Simulazione di sistemi autonomi tramite aggregate programming con ScaFi

Cristian Paolucci

31 ottobre 2017

Indice

Capitolo 1

Abstract

In questo progetto si valuterà l'efficacia dell'uso di un framework basato su aggregate programming come ScaFi nel campo dei sistemi autonomi. Per confrontare i risultati verrà fatto un paragone con Jason, un linguaggio utilizzato per sistemi automi multi-agente durante il corso. L'elaborato in sé consiste in una simulazione di entità autonome in un ambiente. Ogni entità col suo ciclo di controllo sceglie il piano da eseguire fino alla sua morte. Più nello specifico è stato usato uno scenario dove degli animali carnivori (leoni) inseguono degli animali erbivori (gazzelle e bufali).

Capitolo 2

Introduzione

Il concetto di autonomia spazia in diversi campi, può essere ritrovato in natura in diverse forme: dal microscopico al macroscopico. Tuttavia qui verranno presi in considerazione la varianti autonome appartenenti al mondo dell'informatica, più nello specifico nel campo dei **Multi-Agent Systems** (MAS). L'autonomia in un agente si può definire come la capacità di agire indipendentemente, ovvero incapsulare il controllo del proprio stato interno. Tra i vari agenti di un MAS quindi non ci deve essere mai passaggio di controllo, ma solo di conoscenze e informazioni.

Quindi aggiungendo il concetto di autonomia rendiamo un agente in grado di scegliere il proprio comportamento basandosi sulla percezione del mondo e degli altri agenti. Spesso per descrivere i comportamenti di un agente si usano delle caratteristiche generiche spesso riconducibili al concetto di autonomia in sé. Le più utilizzate sono:

- Reattività: capacità di reagire ai cambiamenti del mondo o degli altri agenti;
- Situazionalità: accoppiamento tra agente e *environment*, col quale interagisce;
- Proattività: capacità di un agente di prendere l'iniziativa;
- Abilità sociali: relazionarsi con altri agenti per raggiungere l'obiettivo.

Un sistema che rappresenta gli aspetti precedentemente descritti può essere considerato autonomo, anche se non fosse convenzionalmente un sistema ad agenti, come *Scafi*, che si occupa di simulare entità tramite l'uso dell'**Aggregate Programming**.

Per creare dei sistemi autonomi è necessario quindi adottare uno schema che permetta di gestire e incapsulare il controllo dello stato interno in un *control loop*. Un metodologia vista durante il corso è il framework **BDI**. Gli agenti BDI sono degli agenti che svolgono un determinato compito tramite l'uso di:

- **Belief**: che rappresentano la conoscenza del mondo;
- **Desires**: che rappresentano lo stato del mondo che si vuole raggiungere;
- **Intention**: che rappresentano i mezzi per soddisfare i **Desires**

Un'implementazione di questo *framework* visto durante il corso è *Jason* che si basa su *AgentSpeak(L)*, quindi verrà utilizzato come paragone per descrivere l'autonomia su *scafi*.

Capitolo 3

Scafi

Scafi è framework basato su Scala, il quale implementa la semantica del *Field Calculus* e fornisce API per simulazione ed esecuzione di applicazioni basate su aggregate-programming.

Nonostante fosse possibile ricreare il sistema utilizzando una infrastruttura modellata in *Java* come *Jason*, *Scafi*, o più in generale l'**Aggregate Programming**, spicca per la sua praticità. Nell'ambiente di *Scafi* è possibile scambiare dati in maniera estremamente semplice tramite funzioni di base o di alto livello, sia che si tratti di inviare a tutti i vicini un valore, o di aggregarli. Oltretutto queste funzionalità sono componibili, quindi è possibile ricavare un valore, inviarlo, riceverlo e rielaborarlo tramite un'unica funzione.

Jason invece non è limitato da una sola implementazione del sistema ad agenti e permette di usare quella preferita. Negli esempi visti a lezione Jason è stato usato basandosi su **Jade**. Questo gestisce le interazioni tra agenti basandosi sullo scambio di messaggi. In una architettura basata sullo scambio di messaggi, il messaggio stesso è l'elemento principale della computazione. Colui che invia il messaggio deve crearlo seguendo determinati pattern, mentre chi lo riceve deve identificare quale pattern è stato utilizzato per comprenderne il contenuto. La comunicazione in *Scafi*, invece, è più trasparente, dato che nasconde i meccanismi di gestione dei messaggi; ogni volta che un nodo richiede i dati ai suoi vicini, è assicurato il fatto che i valori restituiti siano del tipo richiesto.

Scafi, essendo un linguaggio dichiarativo, facilita la componibilità e la comprensibilità del codice, rendendo facile il riuso e la manutenibilità. La semplicità che si ottiene nel gestire un sistema distribuito con l'**Aggregate Programming** è probabilmente uno dei punti chiave di Scafi, che permette di

sviluppare comportamenti anche autonomi. Jason utilizza Java e tuProlog che da quel punto di vista non offrono gli stessi vantaggi naturalmente.

Scafi quindi simula delle entità che sono simili ad agenti come proprietà. Quindi, come è stato possibile sviluppare comportamenti autonomi su un'infrastruttura ad agenti è possibile ricrearli su scafi. Come con Jason l'obiettivo è quello di implementare un sistema ad agenti **BDI**. Sfruttando le capacità del *framework* è implementabile un sistema di questo tipo per quanto basilare con un ciclo di controllo. All'inizio di ogni ciclo vengono creati i Belief. Questi possono essere estratti dal ciclo precedente o calcolati tramite le funzioni dell'aggregate-programming o di Scala. In seguito viene scelto il piano sfruttando delle guardie. Queste possono essere direttamente collegate all'esecuzione di un piano per quanto complesse, oppure potrebbero essere usate per creare un macro-piano composto da diversi piani sfruttando la componibilità di Scafi. All'interno delle guardie e dei piani vengono codificati **Desires** e **Intention**, ovvero lo stato finale che deve raggiungere il mondo e come farlo. Questa parte differisce da Jason solo nell'uso delle tecnologie poiché si rifanno allo stesso framework teorico per quanto tutt'ora sia limitato.

Le differenze principali che si evincono tra Jason e Scafi sono la predisposizione e l'efficacia degli strumenti. Jason è nato per sviluppare un sistema ad agenti autonomi e sfrutta le capacità di AgentSpeak(L)estendendolo e permettendo la scelta dell'infrastruttura ad agenti, mentre scafi ha come obiettivo la simulazione di entità sfruttando l'aggregate programming ignorando la piattaforma che poi si andrà ad utilizzare. Jason supera l'uso di Scafi singolarmente poiché implementa a pieno tutto AgentSpeak(L) e le semantiche operazionali. Oltretutto supporta diversi linguaggi di comunicazione tra agenti ed è altamente personalizzabile. Quindi Scafi necessità di lavoro per essere trasformato in un vero e proprio framework per lavorare efficientemente con dei sistemi autonomi come descritti in letteratura, tuttavia fornisce una piattaforma adatta alla simulazione.

Capitolo 4

Caso di studio

La demo sviluppata consiste in una simulazione di uno scenario in cui ci sono entità diverse che hanno lo scopo di sopravvivere.

Tipologie di agente simulato:

- Gazzella: passa la sua esistenza a brucare l'erba e reagisce alla presenza di leoni scappando;
- Bufalo: seguendo regole di flocking si sposta in branco e reagisce alla presenza di leoni cambiando direzione;
- Leone: evita gli altri leoni e si muove casualmente nello spazio, quando percepisce una erbivoro scatta per cacciarlo.

Ogni entità può morire e questo avviene: entro una certa distanza da un predatore per gli erbivori o dopo un numero di cicli senza mangiare per i carnivori.

I belief devono essere condivisi data la natura di scafi per scambiare informazioni tra i nodi e sono calcolati all'inizio di ogni round o mantenuti dal ciclo precedente. In questa demo per esempio lo status isAlive viene passato da un ciclo all'altro senza essere ricalcolato, mentre i vicini vengono riesaminati per non contare nodi che sono usciti dal range o si sono spostati.

```
...

var isAlive = tuple._2 // Where tuple is the output of the previous round

val n = foldhood[Seq[(Int, T)]](Seq[(Int, T)]())(_ ++ _) {
  Seq(nbr{mid().asInstanceOf[Int]} -> nbrVector())
```

```
}.toMap // Map neighbors with their ID and direction

// Get subset based on roles
val gazelleNeighbor = foldhood[Seq[Point3D]](Seq[(Point3D)]())(_ ++ _){
  mux(nbr(sense1))(Seq(n.apply(nbr(mid().asInstanceOf[Int]))))(Seq())
}

val buffaloNeighbor = foldhood[Seq[Point3D]](Seq[(Point3D)]())(_ ++ _){
  mux(nbr(sense2))(Seq(n.apply(nbr(mid().asInstanceOf[Int]))))(Seq())
}

val lionNeighbor = foldhood[Seq[Point3D]](Seq[(Point3D)]())(_ ++ _){
  mux(nbr(sense3))(Seq(n.apply(nbr(mid().asInstanceOf[Int]))))(Seq())
}


val predatorDistance = distanceTo(sense3) // Get predator distance

...
```

Tramite la foldhood è possibile raccogliere informazioni dai nodi vicini, nel nostro caso si ricava nbrVector che è il vettore di spostamento che deve essere usato per raggiungere un determinato nodo. Quindi con le varie foldhood è possibile sapere il vettore per raggiungere ogni vicino in base al tipo, questo verrà utilizzato per gli spostamenti in seguito. Il distanceTo invece restituisce la distanza da un determinato nodo se presente un collegamento, anche indiretto, altrimenti restituisce un valore tendente ad infinito.

La scelta del piano da eseguire avviene tramite delle guardie. Queste sono create con i costrutti di scafi ovvero **mux** e **branch**. Il primo restituisce in output la computazione di uno di due casi in base alla condizione, ma effettua entrambe le computazioni senza dividere i campi computazionali. Il secondo invece divide i campi in base alla condizione permettendo una computazione più leggera nel caso in cui non fossero necessarie interazioni. Dato che tutte le interazioni sono eseguite all'inizio per calcolare i Belief in questo caso è possibile utilizzare solo dei branch.

```
...

// Plan selection and execution

branch(sense1){
  branch(isAlive) {
    branch(lionNeighbor.nonEmpty){
```

```
      branch(predatorDistance < deathDistance){
        (die(), !isAlive)
      } {
        (liveAndRunGazelle(gazelleNeighbor ++ buffaloNeighbor, sense1 ||
            sense2, lionNeighbor, tuple._1), isAlive)
      }
    } {
      (liveGazelle(gazelleNeighbor ++ buffaloNeighbor, sense1 || sense2,
          tuple._1), isAlive)
    }
  } {
    ((.0,.0), isAlive)
  }
} {
  branch(sense2){
    branch(isAlive) {
      branch(lionNeighbor.nonEmpty){
        branch(predatorDistance < deathDistance){
          (die(), !isAlive)
        } {
          (liveAndRunBuffalo(buffaloNeighbor, sense2, lionNeighbor,
              tuple._1), isAlive)
        }
      } {
        (liveBuffalo(buffaloNeighbor, sense2, tuple._1), isAlive)
      }
    } {
      ((.0,.0), isAlive)
    }
  } {
    branch(sense3){
      branch(isAlive) {
        branch(gazelleNeighbor.nonEmpty){
          (hunt(gazelleNeighbor, sense1, tuple._1), isAlive)
        } {
          branch(buffaloNeighbor.nonEmpty && buffaloNeighbor.size < 5){
            (hunt(buffaloNeighbor, sense2, tuple._1), isAlive)
          } {
            branch(timer(10000.0)){
              (die(), !isAlive)
            } {
              branch(lionNeighbor.size > 1) {
                (avoidLions(tuple._1, lionNeighbor), isAlive)
              } {
                (wander(tuple._1), isAlive)
              }
            }
          }
        }
```

```
      } {
        ((.0,.0), isAlive)
      }
    } {
      ((.0,.0), isAlive)
    }
  }
}

...

// Plans

def die(): (Double, Double) = (10.0,10.0) // Out of bound -> position a
    node in point (0,0)

def liveAndRunGazelle(neighbor: Seq[Point3D], herbivore: Boolean,
    predator: Seq[Point3D], lastVec: (Double, Double)): (Double, Double) = {
  val vectorRepulsion: (Double, Double) =
      this.separation(repulsionGazelleRange, neighbor)
  val vectorAlignment: (Double, Double) = this.alignment(lastVec,
      herbivore)
  val vectorAttraction: (Double, Double) = this.cohesion(neighbor)
  val vectorObstacle: (Double, Double) = this.obstacle(predator)

  val vectorX: Double = (vectorAttraction._1.toDouble * attractionForce
    + vectorRepulsion._1.toDouble * repulsionForce
    + vectorAlignment._1.toDouble * alignmentForce
    + vectorObstacle._1.toDouble * obstacleForce)
  val vectorY: Double = (vectorAttraction._2.toDouble * attractionForce
    + vectorRepulsion._2.toDouble * repulsionForce
    + vectorAlignment._2.toDouble * alignmentForce
    + vectorObstacle._2.toDouble * obstacleForce)

  val (newX, newY) = normalizeToScale(vectorX, vectorY)
  (newX, newY)
}

def liveGazelle(neighbor: Seq[Point3D], herbivore: Boolean, lastVec:
    (Double, Double)): (Double, Double) = {
  // Graze simulation
  (.0,.0)
}

def liveAndRunBuffalo(neighbor: Seq[Point3D], herbivore: Boolean,
    predator: Seq[Point3D], lastVec: (Double, Double)): (Double, Double) = {
  val vectorRepulsion: (Double, Double) =
      this.separation(repulsionBuffaloRange, neighbor)
  val vectorAlignment: (Double, Double) = this.alignment(lastVec,
```

```
      herbivore)
  val vectorAttraction: (Double, Double) = this.cohesion(neighbor)
  val vectorObstacle: (Double, Double) = this.obstacle(predator)

  val vectorX: Double = (vectorAttraction._1.toDouble * attractionForce
    + vectorRepulsion._1.toDouble * repulsionForce
    + vectorAlignment._1.toDouble * alignmentForce
    + vectorObstacle._1.toDouble * obstacleForce)
  val vectorY: Double = (vectorAttraction._2.toDouble * attractionForce
    + vectorRepulsion._2.toDouble * repulsionForce
    + vectorAlignment._2.toDouble * alignmentForce
    + vectorObstacle._2.toDouble * obstacleForce)

  val (newX, newY) = normalizeToScale(vectorX, vectorY)
  (newX, newY)
}

def liveBuffalo(neighbor: Seq[Point3D], herbivore: Boolean, lastVec:
    (Double, Double)): (Double, Double) = {
  val vectorRepulsion: (Double, Double) =
      this.separation(repulsionBuffaloRange, neighbor)
  val vectorAlignment: (Double, Double) = this.alignment(lastVec,
      herbivore)
  val vectorAttraction: (Double, Double) = this.cohesion(neighbor)

  val vectorX: Double = (vectorAttraction._1.toDouble * attractionForce
    + vectorRepulsion._1.toDouble * repulsionForce
    + vectorAlignment._1.toDouble * alignmentForce)
  val vectorY: Double = (vectorAttraction._2.toDouble * attractionForce
    + vectorRepulsion._2.toDouble * repulsionForce
    + vectorAlignment._2.toDouble * alignmentForce)

  val (newX, newY) = normalizeToScale(vectorX, vectorY)
  (newX, newY)
}

def hunt(prey: Seq[Point3D]): (Double, Double) = {
  val vectorAttraction: (Double, Double) = this.cohesion(prey)

  val vectorX: Double = vectorAttraction._1.toDouble * attractionForce
  val vectorY: Double = vectorAttraction._2.toDouble * attractionForce

  val (newX, newY) = normalizeToScale(vectorX, vectorY)
  (newX * 2, newY * 2)
}

def avoidLions(neighbours: Seq[Point3D]): (Double, Double) = {
  val vectorObstacle: (Double, Double) = this.obstacle(neighbours)
```

```
  val vectorX: Double = vectorObstacle._1.toDouble * obstacleForce
  val vectorY: Double = vectorObstacle._2.toDouble * obstacleForce

  val (newX, newY) = normalizeToScale(vectorX, vectorY)
  (newX * 2, newY * 2)
}

def wander(lastVec: (Double, Double)): (Double, Double) = {
  movement(lastVec)
}
```

Gazzella (sense1):

- liveAndRunGazelle: tramite delle regole di flocking scappa dai leoni e segue qualsiasi altro erbivoro trovi sulla sua strada;
- liveGazelle: rimane ferma simulando di brucare l'erba o comunque rimanere in allerta;
- die: sposta la gazzella nel punto (0,0) e imposta lo stato isAlive a false.

Bufalo (sense2):

- liveAndRunBuffalo: tramite delle regole di flocking scappa dai leoni e segue solo altri bufali;
- liveBuffalo: tramite delle regole di flocking segue solo altri bufali;
- die: sposta il bufalo nel punto (0,0) e imposta lo stato isAlive a false.

Leone (sense3):

- hunt: insegue le prede più vicine, dando precedenza alle gazzelle;
- wander: effettua spostamenti semi-casuali per cercare eventuali prede;
- avoidLions: evita ogni altro leone nelle vicinanze;
- die: sposta il leone nel punto (0,0) e imposta lo stato isAlive a false.

Capitolo 5

Conclusioni

In definitiva si è riscontrato che Scafi offre un buon supporto per sistemi autonomi ed il progetto potrebbe essere portato avanti per conferire una modularità e funzionalità migliorate. In rapporto con altri linguaggi specifici per i sistemi autonomi quindi risulta ancora grezzo, però è stato comunque possibile realizzare una demo funzionale per dimostrare che Scafi permette di sviluppare comportamenti anche complessi come il flocking.