani per accendere un fuoco. Il primo piano utilizza l'azione soffiaSulLegno, mentre il secondo usa l'azione creaScintilla. Quando il primo piano viene innescato, l'azione soffiaSulLegno fallisce nell'intento di accendere il fuoco, il piano fallisce, e tramite backtracking viene innescato il secondo piano, che riuscirà ad accendere il fuoco.

Il comportamento sopra descritto è lontano dall'assomigliare a quello di *Jason*. Indipendentemente dalla ragione per cui un piano fallisce, *Jason* genera un evento chiamato goal deletion - un evento della forma "-!goal" - qualora il piano per un goal achievement fallisca. L'idea di *Jason* è che un piano per la goal deletion venga utilizzato come un piano di "clean-up", eseguito prima di un eventuale "backtracking" (ovvero prima di tentare un altro piano per raggiungere l'obiettivo per cui un piano è fallito) [7]. Le azioni eseguite da un piano prima del suo fallimento infatti non vengono annullate con il backtracking di Prolog, ed è qua che la goal deletion risulta utile: dare la possibilità al programmatore di annullare le azioni intraprese da un piano prima del suo fallimento per poi provare un altro piano. L'idea quindi è che la goal deletion sostituisca il backtracking sfruttato da Poli, come illustrato nell'immagine seguente: se si vuole ritentare il piano fallito, il programmatore lo deve esplicitare nel piano per la goal deletion.

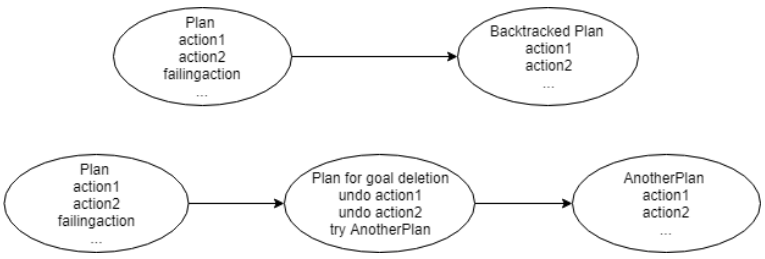

Figura 5.1: Comportamento iniziale e comportamento ricercato, a confronto.

Per rendere possibile lo scenario sopra descritto, si è reso necessario apportare delle modifiche al reasoner di Poli che verranno di seguito illustrate.

Prima di tutto, è stato modificato il predicato del_desire/1 e quando viene richiamato: nello scenario iniziale infatti viene utilizzato tutte le volte che un piano viene eseguito nella sua

interezza, mentre da ora in avanti viene chiamato automaticamente solo quando una delle azioni all'interno del piano fallisce. Sfruttando la Eremic Logic [8], si tiene traccia dell'obiettivo (desire) che si sta cercando di portare a termine, in modo tale che quando una azione fallisce, sono sempre a conoscenza dell'obiettivo per cui l'azione è fallita. Quando l'azione fallisce, chiamo del_desire/1 passandogli il desire in cui è avvenuto il fallimento. I predicati go_on/0 ed extract_task/1 sono stati quindi modificati come segue:

```
go_on :-
    desire D,
    add D && _ => _,
    findall(
        X,
        (add D && C => A, append([C],A,Temp), append(Temp,[retract(desire
            D)],X)),
        Plans
    ),
   assert(/current_desire/current:D),
    assert(/active_task/plans:Plans).
```

```
extract_task(A) :-
    (A = [use_artifact(Ref,Action)|N], !,
        use_artifact(Ref,Action,Ret),
        append(Ret,N,Res),
        set_active_task(Res))
    ;
    (A = [cr (@Ref,M)|N], !,
        set_active_task((@Ref,M,N)))
    ;
    (A = [cr M|N], !,
        set_active_task((M,N)))
    ;
   (A = [M|N],
        M,
        set_active_task((N))
      ;
      /current_desire/current:D,
      del_desire(D)
   ).
```

Per un piano chiamato example, si avrà quindi la seguente situazione:

```
desire example.

add example && true => [
actions...
].

del example && true => [
undo actions,
add_desire(example)
].
```

in cui il piano del example rappresenta il piano per la goal deletion, innescato quando una delle azioni nel piano add example fallisce.

5.2 Estensione delle API di Poli con la delegazione delle azioni interne

Una delle funzionalità presenti nel modello BDI di Poli, è la possibilità da parte degli agenti di imparare piani da altri agenti. Quando ciò avviene, può capitare tuttavia che un agente A richiedente un piano ad un agente B, non conosca le azioni interne (metodi C#) definite e implementate all'interno dello script dell'agente B, interrompendo l'esecuzione di tutto il MAS. Per impedire a questo scenario distruttivo di manifestarsi, ho progettato e implementato un meccanismo tramite il quale tutte le azioni interne di un piano appreso non presenti nello script dell'agente che le andrà ad eseguire, verranno richiamate sull'agente da cui il piano è stato acquisito, delegando a quest ultimo la loro esecuzione.

Il piano ricevuto viene iterato di tutte le sue istruzioni, e viene controllato che tutte le eventuali azioni interne presenti nel piano siano implementate nello script dell'agente ricevente. Nel caso in cui l'azione non sia implementata, essa viene contrassegnata in maniera tale che venga richiamata sull'agente mittente. Per verificare la presenza dell'azione all'interno di uno script, è stata utilizzata la reflection:

```
public bool IsMethodAvailable(object classToCheck, object methodToCheck)
{
    string c = classToCheck.ToString().Split('(')[1].Split(')')[0];
    string m = methodToCheck.ToString().Split('(')[0];

    Type type = Type.GetType(c);
    MethodInfo methodInfo = type.GetMethod(m, BindingFlags.IgnoreCase |
        BindingFlags.Public | BindingFlags.Instance);
    if (methodInfo == null)
    {
        return false;
    }
    return true;
}
```

La marcatura delle azioni da richiamare sull'agente mittente avviene grazie al predicato find_actions/3 da me implementato, e richiamato automaticamente tutte le volte che un agente vuole apprendere un piano da un altro agente:

```
find_actions(A, List, Result) :-
   (A = [act Action|T],
      (Action = (M,_) ; Action = M, M \= (_,_,_), M \= (_,_)),
      get_ref(Ref),
      ref_to_agent(Ref,Ag),
      call_method(Ref,'IsMethodAvailable'(Ag,M),Ret),
      (
         Ret = true,
         append(List, [act Action], Conc),
         find_actions(T,Conc, Result)
         ;
         Ret = false,
         ref_agent_to_learn(RefAgentToLearn),
         (
            Action = (M,R),
            append(List, [act (@RefAgentToLearn,M,R)], Conc)
            ;
            append(List, [act (@RefAgentToLearn,M)], Conc)
         ),
         find_actions(T,Conc, Result)
      )
   );
   (A = [cr Coroutine|T],
      Coroutine \= (_,_),
      get_ref(Ref),
      ref_to_agent(Ref,Ag),
      call_method(Ref,'IsMethodAvailable'(Ag,Coroutine),Ret),
      (
         (Ret = true,
         append(List, [cr Coroutine], Conc),
         find_actions(T,Conc, Result))
         ;
         (Ret = false,
         ref_agent_to_learn(RefAgentToLearn),
         append(List, [cr (@RefAgentToLearn,Coroutine)], Conc),
         find_actions(T,Conc, Result))
      )
   );
   (A = [Something|T],
      append(List, [Something], Conc),
      find_actions(T,Conc, Result))
   ;
   (A = [],
      Result = List,
      retract(ref_agent_to_learn(RefAgentToLearn))
   ).
```

La delegazione delle azioni così progettata risolve il problema iniziale che causava il blocco del MAS a causa di eccezioni lanciate a runtime, tuttavia permane un'importante limitazione, per ora irrisolta: se l'azione delegata modifica lo stato interno dell'agente, può dare esito a risultati inconsistenti. D'altra parte, se l'azione in questione non modifica lo stato interno, tutto il sistema funzionerà come previsto in fase di design del MAS.

6 Integrazione tra modello BDI e interazione basata su spazi di tuple

Per testare e prendere familiarità con le API descritte in sezione 5, si è deciso di implementare una versione naïve di uno dei casi di studio descritti in [9] utilizzando entrambi i lavori di Cerbara e Poli. Il caso di studio in esame riguarda la coordinazione di una squadra di soccorritori che rispondono alle richieste di soccorso.

Il MAS avrà due tipi di agenti: **Rescuer** e **Person**, con i relativi comportamenti così definiti:

```
desire work.

add work && true => [
   add_desire(rescue)
].

add rescue && in(warning(injured)) => [
   log("Rescuer(PROLOG SIDE): in(warning(injured)) on my KB successful, im
       going to rescue the injured person"),
   cr goTo($'ToRescue')
].
```

```
desire dostuff.

add dostuff && true => [
   cr dostuff,
   act reportinjury
].
```

L'agente Person esegue per un periodo random la coroutine dostuff implementata all'interno del suo script, al termine della quale rilascia una tupla attorno a se con l'azione reportinjury, di seguito mostrata:

```
public void ReportInjury()
{
    print("Person: I'm injured, creating region...");
    Region region = new Region("Zone", transform.position, new Vector3(35f,
        10f, 20f), PrimitiveType.Cube, true, Quaternion.identity);
    LindaCoordinationUtilities.SendMessageToRegion(this.warningTuple, region);
}
```

Quando ciò avviene, il "corpo" (gameobject) dell'agente Rescuer collide con la regione in cui è stata rilasciata la tupla, e viene eseguita una Linda out su di esso:

```
private void OnTriggerEnter(Collider coll)
{
    if (LindaCoordinationUtilities.RetrieveMessageFromSituatedObjectTriggered
   (this.warningTuple, coll).Equals(ReturnTypeKB.True))
    {
        print("Rescuer: Injured person detected, adding tuple to my kb");
        LindaLibrary.Linda_OUT(this.warningTuple, base.myKB);
    }
}
```

In questo modo, la precondition in(warning(injured)) del piano add rescue definito per l'agente Rescuer è soddisfatta, ed è possibile eseguire il piano.

Scena su Unity:

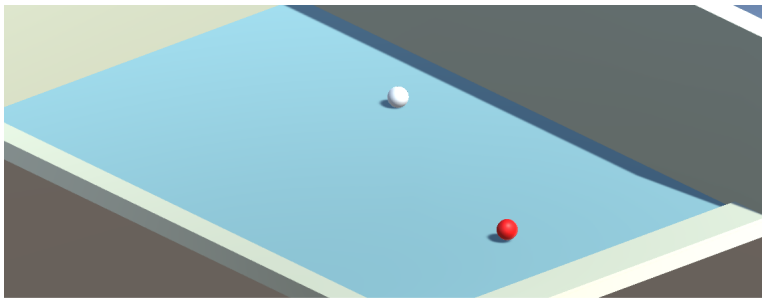

Figura 6.1: Situazione iniziale coi due agenti Person (sfera bianca) e Rescuer (sfera rossa).

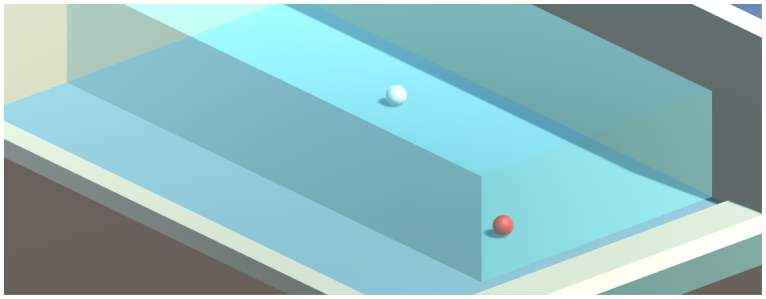

Figura 6.2: Person rilascia una tupla attorno a sè creando una regione.

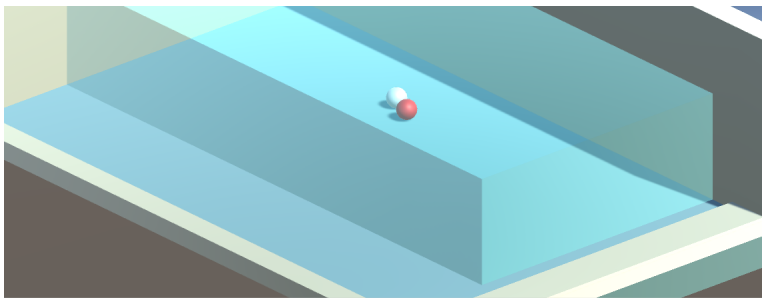

Figura 6.3: Rescuer percepisce la tupla grazie alla collisione con la regione e si muove verso Person.

7 Conclusioni e sviluppi futuri

I modelli ideati e implementati dai due pionieri Poli e Cerbara si sono dimostrati essere fruibili e facilmente espandibili con nuove funzionalità come dimostrato nelle sezioni precedenti. Le strade esplorabili ed eventualmente percorribili comunque non sono esaurite. Una di queste può essere quella di sfruttare le due API sviluppate e descritte fin qua per implementare il più fedelmente possibile il modello *Spatial Tuples* [9], una estensione del modello basato su tuple per la coordinazione degli agenti nei MAS, secondo il quale:

1. Le tuple sono localizzate in regioni del mondo fisico ed eventualmente si muovono insieme ad un dispositivo mobile;
2. Il comportamento delle primitive di coordinazione di Linda è esteso in modo tale da dipendere sulle proprietà spaziali degli agenti, delle tuple, e sulla topologia dello spazio;
3. Lo spazio di tuple può essere pensato come ad uno layer virtuale che aumenta la realtà fisica.

Spatial Tuples ha a che fare principalmente, come il nome suggerisce, con tuple spaziali, ovvero tuple associate ad una informazione spaziale. Le informazioni spaziali possono essere di qualunque tipo, come ad esempio coordinate GPS, posizionamento metrico (100 metri attorno alla mia posizione corrente), o amministrativo (via Zamboni 33, Bologna, Italia). In qualunque caso, associa la tupla a qualche regione nello spazio fisico. Le potenzialità di Unity unite agli sviluppi di Poli e Cerbara possono essere sfruttate in questa direzione, e sarà l'argomento cardine del lavoro che intendo portare avanti per la tesi.

RIFERIMENTI BIBLIOGRAFICI

[1] A. Omicini *Dispense lezioni Sistemi Autonomi.*, 2017.

[2] A. Omicini, A. Ricci, M. Viroli *"Artifacts in the A&A meta-model for multi-agent systems"*, 2008.

[3] S. Mariani, A. Omicini *"Game Engines to Model MAS: A Research Roadmap"*, 2016.

[4] M. Cerbara, N. Poli *"Game Engines and Multi-Agent Systems: a Marriage?"*, 2016.

[5] M. Cerbara, N. Poli *"Toward MAS in Game Engines: A Simple BDI Architecture in Unity3D"*, 2017.

[6] A. Ricci, M. Viroli, A. Omicini *"CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS"*, 2007.

[7] R. H. Bordini, J. F. Hübner, M. Wooldridge *"Programming multi-agent systems in AgentSpeak using Jason"*, 2007.

[8] I. Horswill *"UnityProlog"* , https://github.com/ianhorswill/UnityProlog, 2015.

[9] A. Ricci, M. Viroli, A. Omicini, S. Mariani, A. Croatti, D. Pianini *"Spatial Tuples: Augmenting Reality with Tuples"*, 2018.