Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

Programmazione di Sistemi Embedded con Linguaggi ad Agenti: un Caso di Studio basato su Jason e Arduino

Relatore:
Prof.
Alessandro Ricci

Presentata da:
Francesco Cozzolino

Sessione II
Anno Accademico 2014/2015

PAROLE CHIAVE

Programmazione ad Agenti

Sistemi Embedded

Il linguaggio ad Agenti Jason

Architettura BDI

Arduino

Introduzione

Nel corso degli anni, il notevole aumento di complessità dei sistemi, ha richiesto lo sviluppo di nuovi paradigmi di programmazione e, grazie ad essi, è possibile esprimere livelli di astrazione sempre più elevati. Uno di questi è il paradigma ad agenti, attraverso il quale, a differenza del paradigma ad oggetti, introduce astrazioni di prima classe per rappresentare entità autonome, ossia entità attive che incapsulano un flusso di controllo logico autonomo. Tale paradigma è particolarmente utilizzato nell'ambito dell'intelligenza artificiale, per progettare entità e insiemi di entità in grado di svolgere autonomamente compiti di varia complessità.
L'obiettivo di questo elaborato è discutere l'applicazione del paradigma ad agenti, e dei linguaggi di programmazione ad agenti in particolare, all'ambito dei sistemi embedded. Un agente per definizione rappresenta un entità progettata per svolgere autonomamente uno o più compiti interagendo con l'ambiente in cui è situato, mediante percezioni ottenute da sensori (input) e azioni realizzate mediante attuatori (output). Oltre a ciò, un agente comunica con altri agenti mediante forme di scambio di messaggi. Queste caratteristiche lo rendono quindi particolarmente efficace per modellare in modo naturale la parte di controllo di sistemi embedded.
Questo elaborato ha lo scopo in particolare di esplorare e discutere l'uso di linguaggi ad agenti per la programmazione di sistemi embedded, prendendo come riferimento la piattaforma Jason, fra le tecnologie ad agenti più note e utilizzate nello stato dell'arte. Lato embedded, verrà considerata la piattaforma Arduino, fra le più note e utilizzate in ambito di prototipazione e

accademico.

L'elaborato è organizzato come segue: nel primo capitolo verranno presentati i concetti fondamentali del paradigma ad agenti.

Dopo aver analizzato i concetti del paradigma, la trattazione proseguirà mostrando i costrutti base della piattaforma Jason.

All'interno del terzo capitolo verrà data una definizione di ambiente, definendo inoltre i concetti fondamentali. L'ambiente è un concetto fondamentale nel paradigma ad agenti, poiché è in grado di interagire con esso. Come per gli agenti, verranno mostrati i costrutti base per lo sviluppo di un ambiente.

Verranno inoltre trattati due capitoli riguardanti i sistemi embedded e Arduino.

Per concludere la trattazione, attraverso la realizzazione di una libreria, verrà proposto un metodo per permettere l'applicazione del paradigma ad agenti sulla scheda di prototipazione Arduino.

Indice

Capitolo 1

Sistemi multi-agente

Con il progresso tecnologico e il continuo aumento di complessità dei sistemi, l'informatica si è sviluppata di pari passo, fornendo metodi e approcci per costruire tali sistemi.
La programmazione orientata agli oggetti è uno dei paradigmi di riferimento nella modellazione e progettazione del software, in quanto i sistemi hanno raggiunto una complessità tale che rendono difficoltoso l'utilizzo della programmazione procedurale. La programmazione ad oggetti consente di concepire il sistema da modellare in termini di entità, dotati di alcuni attributi e in grado di compiere azioni. Ciò permette di raggiungere un livello di astrazione efficace per catturare gli aspetti essenziali del dominio, in particolar modo gli aspetti strutturali. Nell'ambito dell'intelligenza artificiale, tale approccio non è soddisfacente e, grazie allo sviluppo tecnologico, è possibile modellare i sistemi utilizzando il paradigma ad agenti.
Nei paragrafi seguenti verranno esaminati i concetti di questo paradigma di programmazione.

1.1 Agente

Il termine agente è molto utilizzato di recente. Con questo termine si indica una qualsiasi entità in grado di percepire continuamente e in modo

autonomo, grazie all'utilizzo di sensori, le informazioni relative all'ambiente in cui si trova e di modificarlo attraverso l'utilizzo di attuatori. Un agente, oltre ad interagire con l'ambiente circostante, è in grado di comunicare con altri agenti. Un sistema formato da più agenti prende il nome di sistema multi-agente.
Nella figura seguente è possibile notare uno schema di interazione tra un agente e l'ambiente in cui si trova.

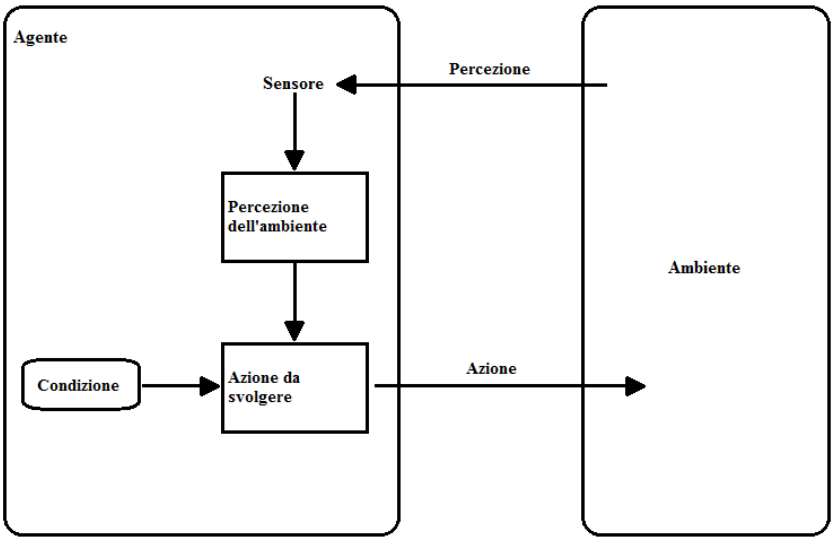

Figura 1.1: Interazione di un agente con l'ambiente

Come si evince dalla figura, tramite l'utilizzo di sensori, l'agente percepisce l'ambiente in cui è situato. Al verificarsi di una condizione (es: la temperatura atmosferica oltrepassa una certa soglia), tramite l'utilizzo di attuatori, può svolgere delle azioni che modificano l'ambiente.

1.2 Proprietà di un agente

Un agente, oltre a essere situato in un ambiente, deve possedere le seguenti proprietà: *autonomia*, *pro-attività*, *reattività* e *comunicazione*.

Autonomia

Un agente deve possedere la capacità di agire in maniera autonoma, ossia, deve essere in grado di prendere decisioni in base alle informazioni che possiede, in modo da raggiungere l'obiettivo prefissato.
I paradigmi di programmazione convenzionali sono sprovvisti di questa proprietà, in quanto, il normale flusso di esecuzione del programma è formato sostanzialmente da: input, elaborazione dei dati forniti in input e da un output del risultato dell'elaborazione. Tali costrutti non possono prendere delle decisioni in modo autonomo e alterare il proprio flusso di esecuzione. Un agente invece, è in grado di aggiornare le proprie conoscenze e, in base a tali informazioni, è in grado di determinare se continuare nel perseguire il proprio obiettivo oppure se demandarlo a un altro agente, in modo da poter perseguire un nuovo intento.

Pro-attività

Capacità di un agente di prendere iniziative per raggiungere l'obiettivo che gli è stato assegnato. Un agente è un'entità attiva che, tramite delle azioni, è in grado di completare un compito che gli è stato assegnato.

Reattività

Capacità di un agente di reagire in modo tempestivo ai cambiamenti dell'ambiente. Raramente il raggiungimento di un obiettivo procede senza intoppi, in quanto si possono presentare delle situazioni che compromettono il normale svolgimento delle azioni necessarie per il raggiungimento dell'obiettivo prefissato. Un agente deve essere in grado di reagire tempestivamente a un imprevisto, in modo tale da raggiungere il proprio scopo.

Comunicazione

Capacità di un agente di co-operare e co-ordinare le attività con altri agenti. Al fine di realizzare gli obiettivi preposti, un agente può aver bisogno

di comunicare con altri agenti. Come vedremo all'interno di questa trattazione, non è sufficiente che gli agenti comunichino con un semplice scambio di byte, ma la comunicazione deve avvenire a livello di conoscenza. Si vuole che gli agenti siano in grado di comunicare le proprie conoscenze, convinzioni e i propri piani. Un esempio di questi linguaggi di comunicazione sono KQML e FIPA-ACL.

1.3 Comunicazione tra agenti

Nell'ambito delle intelligenze artificiali, gli approcci tradizionali della comunicazione tra agenti, presuppongono che essi siano costruiti sulla base di un modello cognitivo; in particolar modo sul modello BDI. Come in qualsiasi metodo di comunicazione, è di fondamentale importanza che le parti coinvolte siano d'accordo sulla forma e sulla semantica dei messaggi scambiati. Per poter avviare una comunicazione in un sistema multi-agente, ciò non è sufficiente, poiché è necessario definire un'ontologia comune. Un'ontologia è una rete semantica di concetti appartenenti a un dato dominio, legati tra loro da particolari relazioni.
Nel seguente paragrafo vedremo due linguaggi utilizzati per la comunicazione in un sistema multi-agente: KQML e FIPA-ACL.

1.3.1 KQML

KQML (**K**nowledge **Q**uery and **M**anipulation **L**anguage) propone un numero ridotto di primitive, come le query e le tell. L'idea è che ogni primitiva possa dare un significato semantico, basato sull'effetto che ha avuto sulle conoscenze dell'agente. Nello specifico, un agente potrebbe inviare un tell contenente parti delle proprie conoscenze. L'agente destinatario che riceverà il tell, aggiungerà il contenuto del messaggio alle proprie conoscenze.
KQML è abbastanza restrittivo, poiché presuppone che gli agenti all'interno del sistema siano co-operativi e progettati dallo stesso progettista.
Di seguito è presente un esempio di un possibile scambio di messaggi tra due

agenti.

```
(ask-one                                    (a)
  :sender labrou
  :receiver laptop-center
  :content (CPU libretto50 ?processor)
  :onotlogy electronics
  :language  kf)
```

```
(tell                                       (b)
      :sender laptop-center
      :receiver labrou
      :content (CPU libretto50 pentium)
      :ontology electronics
      :language kf)
```

Figura 1.2: (a) Richiesta informazioni. (b) Messaggio di risposta

Come è possibile notare dalle figura, l'interpretazione del messaggio è abbastanza intuibile. Nella figura 1.2 (a), l'agente mittente richiede il tipo di CPU di un specifico computer. La parola chiave *ask-one* indica che la richiesta riguarda una sola informazione (Per dettagli consultare il paragrafo 2.4). I restanti campi rappresentano gli attributi del messaggio.
Gli attributi *:sender* e *:receiver* indicano il destinatario e il mittente del messaggio.
L'attributo *:content* è senz'altro l'attributo più importante del messaggio, poiché specifica il contenuto del messaggio.
L'attributo *:language* specifica in quale linguaggio è espresso il contenuto del messaggio.
Per concludere, l'attributo *:ontology* fornisce un contesto al messaggio.

1.3.2 FIPA-ACL

A differenza di KQML, FIPA-ACL (**F**oundation for **I**ntelligence **P**hysical **A**gents-**A**gent **C**ommunication **L**anguage) si pone come scopo quello di specificare una sintassi che permette di agevolare l'interoperabilità tra agenti creati da diversi progettisti. Inoltre specifica la semantica delle primitive e si basa sulla formalizzazione dei concetti cognitivi, come le consocenze e gli scopi degli agenti. FIPA-ACL si tratta di un protocollo che specifica la comunicazione su cinque livelli: *protocol*, *communicative act* (CA) , *messaging*, *content language*, *ontology*.
Protocol definisce le regole sociali per la struttura del dialogo tra due agenti. Communicative act definisce il tipo di comunicazione in corso. Per esempio

una richiesta è un tipo di CA.
Messaging definisce meta-informazioni riguardanti il messaggio, compreso l'agente destinatario, il mittente e il contenuto del messaggio.
Content Language definisce la grammatica e la semantica associata per esprimere il contenuto del messaggio.
Ontology definisce il contesto del messaggio.
Nella figura 1.4 è possibile notare un esempio di un messaggio FIPA-ACL.

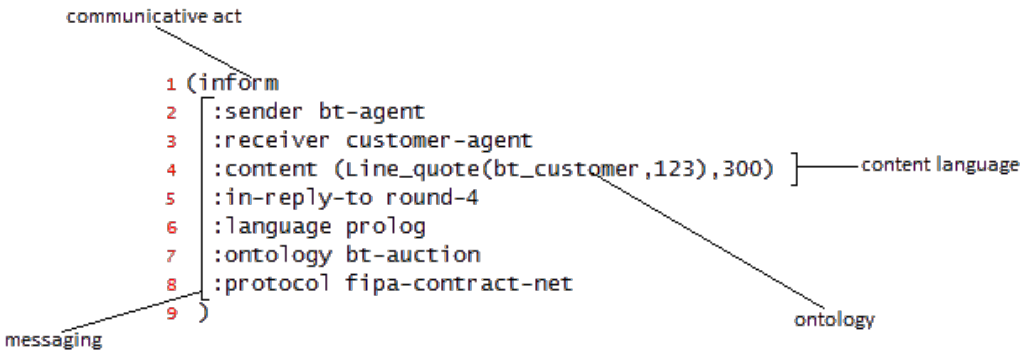

Figura 1.3: Struttura di un messaggio FIPA-ACL

1.4 Modello BDI

Come citato nel paragrafo 1.2, un agente deve possedere delle proprietà fondamentali. Tali proprietà permettono di concepire un agente come se fosse un'entità pensante. Grazie ad esse, un agente può essere accomunato a un essere umano.
Durante la metà degli anni 80', Michael Bratman concepì un modello del ragionamento umano, chiamato BDI (**B**eliefs-**D**esires-**I**ntentions).
Un agente, per poter percepire e relazionarsi con l'ambiente, necessita di un modello cognitivo. Tra i vari modelli, il più classico è il modello BDI.
I *Beliefs* sono le informazioni che l'agente ha sull'ambiente in cui è posto. Possono variare in base alle informazioni percepite dall'agente stesso o tramite la comunicazione con altri agenti.
I *Desires* sono i possibili obiettivi che un agente potrebbe perseguire. Avere

degli obiettivi non implica che un agente possa agire su di essi, ma si tratta di un fattore che può influenzare le azioni intraprese dall'agente. Infatti un agente per perseguire un obiettivo potrebbe scegliere di non perseguirne un altro, oppure, demandarlo a un altro agente.
I Desires che un agente intende perseguire, iniziando una serie di azioni, sono detti *Intentions*. Un agente può perseguire un obiettivo perchè gli è stato delegato oppure in seguito a una scelta.

1.4.1 Practical Reasoning

Il processo decisionale adottato dal modello BDI prende il nome di Practical Reasoning. Con il termine practical reason si intende la capacità di utilizzare la ragione per prendere una decisione su come agire. Dato che un agente può essere pensato come un'entità con uno stato mentale, simile a quello dell'uomo, si prenda in esempio la vita quotidiana di una persona. Quando un essere umano si trova di fronte a un problema, deve scegliere quale percorso intraprendere per risolverlo. La scelta su quale percorso intraprendere è guidata da un ragionamento. L'essere umano analizza i vari aspetti che formano il problema, considera le possibili soluzioni e, in base alle proprie capacità, concepirà una soluzione, intraprendendo una serie di azioni che lo porteranno alla risoluzione del problema. Il modello decisionale utilizzato dal modello BDI opera esattamente nello stesso modo.
Questo modello è composto da due attività: *deliberation* e *means-ends reasoning*.
Nell'attività di *deliberation*, hanno un ruolo di primaria importanza le intentions, poiché rappresentano la volontà di perseguire un obiettivo.
Si consideri ancora una volta il comportamento umano. Se una persona ha intenzione di perseguire uno scopo, cercherà di raggiungerlo svolgendo una serie di azioni. Nel caso in cui dovesse fallire, è lecito aspettarsi un nuovo tentativo da parte sua; fino a quando non sarà palese l'impossibilità di raggiungere l'obiettivo. Determinato un obiettivo, le capacità di ragionamento vengono offuscate, in quanto non vengono considerati in modo appropriato

gli scopi che sono incompatibili con l'obiettivo prefissato. Infine, un obiettivo è strettamente legato alle conoscenze che si hanno sul futuro. In particolare, fissatone uno, si pensa che in linea teorica e in condizioni normali, sia possibile avere successo nel raggiungimento del proprio obiettivo. Riassumendo, questa attività determina quale obiettivo perseguire, analizzandone la fattibilità in base alle proprie capacità.
Se l'attività di deliberation determina il *cosa* fare, l'attività di *means-ends reasoning* determina il *come* perseguire l'obiettivo prefissato; in base ai mezzi a disposizione. Nell'ambito delle intelligenze artificiali, il termine means-ends reasoning è conosciuto con il termine *Planner*. Per il raggiungimento di un obiettivo, un planner genera un piano composto da un corso di azioni, in base all'obiettivo da raggiungere, alle conoscenze dell'agente riguardanti l'ambiente e dalle azioni che può compiere un agente.

1.5 Confronto tra agenti e oggetti

Dopo aver fornito una definizione di agenti e mostrato le caratteristiche salienti di queste entità, si vuole concludere il capitolo con un confronto più dettagliato con il paradigma ad oggetti.
Gli agenti sono autonomi e hanno il controllo del proprio comportamento, mentre gli oggetti sono delle entità passive. Ad esempio, qualora venisse invocato un metodo di un oggetto, esso non avrebbe scelta e sarebbe costretto ad eseguirlo. Si potrebbe implementare il metodo in modo tale da non far nulla in determinati casi, ma un oggetto non possiede la capacità di rifiutare l'invocazione di un metodo.
Gli agenti sono più flessibili rispetto agli oggetti. Nella programmazione ad oggetti, una volta che un oggetto è stato creato, non potrà mai cambiare la classe di appartenenza. Inoltre, gli agenti possiedono la capacità di imparare e possono aggiungere o eliminare features dinamicamente.
Un oggetto può essere pensato come un organizzazione centralizzata, poiché le operazioni che implementa vengono invocate sotto il controllo di altri com-

ponenti del sistema. Un agente invece, può far parte di un sistema centralizzato o distribuito.
All'inizio del capitolo, il paradigma ad agenti è stato introdotto come una particolare esigenza per poter sviluppare determinati sistemi. E' sbagliato affermare che questo tipo di paradigma sia un sostituto o un evoluzione della programmazione orientata agli oggetti, nonostante permetta un livello di astrazione maggiore. Per concludere, questi due tipi di paradigmi sono nati con scopi differenti e, nella realizzazione di un sistema, l'utilizzo di una metodologia non esclude necessariamente l'altra.

Capitolo 2

Jason: un linguaggio ad agenti

Jason è un linguaggio di programmazione orientato agli agenti sviluppato in Java. Si tratta di un estensione di AgentSpeak, linguaggio basato sulla programmazione orientata agli agenti che utilizza il modello BDI. Jason implementa la semantica operativa di AgentSpeak e fornisce una piattaforma per lo sviluppo di sistemi multi-agente. All'interno di questo capitolo verranno definiti i costrutti base, fornendo alcuni esempi per illustrare nel modo più chiaro e dettagliato possibile i vari concetti di questo linguaggio di programmazione.
Quando si introduce un nuovo linguaggio di programmazione, è consuetudine mostrare un esempio; solitamente il classico "Hello World". Per quanto sia triviale, un esempio del genere aiuta il programmatore ad assimilare i primi costrutti del linguaggio di programmazione. Nella figura successiva è mostrato l' "Hello World" in Jason.

```
/* Initial beliefs and rules */

/* Initial goals */
!start.

/* Plans */

+!start
<- .print("hello world.").
```

Figura 2.1: Hello World in Jason

Come si può notare da questo esempio, la struttura di un agente è suddivisa in tre parti: beliefs iniziali, obiettivi, piani. Nei paragrafi successivi verranno analizzati nel dettaglio.
Sebbene in questo esempio non ce ne siano, è possibile impostare i beliefs iniziali dell'agente, ovvero le conoscenze riguardanti l'ambiente.
E' possibile notare come sia stato imposto all'agente di perseguire subito un obiettivo, tramite l'istruzione *!start*.
I piani sono i vari corsi di azioni che può intraprendere un agente per perseguire un obiettivo. Un agente, prima di applicare un piano, ne controlla la condizione. Nella figura 2.1 non vi è nessuna condizione nel piano *start*, di conseguenza potrà essere sempre scelto. Le istruzioni presenti all'interno di un piano rappresentano le azioni da eseguire.
Il simbolo '.' è un separatore sintattico, come il ';' in C o in Java.

2.1 Belief

Un agente possiede un insieme di beliefs, organizzati in una lista dinamica chiamata belief base. Come è già stato menzionato, un agente può rimuovere dei beliefs oppure, può ottenerli all'atto della creazione, dalla comunicazione con altri agenti e dall'ambiente in cui è situato. I beliefs sono collezioni di letterali che esprimono, sotto forma di predicati, proprietà individuali o di un oggetto. Ad esempio:

tall(john).

Il belief esprime una particolare proprietà. In questo caso indica che John è alto. Per rappresentare una relazione tra due o più oggetti è possibile utilizzare un predicato come:

likes(john,music).

Nella programmazione logica, è possibile trovare una sintassi di costruzione dei predicati simile a quella vista nei due esempi precedenti. Ad esempio, in Prolog, ogni sequenza di caratteri che inizia con una lettera minusco-

la, prende il nome di atom. Un atom è l'equivalente di una costante nella programmazione dichiarativa.

2.1.1 Annotations

Tramite l'utilizzo di un costruttno denominato annotation, Jason consente di arricchire il contenuto informativo dei beliefs. Dal punto di vista sintattico, un annotation è racchiusa tra parentesi quadre a seguito di un belief.

busy(john) [expires(autumn)].

L'esempio indica che John è occupato fino all'inizio dell'Autunno. Naturalmente scrivere quanto segue è del tutto equivalente:

busy_until(john,autumn).

Tuttavia, vi sono due vantaggi nell'utilizzo delle annotations. Il primo vantaggio è meramente estetico, in quanto l'utilizzo delle Annotations rendono la belief base più leggibile. Il secondo vantaggio semplifica la gestione della belief base, in quanto, un annotation potrebbe influenzare la durata di un belief nella belief base.

2.1.2 Strong negation

La negazione causa molte difficoltà nei linguaggi di programmazione logica.

In Jason vi sono due tipi di negazione: uno definito dall'operatore *not* e un altro tipo denominato *strong negation*.

Tramite l'operatore *not*, la valutazione di una formula negata, risulta falsa se e solo se, tramite le regole e i belief, l'agente non riesce a derivarla.

L'operatore *strong negation*, indicato dal simbolo "∼", è utilizzato per specificare che l'agente reputa falsa una determinata informazione. Ad esempio, l'agente assumerà che il belief *tall(john)* sia vero, mentre ∼*tall(john)* porterà l'agente a credere che John non sia alto.

2.1.3 Rules

Tramite le *Rules* è possibile definire le regole per derivare la condizione di verità di un predicato. Sono largamente utilizzate nei linguaggi di programmazione logici e, per chi ha familiarità con Prolog, non noterà molte differenze. Si consideri la seguente *Rule*:

likely_colour(C,B)

:- colour(C,B) [source(S)] & (S == self — S==percept).

La regola si può tradurre in linguaggio naturale nel seguente modo: "il miglior colore è quello dedotto dall'agente stesso o percepito dall'ambiente".
Passando all'analisi della sintassi, a sinistra dell'operatore ":-" può essere presente solo un letterale, mentre a destra sono presenti i predicati da derivare.

2.2 Goals

Come già ampiamente detto all'interno di questa trattazione, i beliefs esprimono le proprietà di un ambiente in cui l'agente è situato. I goals invece, possono essere definiti come proprietà riguardo lo stato dell'ambiente che l'agente intende perseguire.
In AgentSpeak vi sono due tipologie di goal: *achievement goals* e *test goals*.
L'*achievement goal*, detto anche *declarative goal*, è denotato dall'operatore '!'. Per esempio, *!own(house)* indica che l'agente ha l'obiettivo di credere di possedere una casa. Ne possiamo dedurre che al momento esso crede che own(house) non sia vero.
L'operatore '?' permette di utilizzare i *test goals* e consentono di recuperare informazioni dalla belief base. Per esempio, tramite *?bank_balance(BB)* si vuole istanziare la variabile *BB* con il valore del saldo bancario che l'agente in quel momento crede di avere.

2.3 Plans

L'esecuzione dei piani del modello BDI è importante, poiché rappresentano il know-how dell'agente; i beliefs e i goals ne determinano l'esecuzione. Un piano è composto da tre elementi fondamentali: *trigger event*, *context* e *body*.

Trigger event e *context* costituiscono insieme l'head del piano. Nella loro totalità sono sintatticamente separati dagli operatori ':' e '←'; come è possibile notare nel seguente esempio:

triggering_event: context ← *body.*

Di seguito verranno analizzate nel dettaglio le tre componenti di un piano.

2.3.1 Triggering event

Due delle proprietà più importanti di un agente sono senz'altro reattività e pro-attività.

Gli agenti hanno degli obiettivi da raggiungere che ne determinano il comportamento. Nonostante l'agente agisca in modo da perseguire un scopo, deve prestare attenzione ai cambiamenti dell'ambiente, poiché possono determinare il raggiungimento dell'obiettivo oppure se ne possono presentare di nuovi. L'evento scatenante può essere quindi causato da una modifica nell'ambiente oppure, da modifiche nell'assegnazione degli obiettivi.

Di seguito è mostrata una tabella con la sintassi di tutti i possibili eventi scatenanti.

Notation	Name
$+l$	Belief addition
$-l$	Belief deletion
$+!l$	Achievement-goal addition
$-!l$	Achievement-goal deletion
$+?l$	Test-goal addition
$-?l$	Test-goal deletion

Figura 2.2: Triggering events

Le prime due righe si riferiscono a cambiamenti nella belief base dell'agente. Ciò accade quando vi sono dei cambiamenti nella percezione dell'ambiente. Le restanti righe si riferiscono invece agli eventi generati in seguito a cambiamenti nei goals. Come visto nel paragrafo 2.2, i goals possono essere di tipo achievement o test.

2.3.2 Context

Il context di un piano è utilizzato per controllare se sussistono le condizioni per considerare il piano applicabile, basandosi sulla belief base dell'agente. Dal punto di vista sintattico, un context è formato da uno o più letterali congiunti tra loro, tramite gli operatori logici AND (identificato dal simbolo '&') e OR (identificato dal simbolo '|'). Sono inoltre ammessi gli operatori *not* e *strong negation* (identificato dal simbolo '∼'). Sebbene tali operatori siano stati descritti nel paragrafo 2.1.2, è necessario approfondirli ulteriormente per quanto riguarda il loro utilizzo nei context. Applicando l'operatore *not* ad un letterale, si sta verificando la sua assenza nella belief base. L'utilizzo della strong negation, invece, verifica la presenza del letterale e che l'agente creda che tale predicato sia falso. Nella figura 2.3 è possibile notare come impiegare i congiunti not e strong negation.

Syntax	Meaning
l	The agent believes *l* is true
~l	The agent believes *l* is false
not *l*	The agent does not believe *l* is true
not *~l*	The agent does not believe *l* is false

Figura 2.3: Operatori not e strong negation

Si consideri il seguente esempio per familiarizzare con le nozioni fornite fino ad ora:

+!buy(Something)
: not∼legal(Something) & price(Something,P)
& bank_balance(B) & B>P
← . . . .

Il piano può essere applicato se il predicato *legal(Something)* non è presente nella belief base dell'agente e il suo prezzo è inferiore al saldo del conto corrente.

2.3.3 Body

Il corpo di un piano consiste in una sequenza di azioni, ognuna separata dall'altra dal simbolo ';', che vengono eseguite quando il piano è applicato. Vi sono cinque tipologie di azioni che possono essere presenti nel corpo di un piano e sono: *actions*, *achievement goals*, *test goals*, *mental notes* e *internal actions*. Di seguito verranno analizzate le varie tipologie.

Actions

Se un agente è in grado di modificare l'ambiente in cui è situato, è per merito delle actions. Le actions rappresentano le abilità che un agente è in grado di compiere. In Jason, tale costrutto è definito sotto forma di predicato e prende il nome di *predicate ground*.
E' lecito chiedersi come faccia l'interprete a distinguere il significato dei predicati. La distinzione avviene grazie alla posizione all'interno del piano. Per esempio, i predicati all'interno del context, piuttosto che essere visti come actions, saranno confrontati con le beliefs dell'agente. I predicati all'interno del body saranno invece interpretati come actions.
Si consideri il seguente esempio: dato un robot dotato di due braccia, si vuole ruotare una delle due di un certo numero di gradi.
Il corpo del piano sarà composto semplicemente dal seguente predicato:

rotate(left_arm,45).

Come per gli oggetti, le actions eseguono operazioni e restituiscono un valore booleano che ne determina lo stato di esecuzione. Sebbene l'operazione restituisca true come risultato, non è detto che abbia effettivamente modificato l'ambiente, ma indica solo che essa è stata eseguita. Per verificare se l'ambiente è stato modificato, è necessario percepirlo tramite i sensori.
L'esecuzione del piano rimane in sospeso fino a quando l'action non termina.

Achievements goals

Per la realizzazione di agenti complessi, potrebbero non bastare delle semplici actions in sequenza da eseguire. Per esempio, un agente può eseguire alcune azioni in seguito al raggiungimento di un determinato obiettivo.
Nel corpo di un piano è possibile aggiungere dei goal. Si supponga di avere il corpo di un piano organizzato nel seguente modo:

action1;
!goal;
action2.

L'agente eseguirà la prima azione e al suo termine perseguirà un nuovo obiettivo. La seconda azione presente nell'esempio, non verrà eseguita fino a quando il nuovo obiettivo non verrà raggiunto. Pertanto, l'esecuzione del codice risulterà essere sequenziale. Questo comportamento potrebbe non essere sempre accettabile. AgentSpeak prevede un operatore per eseguire in concorrenza diverse parti di un piano; l'operatore '!!'. Riscrivendo l'esempio precedente, utilizzando l'operatore '!!' si ottiene:

action1;
!!goal;
action2.

Tramite l'utilizzo di tale operatore, è possibile eseguire *action2* anche se l'obiettivo *goal* non è ancora stato raggiunto.

Test goals

I test goals consentono di reperire le informazioni dalla belief base e sono tipicamente utilizzati prima di eseguire determinate azioni all'interno di un piano. Si potrebbe utilizzare il context di un piano per verificare una belief di un agente, ma potrebbe essere necessario ottenere le informazioni solo quando esse siano realmente disponibili o dopo delle modifiche. Per esempio, se un agente possiede delle informazioni riguardanti le coordinate di un bersaglio, ed esse sono continuamente aggiornate, ed è necessario reperire le informazioni riguardanti la posizione prima dell'esecuzione di una determi-

nata azione, ha più senso utilizzare *?coords(Target,X,Y)* prima dell'azione piuttosto che nel context del piano.
Se un test goals dovesse fallire, fallirebbe l'intero piano di esecuzione. L'interprete Jason mette a disposizione un meccanismo per la gestione dei fallimenti dei piani. Infatti, una volta fallito il test goal, l'interprete proverà a creare un evento di tipo test-goal addition. Se il progettista ha implementato un piano per poter gestire questa situazione, l'esecuzione proseguirà, altrimenti l'esecuzione del piano si può considerare fallita.

Mental notes

I belief provenienti dall'ambiente in cui è posto l'agente vengono automaticamente aggiunti da Jason e non richiedono l'intervento esplicito del programmatore. Tuttavia, durante l'esecuzione di un piano, potrebbe essere necessario aggiungere o rimuovere dei belief. Questi belief prendono il nome di mental notes e sono contraddistinti dall'annotazione *[source(self)]*. Per poter aggiungere o rimuovere i mental notes dalla belief base, si utilizzano gli operatori '+' e '-'. Si considerino i seguenti esempi:

+handled(yes) [source(self)];
-to_do(clean,kitchen) [source(self)];
-+current_targets(NumTargets) [source(self)];

Si presti particolare attenzione all'ultimo belief. L'operatore '-+' può essere utilizzato per rimuovere una belief (se esistente) durante l'aggiunta di una nuova belief. Utilizzare l'operatore '-+' è l'equivalente di:

-current_targets(NumTargets) [source(self)];
+current_targets(NumTargets) [source(self)];

Internal actions

Le actions si possono catalogare come delle azioni eseguite dall'agente che modificano l'ambiente in cui è immerso. Le internal actions sono delle azioni che non modificano in alcun modo l'ambiente esterno all'agente, ma

avvengono solamente all'interno dello stato mentale dell'agente. Dal punto di vista sintattico, un internal actions si contraddistingue da un actions grazie al simbolo '.', anteposto al nome dell'azione. Il carattere '.' è utilizzato per separare il nome della libreria dal nome dell'internal action. Si consideri il seguente esempio:

.print(?Hello world!?);

dip.rotate(left_arm,45);

Jason fornisce una serie di internal actions predefinite che vengono espresse senza l'utilizzo della libreria.

2.4 Comunicazione ed interazione in Jason

Nel primo capitolo si è discusso della comunicazione tra agenti all'interno di un sistema e sono stati trattati due linguaggi: KQML e FIPA-ACL. All'interno di questo paragrafo vederemo come la comunicazione avviene in Jason.

Jason utilizza KQML per permettere la comunicazione tra agenti e mette a disposizione una serie di internal actions per favorirla. La sintassi generale per l'invio di un messaggio è la seguente:

.send(receiver, illoc_force, propositional_content);

Il parametro *receiver* contiene il nome dell'agente destinatario presente nel file di configurazione del sistema. Può contenere anche una lista di nomi, in modo da inviare un messaggio multicast a un gruppo di agenti. Il parametro *propositional_content* può contenere una collezione di letterali o dei triggering event. Con il parametro *illoc_force* si indicano le intenzioni del mittente definite anche come *performative*.

Di seguito è fornito un elenco delle performative messe a disposizione da Jason. Con il carattere 'S' si denota il mittente e con il carattere 'R' il destinatario.

tell S vuole che R creda che la collezione di letterali contenuta nel mes-

saggio sia vera;

untell: S vuole che R non creda che la collezione di letterali contenuta nel messaggio sia vera;

achieve: S chiede a R di iniziare a perseguire un obiettivo quando la collezione di letterali contenuta nel messaggio è vera;

unachieve: S richiede a R di non perseguire l'obiettivo quando la collezione di letterali contenuta nel messaggio è vera;

askOne: S vuole sapere da R se il contenuto di un messaggio è vero;

askAll: S vuole tutte le possibili risposte da R riguardanti una domanda;

tellHow: S fornisce informazioni ad R su un piano;

untellHow: S richiede a R di eliminare un determinato piano dalla libreria;

askHow: S richiede a R quali dei suoi piani siano rilevanti per il trigger event contenuto nel messaggio.

Tramite il parametro *receiver* è possibile specificare un elenco di destinatari. Qualora si volesse inviare un messaggio broadcast, anziché utilizzare la primitiva *.send* e specificare nel parametro *receiver* tutti gli agenti del sistema, Jason fornisce la seguente internal action:

.broadcast(illocutionary_force,propositional_content);

Gli agenti possono ignorare i messaggi che ricevono. Ogni volta che un agente accetta un messaggio, viene generato un evento gestito dai piani predefiniti di AgentSpeak.

2.4.1 Scambio di informazioni

Tramite le performative tell e untell, un agente può modificare le beliefs dell'agente destinatario. Naturalmente se l'agente mittente invia un messaggio contenente la performative tell, implicitamente il mittente crede che sia vero. Di seguito alcuni esempi sull'utilizzo delle due performative.

.send(r,tell,open(left_door));
Il belief *open(left_door) [source(s)]* verrà aggiunto alla belief base dell'agente "r". Una volta ricevuto il messaggio, l'agente crederà che *open(left_door)* sia vero.

.send([r1,r2],tell,open(left_door);
Il belief *open(left_door) [source(s)]* verrà aggiunto alla belief base degli agente "r1" e "r2". Analogamente all'esempio precendete, gli agenti crederanno che *open(left_door)* sia vero.

.send(r,tell,[open(left_door),open(right_door)]);
Il belief *open(left_door) [source(s)]* verrà rimosso dalla belief base dell'agente "r". Se nella belief base dell'agente destinatario è presente una collezione di letterali del tipo *open(left_door) [source(s), source(percept), source(t)]*, all'atto della ricezione del messaggio, non verrà rimosso l'intero belief, ma verrà eliminata l'annotazione dell'agente mittente, ottenendo il seguente belief: *open(left_door) [source(percept), source(t)]*.

2.4.2 Delegazione di obiettivi

Tramite la performative achieve, un agente può delegare il raggiungimento di un obiettivo ad un altro agente. Inoltre, può richiedere di rinunciare a perseguire un obiettivo tramite la performative unachieve. Di seguito due esempi sull'utilizzo delle due performative.

.send(r, achieve, open(left_door);

L'evento *(+!open(left_door) [source(s)], T)* viene creato e aggiunto ai goal dell'agente "r". Se l'agente dispone di un piano adatto a perseguire l'obiettivo, allora verrà eseguito. Si noti che Jason applica un'annotazione per indicare come l'obiettivo sia stato delegato da un altro agente. Questa informazione può tornare utile nella scelta dei piani, in modo da scegliere eventualmente il miglior corso di azioni da intraprendere.

.send(r, unachieve, open(left_door);

L'internal action *.drop_desire(open(left_door))* viene eseguita rimuovendo il goal *!open(left_door)* dall'elenco degli obiettivi da perseguire, senza generare un fallimento dei piani che richiedono il raggiungimento dell'obiettivo.

2.4.3 Richiesta informazioni

Un agente, oltre a ricevere informazioni dall'ambiente e da altri agenti, può richiedere delle informazioni tramite le performative askOne e askAll. Di seguito alcuni esempi sull'utilizzo delle due performative.

.send(r, askOne, open(Door), Reply);

L'esecuzione di questa internal action, provocherà la sospensione del piano fino a quando non riceverà una risposta dall'agente "r". Se l'agente "r" accetterà il messaggio, verrà utilizzato il piano predefinito di Jason per gestire la richiesta. Quando l'agente "s" riceverà una risposta da parte di "r", il contenuto del messaggio verrà memorizzato nella variabile *Reply* e il piano riprenderà la propria esecuzione. Nel caso in cui il messaggio fosse multicast, il piano riprenderà l'esecuzione non appena "s" riceverà una risposta da parte di uno degli agenti destinatari. L'invio di un messaggio di questo tipo consente di poter effettuare dei test goal nelle belief base di altri agenti.

.send(r, askOne, open(Door));

Il comportamento è analogo all'esempio precedente, ma asincrono. Il mes-

saggio viene inviato, ma il piano di esecuzione non viene sospeso in attesa di risposta da parte di uno degli agenti destinatari.

.send(r, askAll, open(Door), Reply);

Il comportamento è analogo al primo esempio, ma *Reply* potrà contenere tutte le possibili risposte riguardanti il propositional_content. Per esempio, se l'agente "r" crede che *open(left_door)* e *open(right_door)* siano vere, *Reply* verrà inizializzata come segue: *[open(left_door),open(right_door)]*.

Capitolo 3

Environment

Dopo aver dato una definizione di agente e illustrato le caratteristiche principali, all'interno di questo capitolo verrà trattato l'ambiente di un sistema multi-agente. E' possibile fornire due definizioni di ambiente: una definizione legata all'ambito dell'intelligenza artificiale e una recentemente sviluppata nel contesto dell'ingegneria del software orientata agli agenti.
Nel contesto dell'intelligenza artificiale, l'ambiente può essere definito come il mondo esterno (rispetto al sistema, composto da uno o più agenti) che viene percepito e modificato dall'agente in modo da raggiungere i propri scopi. Una seconda definizione, sviluppata di recente nell'ambito dell'ingegneria del software orientata agli agenti, definisce l'ambiente come un luogo adatto per incapsulare funzionalità e servizi a supporto delle attività degli agenti. L'ambiente fa parte del sistema multi-agente e può essere opportunamente progettato in modo da migliorare lo sviluppo complessivo del sistema. Con quest'ultima definizione, l'ambiente viene considerato come parte attiva del sistema multi-agente.
All'interno di questo capitolo e, nel prosieguo della trattazione, come definizione di ambiente verrà considerata solo ed unicamente la definizione sviluppata nel contesto dell'ingeneria del software orientata agli agenti.

3.1 Caratteristiche dell'ambiente

Come detto in precedenza, l'ambiente può essere progettato in modo da migliorare lo sviluppo complessivo del sistema. L'ambiente può essere utilizzato per progettare la parte computazionale del sistema che è funzionale all'agente. Questi ambienti sono denominati ambienti endogeni. In questo contesto, l'ambiente viene considerato come una parte ortogonale agli agenti, ma strettamente correlata ad essi. Il concetto di ortogonalità viene definito come *separation of concerns*: gli agenti rappresentano l'astrazione di base di un sistema multi-agente, in grado di avere un comportamento autonomo e pro-attivo. L'ambiente invece, rappresenta lo spazio computazionale, fornendo un supporto alle attività degli agenti.
Un ambiente endogeno è caratterizzato dalle seguenti proprietà.

Astrazione: Il modello adottato deve preservare il livello di astrazione di agente. I concetti utilizzati dalla programmazione dell'ambiente devono essere consistenti con i concetti degli agenti.

Ortogonalità: Come detto in precedenza, il modello deve essere il più ortogonale possibile, in modo da favorire lo sviluppo di sistemi eterogenei.

Generalità: Questa proprietà consente di sviluppare diversi tipi di ambienti, operanti su domini applicativi differenti tra loro, utilizzando lo stesso set di concetti e costrutti.

Modularità: Il modello dovrà essere modulare, evitando aspetti monolitici e centralizzati.

Estensibilità dinamica: Il modello dovrà sostenere lo sviluppo dinamico, consentendo la sostituzione,l'aggiunta o la rimozione di parti dell'ambiente in modo dinamico.

Riusabilità: Come per la generalità, le varie parti del sistema dovranno poter essere riutilizzate in contesti differenti.

Oltre agli aspetti generali appena descritti ve ne sono altri che caratterizzano la programmazione di un ambiente, vale a dire: *action model*, *perception model*, *environment computational model*, *environment data model* e *environment distribution model*. Di seguito verranno analizzati singolarmente.

3.1.1 Action model

Come visto nel capitolo riguardante Jason, un'azione eseguita da un agente può avere successo o meno. La buona riuscita dell'esecuzione dell'azione non implica che gli attuatori abbiano effettivamente modificato l'ambiente. L'agente, per assicurarsi che l'ambiente sia stato modificato in seguito all'esecuzione di un'azione, deve aggiornare le proprie beliefs. Gli ambienti endogeni forniscono una semantica più ricca, garantendo all'agente, in caso di successo, la corretta esecuzione dell'azione.

3.1.2 Perception model

Questo aspetto definisce il modo in cui l'agente percepisce l'ambiente, fornendo due semantiche da adottare: *state-based* ed *event-based*. Nel primo caso, ogni percept rappresenta le informazioni sullo stato attuale dell'ambiente e vengono recuperate dall'agente nella fase di percezione dell'ambiente.
Nel secondo caso, ogni percept rappresenta le informazioni sui cambiamenti che si sono verificati nell'ambiente. L'ambiente provvederà ad inviarli all'agente in modo automatico, indipendentemente dallo stato di esecuzione dell'agente.

3.1.3 Environment computational model

Questo aspetto definisce come progettare le funzionalità dell'ambiente. L'approccio più semplice è quello monolitico, rappresentando la struttura e

il comportamento dell'ambiente con un unico oggetto dotato di un singolo stato. Sebbene Jason, 2APL e GOAL utilizzino questo approccio, risulta essere poco pratico in termini di riusabilità. E' preferibile utilizzare un approccio modulare in modo da modularizzare le funzionalità dell'ambiente, favorendone così la riusabilità e la manutenibilità.

3.1.4 Environment data model

Questo aspetto definisce i tipi di dati che vengono scambiati tra l'agente e l'ambiente, definendo: la codifica dei parametri, i feedback delle azioni, il contenuto delle percepts e la loro rappresentazione. Per favorire l'interoperabilità tra sistemi sviluppati con tecnologie diverse, è necessario definire in modo esplicito le possibili strutture dati coinvolte in azioni e percepts. Inoltre è necessario definire un linguaggio comune per la rappresentazione dei dati (basato su XML).

3.1.5 Environment distribution model

Per concludere, questo aspetto definisce le modalità secondo le quali il sistema deve essere distribuito su più nodi di rete. A tal fine, viene introdotto il concetto di *luogo*, definendo inoltre le modalità di connessione tra i luoghi. In questo modello, anche il *tempo* assume un aspetto importante. Nei sistemi multi-agente distribuiti non è possibile avere una singola nozione di tempo all'interno del sistema, sia dal punto di visto teorico che pratico. Nella maggior parte dei sistemi distribuiti si cerca di uniformare questo concetto, con lo scopo di unificare le azioni e gli eventi degli agenti sul sistema.

3.2 Artifacts

L'ambiente è concepito come un insieme dinamico di entità computazionali chiamati artefatti. Un insieme di artefatti può essere organizzato in uno o più workspaces, eventualmente distribuiti nei diversi nodi della rete. Dal

punto di vista dei progettisti e programmatori di sistemi multi-agenti, un artefatto è un'astrazione, una base per strutturare e organizzare l'ambiente, fornendo una programmazione general-purpose e un modello di calcolo per modellare le funzionalità disponibili per gli agenti.
Nella figura 3.1 è possibile notare una rappresentazione astratta di un artefatto.

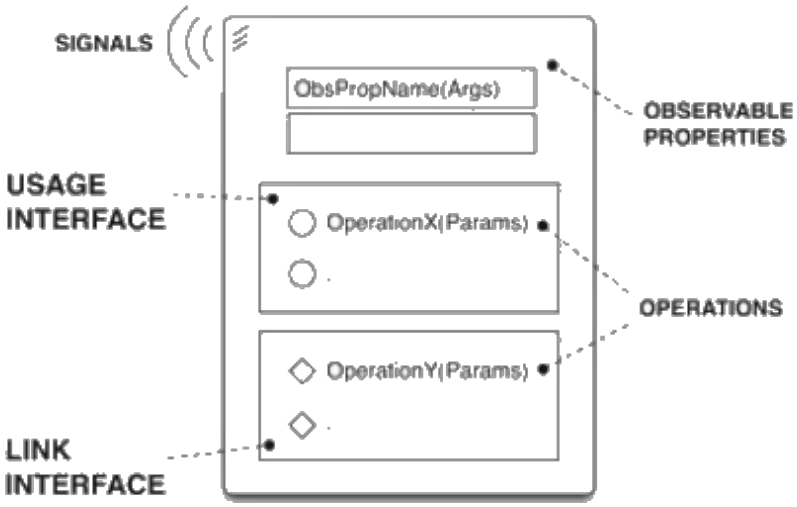

Figura 3.1: Rappresentazione astratta di un artefatto

Un artefatto fornisce agli agenti un insieme di operazioni e proprietà osservabili. Le operazioni sono dei processi eseguiti all'interno dell'artefatto e possono essere attivati dagli agenti o da altri artefatti. Solo le operazioni presenti nella *usage interface* possono essere utilizzate dagli agenti. Le proprietà osservabili rappresentano le variabili di stato, il cui valore può essere percepito dagli agenti.
L'esecuzione di un'operazione può generare dei segnali percepiti dagli agenti. Diversamente dalle proprietà osservabili, i segnali, sono utili per rappresentare degli eventi osservabili non persistenti verificatesi all'interno di un artefatto.
Gli artefatti possono essere collegati tra loro attraverso un meccanismo di linking, in modo che uno di essi possa eseguire le operazioni di un altro. A

tale scopo, un artefatto contiene un'interfaccia chiamata *link interface*, all'interno della quale, vengono definite le operazioni che possono essere eseguite da altri artefatti. Tali operazioni non sono accessibili agli agenti. Tramite il meccanismo di linking è possibile realizzare ambienti distribuiti, in modo da collegare artefatti presenti in workspaces differenti, collocati nei diversi nodi di rete.

Dopo aver definito i concetti di base, è possibile proseguire definendo i meccanismi attraverso i quali è possibile accedere e utilizzare gli artefatti.

3.2.1 Utilizzo di un artefatto

Un agente può utilizzare un artefatto attraverso due modalità: eseguendo le operazioni presenti nella usage interface oppure percependo le proprietà osservabili.
Un'operazione viene eseguita attraverso l'azione:

use(ArId, OpName (Params)):OpRes

Il parametro *ArId* identifica l'artefatto, *OpName(Params)* specifica l'operazione da eseguire con eventuali parametri e *OpRes* il risultato dell'operazione. Quando un agente invoca l'azione use, il suo piano viene sospeso fino a quando non viene segnalato il termine dell'operazione. Terminata l'operazione, il piano riprende la propria esecuzione. Durante l'esecuzione di un'operazione, l'agente non è bloccato, in quanto il ciclo prosegue continuando a processare percepts ed eseguire altri piani.

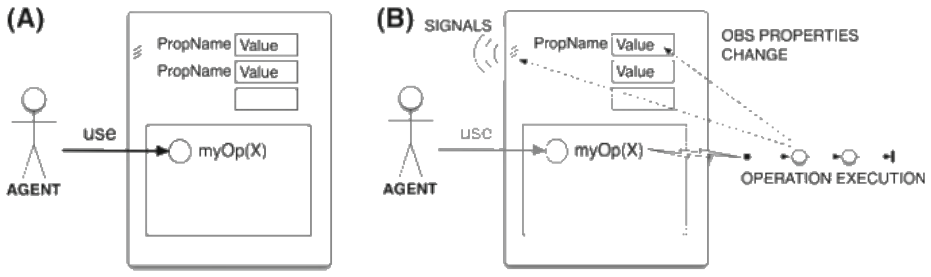

Figura 3.2: (A) Invocazione di un'operazione da parte di un agente. (B) Modifica proprietà osservabili e generazione segnali.

Tramite l'azione *focus(ArId,Filter)*, l'agente può osservare l'artefatto, percependo le proprietà osservabili e i segnali. Tramite il parametro *Filter*, è possibile specificare un sottoinsieme di eventi a cui si è interessati. Un agente può osservare più di un artefatto. Tramite l'azione *stopFocus(ArId)*, l'agente terminerà di seguire un determinato artefatto.

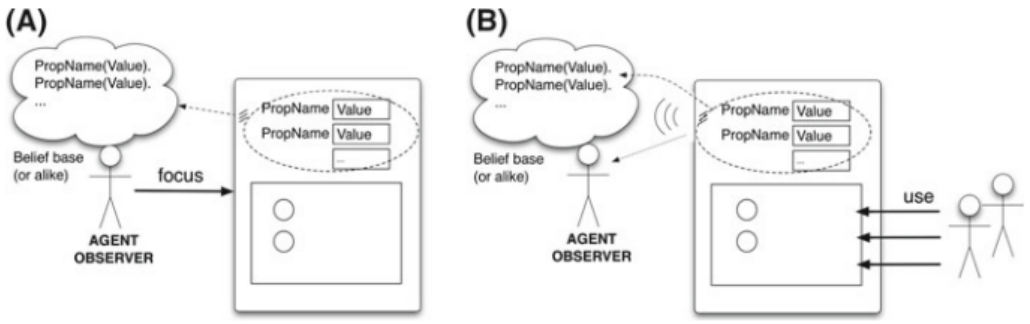

Figura 3.3: Osservazione di un artefatto

Per concludere, con l'esecuzione dell'azione:

observeProperty (PropName):PropValue

è possibile ottenere il valore di una proprietà osservabile. L'esecuzione di questa azione non comporta l'aggiornamento della belief base, ed è particolarmente utile quando un agente non ha bisogno di essere continuamente aggiornato sul valore di una proprietà osservabile.

3.2.2 Linking e unlinking di un artefatto

Tramite il meccanismo di linking è possibile collegare degli artefatti in modo che uno possa eseguire le operazioni di un altro. Tramite le azioni:

linkArtifacts(LinkingArId,LinkedArId,Port)

unlinkArtifacts (LinkingArId, LinkedArId)

è possibile collegare e scollegare due artefatti. Il parametro *Port* è necessario quando ad un artefatto si vogliono collegare più artefatti.

3.3 CArtAgO

CArtAgO (**C**ommon **AR**tifact infrastructure for **AG**ents **O**pen environments) è un framework e un'infrastruttura per lo sviluppo ed esecuzione di environment, basato sul meta-modello Agents and Artifacts. Come framework, fornisce delle API in Java che consentono la progettazione e l'esecuzione di artefatti. Come infrastruttura, invece, fornisce delle API con le quali è possibile estendere i meccanismi di base degli agenti, in modo da favorire l'integrazione con gli artefatti.
Come per Jason, di seguito verranno analizzati i costrutti di base e i meccanismi riguardanti questo framework.

Le API basate su Java consentono di sviluppare artefatti utilizzando i concetti della programmazione ad oggetti, senza la necessità di utilizzare un nuovo linguaggio.

```
import cartago.*;

public class Count extends Artifact {
    void init(int initialValue) {
        defineObsProperty("count", 0);
    }

    @OPERATION
    void inc() {
        int c = getObsProperty("count").intValue();
        c++;
        updateObsProperty("count",c);
    }
}
```

Figura 3.4: Implementazione di un artefatto

Nella figura 3.4 è mostrato un semplice esempio di artefatto che implementa un contatore. Per creare un artefatto è di fondamentale importanza estendere la classe Artifact. Il metodo *init* viene utilizzato per inizializzare l'artefatto

all'atto della creazione. Nell'esempio mostrato nella figura 3.4, viene definita una proprietà osservabile tramite l'istruzione *defineObsProp(key,value)*. Per poter leggere e modificare il valore di una proprietà osservabile, si utilizzano rispettivamente *getObsProperty* e *updateObsProperty*.
I metodi che definisco le operazioni sono caratterizzati dall'annotazione *@OPERATION* e non restituiscono alcun valore. Nell'esempio precedente viene definita l'operazione *inc* che si occupa di incrementare il valore della proprietà osservabile *count*. I parametri dell'operazione possono essere utilizzati sia come come parametri di input che come parametri di output.
Nella seguente figura è possibile notare un esempio di un'operazione con parametri di input e output.

```
@OPERATION
void sumWithResult(int x,int y,OpFeedbackParam<Integer> result){
    result.set(x+y);
    signal("sum calculated");
}
```

Figura 3.5: Operazione con parametri

I parametri x e y sono i parametri di input, mentre *result* è il parametro di output. Dopo aver impostato il valore del parametro di output, viene generato un segnale attraverso la primitiva *signal*, che verrà percepita da tutti gli agenti che osservano l'artefatto.
Tramite l'annotazione *@INTERNAL_OPERATION* è possibile definire delle operazioni interne ad un artefatto; l'equivalente di un metodo privato nella programmazione ad oggetti. Per eseguire un'operazione interna, è necessario utilizzare la primitiva *execInternalOp*. L'esecuzione di un'operazione interna avviene in modo asincrono.
Di seguito è mostrato un esempio di codice riguardante l'utilizzo di un'operazione interna.

```
public class Clock extends Artifact {
    private final static int TICK= 1000;
    private boolean working;

    void init(){
        working=false;
    }

    @OPERATION
    void start(){
        if(!working){
            working=true;
            execInternalOp("work");
        }else{
            failed("already_started");
        }
    }

    @OPERATION
    void stop(){
        working=false;
    }

    @INTERNAL_OPERATION
    void work(){
        while(working){
            signal("tick");
            await_time(TICK);
        }
    }
}
```

Figura 3.6: Implementazione di un'operazione interna

L'esempio inoltre mostra come sia possibile, attraverso l'uso della primitiva *failed*, specificare come un'operazione non sia riuscita. Tale primitiva genera un'eccezione che interrompe l'esecuzione dell'operazione.
Come detto in precedenza, un artefatto può eseguire le operazioni di un altro artefatto tramite il meccanismo di linking. Tramite l'annotazione *@LINK* è possibile specificare quali operazioni possono essere eseguite da altri artefatti. Un artefatto può eseguire un'operazione presente nella link interface di un altro artefatto tramite la primitiva *execLinkedOp*. Un'operazione può presentare le annotazioni *@LINK* e *@OPERATION*. In questo modo, un'operazione potrà essere eseguita sia da un artefatto che da un agente.

(A)

```
class LinkedArtifact extends Artifact{
    ...
    @LINK void linkedOp(String s, OpFeedbackParam<Integer> r){
        r.set(s.length());
    }
}
```

(B)

```
class LinkingArtifact extends Artifact{
    ...
    @OPERATION void myOp(String s){
        OpFeedbackParam<Integer> res=
                new OpFeedbackParam<Integer>();
        execLinkedOp("linkedOp",s,res);
        int v=res.get();
        ...
    }
}
```

Figura 3.7: (A) Artefatto con operazione presente nella link interface. (B) Esecuzione dell'operazione presente nella link interface di un altro artefatto.

3.3.1 Integrazione in linguaggi ad agenti

CArtAgO è stato concepito per essere integrato con qualsiasi linguaggio di programmazione ad agenti, offrendo la possibilità di estendere le funzionalità degli agenti. Le funzionalità aggiuntive di cui può disporre un agente comprendono sia le primitive messe a disposizione da CArtAgO (es: makeArtifact, lookupArtifact, focus,etc), sia tutte le operazioni presenti nelle usage interface degli artefatti.
Le proprietà osservabili di un artefatto vengono mappate nella belief base degli agenti, mentre i segnali sono mappati come occorrenze di eventi generati dalle proprietà osservabili, a cui corrispondono uno o più piani.
Siccome all'interno di questa trattazione si è approfondito Jason, nel successivo esempio si vedrà l'integrazione di CArtAgO con Jason. Come artefatti verranno considerati i due artefatti definiti in precedenza: *Counter* e *Clock*.

```
// agentA

!test.

+!test
<-make_artifact("myclock","Clock");
start;
make_artifact("mycount","Count");
inc;
inc [artifact_name("mycount")];
observe_property("mycount",count(V));
println("[agentA] count value : ",V).
```

```
// agentB

!test.

+!test
<-!discover("myclock",Clock);
focus(Clock);
!discover("mycount",Count);
focus(Count).

+count(V)
<-println("[agentB] count value : ",V).

+tick
<-println("[agentB] new tick event perceived").

+!discover(ArtName,Id)
<-lookup_artifact(ArtName,Id).

-!discover(ArtName,Id)
<-!discover(ArtName,Id).
```

Figura 3.8: esempio integrazione di CArtAgO in Jason

Come si evince dalla figura 3.8, *agentA* presenta un solo goal iniziale. Il piano *test* crea un'istanza dell'artefatto *Clock*, denominata *myclock* ed esegue l'operazione *start* dell'artefatto *Clock*. Si ricorda che *start* prevede un'operazione interna, la quale, genererà un segnale fino a quando non verrà eseguita l'operazione *stop*.
Oltre a creare un'istanza dell'artefatto *Clock*, viene creata un'istanza per l'artefatto *Count*, per poi eseguire l'operazione *inc*. L'utilizzo dell'annotazione in *inc [artifact_name("mycount")]*, può essere utile quando vi sono più artefatti che presentano un'operazione con il medesimo nome.
I due artefatti sono osservati da *agentB*. Come detto in precedenza, quando un agente osserva un artefatto, le proprietà osservabili vengono mappate nella belief base dell'agente e aggiornate automaticamente ad ogni cambiamento del valore. I segnali invece non vengono memorizzati automaticamente nella belief base, ma sono mappati come degli eventi che possono scatenare l'esecuzione di un piano. Nell'esempio, agentB ha una percezione sul valore corrente della proprietà osservabile count e percepisce gli eventi generati dall'artefatto *Clock*. *count* e *tick* dell' agentB verranno eseguiti ad ogni cambiamento della proprietà osservabile e ad ogni segnale generato.

Capitolo 4

Sistemi embedded

Un sistema embedded è un sistema di elaborazione special purpose - ovvero, che svolge una specifica funzione o compito, incorporato in un sistema o in un dispositivo elettronico. Oggigiorno i sistemi embedded sono sempre più diffusi, basti pensare ai svariati settori in cui sono presenti: dall'elettronica di consumo all'aviazione, fino al settore medico.
I sistemi special purpose, a differenza dei sistemi general purpose (ad esempio un computer), eseguono uno specifico programma ripetutamente. Sono quindi progettati per eseguire un'applicazione specifica, minimizzando le risorse utilizzate e massimizzando la robustezza.

4.1 Caratteristiche di un sistema embedded

Un sistema embedded per svolgere un compito specifico deve possedere una o più determinate caratteristiche. Le principali caratteristiche di un sistema embedded si possono raggruppare in tre macro-sezioni e sono: *efficienza*, *affidabilità* e *reattività*.

4.1.1 Efficienza

Quando si parla di efficienza nell'ambito di sviluppo del software, solitamente, si intende efficienza del codice. L'obiettivo principale è costruire un

software con un costo computazionale il più basso possibile.
Nei sistemi embedded, questa è solo una delle caratteristiche legate all'efficienza. Sempre legato al codice, un aspetto molto importante è la dimensione. Trattandosi di sistemi special purpose, le risorse di utilizzo sono limitate, di conseguenza non è possibile scrivere del codice che occupa più spazio della capacità massima di memorizzazione.
Altro aspetto importante è l'efficienza legata al peso del dispositivo. Si pensi a uno smartphone. Non può essere troppo pesante in quanto verrebbe a meno la portabilità.
Infine, per concludere questo paragrafo, vi è l'efficienza in termini energetici. Un dispositivo che richiede una ricarica continua potrebbe essere poco pratico. Si consideri ad esempio il pacemaker. Le sue caratteristiche fondamentali sono sicuramente l'efficienza energetica e il peso. Trovandosi all'interno del corpo umano non può essere molto pesante. Dal punto di vista energetico, la batteria deve durare il più possibile, in modo da non sottoporre il paziente a continui interventi.

4.1.2 Affidabilità

Alcune applicazioni dei sistemi embedded includono sistemi critici. Si pensi ai sistemi embedded nel settore bio-medico, trasporto (es: pacemaker, automobili, aerei), etc. Per questi tipi di dispositivi sono richieste le seguenti caratteristiche: *reliability*, *availability*, *safety* e *security*.
Per *reliability* si intende la capacità di un sistema di svolgere un compito o una funzione per un determinato periodo di tempo T. La probabilità che un sistema fallisca in un dato intervallo di tempo, è espresso dal tasso di fallimento, misurato in FIT (Failure In Time). L'inverso del FIT, è chiamato MTTF (Mean Time To Failure). Se il tasso di fallimento ha un valore di 10^{-9} guasti/ora o inferiore, allora si parla di un sistema con un requisito di affidabilità molto elevata.
Con il termine *availability* si intende la percentuale di tempo in cui un sistema è funzionante. Questa caratteristica è strettamente lagata al MTTF

e alla manutenibilità (MMTR). La percentuale di availability è data da: MTTF/(MTTF+MTTR).
Con il termine *safety* si intende la probabilità che il sistema non rechi danni agli utenti o all'ambiente in cui è immerso.
Per concludere, con il termine *security* si intende la capacità di garantire l'autenticità, l'integrità delle informazioni e di negare l'accesso ai servizi e alle informazioni da parte di chi non è autorizzato. Negli ultimi anni, la sicurezza ha assunto un ruolo fondamentale nei sistemi real-time e nell'internet of things, poiché la violazione di un sistema potrebbe causare danni ai dati e all'ambiente in cui esso è posto.

4.1.3 Reattività

Spesso i sistemi embedded sono utilizzati in contesti dove devono prontamente reagire a stimoli che provengono dall'ambiente. E' necessario quindi eseguire elaborazioni ed eventualmente azioni in real-time, senza ritardi. I sistemi legati al vincolo della reattività si possono suddividere in *sistemi hard real-time* e *sistemi soft real-time*.
Per *sistemi hard real-time*, si intendono tutti quei sistemi che devono svolgere assolutamente un compito o una funzione in un determinato periodo di tempo. L'eventuale violazione di deadline può risultare critica per l'intero sistema.
Come intuibile, in un *sistema soft real-time*, la violazione di una deadline può essere problematica, ma senza causare delle criticità all'interno del sistema.
Un esempio di sistema hard real-time può essere un sistema che si occupa della messa in sicurezza di un edificio in caso di incendio. Il sistema, in caso di incendio in una sezione dell'edificio, deve provvedere alla chiusura automatica delle porte anti-incendio, in modo da isolare il principio di incendio, chiudendo le porte in un tempo T. Se la chiusura delle porte non avvenisse entro il tempo prestabilito, l'incendio potrebbe propagarsi nel resto dell'edificio.

4.2 Tipi di sistema

Nel corso degli anni, a seconda delle richieste del mercato, sono stati sviluppati vari tipi di sistemi.
I sistemi CPS (Cyber Phisical Systems) sono dei sistemi di controllo, composti da sottosistemi fisici,computazionali e di rete.
La parte fisica può includere dispositivi meccanici, biologici, processi chimici o umani.
La parte computazionale è data da una o più piattaforme di elaborazione, ognuna contenente sensori, attuatori e uno o più computer.
La parte di rete fornisce meccanismi e supporti per fare in modo che i computer comunichino.
Un esempio di sistema CPS può essere un drone, molto utilizzato recentemente.
I sistemi M2M (Machine to Machine) sono un insieme di tecnologie che supportano la comunicazione cablata e wireless fra dispositivi/macchine, al fine di permettere lo scambio di informazioni locali. Questo tipo di sistema è utilizzato principalmente per funzioni di monitoraggio e controllo.
In ambito industriale, un tipo di sistema M2M è dato dai sistemi SCADA (Supervisory Control and Data Acquisition). Si tratta di sistemi che operano con codifiche di segnali, trasmesse per canali di comunicazione, al fine di realizzare un controllo remoto di sistemi/equipaggiamenti/impianti.
I sistemi M2M, oggigiorno si possono considerare parte integrante dell'IOT (Internet Of Things). Quando si parla di IOT, si intende generalmente una rete di dispositivi in grado di acquisire informazioni dall'ambiente, elaborarle e quindi agire producendo opportuni effetti. Inoltre, come si evince dal nome, i dispositivi sono connessi mediante varie tecnologie di rete che permettono loro di interagire in modo co-operativo. Con il termine *things* si intendono principalmente i dispositivi presenti nella vita quotidiana di una persona, come: orologi, robot, automobili, etc.

4.3 Architettura Hardware

In generale, l'architettura di un elaboratore è data da una CPU, una memoria centrale e da un insieme di device controller connessi attraverso un bus comune.
Come detto nell'introduzione del capitolo, un sistema embedded è progettato per svolgere uno specifico compito, permettendo di ridurre le risorse utilizzate. La maggior parte di questi sistemi hanno dimensioni ridotte e non è possibile utilizzare l'hardware di un elaboratore. Per i sistemi embedded si utilizzano solitamente i *microcontrollori* o i *SOC* (System On a Chip).
I microcontrollori sono dei dispositivi elettronici, nati come evoluzione alternativa ai microprocessori. Essi, integrano su un singolo chip, un sistema di componenti che permette di avere la massima autosufficienza funzionale per applicazioni embedded. Un microcontrollore è composto da un processore, una memoria permanente, una memoria volatile e dai canali di I/O a cui è possibile collegare sensori/attuatori. Inoltre è dotato di un gestore di interrupt ed eventualmente altri blocchi specializzati.
I SOC, a differenza dei microcontrollori, integrano in un unico chip un sistema completo, dotato solitamente di: microprocessore, RAM, circuiteria di I/O, sottosistema video.
La scelta tra microcontrollore e SOC dipende sostanzialmente dai requisiti del progetto.

4.3.1 General purpose I/O pin

I microprocessori e i microcontrollori utilizzati in ambito embedded, contengono usualmente un certo numero di pin, che possono essere utilizzati direttamente per gestire l' I/O e anche i loro controller. I pin sono generalmente general purpose, nel senso che possono essere programmati per fungere da input o da output, a seconda delle necessità. Inoltre, i pin possono essere digitali o analogici. Di seguito verranno presi in considerazione, come caso

di studio, i pin di I/O del microcontrollore ATmega328P.

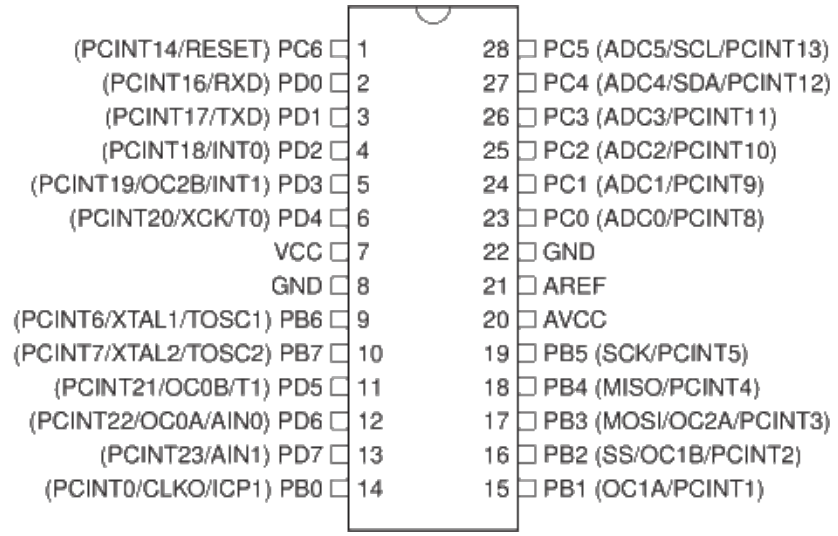

Figura 4.1: Microcontrollore ATmega328P

L'ATmega328P ha 23 linee di I/O, raggruppate in tre porte da 8 bit l'una, denominate B, C e D. Le porte B e D contengono i pin digitali, mentre la porta C contiene i pin analogici. Ogni pin è identificato da una sigla, come ad esempio PC2, che identifica il secondo pin analogico della porta C.
Ogni porta è gestita da tre registri: DDRx, PORTx e PINDx.
Il registro DDRx è un registro di lettura e scrittura che contiene la direzione dei pin ad esso collegati. Con il valore 0 il pin viene impostato come input, mentre il valore 1 imposta il pin come output.
Il registro PORTx è un registro di lettura e scrittura che contiene lo stato dei pin. Se il pin è impostato come input, un bit a 1 attiva la resistenza di pull-up, mentre a 0 la disattiva. Se il pin è impostato come output, un bit a 1 indica uno stato HIGH, mentre 0 indica uno stato LOW.
Il registro PINDx è un registro di sola lettura e contiene la lettura del segnale collegato al pin: 1 per un segnale alto (HIGH), 0 per un segnale basso (LOW).
Sei dei pin digitali di questo microcontrollore possono essere utilizzati con modulazione di frequenza (PWM). Si tratta di una tecnica di pilotaggio di dispositivi di output, che permette di emulare in uscita un segnale analogico,

generando dei segnali detti PWM. Il segnale analogico viene emulato tramite la modulazione del duty cycle di un'onda quadra.
Il microcontrollore ATmega328P è fornito di pin analogici. Un sistema di elaborazione, per poter elaborare un segnale analogico, deve convertirlo. La conversione avviene mediante un convertitore analogico-digitale che mappa il valore continuo in un valore discreto in un certo range. Il numero di bit utilizzati dal convertitore rappresentano la risoluzione del convertitore e ne determinano la precisione.

4.3.2 Bus e protocolli

I protocolli di scambio informazioni mediante singoli pin, possono diventare molto complessi, a seconda dei dispositivi di I/O con cui interfacciarsi. Per facilitare la comunicazione, sono stati introdotti nel tempo varie interfacce e protocolli, che possono essere classificati in *seriali* e *parallele*.
Le interfacce *parallele* permettono di trasferire più bit contemporaneamente, utilizzando un bus a 8,16 o più fili.
Le interfacce *seriali*, permettono di inviare i bit in uno stream sequenziale, bit per bit, utilizzando un solo filo o comunque un numero ridotto. Tali interfacce possono essere suddivise in due categorie: *sincrone* e *asincrone*. In modalità *asincrona*, il trasmettitore e il ricevitore si sincronizzano utilizzando i dati stessi. Il trasmettitore invia un bit di "partenza", successivamente il dato vero e proprio e, infine, un bit di "stop". In modalità *sincrona*, la trasmissione dei dati è sincronizzata con il clock del microcontrollore/microprocessore.

Capitolo 5

Arduino

Arduino è una piattaforma di sviluppo open source basata su microcontrollore. E' composto da una piattaforma hardware sviluppata presso l'Interaction Design Institute, un istituto di formazione post-dottorale con sede a Ivrea, fondato da Olivetti e Telecom Italia. All'hardware viene affiancato un ambiente di sviluppo integrato multipiattaforma.
La scheda Arduino è in grado di interagire con l'ambiente in cui si trova, ricevendo informazioni dai sensori ed è in grado di modificarlo tramite l'utilizzo di attuatori.
Nel corso degli anni sono state immesse sul mercato varie tipologie di schede, con caratteristiche hardware simili, a seconda dello scopo specifico. Sono ora disponibili soluzioni nelle fasce: entry level, internet of things e wearable.

5.1 Hardware

La maggior parte delle schede Arduino, consistono in un microcontrollore a 8 bit AVR prodotto da Atmel, con l'aggiunta di componenti complementari per facilitarne l'incorporazione in altri circuiti. In queste schede sono usati chip della serie megaAVR - nello specifico i modelli ATmega8, ATmega168, ATmega328, ATmega1280 e ATmega2560, a seconda del modello. Il modello Arduino Due è il primo a integrare una CPU Atmel SAM3X8E con infra-

struttura ARM Cortex-M3.
Per implementare il comportamento interattivo, tutti i modelli Arduino sono forniti di funzionalità di I/O, grazie alle quali essi ricevono i segnali raccolti da sensori esterni. In base a tali valori, il comportamento della scheda è gestito dal microcontrollere, in base alle decisioni determinate dal particolare programma in esecuzione in quel momento. L'interazione con l'esterno avviene attraverso gli attuatori pilotati dal programma. Il numero di connettori di I/O a disposizione variano a seconda del modello. I connettori sono suddivisi in I/O digitale, di cui una parte possono produrre segnali in modulazione di frequenza (PWM) e in input analogici. Gli input analogici possono essere riprogrammati per funzionare come normali pin di I/O digitali. Per poter leggere i valori analogici, le schede Arduino sono dotate di un convertitore analogico-digitale. La scheda Arduino Due è inoltre dotata di due convertitori digitale-analogico, in modo da avere due pin analogici di output.
Per favorire il caricamento del programma sul microcontrollore, le schede Arduino sono dotate di una porta USB. Integrano inoltre un chip per la conversione del segnale digitale da USB a seriale. Nei modelli più recenti, in particolare Arduino Leonardo e Arduino Esplora, tale chip non è presente, poiché il microcontrollore integra già questa funzione. Oltre a supportare la comunicazione seriale, le schede Arduino sono in grado di supportare i protocolli di comunicazione I2C e SPI.
Molte schede includono un regolatore lineare di tensione a 5 Volt e un oscillatore a cristallo a 16 MHz.
L'alimentazione della scheda può avvenire tramite porta USB, attraverso un adattatore in corrente continua oppure utilizzando i pin Vin e GND. Si sconsiglia quest'ultimo metodo in quanto un accidentale inversione dei poli potrebbe danneggiare la scheda.
Per maggiori dettagli sulle caratteristiche tecniche dei vari modelli, si consiglia di consultare [5].

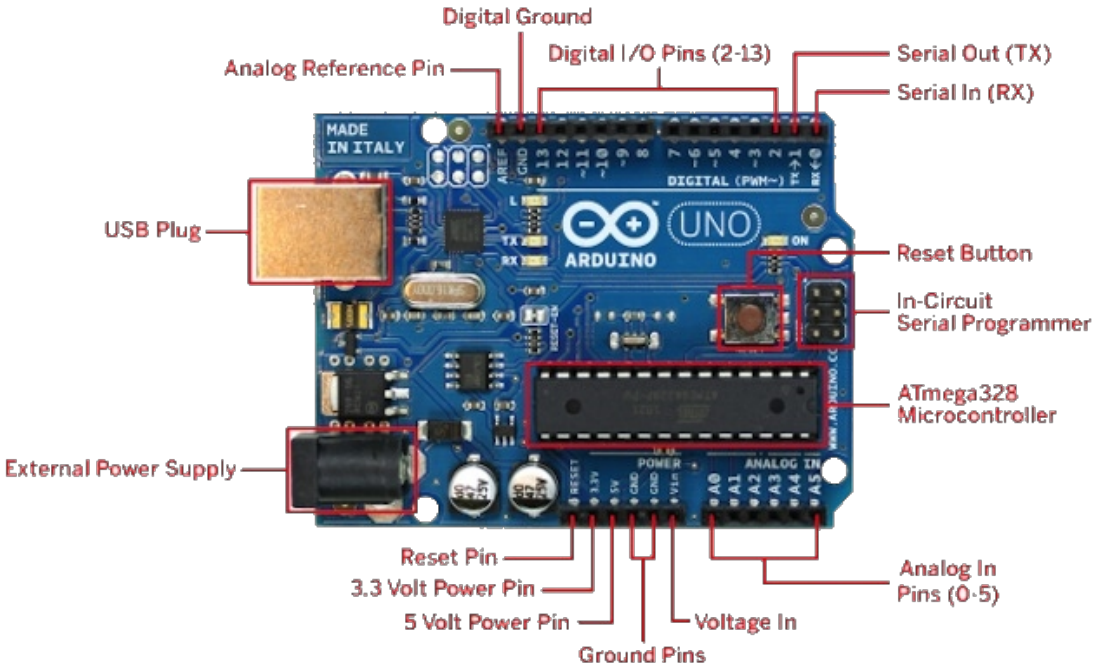

Figura 5.1: Architettura Arduino Uno

5.2 Software

5.2.1 Linguaggio di programmazione

Il framework di riferimento per la programmazione di Arduino si chiama Wiring; un linguaggio di programmazione derivato da C/C++ . Si tratta di un framework open-source per la programmazione di microcontrollori. Prima dell'avvento di Arduino, la programmazione di un microcontrollore avveniva tramite linguaggi a basso livello, come C o Assembly. Inoltre richiedeva una conoscenza dei concetti di elettronica, come gli interrupt e le porte logiche. L'utilizzo di Wiring da parte di Arduino, ha semplificato tutto questo e permette di programmare il microcontrollore facilmente, attraverso un ambiente C/C++ appositamente realizzato per sistemi a microcontrollore.
I programmi realizzati per Arduino vengono chiamati in gergo "sketch". Per

realizzare un programma per Arduino, al programmatore viene richiesto di definire almeno due metodi:

void setup()
Funzione invocata una sola volta all'inizio di un programma. Generalmente utilizzata per i settaggi iniziali, come il metodo di funzionamento di un pin (input o output), stabilire una connessione seriale, etc...

void loop()
Funzione invocata ripetutamente, la cui esecuzione si interrompe solo con lo spegnimento della scheda.

5.2.2 Bootloader

I microcontrollori sono solitamente programmati tramite un hardware esterno detto programmatore. Il bootloader è un firmware che i progettisti di Arduino hanno inserito nei chip, il cui scopo è quello di caricare gli sketch nel microprocessore senza l'ausilio del programmatore. Se si desidera costruire una propria scheda, seguendo gli schemi disponibili in [5], è necessario definire un proprio bootloader, in modo da evitare l'utilizzo di un hardware esterno per il caricamento degli sketch.
Quando una scheda Arduino è avviata, il bootloader viene eseguito, ritardando di pochi secondi l'avvio di un eventuale sketch presente in memoria. All'avvio, il bootloader verifica la presenza di comunicazioni inerenti ad eventuali sketch da caricare. In caso di uno sketch da caricare, il bootloader effettuerà il trasferimento dello sketch sulla memoria del microcontrollore. Una volta terminato il trasferimento, il bootloader avvierà lo sketch appena caricato. Nel caso in cui non ci fossero comunicazioni, il bootloader provvederà ad avviare l'ultimo sketch caricato in memoria.

5.3 Shield

Le shield sono delle schede compatibili con alcuni modelli di Arduino e, grazie ad esse, è possibile estendere le funzionalità base che fornisce la scheda di prototipazione.
Tramite le shield è possibile connettere Arduino a una rete WI-FI, collegarlo a una rete GSM, controllare attuatori in modo più semplice, etc.

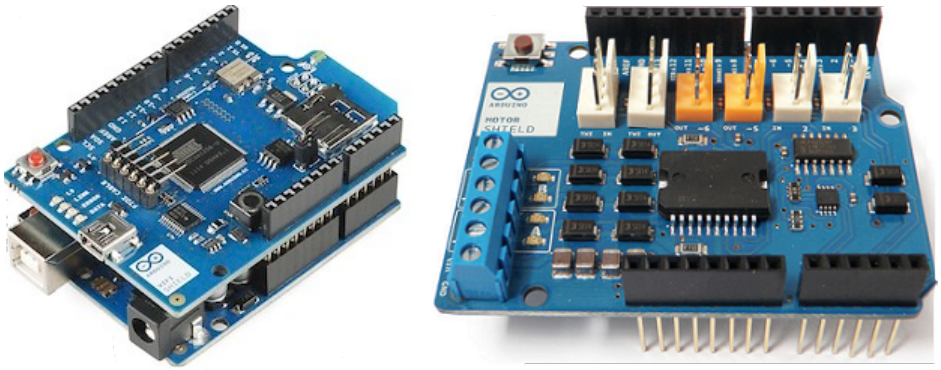

Figura 5.2: A sinistra un Arduino dotato di shield WI-FI. A destra una shield per il controllo dei motori

5.4 Hello World

Come di consueto, procederemo a illustrare il classico "Hello World" per Arduino. Per quanto sia possibile mostrare in output la stringa "Hello World", per fornire una visione delle funzionalità di I/O di Arduino, verrà fatto lampeggiare un led. Nella figure seguenti è possibile notare il codice e lo schema di realizzazione.

```
void setup()
{
  pinMode(13,OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  digitalWrite(13,HIGH);
  Serial.println("Led acceso");
  delay(1000);
  digitalWrite(13,LOW);
  Serial.println("Led spento");
  delay(1000);
}
```

Figura 5.3: Codice

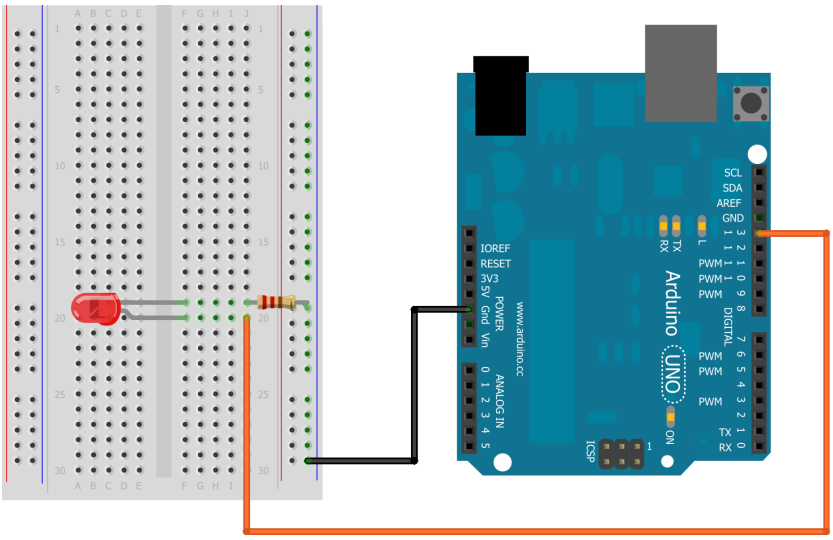

Figura 5.4: Schema circuito

Come detto in precedenza, per creare un programma per Arduino, lo sketch deve contenere almeno due funzioni: setup e void.
Nella funzione setup, tramite l'istruzione *pinMode*, si dichiara l'intenzione di voler utilizzare il pin digitale 13 in modalità di output. L'istruzione *Serial.begin* stabilisce una connessione di tipo seriale; il parametro indica il

baud rate.
Siccome non è possibile eseguire il debug su Arduino, in fase di realizzazione degli sketch, la comunicazione seriale con il pc viene spesso utilizzata per simulare una sorta di debug, visualizzando ad esempio lo stato delle variabili.
Nella funzione loop, con l'istruzione digitalWrite, è possibile impostare lo stato di un pin digitale; *HIGH* o *LOW*. Quando si imposta lo stato *HIGH*, Arduino fa passare corrente al pin 13, erogando una tensione di 5V e 20mA di corrente, accendendo di fatto il led. Utilizzando *LOW*, Arduino non lascia passare corrente al pin, spegnendo il led.
L'istruzione *delay* sospende l'esecuzione della funzione loop. Il parametro indica il numero di millisecondi.
Come intuibile, *Serial.println* permette di stampare a video.

Capitolo 6

Caso di studio: Applicazione di Jason su Arduino

Dopo aver definito il paradigma ad agenti, all'interno di questo capitolo verrà mostrata l'applicazione di tale metodologia sulla scheda di prototipazione Arduino. In particolare, verrà realizzata una libreria per consentire la realizzazione di un qualsiasi sistema embedded.
Anche se Arduino dispone di un proprio linguaggio di programmazione, si vuole pilotare la scheda tramite Jason per due motivi. Il primo motivo consiste nella realizzazione di un sistema ad agenti nell'ambito dei sistemi embedded. Il secondo motivo riguarda gli aspetti che un paradigma è in grado di modellare nella realizzazione di un sistema embedded. Tramite la programmazione ad agenti, è possibile modellare alcuni aspetti che non è possibile catturare attraverso la programmazione procedurale o ad oggetti, come la pro-attività e la reattività.
Come detto nel capitolo precedente, il framework per la programmazione di Arduino si chiama Wiring; un derivato di C/C++. Siccome Jason è sviluppato in Java, è impossibile mandarlo in esecuzione su Arduino. Come soluzione al problema, Arduino verrà pilotato da remoto da un programma Jason. Nello specifico verrà realizzata una libreria lato host, eseguendo su Arduino un codice che provvederà a ricevere i comandi e ad inviare le infor-

mazioni riguardanti i pin.
Per permettere la comunicazione tra l'host e Arduino, con la più bassa latenza possibile, verrà adottata una soluzione cablata, utilizzando la porta USB in dotazione su Arduino.
Nonostante Arduino presenti caratteristiche che non consentono l'applicazione diretta di Jason, si è scelto di utilizzare questa scheda di prototipazione poiché è una delle più utilizzate attualmente.
All'interno di questo capitolo, oltre a mostrare lo sviluppo della libreria, verrà mostrato anche il suo utilizzo a titolo di esempio.

6.1 Firmata

Per pilotare Arduino da un host, come un computer ad esempio, è necessario stabilire un protocollo di comunicazione. Poiché questo aspetto non è fondamentale ai fini della trattazione, si è scelto di non implementare un proprio protocollo di comunicazione, ma di adottarne uno sviluppato da terzi: Firmata [6].
Firmata è un protocollo per la comunicazione tra microcontrollori e software in esecuzione su un host (computer, smartphone, tablet). La comunicazione avviene tramite messaggi formato MIDI, sebbene Firmata non ne implementi il protocollo. Firmata può essere implementato su qualsiasi microcontrollore e su qualsiasi host. Per l'implementazione del protocollo lato host, attualmente sono disponibili varie librerie [7]. A seconda della libreria utilizzata, la comunicazione tra l'host e il microcontrollore può avvenire via cavo, wifi o ethernet.
Per lo sviluppo della libreria è stata utilizzata la libreria Firmata4j [8].

6.2 Realizzazione libreria

Per la realizzazione della libreria, è di fondamentale importanza determinare il comportamento della scheda Arduino. Si è scelto di programmare

la scheda in modo tale da avere un unico sketch compatibile con qualsiasi sistema ad agenti. L'ide di Arduino fornisce degli sketch di esempio e, tra questi, ne è presente uno riguardante Firmata [9]. Tale sketch risulta essere perfetto per le nostre necessità, in quanto il programma consiste nell'invio dei valori dei pin e nell'esecuzione dei comandi ricevuti.
La libreria è composta da più artefatti. Un artefatto rappresenta Arduino, mentre i restanti consentono di pilotare sensori e attuatori, come: led, sensori di prossimità, interruttori, etc. Inoltre, come visibile nel diagramma delle classi, sono state realizzate delle semplici classi Java per favorire lo sviluppo delle funzionalità dell'artefatto che rappresenta Arduino.
L'artefatto che rappresenta la scheda di prototipazione, è dotato di alcune operazioni che permettono di pilotare Arduino, utilizzabili sia da altri artefatti che da agenti. Si vuole permettere l'utilizzo delle operazioni ad altri artefatti in modo che essi possano rappresentare i trasduttori collegati alla scheda, ottenendo così un livello di astrazione superiore, favorendone inoltre la modularità e la riusabilità.
Siccome è impossibile rappresentare tutti i sensori e attuatori presenti sul mercato, nella libreria ne sono stati modellati solo alcuni. Tramite le operazioni presenti nella link interface dell'artefatto che rappresenta il device, è possibile modellare un particolare sensore o attuatore.
La libreria è composta dai seguenti artefatti: ArduinoBoard, IRSensor, Led, Switch, Servo.
Prima di procedere con l'analisi degli artefatti che compongono la libreria, si ricorda che le classi utilizzate fanno riferimento alla libreria Firmata4j [8].

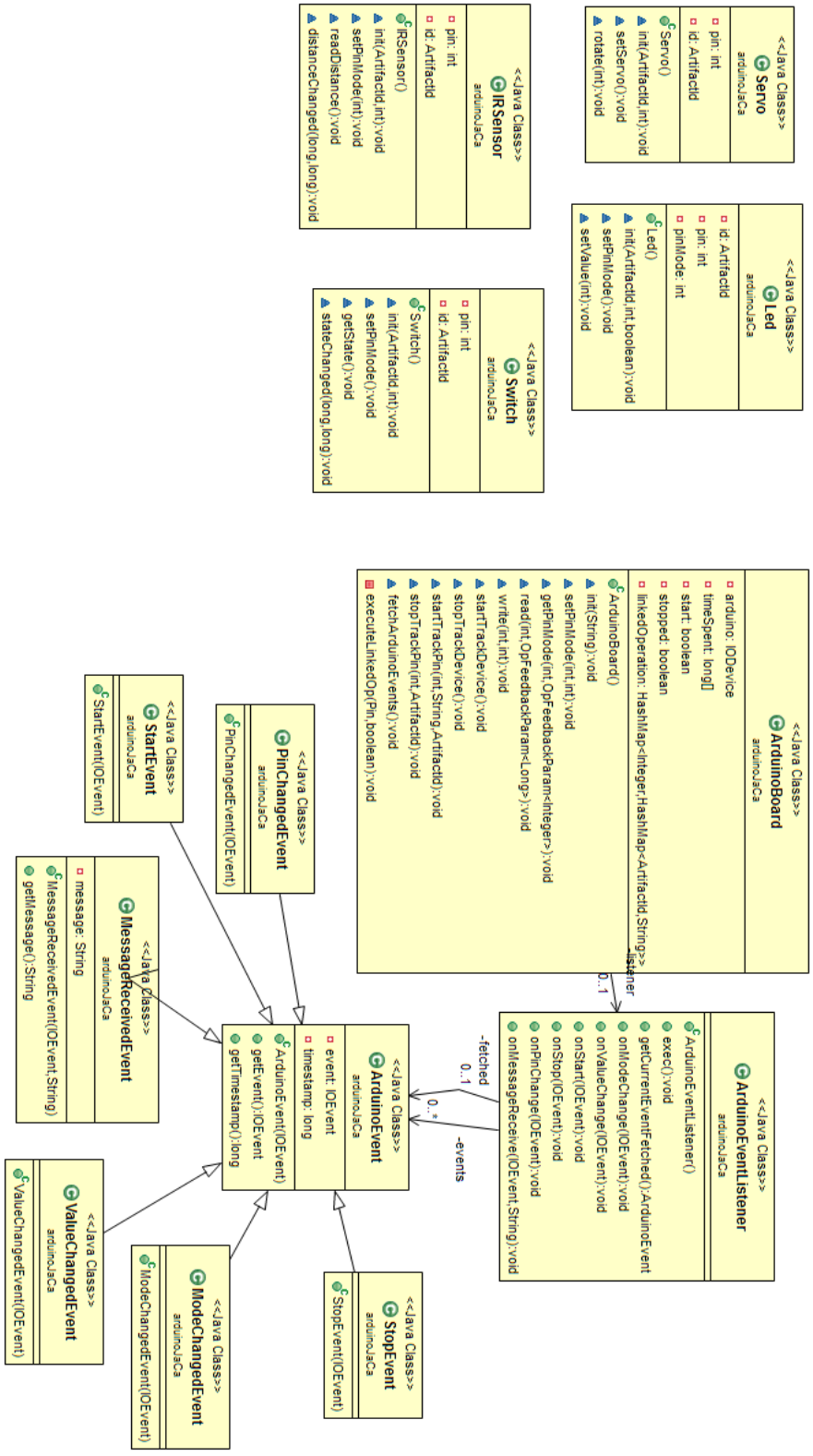

Figura 6.1: Diagramma delle classi

6.2.1 ArduinoBoard

Questo artefatto rappresenta il cuore pulsante della libreria, poiché fornisce le operazioni necessarie per pilotare i pin della scheda.
Per avere il pieno controllo su Arduino tramite Jason, l'artefatto deve essere dotato delle seguenti funzionalità: connessione al device, lettura/scrittura dei pin, impostazione della modalità dei pin (input, output, pwm, analog) e aggiunta/rimozione dei listener.

```
public class ArduinoBoard extends Artifact  {

    private IODevice arduino=null;
    private long[] timeSpent;
    private boolean start=false;
    private ArduinoEventListener listener;
    private boolean stopped;
    private HashMap<Integer,HashMap<ArtifactId,String>> linkedOperation=
            new HashMap<>();

    void init(String port) {

        arduino = new FirmataDevice(port);
        stopped = false;
        try {
            arduino.start();
            arduino.ensureInitializationIsDone();
        } catch (IOException | InterruptedException e) {
          System.out.println("!! ERROR: device start failed.");
          e.printStackTrace();
        }

        for(int i=0;i<arduino.getPinsCount();i++){
            defineObsProperty("pin"+i+"Mode",Mode.INPUT.ordinal());
            defineObsProperty("pin"+i+"Value",(long)-1);
            defineObsProperty("pin"+i+"TimeSpent",(long)-1);
        }

        timeSpent= new long[arduino.getPinsCount()];
        listener = new ArduinoEventListener();
        execInternalOp("fetchArduinoEvents");

    }

    ...
```

Figura 6.2: Stabilimento connessione

Alla creazione dell'artefatto viene stabilita una connessione con il microcontrollore, indicando la porta sulla quale esso è presente (es: COM4). Inoltre, una volta stabilita la connessione, per ogni pin vengono definite tre proprietà osservabili, che tengono traccia del tempo impiegato per l'esecuzione dei co-

mandi, del cambio di modalità e del valore del pin.
Nel caso in cui non fosse possibile stabilire una connessione, verrà generata un'eccezione.
Come visibile, viene istanziato un oggetto di tipo *ArduinoEventListener*, che verrà utilizzato come listener per poter ascoltare gli eventi generati da Arduino. L'operazione interna effettua il fetch degli eventi generati da *ArduinoEventListener*.

```
...

@LINK
@OPERATION
void setPinMode(int pin, int mode){
    try
    {
        if(arduino!=null){

            if(mode==Mode.INPUT.ordinal()){
                arduino.getPin(pin).setMode(Mode.INPUT);
            }else if(mode==Mode.OUTPUT.ordinal()){
                arduino.getPin(pin).setMode(Mode.OUTPUT);
            }else if(mode==Mode.PWM.ordinal()){
                arduino.getPin(pin).setMode(Mode.PWM);
            }else if(mode==Mode.ANALOG.ordinal()){
                arduino.getPin(pin).setMode(Mode.ANALOG);
            }else if(mode==Mode.SERVO.ordinal()){
                arduino.getPin(pin).setMode(Mode.SERVO);
            }

            if(start){
                getObsProperty("pin"+pin+"Mode").updateValue(mode);
                getObsProperty("pin"+pin+"Value").updateValue(arduino.getPin(pin).getValue());
                timeSpent[pin]=System.nanoTime();
            }

        }
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }

}

...
```

Figura 6.3: Operazione che permette di impostare lo stato dei pin

Stabilita la connessione, è possibile utilizzare le altre funzionalità, come, ad esempio, l'operazione *setPinMode*. Tramite questa operazione è possibile impostare la modalità di funzionamento di un determinato pin. Le modalità *INPUT* e *OUTPUT* impostano il pin in modalità di I/O digitale. *PWM* imposta un pin digitale in modalità di output con modulazione di frequenza. *SERVO* imposta un pin in modalità di output in modo da pilotare dei

servomotori. Per concludere, *ANALOG* imposta il pin in modalità di input analogico.
Siccome un pin analogico può anche essere utilizzato come un pin digitale, tramite la modalità *INPUT* o *OUTPUT*, è possibile utilizzare i pin analogici come dei pin digitali di I/O.
Qualora venisse impostata una modalità non supportata da un pin (es: modalità pwm su un pin analogico), verrà generata un'eccezione.
Per quanto concerne il parametro *pin*, l'intero rappresenta il numero del pin sul dispositivo. Ad esempio:

arduino.getPin(3);

arduino.getPin(14);

Prendendo in considerazione la scheda Arduino Uno, la prima istruzione restituirà il terzo pin digitale, mentre la seconda istruzione restituirà il primo pin analogico.

```
...

@LINK
@OPERATION
void read(int pin, OpFeedbackParam<Long> value){
    if(arduino!=null && pin>=0 && pin<arduino.getPinsCount()){
        value.set(arduino.getPin(pin).getValue());
    }
}

@LINK
@OPERATION
void write(int pin, int value){
    if(arduino!=null){
        Pin pinArduino = arduino.getPin(pin);
        Mode mode = pinArduino.getMode();
        try{
            if((mode==Mode.OUTPUT && (value==0 || value==1))||
                    ((mode==Mode.PWM && value>=0 && value<=255)||mode==Mode.SERVO)){
                pinArduino.setValue(value);

                if(start){
                    timeSpent[pin]=System.nanoTime();
                }
            }
        }catch (IllegalStateException e) {
            e.printStackTrace();
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
}

...
```

Figura 6.4: Operazioni di lettura e scrittura dello stato dei pin

Come si evince dalla figura 6.4, tramite le operazioni *read* e *write* è possibile leggere e modificare lo stato di un pin. L'operazione *read* definisce un parametro di output che rappresenta il valore del pin.
Comportamento del tutto simile per quanto riguarda l'operazione *write*. Qualora venisse impostato un valore non compatibile con la modalità del pin (es: assegnare il valore 255 ad un pin impostato come output digitale, oppure assegnare un valore ad un pin di input), verrà generata un'eccezione.
Oltre alle funzionalità appena descritte, ArduinoBoard offre la possibilità di aggiungere e rimuovere due listener, che consentono di percepire tutti gli eventi generati dal device e/o da un pin specifico.
Prima di procedere con l'analisi delle operazioni che consentono a un agente di aggiungere un listener, è necessario analizzare la classe *ArduinoEventListener*

```
public class ArduinoEventListener implements IBlockingCmd, IODeviceEventListener, PinEventListener  {

    private BlockingQueue<ArduinoEvent> events;
    private ArduinoEvent fetched;

    public ArduinoEventListener(){
        events = new java.util.concurrent.ArrayBlockingQueue<ArduinoEvent>(100);
        fetched = null;
    }

    public void exec(){
        try {
            fetched = events.take();
        } catch (Exception ex){
        }
    }

    public ArduinoEvent getCurrentEventFetched(){
        return fetched;
    }

    ...
```

Figura 6.5: Classe ArduinoEventListener

Come si evince dalla figura 6.5, la classe implementa le interfacce *IODeviceEventListener*, *PinEventListener* e *IBlockingCmd*.
La classe implementa una coda in cui verranno inseriti gli eventi generati da Arduino.
L'interfaccia *IBlockingCmd* definisce il metodo *exec*, il quale si pone in attesa di ricevere gli eventi dalla coda.

```
...

@Override
 public void onModeChange(IOEvent event) {
    try {
        events.put(new ModeChangedEvent(event));
    } catch (Exception ex){}
}

@Override
public void onValueChange(IOEvent event) {
    try {
        events.put(new ValueChangedEvent(event));
    } catch (Exception ex){}
}

@Override
public void onStart(IOEvent event) {
    try {
        events.put(new StartEvent(event));
    } catch (Exception ex){}
}

@Override
public void onStop(IOEvent event) {
    try {
        events.put(new StopEvent(event));
    } catch (Exception ex){}
}

@Override
public void onPinChange(IOEvent event) {
    try {
        events.put(new PinChangedEvent(event));
    } catch (Exception ex){}
}

@Override
public void onMessageReceive(IOEvent event, String message) {
    try {
        events.put(new MessageReceivedEvent(event,message));
    } catch (Exception ex){}
}
}
```

Figura 6.6: Classe ArduinoEventListener

L'interfaccia *IODeviceEventListener* definisce i metodi: *onStart*, *onStop*, *onPinChange*, *onMessageReceive*. Al verificarsi di un evento inerente al device, verrà invocato uno dei quattro metodi.

I metodi *onStart* e *onStop* verranno invocati alla connessione e alla disconnessione di Arduino dall'host.

Il metodo *onPinChange* verrà invocato quando si verificherà una modifica a un pin del device.

Il metodo *onMessageReceive* verrà invocato quando Arduino invierà un mes-

saggio all'host.

L'interfaccia *PinEventListener* definisce i metodi *onModeChange* e *onValueChange*. Al verificarsi di un evento inerente a un determinato pin, verrà invocato uno dei due metodi.

Il metodo *onModeChange* verrà invocato quando si verificherà un cambio di modalità del pin (es: da output a input).

Il metodo *onValueChange* verrà invocato quando si verificherà un cambio di valore del pin.

Come si evince dalla figura 6.6, al verificarsi di uno degli eventi appena descritti, esso verrà aggiunto alla coda.

Nella figura seguente sono presenti le operazioni che permettono a un agente l'aggiunta e la rimozione dei listener inerenti al device e ai pin.

```
...

@OPERATION
void startTrackDevice(){
    if(arduino!=null){
        arduino.addEventListener(listener);
        start=true;
    }
}

@OPERATION
void stopTrackDevice(){
    if(arduino!=null){
        arduino.removeEventListener(listener);
        start=false;
    }
}

@OPERATION
void startTrackPin(int pin, String methods,ArtifactId id){
    if(arduino!=null){
        arduino.getPin(pin).addEventListener(listener);
        if(linkedOperation.containsKey(pin)){
            HashMap<ArtifactId,String> tmp=linkedOperation.get(pin);
            tmp.put(id,methods);
            linkedOperation.put(pin,tmp);
        }else{
            HashMap<ArtifactId,String> tmp= new HashMap<>();
            tmp.put(id,methods);
            linkedOperation.put(pin,tmp);
        }
    }
}

@OPERATION
void stopTrackPin(int pin,ArtifactId id){
    if(arduino!=null){
        arduino.getPin(pin).removeEventListener(listener);
        HashMap<ArtifactId,String> tmp=linkedOperation.get(pin);
        if(tmp!=null){
            tmp.remove(id);
        }
    }
}
```

Figura 6.7: Operazioni aggiunta/rimozione listener

Per entrambi i listener viene passato come parametro l'oggetto della classe *ArduinoEventListener*.
Per l'aggiunta di un listener su un pin specifico, l'operazione *startTrackPin*, oltre al parametro *pin*, richiede l'*id* di un artefatto e una stringa contenente i nomi delle operazioni da invocare. La stringa dovrà contenere i nomi di due operazioni e dovrà rispettare la seguente sintassi:

operazione_cambio_modalità:operazione_cambio_valore

Le operazioni dovranno essere presenti nella link interface dell'artefatto corrispondente al parametro *id*. In questo modo un artefatto potrà ascoltare gli eventi generati dai pin e, in base all'evento verificatosi (*onModeChange* o *onValueChange*), potrà mettere in atto una serie di operazioni. Qualora non si volesse implementare una delle due operazioni, la stringa dovrà rispettare la seguente sintassi:

null:operazione_cambio_valore o *operazione_cambio_modalità:null*

Inoltre, le operazioni da implementare dovranno accettare due parametri long, che indicano rispettivamente il valore o la modifica del pin e la data, espressa in nanosecondi, dell'invio del comando ad Arduino. L'invio della data permette di calcolare il tempo di esecuzione dell'operazione. Di seguito un esempio di come dovrà essere implementata l'operazione:

```
@LINK
void nome_operazione(long value,long startTime){
...
}
```

Come visto all'inizio del paragrafo, all'atto della creazione dell'artefatto, verrà eseguita un'operazione interna che effettua il fetch di eventuali eventi generati dai listener aggiunti al device e ai pin. Nella figura seguente è possibile notare l'implementazione di tale operazione.

```
...

@INTERNAL_OPERATION
void fetchArduinoEvents(){
    while (!stopped){
        await(listener);
        ArduinoEvent aev = listener.getCurrentEventFetched();

        if (aev instanceof PinChangedEvent){
            Pin pin = aev.getEvent().getPin();
            int npin = pin.getIndex();
            ObsProperty mode = getObsProperty("pin"+npin+"Mode");
            ObsProperty value = getObsProperty("pin"+npin+"Value");

            int currMode = pin.getMode().ordinal();
            if ((int)mode.getValue()!=currMode){
                mode.updateValue(currMode);
                getObsProperty("pin"+npin+"TimeSpent").updateValue(System.nanoTime()-timeSpent[pin.getIndex()]);
            }

            long currValue = pin.getValue();
            if ((long)value.getValue()!=currValue){
                value.updateValue(currValue);
                if(!pin.getMode().equals(Mode.ANALOG) && !pin.getMode().equals(Mode.INPUT)){
                    getObsProperty("pin"+npin+"TimeSpent").updateValue(System.nanoTime()-timeSpent[pin.getIndex()]);
                }
            }
        }else if (aev instanceof ModeChangedEvent){
            executeLinkedOp(aev.getEvent().getPin(),true);
        } else if (aev instanceof ValueChangedEvent){
            executeLinkedOp(aev.getEvent().getPin(),false);
        } else if (aev instanceof StartEvent){
            signal("device_connected");
        } else if (aev instanceof StopEvent){
            signal("device_disconnected");
        } else if (aev instanceof MessageReceivedEvent){
            signal(((MessageReceivedEvent)aev).getMessage());
        }
    }
}

private void executeLinkedOp(Pin pin,boolean modeChanged){
    int index=modeChanged?0:1;
    int npin = pin.getIndex();
    HashMap<ArtifactId,String> tmp= linkedOperation.get(npin);
    if(tmp!=null){
        for (ArtifactId key: tmp.keySet()){
            String method = tmp.get(key).split(":")[index];
            if(!method.equals("null")){
                long time = (!modeChanged &&!pin.getMode().equals(Mode.INPUT) &&
                        !pin.getMode().equals(Mode.ANALOG)) || modeChanged?timeSpent[npin]:-1;

                try {
                    execLinkedOp(key,method,pin.getValue(),time);
                } catch (OperationException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

...
```

Figura 6.8: Operazione interna FetchArduinoEvents

Il metodo *await* accetta un oggetto di una classe che implementa l'interfaccia *IBlockingCmd*. La primitiva *await* blocca il flusso delle operazioni fino a quando il metodo *exec* non avrà terminato la propria esecuzione. In questo modo, l'operazione interna si pone in ascolto degli eventi generati dal listener e li processerà uno ad uno, evitando corse critiche e, garantendo così, la corretta esecuzione delle operazioni successive.
A seconda del tipo di evento verrà intrapreso un corso di azioni differente. Si ricorda che gli eventi *StartEvent*, *StopEvent*, *PinChangedEvent* e *MessageReceivedEvent* sono inerenti agli eventi generati dal listener del device, mentre *ModeChangedEvent* e *ValueChangedEvent* a un pin specifico.
Per gli eventi *StartEvent*, *StopEvent* e *MessageReceivedEvent* verrà generato un signal.
Al verificarsi dell'evento *PinChangedEvent* verranno modificate le proprietà osservabili inerenti al pin specifico. In caso di modifica del valore del pin, il tempo di esecuzione dell'operazione verrà aggiornato solo nel caso in cui il pin sia impostato in una delle modalità di output. Per gli eventi *ModeChangedEvent* e *ValueChangedEvent* verranno invocate le operazioni presenti nella link interface degli artefatti posti in ascolto del pin che ha generato l'evento.

6.2.2 Sensori e attuatori

Gli artefatti che rappresentano i vari trasduttori, sono del tutto simili tra loro. All'interno di questo paragrafo verrà mostrata l'implementazione di ciascuno di esso, analizzando gli aspetti particolari che li contraddistinguono.
Nella figura seguente è possibile notare l'implementazione di un artefatto che modella l'utilizzo di uno degli attuatori più comuni: il led.
Come si evince dalla figura, il metodo *init* presenta tre parametri. Come intuibile, il parametro *pin* indica il pin a cui è connesso il led, mentre il parametro *pwmMode* indica se l'uscita del connettore dovrà essere impostata come digitale o in modalità pwm. Si presti particolare attenzione al parametro *id*. Siccome l'artefatto utilizza le operazioni messe a disposizione da *ArduinoBoard*, nell'utilizzo delle operazioni presenti nella link interface, è ne-

cessario indicare l'id dell'artefatto a cui fare riferimento.

```
public class Led extends Artifact {
    private ArtifactId id;
    private int pin;
    private int pinMode;

    void init(ArtifactId arduinoId,int pin,boolean pwmMode) {
        this.id=arduinoId;
        this.pin=pin;
        pinMode = pwmMode==true ? Mode.PWM.ordinal() : Mode.OUTPUT.ordinal();
    }

    @OPERATION
    void setPinMode(){
        try {
            execLinkedOp(id, "setPinMode",pin,pinMode);
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

    @OPERATION
    void setValue(int value){

        try {
            execLinkedOp(id, "write",pin,value);
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

}
```

Figura 6.9: Artefatto Led

L'operazione *setPinMode* provvede a impostare lo stato del pin, tramite l'utilizzo dell'operazione presente nella link interface di ArduinoBoard. Trattandosi di un attuatore, è possibile impostare solo due stati: uscita digitale o uscita pwm. Tramite l'operazione *setValue* è possibile pilotare lo stato del led. Il valore che può assumere dipende dalla modalità: 0 o 1 per l'uscita digitale, da 0 a 255 per la modalità pwm.
Nella figura 6.10 è possibile notare l'implementazione generica di un servomotore. Non è necessario richiedere la modalità del pin all'atto della creazione dell'artefatto, in quanto la libreria Firmata supporta la modalità servo. Tramite questa modalità, è sufficiente impostare il valore (positivo o negativo) che si intende assegnare per effettuare la rotazione del servomotore.

```
public class Servo extends Artifact {

    private int pin;
    private ArtifactId id;

    void init(ArtifactId arduinoId,int pin) {
        this.pin=pin;
        this.id=arduinoId;
    }

    @OPERATION
    void setServo(){
        try {
            execLinkedOp(id, "setPinMode",pin,Mode.SERVO.ordinal());
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

    @OPERATION
    void rotate(int value){
        try {
            execLinkedOp(id, "write",pin,value);
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

}
```

Figura 6.10: Artefatto Servo

```
public class Switch extends Artifact {
    private int pin;
    private ArtifactId id;

    void init(ArtifactId arduinoId,int pin) {
        this.pin=pin;
        this.id=arduinoId;
    }

    @OPERATION
    void setPinMode(){
        try {
            execLinkedOp(id, "setPinMode",pin,Mode.INPUT.ordinal());
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

    @OPERATION
    void getState(OpFeedbackParam<Long> result){
        try {
            execLinkedOp(id, "read",pin,result);
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

    @LINK
    void stateChanged(long value,long startTime){
        String tmp=value==0?"off":"on";
        signal(tmp);
    }

}
```

Figura 6.11: Artefatto Switch

Dopo aver analizzato l'implementazione di due artefatti che modellano il comportamento di due attuatori, nella figura 6.11 è possibile notare l'implementazione di uno switch. Lo switch non è altro che un interruttore dotato di due stati: on e off.
Trattandosi di un sensore digitale, non è necessario richiedere la modalità del pin, ma verrà impostata automaticamente la modalità di input digitale.
Tramite l'operazione *getState* è possibile ottenere lo stato dello switch. Una volta terminata l'esecuzione dell'operazione presente nella link interface dell'artefatto ArduinoBoard, verrà impostato il valore del parametro di output.
Come detto nel paragrafo precedente, ArduinoBoard permettere di aggiungere un listener su un pin specifico, a condizione che venga specificata un'operazione nella link interface dell'artefatto che modella il sensore/attuatore collegato. Nella figura 6.11 è possibile notare l'implementazione di tale operazione; *stateChanged*. Ogni qualvolta che verrà eseguita questa operazione da ArduinoBoard, verrà generato un signal.
Per concludere, l'ultimo artefatto che compone la libreria, modella il comportamento di un sensore di prossimità IR. A differenza dello switch, è necessario indicare la modalità del pin, in quanto in commercio sono disponibili sia sensori analogici che digitali. Analogamente allo switch, sono state definite due operazioni che permettono la lettura del sensore. L'operazione *readValue*, permette la lettura del sensore quando lo si desidera, mentre l'operazione *distanceChanged* verrà utilizzata da ArduinoBoard nel caso in cui venisse aggiunto un listener riguardante il pin a cui è connesso il sensore.

```
public class IRSensor extends Artifact {
    private int pin=0;
    private ArtifactId id;

    void init(ArtifactId arduinoId,int pin) {
        defineObsProperty("irValue",-1);
        this.pin=pin;
        this.id=arduinoId;
    }

    @OPERATION
    void setPinMode(int mode){
        try {
            execLinkedOp(id, "setPinMode",pin,mode);
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

    @OPERATION
    void readValue(OpFeedbackParam<Long> result){
        try {
            execLinkedOp(id, "read",pin,result);
        } catch (OperationException e) {
            e.printStackTrace();
        }
    }

    @LINK
    void distanceChanged(long value,long startTime){
        getObsProperty("irValue").updateValue(value);
    }
}
```

Figura 6.12: Artefatto IRSensor

6.3 Utilizzo della libreria

Dopo aver mostrato gli artefatti che compongono la libreria e che consentono, di fatto, di applicare la programmazione ad agenti su una scheda Arduino, all'interno di questo paragrafo verrà mostrata un'applicazione pratica di quanto fatto in precedenza.

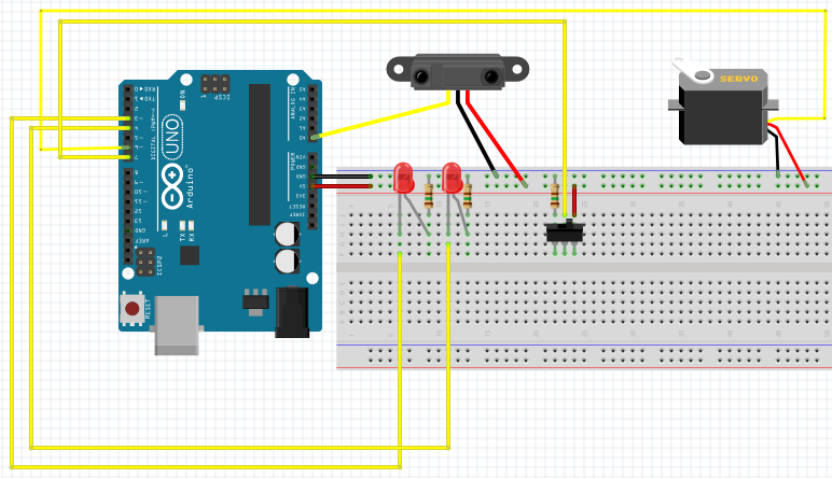

Figura 6.13: Schema circuito

A titolo di esempio, si vuole realizzare un sistema che, a seconda della distanza rilevata da un sensore IR, regoli la luminosità di un led. Inoltre, a seconda dello stato di un interruttore (on/off), verrà regolata l'accensione/spegnimento di un secondo led e l'azionamento di un servomotore.
Come già detto, per quanto concerne il codice inerente ad Arduino, si è scelto di adottare uno sketch di esempio del protocollo Firmata. Per ulteriori informazioni consultare [9].
Nella figura 6.14 è possibile notare l'implementazione dell'agente. L'agente perseguirà subito l'obiettivo *start*. Il piano è costituito dalla creazione dei vari artefatti; secondo i parametri illustrati nel paragrafo precedente. Una volta creati gli artefatti, si procede con l'esecuzione delle operazioni necessarie a impostare le modalità dei vari pin. Si noti l'utilizzo dell'annotation, in quanto più artefatti definiscono l'operazione *setPinMode*.
Tramite la primitiva *focus*, l'agente osserva gli artefatti inerenti al sensore IR, allo switch e al device. In particolare, tramite l'operazione *startTrackDevice*, come visto nel paragrafo precedente, viene aggiunto un listener inerente al device. Poiché vi sono due sensori collegati ad Arduino e si desidera avere una lettura dei due trasduttori sempre aggiornata, tramite l'operazione *startTrackPin* si aggiunge un listener al pin specifico.

```
/* Initial goals */
!start.

/* Plans */
+!start : true
 <- makeArtifact("arduinoArtifact","arduinoJaCa.ArduinoBoard",["COM4"],IdArduinoBoard);
 makeArtifact("irSensor","arduinoJaCa.IRSensor",[IdArduinoBoard,14],IdSensor);
 makeArtifact("ledpwm","arduinoJaCa.Led",[IdArduinoBoard,3,true],IdLedPwm);
 makeArtifact("led","arduinoJaCa.Led",[IdArduinoBoard,4,false],IdLed);
 makeArtifact("switch","arduinoJaCa.Switch",[IdArduinoBoard,7],IdSwitch);
 makeArtifact("servo","arduinoJaCa.Servo",[IdArduinoBoard,6],IdServo);
 setServo;
 setPinMode [artifact_name("ledpwm")];
 setPinMode [artifact_name("led")];
 setPinMode(2) [artifact_name("irSensor")]; //analog mode
 setPinMode [artifact_name("switch")];
 focus(IdArduinoBoard);
 focus(IdSensor);
 focus(IdSwitch);
 startTrackDevice;
 startTrackPin(14,"null:distanceChanged",IdSensor);
 startTrackPin(7,"null:stateChanged",IdSwitch).

+irValue(V): V>=208 & V<=521
<-setValue(math.round(255/((2087/V)-3))) [artifact_name("ledpwm")].

+irValue(V): V<208 | V>521
<-setValue(0) [artifact_name("ledpwm")].

+on
<-rotate(-10);
setValue(1) [artifact_name("led")].

+off
<-rotate(10);
setValue(0) [artifact_name("led")].

+pin4Value(V): V==1
<-println("Led : on").

+pin4Value(V): V==0
<-println("Led : off").

+pin4TimeSpent(V)
<-println("Time spent in nanoseconds : ",V).
```

Figura 6.14: Implementazione agente

Come visto in precedenza, *irValue* identifica la proprietà osservabile definita all'interno dell'artefatto IRSensor, mentre *pin4Value* e *pin4TimeSpent* identificano le proprietà osservabili definite all'interno dell'artefatto ArduinoBoard. I signal generati dall'artefatto Switch sono rappresentati da *on* e *off*.

Ad ogni modifica della proprietà osservabile *irValue*, a seconda del valore della lettura del sensore, verrà regolata la luminosità del led collegato al pin 3. Qualora il sensore dovesse restituire un valore inferiore a 208 o maggiore a 521, il led verrà spento.

Come si evince dalla figura, a seconda del segnale generato dall'artefatto Switch, verrà acceso o spento il led collegato al pin 4, oltre all'azionamento del servomotore. Inoltre, sono stati definiti dei piani riguardanti le proprietà osservabili dell'artefatto ArduinoBoard e, ad ogni cambiamento di stato del led collegato al pin 4, verrà stampato a video lo stato corrente del led e il

tempo impiegato nell'esecuzione del comando di modifica del valore del pin. Nella figura seguente è possibile notare la stampa a video dello stato del led e del tempo di esecuzione.

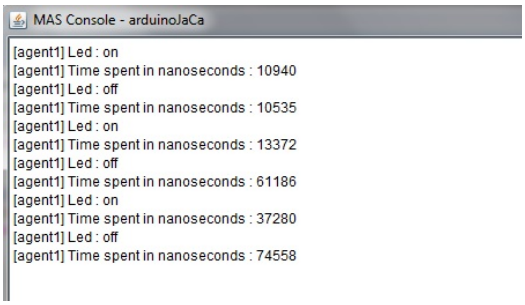

Figura 6.15: Tempo di esecuzione del cambio di stato del pin da controllo remoto

Per avere un confronto dei tempi di esecuzione, su Arduino è stato eseguito uno sketch che pilotasse direttamente lo stato di un pin. Inoltre, per permettere un confronto equo, per entrambe le rilevazioni è stata utilizzata la scheda Arduino uno. Di seguito è possibile trovare lo sketch e i relativi tempi di esecuzione.

```
unsigned long int startTime,endTime;
int state;
String stateLed;

void setup() {

  pinMode(4,OUTPUT);
  state=LOW;
  digitalWrite(4,state);
  Serial.begin(9600);

}

void loop() {

  state=digitalRead(4)==LOW?HIGH:LOW;
  stateLed=digitalRead(4)==LOW?"on":"off";

  startTime = micros();
  digitalWrite(4,state);
  endTime=micros();
  Serial.println("Led: "+stateLed);
  Serial.print("Time spent in microseconds: ");
  Serial.println(endTime-startTime);
  delay(500);
}
```

Figura 6.16: Sketch che esegue un cambio di stato del pin

```
Led : on
Time spent in microseconds : 8
Led : off
Time spent in microseconds : 8
Led : on
Time spent in microseconds : 8
Led : off
Time spent in microseconds : 8
Led : on
Time spent in microseconds : 8
Led : off
Time spent in microseconds : 8
Led : on
Time spent in microseconds : 8
Led : off
Time spent in microseconds : 12
```

Figura 6.17: Tempo di esecuzione del cambio di stato del pin da Arduino

Come si evince dalle immagini, la differenza tra i tempi non è trascurabile. Il risultato tutto sommato si può considerare accettabile, poiché l'ordine di grandezza è rimasto invariato.
Concludendo, l'applicazione di Jason su Arduino tramite controllo remoto, si può ritenere soddisfacente. Considerando i tempi di esecuzione nel pilotare lo stato di un pin, tale approccio non potrà essere utilizzato in sistemi hard real time, ma può essere considerata una valida alternativa nella realizzazione di sistemi dove il tempo non riveste un ruolo chiave.

Conclusioni

Il paradigma ad agenti permette la realizzazione di sistemi sempre più complessi, utilizzando un livello di astrazione più elevato. Inoltre, a differenza di altre metodologie, permette di modellare alcuni aspetti del sistema che non è possibile esprimere con altri paradigmi. Tale paradigma consente di descrivere il sistema in termini di obiettivi e credenze; piuttosto che in termini di operazioni e proprietà.
Sebbene tale metodologia di programmazione sia stata sviluppata inizialmente nell'ambito dell'intelligenza artificiale, è possibile applicare la programmazione ad agenti nella realizzazione di sistemi embedded. In particolare, all'interno di questa trattazione è stata affrontata l'applicazione di Jason su Arduino, proponendo, attraverso il controllo remoto della scheda di prototipazione, un metodo per l'applicazione del paradigma ad agenti in ambito embedded.
L'applicazione del paradigma ad agenti fornisce astrazioni di prima classe, che consentono di modellare aspetti fondamentali dei sistemi embedded, come la reattività, la pro-attività e l'autonomia. Inoltre, nella realizzazione di un sistema multi-agente, potrebbe essere necessario interagire con dei sistemi embedded e, tramite l'utilizzo della metodologia proposta, questo è possibile.
Come visto nell'esempio di utilizzo della libreria, la soluzione proposta presenta anche alcuni svantaggi. Poiché il controllo remoto introduce inevitabilmente un ritardo nell'esecuzione delle operazioni da parte di Arduino, tale metodo può essere utilizzato nella realizzazione di sistemi che non richiedono vincoli di reattività hard real time. Inoltre, a differenza del paradigma ad

oggetti, la metodologia ad agenti è stata sviluppata di recente e risulta essere ancora acerba.

In generale, l'applicazione del paradigma ad agenti in ambito embedded, può trovare applicazione in qualsiasi tipo di sistema, indipendentemente dai vincoli richiesti.

Oltre al metodo proposto, per la realizzazione di sistemi embedded è possibile utilizzare Jason senza l'ausilio di un controllo remoto. A seconda dell'architettura hardware del sistema, vi sono varie soluzioni.

Per i sistemi che utilizzano un SOC, su alcuni di essi, come Raspberry PI, è possibile applicare Jason direttamente.

Per i sistemi che utilizzano un microcontrollore, invece, l'applicazione risulta essere più complessa. Il microcontrollore esegue le istruzioni in modo ciclico, proprio come l'interprete di Jason. A causa delle risorse limitate di un microcontrollore, su di esso si potrebbe implementare una versione ridotta dell'interprete di Jason, in modo da consentire l'applicazione del paradigma ad agenti.

Bibliografia

[1] Rafael H. Bordini, Jomi Fred Hubner, and Michael Wooldridge. *Programming multi-agent system in AgentSpeak using Jason*. Wiley, 2007.

[2] Alessandro Ricci, Michele Piunti, and Mirko Viroli. *Environment programming in multi-agent systems: an artifact based perspective. Autonomous Agents and Multi-Agent Systems*, 23(2):158?192, 2010.

[3] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Springler, second edition

[4] CArtAgO. *http://cartago.sourceforge.net*

[5] Arduino. *http://arduino.cc*

[6] Firmata. *http://firmata.org*

[7] Libreria Firmata. *https://github.com/firmata/protocol*

[8] Firmata4j. *https://github.com/kurbatov/firmata4j*

[9] Sketch protocollo Firmata. *https://github.com/firmata/arduino/tree/master/examples*