Progetto di Sistemi Intelligenti Distribuiti L-S
2005 – 2006

AGENTI CHE GIOCANO A CALCIO A 5

USO DEL SOFTWARE REPAST –PY COME AMBIENTE DI SVILUPPO

REALIZZATO DA:

Filippo Villani

INTRODUZIONE

Il mio progetto sui sistemi ad agenti vorrebbe riprodurre una situazione di gioco, anche perché si è visto come il gioco e la sua simulazione si presti in modo molto favorevole ad essere sviluppato tramite sistemi ad agenti.
Lo scopo finale è quello di riprodurre 10 agenti che giocano a calcio a 5 in un modo "intelligente".
Nella mia idea vorrei fossero presenti oltre ai 10 agenti Giocatori anche un agente Pallone e forse un arbitro per imporre anche delle regole e quindi, se infrante, delle punizioni.
Inoltre bisogna pensare ad un ambiente dove i 10 Giocatori sono suddivisi 2 in squadre, come normalmente avviene, e quindi anche introdurre comportamenti di tipo Cooperativo per agenti della stessa squadra e di tipo Rivalità tra agenti di squadre opposte.
L'Agent Tool Kit consigliatomi dal professore, con il quale realizzare il mio progetto è stato il Repast.
Il mio lavoro è poi stato il risultato di un continuo feedback con il professore ed insieme siamo convenuti che prima di tutto era più conveniente ai fini della crescita del progetto dedicare una parte all'introduzione ed alla descrizione del tool Repast (ed in particolare Repast Py) e poi porre le prime basi per un progetto.
Quindi dopo una parte descrittiva del tool per la simulazione ad agenti, abbiamo deciso di improntare un progetto più semplificato che permettesse di porre le basi per uno sviluppo poi futuro.
Di conseguenza ho simulato con Repast Py 2 soli giocatori che corrono (e questa deve essere una azione) verso la palla e una volta arrivati la calciano (2a azione) verso la porta.

Il tool: Repast

Repast è uno dei maggiori toolkit per la modellazione di sistemi ad agenti disponibili all'oggi
Vi sono 3 concrete implementazioni di queste specifiche, ovviamente ognuna di esse manterrà lo stesso Core Services costituito dal sistema Repast.
Queste implementazioni differiscono in base alla piattaforma sottostante di utilizzo e dai linguaggi su come sviluppare i modelli.
Le tre implementazioni sono:

Repast per Java: Repast J

Repast per Microsoft.Net: Repast.Net

Repast per gli script di Python: Repast Py

In generale viene consigliato di scrivere il modello base in Python usando appunto Repast Py data la sua Visual Interface molto comoda per improntare il lavoro; solo per i modelli più avanzati si consiglia di usare linguaggi di scrittura come Java o C# e quindi Repast J o Repast.Net.
Come tool repast offre completa flessibilità nello specificare proprietà e comportamenti degli agenti.
Il mio progetto si è quindi basato esclusivamente sull'uso di Repast Py; perché quello che volevo fare era capire questo tool ed improntare un lavoro di simulazione grafica, che proprio grazie alla Grid in 2-D è veramente immediata, in modo tale che poi questo lavoro possa essere preso d'esempio per sviluppi futuri.

Repast Py

Repast Py è un applicazione di ambiente di sviluppo per simulazioni Repast.
Le intenzioni di questo tool sono quelle di permettere ad un user, che si avvicina le prime volte a questi tipologie di sistemi, di creare una simulazione, mediante gradevole interfaccia grafica, come il costrutto, l'insieme di singoli componenti.
Per questi componenti è possibile poi esprimerne i comportamenti, ed in generale i comportamenti della simulazione, mediante l'uso di uno speciale subset di linguaggio Python.

Ecco come si presenta il menu di presentazione:

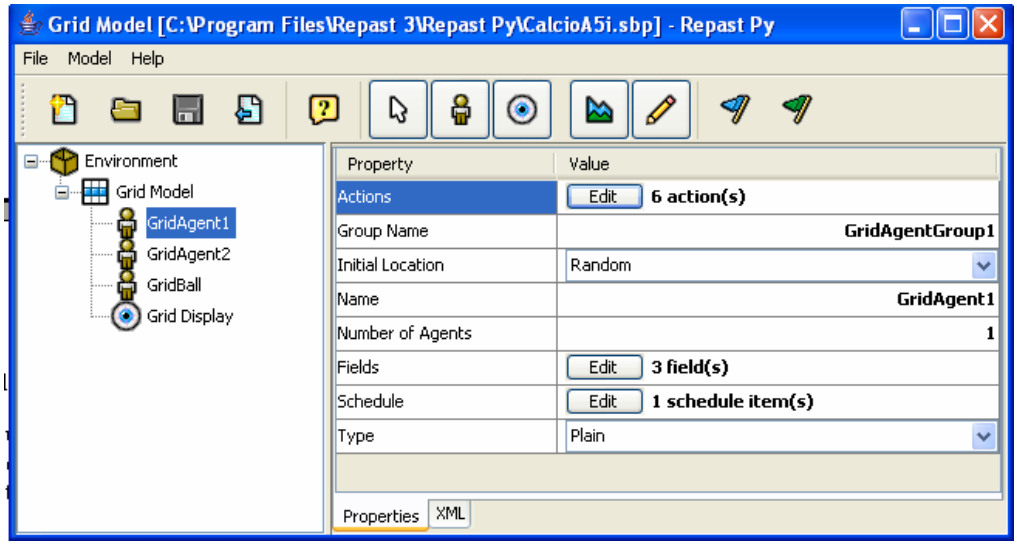
Una simulazione di Repast Py è organizzata all'interno di progetti, ogni progetto è poi formato da componenti scelti dalla *Barra dei Tool* e mostrati poi nel *Pannello di Progetto*.
Ogni componente, scelto dalla Tool Bar, ha delle caratteristiche che vengono mostrate nel *Pannello delle Proprietà* attraverso il quale l'utente può modificarle per modellare la propria simulazione.
Repast Py compila poi le componenti del Pannello di Progetto in codice Java compatibile con il framework di simulazione Repast.
Per eseguire la simulazione del modello Repast si deve compilarlo, mediante il pulsante  ; in caso di errori, Repast ne fornisce una descrizione completa e di grande aiuto.
La directory di uscita è specificata nel Pannello delle Proprietà del progetto sotto la voce: *Compiled Model Output Directory*. All'interno di essa vengono salvato i file di progetto.
Una volta compilato il progetto è possibile eseguirlo mediante il comando *run* dal menu *Model* oppure clickando il run botton  .

Ogni progetto che si decide svolgere mediante Repast va incontro all'uso ed allo studio di tre aspetti fondamentali che possiamo riscontrare, nel Pannello delle Proprietà di ogni singolo componente di Progetto:

- Action

- Fields

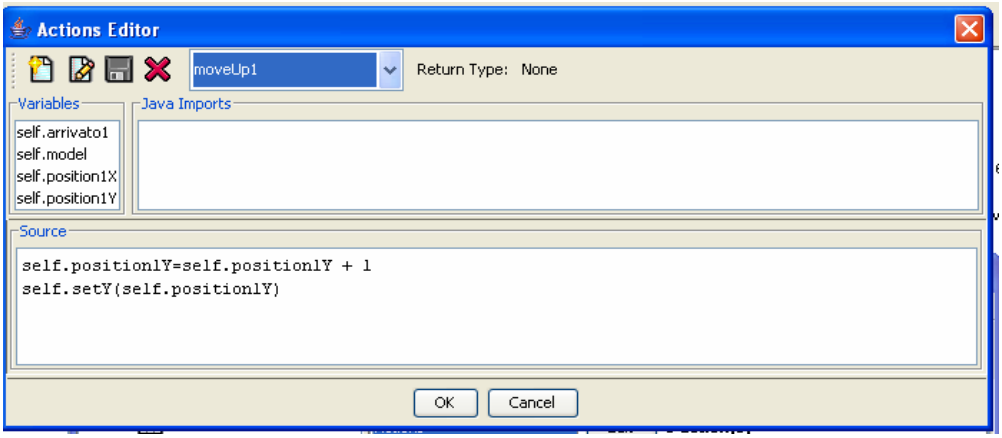
- Schedule



Studiando e Customizzando queste proprietà possiamo cambiare e modellare la natura dei nostri Agents e quindi del progetto intero.

Action User Interface

L'*Action Editor* è dove definiamo il comportamento (behavior) del nostro componente; cioè definiamo il setup e quello che il componente fa all'interno del componente.
Di seguito ecco un esempio di Action Editor che io stesso ho usato nel mio progetto:



Per esempio un'azione stessa è una collezione di Statement in Python che verranno eseguiti in fase di simulazione.

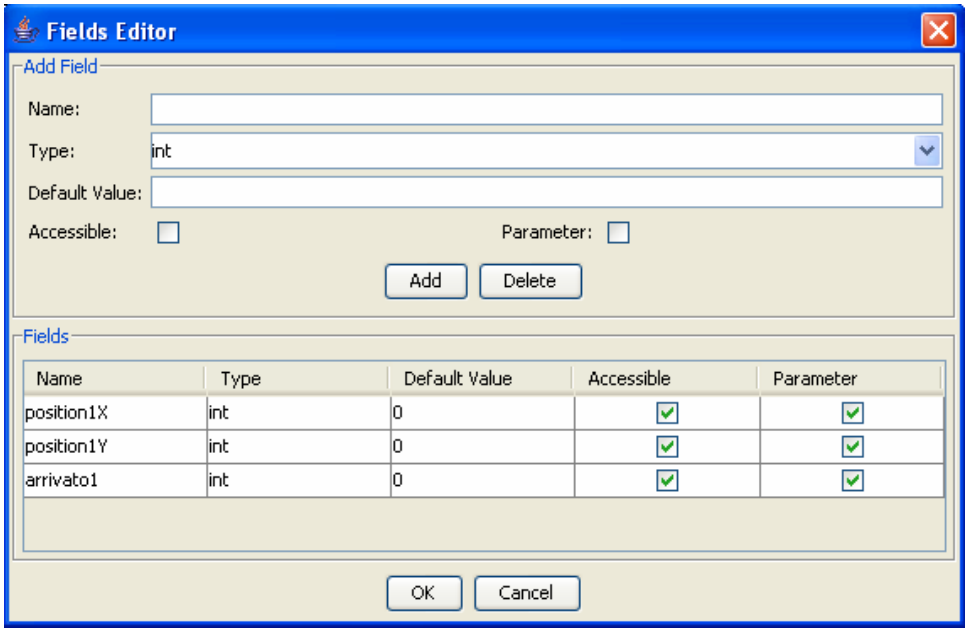
- Il pannello *Variables* mostra tutte le variabili che possono essere utilizzate dalla Action le variabili sono state precedentemente definite "dentro" Fields, come poi verrà illustrato.

- Il pannello *Java Imports* permette di importare classi e package java delle quali non è a conoscenza. Lo statement dell'import include solo il package ed il nome della classe. Es: **`java.awt.Color`**

- Infine il pannello di *Source* è un semplice text editor dove viene scritto il codice NQPy (Not Quite Python) per l'azione corrente.NQPy è un sottoinsieme del linguaggio Python e viene usato per implementare le azioni in RepastPy

Fields User Interface

La proprietà Fields che vediamo nel Pannello delle Proprietà è un insieme di field. Per meglio spiegare, definiamo field come uno speciale tipo di variabile che diviene parte del componente una volta che viene compilato.
Una volta premuto il pulsante Fields Editor nel pannello delle proprietà di un componente ecco cosa ci appare (prenderò sempre in esempio il mio progetto):



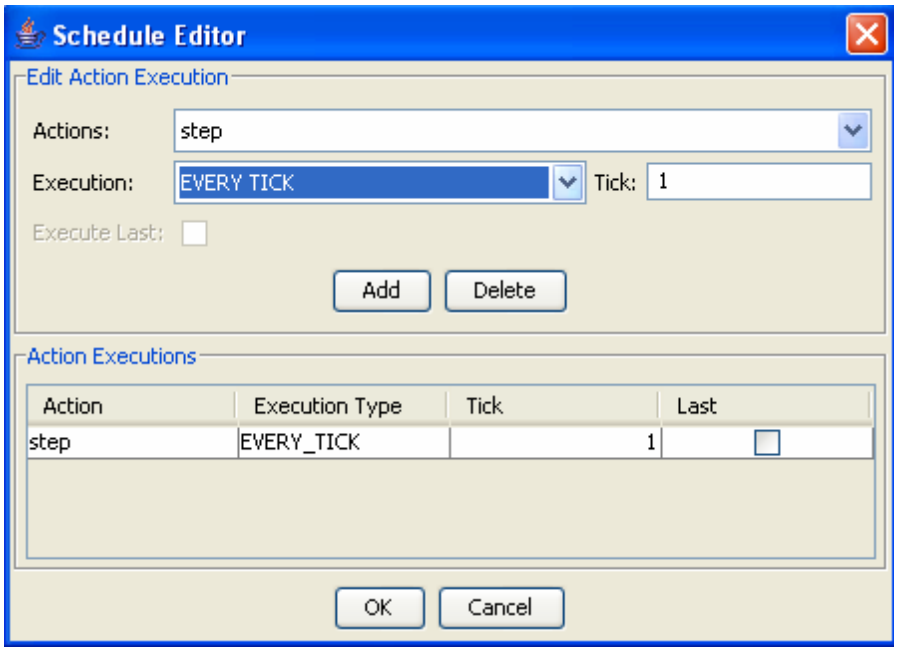
Il *Fields Editor* lo possiamo vedere come l'insieme di due parti:

- l'Add Field, dove si dichiara il nome della variabile che si vuole definire, il tipo che essa rappresenta ed il valore di default della stessa. Rendendo accessibile un field, mediante il check *Accessibile*, Repast crea in automatico metodi per leggere e/o settare il field anche al di fuori di un componente.
  Se definisco (come poi ho fatto) *position1X*, all'intenro di un componente Gamma generico, e rendendolo accessibile allora potrò anche andare a leggerlo o settarlo da un altro componente attraverso action, create automaticamente di default, quali *getPosition1X()* e *setPosition1X()*.

- ed il pannello Fields nel quale vengono elencati i vari field che sono stati definiti dall'utente o che Repast Py ha definito implicitamente.

Schedule User Interface

Repast Py è un tool per produrre simulazioni Repast. Una simulazione Repast funziona sostanzialmente così: tira fuori azioni da una lista, una coda, e le esegue.
La cosa fondamentale è che queste azioni siano ordinate l'una rispetto alle altre in modo da ordinarle e da "avviarle" ad un determinato contatore di *tick*.
Di seguito ecco un esempio di pianificazione che ho utilizzato nel mio progetto:



Lo *Scheduler Editor* viene utilizzato per decidere quando vogliamo che un'azione avvenga. Esso è composto da due parti:

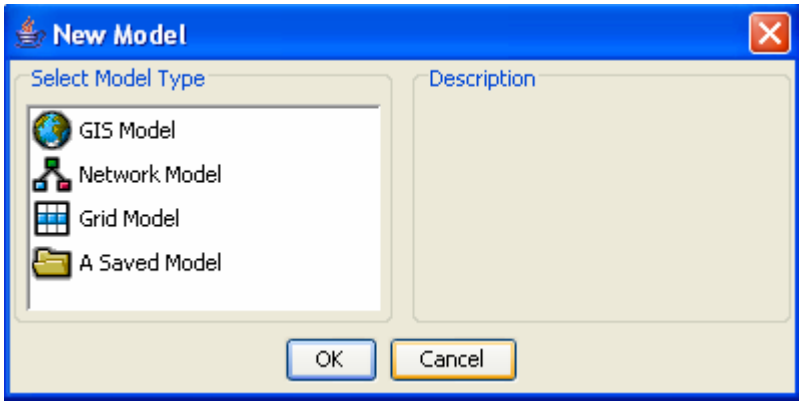
- il pannello *Edit Action Execution* nel quale in base alle action (in questo nell'immagine vediamo *step*) che abbiamo definito per componente di progetto decidiamo il momento specifico di esecuzione nel menù a tendina *Execution* ed eventualmente il *Tick* al quale far partire, durare o terminare l'action. Interessante vedere i vari tipi di Execution:
  - *Every Tick*: l'azione verrà eseguita ad ogni tick, cioè ad ogni iterazione, iniziando dal tick specificato nel campo Tick
  - *At A Single Tick*: in questo caso la specifica azione viene eseguita una volta sola ed in particolare al Tick specificato nel campo Tick
  - *At Interval*: la action ora verra eseguita ad intervalli regolari, specificati nel Tick field (esempio: se il campo Tick contiene 3 l'azione è eseguita ogni 3 tick)
  - *At End*: l'azione è eseguita alla fine del run cioè alla fine della simulazione
  - *At Pause*: viene eseguita ogni qual volta la simulazione viene messa in pausa
- e l'*Action Executions* che è la lista delle azioni che abbiamo pianificato nella Edit Action Execution

Grid Model & my Project

Il mio progetto si basa sulla simulazione di un sistema basato ad agenti che "Giocano" a calcio a 5 (o calcetto che dir si voglia..).
Dopo aver studiato le opportunità ed i vincoli che il tool mi offriva ho visto il mio sistema ad agenti così composto:

- 2 agenti giocatore: in questo caso per simulare l'idea di due concorrenti, ovviamente sviluppando, grazie all'ausilio di maggior tempo, il sistema si può arrivare ad avere 10 agenti giocatore (tutti da .."organizzare" in termini di comportamenti ed interazioni) oppure due agenti squadre, composte ognuna da 5 Agents (i nostri player appunti).
- Un agente pallone: in questo specifico progetto il comportamento dell'agente pallone è completamente passivo, l'idea finale sarebbe quella di avere un pallone in grado di reagire in base alla eventuale potenza di tiro e soprattutto alla posizione dalla quale viene calciato.

Poi una serie di agenti più o meno attivi potrebbero essere inseriti sempre nella mia idea di progetto finale emergono eventuali agenti linee di campo o un agente arbitro che decide in base allo svolgimento del gioco se sanzionare o meno qualche giocatore e/o qualche squadra.



Quando iniziamo un progetto con Repast Py, il tool ci permette di scegliere fra tre tipologie di progetti differenti (immagine sopra).

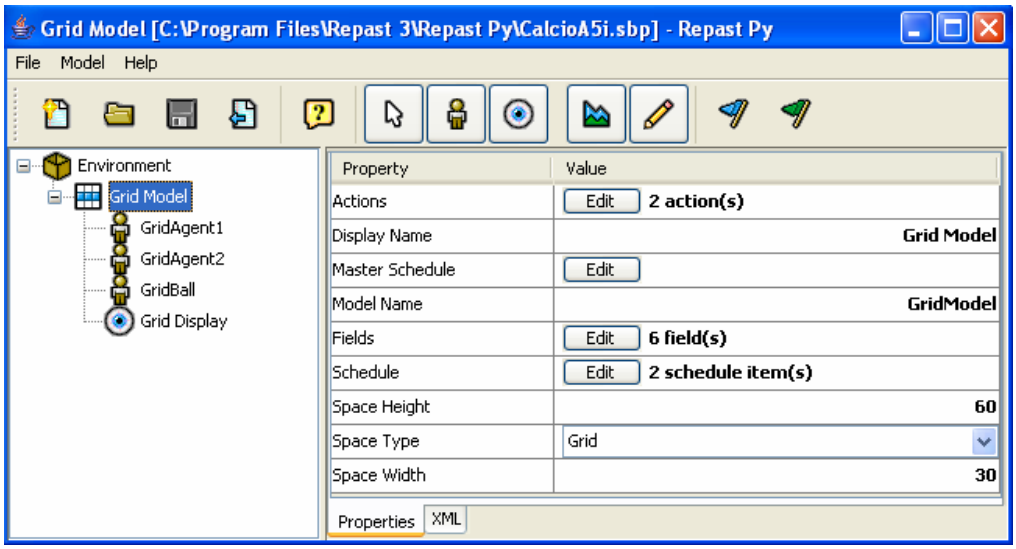
- *Gis Model*

- *Network Model*

- *Grid Model*

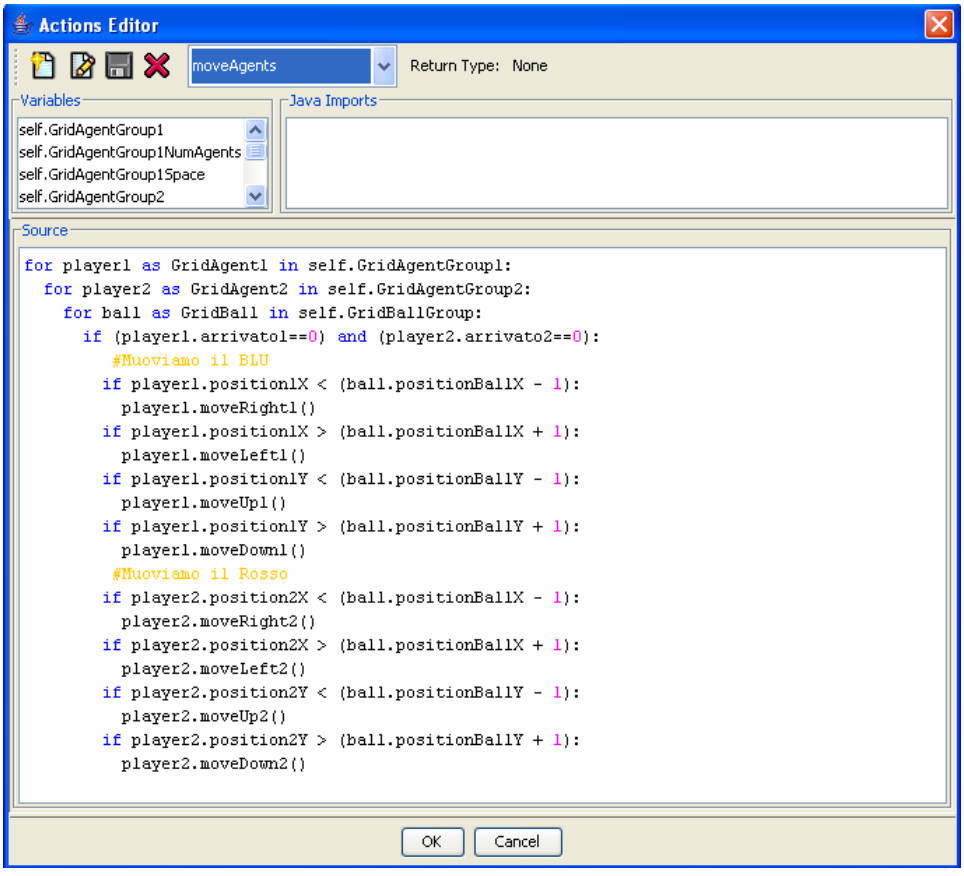
Nel mio progetto ho scelto, ovviamente un Grid Model, proprio perché è una perfetta base di partenza per chi volesse eseguire simulazione grafiche in particolare su piattaforme a 2-D.
In un modello Grid un agente occupa al più una cella della griglia ad ogni istante di tempo (tick).

Nella figura sottostante si può vedere chiaramente come ho improntato il mio progetto e quali features permette di modellare un Grid Model.



Ora visualizziamo i vari campi del modello:

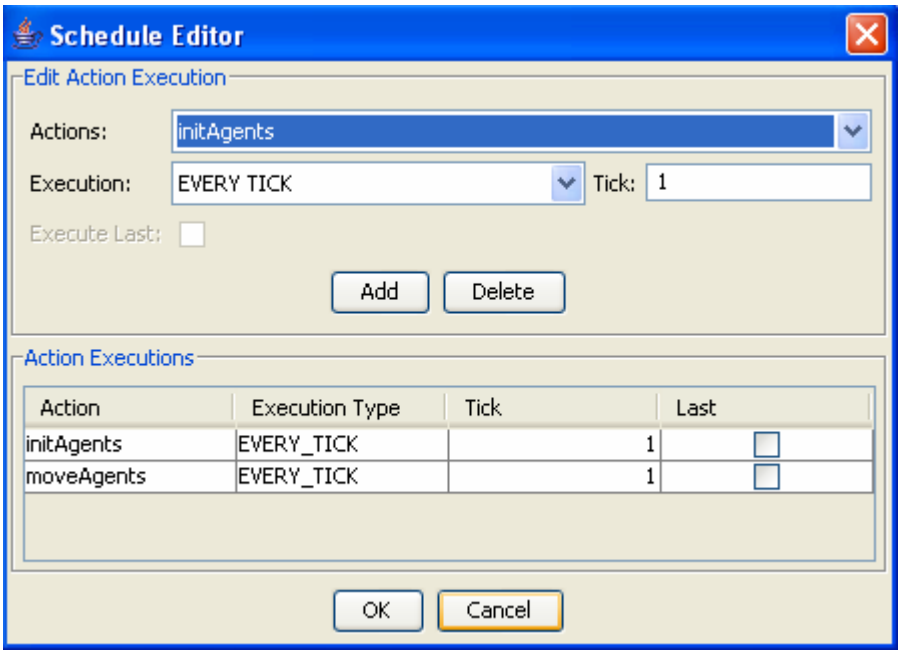
Action: nel quale vi sono due tipi di azioni: initAgent e moveAgent. Con la initAgent il modello grid si “aggiorna” sullo stato del sistema, mentre con la moveAgent muove gli agenti alla conquista della palla (sempre che uno dei due agenti non vi sia gia arrivato..).



In questa action vediamo come per ogni giocatore (Blu e Rosso) si controlla continuamente la posizione rispetto al pallone e, se nessuno dei due è arrivato sul pallone, vengono chiamate funzioni specifiche del componente Player1 e Player2, che poi sarebbero i nostri agenti.
Ad esempio quando viene chiamata la funzione **`player1.moveRight1()`**, *moveRight1()* è un metodo (o action) definito dentro al componente Player1, che in questo caso rinomina GridAgent1.

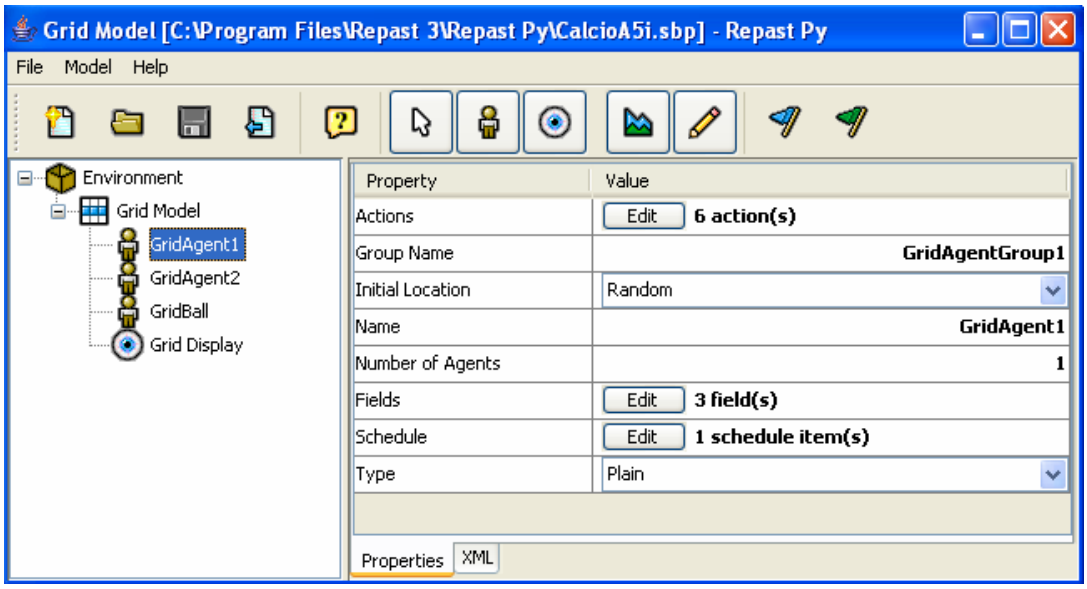
Il modello Grid essendo, dopo l'environment, il padre dei vari componenti che allochiamo importa tutte le variabili (fields) e tutti i metodi (actions) definiti nei singoli componenti figli, quindi diciamo pure negli agenti figli.
Per completezza diciamo che in questo caso, nulla conta il fatto di aver reso accessibile una variabile, infatti se viene "checkato" il campo *accessibile* di una variabile la rendiamo visibile e modificabile all'esterno anche da componenti di pari livello. Qui in questo caso è il Grid Model che ha riferimento ed accesso a tutto.

Nel modello sotto la voce Schedule si elencano e soprattutto si pianificano le varie action che il modello definisce:
Nel mio caso sono 2: initAgents e moveAgents, entrambe vengono eseguite ad ogni tick.



La action moveAgents come già detto muove gli agenti fino a quando uno dei due non raggiunge il pallone per primo, dopo di che anche l'agente "arrivato" secondo si ferma. Questa è stata una mia scelta progettuale perché ipotizzo che per un'eventuale sviluppo futuro, da questo momento in poi inizi una fase di gioco tra attacco e difesa, quindi ora chi non ha la palla inizia a fare il "difensore".

Per quanto riguarda gli agenti giocatori RepastPy mi ha permesso di realizzarli così:



Per ogni agente ho definito 6 Actions, esempio per l'agente *GridAgent1* cioè il giocatore Blu ho creato:

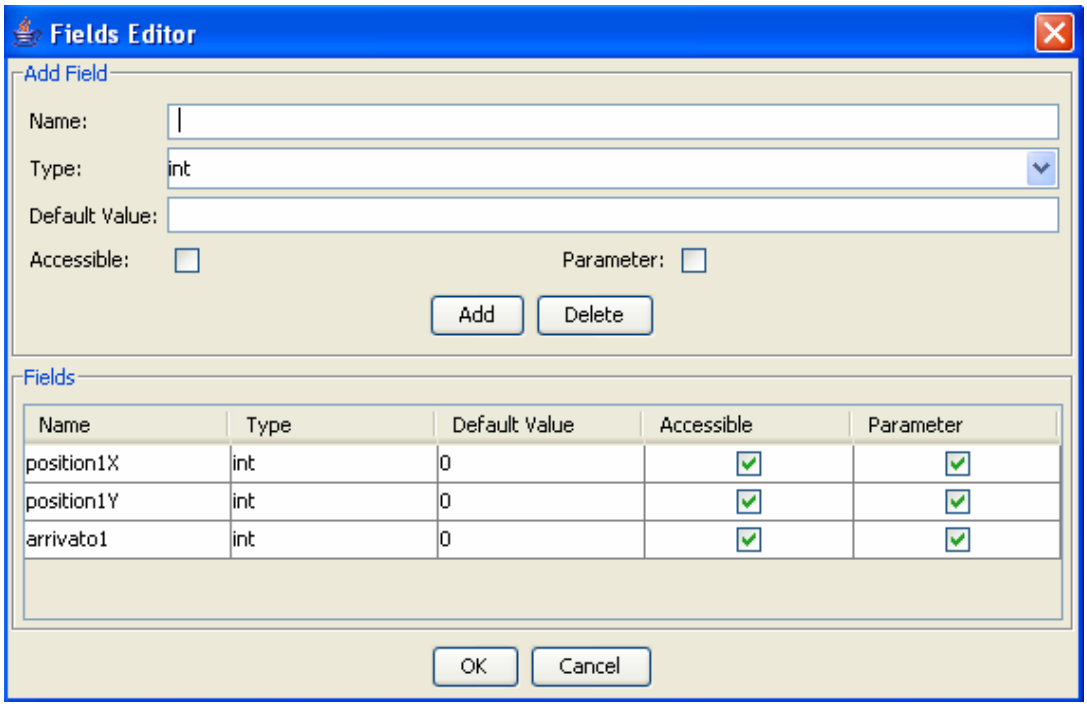
- *Step*: in cui semplicemente definiamo il colore dell'agente (BLU)
- *updateState1*:dove viene letto ed aggiornata la posizione del giocatore, quindi in quale grid-cell si trova. Questo mediante le operazioni:

```
self.position1X=self.getX()
self.position1Y=self.getY()
```

- *moveUp1*: action che muove il giocatore 1 di una posizione in alto
- *moveDown1*: stessa cosa, ma ovviamente verso il basso
- *moveRight1*: movimento verso destra
- *mveLeft1*:movimento verso sinistra

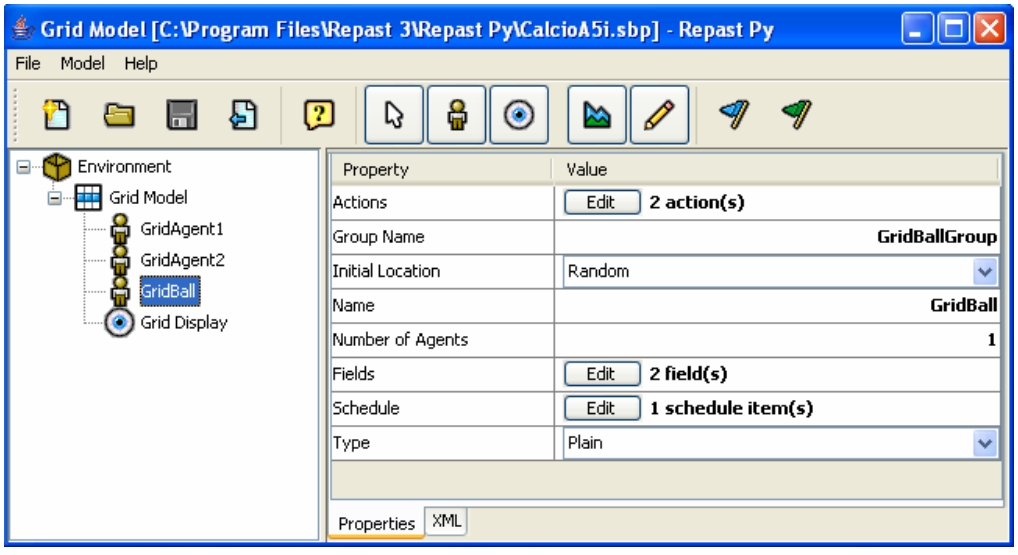
La posizione iniziale dell'agente la si può determinare nel campo *Initial Location* scegliendo dal menù a tendina o *Random* o *Manual*.
Per scelta progettuale ho preferito scegliere random per tale posizione, e questo per entrambe gli agenti, in modo da iniziare ogni volta un "gioco nuovo". Ecco così anche svelata l'importanza dell' action updateState1() ce legge dove casualmente è stato posto il giocatore Blu (tutto ciò vale anche per il giocatore Rosso, quindi con updateState2()).

Per ogni GridAgent ho definito 3 campi di variabili:

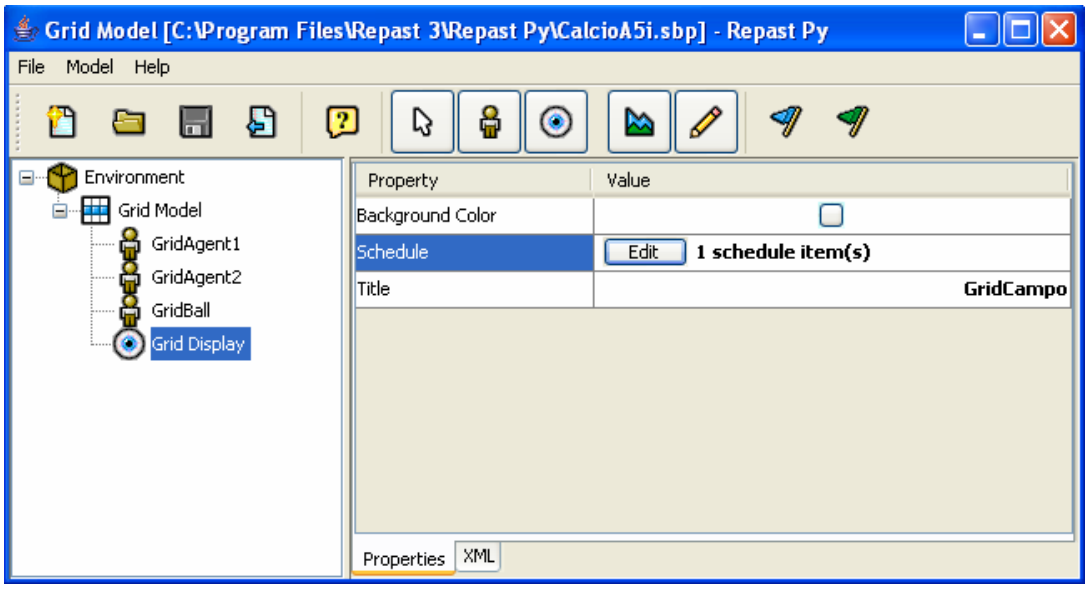


position1X e position1Y come si può capire servono per gestire la posizione del mio agente sulla griglie, mentre arrivato1 mi aiuta a capire quando un agente è arrivato sul pallone.
Per eventuale sviluppo futuri si potrebbe inserire variabili come la stanchezza, la velocità, la precisione di passaggio o tiro. Tutto insomma per caratterizzare sempre meglio il comportamento degli agenti.

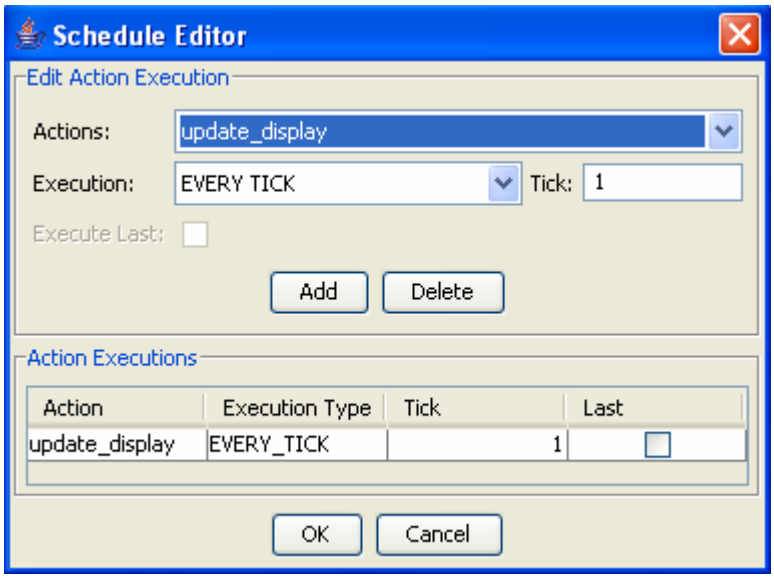
Ultimo agente che ho posizionato nel sistema è stato il pallone, per il quale non ho definito particolari azioni se non quella di *Step()*, per definire il colore della palla, e quella di *updateStateBall()* per aggiornare la posizione del pallone.



In ultimo molto utile e comodo è il fatto che un Model Grid permette di definire con un semplice pulsante  un Grid Display per permettere immediatamente ed anche facilmente un layout della simulazione. Tutti questi comandi possono aiutare anche chi si affacciasse al mondo degli agenti e delle simulazioni per la prima volta.



Per il display vengono scelti il *Background Color*, lo *Schedule* (semplificato) ed il *Title* della finestra display che poi ci mostrerà la simulazione.
Per quanto riguarda la pianificazione degli eventi del display, abbiamo un'unica action (vedi figura sotto) definita di default da Repast Py: *update_display()*

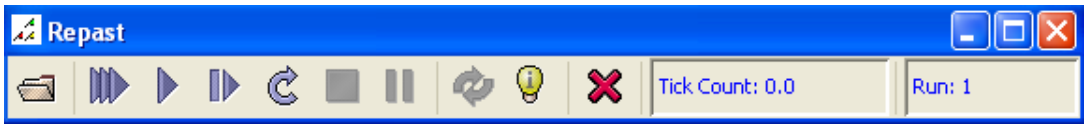


Si può comprendere come tale action venga eseguita ad ogni tick temporale, in modo tale da mantenere la vista sempre aggiornata.

Simulazione Finale

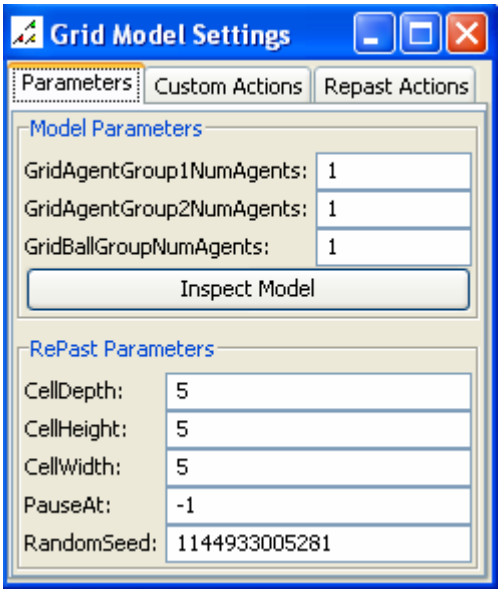
Repast Py offre una gradevole interfaccia anche per eseguire la simulazione finale.
Dopo aver compilato il progetto ed aver evidenziato l'assenza di errore, come detto in precedenza, è possibile eseguire la simulazione clicckando semplicemente il tasto run  dalla tool bar.

A questo punto avremo a disposizione una barra per la gestione della simulazione:

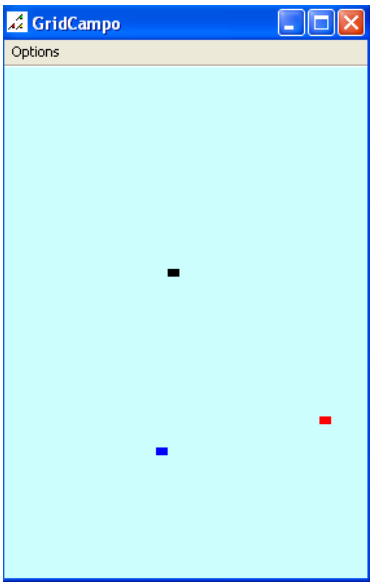


dove attraverso la quale è possibile aprire altri progetti, far partire la simulazione di uno  o più  progetti concorrenti, eseguire singoli tick di un progetto  (in questo caso si vedrà il Tick Count incrementarsi di 1).
Inoltre è possibile fermare il run mediante il tasto  ; e ricaricare la simulazione per eventualmente ripeterla tramite il tasto  di reload.
Ovviamente terminarla con il pulsante  .

Inoltre Repast Py offre una finestra, durante la simulazione durante la quale settare la propria applicazione (figura sotto):

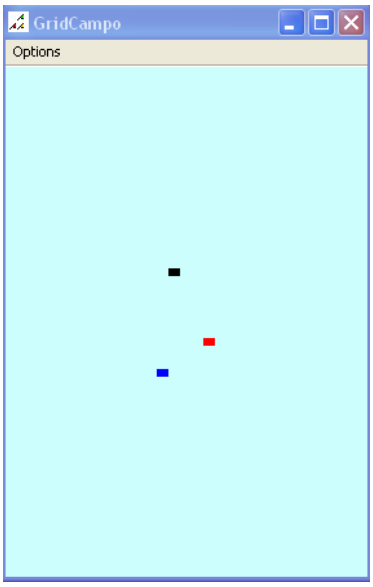


La mia simulazione offre il seguente output, il display *GridCampo* (nome del display inserito dal Pannello delle Proprietà del componente GridDispaly) mostra o meglio rappresenta il nostro campo di gioco (le dimensioni del campo sono state definite in fase di dichiarazione del Model Grid).
All'interno del mio campo ho posto due Giocatori (Blu e Rosso: *GridAgent1* e *GridAgent2*) e un agente Pallone (Nero: GridBall).



Mediante il pulsante  , ripetuto più volte, eseguo la simulazione tick by tick, in modo da poter monitorare ogni variazione di stato dell'intero sistema.

Proseguendo con la simulazione ecco come si presenta l'andamento del sistema (come poi è possibile vedere scaricando il tool ed il mio file di progetto, ed eseguendolo)
Vediamo come entrambe i giocatori (figura a destra) si avvicinino verso la palla (cella nera) entrambe senza al momento poter capire che a parità di velocità il primo ad arrivare sarà anche quello più vicino.
Quindi possiamo capire come anche il concetto di vista (nel senso di rendere un giocatore capace di vedere e/o di prevedere che l'avversario è più vicino e quindi arriverà prima) è un aspetto implementabile e molto interessante per questo tipo di simulazioni.



Procedento nella simulazione sempre mediante il tasto di Play tick by tick arriviamo sino alla situazione finale in cui uno dei due giocatori arriva fino alla palla (figura a destra) ed a questo punto si possono aprire una serie infinità di scenari, dal più semplice: una **`print("Sono il Giocatore X e sono 1°")`** che comunica al sistema sull'output window che è arrivato primo fino ad attivare nuovi scenari di possibili Game e di complesse funzioni o action tutte implementabili tramite RepastJ o Repast.Net.