

Spatial Tuples in Unity3D: a solution based on TuSoW and JaCaMo

Davide Giacomini

davide.giacomini2@studio.unibo.it

Andrea Placuzzi

andrea.placuzzi@studio.unibo.it

Giacomo Venturini

giacomo.venturini2@studio.unibo.it

19 settembre 2019

Questo progetto studia la possibilità di implementare il concetto di *tupla spaziale* combinando le funzionalità spaziali dell'engine Unity con quelle distribuite e logiche offerte da TuSoW. Dato che l'ambito applicativo si presta in modo naturale alla simulazione di scenari “real world”, verrà utilizzato il framework JaCaMo per la realizzazione di agenti BDI.

1 Requisiti

Il progetto nasce dall'idea dell'unire MAS e game engine con un middleware in modo da abilitare e semplificare la gestione di scenari di simulazione, serious gaming e realtà aumentata. L'obiettivo è quindi studiare la possibilità implementativa dei principali pattern di coordinazione citati nel paper “Spatial Tuples: Augmenting reality with tuples” [2] utilizzando:

- TuSoW come base di conoscenza web-based
- JaCaMo come framework per supporto ad agenti BDI
- Unity come motore grafico in grado di fornire strumenti adatti alla gestione spaziale

2 Analisi dei Requisiti

2.1 Tuple spaziali - primitive

Il paper[2] definisce una tupla spaziale come una tupla associata ad una certa regione o area. Per poter lavorare con questo concetto vengono definite le seguenti operazioni:

- **out(t @ r)** - inserimento della tupla t nella regione r
- **in(tt @ rt)** - recupera e rimuove dallo spazio di tuple rt una tupla che corrisponde al template tt . È bloccante, quindi non ritorna finché non esiste un template tt che fa match
- **rd(tt @ rt)** - uguale alla **in()** ma effettua una lettura non distruttiva
- **no(tt @ rt)** - chiamata *test-for-absence*, non ritorna finché non esiste più una tupla che matcha tt nella regione rt
- **inp(tt @ rt), rdp(tt @ rt), nop(tt @ rt)** - versioni *predicative* delle operazioni soprastanti, nel caso in cui nessun match viene trovato la chiamata non è bloccante e ritorna
- **rd_all(tt @ rt , L), in_all(tt @ rt , L), no_all(tt @ rt , L)** - versioni *bulk* delle operazioni soprastanti, ritornano tutte le tuple che fanno match col template

2.2 Pattern da implementare

Tra i pattern elencati nel paper, ci è stato richiesto di implementare quelli sotto le categorie:

- Situated knowledge sharing
- Spatial synchronization and mutual exclusion

dato che la loro implementazione copre le casistiche base necessarie all'implementazione di situazioni più complesse e articolate. Di seguito parleremo più in dettaglio delle loro specifiche.

2.2.1 Situated knowledge sharing

In questa categoria di pattern le tuple spaziali sono utilizzate per inviare indirettamente un messaggio a tutti gli agenti situati in una determinata regione. Il messaggio è rappresentato come una tupla associata ad una regione.

Find me!

In questo pattern un certo numero di dispositivi mobili deve raggiungere una regione utilizzata come luogo di meeting (*meetingPlace*). Un agente *meetingControl* controlla l'arrivo dell'aspettato numero di device nell'area assumendo che ognuno di essi effettui il deposito di una tupla con una **out(hereIam(id) @ here)**. L'agente *meetingControl* recupererà le tuple effettuando ripetutamente una **in(hereIam(Device) @ meetingPlace)** finchè tutti i dispositivi non avranno raggiunto l'area stabilita.

Breadcrumbs or Follow me!

In questo pattern un agente *hansel* lascia una traccia di tuple (*breadcrumbs*) spaziali mentre si sposta. Ognuna contiene informazioni riguardo all'ordine di rilascio, ad esempio incrementando un contatore. Le tuple rilasciate attraverso una **out(wasHere(hansel, C) @ here)** permettono così di individuare la traiettoria dell'agente osservando la loro distribuzione spaziale. C'è da notare che il paper non impone nessun vincolo implementativo sulla strategia con cui l'agente *follower* dovrà cercare o individuare la successiva *breadcrumb* da raggiungere.

2.2.2 Spatial synchronization and mutual exclusion

Spatial synchronization

Si parla di sincronizzazione spaziale quando un agente *A* effettua un task *T* non appena un altro agente ha finito di svolgere un altro task *T'* in una specifica regione *R*. Inoltre:

- l'agente *A* non ha necessità di sapere chi sta effettuando il task *T'* nella regione
- l'agente che sta effettuando il task *T'* non ha necessità di sapere chi è interessato alla terminazione e al risultato del task *T'*

Un possibile caso d'uso è il seguente: un team di agenti è incaricato di consegnare dei pacchi in una posizione con una determinata area. Un agente effettua la consegna collocando una tupla spaziale nella regione come segue: **out(delivered(package('code - 1002','toys')) @ address)**. Un altro team di agenti, situato nell'area, è incaricato di reperire i pacchi appena sono disponibili nella loro vicinanza effettuando una **in(delivered(package(Code, Desc)) @ my_assigned_pickup_region)**.

La sincronizzazione può avvenire considerando le posizioni degli agenti come segue: una volta effettuata la consegna, un agente potrà lasciare l'area solo quando un altro agente raggiunge il punto di meeting, effettuando un'attesa attraverso una **rd(hereIam(_) @ deliveringPlace)**.

Mutual exclusion

Situazione in cui si vuole permettere ad un solo agente di entrare in una determinata regione. Una tupla spaziale può essere utilizzata come una sorta di lock, richiedendo ad ogni agente che vuole entrare nella regione di ottenerla e rilasciarla non appena ne esce, effettuando rispettivamente una **in(lock @ mutex_region)** e una **out(lock @ mutex_region)**.

2.3 TuSoW

TuSoW è uno strumento in grado di gestire spazi di tuple rendendole disponibili as-a-service attraverso un'interfaccia REST. In questo modo permette di sfruttare la programmazione logica senza la necessità di avere un suo supporto diretto all'interno dei linguaggi di sviluppo utilizzati. A questo vantaggio si aggiunge la possibilità di poter condividere le tuple inserite al suo interno, sfruttando il servizio come base di conoscenza distribuita.

2.4 JaCaMo

JaCaMo è un framework per la programmazione di sistemi multi-agente. È la combinazione di 3 piattaforme:

- Jason: programmazione di agenti autonomi
- Cartago: programmazione di artefatti nell'ambiente
- Moise: organizzazione tra agenti

È stato scelto perché offre un buon livello di astrazione che permette di creare agenti BDI e produrre rappresentazioni di oggetti nell'ambiente con il minimo sforzo.

2.5 Synapsis

Synapsis è un middleware che abilita la comunicazione tra agenti *JaCaMo* e *GameObject Unity*, instaurando una relazione mente-corpo. È venuto quindi naturale utilizzarlo dato che questo tipo di interazione è necessaria per la realizzazione di questo del progetto.

2.6 Unity

Essendo il progetto focalizzato principalmente sull'aspetto spaziale è venuto spontaneo considerare l'utilizzo di un *game engine* proprio per le feature focalizzate su questo aspetto (es. intersezione, gestione collisioni). La scelta su quale *game engine* da utilizzare è ricaduta su *Unity*, per i seguenti motivi:

- offre tutte le funzionalità richieste per la gestione dello spazio
- è cross-platform
- **Synapsis**: middleware che tramite websocket abilita la comunicazione tra il framework *JaCaMo* e *Unity*
- **Wrld3D**: libreria per posizionare geograficamente le entità all'interno della scena

Si deve notare come **Synapsis** sia incentrato solamente sulla comunicazione e non abbia alcun riferimento a informazioni spaziali. Sarà quindi necessario estenderlo per aggiungere feature improntate sulla gestione di operazioni di carattere spaziale (sottosezione 3.3).

3 Design

L'architettura del nostro sistema è fortemente influenzata dagli strumenti imposti dai requisiti e quelli disponibili per semplificare l'interazione JaCaMo - Unity. In Figura 1 viene mostrato come:

- attraverso il middleware Synapsis:
 - un agente JaCamo invia le azioni da svolgere al rispettivo GameObject Unity
 - un GameObject Unity comunica gli eventi spaziali (es. entrato in una regione, uscito da regione, crea regione) all'agente JaCaMo
- attraverso chiamate REST gli agenti JaCaMo effettuano le operazioni **out(...)**, **in(...)** e **rd(...)** aggiornando e recuperando le informazioni memorizzate nella base di conoscenza di TuSoW

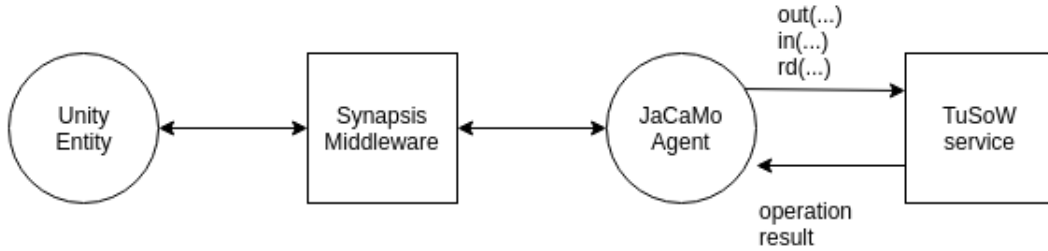


Figura 1: Diagramma che mostra l'interazione tra i diversi elementi del sistema. Per ogni agente JaCaMo è presente un GameObject lato Unity. La gestione della comunicazione tra questi sistemi è demandata al middleware Synapsis.

3.1 Modello tupla spaziale

La struttura scelta per modellare una tupla spaziale è la seguente:

$$tuple(t, region(regionId)).$$

Il funtore *tuple* rappresenta una tupla spaziale in cui *t* è il contenuto della tupla, *region* è il funtore che rappresenta la spazialità della tupla e *regionId* è l'identificativo della regione a cui appartiene la tupla. L'identificatore è generato da *Unity* come risultato dell'azione di creazione di una nuova regione, per maggiori dettagli vedere la sottosezione 4.3.

Inoltre, come si può vedere non viene specificata l'estensione della regione in quanto né lato JaCaMo né lato TuSoW devono essere effettuate computazioni spaziali. L'adozione di questo modello presenta in aggiunta il vantaggio di supportare la *out(t @*

me), dato che non specifica il centro della regione che in questo caso cambia al variare della posizione dell'agente che l'ha effettuata. Infatti la presenza di questa informazione avrebbe implicato il suo aggiornamento ad ogni spostamento dell'agente, causando possibili inconsistenze dovute al fatto che attualmente in TuSoW non è possibile aggiornare il contenuto delle tuple in modo atomico.

3.2 Comunicazione JaCaMo - TuSoW

In questa sezione introdurremo come è stata progettata la comunicazione tra JaCaMo e TuSoW, al fine di consentire ad agenti Jason di effettuare le operazioni primitive definite in sottosezione 2.1. Si è quindi previsto un agente che definisce tutti i piani necessari per eseguire le diverse varianti delle operazioni *in*, *out*, *rd* e che si appoggia ad un proprio artefatto per eseguire tutte le chiamate di rete necessarie per svolgerle.

Per ogni agente che dovrà comunicare con TuSoW sarà quindi sufficiente includere l'agente precedentemente descritto ed utilizzare i piani che mette a disposizione.

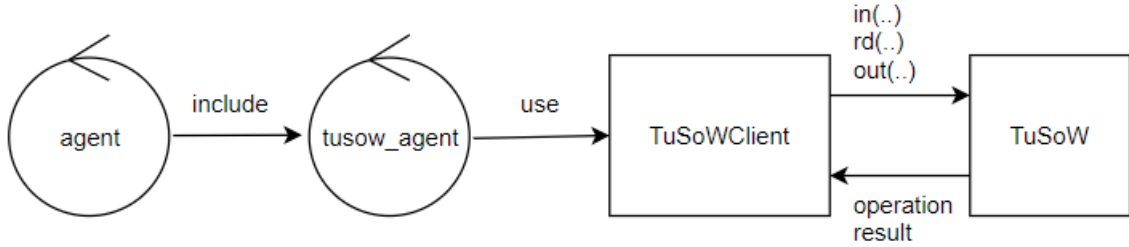


Figura 2: Un *agent* che deve comunicare con TuSoW include *tusow_agent* che incapsula i piani per effettuare le operazioni primitive di Linda. Le azioni vengono quindi attuate attraverso l'artefatto *TuSoWClient*.

3.3 Estensione Unity: Spatial Agent Body

In questa sezione introdurremo la classe *SpatialAgentBody*, creata per estendere le funzionalità del middleware **Synopsis** con altre più orientate alla gestione spaziale. Per utilizzarla sarà sufficiente che lo script collegato al GameObject che rappresenta il corpo dell'agente la estenda.

L'obiettivo principale è quello di permettere al corpo di un agente (rappresentato lato Unity dalla classe *SynopsisBody*) di eseguire nuove azioni per la creazione di entità nello spazio (regioni) e di gestire eventi inerenti all'interazione con esse.

Dopo un'analisi di tutti i pattern oggetto di questo progetto, sono state individuate le seguenti operazioni:

- **CreateRegion**: creazione di una nuova regione, è possibile specificarne la forma e l'estensione

- **GetCurrentRegions:** metodo per ottenere tutte le regioni all'interno delle quali si trova l'agente
- **GetPosition:** restituisce informazioni riguardo alla posizione corrente dell'agente (in termini di coordinate geografiche)
- **GoToPosition:** segnala all'agente di muoversi verso una specificata posizione (definita tramite coordinate geografiche)
- **GoToRegion:** versione estesa dell'operazione precedente, che segnala all'agente di muoversi verso una regione con un nome specifico. Si può inoltre scegliere se avvicinarsi solamente alla regione o entrarci dentro completamente
- **LeaveRegions:** operazione opposta rispetto alla precedente, forza l'agente a uscire dalle regioni specificate (o da tutte se non viene passato nessun nome)
- **Move:** operazione utilizzata per segnalare all'agente di muoversi per una certa distanza spaziale. È possibile specificare anche un angolo di sterzata

Per poter eseguire correttamente le operazioni sopra citate, sarà inoltre necessario intercettare tutti gli eventi di collisione del *body*. Grazie al supporto diretto del *game engine* è quindi bastato intercettare alcuni eventi scatenati da *Unity*. Di seguito riportati quelli gestiti, quando l'entità con cui si collide è una *region*:

- *OnTriggerEnter*: scatenato quando il collider dell'entità inizia a rilevare una collisione con un'altra entità
- *OnTriggerStay*: lanciato fintanto che l'entità collide con un'altra
- *OnTriggerExit*: scatenato quando l'entità smette di collidere con un'altra, ovvero quando non c'è più intersezione tra le aree dei due collider interessati

Per quanto riguarda gli eventi *OnTriggerEnter* e *OnTriggerExit*:

- nel caso in cui non ci sia in esecuzione alcuna azione, sono gestiti inviando alla *mind* un messaggio
- se l'agente sta compiendo un'azione invece, viene controllato se la condizione per interromperla è soddisfatta e, in tal caso, viene interrotta

L'evento *OnTriggerStay* viene utilizzato per fare in modo che l'agente possa verificare quando si trova completamente dentro la regione in cui collide (molto utile per gestire l'azione *GoToRegion*).

La gestione di questi trigger lato Unity non dovrebbe risultare pesante dal punto di vista computazionale anche a fronte di una mole elevata di eventi, come ad esempio in casi in cui venga simulato uno spazio fisico reale con molti elementi. Infatti, nonostante

non sia possibile dare una risposta precisa se non tramite test approfonditi, è ragionevole pensare che *Unity*, essendo un Game Engine, sia stato progettato per gestirne una quantità sicuramente elevata¹.

Concludiamo infine specificando come si intende riutilizzare il più possibile gli oggetti messi a disposizione da *Synapsis*, cercando quindi semplicemente di aggiungere le nuove funzionalità. In Figura 4 vengono rappresentate con un semplice UML le relazioni tra gli artefatti preesistenti e quelli nuovi.

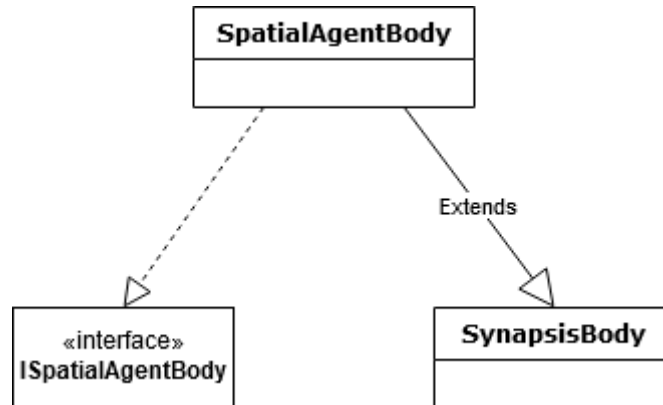


Figura 3: Diagramma delle classi per mostrare le relazioni tra le nuove e vecchie classi.

Si noti come l'interfaccia *ISpatialAgentBody* definisca tutte le operazioni illustrate qui sopra, la cui implementazione è appunto fatta all'interno della nuova classe *SpatialAgentBody*, che, estendendo *SynapsisBody*, e quindi *MonoBehaviour*, ha la possibilità di intercettare gli eventi sopra citati.

4 Dettagli implementativi

4.1 Estensione TuSoW

Al momento TuSoW consente di eseguire ricerche (*rd* e *in*) specificando un singolo template. Nel nostro caso questo comportamento è limitante in quanto può essere necessario cercare una tupla in più regioni (es. sovrapposte). Per supportare tale comportamento si è esteso TuSoW in modo tale da consentire di specificare più template durante una ricerca.

4.1.1 Read multi-template

L'implementazione della read multi-template prevede semplicemente di eseguire una read su *Linda* per ogni template con la medesima configurazione richiesta. I risultati provenienti dalle singole read sono quindi combinati mediante *CompositeFuture* come segue:

¹<https://answers.unity.com/questions/504353/how-many-triggers-is-too-many-do-they-affect-perfo.html>

- si utilizza una *CompositeFuture.all* nel caso si debba realizzare una read *bulk*
- si utilizza una *CompositeFuture.any* nel caso si debba restituire la prima tupla che fa match con uno dei template

4.1.2 In multi-template

L'estensione della *in* è più complessa rispetto a quella della *read* in quanto la primitiva genera side-effect. Non è possibile adottare la stessa strategia della *read* perchè nel caso di *in* non *bulk* le *future* che non hanno sbloccato la *CompositeFuture.any* rimangono attive e quindi possono eliminare delle tuple senza che poi vengano restituite. Per questo motivo in caso di *in* non *bulk* si esegue prima una *read* multi-template con medesime opzioni, quindi, quando si ottiene il template che ha fatto match, si esegue una *in* predicativa solo su quel template. Nel caso la *in* prelevi una tupla viene restituito il risultato, viceversa si ripete la *read* multi-template.

4.1.3 Drawbacks

I drawbacks sono:

- quando si utilizza una *CompositeFuture.any* e viene risolta, tutte le *future* non ancora risolte rimangono “aperte”, potenzialmente arrivando ad un numero elevato
- la *in* multi-template non è atomica

Per questi motivi, una possibile e probabilmente migliore soluzione implementativa delle operazioni multi-template sarebbe quella di aggiungerle come primitive native in Linda. In questo modo si evita così di ricorrere allo stratagemma delle *CompositeFuture*.

4.2 Comunicazione JaCaMo - TuSoW

Per interagire con TuSoW è stato necessario implementare un web client. Dato che in java sono disponibili diverse soluzioni abbiamo valutato la scelta secondo i seguenti criteri:

1. possibilità di effettuare una chiamata GET con body
2. semplicità implementativa
3. evitare se possibile l'utilizzo di librerie esterne

Il primo punto è stato quello che ha determinato l'esclusione delle librerie *Unirest* e *OkHttp*, dato che in HTTP/2 una GET con body è considerata non standard e non era quindi permessa.

Abbiamo anche escluso l'implementazione utilizzando il WebClient Vert.x dato che, nonostante soddisfacesse il primo punto, avrebbe introdotto grosse dipendenze dalla

struttura del framework. Ad esempio ogni agente avrebbe avuto la necessità di creare un'istanza *Vert.x*, implicando una maggiore lentezza nel setup.

In definitiva abbiamo optato per il Web Client introdotto in Java 9 e perfezionato in Java 11, che presenta le seguenti caratteristiche:

- soluzione completamente nativa, che non introduce dipendenze a librerie/framework di terze parti, che evita così di dover sottostare a paradigmi specifici
- supporto sia per HTTP/2 che per Web Socket, nel caso in futuro TuSoW permetta il loro utilizzo
- supporto per chiamate asincrone via *CompletableFuture*

Come descritto nella sottosezione 3.2 si è implementato il *tusow_agent* che incapsula i piani per le varie operazioni. Sono stati definiti solo i piani per effettuare quelle non predicative in quanto gli unici a noi necessari. Gli altri saranno implementabili in modo analogo. I piani espongono la possibilità di definire 0, 1 o 2 variabili che verranno unificate, una volta ricevuta risposta da TuSoW, con i valori delle variabili definite nel template della *read* o della *in*.

Per rispettare la semantica bloccante delle primitive Linda implementate, il metodo dell'artefatto (e quindi del rispettivo piano dell'agente) che esegue le richieste a TuSoW viene sospeso, attraverso l'*await* di Cartago. La sua esecuzione viene ripresa quando viene soddisfatta una guardia, ovvero alla ricezione della risposta da TuSoW. Inoltre i piani che eseguono le richieste a TuSoW sono stati definiti come *atomici*, impedendo così l'esecuzione di altri piani alla sospensione di quello in esecuzione. Nel nostro caso questo ha semplificato l'implementazione dei vari pattern dato che garantisce che i *belief* dell'agente non siano stati modificati, evitando così ulteriori controlli aggiuntivi. Siamo consapevoli del fatto che in altri contesti applicativi tale comportamento potrebbe essere limitante: in tal caso sarà sufficiente aggiungere la rispettiva implementazione non atomica dei piani.

Dato il diverso formato di rappresentazione delle tuple su Jason e TuSoW è stato necessario implementare un parser per convertire le tuple dal modello Jason al modello di TuSoW prima di potergli fare richieste.

4.3 Creazione regione

Come visto nella sottosezione 3.3, la creazione di una regione avviene in seguito alla ricezione dell'azione *CreateRegion*. Una fase cruciale di questa operazione è la generazione del rispettivo id, utile per individuarla univocamente sia all'interno che all'esterno di Unity. L'identificatore, chiamato nel nostro caso *regionId*, è ottenuto concatenando i seguenti valori separati dal carattere “-”:

1. prefisso statico “**region**”
2. nome della regione, dato come parametro alla richiesta di creazione

3. il forma della regione (*square* o *sphere*)
4. l'estensione della regione

Ad esempio nel caso si voglia creare una regione “university”, di forma quadrata e di estensione 70, il rispettivo identificatore generato sarà *region-university-square-70*.

C'è da sottolineare che in questa implementazione la posizione del centro della regione non è considerata per la generazione del *regionId*, dato che non possono esistere due regioni con medesimo id e centri diversi. Ovviamente nel caso di centro uguale non verrà creata una nuova regione in quanto già esistente.

4.4 Pattern Follow Me!

Nella sezione 2.2.1, la descrizione del pattern traspota dal paper Spatial Tuples non impone alcun vincolo su come deve essere implementato l'agente follower. Nella tesi di Bagnoli[1] si assume che una tupla venga posizionata in un intorno di 70 metri dalla sua posizione e che l'agente rimanga in attesa con una **in(wasHere(hansel, C) at region(me, square(70)))**.

Questo approccio non è tuttavia applicabile nel nostro caso, dato che lato TuSoW non è presente il concetto di spazialità. Infatti, anche con una *in* multi-template, ovvero un template per ogni regione nel raggio di visione, eventuali tuple inserite successivamente in nuove regioni non saranno considerate in quanto il match sulle regioni è puramente basato sul loro nome.

Abbiamo risolto quindi il problema facendo effettuare una **rd(wasHere(hansel, C), R)** all'agente follower in modo tale da recuperare la regione in cui è situata la tupla successiva da raggiungere. Da notare che *C* viene precedentemente unificato con il valore del contatore desiderato.

4.5 Spatial Agent Body

Per quanto riguarda la classe *SpatialAgentBody*, l'implementazione è stata relativamente veloce e lineare. Questo grazie soprattutto alla presenza della classe *SynapsisBody*, che offre già tutta la logica di gestione della comunicazione.

È stato quindi sufficiente creare l'interfaccia *ISpatialAgentBody*, implementarla nella classe *SpatialAgentBody* ed eseguire l'override degli handler per i 3 eventi di trigger di collisione (*OnTriggerEnter*, *OnTriggerStay*, *OnTriggerExit*).

Degna di nota è la gestione della azione corrente modellata creando una classe specifica *ActionInfo*, che viene utilizzata per memorizzare:

- l'identificatore dell'azione
- una *IStopCondition*

- una *Action*² da eseguire dopo aver terminato l'azione. L'handler mette a disposizione due parametri, il collider che ha scatenato l'evento e una enumerazione per indicare quale evento è stato scatenato.

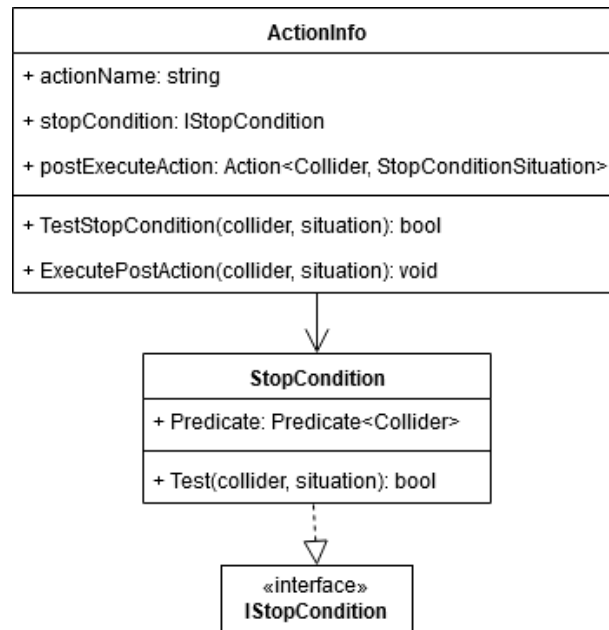


Figura 4: Diagrammi delle classi inerente alla struttura delle action info

Questo tipo di struttura permette di generalizzare, rendere agile ed incapsulare meglio la gestione dell'esecuzione di una azione. Infatti in un singolo oggetto si trova sia la funzione per rilevare quando interrompere l'azione sia l'eventuale azione da eseguire immediatamente dopo. La logica viene quindi definita assegnando i campi all'interno di questo oggetto, una volta ricevuto quale azione eseguire.

Altro vantaggio è che la gestione degli eventi del *game engine* risulta anch'essa semplificata proprio perché si rende solo necessario richiamare il metodo dell'oggetto in questione. Viene inoltre usato per determinare se interrompere l'azione corrente ed invocare la *PostExecuteAction*, qualora fosse disponibile.

4.6 Setup scene Unity

Per poter eseguire i vari pattern è stato necessario creare delle scene Unity per ogni scenario. Ciascuna scena quindi presenta un certo numero di *GameObject*, ai quali è stato aggiunto uno script che estende la classe *SpatialAgentBody*, per fare in modo che possa essere controllato da un agente *JaCaMo*. Ove necessario sono anche state posizionate delle regioni. Si specifica che tutte le scene sfruttano *Wrld3D* e che le entità sono state posizionate dichiarandone latitudine e longitude, sfruttando il componente *GeographicTransform*.

²<https://docs.microsoft.com/en-us/dotnet/api/system.action-1>

5 Istruzioni di deployment

I seguenti strumenti sono necessari per effettuare il building del progetto:

- Java 12
- sbt
- Unity 2019.2

Il progetto è scaricabile al seguente [repo](#). Il suo funzionamento dipende da [TuSoW](#) (branch feature/multiple-template) e da [Synapsis](#) e sarà quindi necessario scaricarli dai rispettivi repository. Per velocizzare il loro download è stato predisposto uno script batch *setup.bat* che si occupa della loro corretta predisposizione all'interno del directory tree del progetto.

Per testare i pattern di sincronizzazione implementati è necessario:

- avviare e aspettare il corretto setup di TuSoW e del middleware. Utilizzare lo script batch *launch-services.bat* per velocizzare l'operazione.
- selezionare e avviare la scena unity corrispondente al pattern che si vuole testare
- avviare il task gradle relativo al pattern da testare
 - `gradlew.bat findMe`
 - `gradlew.bat followMe`
 - `gradlew.bat spatialSynchronization`
 - `gradlew.bat mutex`

Anche in questo caso sono stati predisposti degli script batch per velocizzare l'operazione.

6 Esempi di utilizzo

Per evitare di dover compilare e avviare ogni volta gli esempi abbiamo caricato nella sezione [download](#) del repository dei video che mostrano il funzionamento dei pattern.

7 Conclusioni

Avendo implementato i pattern fondamentali di sincronizzazione spaziale abbiamo dimostrato la possibilità di modellare il concetto di tupla spaziale attraverso Unity, JaCaMo e TuSoW.

7.1 Sviluppi futuri

In questa sezione parleremo di possibili miglioramenti degli strumenti a nostra disposizione utilizzati per la realizzazione dei pattern spaziali.

7.1.1 TuSoW - modellazione di tupla spaziale

Come evidenziato nella sottosezione 4.4, qualora un agente sia in attesa di una tupla con template TT in una regione R ed essa venga rilasciata in una regione R' contenuta o parzialmente sovrapposta ad R , la tupla sarà ignorata in quanto il nome di R è diverso da quello di R' . Questa limitazione è dovuta al fatto che in TuSoW non è modellato il concetto di tupla spaziale. Una possibile soluzione potrebbe essere quella di utilizzare una sintassi simile a quella dei subgoal Prolog per effettuare il match di tuple spaziali anche con quelle presenti in regioni contenute o parzialmente sovrapposte a quella specificata.

7.1.2 TuSoW - supporto per web socket o mqtt

Aspetto fondamentale per l'evoluzione futura del sistema è l'aggiunta a TuSoW del supporto per web socket o mqtt dato che gli agenti potrebbero avere bisogno di interagire ripetutamente con il servizio per inserire e recuperare le tuple al suo interno. Questa aggiunta inoltre permette di avere interazioni più veloci, utile per gestire situazioni time critical.

7.1.3 TuSoW - comunicazione diretta con Unity

Ci sono situazioni in cui si potrebbe pensare di fare comunicare direttamente TuSoW con l'engine Unity. Questo approccio ad esempio risulterebbe utile per la ricerca di una tupla in più regioni collegate (con operatori di intersezione o contenimento), in modo da specificare a TuSoW solo la regione principale e delegargli il compito di recuperare quelle correlate richiedendole a Unity.

L'implementazione di questa comunicazione garantisce un ulteriore vantaggio: infatti permette all'engine di poter aggiornare una richiesta TuSoW con nuove relazioni tra le regioni. Una possibilità implementativa potrebbe essere quella di mantenere lato Unity memoria delle query in corso e modificarle all'aggiunta o alla rimozione di regioni.

A fronte di queste considerazioni, l'applicazione di questo approccio potrebbe essere utilizzata ad esempio per risolvere la problematica emersa nella sottosezione 4.4.

7.1.4 Unity - configurazione mappa WRLD

A inizio progetto c'è stato suggerito di utilizzare le mappe fornite da WRLD per semplificare la fase di setup. Tuttavia dopo averci preso confidenza possiamo dire che posizionare i GameObject al suo interno (e quindi configurare le scene a proprio piacere) sicuramente non è semplice, dato che il rendering delle mappe avviene solo quando l'engine è in modalità "running".

7.2 Synapsis - middleware

Al momento nel middleware e nelle rispettive librerie JaCaMo e Unity sono presenti funzionalità non generali utilizzate dall'esempio "recycling robots". Ripulendolo da questi

aspetti renderebbe sicuramente più semplice ed immediato il suo utilizzo, permettendo di individuare più velocemente i metodi di interesse per gli aspetti comunicativi e di gestione spaziale.

Un altro aspetto da considerare è il fatto che il suo startup è molto lento, dato che usa sbt. Sarebbe quindi comodo avere:

- sbt-extras, che come il gradle wrapper evita di dover installare il building tool sul proprio sistema operativo.
- il jar direttamente avviabile che evitan il ciclo check di dipendenze, build, execute.

Un aspetto omesso nella configurazione del middleware è la possibilità di decidere se applicare il timeout (e nel caso definire la sua durata) che disconnette “mind” e “body” quando non comunicano per un certo lasso di tempo. Infatti al momento questa proprietà non è configurabile e l’intervallo è troppo breve per alcune casistiche e ne impedisce il corretto funzionamento dato che alla riconnessione gli agenti JaCaMo vengono reinizializzati. Ad esempio nel pattern “mutual exclusion”, la disconnessione avviene mentre gli agenti sono in attesa del rilascio della mutex.

7.3 Synapsis - libreria JaCaMo

Per la libreria lato JaCaMo abbiamo riscontrato i seguenti problemi:

- La libreria generata non dichiara un manifest da cui scaricare le dipendenze, quindi una volta caricata non funziona senza importarle manualmente.
- Il piano *!createSynapsisMind*. non sembra funzionare correttamente, quello con argomenti *!createSynapsisMind([])*. invece funziona senza problemi.
- Gli esempi JaCaMo forniti non importano la libreria per comunicare col middleware Synapsis come jar. Sugeriamo pertanto di effettuarne l’import dato che potrebbero emergere altri problemi oltre a quelli sopracitati non presenti utilizzando direttamente una copia dei sorgenti.

7.4 Synapsis - libreria Unity

- l’oggetto *SynapsisBodyInfo* è dichiarato *private*, ma è necessario anche alle classi che estendono *SynapsisBody*, ad esempio nell’invio di una percezione tramite il metodo *TransmitPerception*.
- Nell’invio di una percezione il parametro *sender* sembra essere ignorato.
- I metodi presenti per l’invio delle percezioni e le costanti che gestiscono le azioni non sono generali.
- La libreria è modificabile a piacere dall’IDE, non sappiamo se questa cosa è voluta o meno.

- Sarebbe comodo avere una implementazione di default per i metodi *CounterpartEntityReady* e *CounterpartEntityUnready*, dato che non è detto che debbano essere gestiti.

Riferimenti bibliografici

- [1] Alessandro Bagnoli. Game engines e mas: Spatial tuples in unity3d, 2018.
- [2] Giovanni Ciatto, Lorenzo Rizzato, Andrea Omicini, and Stefano Mariani. Tusow: Tuple spaces for edge computing. 2018.
- [3] Alessandro Ricci, Mirko Viroli, Andrea Omicini, Stefano Mariani, Angelo Croatti, and Danilo Pianini. Spatial tuples: Augmenting reality with tuples. *Expert Systems*, 35(5), October 2018. Special Issue: New trends and innovations in intelligent distributed computing.
- [4] Lorenzo Rizzato. Coordination as a web service: una moderna implementazione del modello linda, 2018.