Sistemi Multi-Agente LM

AGV-Collision

Manuel Bartolini (manuel.bartolini2@studio.unibo.it)
Paolo Contessi (paolo.contessi@studio.unibo.it)
Luca Guerra (luca.guerra3@studio.unibo.it)

Sommario

Nella vita di tutti i giorni (quando camminiamo, guidiamo, ...) ci muoviamo con l'attenzione di evitare collisioni con chi o cosa ci circonda.

Allo stesso modo sarebbe importante (soprattutto nell'ambito AGV) che un agente autonomo, collocato in uno spazio, assieme ad altri agenti, fosse in grado di percepire il moto delle entità attorno a se e capire come perseguire il proprio scopo muovendosi, senza per questo collidere con altri. Per questo abbiamo pensato alla possibilità che agenti dotati di intelligenza possano interagire fra di loro, nel caso prevedano il coinvolgimento in uno scontro, e attraverso un protocollo di negoziazione possano stabilire le rispettive traiettorie da intraprendere; dal momento che non tutte le entità devono avere lo stesso grado di intelligenza i protocolli dovranno adattarsi a tali evenienze. Per questo l'obiettivo è quello di simulare un magazzino, nel quale dei muletti autonomi si occupino di stoccare della merce arrivata ai dock di input e di evadere degli ordini prelevando dagli scaffali il materiale e portandolo ai dock di output.

Per modellare diversi gradi di intelligenza, e quindi diversi approcci alla negoziazione, si è pensato alla possibilità che i muletti possano perdere parte della merce durante il tragitto; in tal caso alcune persone (dotati di un carrello manuale) si preoccuperanno del recupero della merce dispersa, al fine di ripristinare la normalità. I pallet e le persone non potranno interagire con i veicoli come visto in precedenza: la merce non ha possibilità di spostarsi e comunicare, così come un essere umano non ha l'abilità di concordare tempestivamente una soluzione (normalmente ci si limita a scappare); per questo motivo sarà necessario studiare delle soluzioni differenti che garantiscano comunque la sicurezza.

INDICE

1

ANALISI DEI REQUISITI

Appare chiaro, dal sommario, come un ambito applicativo come quello scelto si apra ad una miriade di possibilità e scenari di modellazione, primo fra tutti un agente in grado di guidare autonomamente un auto per le vie cittadine. Lo scenario applicativo che si è scelto di sviluppare però non è questo, infatti si è scelto di modellare uno scenario di ambito più prettamente industriale ovvero il movimento di muletti all'interno di magazzini di stoccaggio merci, che come sappiamo oggi sono molto diffusi. Si capisce quindi come sia necessario, in un ambiente applicativo di questo tipo, modellare alcuni oggetti basilari per lo sviluppo dell'applicazione: primo fra tutti il veicolo, capace di muoversi in qualche modo nell'ambiente circostante e dotato di features che gli permettano l'individuazione di ostacoli, sia fissi che in movimento e di distinguere tra essi, ma soprattutto che sia in grado, attraverso un determinato protocollo, di negoziare con gli altri veicoli per evitare possibili collisioni. Il modello, inoltre, prevede la possibilità che il veicolo perda il pacco, cosa che nella realtà può accadere, o perché il pacco è fissato male o per effetto di una curva troppo veloce. Ci sono molte cause possibili perché questo succeda, ma a noi questo in realtà non importa, l'importante è modellare qualcosa che rimedii a questa situazione; ad esempio una diversa tipologia di veicolo o, eventualmente, un agente che simuli un eventuale intervento umano di qualche tipo teso al recupero di questi pacchi.

A questo punto è palese come serva anche aggiungere una qualche entità che funga da manager e gestisca ad un livello più alto tutte le situazioni, grazie ad un certo grado di conoscenza della situazione nel magazzino, così come una nozione di percorso che permetta al veicolo di raggiungere determinate destinazioni scegliendo il miglior percorso possibile; ultimo, ma non in ordine di importanza, un environment dentro al quale i nostri agenti possano comunicare tra loro ed in conseguenza decidere il proprio comportamento.

1.1 SCENARI

Gli scenari possibili che si vengono a creare all'interno del magazzino sono fondamentalmente quattro:

MOVIMENTO PACCHI I pacchi vengono spostati dai muletti a seconda che siano in entrata o in uscita dal magazzino. Quando un pacco arriva al dock d'entrata del magazzino, uno dei muletti liberi viene chiamato per portarlo allo scaffale assegnato. Il muletto si reca al dock, preleva il pacco, lo porta a destinazione e dopo averlo depositato rimane in attesa di altri ordini. In maniera analoga, quando un pacco deve uscire dal magazzino, il muletto si reca allo scaffale dove il pacco è stoccato, lo preleva e lo porta al dock d'uscita dove lo rilascia. In entrambi i casi, il muletto una volta finito il task controlla se ci sono altri task pendenti, si svolge una negoziazione tra i vari muletti "liberi" per decidere chi se ne incarica e se non ci sono altri task (o se comunque non ce n'erano fin dall'inizio) si preoccupa di non essere d'intralcio ad altri; ad esempio si possono prevedere delle aree di parcheggio alle quali un veicolo possa afferire in questa circostanza.

RICARICA Nel momento in cui definiamo delle aree di parcheggio diventa possibile anche introdurre un concetto di ricarica: in caso un muletto, dopo aver completato il suo task corrente, si accorga di avere una carica troppo bassa allora si porta nell'area di ricarica, in cui si parcheggia e nel contempo ricarica senza però controllare se ci sono task pendenti. Il muletto è di fatto non disponibile finché non ha completato la ricarica.

CADUTA PACCHI Durante il movimento dei pacchi può accadere, come già accennato, che il pacco trasportato cada a terra. Se ciò accade, questo viene notificato e si mette in moto il meccanismo di recupero. Tutta l'attività all'interno del magazzino viene interrotta per permettere il recupero del pacco da chi di dovere. Quando il pacco viene recuperato, viene portato al dock d'entrata dove sarà poi reindirizzato o verso l'uscita o verso lo scaffale.

GESTIONE COLLISIONI Può accadere spesso, che un muletto attraversi la strada di un altro, o che si debbano accodare. In queste situazioni, i due muletti se ne accorgono e tramite una politica di negoziazione che sarà poi spiegata nel dettaglio più avanti decidono come comportarsi, chi dovrà passare per primo o chi dovrà accodarsi a chi.

Il concetto di *pacco* viene qui utilizzato per definire un certo quantitativo di merce che deve essere trasportata, senza alcun riferimento alla dimensione o alla forma (supposta standard). Un sinonimo di pacco, usato anche in futuro sarà *pallet*.

Emerge chiaramente il fatto che gli scenari di *Ricarica*, *Caduta Pacchi*, e *Gestione Collisioni* non sono altro che derivazioni dal primo, cioè sono casi particolari che potrebbero verificarsi durante il normale esercizio del sistema (*Movimento Pacchi*) e che quindi si è tenuti a considereare.

2

ANALISI E PROGETTAZIONE ARCHITETTURALE

Il problema al centro del progetto è la realizzazione di un sistema in grado di prevedere ed evitare le collisioni fra veicoli, ma prima di poter arrivare a questo punto è necessario realizzare un sistema in cui un insieme di veicoli rischino di collidere.

Rifacendoci allo scenario precedentemente illustrato siamo in presenza di un *magazzino*, in cui un insieme di *muletti* (auto-guidati, AGV) viene impiegato per spostare dei *pallet*; in particolare questi ultimi possono essere stoccati, una volta arrivati al magazzino, oppure recuperati e spediti, così come persi durante il tragitto costringendo un *recovery officer* al recupero. Inoltre i muletti sono dotati di una carica che permette loro di muoversi e che li obbliga a rientrare periodicamente per ricaricarsi.

Dal momento che non disponiamo fisicamente di un magazzino il tutto dovrà essere simulato, ciò nonostante il problema risulta complesso, in quanto il sistema vive grazie all'interazione di un insieme non trascurabile di componenti. Tale complessità risulta più trattabile ragionando in termini di sistemi multi-agente e vedendo lo scenario come un insieme di entità attive, autonome e situate in un ambiente dal quale possono trarre le informazioni necessarie a raggiungere il proprio scopo.

Riassumiamo quindi i requisiti (nonché problemi da risolvere) che il progetto evidenzia:

MODELLO DEI MULETTI è necessario che i muletti espongano un'interfaccia che ne consenta la guida, così come è necessario progettare un'entità in grado di manovrarli.

PIANIFICAZIONE DEGLI SPOSTAMENTI La logica che si preoccuperà di guidare dovrà scegliere un metodo per orientarsi nello spazio a disposizione: basarsi su una mappa nota, costruirsi una mappa, muoversi in modo reattivo, ...

ASSEGNAMENTO DEI TRASFERIMENTI DI MERCE Avendo un insieme di trasferimenti da compiere ed un insieme di muletti in grado di svolgere tale compito, l'assegnamento dovrà essere gestito.

COORDINAMENTO FRA I VEICOLI Con l'obiettivo di portare a termine un task l'autista del veicolo non dovrà solo preoccuparsi di capire come raggiungere le destinazioni stabilite, ma dovrà anche assicurarsi di non andare a collidere con altri veicoli (così come con l'arredamento del magazzino o altro).

RECUPERO DEI PACCHI PERSI Il sistema prevede la possibilità di perdere pacchi durante il loro trasferimento, perciò è necessario definire contestualmente una politica che ne consenta il recupero permettendo la ripresa del normale svolgimento delle attività.

GESTIONE DELL'ENERGIA Come nella realtà i veicoli non possono funzionare indipendentemente da una qualsiasi fonte di energia. Tale fattore dovrà essere preso in considerazione, partendo dal presupposto che sia necessario effettuare periodicamente un rifornimento (ad esempio per ricaricare la batteria che alimenta l'AGV).

2.1 POSSIBILI SOLUZIONI AI PROBLEMI

Ciascuno dei problemi illustrati richiede una soluzione che concorra alla riduzione del rischio di collidere, per questo li analizzeremo uno per uno soppesando i pro e i contro di ciascuna opzione.

2.1.1 *Modello dei muletti*

Un muletto è un veicolo in grado di trasportare un insieme di pallet da un luogo ad un altro. Generalmente viene guidato da un autista in carne ed ossa, ma in alcuni casi (come in questo progetto) può essere guidato da un agente software, in quest'ultimo caso si parla di *Automatic Guided Vehicle* (AGV). Questa bivalenza, tipica dei sistemi socio-tecnici, richiede la definizione di un modello dei muletti, in particolare di una interfaccia che esponga una serie di funzionalità con esito predicibile, che permettano all'autista, detto anche *driver*, di guidare e caricare/scaricare i pallet (indipendentemente dalla sua natura).

Le funzioni che devono essere supportate sono:

CARICAMENTO/SCARICAMENTO PALLET un muletto deve essere in grado di recuperare un pallet (più o meno grande) da un rack o un dock così come deve essere in grado di depositarlo successivamente in un rack o ad un dock; l'ipotesi assunta è che i pallet siano entità inscindibili e trattate (stoccaggio e trasporto) singolarmente

ACCELERAZIONE l'autista dev'essere in grado di istruire il motore affinché aumenti la velocità di percorrenza

FRENATA l'autista dev'essere in grado di frenare, per poter approcciare alla destinazione ed evitare le collisioni

STERZO un organo di sterzo è indispensabile per indirizzare il veicolo nella corretta destinazione

Accelerazione, Frenata e Sterzo costituiscono l'insieme minimale di mosse necessarie per parlare di guida e sono operazioni *time-critical*, infatti se così non fosse sarebbe impossibile governare un veicolo.

Un modello che è possibile applicare è quello fisico ($\vec{F} = m \cdot \vec{a}, \dots$): il principale vantaggio portato da questa scelta è il realismo della simulazione; come controparte abbiamo una discreta complessità di guida. Data una traiettoria che si intende percorrere, l'autista dovrà calcolare la forza da applicare per ottenere lo spostamento desiderato, questo comportamento richiede una discreta reattività, come quando ogni giorno ciascuno di noi guida la propria automobile. Una corretta valutazione del modello ha bisogno che il tempo (logico) scorra regolarmente, indipendentemente dai tempi di reazione del guidatore; inoltre la presenza di un tempo globale donerebbe coerenza al modello evolutivo del sistema, al prezzo però di una centralizzazione della logica di simulazione, o dell'inserimento di una logica di coordinamento delle singole entità concorrenti.

Un approccio alternativo sarebbe quello di definire i muletti come *elementi collocabili* e demandare la logica di movimento a una differente entità al di fuori dei veicoli; il vantaggio ottenibile è sicuramente in termini di riduzione della complessità, fermo restando che ne risulterebbe una modellazione meno realistica.

2.1.2 *Pianificazione degli spostamenti*

Per poter guidare un muletto, l'entità *driver* dovrà orientarsi all'interno del magazzino.

Una prima soluzione potrebbe essere quella di scegliere una direzione di movimento e reagire ad ogni variazione dell'ambiente, senza una pianificazione iniziale. Vista l'alta inefficienza di questa scelta (in termini di job completati per unità di tempo) una possibile variante potrebbe essere quella di registrare informazioni sui tragitti scelti in passato, con lo scopo di costruire una mappa che, a regime, permetterà di rendere più efficienti gli spostamenti (avvalendosi di una fase di pianificazione).

Un'ulteriore soluzione è quella di definire *a priori* una mappa del magazzino, alla quale ogni autista possa accedere per pianificare il percorso da seguire. Questo approccio offre una serie di vantaggi:

1. Si elimina il transitorio durante il quale gli autisti sarebbero costretti a costruirsi la mappa, rendendo subito efficiente il sistema
2. La logica dell'entità *driver*, una volta prevista una fase di pianificazione che potrebbe anche essere offerta come servizio accessorio condiviso, risulta semplificata
3. Affidandosi alla mappa l'autista può avere la garanzia di dover gestire solo collisioni con altri veicoli o pacchi perduti, inoltre è possibile regolamentare la circolazione all'interno del magazzino
4. In caso di situazioni di emergenza un *supervisore* potrebbe modificare la viabilità del magazzino per evitare la zona in cui l'emergenza si è verificata[1]

L'utilizzo di una mappa porterebbe all'introduzione del concetto di traiettoria come direttrice del movimento dei muletti: in base al modello scelto essa rappresenterà il mezzo per determinare la forza da applicare al veicolo (quanto accelerare/frenare e quanto sterzare), oppure la successiva collocazione in relazione ad una velocità di percorrenza desiderata. Rimangono aperte quindi due questioni: *come modellare una eventuale mappa? come modellare una traiettoria?*

La risposta alla prima domanda può essere mutuata dall'ambito della *ricerca operativa* (e dell'intelligenza artificiale): un *grafo* può rappresentare correttamente una mappa. Un nodo per ciascuna destinazione raggiungibile e un arco orientato (o un lato) per ogni percorso che colleghi due destinazioni; inoltre, per semplificare la definizione di una traiettoria ammissibile (cioè che non collida con elementi statici del magazzino), può risultare conveniente l'introduzione di un set di nodi che permetta di adattare i percorsi alla topologia del magazzino. Partendo da questi presupposti è poi possibile sfruttare algoritmi noti (come quello Dijkstra) per realizzare la pianificazione iniziale.

Per quanto riguarda la modellazione delle traiettorie: una traiettoria non è altro che una generica curva che collega il punto di partenza al punto d'arrivo, alternando tratti rettilinei a curve sufficientemente dolci per essere percorse da un veicolo in movimento; allo scopo esistono diverse formulazioni possibili: Bézier, Spline, NURBS, ... Ciascuna delle formulazioni citate offre proprietà interessanti per il nostro scopo:

DESCRIZIONE DATA DAI PUNTI DI CONTROLLO la curva è descritta da una sequenza di punti di controllo (che possiamo ricavare dalla mappa), inoltre è possibile ottenere delle curve che raccordino gli spigoli quasi gratuitamente

1 Tale situazione non è stata considerata per il momento, rappresenta un possibile miglioramento futuro

LA CURVA E' RACCHIUSA NEL GUSCIO CONVESSO DEI PUNTI DI CONTROLLO la convex-hull dei punti di controllo determina la regione di massima in cui il veicolo potrà passare

SONO CURVE PARAMETRICHE una volta trovata una parametrizzazione adatta si ottiene la possibilità di trovare punti sulla traiettoria al variare di un parametro scalare (non ho bisogno di ragionare in due dimensioni)

2.1.3 *Assegnamento dei trasferimenti di merce*

Il business di un magazzino consiste nel ricevere delle merci e stoccarle in attesa che qualcuno le richieda. Per questo nel sistema è necessario prevedere un'entità (*manager*) che si preoccupi di gestire il flusso di merci in entrata ed uscita, scelga come disporre la merce e si preoccupi di informare gli autisti dei lavori da svolgere.

Il primo compito non è vitale ai fini del progetto che intendiamo realizzare, quindi verrà tralasciato, mentre l'assegnazione dei trasferimenti può influire direttamente sulla probabilità di collisione. Le possibili scelte sono:

DEMANDARE LA RESPONSABILTA' AL MANAGER Ogni volta che un nuovo *job* deve essere portato a compimento il manager valuta le disponibilità e prende una decisione in base a criteri basati su posizionamento dei veicoli, evoluzione dello stato del magazzino, ...

PERMETTERE AGLI AUTISTI DI COORDINARSI Il manager informa i driver che c'è un job da portare a termine, i singoli autisti si accordano su chi dovrà occuparsene, in relazione alla propria disponibilità

Elaborare un modello per la previsione delle conseguenze degli assegnamenti risulterebbe molto complesso a causa del numero di fattori che possono incidere sulla sua evoluzione: la posizione corrente dei veicoli, la velocità di movimento, la traiettoria per recuperare il pacco, quella per portarlo a destinazione, la previsione delle contingenze che potranno verificarsi e così via. Per tale ragione il secondo scenario (la coordinazione fra gli autisti) sembra essere più trattabile; d'altra parte la visione parziale degli autisti ci obbliga a scegliere una soluzione euristica rispetto ad una previsione esatta o comunque più precisa.

Un contributo che il manager può fornire è la definizione e calibrazione di una politica di stoccaggio[2] che ottimizzi gli spostamenti.

2.1.4 *Coordinamento fra i veicoli*

Come accennato precedentemente è necessario prevedere una forma di interazione fra i veicoli (o meglio fra gli autisti) e con l'ambiente circostante, per permettere loro di evitare collisioni.

Innanzitutto, man mano che procede, l'autista deve avere una percezione del mondo intorno a se e, sulla base di tale percezione, deve capire se può proseguire senza problemi oppure no; bisognerà quindi individuare un metodo per valutare il rischio di collisione e, conseguentemente, quali informazioni siano necessarie per svolgere la previsione. In un secondo tempo poi sarà necessario offrire una "tavolo di contrattazione" al quale possano

2 Intesa come la scelta di dove stoccare i pallet in ingresso

afferire gli autisti in situazione di pericolo, con lo scopo di concordare una soluzione alla contingenza.

Questo passaggio è ad elevata criticità, principalmente perché deve essere garantita la sua risoluzione entro tempi prestabiliti, al fine di garantire un margine sufficiente per reagire in tempo. Per questo motivo la politica da impiegare dovrà essere semplice quanto efficace ed il resto del sistema dovrà essere concepito al fine di semplificare il tutto. In particolare dovrà essere garantita una *emergency policy*, da applicare nel caso in cui la contrattazione non si concluda (ad esempio fermarsi), dopodiché sarà necessario cercare di individuare un numero finito di casistiche che possano verificarsi. Prendiamo in esame lo scenario in cui esista una viabilità interna al magazzino, cablata in una mappa condivisa utilizzabile per una pianificazione. Se si prevedono carreggiate a doppia corsia (una per senso di marcia) è semplice capire come il tipo di collisioni verificabili si riduca a due semplici casi: un incrocio ed un accodamento; il primo andrà risolto stabilendo un ordine di attraversamento del punto critico, mentre il secondo richiederà che il veicolo in coda adegui la propria velocità rispetto a quello che lo precede. Stabilire un ordine di attraversamento è una operazione che può risultare abbastanza semplice, inoltre, nell'ipotesi che i diversi contendenti abbiano la stessa visione del mondo, è possibile garantire un esito consensuale. Alcuni esempi di regole di ordinamento sono la precedenza a destra, la vicinanza al punto critico, l'inerzia del veicolo e così via.

2.1.5 *Recupero dei pacchi persi*

Per fronteggiare la contingenza della perdita dei pacchi (*pallet*) è necessario:

1. Informare il sistema della perdita di un pacco
2. Evitare che un veicolo vada a collidere con il pacco caduto
3. Provvedere al recupero del pacco (utilizzando gli *officers*) ed evitando di collidere con altri elementi
4. Ripristinare il normale comportamento dopo che la contingenza è stata risolta

Il primo punto non risulta essere molto complesso, infatti è sufficiente che l'entità che si accorge della perdita lo comunichi agli altri; i restanti punti richiedono invece maggiore attenzione. Il modo più semplice per evitare che i veicoli collidano con il pacco caduto è fare in modo che si fermino finché la contingenza non viene risolta; in alternativa, simulando il caso reale, un muletto potrebbe decidere di proseguire per la sua strada per quanto possibile, cioè fintanto che non incrocia il pallet perduto sulla propria strada, eventualità che potrebbe anche non verificarsi.

Il secondo scenario oltre che più realistico è quello che impatta meno sulle "prestazioni" del magazzino; il rovescio della medaglia è che la politica di collision avoidance dovrà essere in grado di gestire questa nuova eventualità così come dovrà verificare anche le collisioni con gli officers. Logicamente evitare oggetti in movimento è più difficile che schivare oggetti fermi. Un'ulteriore facilitazione potrebbe essere quella di prevedere dei percorsi specifici per gli addetti al recupero, con lo scopo di accelerare il recupero grazie alla riduzione delle possibilità di collisione.

Una volta risolta la situazione di emergenza i muletti potranno riprendere le loro attività, come se nulla fosse. Ovviamente il muletto che ha perso il pallet non potrà completare il suo lavoro, quindi dovrà preoccuparsi di trovare un nuovo job.

2.1.6 *Gestione dell'energia*

Se consideriamo dei muletti dotati di una fonte di energia (ad esempio una batteria elettrica) che ne consente il movimento dobbiamo valutare anche la necessità di rifornirsi.

Questa necessità si traduce in due requisiti:

1. Bisogna prevedere delle **aree di parcheggio/ricarica** alle quali un muletto può operare un rifornimento senza per questo essere d'intralcio al normale funzionamento del magazzino
2. Prima di accettare un lavoro *il muletto deve chiedersi se avrà l'energia sufficiente* per portarlo a termine. In caso contrario il muletto dovrà preoccuparsi di raggiungere l'area di ricarica per poi rendersi nuovamente disponibile.

Ovviamente dovrà essere definito anche il modo in cui l'energia verrà consumata durante gli spostamenti, presumibilmente in funzione degli spostamenti compiuti.

2.2 operiamo delle scelte

Per poter procedere nella progettazione è necessario compiere delle scelte, alla luce delle valutazioni presentate:

1. I muletti si avvarranno di un modello fisico e la simulazione sarà, almeno in prima istanza, centralizzata, con lo scopo di avere una evoluzione coerente e più deterministica del moto, semplificando le considerazioni sulle collisioni.
2. La pianificazione sarà basata su una mappa prefissata, dalla traiettoria sarà possibile estrarre le informazioni necessarie per pilotare il veicolo e per prevedere le collisioni.
3. La mappa prevederà carreggiate a doppia corsia, per ridurre il numero di casi di collisione e semplificarne la gestione.
4. L'assegnamento dei job verrà gestito attraverso un'asta fra gli autisti disponibili.

2.3 formalizzazione dell'analisi

Ora che abbiamo completato la panoramica sui requisiti cerchiamo di formalizzare l'analisi avvalendoci quindi di una metodologia che ci possa guidare nello sviluppo del progetto. Utilizzeremo *SODA* (Societies in Open Distributed Agent spaces) con lo scopo di giungere alla definizione di un insieme di *ruoli* e *risorse* che poi si tradurranno rispettivamente in *agenti* e *artefatti*; il tutto avverrà senza preoccuparsi dell'effettiva tecnologia con la quale il sistema verrà realizzato, questo aspetto sarà trattato successivamente.

2.3.1 *Ambiente della simulazione*

L'ambiente ha un ruolo fondamentale, infatti rappresenta lo spazio in cui i diversi agenti saranno situati e la fonte delle informazioni che permetterà loro di raggiungere il proprio *goal* (insieme all'interazione fra agenti).

Dai requisiti emergono i seguenti elementi, come costituenti l'*external-environment* con il quale il sistema interagirà:

1. Il sistema è situato in un *magazzino* (**warehouse**) all'interno del quale delle merci (organizzate in **pallet**) entrano ed escono da dei **dock** e vengono stoccate in un insieme di scaffali (**rack**) identificabili univocamente.
2. Le merci fluiscono grazie ad un insieme di muletti (**vehicle**) che *dovranno essere governati da un agente software* che si dovrà anche preoccupare di evitare collisioni con altre entità
3. Gli scaffali sono tipicamente organizzati per corsie, con un corridoio di accesso centrale
4. Il magazzino conterrà anche delle aree riservate al parcheggio e rifornimento dei veicoli
5. Un insieme di addetti (recovery **officer**) potranno essere impiegati per il recupero di pallet caduti durante il trasporto

Una ipotesi che è necessario introdurre è che ciascuna di queste entità sia osservabile, cioé che il sistema che costruiremo abbia la possibilità (attraverso un qualche *legacy-system*) di deteminare lo stato di un rack (pieno o vuoto), la presenza di pacchi da stoccare (recuperabili dal dock di ingresso), il posizionamento attuale dei veicoli e così via; in questo modo possiamo definire una serie di *funzioni* che potremo utilizzare:

PLANNING Dato un punto di partenza A, uno di arrivo B e una mappa trovare la strada da percorrere per raggiungere B da A

PALLET LOADING/UNLOADING Dato un pallet lo carica sul veicolo oppure lo scarica in un dock o un rack

DRIVING FUNCTIONS Le funzioni base per poter guidare un veicolo: accelerazione, frenata, sterzo (vedi Sezione 2.1.1)

RECHARGING Ricarica il muletto

PALLET LOST NOTIFICATION Notifica il sistema della perdita di un pallet

SIMULATION TUNING Calibra i parametri della simulazione

2.3.2 *Task*

Ora che abbiamo elencato una serie di funzionalità di cui possiamo usufruire, dobbiamo individuare quali e quanti task serviranno per coprire i requisiti.

Un primo elenco potrebbe essere il seguente:

HANDLE REQUEST Una particolare merce è stata richiesta da un cliente, oppure un fornitore ha portato della merce; implica un task di tipo "Stock/Retrieve pallet"

STOCK PALLET Un pallet deve essere recuperato dal dock in ingresso per poi essere portato ad uno scaffale (rack); implica un task di tipo "Reach destination"

RETRIEVE PALLET Un pallet deve essere recuperato da uno specifico scaffale per essere poi portato al dock di uscita; implica un task di tipo "Reach destination"

REACH DESTINATION Il veicolo deve essere guidato dalla posizione corrente fino alla destinazione desiderata; implica un task di tipo "Collision Avoidance"

COLLISION AVOIDANCE Durante il movimento il veicolo deve evitare di scontrarsi con altri elementi all'interno del magazzino (principalmente veicoli)

RECOVER PALLET Un pallet è stato perso e deve essere quindi recuperato per poter ripristinare il normale funzionamento del magazzino

LOSE PALLET Il pallet attualmente trasportato deve essere perso; l'unica utilità di questo task è per generare una perdita controllata dei pacchi ai fini della simulazione

Come è chiaro per poter svolgere un task quale "Reach destination" sarà necessario avvalersi delle funzionalità di *Planning* e *Driving*, infatti sarà necessario innanzitutto pianificare il raggiungimento della destinazione, per poi poter guidare il veicolo. Un task come "Stock/-Retrieve pallet", oltre ad includere "Reach destination" dovrà anche utilizzare le funzioni di *Pallet loading/unloading*. In base al meta-modello di SODA le singole funzionalità (o classi di funzionalità) saranno poi raggruppate in un insieme di risorse, quindi di *artefatti*.

I task elencati inoltre possono essere composti in *ruoli*, portando così alla definizione di un insieme di *agenti*, il cui *goal* sarà relativo ai task che sono in grado di svolgere:

MANAGER : Handle request

DRIVER : Stock pallet, Retrieve pallet, Collision Avoidance, Lose pallet

RECOVERY OFFICER : Recover pallet

Come si può notare l'agente *driver* qui proposto risulta essere quello più complesso. Per questo motivo possiamo scegliere di operare uno *zoom-in* e vedere quest'ultimo come la composizione di un insieme di sottoelementi.

2.3.3 *Il ruolo di Driver*

I compiti di un *driver* possono essere riorganizzati nel seguente modo:

TASK HANDLING deve interagire con l'entità *manager* per sapere quali job svolgere, informare della conclusione di quest'ultimo oppure dello smarrimento del pacco, pianificare il trasporto

NAVIGATION deve guidare il veicolo lungo la traiettoria prestabilita

COLLISION DETECTION AND AVOIDANCE deve prevedere il verificarsi di una collisione, negoziarne la risoluzione e far si che la strategia elaborata venga applicata

Ciascuno di questi compiti può essere assegnato ad una sotto entità, ad esempio il secondo compito (*navigation*) può essere realizzato attraverso un artefatto, infatti la sua logica è prettamente reattiva: data una traiettoria da seguire ed una velocità da mantenere è possibile muovere un veicolo in modo quasi automatico; nel momento in cui sorgesse un pericolo basterebbe aggiornare i dati di navigazione per reagire correttamente. I compiti restanti invece hanno una connotazione proattiva. Giungiamo quindi alla conclusione che il ruolo di driver può essere visto come una *società di agenti*.

Una ulteriore osservazione che possiamo effettuare riguarda la natura del comportamento di questa società di agenti. Se pensiamo a come una persona si comporta durante la guida ci rendiamo conto che esiste il nostro agire dipende da un insieme di priorità che ci prefiggiamo: esiste un comportamento ordinario e un insieme di eventualità (straordinarie) alle quali reagiamo prontamente con lo scopo di ritornare alla normalità; in questo senso possiamo dire che abbiamo un insieme di obiettivi più o meno prioritari e che le nostre azioni sono determinate dal desiderio di raggiungere l'obiettivo con maggiore priorità. Questo modo di agire è tipico del modello BDI e del *practical reasoning*.

2.3.4 *Applicazione del modello BDI*

Applicare un modello BDI significa avvalersi di un insieme di strutture dati: *Belief*, *Desire*, *Intent*. Un belief è una consapevolezza che l'agente ottiene principalmente attraverso la percezione dell'ambiente circostante e che andrà a formare una *knowledge base*, cioè l'insieme di conoscenze che l'agente stesso ha accumulato fino a quell'istante. Un desire invece rappresenta un obiettivo da raggiungere: a run-time i desire potranno tradursi in intention, cioè nella decisione di perpetrare un insieme di azioni, detto *piano*, con lo scopo di raggiungere uno specifico goal (il desiderio selezionato); il fatto che i desideri dell'agente si traducano in azioni è alla base di quello che viene definito *practical reasoning*. Il meccanismo grazie al quale viene determinata una intention pone le sue fondamenta sulla esistenza di un piano che compatibilmente con lo stato attuale dell'ambiente (codificato nella knowledge base) sia in grado di ottenere lo scopo; il numero di intention gestibili ad ogni ciclo di reasoning dipende dall'architettura utilizzata.

Tornando al progetto un *driver* avrà i seguenti desideri:

1. Svolgere un particolare lavoro (job), quindi far fluire merce
2. Evitare collisioni
3. Assicurarsi di avere sufficiente energia per proseguire con il proprio lavoro
4. Non essere di intralcio agli altri veicoli
5. Favorire il recupero di un pallet disperso, per risolvere il prima possibile l'anomalia

E' ovvio che il goal primario sarà il *far fluire la merce* (1), ma i restanti obiettivi sono tutti funzionali al fornire le condizioni per potare a termine tale goal. Possiamo dire che questi ultimi rappresentino l'insieme di contingenze a cui un autista (quindi il sistema) dovrà tempestivamente far fronte altrimenti non sarà nelle condizioni di proseguire; per questo saranno desideri più prioritari rispetto a (1).

3 SCELTE TECNOLOGICHE

Arrivati a questo punto siamo in grado di passare alla realizzazione del sistema. In questa fase ci porremo la problematica di quale tecnologia scegliere a supporto delle valutazioni svolte nelle sezioni precedenti: sarà infatti importante determinare la piattaforma ottimale, in grado di offrire l'astrazione più adatta per rappresentare i concetti fino ad ora discussi e nel contempo garantire le prestazioni richieste; come abbiamo infatti accennato nella sezioni 2.1.1 e 2.1.4 alcune funzionalità richiedono delle tempistiche specifiche al fine di garantire governabilità ed efficacia.

Le principali tecnologie candidate[1] sono:

JASON + CARTAGO (JACA) La piattaforma `Jason` offre una consolidata implementazione del modello BDI e del practical reasoning [4], che come abbiamo visto rappresenta una valida astrazione per il nostro sistema; l'integrazione di `CArtAgO` ha lo scopo di permettere l'interazione degli agenti `Jason` (specificati in `AgentSpeak`) con degli artefatti (specificati in Java), attraverso un meccanismo di estensione dell'insieme di azioni che l'agente è in grado di svolgere, appunto avvalendosi dei suddetti artefatti [1].

TUCSON La piattaforma `TuCSoN` invece concepisce un insieme di agenti (specificati in Java) interagenti attraverso uno o più *coordination media* (detti anche spazi), inoltre unito a ReSpecT offre la possibilità di cablare all'interno di ciascuno spazio un insieme di *coordination law* in grado di regolare l'aspetto interattivo fra i diversi agenti

Nel seguito approfondiremo singolarmente le tecnologie, valutando i pro e i contro e giungendo infine ad una scelta.

3.1 JASON + CARTAGO (JACA)

L'abbinamento di `Jason` e `CArtAgO` rappresenta la scelta più intuitiva perché propone l'astrazione di *agente intenzionale* [4], in grado di interagire con altri agenti e con artefatti. Questo fa si che esista una corrispondenza diretta fra le entità definite durante l'analisi e quelle che poi dovremo implementare, inoltre il minimo *gap* fra i concetti offerti e quelli desiderati facilita la realizzazione del sistema. Questo indubbio vantaggio porta però con se una controindicazione importante: l'infrastruttura fornita potrebbe essere più complessa del necessario e generare un overhead incompatibile con i requisiti illustrati.

Per poter fare delle valutazioni tangibili è necessario costruire un *primo prototipo*, con l'obiettivo di implementare alcune funzionalità base e verificare i *timings* del sistema: per questo motivo le funzioni da realizzare saranno (*i*) rendering e (*ii*) guida dei muletti (le operazioni più delicate), in questo modo sarà possibile sperimentare diverse implementazioni e determinare la soluzione migliore, in grado di garantire un margine sufficiente per la procedura di previsione e risoluzione delle collisioni.

1 Con riferimento alle tecnologie sperimentate durante il corso [3]

Il risultato di questi primi test non è stato incoraggiante. Infatti sono state provate diverse modalità di guida, con l'obiettivo di trovare il metodo più performante ed in linea con le astrazioni; ciò nonostante è stata riscontrata una grande variabilità dei tempi di esecuzione (*jitter*) che, in conclusione, rende la piattaforma inadatta per i nostri scopi; in quanto l'overhead generato dall'infrastruttura (a nostro avviso causa del jitter) non è controllabile.

3.2 TUCSON (+ RESPECT)

Il vantaggio che l'utilizzo di `TuCSoN` può offrire, rispetto ai problemi evidenziati con `JaCA`, è una infrastruttura più leggera legata essenzialmente a due motivi:

1. Il concetto di agente offerto da questa piattaforma è molto semplice: *any autonomous software component embedding its own control flow(s) can be considered an agent* [2]

2. La piattaforma è intrinsecamente distribuita (a differenza di Jason) quindi il carico risulta ripartibile; lo spazio (programmabile) che abilita l'interazione fra i diversi agenti è un processo indipendente accessibile attraverso TCP nel quale ciascuna entità può depositare, leggere e prelevare informazioni sottoforma di *tuple*.

Come controparte del punto (1) nel momento in cui ritenessimo opportuno impiegare il modello BDI, invece di elaborare una possibile alternativa, saremmo costretti ad implementarlo. Va comunque detto che in questo modo è possibile estrarre realizzarne una versione *ad-hoc* e quindi più performante.

Un altro aspetto interessante è la possibilità di *programmare lo spazio*. Infatti questo permette di demandare al nodo su cui risiede lo spazio parte dello sforzo computazionale e rendere gli agenti più "responsivi"; un esempio potrebbe essere il mantenimento dello stato attuale del sistema derivato dall'aggregazione delle informazioni che man mano si rendono disponibili, con lo scopo di fornire all'agente un quadro coerente. Inoltre questo meccanismo consente di definire il comportamento degli agenti astraendo da alcuni dettagli implementativi e favorendo una maggiore modularità; ad esempio potrebbe essere la chiave per far si che le informazioni siano veicolate alle entità interessate senza un indirizzamento esplicito, così come il modo per recuperare alcuni elementi del modello BDI.

Le sperimentazioni condotte hanno portato a concludere che l'utilizzo di `TuCSoN` + `ReSpecT` offriva delle prestazioni compatibili con i requisiti, senza per questo complicare eccessivamente la realizzazione. E' comunque opportuno evidenziare che per garantire la *predicibilità* delle funzioni di guida è stato necessario rivedere leggermente il modello dei muletti, offrendo un controllo più puntuale e preciso del veicolo.

4 PROGETTAZIONE ED IMPLEMENTAZIONE

La prima cosa che si è scelta di modellare è stato il magazzino,la scelta architetturale è stata quella di modellarlo come un grafo orientato completamente connesso, in modo che dati due nodi fosse sempre possibile costruire un cammino fra questi. I nodi dal canto loro, sono stati piazzati in corrispondenza dei rack, dell'entrata delle corsie laterali, ai dock di ingresso e di uscita e alle zone di parcheggio.

Il cammino, che sarà composto dai nodi e dagli archi che li uniscono, corrisponde al percorso logico all'interno del magazzino. Per fare si che i muletti riescano a muoversi nello spazio è necessaria una mappa fisica, così che ogni arco sia stato trasformato in una curva di Bezier di secondo grado.

L'insieme delle curve va a definire la traiettoria che unisce fisicamente due punti nel magazzino. Come tecnologia È stato scelto di utilizzare `TuCSoN`, perché, come già sottolinato, la possibilità di programmare i centri di tuple ha portato notevoli vantaggi.

Il sistema è composto da un *Manager* che ha il compito di generare lavori da svolgere (flusso della merce), uno o più *Driver* (definiti come una società di agenti) che ha il compito di negoziare lo svolgimento di un lavoro e portarlo a termine, degli *Officer* che intervengono quando all'interno del magazzino si verificano delle situazioni di pericolo (pallet perso da un muletto) per ripristinare l'ambiente ideale per il corretto funzionamento e infine un artefatto *MapArtifact*, che fornisce la funzionalità per generare percorsi che uniscano due punti all'interno del magazzino.

4.1 AGENTI E BDI

Il macro comportamento degli agenti del sistema è definito come un automa a stati finiti, per fare ciò, è stato esteso il modello di agente offerto da `TuCSoN`, "Automaton".

Il modello di agente è stato definito come un automa in cui, a seconda dello stato e delle percezioni dell'ambiente, il comportamento può essere più o meno intelligente. Per questo motivo è nata la necessità di avere degli agenti *strong* che avessero già *built-in* il modello *Belief Desire Intention*.

Per permettere all'agente di avere delle percezioni sull'ambiente in ogni instante, gli sono stati affancati uno o più [1] *Sensor* che hanno il compito di fare, appunto, del *sensing* sull'environment ad esso associato. Ogni qualvolta questi riscontrano un cambiamento notificano all'agente il nuovo stato dell'ambiente. Per poter svolgere il loro lavoro i Sensor effettuano un operazione di *in* sul centro di tuple con un dato *template* (*in(percept(myID,Event)*).

La *knowledge base* dell'agente quindi si popola di *Belief* ogni qualvolta un Sensor notifica che nell'ambiente è avvenuto un certo *Event*. In questo modo l'agente senza nessuna operazione esplicita si crea comunque la propria base di conoscenza. A seconda del comportamento che un agente vuole adottare in un determinato stato può definire i propri *Desires*, ognuno con una

1 Un agente può essere *situated* in più di un ambiente

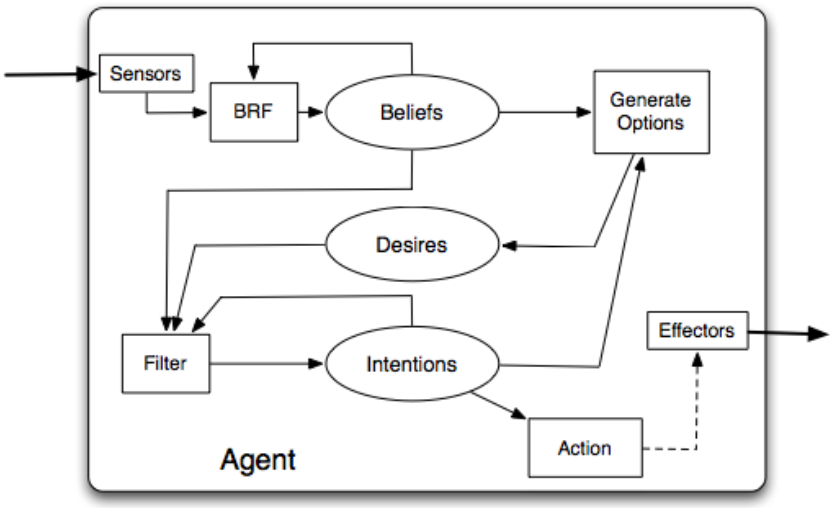

Figura 1: Russel and Norvig

propria priorità, così che quando questi troveranno riscontro nella knowladge base diventino *Intention* che saranno poi eseguite dall'agente.

4.1.1 *Manager*

Il Manager ha il compito di generare lavori da svolgere, ha un comportamento semplice (*weak agent*) che viene definito come un automa composto da tre stati: *waitForJob*, *checkJob* e *publishJob*. L'agente rimane nello stato di waitForJob fino a quanto non riceve un nuovo job da assegnare[2]. Ricevuto il possibile job il manager transita nello stato checkJob dove controlla la fattibilità rispetto all'ambiente. Nel caso il lavoro da svolgere non sia soddisfacibile l'agente ritornerà in waitForJob mentre in caso contrario transiterà nello stato successivo. Quando l'agente si trova in publishJob va a notificare all'ambiente la presenza del nuovo job da svolgere, fatto ciò ritorna in waitForJob.

4.1.2 *Driver*

Il Driver è un'entità che si occupa di più task, deve interagire con altri driver per negoziare lo svolgimento di un lavoro, successivamente deve pianificare il percorso ottimo per soddisfare il job (se lo ottiene) e infine guidare il muletto evitando possibili collisioni. Avendo così tante e non semplici operazioni da svolgere il driver è stato modellato come una società di agenti e artefatti composta da :

1. TaskHandler: agente che negozia e decide quale *task* svolgere;

2 Essendo questo un prototipo il nuovo job viene generato da una GUI

2. CollisionDetector: agente che negozia e risolve le possibili collisioni;

3. NavigatorArtifact: è l'artefatto che fa da interfaccia al veicolo.

Gli attori della società comunicano tra loro tramite un environment interno (nodo Tucson) chiamato d'ora in avanti *Brain*.

TaskHandler

Il TaskHandler è l'agente più importante all'intero della società in quanto è colui che decide quale sia il *task* (interno) da svolgere.

Il macro comportamento si può riassumere in due stati: *DecideTask* e *Moving*, infatti un muletto può essere solo fermo oppure in movimento. Sia che si trovi in un stato o nell'altro il TaskHandler deve sempre monitorare lo stato dell'ambiente per reagire in modo consono a seconda del task che sta svolgendo.

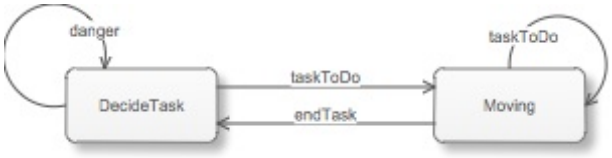

Figura 2: TaskHandler FSA

DECIDETASK Quando l'agente si trova in questo stato, come si intuisce dal nome, deve scegliere qual è il *task* da svolgere; le situazioni in cui si può trovare al momento della decisione sono essenzialmente tre:

1. se lo stato di carica del veicolo è sotto una certa soglia l'agente deve pianificare il percorso migliore per arrivare ad una zona di ricarica;

2. se il muletto non ha compiti da svolgere e si trova nel mezzo del magazzino, onde evitare che intralci il lavoro di altri driver, pianifica il percorso che gli consente di andare in una zona di parcheggio.

3. se il muletto ha batteria sufficiente, definisce il suo desiderio di compiere un lavoro e attende che questo si trasformi in intenzione. Quando l'intenzione si è verificata, l'agente inizierà una negoziazione con altri TaskHandler per l'assegnazione del lavoro, in caso di sconfitta il suo desiderio rimarrà immutato e l'agente ritornerà in attesa, in caso contrario sarà definito un nuovo task che ha per oggetto il lavoro negoziato.

Quando viene generato uno dei tre *task* il taskHandler transita nello stato di *Moving*.

MOVING In questo stato il muletto è realmente in movimento, quindi il TaskHandler modificherà lo stato del Brain con la sua intenzione di fare muovere il muletto. A seconda del task che sta svolgendo, l'agente setta i propri desideri in modo da reagire a determinati cambiamenti dell'environment.

Quando il task corrente è terminato e se non sono stati generati altri task, l'agente transiterà in DecideTask, altrimenti rimarrà nello stato Moving.

TASK Il Task definisce il compito che il TaskHandler svolge quando si trova nello stato Moving. È formato da una lista di percorsi che il muletto deve compiere per portare a termine un compito. A seconda del tipo di task che sta svolgendo l'agente ha determinati desideri che determineranno delle reazioni ai cambiamenti dell'ambiente.
Primi di elencare i task che il TaskHandler può svolgere, sono sotto riportati i desideri comuni a tutti i task:

1. *DANGER*, l'agente deve essere in grado di percepire una situazione di pericolo che si è venuta a creare nell'ambiente (perdita di un pallet), per bloccare prontamente il moto del veicolo.

2. *PASS ON NODE*, l'agente percepisce questo evento dal Brain tutte le volte che il muletto passa su un nodo noto del grafo.

3. *TARGET REACHED*, il TaskHandler percepisce questo evento quando il muletto ha terminato il suo cammino.

4. *FREE WAY*, l'agente percepisce dall'ambiente del sistema che la situazione di pericolo è rientrata e quindi può lasciare nuovamente il muletto libero di muoversi.

I possibili task del sistema sono:

1. *Work* questo task viene generato quando l'agente vince la negoziazione per l'assegnazione di un lavoro, la quale consiste nel caricare un pallet nel punto A e scaricarcarlo in B. Il compito prevede essenzialmente di transitare lungo due percorsi, uno che permette al muletto di raggiungere il luogo in cui si trova il pallet da caricare e l'altro che definisce il percorso da fare per poi scaricare il carico. Definiti i *path* l'agente modifica lo stato del Brain con il percorso che si deve compiere e attende che il desiderio *TARGET REACHED* diventi intenzione. Quando questo avviene, se il percorso era il primo dei due, da al muletto il comando di caricare il pacco, altrimenti dopo aver scaricato il pallet considererà completato il task.

   Oltre ai desideri comuni, a scopo di simulazione del sistema, è stato inserito un altro desiderio di perdita pacco (*LOSE PALLET*) per fare in modo che il muletto reagisca ad un comando esterno che gli "imponga" in qualche modo di perdere il pacco che sta trasportando. Quando il TaskHandler compie questa intenzione esso lascia cadere il pallet e blocca il moto del muletto, inserendo prontamente nell'ambiente la situazione di pericolo.

   Il TaskHandler non darà comandi di ripresa del moto al proprio muletto fino a quando non si concretizzerà l'intenzione *FREE WAY*. Quando l'agente compie questa intenzione avendo fallito il task che stava compiendo ripianifica un nuovo task per guidare il muletto ad un area di parcheggio.

2. *Park* questo task viene generato dal TaskHandler quando il muletto, privo di compiti da svolgere, si trova in un luogo del magazzino che non sia il parcheggio. Con questo compito il TaskHandler pianifica il percorso migliore che il Driver deve compiere per portare il proprio veicolo nella zona di parcheggio.

   L'agente, oltre ai desideri comuni, quando non ha compiti da svolgere ha il desiderio *MOVE PALLET*. Quando nella propria knowledge base viene aggiunto un belief che

consiste nello spostamento di un pallet, l'agente come succede nello stato DecideTask, negozierà il "job" con altri TaskHandler e nel caso gli venga assegnato lo spostamento del pallet, notificherà nel Brain la cessazione del percorso corrente, attendendo poi che il desiderio *TARGET REACHED* si avveri così da ripianificare il nuovo task Work.

3. *Recharge* quest'ultimo task viene creato ed eseguito quando il TaskHandler si accorge che la carica del proprio muletto è in riserva oppure quando, in fase di negoziazione, l'agente ha dovuto ritirarsi a causa dell'autonomia troppo bassa del veicolo, non sufficiente per compiere il job.
   L'agente con questo task pianifica e informa il Brain del path per raggiungere la zona di carica più vicina. Una volta giunto a destinazione (quindi il desiderio *TARGET REACHED* èè esaudito) il muletto inizia la fase di ricarica. Il task si considera concluso quando la carica del veicolo ritorna massima.

NavigatorArtifact

Il NavigatorArtifact è un artefatto che interagisce solo con il brain ed è definito anch'esso come un automa formato da due stati: *Idle* e *Ready*. Nello stato di idle l'artefatto attende che nell'ambiente appaia il grafo che rappresenta percorso logico che il muletto deve compiere. Quando appare transita nello stato Ready. Nello stato Ready l'artefatto è "reattivo" ai seguenti comandi:

1. *throttle*: questo comando implica il movimento del muletto;
2. *brake*: con questo comando l'artefatto da il comando di frenata al veicolo.
3. *abort*: quando l'artefatto riceve questo comando significa che deve tagliare la traiettoria che sta percorrendo e fermarsi nel primo punto utile, che corrisponde ad un nodo del grafo del magazzino.

Oltre ad essere reattivo ai comandi sopra elencati, l'artefatto fa una notifica nell'ambiente tutte le volte che che il muletto passa sopra un nodo del grafo, per fare in modo nell'ambiente ci sia traccia del "cammino logico" del muletto.

CollisionDetector

Il CollisionDetector, come si evince dal nome, ha il compito di evitare gli scontri; anch'esso è un automa formato da due soli stati: *Idle* e *Moving*. Questo agente lavora in sinergia con il NavigatorArtifact, infatti una volta accortosi che l'artefatto è pronto a guidare il muletto transita dallo stato Idle a Moving.

In questo stato l'agente prima di dare il comando di accelerazione al NavigatorArtifact setta i proprio desideri in ordine ascendente secondo la loro importanza:

1. *END*: questo desiderio diventa intenzione quando lo stato del Brain cambia come conseguenza della notifica di percorso completato fatta dall'artefatto.
2. *CURRENT CURVES*: questo diventa intenzione tutte le volte che il NavigatorArtifact notifica nell'ambiente che il muletto è transitato su un nodo del grafo. Quando questo avviene il CollisionDetector si crea l'equazione della retta che passa per il punto del

grafo appena passato e il successivo, cambiando poi lo stato dell'ambiente del sistema con il proprio dato.

3. *COLLISION*: questo desiderio si trasforma in intenzione quando il significato del cambiamento di stato dell'ambiente consiste in una collisione tra il muletto pilotato dal Driver a cui appartiene l'agente e un altro. Quando si verifica ciò il CollisionDetector apre un tavolo di trattativa direttamente con l'altro agente coinvolto. La collisione può derivare dall'incrocio di traiettorie (*cross*), oppure sovrapposizione delle traiettorie parziali (*back*). Se si tratta di una collisione *cross* il CollisionDetector il cui veicolo si trova più vicino al punto di collisione avrà il diritto di passare mentre l'altro sarà costretto a fermarsi. Se invece si tratta di una collisione *back* il muletto che si trova dietro frenerà.

4. *FREE WAY*: questo desiderio, diverso da quello del TaskHandler, diventa intenzione quando nell'environment si verifica l'evento derivante dalla risoluzione di una precedente collisione.

La gestione delle collisioni viene interamente lasciata all'ambiente, il CollisionDetector si limita solamente a negoziare per scoprire se ha la precedenza. I muletti usciti vincitori dalle negoziazione hanno poi il compito di dare il via libera agli sconfitti. Tutte le volte che si verifica l'intenzione *CURRENT CURVES* l'agente notifica nell'ambiente che la collisione è risolta così che i muletti che hanno perso la negoziazione possano riprendere il proprio moto.

4.1.3 *Officer*

Nel sistema sono presenti due agenti Officer che hanno il compito di recuperare i pallet persi dai Driver. Questi hanno un unico desiderio: *DANGER*. Una volta che il desiderio diventa intenzione i due Officer si accordano su chi dei due risolverà il problema. Il vincitore della negoziazione andrà a recuperare il pallet perso e lo riporterà nel dock di ingresso.

4.2 PROGRAMMAZIONE DEI NODI TUCSON

Come anticipato in precedenza la programmazione dei nodi Tucson porta notevoli vantaggi, più importante fra tutti la possibilità di definire *strong agent* grazie al modello *BDI*.

4.2.1 *Nodo environment*

Inserimento e assegnazione job

La prima "feature" programmata nell'ambiente riguarda l'inserimento di nuovi lavori da svolgere (movimento pallet) da parte del Manager. L'idea è che il Manager generi un semplice evento nell'ambiente, sarà poi quest'ultimo a tenere traccia dei lavori da compiere e attivare il processo di negoziazione fra i vari Driver disponibili.

```
\% Manager agent inserts new jobs in the tuple centre:
\% each one is enqueued in an ordered task-list, whose head will be next job to

reaction(
```

```
  out(job(ID,P,F,T)),
  from_agent,
  (
    in(job(ID,P,F,T)),
    out(newTask(task(ID,P,F,T)))
  )
).

reaction(
  out(newTask(T)),
  from_tc,
  (
    in(newTask(T)),
    (
      no(taskCache(L)),
      L1 = [T]
      ;
      in(taskCache(L)),
      append(L,[T],L1)
    ),
    out(taskCache(L1)),
    (
      no(barrier(_)),
      no(danger(_)),
      out(barrier(0))
      ;
      true
    )
  )
).

\% Creates an agent barrier, in order to synchronize taskhandlers for job assignm
\% The barrier is initialized with available vehicles count value (if any).

reaction(
  out(barrier(N)),
  from_tc,
  (
    (
      in(available(X)),
      rd(taskCache([Task|L])),
      out(th_percept(X,movePallet(Task))),
      in(barrier(N)),
      N1 is N + 1,
      out(barrier(N1))
```

```
      ;
      (
        rd(barrier(0)),
        in(barrier(0))
        ;
        in(taskCache([Task|L])),
        out(taskCache(L))
      )
    )
  )
).
```

Come si legge nella prima *reaction*, un agente (il Manager) inserisce nel centro una tupla $job(...)$ che fa scattare una reazione che va ad inserire nella lista di lavori da svolgere il job. Dopo averlo inserito, se non sono in corso delle negoziazioni ($no(barrier(\_))$) e non sono presenti situazioni di pericolo ($no(danger(\_))$), viene inserita nel centro di tuple una barriera. Questa barriera "triggera" un'altra reazione che va a contare tutti i Driver che si sono dichiarati disponibili, e per ognuno genera una tupla $out(th\_percept(X, movePallet(Task)))$. Questa tupla sarà perciò percepita dal TaskHandler interessato diventando così un belief, che se associato ad un desiderio si trasformerà nell'intenzione che gli farà iniziare la negoziazione.

Disponibilità Driver

```
\% When a driver becomes available, it notifies the tuple centre (sending a tuple
\% on negative outcome the sender agent falls asleep, otherwise the AVAILABLE tup
\% which contains agent ID, is inserted.

reaction(
  inp(becomeAvail(ID)),
  request,
  (
    no(barrier(N)),
    out(available(ID)),
    out(becomeAvail(ID))
   )
).

reaction(
  out(available(X)),
  from_tc,
  (
    no(danger(_)),
    rdp(taskCache(L)),
    L \= [],
```

```
    out(barrier(0))
  )
).
```

Il codice mostra il processo con il quale un Driver (più precisamente un TaskHandler), si dichiara disponibile. I TaskHandler però non possono dichiararsi *available* in ogni momento. Per questo motivo l'agente dovrà fare una richiesta sul centro, il quale cambierà il suo stato rispondendo alla richiesta. Questo però accade solo se nel dato momento non è in corso una negoziazione, altrimenti l'ambiente non farà nessun cambiamento facendo così percepire all'agente l'impossibilità di dichiararsi disponibile.

Negoziazione job

Quando nei vari TaskHandler si concretizza l'intenzione *MOVE PALLET*, se la carica rimanente del muletto è sufficiente, essi calcolano in quando tempo riusciranno a compiere il lavoro e fanno la loro "puntata" nell'ambiente. Il centro di tuple dopo aver ricevuto la richiesta di puntata ($bet(ID, Time)$) può adottare uno dei seguenti comportamenti:

1. nel qual caso la puntata sia la prima viene generata nello spazio una tupla ($winning(ID, T)$) che mantiene l'attuale vincitore e il proponente si mette in attesa di ricevere l'esito della negoziazione.
2. se è già presente nell'ambiente un temporaneo vincitore, viene confrontata la corrente offerta con quella proposta dall'agente:
   a) se la proposta è migliore della precedente, viene cambiata la tupla che contiene l'attuale vincitore, viene accettata la richiesta del proponente e viene notificato al vincitore uscente di aver perso la competizione;
   b) se la proposta è peggiore, non viene accetta la richiesta cosicché il proponente si accorga che la propria offerta non è stata sufficiente.

```
reaction(
  inp(bet(ID,Time)),
  request,
  (
    rd(winning(ID1,T)),
    (
      T > Time,
      in(winning(ID1,T)),
      out(winning(ID,Time)),
      out(winner(ID1,'false')),
      out(bet(ID,Time))
      ;
      true
    ),
    out(less_one(ID))
```

```
    ;
    out(winning(ID,Time)),
    out(bet(ID,Time)),
    out(less_one(ID))
  )
).
```

Tutte le volte che nello stato viene fatta una richiesta viene decrementato lo stato della barriera precedentemente calcolata. Quando questa raggiunge lo zero, significa tutti gli agenti disponibili hanno fatto la loro puntata e viene data notifica di vittoria al vincitore.

```
reaction(
  out(less_one(X)),
  from_tc,
  (
    in(less_one(X)),
    in(barrier(N)),
    (
      N = 1,
      in(winning(ID,T)),
      out(winner(ID,'true'))
      ;
      N1 is N - 1,
      out(barrier(N1))
    )
  )
).
```

Perdita pallet

La prima reaction è stata inserita solo ai fini della simulazione di questo scenario di perdita pallet da parte di un muletto, infatti all'inserimento della tupla $lost\_pallet(ID)$ ne viene generata un altra, fedele al modello BDI degli agenti, per fare in modo che il Driver del veicolo ID perda il proprio carico.

```
reaction(
  out(lost_pallet(ID)),
  from_agent,
  (
    in(lost_pallet(ID)),
    current_time(CurrentT),
    out(th_percept(ID,lost_pallet(CurrentT)))
  )
).
```

Quando un Driver perde il proprio pallet inserisce nell'ambiente la tupla $danger(D)$, questo fa scattare una reazione che va a generare due eventi che saranno poi percepiti dagli officer. Oltre a questi il centro di tuple va a recuperare la lista dei Driver che si stanno muovendo all'interno del magazzino[3] e fa scattare la reazione $notifyPL(L)$ che genera le percezioni di *DANGER* che faranno poi "scattare" l'intenzione di fermarsi dei TaskHandler.

```
reaction(
  out(danger(D)),
  from_agent,
  (
    in(danger(D)),
    out(off_percept('off1',D)),
    out(off_percept('off2',D)),
    out(danger(D)),
    rd(moving(L)),
    out(notifyPL(L))
  )
).

reaction(
  out(notifyPL(L)),
  from_tc,
  (
    in(notifyPL(L)),
    (
      L \= [],
      L = [ID|T],
      current_time(Time),
      out(th_percept(ID,danger(Time))) ,
      out(notifyPL(T))
      ;
      true
    )
  )
).

reaction(
  out(move(ID)),
  from_agent,
  (
    in(move(ID)),
```

3 Tutte le volte che il TaskHandler transita nello stato di Moving lo notifica ($out(move(ID))$) all'environment il quale va a compilare una lista che contiene tutti gli ID dei muletti in movimento. Quando l'agente si prepara a fare una transizione dallo stato Moving notifica all'ambiente il termite del proprio moto.

```
    (
      in(moving(L)),
      out(moving([ID|L]))
      ;
      out(moving([ID]))
    ),
    (
      no(danger(D)),
      true
      ;
      out(th_percept(ID,danger(Time)))
    )
  )
).

reaction(
  out(endMove(ID)),
  from_agent,
  (
    in(endMove(ID)),
    in(moving(L)),
    remove(ID,L,Lo),
    out(moving(Lo))
  )
).
```

Quando l'officer che si è preso in carico di portare in sicurezza il magazzino ha terminato il suo compito, rimuove dal centro di tuple lo stato $idanger(\_)$. Questo genera una reazione, la quale andrà a generare a sua volta una nuova percezione nei TaskHandler di *FREE WAY*.

```
reaction(
  in(danger(_)),
  from_agent,
  (
    rd(moving(L)),
    out(notifyFW(L)),
    (
      rdp(available(X)),
      rdp(taskCache(L)),
      L \= [],
      out(barrier(0))
      ;
      true
    )
  )
```

```
).

reaction(
  out(notifyFW(L)),
  from_tc,
  (
    in(notifyFW(L)),
    (
      L \= [],
      L = [ID|T],
      current_time(Time),
      out(th_percept(ID,free_way(Time))) ,
      out(notifyFW(T))
      ;
      true
    )
  )
).
```

Gestione collisioni

Per quel che riguarda le collisioni è stato deciso di lasciare il compito di individuare le collisioni interamente al centro di tuple. Infatti, come già spiegato quando è stato trattato il CollisionDetector, ogni qualvolta il veicolo passa su un nodo del grafo del magazzino viene inserita nell'ambiente l'equazione della retta passante per il nodo appena etichettato e il nodo successivo. L'inserimento del $path(ID, P, F)$, (formato dall'ID dell'agente, dal punto in cui si trova l'agente al momento della segnalazione e dalla funzione implicita della retta) fa scattare una reazione che va ad inserire il cammino nella lista dei cammini, cancellando da essa un cammino che ha lo stesso id. Dopo averlo inserito, il centro fa partire una computazione *checkPC* che restituisce la lista formata dalla strutture $pcol(ID, ID1, P)$ che servono ad indicare il fatto che i muletti associati a ID e ID1 sono prossimi ad una collisione nel punto P.

```
reaction(
  out(path(ID,P,F)),
  from_agent,
  (
   in(path(ID,P,F)),
   (
     in(traj(L)),
     remove(path(ID,P,F),L,List) ,
     Ltmp = List,
     checkPC(path(ID,P,F),Ltmp,COL),
     out(notifyPC(COL)),
     out(traj([path(ID,P,F)|List]))
     ;
```

```
      out(traj([path(ID,P,F)]))
    )
  )
).

reaction(
  out(notifyPC(L)),
  from_tc,
  (
    in(notifyPC(L)),
    (
      L \= [],
      L = [pcol(ID,ID1,P)|T],
      out(cd_percep(ID,collision(ID1,P))),
      out(cd_percep(ID1,collision(ID,P))),
      out(notifyPC(T))
      ;
      true
    )
  )
).

reaction(
  out(finish(ID)),
  from_agent,
  (
    in(finish(ID)),
    in(traj(L)),
    remove(path(ID,_,_),L,List) ,
    out(traj(List))
  )
).
```

La reazione sottostante viene generata quando un CollisionDetector che è uscito vincitore da una negoziazione termina il percorso che dava vita alla collisione. A questo punto notifica il via libera all'agente che si era bloccato per evitare la collisione. Questa reazione genera una percezione, la quale diverrà poi intenzione del CollisionDetector che ha fermato il proprio muletto, così da far ripartire il moto del veicolo.

```
reaction(
  out(free_way(ID,ID1)),
  from_agent,
  (
    in(free_way(ID,ID1)),
    out(cd_percep(ID1,free_way(ID)))
  )
```

```
).



addPath(X,L,O) :- append([X],L1,O).

remove(X,[],[]).
remove(path(ID,_,_),[path(ID,_,_)|T],T) :- !.
remove(ID,[ID|T],T) :- !.
remove(X,[H|T],[H|T1]) :- remove(X,T,T1).


intersection(line(_,0.0,_),line(_,0.0,_),_) :- !,false.
intersection(line(0.0,_,_),line(0.0,_,_),_) :- !,false.
intersection(line(A1,B1,C1),line(A2,B2,C2),point('e','e')) :-
    A is sqrt((A1-A2)*(A1-A2)), A < 0.01,
    B is sqrt((B1-B2)*(B1-B2)), B < 0.01,
    C is sqrt((C1-C2)*(C1-C2)), C < 0.01, !.

intersection(line(A1,B1,C1),line(A2,B2,C2),_) :-
    M1 is (-A1/B1),
    M2 is (-A2/B2),
    M1 = M2, !, false.
intersection(line(0.0,B1,C1),line(A2,B2,C2),X) :- !,
    intersection(line(A2,B2,C2),
    line(0.0,B1,C1),X).
intersection(line(A1,B1,C1),line(A2,B2,C2),point(X,Y) ) :-
    Y is (A2*C1 - A1*C2) / (A1*B2 - A2*B1) ,
    X is -(B1/A1)*Y - C1/A1.

checkPC(P,[],[]).
checkPC(path(ID,P,F),[path(ID1,P1,F1)|T],[pcol(ID,ID1,Point)|T1]):-
    intersection(F,F1,Point),
    (
      Point = point('e','e'),
      distance(P,P1,D) , D < 20
      ;
      distance(P,Point,D), D < 20,
      distance(P1,Point,D1),D1 < 20,
      distance(P,P1,D2), D2 < 20
    ),!,
checkPC(path(ID,P,F),T,T1).
checkPC(P,[H|T],T1) :- checkPC(P,T,T1).
```

```
distance(point(X1,Y1),point(X2,Y2),D) :-
    D is sqrt(((X1-X2)*(X1-X2)) + ((Y1-Y2)*(Y1-Y2))).

decrease(N,O) :- O is N - 1.
```

4.2.2 *Nodi Brain*

La programmazione dei Brain è stata fatta semplicemente per fare in modo che un singolo cambiamento dell'ambiente, effettuato da un componente della società, sia percepito da tutti gli altri componenti.

```
reaction(
  out(follow(Traj)),
  from_agent,
  (
    in(follow(Traj)),
    out(nav_cmd(follow(Traj)) )
  )
).

reaction(
  out(throttle(Vel)),
  from_agent,
  (
    in(throttle(Vel)),
    out(nav_cmd(throttle(Vel)) )
  )
).

reaction(
  out(brake(Vel)),
  from_agent,
  (
    in(brake(Vel)),
    out(nav_cmd(brake(Vel)) )
  )
).

reaction(
  out(change_speed(Vel)),
  from_agent,
  (
    in(change_speed(Vel)),
    out(nav_cmd(change_speed(Vel)) )
  )
).
```

```
reaction(
  out(soft_brake(T)),
  from_agent,
  (
    in(soft_brake(T)),
    out(nav_cmd(soft_brake(T)))
  )
).

reaction(
  out(arrived(T)),
  from_agent,
  (
    in(arrived(T)),
    out(cd_percep(end(T))),
    out(th_percept(targetReached(T)))
  )
).

reaction(
  out(currentCurves(X)),
  from_agent,
  (
    in(currentCurves(X)),
    out(th_percept(passOnNode(X))),
    out(cd_percep(currentCurves(X)))
  )
).
```

5 CONCLUSIONI

Il primo prototipo costruito, seppur embrionale, è in grado di offrire un ambiente di simulazione in cui i veicoli sono in grado di trasferire merce evitando di collidere e i pacchi persi vengono recuperati. Come ogni prototipo offre diverse possibilità di miglioramento, alcune sono elencate in seguito:

RISOLUZIONE BUG Dato il non-determinismo del sistema possono emergere casi non previsti e difficilmente riproducibili.

SIMULAZIONE APPOGGIATA ALLO SPAZIO Un'alternativa all'attuale implementazione è quella di utilizzare lo spazio per pubblicare tutte le informazioni necessarie alla rappresentazione dell'entità e dedicare un agente (o artefatto) al rendering, coerentemente alle informazioni pubblicate nello spazio; questa situazione risulterebbe anche più applicabile a un caso reale, non simulato.

RAFFINAMENTO E OTTIMIZZAZIONE DELLA COLLISION-DETECTION Una volta acquisite competenze derivanti dalla prima stesura è possibile migliorare i sistemi di detection, soprattutto in termini di accuratezza.

ADATTAMENTO DELLA MAPPA Come accennato nella sezione 2.1.2, in seguito alla caduta di un pallet potrebbe essere interessante modificare la viabilità del magazzino (favorendo il recupero) e costringere gli autisti a ri-pianificare; in questo modo tale contingenza non bloccherebbe l'intera attività.

Concludendo si può affermare che il paradigma ad agenti è una valida soluzione per la realizzazione di simulazioni di questo tipo, soprattutto dal punto di vista della modellazione. Per quanto riguarda le prestazioni del sistema ciò dipende molto dalla piattaforma specifica: più sono le astrazioni, più è probabile che l'infrastruttura produca overhead; la realizzazione di applicazioni *real-time*, come questa se applicata nella realtà, potrebbe risultare problematica.

BIBLIOGRAFIA

[1] *CArtAgO manual (http://cartago.sourceforge.net/?page_id = 47).*

[2] *TuCSoN manual (http://apice.unibo.it/xwiki/bin/download/TuCSoN/Documents/manual.pdf).*

[3] Materiale del corso (http://apice.unibo.it/xwiki/bin/view/courses/smalm1011), 2010-2011.

[4] Michael Wooldridge Rafael H. Bordini, Jomi Fred Huebner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. WILEY, 2007.