Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Behavior trees per la rappresentazione dei processi decisionali nei sistemi multiagente.

Relatore: Prof. Ferrari Carlo

Studente: Davide Fiorentin

4 Dicembre 2017
Anno Accademico 2016/2017

Non è vero che abbiamo poco tempo:
la verità è che ne perdiamo molto.

SENECA

Abstract

Un Behavior Tree (BT) è un modello che permette la gestione di un processo decisionale in un agente autonomo, il quale deve variare il proprio comportamento in base alle esigenze.
Introdotti recentemente prima nell'industria videoludica e successivamente applicati in robotica, i BT offrono significativi benefici in termini di robustezza, manutenibilità, leggibilità e scalabilità rispetto alle alternative quali le Macchine a Stati Finiti, sia normali che gerarchiche.
Poiché l'enorme complessità legata alla gestione del processo decisionale in un sistema multiagente quale un'applicazione di IoT è una sfida attuale, l'applicazione dei BT in questi ambiti può fornire un miglioramento significativo alle tecnologie legate ai sistemi di controllo attuali proprio in virtù dei benefici che questo modello possiede. Dopo aver analizzato lo stato dell'arte e presentato in maniera formale i Behavior Trees, verrà proposta una struttura di riferimento per la realizzazione delle applicazioni modellabili come sistemi multiagente. Per concludere, si applicherà questa struttura ad un'applicazione IoT per sottolinearne l'utilità ed utilizzabilità.

Indice

Acronimi

AI - Artificial Intelligence
BT - Behavior Tree
DT - Decision Tree
FMS - Finite State Machine
HFMS - Hierarchical Finite State Machine
HTN - Hierarchical Task Network
IoT - Internet of Things
Iota - Internet of Things application
MAS - Multi Agent System

Capitolo 1

Introduzione

La realizzazione dei processi decisionali nei vari sistemi, siano questi industriali, logistici o applicati ad elementi robotici, è un tema aperto di notevole interesse. La diminuzione dei costi, la miniaturizzazione e l'aumento delle capacità di calcolo dei dispositivi hanno dato il via alla proliferazione capillare di dispositivi di controllo in questi ambiti, portando alla luce nuove problematiche.
Le applicazioni attuali legate alla domotica o all'IoT in particolare si basano su un gran numero di piccoli sensori con una limitata capacità di calcolo che devono riuscire a coordinarsi e a prendere delle decisioni, non limitandosi solamente al singolo dispositivo ma all'unione dell'intero gruppo, rendendo la gestione del processo decisionale ancora più complesso. Per riuscire ad affrontare il problema, una delle strade percorse è quello di immaginare l'insieme dei sensori di un'applicazione IoT come un *Sistema Multiagente (MAS)* [46] permettendo così di dividere in elementi via via sempre più piccoli il sistema stesso. Per quanto riguarda la realizzazione del processo decisionale tuttavia non si è ancora arrivati ad uno standard, in quanto le diverse applicazioni possono richiedere scelte implementative differenti. [1]
Nell'ambito dei MAS, spesso per le scelte decisionali si fa riferimento alle *Macchine a Stati Finiti (FMS)* [2] [3] o ad una loro estensione, ovvero le *Macchine a Stati Finiti Gerarchiche (HFMS)*; se tuttavia consideriamo un ambito IoT, questi è caratterizzato da un elevato numero di componenti spesso eterogenei tra loro che possono essere inseriti o rimossi nel sistema molto frequentemente. La modifica al sistema decisionale basato su FMS che ne consegue è critica: inserire o rimuovere un elemento potrebbe invalidare uno stato del sistema, rendere impraticabili alcune transazioni o semplicemente è troppo oneroso riprogrammare la FMS per tutti i vari componenti alla modifica di uno stato. Per questo motivo è necessario trovare un'alternativa che possa aiutare a risolvere queste problematiche.

Alcuni lavori sono stati effettuati per estendere le FMS applicando ad esempio algoritmi evolutivi [4],[5]. In questo lavoro di tesi vengono proposti i Behavior Tree come soluzione alternativa alle FMS per la progettazione del processo decisionale in un sistema Multi Agente.

1.1 Research Question

Lo scopo di questa tesi può essere rappresentato da questa Research Question:

"Può essere utilizzato un Behaviour Tree per realizzare un sistema di controllo intelligente basato su modello MAS?

Quest'ultima è riepilogativa di una serie di ulteriori Research Question, qui elencate per completezza:

- **E' possibile estendere le funzionalità di un BT ad altri ambiti non ancora esplorati?**
- **E' possibile implementare un BT in ambiti reali legati a domotica e robotica per superare le problematiche degli FMS attuali?**
- **Potranno partire da questo per realizzare applicazioni basate su Machine Learning o Reinforce Learning?**

1.2 Struttura della tesi

La prima parte della tesi ha lo scopo di presentare la metodologia generale ed il formalismo relativo ai Behavior Tree e ai sistemi multiagente. La seconda parte invece entrerà nel dettaglio dell'implementazione, presentando la soluzione architetturale proposta per questa tipologia di problemi. Infine, si parlerà dell'applicazione di questa struttura in ambito IoT e di eventuali ulteriori sviluppi futuri.

Capitolo 2

Stato dell'Arte

Per gli scopi di questo lavoro di tesi, dopo aver brevemente ricapitolato i concetti chiavi dei sistemi multi agente, sarà analizzato nel dettaglio solo la parte relativa al processo decisionale di un MAS, fornendo una breve panoramica sui temi dell'IA e delle varie tecniche implementative utilizzate in alcuni lavori collegati al tema che rappresentano la base su cui sono stati sviluppati i Behavior Trees. I BT non verranno trattati in questo capitolo in quanto saranno ampiamente analizzati nel capitolo 3.

2.1 Sistemi Multi Agente

Negli ultimi anni il paradigma degli agenti software intelligenti è diventato un tema aperto ed ampiamente trattato, soprattutto grazie ai vari campi di applicazione [6] [7].
Un **sistema multiagente** può esser considerato come un insieme di entità eterogenee cooperanti con diversi gradi di intelligenza. Questo gruppo di agenti possiede uno scopo comune e le azioni intraprese sono volte appunto a soddisfarlo, pur avendo al contempo poca o nessuna informazione relativa all'ambiente in cui operano. In base alle varie percezioni derivanti dell'ambiente o dall'interazione con altri, i singoli agenti possono modificare il proprio comportamento, effettuando un ragionamento basato sul proprio bagaglio informativo e decidendo di conseguenza.
Un agente intelligente infatti può inviare e ricevere informazioni tramite protocolli, generare più obiettivi da raggiungere progressivamente o realizzare una pianificazione anche per gli altri agenti. Questi servizi messi a disposizione da un MAS supportano una rappresentazione information-rich, mantengono esplicito il modello per se stessi e gli altri e può essere utilizzato anche in presenza di informazioni incomplete, inconsistenti o incerte. Per-

mette anche il ragionamento basato sulle capacità che l'agente stesso non possiede ma che sono proprie di altre entità presenti nel sistema, utilizzando la comunicazione e negoziazione. Per fare ciò è richiesto un linguaggio di specifica e un meccanismo che supporta l'apprendimento. E' inoltre possibile assegnare ruoli diversi ad agenti diversi nello stesso sistema, quali ad esempio dei ruolo di coordinatori o di esecutori di azioni fisiche. A supporto di ciò deve esistere un architettura ed una gestione organizzativa che permetta il corretto funzionamento del sistema.

Recentemente [8], il termine sistema multiagente ha assunto un significato più generale ed esteso, ed è ora utilizzato per indicare tutti i tipi di sistema composti da componenti autonomi che mostrano le seguenti caratteristiche [9] [10] [11]:

- ogni agente ha capacità limitate per raggiungere il proprio obiettivo;
- non esiste un sistema di controllo globale completamente informato;
- i dati sono decentralizzati;
- la computazione è asincrona.

Un applicazione di tale sistema è un MAS per un'azienda produttiva.

2.1.1 Architettura MAS

Il gruppo di agenti operativi (i componenti robotici) rappresentano il livello più basso di un sistema multiagente - **il livello operazionale**.
Gli *agenti operazionali intelligenti* rappresentano entità che agiscono indipendentemente effettuano le operazioni su richiesta dal livello superiore.
Gli agenti gestionali popolano il livello superiore di un MAS, chiamato **livello gestionale.** [12] [13].

Lo scopo principale di un sistema multiagente è quello di risolvere problemi comuni, che possono essere suddivisi in sottoproblemi sempre più piccoli che vengono distribuiti dal livello gestionale agli agenti operazionali. La distribuzione dei compiti da un lato e l'esecuzione dei task dall'altro richiedono due differenti unità di cooperazione e negoziazione nello stesso sistema. Per questo il livello superiore continee due agenti, il *Contract Manager* ed il *Conflict Manager*.
Il **Contract Manager** si occupa della distribuzione dei task ogni volta che un nuovo job entra nel sistema. Deve gestire tutti gli agenti operazionali

specificando i singoli compiti che devono affrontare. Questo può essere effettuato in due diversi modi: il Contract Manager può distribuire i compiti agli agenti in maniera gerarchica semplicemente assegnando un job ad un singolo agente, oppure può richiedere che un problema venga risolto dall'intera comunità di agenti operazionali. Questo instaura un processo di negoziazione tra il contract manager ed i singoli agenti, simile a quello che avviene in ambito finanziario. [13] [14]
Il **Conflict Manager** invece ha lo scopo di gestire le procedure di cooperazione e negoziazione mentre i singoli agenti operazionali si occupano di svolgere il proprio lavoro. I singoli comportamenti di tutti gli agenti votati al singolo scopo non possono essere predetti in anticipo, ma si può evidenziare un possibile conflitto quando diversi agenti hanno scopi differenti, portando quindi un instabilità nel sistema. Per evitare questo il Conflict Manager si occupa tramite la negoziazione di arrivare ad una soluzione del problema che possa essere accettabile per le varie parti, secondo un criterio precedentemente stabilito. [15].
La struttura gerarchica di un sistema multiagente è mostrata in Figura 2.1.

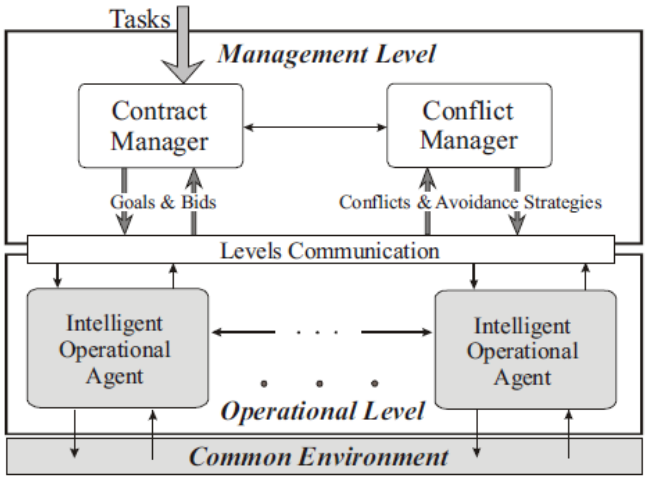

Figura 2.1: Struttura generale di un sistema multiagente

Ai fini di questo lavoro di tesi, verrà posto particolare interesse alla struttura interna di un agente operazionale ed in particolare alla gestione dei processi decisionali per la risoluzione del proprio job; per questo motivo la scelta comportamentale può essere considerata un'**Artificial Intelligence (AI)**.

2.2 AI

In questa sezione verranno illustrate le tematiche principali dei sistemi che implementano una forma di AI, necessaria per la realizzazione di un sistema decisionale, richiamando principalmente le informazioni del manuale di Russell & Norvig (2009)[16].
Il termine AI è stato utilizzato per riferirsi ai diversi sistemi automatizzati, ma in ambito accademico può essere separato in 4 definizioni principali:
l'AI è un sistema che

- pensa come un umano
- agisce come un umano
- pensa razionalmente
- agisce razionalmente

In questo elaborato, sarà utilizzata l'ultima definizione per riferirsi agli agenti, ovvero *agenti che agiscono razionalmente*. Un **agente** è semplicemente qualcosa che agisce, che percepisce l'ambiente tramite l'uso di sensori ed agisce sull'ambiente tramite degli attuatori. La *percezione* è definita dagli input che l'agente ottiene in un determinato istante temporale. La *sequenza percepita* è la storia completa di tutto ciò che l'agente ha percepito. Una rappresentazione generale di un agente può essere vista in Figura 2.2.

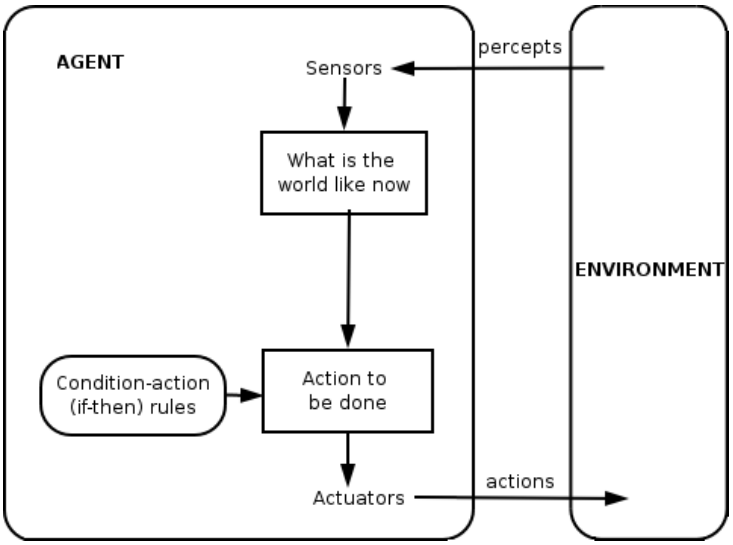

Figura 2.2: Struttura generale di un AI

Un *agente razionale* è tale se cerca di raggiungere il risultato migliore. La razionalità è basata sulle misure di performance che definiscono i criteri di successo: la conoscenza di un agente del proprio ambiente, le azioni che effettua e la sequenza percepita.

2.2.1 Tipi di AI

Esistono diverse tipologie di agenti, che per la loro natura possiedono diversi livelli di razionalità.

2.2.1.1 Simple reflex agents

Questi agenti agiscono solo sulla base della percezione corrente, ignorando il resto della storia percepita. Le funzioni di questi agenti sono basate sulle regole di condizione-azione: se c'è una particolare condizione, allora attiva l'azione.
Questa tipologia di agenti è tipicamente utilizzata in ambienti completamente osservabili per evitare possibili ambiguità nel caso di input aliasing. Ricerche recenti tuttavia hanno dimostrato che l'aliasing non è così svantaggioso come lo si credeva: Nolfi (2002)[17] ha dimostrato che l'agente può utilizzare la propria interazione con l'ambiente per estrarre informazioni addizionali per rimuovere l'ambiguità in input grazie alla coordinazione senso-motoria. Effettuare azioni che facilitano il riconoscimento dei pattern da parte del sensore è definito come *percezione attiva*.

2.2.1.2 Model-based reflex agents

Un modo per gestire i problemi che emergono utilizzando i simple reflex agents che operano in ambienti parzialmente osservabili è introdurre il concetto di *stato interno dell'agente*. Lo stato è dipendente dalla storia percepita e può essere utilizzato per ragionare su determinati aspetti non osservati relativi allo stato corrente. Aggiornare lo stato tipicamente richiede informazioni relative all'evoluzione dell'ambiente e all'effetto dell'agente nel mondo, richiedendo quindi una implementazione di modello del mondo.

2.2.1.3 Goal-based agents

Questi differiscono ai simple reflex agents in quanto considerano il futuro in due modi: quali saranno gli effetti delle azioni dell'agente e quali azioni aiuteranno l'agente a raggiungere il proprio obiettivo. Queste due informazioni combinate garantiscono all'agente una forma di ragionamento e non

richiedono una mappatura diretta stato-azione che limitano gli agenti di tipo riflessivo. Questi agenti scelgono l'azione migliore che permette loro di raggiungere il proprio obiettivo utilizzando un modello per descrivere l'ambiente circostante. I campi che si occupano specificatamente della decisione delle azioni da intraprendere sono quelli legati al *search and planning*. La struttura di un agente goal-based generale può essere visto in Figura 2.3.

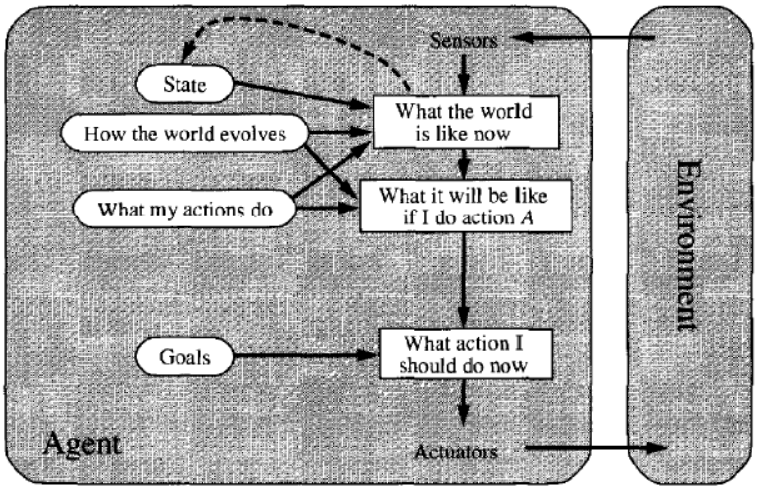

Figura 2.3: Struttura di un agente Goal-Based

Gli agenti goal-based sono meno efficienti di un reflex-based agent, in quanto l'azione più razionale dev'essere determinata considerando molte variabili contemporaneamente, ma tipicamente sono più flessibili in quanto la conoscenza a supporto del processo decisionale è rappresentata esplicitamente e può essere modificata.

2.2.1.4 Utility-based agents

Per generare comportamenti di alta qualità bisogna applicare alcune misure per quantificare quanto un determinato stato è utile al fine di raggiungere lo scopo finale dell'agente. Questo può essere rappresentato da una *funzione di utilità* che mappa gli stati in un valore numerico che rappresenta l'utilità dello stato, ovvero la qualità nella scelta dell'essere utili. Un agente razionale utility-based sceglie l'azione che massimizza l'utilità delle prossime azioni, esaminando l'utilità di ogni scelta. La Figura 2.4. mostra un agente utility-based generale. Questo tipo di agente è molto utile se deve considerare molteplici obiettivi conflittuali tra loro, ognuno con una certa possibilità di

successo. Tutti gli agenti razionali devono ragionare come se avessero una funzione di utilità, sia questa esplicita o no.

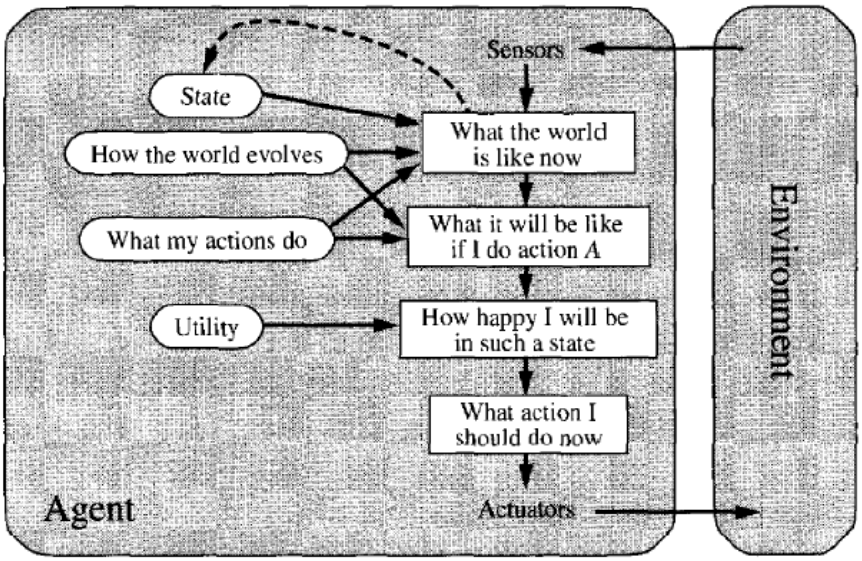

Figura 2.4: Struttura di un agente Utility-Based

2.2.1.5 Learning agents

Definire manualmente la totalità del comportamento di un agente complesso può risultare impossibile per il programmatore, il cui desiderio è quello di un agente che apprenda il proprio comportamento tramite l'interazione con l'ambiente. Un **learning agent** ragiona proprio in questo modo, e la sua struttura può essere riassunta nella figura 2.5, dove vengono evidenziate le 4 componenti principali:

- **Critic** - Determina il modo in cui le performance dell'agente dovrebbero cambiare per migliorare la propria utilità, e questa informazione viene mandata tramite feedback al Learning element.
- **Learning element** - Responsabile di effettuare i miglioramenti.
- **Performance element** - Responsabile della selezione delle azioni esterne.
- **Problem generator** - Suggerisce quale azione intraprendere per effettuare nuove esperienza che aumentino il patrimonio informativo dell'agente.

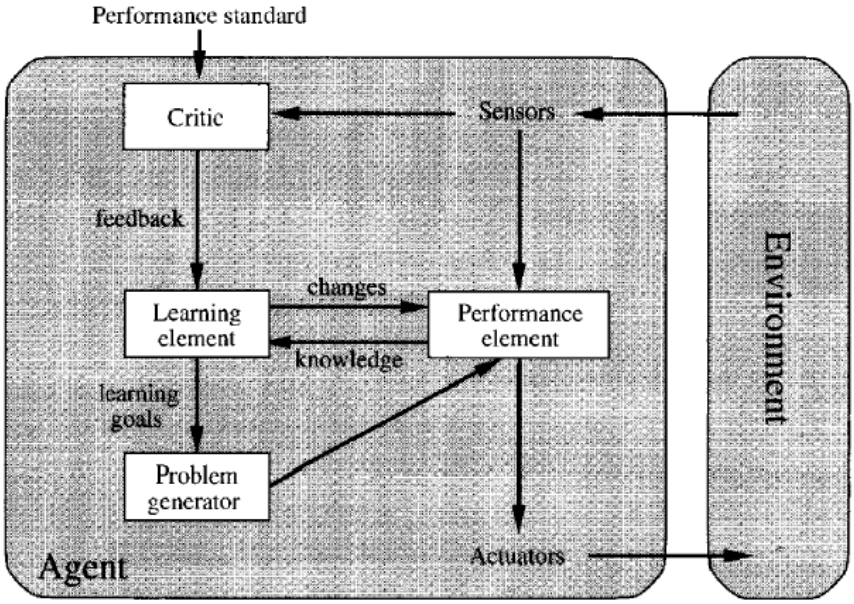

Figura 2.5: Struttura di un Learning Agent

Come si può notare, il Learning Element è strettamente dipendente dal performance element, ragion per cui quest'ultimo viene implementato in un secondo momento. Critic usa il feedback per informare il Learning di quanto bene stia andando rispetto ad una data funzione di utilità. Questo è particolarmente importante in quanto il percettore non fornisce abbastanza informazioni relative alle performance dell'agente. Il Problem generator suggerisce delle azioni esplorative che potrebbe portare ad un'azione sub-ottima nel breve termine, ma tramite le nuove informazioni ottenute portare ad una convergenza verso una soluzione ottima a lungo termine.

2.2.2 Adattamento ed apprendimento

Codificare manualmente un'AI complessa multifunzionale può essere molto complesso, oneroso in termini di tempo o semplicemente impossibile a causa della limitata conoscenza del sistema nel quale l'AI opera. All'aumentare della complessità dei task effettuati dagli agenti diventa fondamentale che questi ultimi imparino a migliorare le proprie performance automaticamente. L'abilità per un agente intelligente di adattare il proprio comportamento risulta inoltre indispensabile nel momento in cui avvengono dei cambiamenti nell'ambiente stesso. Questa sezione cerca di riassumere alcuni degli elementi utilizzati, enunciando alcune informazioni basilari di Machine Learning.

E' importante sottolineare una differenza fondamentale tra *apprendimento* ed *adattamento.*, riportando il lavoro fatto da Arkin (1998) [18]
L'**adattamento** è un processo per il quale un agente gradualmente cambia il proprio comportamento per seguire l'ambiente in cui è immerso al fine di migliorare le propria performance, senza tuttavia subire drastici cambiamenti alla propria struttura comportamentale. Le tre categorie principali dell'adattamento sono le seguenti:

- **Behavioural Adaptation** - I comportamenti del singolo agente vengono modificati dall'agente stesso.
- **Evolutionary Adaptation** - I discendenti modificano il loro comportamento in base al successo o al fallimento dei propri predecessori immersi nel proprio ambiente. Ovviamente per poter applicare questo metodo bisogna che gli agenti siano in grado di riprodursi, e le modifiche avvengono su un periodo particolarmente lungo.
- **Sensor Adaptation** - Le percezioni di un agente si raffinano in maniera direttamente proporzionale al tempo in cui quest'ultimo è immerso nell'ambiente.

L'**apprendimento** invece rappresenta una modifica sostanziale alla base informativa o alla struttura rappresentativa del comportamento dell'AI. Per esempio questo può avvenire in seguito all'introduzione di nuova conoscenza nel sistema, dalla caratterizzazione di un sistema o dalla generalizzazione di alcuni concetti derivanti da esami multipli dell'ambiente o dall'esterno (ad esempio dalla storia pregressa di altri agenti simili). I metodi di apprendimento si differenziano in base alla scala su cui vengono applicati (singolo agente o comunità di agenti) oppure in base alla sorgente delle informazioni contenute nel feedback del Critic. Questo feedback può essere ritenuto una

sorta di Learning supervionato, con sfumature che variano dal Supervised all'Unsupervised Learning:

- **Unsupervised Learning** - l'informazione disponibile per l'agente è tipicamente ristretta solamente ai dati di input e non ci sono indicazioni riguardanti quale sia lo stato ottimo. Le performance dell'agente vengono determinate dal Critic, e queste informazioni vengono poi inviate tramite feedback all'agente. Questa tipologia di apprendimento viene solitamente utilizzata per riconoscenere degli andamenti o dei pattern nei dati.

- **Supervised Learning** - l'agente dispone dell'informazione completa sia dell'input che dell'output, oltre a quella relativa allo stato ottimo o dell'output ottimo da raggiungere.

Tutti i sistemi di apprendimento rientrano in uno stadio compreso tra questi due estremi. In particolare. è stato classificato [18] il meccanismo legato all'appprendimento:

- **Reinforcement Learning** - Vengono fornite all'AI delle ricompense volte a modificare alcuni parametri tramite un processo di *trial & error* basandosi su una funzione d'utilità dell'agente. L'agente cerca di massimizzare le ricompense ottenute, migliorando in questo modo le proprie performance tramite un processo di feedback basato sul Critic. La figura 2.6 mostra in maniera grafica il funzionamento di questo metodo.

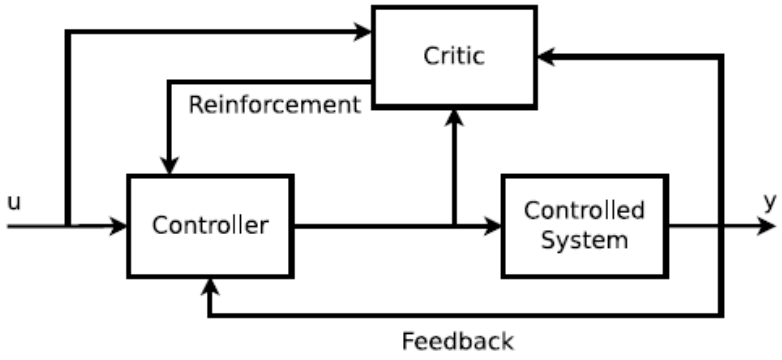

Figura 2.6: Struttura del Reinforcement Learning

- **Evolutionary Learning** - Le modifiche all'AI vengono derivate in maniera deduttiva basandosi sulle alterazioni della popolazione iniziale

definita dal programma, utilizzando operatori genetici basati su un confronto tra la popolazione attuale ed una funzione ideale.

- **Learning by Imitation** - Una forma simile a quella in cui gli animali apprendono quando viene mostrato loro il modo in cui bisogna effettuare una particolare azione. Recentemente (2016) si è iniziato a parlare anche di modelli di Mimesi [19].

2.3 Tecniche per implementare un'AI

Ogni agente deve analizzare le informazioni che ha percepito e tramite una qualche procedura deve determinare l'azione razionale che dev'essere presa. L'input sono le informazioni ottenute dall'ambiente, siano queste informazioni istantanee o storia pregressa. L'output è l'azione da intraprendere. Questo processo decisionale può essere formulato in diversi modi; questa sezione ha lo scopo di descrivere alcuni del più popolari metodi elencandone i pro e contro. Particolarmente importanti sono le problematiche legate alle Macchine a Stati Finiti in quanto a partire da queste sono stati sviluppati i Behavior Tree; di questi ultimi la parte strettamente teorica verrà approfondita nel capitolo 3, mentre la struttura implementativa proposta sarà discussa nel capitolo 4.

2.3.1 Decision Tree

La corrispondenza tra un segnale in ingresso derivante dall'analisi dell'ambiente ed il corrispondente output di un agente può essere particolarmente complesso; un modo per semplificare questo processo è quello di realizzare questa corrispondenza tramite una matrice di scelte che combinate tra loro possano descrivere un sistema complesso. Un esempio di Decision Tree può esser visto in figura 2.7.

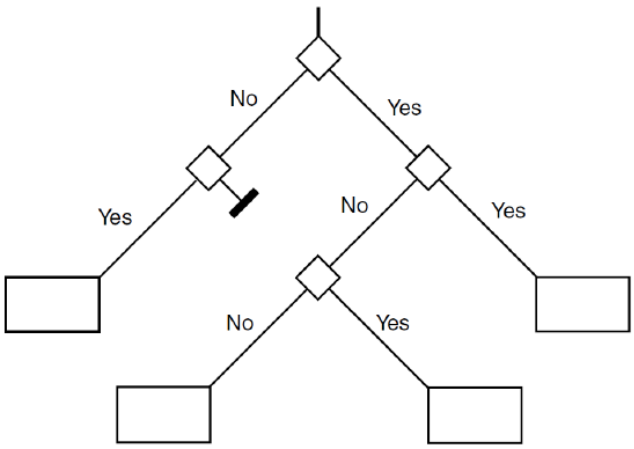

Figura 2.7: Struttura di un Decision Tree

Gli alberi decisionali hanno origine dal nodo radice e si propagano lungo l'albero valutando le varie decisioni finchè non è più possibile effettuare

scelte. I nodi che non possiedono altre scelte che si propagano verso il basso si chiamano foglie e sono composte da azioni da intraprendere. La maggior parte dei nodi decisionali sono di tipo binario, rendendo la computazione dell'albero molto veloce. Combinando le decisioni è possibile mappare un comporamento complesso in logica booleana.
La semplificazione del processo decisionale in una serie di scelte semplici rende i Decision Tree una tecniche veloce, facile da implementare e semplice da capire. L'utilizzo di nodi standard rende i Decision Trees modulari e di conseguenza facili da creare.
Questa semplice decomposizione di scelte tuttavia porta con se un problema intrinseco: l'albero contiene un numero di decisioni molto superiori rispetto a quelle effettivamente necessarie per determinare un'azione; questo può essere visto in quanto ad ogni scelta un solo ramo viene selezionato mentre gli altri vengono scartati, richiedendo una ripetizione dello stesso comportamento e scelte decisionali nella struttura dell'albero.
Un *albero bilanciato* è un albero che ha lo stesso numero di nodi decisionali tra la radice ed una qualsiasi foglia. In questo modo l'albero si esamina con un totale di $O(\log_2 n)$ decisioni, dove n è il numero di nodi decisionali dell'albero. Si può vedere come un albero sbilanciatio possa portare ad un degrado delle prestazioni fino a $O(n)$.
Recentemente [20] è stata anche implementato un albero decisionale utilizzando la Fuzzy Logic, che tenta di migliorare e rendere applicabile questo metodo ad alcune realtà attuali.

2.3.2 Finite State Machine

Questo tecnica di scelta decisionale parte dall'idea che un agente occupa un particolare stato in un dato momento, e continuerà a ripetere le azioni relative fino a quando non avviene un evento o una condizione che porta ad una modifica dello stato dell'agente. Le connessioni tra gli stati si chiamano *transizioni*,e collegano un solo stato ad un altro. L'attivazione di una transizione a causa di un evento è chiamata *trigger*. L'insieme degli stati, transizioni e condizioni di trigger definisce completamente una FSM.
A differenza dei Decision Trees, le FSM non considerano solo l'ambiente ma anche lo stato interno dell'agente. Per rappresentarle si utilizza un diagramma a stati introdotto da Harel nel 1987[21] e un esempio si può vedere nella figura 2.8.
Le FSM sono già state studiate in maniera approfondita in diversi campi di applicazione, ragion per cui esistono diversi strumenti volti al loro design e all'ottimizzazione di queste tecniche.

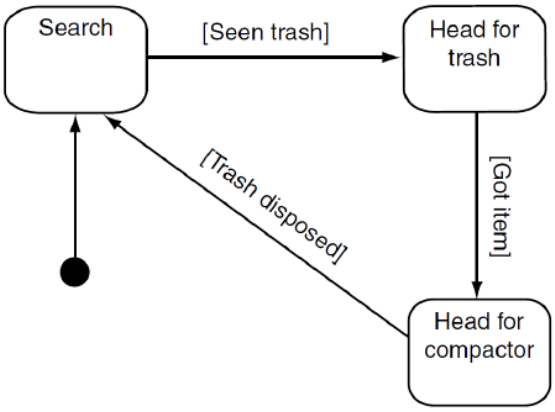

Figura 2.8: Struttura di un FSM

Ci sono molti modi per implementare una FSM che dipendono dall'uso finale, ma tutte rimangono relativamente semplici da realizzare ed implementare. Tutte queste implementazioni soffrono tuttavia di un problema intrinseco, chiamato **state explosion**. All'aumentare del numero di stati la commplessità del sistema aumenta in modo esponenziale, rendendo impossibile realizzare comportamenti complessi o consentire eventuali espansioni future in quanto l'aggiunta di un singolo stato richiede solitamente l'aggiunta di $2(N-1)$ transazioni, con N numero di stati. Per illustrare questo problema, sarà utilizzato l'esempio proposto da Millington & Funge (2009)[22] riportato in figura 2.9.

A causa di questo problema diventa impossibile mantenere larghe FSM, anche perchè non sono supportate la modularità e la flessibilità, visto che l'aggiunta o la rimozione di uno stato richiede un lavoro di aggiustamento su tutti gli agenti del sistema, che devono essere ricompilati o bloccati forzatamente. Esistono metodi per collegare a run time dei nuovi stati, ma lo sviluppo di questi metodi è particolarmente oneroso. Per questi motivi le FSM sono utilizzate solo per agenti che possiedono un numero esiguo di operazioni e che non verranno aggiornati quasi mai.

2.3.3 Hierarchical Finite State Machine

Il problema dello state explosion può essere limitato introducendo una gerarchia nelle FSM. Gli stati possono essere raggruppati in altri stati in maniera gerarchica in modo da ridurre in maniera significativa il numero di transizioni

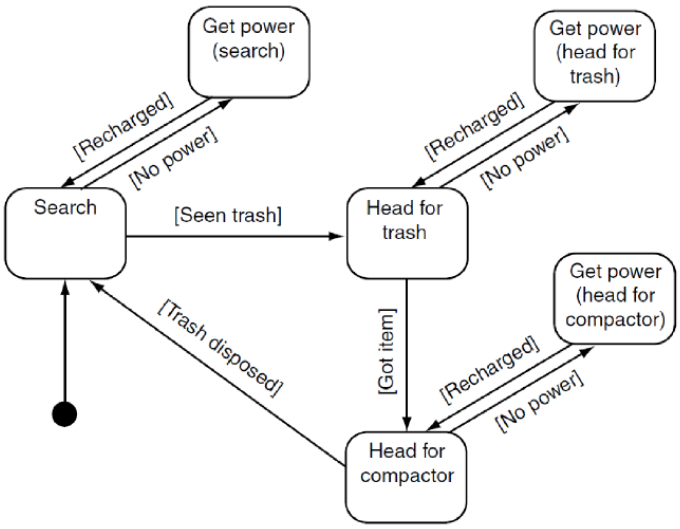

Figura 2.9: Esplosione di stati in una FSM

delle FSM. In figura 2.10 si vede come la complessità del problema si sia ridotto notevolmente rispetto al caso precedente.

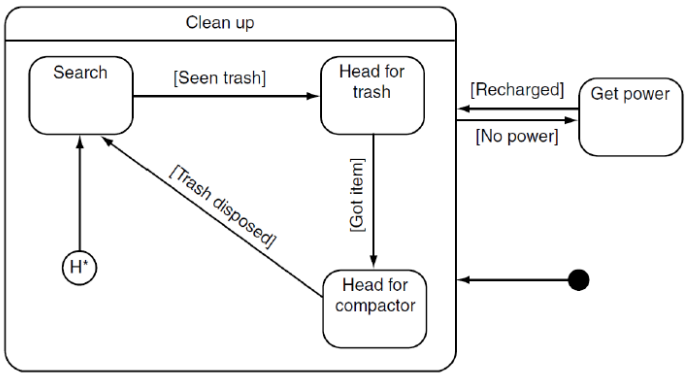

Figura 2.10: Esempio dell'HFSM per risolvere il problema dello state explosion

Le HFSM sono abbastanza efficienti e operano con una performance di $O(Nt)$ dove t è il numero di transizioni per stato. Anche se le HFSM sono utili per ridurre la complessità di un agente rispetto alle FSM, rimane il problema della mancanza di modularità e flessibilità in quanto le transizioni tra gli stati sono ancora hard-coded e rende impossibile gestire agenti di una certa complessità.

2.3.4 Hierarchical Task Network

Questo metodo serve per una pianificazione automatica al fine di generare una sequenza di azioni volte a raggiungere uno scopo. E' interessante osservare la differenza tra le *Hierarchical Task Network (HTN)* e le HFSM.
Le HTN utilizzano il concetto della decomposizione gerarchica per decomporre in maniera ricorsiva l'obiettivo fino a ridurlo ad una serie di azioni *primitive*, ovvero delle azioni semplici ed immediatamente eseguibili, che comporranno la sequenza finale. Le operazioni *non-primitive* rappresentano dei sotto-obiettivi che possono ancora essere ulteriormente scomposti. Un'altra differenza tra le HTN e le HFSM risiede nella descrizione della sequenza obiettivo: non è più una sequenza di stati finali, ma un'insieme di azioni che devono essere svolte per raggiungere lo scopo. [29] La Figura 2.11 mostra la decomposizione del processo in una HTN.
Per guidare il processo di decomposizione vengono utilizzati una serie di metodi che dividono i task in sotto-task. Questo avviene tipicamente utilizzando della conoscenza specifica del dominio per creare una serie di regole che riducono lo spazio di ricerca, aumentando l'efficienza complessiva del sistema.

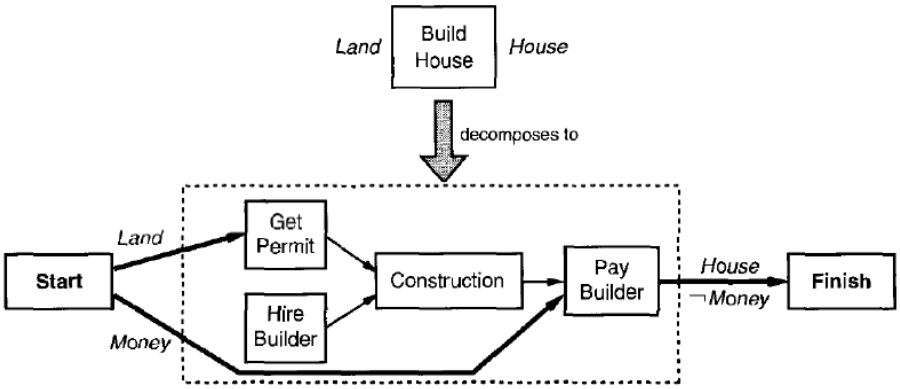

Figura 2.11: Esempio applicativo di una HTN

Capitolo 3

Behavior Tree

In questo capitolo viene introdotta la parte teorica alla base dei Behavior Tree, basandosi sulle linee guida presenti in [23], [24], [25] e sul formalismo utilizzato per l'applicazione in Robotica introdotto da questo articolo.[26]

3.1 Definizione

I **Behavior Tree** sono grafi aciclici orientati (*Directed acyclic graph, DAG*) $G(V; E)$ con $|V|$ nodi e $|E|$ archi utilizzati per rappresentare i processi decisionali. I nodi interni rappresentano le scelte decisionali, i nodi esterni o *foglie* i comportamenti da eseguire. L'unico nodo che non ha genitore viene detto *radice*. Ogni arco collega un nodo a profondità minore ad uno di profondità maggiore, iniziando dalla radice, unico nodo a profondità 0, e non creando in questo modo cicli. In questo modo si riesce a rappresentare in maniera più compatta un processo decisionale rispetto agli alberi decisionali. A differenza dei tradizionali alberi derivanti dalla teoria dei grafi[27], qualsiasi nodo presente nel BT ad eccezione della radice e del suo unico figlio potrebbe avere un qualsiasi numero di genitori[23]. Questo permetterebbe il riutilizzo di sottoalberi senza necessità di copiarli, ma riduce drasticamente la leggibilità; per questo motivo esiste la convezione per cui sono proibiti i nodi che hanno diversi genitori e la riutilizzabilità dei sotto alberi non avviene a livello dei nodi di controllo ma ai nodi foglia.

I Behavior Tree combinano i diversi punti di forza degli altri metodi:

- L'uso delle gerarchie delle Hierarchical Finite State Machine (HFSM) facilita la riutilizzabilità del codice in quanto basta modificare solo gli stadi più bassi per adattare il BT al caso in esame.
- Esprimere i comportamenti in termini di insiemi di task visti in una Hierarchical Task Network (HTN) permette una facile decomposizione dei behavior.
- La semplicità derivata dagli alberi decisionali permette l'utilizzo della ricorsione e facilita l'implementazione generale dei BT.
- La rappresentazione grafica degli stati derivata dalle FSM fornisce una buona visione d'insieme del comportamento riducendo la complessità rappresentativa.

A differenza delle FSM, i BT arrivano al proprio scopo semplificando ricorsivamente il comportamento desiderato in task in maniera simile a quello che avviene nelle HTN. Questa combinazione di gerarchia e ricorsività rendono i BT uno strumento molto potente per descrivere comportamenti complessi, in particolare in ambienti dove coesistono diverse elementi eterogenei tra loro o dove la scalabilità del processo decisionale ha un ruolo critico.

3.2 Sintassi e Semantica

Un BT è rappresentato sintatticamente da una struttura ad albero, basata da una notevole varietà di nodi che possiedono ciascuno una funzione interna individuale, ma esibiscono esternamente la stessa interfaccia.
Ogni nodo di un BT, ad eccezione della radice, è di uno dei sei possibili tipi, divisi per categoria:

- i nodi interni utilizzati per il controllo decisionale appartengono alla categoria *Composite* e sono di 4 tipi:
  - *Selector;*
  - *Sequence;*
  - *Parallel;*
  - *Decorator.*
- i nodi foglia rappresentano le azioni eseguite e appartengono alle categorie *Action* e *Condition*.

Le loro funzioni vengono riassunte nella Figura 3.1 [26], ed analizzati nel dettaglio nei paragrafi seguenti.

Node Type	Symb.	Succeeds if	Fails if	Runs if
Root	∅	tree S	tree F	tree R
Selector	?	1 Ch S	N Ch F	1 Ch R
Sequence	→	N Ch S	1 Ch F	1 Ch R
Parallel	⇉	$\geq S$ Ch S	$\geq F$ Ch F	otherwise
Decorator	◇	varies	varies	varies
Action n	□	$X_n(t) \in \mathsf{S}_n$	$X_n(t) \in \mathsf{F}_n$	$X_n(t) \in \mathsf{R}_n$
Condition n	○	$X_n(t) \in \mathsf{S}_n$	$X_n(t) \in \mathsf{F}_n$	never

Figura 3.1: Funzioni dei nodi: Ch = children, S = succeeded, F = failed, R = running, N = # children, S;F $\in N$ sono parametri dei nodi

La radice periodicamente, con frequenza f_{tick}, genera un segnale abilitante chiamato *tick* che viene propagato nei rami seguendo l'algoritmo definito da ogni nodo. Quando un tick raggiunge una foglia, esegue un ciclo di *Action* o *Condition*. Le *Action* possono alterare la configurazione del sistema ritornando uno dei tre possibili valori di stato: *Success, Failure* oppure *Running*. Le *Condition* non possono alterare il sistema e ritornano solo uno dei due possibili valori di stato: *Success o Failure*. Questi valori di ritorno vengono propagati nell'albero abilitando altre funzioni fino a tornare alla radice. Un'esecuzione è completa quando la radice ritorna Success oppure quando tutti i figli terminano con Failure. I nodi che non sono stati sottoposti al tick entrano in uno stato speciale, *NotTicked*. Il BT attende prima di mandare un nuovo tick in modo da mantenere f_{tick} costante.

3.2.1 Valori di Ritorno

Ogni nodo di un BT ha un valore di ritorno, generalmente Success o Failure. *Success* rappresenta è lo stato di un nodo che è stato eseguito con successo. Al contrario, il valore *Failure* è utilizzato quando un nodo fallisce durante l'esecuzione. Questo tuttavia non definisce la condizione sotto il quale un nodo fallisce, ragion per cui alcune estensione dei BT hanno introdotto i valori *Exception* o *Error* che forniscono queste informazioni. Le informazioni ottenute dal valore di ritorno sono quelle su cui si basa l'algoritmo per determinare il proseguo della scelta decisionale.

3.2.2 Tipi di nodo

La divisione principale è tra i nodi foglia e i nodi interni.

3.2.2.1 Nodi Foglia:Actions & Conditions

I nodi foglia sono rappresentati dalle tipologie *Actions* e *Conditions*; devono essere sviluppati specificatamente per il task e per l'ambiente in esame, ma possono essere replicati facilmente in più punti dell'albero nel caso in cui delle azioni debbano essere ripetute in più punti.

Actions

L'agente agisce sull'ambiente circostante utilizzando i nodi **Actions**, che rappresentano il comportamento o parte di esso, in base alla granularità con cui stiamo visualizzando il problema.
I nodi *Actions* possono essere *bloccanti* (forniscono il risultato solo quando l'azione è effettivamente completata) o *non bloccanti*, ma in generale in entrambi i casi ritornano sempre *Success*. Infatti, un'azione solitamente segue un nodo *Conditions* che determina se l'azione è effettuabile oppure no. Alternativamente, se dev'essere applicata al nodo una sorta di precondizione fissata, la condizione è verificata nel nodo stesso, che ritorna *Failure* nel caso in cui questa non sia verificata o avviene un qualche time-out interno.
A livello formale, quando un nodo *Actions*, con indice n, viene abilitato, determina il valore di ritorno esaminando l'attuale configurazione degli spazi degli stati $X_n(t)$ e verificando se appartengono ai sottoinsiemi *Success* S_n, *Failure* F_n o *Running* R_n. Lo pseudocodice può essere visionato di seguito 1.

```
1  if X_n(t) ∈ S_n then
2  |  return Success
3  else
4  |  if X_n(t) ∈ F_n then
5  |  |  return Failure
6  |  else
7  |  |  if X_n(t) ∈ R_n then
8  |  |  |  return Running
9  |  |  end
10 |  end
11 end
```

Algorithm 1: Actions

Conditions

I nodi di tipo **Conditions** hanno lo scopo di effettuare dei test su alcune proprietà dell'ambiente o dell'agente stesso, ritornando *Success* se la condizione è verificata o *Failure* altrimenti.
A livello formale l'algoritmo segue quello dei nodi *Actions*, tranne per il fatto che non possono essere nello stato *Running*; lo pseudocodice viene riportato sotto 2.

```
1 if X_n(t) ∈ S_n then
2 |  return Success
3 else
4 |  if X_n(t) ∈ F_n then
5 |  |  return Failure
6 |  end
7 end
```

Algorithm 2: Conditions

3.2.2.2 Nodi Interni: Composite

Tutti i nodi interni invece sono di tipo **Composite** e tipicamente non sono dipendenti dalla piattaforma ma possono essere riutilizzati in ogni BT. Tutti i nodi in un BT utilizzano un'interfaccia similare in modo da combinare arbitrariamente questi nodi nel BT senza la conoscenza delle altri parti del BT, fornendo delle proprietà di modularità e riutilizzabilità.
Un esempio di BT che sottolinea la rappresentazione grafica dei diversi nodi può essere visualizzato nella figura 3.2.

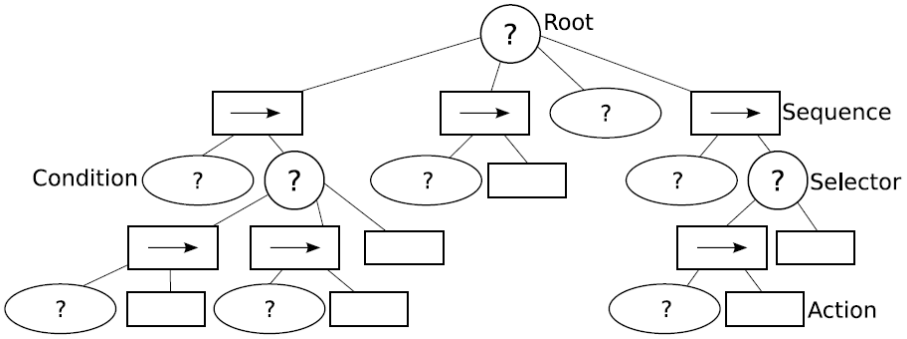

Figura 3.2: Esempio di BT con i vari tipi di nodi

Essendo rami del BT, i nodi *Composite* determinano il modo in cui viene eseguito un BT. Per i fini di questo lavoro, utilizzeremo solamente i nodi di tipo Sequence, Selector e Decorator, anche se ne esistono molti altri utilizzati in pratica. Nella figura 3.3 sono mostrati i diversi tipi di nodi Composite e le loro priorità di valutazione. Il nodo *Radice* è tipicamente un selettore senza il nodo genitore.

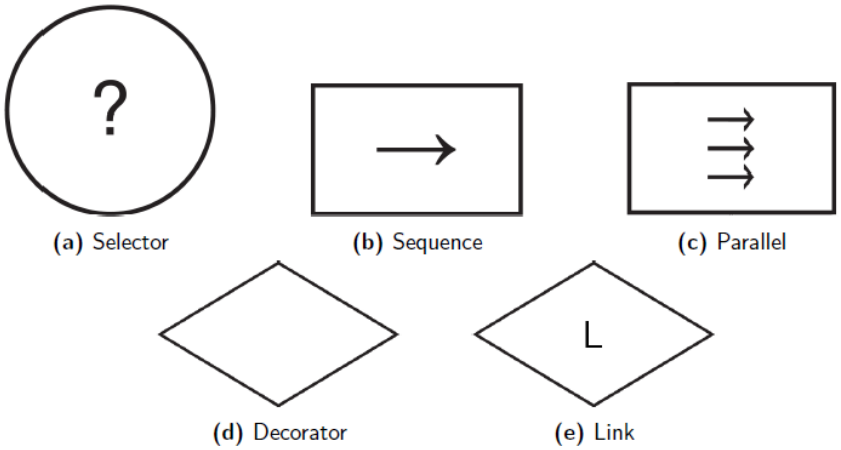

Figura 3.3: Esempio delle varie tipologie di nodi Compositie

Selector

Il nodo *Selector* ritorna *Success* quando uno dei figli ritorna *Success* e *Failure* quando tutti i figli ritornano *Failure*. Il selettore tipicamente testa i propri nodi figli sequenzialmente secondo una certa priorità, di solito da sinistra a destra. I nodi di tipo *Selector* garantiscono un meccanismo di fail-safe in quanto testano i vari figli in sequenza fino a quando una particolare azione sia applicabile allo stato corrente del sistema. L'immagine di un Selettore è il 3.3.a e lo pseudocodice è il seguente:

```
1  for i ← 1 to N do
2  |   state ← Tick(child(i)) ;
3  |   if state = Running then
4  |   |   return Running
5  |   end
6  |   if state = Success then
7  |   |   return Success
8  |   end
9  |   return Failure;
10 end
```

Algorithm 3: Selector

Sequence

Il nodo di tipo *Sequence* ritorna *Failure* se uno dei suoi figli fallisce, e *Success* se tutti i figli ritornano *Success*. Questa è l'operazione opposta di un nodo selettore, mantenendo la valutazione dei propri figli secondo un criterio di priorità, solitamente da sinistra a destra. L'icona di una *Sequence* è presente in 3.3.b ed il codice è riportato in basso. I nodi *Sequence* vengono solitamente utilizzati per effettuare combinazioni di task interdipendenti tra loro.

```
1  for i ← 1 to N do
2  |   state ← Tick(child(i)) ;
3  |   if state = Running then
4  |   |   return Running
5  |   end
6  |   if state = Failure then
7  |   |   return Failure
8  |   end
9  |   return Success;
10 end
```

Algorithm 4: Sequence

Parallel

I nodi *Parallel* differiscono dai precedenti in quanto eseguono tutti i nodi figli concorrentemente come se ciascuno di essi avesse il proprio thread di esecuzione. La condizione di ritorno di un *Parallel* può essere alterata in base al comportamento richiesto: può agire come un selettore, una sequenza o una funzione ibrida dove il valore di ritorno dipende strettamente dal numero di figli che ritornano Failure o Success. Quando un nodo di tipo *Parallel* viene sollecitato, propaga il *tick* a tutti i suoi figli e si mette in attesa dei loro valori di ritorno. Dati $S, F \in N$, se il numero di *Success* $>= S$, ritorna *Success*. Se il numero di *Failure* $>= F$, ritorna *Failure*, altrimenti ritorna *Running*. Solitamente i nodi di tipo *Parallel* sono utilizzati per eseguire concorrentemente rami distinti oppure più frequentemente per continuare a valutare una condizione mentre un azione viene eseguita.
L'icona è presente nella figura 3.3.c. ed il codice riportato nella tabella 5 sottostante.

```
1  for i ← 1 to N do
2  |  state ← Tick(child(i)) ;
3  end
4  if #Succ(state) >= S then
5  |  return Success
6  end
7  if #Fail(state) >= F then
8  |  return Failure
9  else
10 |  return Running;
11 end
```

Algorithm 5: Parallel

Decorator

I nodi di tipo *Decorator* tipicamente hanno un solo figlio e sono utilizzati per forzare un particolare stato di ritorno, per introdurre delle restrizioni temporali oppure per definire il numero di volte in cui un figlio deve agire prima di considerare la propria esecuzione terminata *(Decorator Repeat)*. L'immagine è indicata in Figura 3.3.d. e lo pseudocodice varia in base allo scopo del *Decorator*

Link

Il nodo di tipo *Link* sono dei particolari *Decorator* utilizzati in un BT come puntatore per eseguire un altro BT al posto di un nodo. Il nome dell'albero eseguito è scritto sotto il nome del nodo. Il success e failure di un nodo *Link* è basato sul BT a cui punta. Questo nodo garantisce quindi l'abilità di realizzare BT modulari che supportano diversi tipi di comportamento e permettono il riutilizzo di parti di comportamento. Essendo un tipo di decoratore speciale, il suo simbolo è leggermente diverso e viene rappresentato in Figura 3.3.e.

3.3 Esecuzione di un BT

Il funzionamento di un BT propaga il tick generato dal nodo radice ai vari rami sottostanti, esaminando volta per volta il valore di ritorno dei singoli nodi. Un agente che utilizza un BT per il proprio processo decisionale avrà quindi un codice simile al seguente:

```
1 initialize(agent);
2 BT.parse(agent);
3 while active = true do
4 |   state ← Tick(Root);
5 |   sleep(1/ f_tick)
6 end
7 BT.delete(agent);
8 return 0;
```

Algorithm 6: Main di un agente che utilizza i BT

Un classico esempio per capire il funzionamento della valutazione del BT è simulare il BT presentanto da Millington & Funge (2009), [28] illustrato in figura 2.3. Questo esempio può essere utilizzato per valutare l'apertura di una porta sotto determinate condizioni utilizzando un BT, in particolare mettendo in risalto la modularità tipica di questo strumento. Le uniche azioni implementate sono *Move* e *Barge Door*, ma *Move* viene utilizzata tre volte nell'albero con tre diversi luoghi in cui muoversi. Si noti quindi come i nodi *Actions* non devono esser modificati in base alla loro posizione nell'albero.

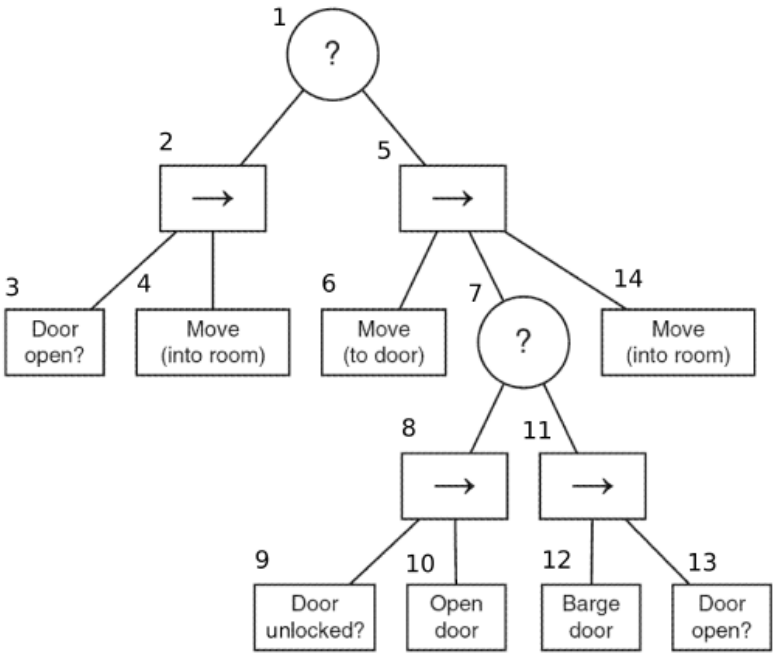

Figura 3.4: Esempio di BT

Assumiamo che i nodi Action dell'albero siano di tipo bloccante.

1. Supponiamo che nel caso che andremo a valutare la porta sia aperta. Inizia l'esecuzione dal nodo radice e si scende verso il primo nodo a sinistra. Poichè il nodo (2) è di tipo Sequence, il tick si propaga al proprio figlio sinistro (3), di tipo Condition. Quest'ultimo valuta la condizione ambientale e ritorna il valore Success al padre. Il nodo (2) quindi prosegue con la sua esecuzione propagando il tick all'azione (4), al termine della quale verrà generato un altro Success. Dato che non sono presenti altri figli e tutti quelli relativi al nodo (2) hanno valutato Success, la Sequence ritorna Success al genitore, primo nodo dell'albero, che terminerà il tick attuale.

2. Supponiamo ora che la porta sia chiusa ma sbloccata. L'esecuzione parte dalla Radice e si scende al nodo (3) che, date le condizioni dell'ambiente, ritornerà Failure a (2); essendo una Sequence l'esecuzione viene subito interrotta e ritorna Failure al genitore, ovvero al nodo Selector (1). Questa volta l'esecuzione non termina ma si scende al figlio (5) e (6) che valuterà Success dopo essersi spostato verso la porta. Il valore ritorna a (5) che fa proseguire l'esecuzione fino a (9). Essendo la

porta sbloccata, (9) ritorna Success e passa il controllo ad (8) e quindi (10). Una volta aperta la porta (10) ritorna Success a (8) che ritorna a (7) che quindi fa ritornare immediatamente Success a (5) che fa proseguire la Sequence a (14). Al termine dell'azione viene generato Success che risale l'albero fino alla radice terminando il tick.

3. Nel caso in cui la porta sia chiusa e bloccata l'esecuzione è simile, solo che al nodo (9) verrà generato il valore Failure e l'esecuzione si sposta al nodo (12). Se l'azione produce esito positivo e riesce quindi ad aprire la porta l'esecuzione procede a (14), al termine della quale ritornerà alla radice il valore Success.

Questa procedura esecutiva è la differenza principale tra i BT e i Decision Tree. I DT solitamente attraversano l'albero dal nodo radice ad uno dei nodi foglia tramite una sequenza di scelte binarie che non si propagano indietro nell'albero. I BT invece contengono dei nodi decisionali che effettuano delle decisioni composte e il processo decisionale è trasversale all'albero stesso e si può muovere a diverse profondità e attraverso diversi rami. Questa mobilità lungo l'albero consente di esprimere lo stesso comportamento grazie ad una piccola ripetizione delle sole azioni, senza dover riscrivere completamente le scelte decisionali da intraprendere.

3.4 Considerazioni

3.4.1 Vantaggi

I BT arrivano al proprio obiettivo scomponendolo in task più piccoli. Questo stile di programmazione facilita la generazione di comportamenti goal-based, in contrasto con l'approccio che utilizza le FSM: il focus lì era mappare l'insieme degli stati e transizioni, e il progettista doveva forzare la mappatura per condurre l'agente allo stato desiderato. Con i BT invece questo non avviene, e sopratutto si evita il problema dello *state explosion* tipico delle FSM.

La soluzione gerarchica delle HFSM non riesce a risolvere completamente il problema in quanto le transizioni tra i vari stati devono essere esplicitamente descritte, quindi per riutilizzare un particolare stato in un'altra area del comportamento bisogna effettuare nuovamente una descrizione esplicita delle transizioni. I BT invece descrivono implicitamente il comportamento in una struttura ad albero e non sulle transizioni tra gli stati, rendendoli modulari, riutilizzabili e sopratutto facilmente scalabili.

Poichè il BT valuta i propri rami secondo un criterio di priorità definito dalla struttura dell'albero, vengono ereditati facilmente diversi metodi per realizzare lo stesso comportamento. Se uno di questi fallisce, semplicemente si passa al successivo seguendo la successione di singoli comportamenti, realizzando una forma di *contingency planning*.
I BT rappresentano una forma molto facile di planning, ma poichè non considerano esplicitamente gli effetti futuri delle azioni correnti basandosi solo su un insieme di comportamenti prestabiliti, ricade nella definizione di *Reactive Planner*. Un **Reactive Planner** o **Dynamic Planner** non è un progettista nel senso tradizionale del termine, in quanto i sistemi reattivi non contengono modelli dell'ambiente nè considerano gli effetti futuri delle azioni. Sono invece basati un un insieme predefinito di azioni basate sullo stato attuale e sul goal da raggiungere. Poichè i BT non considerano esplicitamente l'effetto delle azioni ma osservano solo lo stato corrente, non richiedono un modello dell'ambiente in cui sono immersi e sono quindi molto semplici, con un'ottima capacità espressiva e computazionalmente efficienti da valutare, **rendendoli perfetti per la gestione dei processi comportamentali di piccole piattoforme robotiche**.

3.4.2 Limitazioni

Anche se tutti i comportamenti realizzati tramite FSM possono essere mappati in un BT, uno svantaggio attuale è derivato dal fatto che la gestione di eventi non è implementata naturalmente a differenza di quello che accade con le FSM *event-driven*. Una via per ridurre questo problema è quello di includere una cache interna per immagazzinare questi eventi in modo che il BT li gestisca in qualche tick futuro allo stesso modo in cui gestisce un input di un sensore. Alternativamente, un event handler può essere progettato per chiamare un BT per la gestione di eventi concorrenti al comportamento principale.
Recentemente è stata proposta un'estensione dei BT che gestiscono anche gli eventi utilizzando dei Decoratori realizzati appositamente per questo scopo. [30]

3.5 Lavori correlati

I due campi principali in cui sono già stati implementati i BT sono quelli relativi ai sistemi di intrattenimento, in particolare video-games, dal quale è stato originato questo metodo, e in robotica.
Nel primo caso lo scopo è quello di cercare nuove ed efficenti modalità di implementere l'AI per i Non-Player Characters (NPC). Questo spesso porta a dei framework per i BT che sono tipicamente game-oriented, che pur mettendo in luce alcune interessanti caratteristiche, non possono essere generalizzate in altri campi. Ad esempio prendendo in considerazione le emozioni provate dagli NPC per aumentare la capacità di immersione del giocatore [36], applicandoli ad alcuni giochi commerciali come DEFCON [31], HALO 2 (2005)[42] e FEAR(2005)[43], per poi applcarli ad altre tipologie quali Real Time Strategy game RTS[32] o in alcune competition internazionali quali la Mario AI Competition.[40]
Altri invece sono partiti da questi ultimi per introdurre nuove funzionalità, come ad esempio abilitando le query [37], effettuando delle parametrizzazioni sui vari lati [38] o scegliendo il ramo con una funzione di probabilità [39].
Nel caso invece della robotica, si è iniziato a parlarne con il paper che ha proposto l'utilizzo dei BT come alternativa ai sistemi di controllo dinamico CDS [26], e sono stati successivamente applicati all'authoring [41], per i CSP [44], per il controllo di UAV [25] e recentemente per le self-driving car [45].

Capitolo 4

Architettura BT

Per poter utilizzare i Behavior Tree nelle applicazioni, è necessario realizzare una struttura scalabile che permetta al programmatore di utilizzare questo strumento senza dover ogni volta creare una propria versione e facilitando quindi la portabilità del codice.
La mancanza di uno standard si ripercuote sulle applicazioni già esistenti, che pur basandosi sul formalismo introdotto da Marzinotto[26] ed illustrato nel capitolo 3 non arrivano a convergere su un'architettura comune, rendendo impossibile una migrazione di codice da un ambiente ad un altro e dovendo interpretare il lavoro altrui.
In questo capitolo viene quindi proposta un architettura per i linguaggi Object Oriented che possa essere la più generale possibile in modo da poter essere utilizzata in ogni ambito, mantenendo le proprietà di modularità e scalabilità specifiche dei BT. In rete sono presenti diversi esempi e consigli implementativi [33] [34] [35] ma in ciascuno di essi emergono problemi e criticità di varia natura che ne impediscono l'applicazione generale. Quest'architettura proposta invece risolve i problemi di sicurezza emersi garantendo al contempo le caratteristiche di modularità e semplicità implementativa tipica dei BT, e per questo vuole porsi come standar architetturale per tutti i lavori futuri. In figura 4.1 è possibile vedere lo schema proposto, spiegato nel dettaglio nelle sezioni successive.

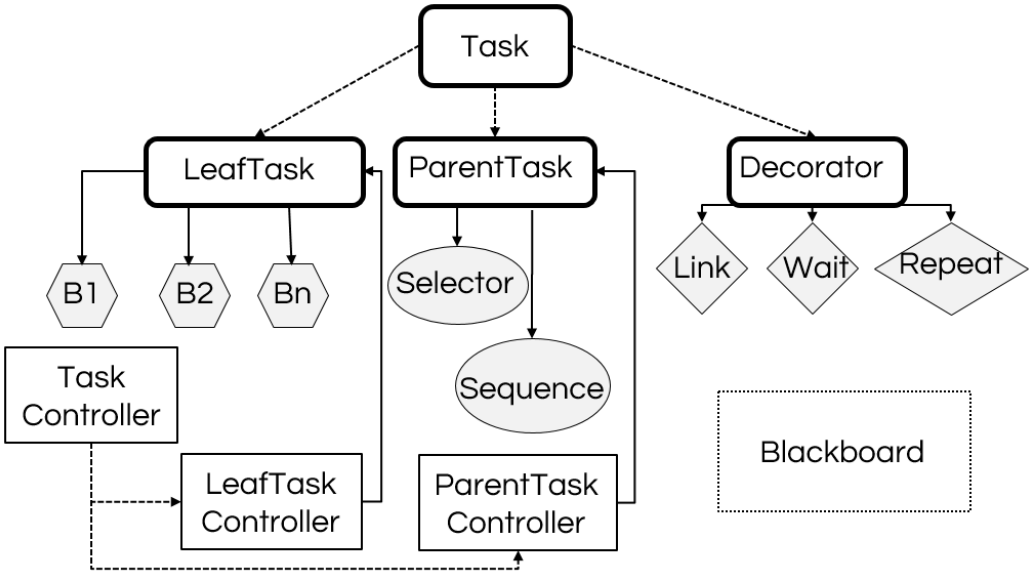

Figura 4.1: Architettura di un'applicazione basata sui BT

4.1 Struttura generale

L'idea generale del sistema Behaviour Tree based è quella di avere una serie di task o comportamenti basilari (MoveTo, OpenDoor ecc) e combinarli per formare un BT. Per riuscire a fare questo bisogna dividere le varie aree di interesse specifiche, raggruppando funzioni comuni e fornendo chiarezza per la divisione dei lavori. Per fare questo lavoro è stato utilizzato il *Design Pattern Composite*. Come linguaggio Oo per gli esempi sotto riportati è stato utilizzato Java.

La classe base **Task** fornisce un'interfaccia per tutti questi task:

- Le foglie dell'albero sono composte dai singoli comportamenti (*Actions & Conditions*).
- I loro genitori sono i nodi interni responsabili della decisione del prossimo nodo da eseguire(*Composite*).
- L'ultima classe è quella legata ai *Decorator* che non rientrano nel processo decisionale ma solo nell'alterazione di alcune proprietà di quest'ultimo. I Decorators infatti sono task che "decorano" un'altra classe fornendo ulteriore logica, ad esempio aumentando la velocità di esecuzione di certi task, resettando dei parametri al termine dell'esecuzione, forzando un attesa ecc.

Queste tre categorie estendono quindi la classe astratta Task, creando quindi *LeafTask*, *ParentTask* e *Decorator*. All'interno di ciascuna di queste vengono poi vengono realizzati i vari comportamenti, indicati in figura 4.1 da *B1,B2,Bn* per le Actions, garantendo libertà al programmatore di estendere la struttura includendo nuovi nodi interni o Decorator realizzati per l'applicazione particolare.

I vari task hanno solo la logica strettamente necessaria alla propria esecuzione; tutta la logica decisionale del tipo "il task è in esecuzione", "il task necessita di un aggiornamento", "si è concluso con successo" sono raggruppati nella classe **TaskController**, ed aggiunti per composizione. Il motivo di questa scelta architetturale risiede in due aspetti fondamentali:

1. *Facilità di lettura del comportamento e scrittura del codice*: tutta la gestione dello stato interno del nodo viene svincolata dall'azione stessa, ovvero quando diventa necessario realizzare una nuova azione, l'attenzione dev'essere focalizzata solo nella stesura del comportamento, non nella gestione dello stesso; quest'ultima dev'essere garantita in egual misura a tutti i nodi dell'albero tramite delle procedure standard di accesso.

2. *Sicurezza*: introdurre delle scelte ad-hoc di controllo per ogni singolo nodo vuol dire aumentare il numero di zone potenzialmente vulnerabili, mettendo così a rischio la struttura del sistema. Con l'introduzione di un controllore esterno tutto il lavoro di gestione dei nodi passa per questo elemento, aumentando la stabilità del sistema stesso.

L'ultima classe presente nell'architettura è **Blackboard**, utilizzata come mezzo di comunicazione per i vari Task. La sua funzione è paragonabile ad un database di conoscenze per i task foglie, dove risiedono quindi i dati utili alle task foglie, generati dalle stesse ed utilizzate per l'aggiornamento od il funzionamento del comportamento.

Un esempio per capire il funzionamento di questa classe è il seguente: supponiamo di voler realizzare all'interno di un sistema multiagente il processo decisionale legato al Follow, ovvero un agente decide di seguirne un altro. Per fare ciò necessiterà di alcune azioni che andranno a realizzare il proprio BT, quali ad esempio scegliere chi seguire, muoversi, evitare le collisioni ecc. Queste azioni risiedono nella struttura del BT che si occupa di effettuare le azioni corrette al momento giusto. In ogni azione però saranno necessarie delle informazioni addizionali volte all'esecuzione dell'azione stessa, in questo caso:

- TargetToFollow: un dato che indica quale altro agente seguire.
- MyPosition: dove mi trovo
- TargetToMove: dove devo andare
- NextStep: qual'è la prossima posizione utile

Queste informazioni potrebbero essere incluse in ogni agente, ma richiederebbe una comunicazione continua tra gli stessi che all'aumentare delle dimensioni del sistema si rivelerebbe critica. Per questo è stato deciso di spostarle in una classe comune per facilitare la scrittura e la consultazione dei vari componenti del sistema.

Presentiamo quindi ora nel dettaglio le varie classi, i vari metodi che devono essere implementati e un esempio applicativo può essere consultato nella sezione 4.6.

4.2 Task

Un codice esemplificativo è riportato qui sotto. In ogni nodo di un BT sono necessarie due funzioni, *CheckConditions* e *Execute*, per verificare se si può aggiornare il nodo e per aggiornarlo effettivamente. Le funzioni *Start* ed *End* sono chiamate subito prima di aggiornare il nodo e al termine della logica del nodo stesso. La classe è di tipo astratto in quanto è già stato inserito un elemento di tipo *Blackboard* in modo da poterlo richiamare immediatamente.

```
public abstract class Task {

        protected Blackboard bb;

        /**
         * Creates a new instance of the Task class
         *
         * @param blackboard
         *        Reference to the AI Blackboard data
         */
        public Task(Blackboard blackboard) {
                this.bb = blackboard;
        }
```

```
        /**
         * Override to do a pre-conditions check.
         *
         * @return True if it can, False if it can't
         */
        public abstract boolean CheckConditions();

        /**
         * Override to add startup logic to the task
         */
        public abstract void Start();

        /**
         * Override to add ending logic to the task
         */
        public abstract void End();

        /**
         * Override to specify task's logic
         */
        public abstract void Execute();

        /**
         * Override to specify task's controller
         *
         * @return The specific task controller.
         */
        public abstract TaskController GetControl();
}
```

4.2.1 LeafTask

La classe *LeafTask* estende Task, introducendo la gestione del TaskController volto a supportare la logica della classe. Non possiede altri particolari metodi o logica decisionale in quanto quest'ultima viene lasciata all'implementazione dei figli di LeafTask. Questi ultimi rappresentano le singole Action o Condition del sistema, sono modulari e derivano tutti da questa classe generale. Come spiegato prima, se i singoli comportamenti devono comunicare tra loro, utilizzano l'oggetto Blackboard per condividere i dati.

```
public abstract class LeafTask extends Task {
        /**
         * Task controller to keep track of the state.
         */
        protected TaskController control;

        /**
         * Creates a new instance of the LeafTask class
         *
         * @param blackboard
         *        Reference to the AI Blackboard data
         */
        public LeafTask(Blackboard blackboard) {
                super(blackboard);
                CreateController();
        }

        /**
         * Creates the controller for the class
         */
        private void CreateController() {
                this.control = new TaskController(this);
        }

        /**
         * Gets the controller reference.
         */
        @Override
        public TaskController GetControl() {
                return this.control;}
}
```

Ora che sono stati esibiti tutti i vari metodi, un nodo foglia estende LeafTask ed implementa se necessario questi ultimi. Riportiamo di seguito un esempio autoesplicativo, legato al Behavior SetDestination, dove la destinazione viene scelta a caso. La valutazione successiva, come ad esempio vedere se la destinazione è libera, se è raggiungibile ecc, viene lasciata ad altri nodi. Come si vede, i singoli metodi realizzati nelle classi astratte precedenti vengono qui utilizzati per la creazione del comportamento definitivo.

```
public class SetDestinationTask extends LeafTask {

Random rng = new Random();
/**
 * Creates a new instance of SetDestinationTask class
 *
 * @param blackboard
 *              Reference of the AI Blackboard data
 */
public SetDestinationTask(Blackboard blackboard){
        super(blackboard);}

/**
 * Check of needed data for the operations
 */
@Override
public boolean CheckConditions() {
        return bb.destination != null;}

/**
 * Select the destination to set
 */
@Override
public void Execute(){
        int x = rng.nextInt() %10;
        int y = rng.nextInt() %10;
        bb.destination.Set(x,y);
        GetControl().FinishWithSuccess();}

/**
 * Ends the task
 */
@Override
public void End() {
        System.out.println("The new destination is: "
                + bb.destination.getX()
                + " ,"
                + bb.destination.getY());

/**
```

```
 * Starts the task
 */
@Override
public void Start() {
        System.out.println("Starting");}

}
```

4.2.2 ParentTask

In maniera analoga, *ParentTask* utilizza una classe di supporto *ParentTaskController* per gestire la logica interna, ad esempio se il nodo ha terminato la propria esecuzione, se è ancora in stato Running ecc, ma la caratteristica principale che li differenzia dai precedenti è la gestione di quale figlio eseguire, in accordo al funzionamento dei nodi interni di un Behavior Tree.
I metodi *Start* ed *End* sono analoghi a quelli visti in *LeafTask*, ma la funzione *Execute* non esibisce la struttura di un comportamento ma ha come obiettivo la selezione del figlio successivo da eseguire. In particolare dovrà:

```
1  Controllare in maniera safe la presenza di tutti i dati necessari
   all'esecuzione delle condizioni e del comportamento, gestendo gli
   eventuali null.;
2  if il nodo foglia corrente non è ancora in esecuzione then
3  |  avviare il suo metodo corrispettivo
4  else
5  |  if il nodo foglia ha completato la propria esecuzione then
6  |  |  valutare il valore di ritorno Success o Failure e decidere quale
   |  |  nodo eseguire successivamente
7  |  else
8  |  |  il figlio dev'essere aggiornato, quindi viene chiamata la funzione
   |  |  Execute
9  |  end
10 end
```

Per chiarire ulteriormente conviene portare alla luce lo sviluppo di due sottoclassi importanti, *Sequence* e *Selector*. In entrambe saranno presenti due metodi aggiuntivi, *ChildSuceeded* e *ChildFailed*, utilizzate per la valutazione del nodo decisionale.

4.2.2.1 Sequence

In un nodo *Sequence*, l'esecuzione inizia sempre dal primo figlio; se termina con *Success* si seleziona il successivo. Se non ci sono altri figli successori, allora tutti i figli hanno terminato con successo e si ritorna Success; se invece uno dei figli termina la propria esecuzione con un valore *Failure*, è necessario interrompere l'esecuzione del genitore *(ParentTask)*. Nel codice seguente si può vedere un'esecuzione della classe *Sequence*; le chiamate di ChildSucceded fanno riferimento a valori derivanti dal nodo controller.

```
public class Sequence extends ParentTask {
/**
 * Creates a new instance of the Sequence class
 *
 * @param blackboard
 *            Reference to the AI Blackboard data
 */
public Sequence(Blackboard blackboard) {
        super(blackboard);
}

/**
 * A child finished with Failure.
 * Failed to update the whole sequence.
 * control is the TaskController for the parent
 */
@Override
public void ChildFailed() {
        control.FinishWithFailure();
}

/**
 * A child has finished with success.
 * Select the next one to update.
 * If it's the last, we have finished with success
 * Else check the next one and it's condition
 */
@Override
public void ChildSucceeded() {
        int curPos = control.gainIndex();
        if (curPos == last) {
```

```
                control.FinishWithSuccess();}
        else {
                curTask = next();
                if (!curTask.CheckConditions())
                        control.FinishWithFailure();
                else {curTask.Execute();}}
        }
}
```

4.2.2.2 Selector

In un nodo *Selector*, l'esecuzione inizia dal primo figlio; se ritorna il valore *Success* si blocca l'esecuzione del *Selector* e si ritorna *Success*, altrimenti si seleziona il successivo fino a quando non sono stati esauriti tutti i figli disponibili, ritornando in quel caso *Failure*. La struttura a livello di codice ricalca la precedente, modificando solamente l'esecuzione del metodo *ChildFailed()*.

```
public class Selector extends ParentTask {
...
/**
* If a child finish with failure
*        find the new one to update;
* Fail if the child was the last one.
*/
@Override
public void ChildFailed() {
        next = NextTask();
        if (next == null) {
                control.FinishWithFailure();
        }
}

/**
 * Chooses the next task to update.
 *
 * @return Task if possibile or null otherwise
 */
public Task NextTask() {
        Task task = null;
        boolean found = false;
        int curPos = control.gainIndex();
```

```
        while (!found) {
                if (curPos == last) {
                        found = true;
                        task = null;
                        break;
                }

                curPos++;

                task = control.taskAt(curPos);
                if (task.CheckConditions()) {
                        found = true;
                }
        }

        return task;
}
...
}
```

4.3 Decorator

La classe *Decorator* è utilizzata per mappare il corrispettivo nodo dei Behavior Tree, integrando nuove funzionalità ai singoli nodi, siano questi singole azioni o scelte decisionali; in entrambi i casi viene effettuato l'override del metodo Execute, mantenendo inalterati gli altri metodi.
Un esempio di un Decorator può essere la classe Restart che ha il compito di far ricominciare un'azione ogni volta che è finita.

```
/**
 * Decorator that set to "Started" the task
 * it is applied to each time it finishes.
 *
 */
public class RestartDecorator extends DecoratorTask{
/**
 * Creates a new instance of the ResetDecorator class
 * @param blackboard Reference to the AI Blackboard data
 * @param task Task to decorate
 */
public RestartDecorator(Blackboard blackboard,Task task){
        super(blackboard, task);
}

/**
 * Does the decorated task's action;
 * if it's done, resets it.
 */
@Override
public void Execute()
{
        this.task.Execute();
        if(this.task.GetControl().Finished())
        {
                this.task.GetControl().Reset();
        }
}
...
```

4.4 TaskController

Il **TaskController** ha lo scopo di tenere traccia dello stato interno del *Task* che sta monitornado. L'utilità è quella di poter rispondere alle varie interrogazioni del tipo *"E' in corso? Dev'essere aggiornata? Ha terminato l'esecuzione? Qual'è il suo valore di ritorno?"*.
Sono quindi necessari dei valori standard per mantenere correttamente la traccia dello stato esecutivo ed alcuni metodi opzionali per ulteriori funzionalità di controllo

```
/**
 * Class added to any task to keep track
 * of the Task state and logic flow.
 *
 */
public class TaskController {
/**
 * Indicates if the task is finished or not
 */
private boolean done;

/**
 * If finished, it indicates
 * if it has finished with success or not
 */
private boolean sucess;

/**
 * Indicates if the task has started or not
 */
private boolean started;

/**
 * Reference to the task we monitor
 */
private Task task;

/**
 * Creates a new instance of the TaskController class
 *
 * @param task
```

```
 *          Task to control.
 */
public TaskController(Task task) {
        SetTask(task);
        Initialize();
}

/**
 * Initializes the class data
 */
private void Initialize() {
        this.done = false;
        this.sucess = true;
        this.started = false;
}

/**
 * Sets the task reference
 *
 * @param task
 *          Task to monitor
 */
public void SetTask(Task task) {
        this.task = task;
}

/**
 * Starts the monitored class
 */
public void SafeStart() {
        this.started = true;
        task.Start();
}

/**
 * Ends the monitored task
 */
public void SafeEnd() {
        this.done = false;
        this.started = false;
        task.End();
```

```
}

/**
 * Ends the monitored class, with success
 */
protected void FinishWithSuccess() {
        this.sucess = true;
        this.done = true;
}

/**
 * Ends the monitored class, with failure
 */
protected void FinishWithFailure() {
        this.sucess = false;
        this.done = true;
}

/**
 * Indicates whether the task finished successfully
 *
 * @return True if it did, false if it didn't
 */
public boolean Succeeded() {
        return this.sucess;
}

/**
 * Indicates whether the task finished with failure
 *
 * @return True if it did, false if it didn't
 */
public boolean Failed() {
        return !this.sucess;
}

/**
 * Indicates whether the task finished
 *
 * @return True if it did, false if it didn't
 */
```

```
public boolean Finished() {
        return this.done;
}

/**
 * Indicates whether the class has started or not
 *
 * @return True if it has, false if it hasn't
 */
public boolean Started() {
        return this.started;
}

/**
 * Marks the class as just started.
 */
public void Reset() {
        this.done = false;
}

}
```

4.5 Blackboard

Per quanto riguarda la classe **Blackboard**, questa purtroppo non presenta architetture particolari e simili alle varie applicazioni; dev'essere realizzata ad-hoc per ogni utilizzo, mantenendo tutti i dati necessari per la comunicazione dei vari comportamenti degli agenti.

4.6 Esempio applicativo - la classe Behavior Tree

Ora che tutta la struttura architetturale è stata definita, rimane l'ultimo elemento da esaminare, ovvero la classe **Behavior Tree**. Quest'ultima è responsabile di legare tutti i singoli elementi architetturali e realizzare così l'albero comportamentale. Di seguito viene riportato un esempio di un BT per la generazione del movimento di un agente. In Figura 4.2 c'è l'albero con i due task generici, mentre in Figura 4.3 c'è il sottoalbero specifico di *MoveToNextTask*. Sotto invece il codice per legare tra loro i vari elementi nella classe Behavior Tree.

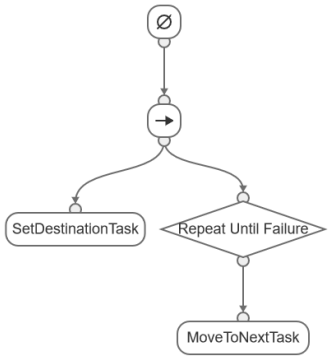

Figura 4.2: Esempio di BT

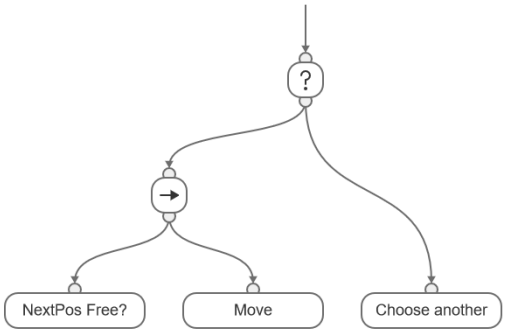

Figura 4.3: Sottoalbero MoveToNextTask

```
public class BehaviorTree{
/**
 * Root task of the behavior tree for the AI
 */
private Task planner;

/**
 * Information blackboard for the AI
 */
private Blackboard blackboard;

/**
 * Creates a new instance of the BehaviorTree
 */
public BehaviorTree(){
        blackboard = new Blackboard();
        CreateBehaviourTree();
}

/**
 * Creates the BT and populates the node hierarchy
 */
private void CreateBehaviourTree(){

//HERE GOES ALL BEHAVIOR
//Planner needed to start BT:
this.planner = new Sequence(blackboard, "Planner");

//Create moveToNextTask and add it to Decorator
Task moveToNextTask = new Selector(blackboard,
         "Subtree MoveToNextTask");
moveToNextTask = new RepeatUntilFailure(blackboard,
         moveToNextTask,"Repeat moveToNext if failure");

//Create subTask selectWhere
Task selectWhere = new Sequence(blackboard,
        "Select where to move");
((ParentTaskController)selectWhere.GetControl()).
        Add(new NextPosFreeTask(blackboard,
                "NextPosFree?"));
```

```
((ParentTaskController)selectWhere.GetControl()).
        Add(new MoveTask(blackboard,
                "Move"));

//Add selectWhere to MoveToNextTask completing subtree
((ParentTaskController)moveToNextTask.GetControl())
        .Add(selectWhere);
((ParentTaskController)moveToNextTask.GetControl())
        .Add(new ChooseAnotherTask(blackboard, "Move"));

//create BT adding SetDestinationTask
// and MoveToNextTask behaviors
((ParentTaskController)this.planner.GetControl()).
        Add(new SetDestinationTask(blackboard,
                                   "SetDestinationTask"));
((ParentTaskController)this.planner.GetControl()).
                           Add(MoveToNextTask);

}

/**
 * Start logic.
 */
public void Start(){
        this.planner.GetControl().SafeStart();
}

/**
 * Update the BT
 */
public void Update() {
        this.planner.Execute();
}

}
```

Capitolo 5

Applicazione BT in ambito MAS IoT

In questo capitolo si partirà dalla struttura generale dell'IoT introdotta da Singh e Copra [46] e verranno applicati i BT al livello della scelta decisionale.

5.1 Strutturare l'IoT come MAS

L'avvento dell'**Internet of Things (IoT)** ha introdotto delle problematiche che richiedono nuove astrazioni e tecniche di calcolo distribuito. L'idea principale dell'IoT è quella di connettere tutti gli oggetti ad Internet, in modo da poter essere monitorati e controllati, esaminando le informazioni prodotte ed utilizzandole per migliorare il processo di scelta decisionale.
Al momento, per realizzare un'applicazione di IoT (da adesso in poi chiamata *Iota*) bisogna considerare tre aspetti fondamentali:

- come raccogliere le informazioni
- come effettuare il processing
- come decidere l'azione da intraprendere.

In questo elaborato utilizzeremo la struttura dei BT precedentemente introdotta nei capitoli 3 e 4 per operare nel terzo aspetto qui indicato,la scelta decisionale, trascurando quindi la gestione dei livelli sottostanti.

Si osservi inoltre un aspetto cruciale delle Iota, che molto spesso viene trascurato: il loro valore aggiunto non è legato alla mera analisi e monitoraggio dei dati ottenuti, ma nella parte successiva, ovvero i servizi che si basano su questi dati. Un esempio può essere l'ambito sanitario: un Iota in

quest'ambito non ha lo scopo di collezionare ed analizzare i dati dei pazienti, ma di fornire ai medici, pazienti e alle loro famiglie dei servizi più informati e che quindi possano aumentare l'efficacia delle terapie stesse. L'aspetto decisionale deve poter agire anche in questo ambito, effettuando un'elaborazione sui dati e fornendo un risultato anche agli utilizzatori finali del sistema.

Alcune delle astrazioni e tecniche architetturali di una Iota fanno riferimento ad un *sistema multi agente (MAS)*, in particolare se riferito all'intelligenza artificiale. L'idea è quella di considerare un sistema IoT come un MAS definito come un insieme di agenti che interagiscono tra loro, ognuno dei quali è considerato una parte autonoma del sistema capace di prendere decisioni basandosi solo sul proprio bagaglio informativo o in seguito ad una decisione comunitaria di più alto livello. Questa astrazione ad alto livello viene utilizzata per gestire anche un gruppo di agenti eterogenei ma che necessitano di un coordinamento.

La ricerca in questo ambito tuttavia pone problematiche se messa in relazione al moderno calcolo distribuito. Alcuni lavori infatti estendono solamente l'astrazione del paradigma del singolo agente *belief-intentions-goals* al sistema multiagente, assumendo però come ipotesi una prospettiva unitaria, omogenea e con la conoscenza totale e perfetta delle informazioni, ipotesi non verificati in alcuni campi applicativi. Altri lavori invece si basano su un repository globale di stati del sistema, rendendo effettivamente il sistema centralizzato ma trascurando l'effetto dei singoli agenti, portando ad una struttura di tipo master-slave.

Per superare queste problematiche ed integrare un un'intelligenza distribuita è stato proposto un sistema multiagente decentralizzato costituito da un insieme di agenti eterogenei ed autonomi. Questo modello ha permesso di partire per analizzare le problematiche di coordinazione e di scelte decisionali, tematiche aperte e rilevanti nell'ambito di IoT. [47]

5.2 Aspetti rilevanti dell'IoT

Richiamiamo brevemente alcuni degli aspetti rilevanti legati all'IoT utilizzati per i nostri scopi attuali.

5.2.1 Utilizzo di diverse tecnologie comtemporaneamente

L'IoT si basa su una confluenza di tecnologie diverse, prime fra tutte quelle a basso livello. Esistono moltissime tipologie di sensori, capaci di ottenere non solo i dati comuni quali temperatura, pressione o posizione, ma anche alcuni specifici per l'applicazione in esame, quali ad esempio componenti chimici (ozono, anidride carbonica, idrocarburi), vibrazioni, salinità e molti altri. Questi sensori sono intrinsecamente limitati dalla banda supportata per inviare dati e ricevere istruzioni e dalla durata della batteria se non collegati ad una fonte energetica. Pur essendo l'IoT un fenomeno moderno, la tecnologia a supporto è stata già sviluppata per anni: esiste una sinergia naturale con il mobile computing, e molte delle tecnologie a basso consumo utilizzate per quell'ambito possono essere trasportate nell'ambito IoT. In un tipico utilizzo, il nodo IoT lavora in una modalità a basso consumo energetico, e le tecniche di mobile computing ad alto consumo sono invece utilizzate per ricevere comandi ed inviare dati tra i vari noti dell'IoT; in questo modo un tradizionale nodo di mobile computing serve come gateway per i dispositivi IoT. Un esempio applicativo è quello legati ai dispositivi indossabili, che formano una rete legata al singolo e gestita dallo smartphone dell'utente, che colleziona ed invia le informazioni ai vari dispositivi.

5.2.2 Applicazioni derivate dall'IoT e Participatory Sensing

Ciò che rende interessante l'IoT non è la tecnologia in sé, quanto le applicazioni che ne derivano. Queste applicazioni possono avere domini applicativi molto differenti, quali ad esempio la manifattura, l'infrastruttura civile, i sistemi energetici, la sanità e l'uso personale. Praticamente qualsiasi elemento può essere un *"thing"* - equipaggiamenti industriali, automobili, strade, anche gli animali e gli esseri umani stessi. L'approccio tipico è quello di piazzare un sensore che permette la connessione ad una rete su qualcosa in modo da garantirgli l'accessibilità alla rete.
L'IoT permette di monitorare in maniera continua e costante le risorse in esame per ottenere dei dati su cui basare le scelte decisionali. Un esempio è

nella sanità, dove i dispositivi vengono utilizzati sia sui pazienti per monitorare i parametri vitali che sull'infrastruttura medica, al fine di migliorare il contesto o l'esperienza pubblica del singolo. La rivelazione di una zona ad alta presenza di ozono (smog) può suggerire alle persone presenti in quella zona di fare un controllo preventivo per evitare eventuali complicazioni asmatiche.
Un'altra famiglia di applicazioni è quelle legata al *participatory sensing*, nel quale gli utenti condividono le informazioni ottenute dai propri sensori [48]. Il Participatory sensing, utilizzato originariamente dagli utenti per condividere le funzionalità dei sensori presenti nei loro telefoni, si utilizza ora per il monitoraggio delle zone urbane ed ambienti naturali, potendo estendere queste informazioni ottenute dai singoli dispositivi in modo da coprire una scala maggiore. Ad esempio la localizzazione può essere utilizzata per esaminare il grado di congestione delle zone urbane.

Figura 5.1: Participatory sensing

L'IoT inoltre guadagna valore dalle tecnologie emergenti dall'interazione

umana; nello specifico, la realtà aumentata può permettere agli utenti di "navigare visivamente" il mondo dei "things" tramite tecniche di aumento delle informazioni basate sui dati ricavati dai sensori. Ad esempio, se stiamo monitorando una fabbrica, un'interfaccia a realtà aumentata può mostrare all'utente le temperature attuali delle varie macchine in uso e riuscire ad individuare in anticipo eventuali problematiche. Stesso discorso può essere fatto se esaminiamo lo stato di deterioramento delle varie macchine o l'ultima volta che sono state revisionate. Queste interfacce possono aiutare gli utenti ad effettuare scelte intelligenti senza sovraccaricare i sistemi di flussi di dati grezzi che da soli non riescono a fornire informazioni rilevanti, ma se messi in esame con gli altri si.

5.3 Architettura

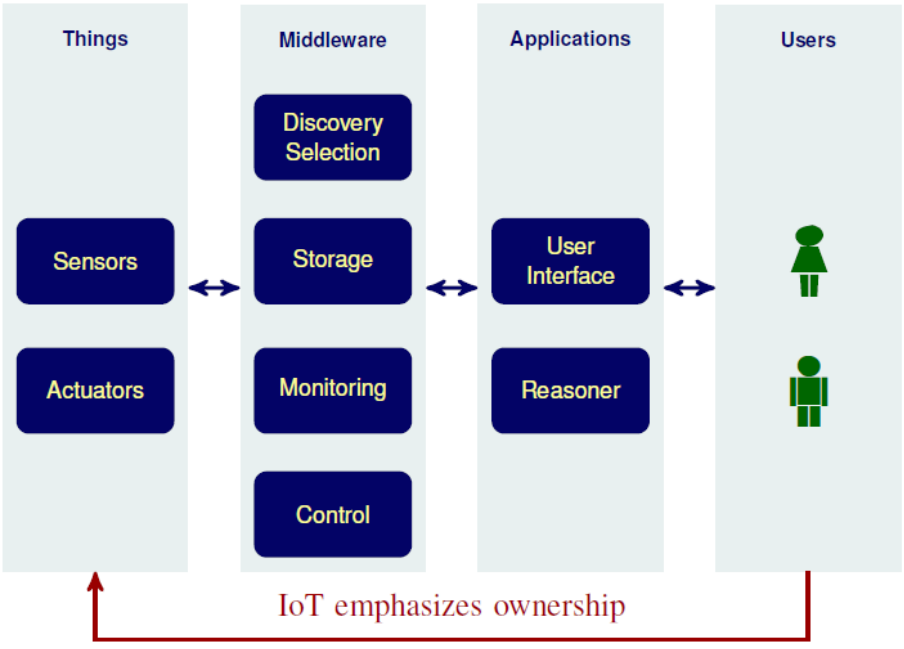

Figura 5.2: Architettura generale di un sistema IoT

La figura 5.2 illustra alcuni degli aspetti più importanti riguardanti l'architettura dell'IoT, in particolare i componenti principali sottostanti una IoTa. Da una parte sono presente i dispositivi legati strettamente al piano fisico dell'IoT, quali sensori ed attuatori. Dall'altra parte si trovano gli utenti finali dell'applicazione. Un aspetto fondamentale delle Iota è l'incorporazione della

sottile relazione tra utente e risorse, che ha un ruolo più incisivo in questo ambito rispetto alle tradizionali applicazioni web, in quanto molti spesso gli utenti finali delle Iota hanno una diretta relazione con le operazioni degli elementi fisici ed infrastrutturali , non solo per quanto riguarda le informazioni scambiate. In altri termini, una IotA consiste in una interfaccia utente e di un modulo decisionale, responsabile delle scelte prese dall'applicazione stessa.

Le Iota interagiscono con i dispositivi IoT non direttamente ma tramite un middleware particolarmente corposo. Alcuni di questi middleware forniscono astrazioni per l'IoT con funzioni di monitoraggio (per i sensori) e controllo (per gli attuatori). Altri invece riguardano il salvataggio dei dati ed alcune scelte di selezione del servizio, entrambe espresse di seguito.
Alcuni aspetti critici che possono rivelarsi dei colli di bottiglia per le applicazioni IoT sono le seguenti:

- Tecnologie di controllo dei dati, ad esempio lo spazio dove immagazzinarli, lo streaming e l'analitica di supporto. Poiché ogni nodo potenzialmente può produrre dati in continuazione, e si utilizzano un numero elevato di dispositivi, la gestione delle informazioni è un problema centrale. Ovviamente alcune applicazioni necessitano di un numero limitato di dispositivi, ma il volume dei dati assume comunque un'importanza da non trascurare.

- Ragionare basandosi su questi dati abbastanza velocemente da permettere di effettuare delle scelte intelligenti è un fattore chiave per l'efficacia delle applicazioni basati sull'IoT. Il miglioramento tecnologico e l'evoluzione algoritmica può aiutare in tal senso, ma proprio in virtù della grande mole di dati diventa insostenibile effettuare decisioni complesse in un singolo elemento. Per questo si sta iniziando ad utilizzare il cloud computing, ed è qui che i Behavior Tree possono migliorare la situazione.

- Tecnologie di semantica delle informazioni, ad esempio le correlazioni tra i dati, possono diventare la base per velocizzare la capacità di scelta decisionale e migliorare il significato ottenuto dal flusso di dati.

- Lo sforzo ingegneristico per mantenere le IoT al passo con l'evoluzione dei dispositivi e degli elementi accessori è notevole, e necessità di strumenti di supporto.

Come possiamo capire da questi punti fondamentali, le scelte devono essere rapide e basate su astrazioni di dati recenti onde evitare di effettuare

scelte obsolete per lo stato attuale del sistema. Per l'alto volume di dati, le scelte basate solo sulle informazioni provenienti dal sensore stesso possono essere effettuate in loco, ma quelle basate sull'analisi di tutti i sensori non è possibile farlo su un dispositivo di tipo embedded, ma dev'essere supportato da un agente esterno.

5.4 Aspetti decisionali di un IoT MAS

Attualmente, una Iota è realizzata da un singolo centro di controllo unitario. Utilizzeremo il termine **agente** per queste singole macchine in modo da mettere in luce l'aspetto del ragionamento ad alto livello. Essenzialmente, l'agente gestisce le decisioni come specificato dal proprio progettista, implementando un qualche tipo di loop *sense-reason-act*. L'agente viene considerato unitario anche se alcuni componenti possono essere distribuiti e può risiedere sullo scalino pià alto di una infrastruttura decentralizzata. In letteratura, le applicazioni sono tipicamente modellate in questo modo, ovvero considerando una macchina sempre unitaria.

L'applicazione mostrata in figura 5.2, pur evidenziando le varie componenti, è oltremodo semplicistica. Ciò che realmente caratterizza una Iota e la distingue dalle altre applicazioni convenzionali è il fatto che tende ad includere diversi domini amministrativi o parti. In generale, ogni parte contribuisce con il bagaglio informativo di alcuni "things", utilizza l'informazione derivata da altri, e fornisce dei servizi a delle terze parti.
Il punto è che le varie parti comunicano tra loro sulla base di alcune regole di incontro, ovvero dei protocolli di interazione. Utilizzando questa dottrina, l'applicazione è specificata dalle interazioni tra le entità autonome, modellate come agenti, ovvero macchine unitarie (che possono esibire una struttura interna ricorsiva). Il protocollo di interazione infatti specifica il MAS catturando il processo sociale che le parti attivano tramite gli agenti. Nell'esempio sanitario, esistono delle regole che vanno dalla condivisione delle informazioni personali solo a terze parti autorizzate, dal tempo minimo di soggiorno dopo esser stati sottoposti ad un intervento e così via.
Specificare un'applicazione in termini di protocolli è la chiave della decentralizzazione. Ogni parte che gioca un ruolo nel protocollo può implementare il proprio agente per automatizzare il processo decisionale e facilitare l'interazione con le altre parti. Seguendo il metodo appena citato, modelliamo una Iota come un singolo agente; dal punto di vista delle interazioni, una Iota è caratterizzata dal protocollo di interazione nel quale gli agenti rappresentano le varie parti del mondo reale, ciascuna con il proprio ruolo. Come mostra

la figura 5.3, emerge un'architettura multi agente, dove ogni singolo agente rappresenta una parte autonoma e contiene uno strumento per le scelte decisionali.

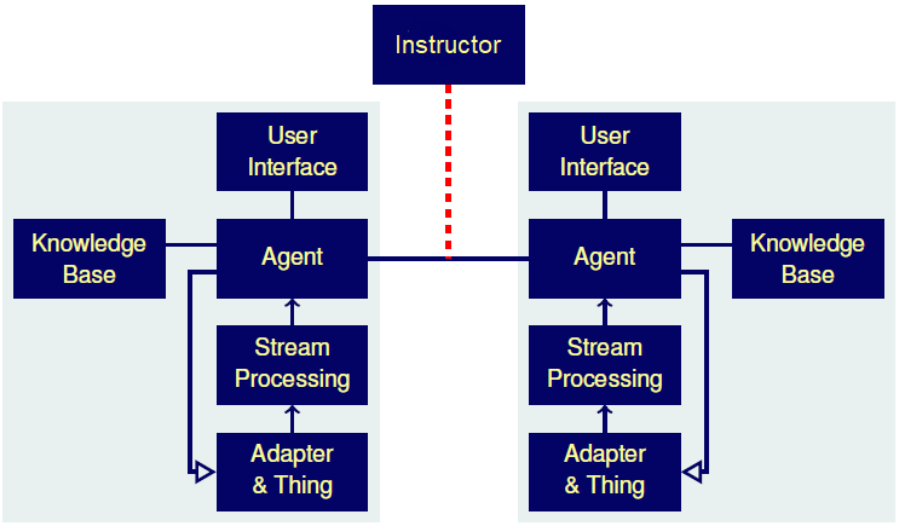

Figura 5.3: Architettura generale di un sistema IoT

La decentralizzazione richiede che gli agenti comunichino tra di loro. Il protocollo di comunicazione agisce proprio a questo livello, gestendo lo scambio di messaggi tra gli agenti. La scelta ed il controllo derivante da questo protocollo è uno dei temi aperti attuali nella computazione distribuita [49] [50], in quanto i messaggi possono essere persi, ritardati, duplicati per una ritrasmissione e così via. Questi aspetti possono portare gli agenti ad operare in stati incompatibili a livello di sistema globale, provocando il fallimento del protocollo a livello dell'interoperabilità.
Per poter gestire eventuali incongruenze è necessario introdurre un elemento super-partes chiamato **Instructor**, illustrato nella parte alta della Figura 5.3, che fornisce delle funzioni di controllo aggiuntive al sistema, quali ad esempio garantire l'autonomia delle singole parti rispettando tuttavia la consistenza, modificare il comportamento dei partecipanti del sistema nel caso in cui avvengono delle modifiche a livello superiore (quali l'aggiunta di una funzionalità a livello di progettazione) oppure fornire dei servizi a livelli sottostanti implementando anche dei concetti di responsabilità e tracciabilità ai vari agenti. [51]

In questo lavoro di tesi non ci concentreremo sul protocollo di comunicazione tra gli agenti ma sul processo decisionale dei singoli e di come modellare il comportamento a fronte di una modifica imposta dai livelli superiori.

5.5 Architettura di un MAS IoT usando Behavior Tree

Come specificato nelle sezioni precedenti, le due principali entità di questo modello sono gli *agenti*, ovvero entità che rappresentano i singoli dispositivi o gruppi di dispositivi, e l'*Instructor*, che è un agente di controllo di un livello più elevato. Entrambi questi elementi possono veder rappresentato il proprio modello decisionale come behavior tree, permettendo la scalabilità e la gestione degli aspetti di cambiamento decisionale.

Per quanto riguarda i singoli agenti, il Behavior Tree viene lasciato libero nell'implementazione in quanto dipendente dal dominio dell'applicazione stessa. Deve comunque poter gestire un eventuale modifica al proprio comportamento dettata dall'Instructor. Una possibile implementazione è mostrata in Figura 5.4.

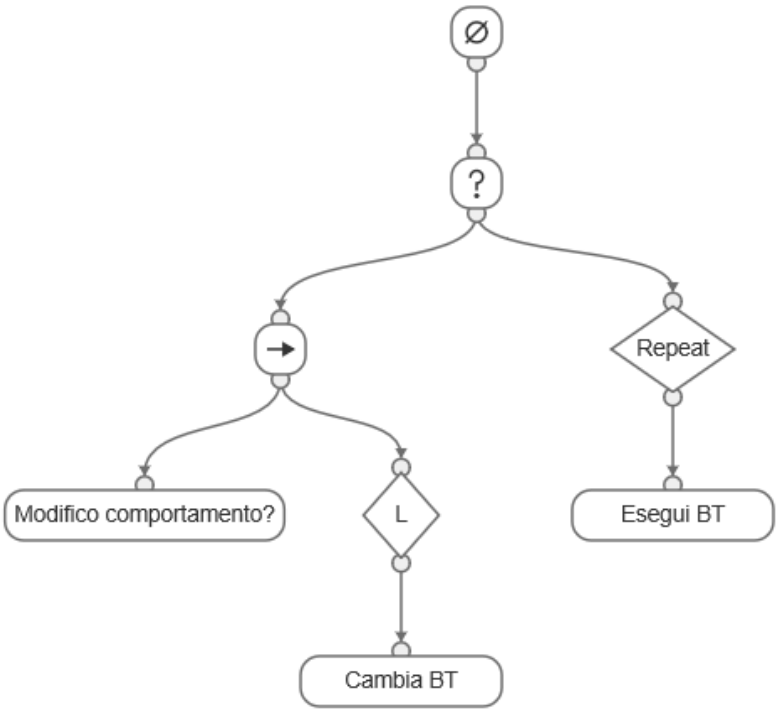

Figura 5.4: BT di un agente del sistema MAS IoT

Per quanto riguarda l'Instructor, il suo funzionamento dipende dalle informazioni che elabora , ad esempio che arrivano da un'applicazione di controllo di livello superiore, e dai servizi che mette a disposizione ai livelli inferiori. Nel momento in cui deve imporre delle modifiche al funzionamento agli agenti sottostanti, lo sviluppo del proprio albero decisionale prosegue entrando nel ramo apposito.
Ogni coordinatore deve quindi avere almeno un comportamento volto alla modifica degli agenti che questi comporta, più altri comportamenti per i vari servizi che questi mette a disposizione, come ad esempio la richiesta di tracciabilità delle operazioni sottostanti ecc. Un possibile BT è quindi quello rappresentato in figura 5.5:

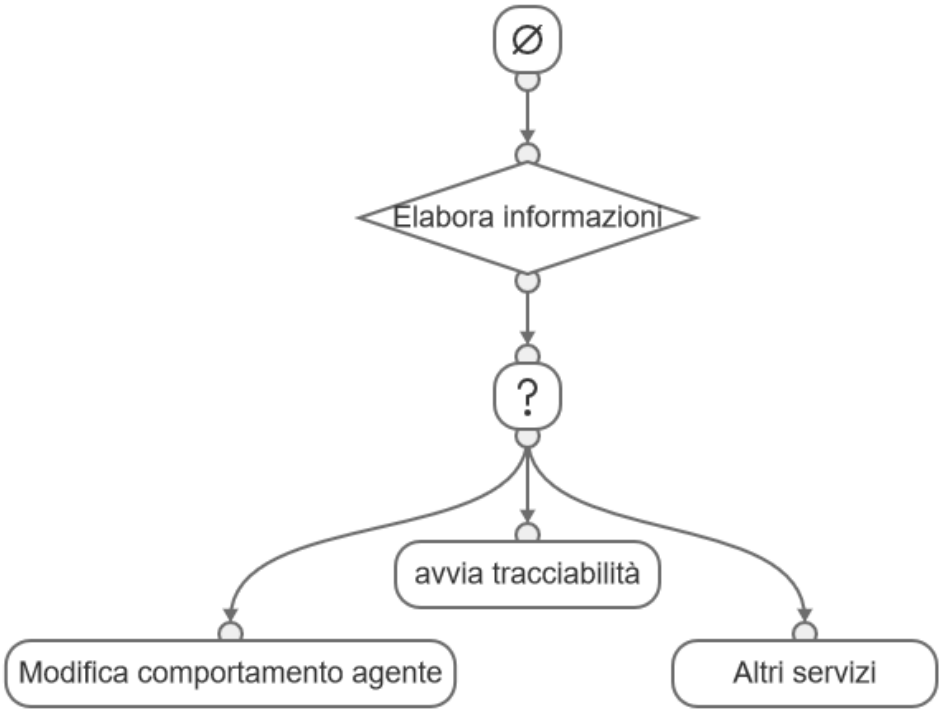

Figura 5.5: BT del coordinatore del sistema MAS IoT

Per quanto riguarda la composizione architetturale, si noti come per la natura intrinseca dei sistemi multi agente, i coordinatori stessi sono essere coordinati a loro volta, trattandoli semplicemente come agenti ed estendendo il proprio BT come fatto per il BT dei singoli agenti.

Capitolo 6

Conclusioni

I Behavior Tree sono uno strumento molto potente per la realizzazione di un processo decisionale, sopratutto grazie alle sue proprietà di modularità e scalabilità che lo differenziano nettamente dalle alternative solitamente utilizzate. Lo scopo di questo lavoro di tesi era analizzare ed applicare questa tecnica nel processo decisionale in sistemi multiagente. Dopo aver evidenziato le varie alternative per la realizzazione del processo decisionale, l'attenzione è stata posta sui BT sia a livello formale che a livello pratico. E 'stato fornito un modello architetturale per la realizzazione di applicazioni basati su Behavior Tree principalmente per Java, ma anche per tutti i linguaggi di programmazione Object Oriented, ed è stato successivamente applicato in un campo in forte espansione quale quello relativo all'IoT.

L'analisi dei BT in ambiti alternativi alla robotica ed alla game industry dal quale sono stati originati si è conclusa con successo, evidenziando come questo metodo utile per effettuare le scelte decisionali possa essere esportato in un qualsiasi campo.
Il modello proposto si pone come base strutturale da cui partire per la realizzazione delle varie applicazioni scritte in linguaggio object oriented, facilitando l'implementazione ed evitando quindi un'interpretazione personale distinta ogni volta che dev'essere realizzata un'applicazione. In particolare si rivela essere molto utile per lo sviluppo di applicazioni IoT data la natura modulare dei sistemi stessi.

6.1 Sviluppi futuri

Il design modulare e facile da leggere derivante dalle proprietà intrinseche dei BT in quanto grafi direzionati porta naturalmente allo sviluppo di una **Graphical User Interface (GUI)** per fornire all'utente un metodo intuitivo ed interattivo per il design dei BT, fornendo magari delle funzioni ausiliari quali la fault detection o degli strumenti di debugging. Attualmente esiste un editor grafico open source con licenza MIT che realizza questo, chiamato *behevior3*, che tuttavia ha funzioni abbastanza limitate. I BT risultanti vengono esportati in formato JSON ma non viene fornito un tool per la lettura automatica e va sviluppato ad-hoc per ogni singola applicazione. In questa tesi era stato iniziato un lavoro di realizzazione di un tool grafico, ma per mancanza di tempo non è stato ultimato. L'estensione possibile sarebbe quello di fornire un export non solo del BT ma anche di uno *skeleton* di un programma, ovvero esportare da un BT direttamente le classi con dei metodi di abstract da implementare, togliendo al programmatore l'onere di ricreare ogni volta la struttura ma richiedendo solamente di completare alcuni semplici metodi relativi alle Actions e alle Conditions, mentre tutto il resto viene fatto in automatico.

La scelta implementativa utilizzata in questo lavoro è stata quella di utilizzare un BT classico, senza estensioni parametriche o probabilistiche. In diverse applicazioni reali tuttavia potrebbe essere necessario introdurre dei parametri per migliorare la scelta decisionale, evidenziando ad esempio quali comportamenti vengono effettuati più spesso e quindi che possano richiedere un'ottimizzazione sui singoli comportamenti, oppure per osservare l'effetto del rumore che può essere generato dall'applicazione pratica. In questi casi i BT possono essere estesi introducendo nuove funzionalità e supportati da alcuni algoritmi di Machine Learning per ottenere il valore dei parametri volti ad ottimizzare il lavoro dell'agente in vista del proprio goal. Si noti come l'introduzione di eventuali estensioni ai nodi interni non modifichi l'architettura di base proposta.

Si noti infine come in molte applicazioni l'utilizzo delle FSM è fondamentale e i BT devono essere visti come un'alternativa per determinate tipologie di utilizzo, non solo un modello che vada a sostituire quello già esistente. Sarebbe quindi interessante valutare uno studio di un modello composto tra FSM e i BT, con le transizioni tra gli stati gestiti a livello di FSM e gli stati interni costituiti da BT, combinando i punti di forza delle tue tipologie.

Bibliografia

[1] Jacak W., Pröll K., Multiagent based intelligent control and internal communication in an autonomous system, SCI'99 and ISAS'99, World multiconference on systemics, cybernetics and informatics, Proceedings Volume III, 1999.

[2] Yazed Al-S., Ajlan Al-A., Khalid A.,Abdullah B., The Development of Multi-Agent System using Finite State Machine, International Conference on New Trends in Information and Service Science, 2009.

[3] Jacak W., Pröll K., Dreiseitl S., Rozenblit J., Conflict Management in Multiagent Robotic System: FSM and Fuzzy Logic Approach, Proceedings of IEEE Intern. Conference System Man Cybernetics SMC'2001, Tucson, USA, 2001, pp. 1593-1598.

[4] P. Petrovi, "Evolving Behavior Coordination for Mobile Robots using Distributed Finite-State Automata," in Frontiers in Evolutionary Robotics, H. Iba, Ed. Intech, 2008, no. April, ch. 23, pp. 413–438.

[5] L. K¨onig, S. Mostaghim, and H. Schmeck, "Decentralized Evolution of Robotic Behavior using Finite State Machines," International Journal of Intelligent Computing and Cybernetics, 2009.

[6] J. Ferber, Multi agent systems, an introduction to distributed artificial intelligence, Addison Wesley/ Pearson, London, 1999.

[7] Alexis M., Nick M., George G., Intelligent Software Agents in Travel Reservation Systems, SURPRISE 97

[8] W. Jacak, K. Pröll - Multiagent Architecture for Intelligent Autonomous Systems - Proceedings of the IEEE 2nd International Symposium on Logistics and Industrial Informatics (Lindi 2009), Linz, Austria, 2009, pp. 114-120

[9] Cohen, P., Levesque, H., Communicative Actions for Artificial Agents, Proceedings of the First International Conference on Multiagent Systems, San Francisco, Cambridge, AAAI Press, 1995, pp. 65-72.

[10] Jacak W., Pröll K., Multiagent Approach to intelligent control of robot,Robotics and Applications RA99, IASTED International Conference,Santa Barbara, USA, 1999.

[11] Brenner W., Zarnekow R., Wittig H., Intelligent Software Agents,Springer-Verlag, 1998.

[12] Jacak W., Pröll K., Simulation based contract management in intelligent robotic agent system, Proceedings of International Mediterranean Modelling Multiconference I3M2007, Genoa, Italy, 2007.

[13] Pröll K., Intelligent Multi-Agent Robotic Systems: Contract and Conflict Management, PhD Thesis, Johannes Kepler University Linz, Austria,2002.

[14] Brenner W., Zarnekow R., Wittig H., Intelligent Software Agents,Springer-Verlag, 1998.

[15] Jacak W., Pröll K., Heuristic Approach to Conflict Problem Solving in an Intelligent Multiagent Systems, Lectue Notes in Computer Science,No. 4739, 2007, pp. 772-780.

[16] Russel S., Norvig P., Artificial Intelligence: A Modern Approach, 3rd Edition, 2009, Perason

[17] Nolfi, S. (2002, January). Power and the Limits of Reactive Agents. Neurocomputing, 42 (1-4), 119–145.

[18] Arkin, R. C. (1998). Behaviour-Based Robotics. MIT Press.

[19] Y. Hou, Y. Zeng, Y. Ong, A Memetic Multi-Agent Demonstration Learning Approach with Behavior Prediction, Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)

[20] Smith, J. F., III, & Nguyen, T. V. H. (2007). Fuzzy Decision Trees for Planning and Autonomous Control of a Coordinated Team of UAVs. Signal Processing, sensor fusion and target recognition XVI , 6567 656708-1 .

[21] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. Science of Computer programming 8 , 231-274.

[22] Millington, I., & Funge, J. (2009). Artificial Intelligence for Games (2nd ed.). Morgan Kaufmann.

[23] A. Johansson and P. Dell'Acqua, "Emotional Behavior Trees," in Computational Intelligence and Games, 2012 IEEE Conference on. IEEE, 2012, pp. 355–362.

[24] A. Shoulson, F. M. Garcia, M. Jones, R. Mead, and N. I. Badler," Parameterizing Behavior Trees," in Motion in Games, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 7060, pp. 144–155.

[25] P. O¨gren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," in AIAA Guidance, Navigation, and Control Conference. AIAA, 2012.

[26] A. Marzinotto, M. Colledanchise, C. Smith, P. O¨ gren, "Towards a Unified Behavior Trees Framework for Robot Control", in Robotics and Automation (ICRA), 2014 IEEE International Conference on , IEEE Robotics and Automation Society, 2014, 5420-5427 p.

[27] J. A. Bondy and U. S. R. Murty, Graph Theory with Applications Macmillan London, 1976, vol. 290.

[28] A. J. Champandard, "Behavior Trees for Next-Gen Game AI," 2007. [Online]: http://aigamedev.com/open/articles/behavior-trees-part1/

[29] Brooks, R. A. (1987), Planning is Just a Way of Avoiding Figuring Out What to do Next, Massachusetts Institute of Technology, Artificial Intellegence Laboratory.

[30] https://github.com/meniku/NPBehave

[31] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON," in Applications of Evolutionary Computation. Springer, 2010, vol. 6024, pp. 100–110.

[32] S. Delmer, "Behavior Trees for Hierarchical RTS AI," Diploma Thesis, Southern Methodist University (SMU), 2012.

[33] http://aigamedev.com/open/articles/bt-overview/

[34] http://aigamedev.com/open/articles/popular-behavior-tree-design/

[35] https://github.com/libgdx/gdx-ai/wiki/Behavior-Trees#behavior-tree-api

[36] A. Johansson and P. Dell'Acqua, "Emotional Behavior Trees," in Computational Intelligence and Games, 2012 IEEE Conference on. IEEE, 2012, pp. 355–362.

[37] G. Fl´orez-Puga, M. A. G´omez-Mart´ın, P. P. G´omez-Mart´ın, B. D´ıaz-Agudo, and P. A. Gonz´alez-Calero, "Query-Enabled Behavior Trees," Computational Intelligence and AI in Games, IEEE Transactions on,vol. 1, no. 4, pp. 298–308, 2009.

[38] A. Shoulson, F. M. Garcia, M. Jones, R. Mead, and N. I. Badler,"Parameterizing Behavior Trees," in Motion in Games, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 7060, pp. 144–155.

[39] R. Colvin, L. Grunske, and K. Winter, "Probabilistic Timed Behavior Trees," in Integrated Formal Methods, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4591, pp. 156–175.

[40] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution," in Applications of Evolutionary Computation, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6624, pp. 123–132.

[41] D. Becroft, J. Bassett, A. Mej´ıa, C. Rich, and C. L. Sidner, "AIPaint: A Sketch-Based Behavior Tree Authoring Tool," in AIIDE, 2011.

[42] Isla, D.: Managing complexity in the Halo 2 AI system. In: Proceedings of the Game Developers Conference (2005)

[43] Orkin, J.: Three states and a plan: the AI of FEAR. In: Proceedings of the Game Developers Conference (2006)

[44] R. J. Colvin and I. J. Hayes, "A semantics for Behavior Trees using CSP with specification commands," Science of Computer Programming,vol. 76, no. 10, pp. 891–914, 2011.

[45] M. Olsson, Behavior Trees for decision-making in autonomous driving, PhD Thesis, 2014

[46] M.P. Singh, A.K. Chopra, "The Internet of Things and Multiagent Systems: Decentralized Intelligence in Distributed Computing", 2017

IEEE 37th International Conference on Distributed Computing Systems (ICDCS). 5-8 June 2017

[47] D. Miorandi, S. Sicari, F. D. Pellegrini, and I. Chlamtac, “Internet of things: Vision, applications and research challenges,” Ad Hoc Networks,vol. 10, no. 7, pp. 1497–1516, 2012.

[48] T. C. King, Q. Liu, G. Polevoy, M. de Weerdt, V. Dignum, M. B. van Riemsdijk, and M. Warnier, “Request driven social sensing (Demonstration),”in Proceedings of the 13th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). Paris: IFAAMAS, May 2014, pp. 1651–1652.

[49] S. Basu, T. Bultan, and M. Ouederni, “Deciding choreography realizability,” in Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Philadelphia, PA, USA: ACM, 2012, pp. 191–202

[50] M. Carbone, K. Honda, N. Yoshida, R. Milner, and S. Ross-Talbot, “A theoretical basis of communication-centered concurrent programming,” October 2006, http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf.

[51] E. A. Mamdani and J. Pitt, “Responsible agent behavior: A distributed computing perspective,” IEEE Internet Computing, vol. 4, no. 5, pp. 27–31, Sep. 2000.