ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
con Sede a Cesena
Corso di Laurea Specialistica in Ingegneria Informatica

PROGETTAZIONE E SVILUPPO DI UN DSL AD AGENTI PER LA PIATTAFORMA ARDUINO

Elaborata nel corso di: Sistemi Multi Agente LS

Relazione di Progetto di:
DENIS BRIGHI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2010 – 2011
SESSIONE III

PAROLE CHIAVE

Embedded System

DSL - Domain Specific Language

Agenti BDI

Arduino

Agentino

Ai miei genitori
A Elisa

Indice

Introduzione

La proliferazione di embedded systems presenta una nuova sfida nell'ambito dei professionisti IT. L'utilizzo pervasivo di device in grado di controllare l'ambiente in cui noi viviamo e comunicare attraverso la rete sta spostando sempre più attenzione verso questi apparati. In particolare i cosiddetti Cyber-Physical System rappresentano la stretta integrazione tra elaborazione e processi fisici uniti in un processo in feedback. Le loro applicazioni – dai forni a microonde, automobili, smartphone, sistemi di sorveglianza, pacemaker – presentano alti gradi di criticità in termini di affidabilità, tempi di risposta certi, risparmio energetico e comunicazione mediante Internet che gli attuali modelli di programmazione non riescono a catturare. Tra le piattaforme embedded a più larga diffusione troviamo Arduino, che si colloca tra i prodotti come una scheda per la prototipazione rapida, user-friendly, ed una ergonomia di design unica nel suo genere ad un costo decisamente accessibile, adatta alla sperimentazione elettronica e sviluppo di nuovi oggetti.

L'attuale metodologia di programmazione (un linguaggio C/C++ corredato da librerie di supporto per L'I/O) permette innumerevoli funzioni, il cui costo viene pagato in termini di semplicità ed eleganza del linguaggio: di man in mano che il programma acquista corposità e funzionalità, il codice diventa sempre più complesso e di difficile manutenzione. Quando un programma tende a diventare complesso, è opportuno difatti avere delle astrazioni di primo piano e degli strumenti concettuali atti ad una programmazione avanzata ed agile, in modo da poter utilizzare meccanismi di alto livello, una migliore modularità, demandando invece ad uno strato software intermedio le operazioni più comuni e meno interessanti dal punto di vista implementativo.

Lo scopo di questa tesi è quindi quella di progettare un Domain-

Specific Language (DSL) che innalzi questo livello di astrazione della programmazione classica C/C++ sperimentando un approccio ad Agenti, introducendo questa astrazione in un campo ancora troppo legato a linguaggi di programmazione standard.

Approfondirò nel primo capitolo i cosiddetti Embedded Systems, con particolare riferimento ai Cyber-Physical Systems (CPS) definendoli, descrivendone le principali caratteristiche, applicazioni e gli attuali strumenti di programmazione più utilizzati. Nel secondo capitolo affronterò la tematica degli Agenti, come astrazione per nuovo paradigma di programmazione, mentre nel terzo capitolo descriverò la piattaforma Arduino, utilizzata per lo sviluppo della parte sperimentale di questa tesi. Infine, svilupperò un modello computazionale di agente che può essere implementato, tenendo conto delle limitazioni hardware imposte dalla piattaforma, descrivendone il processo semantico-deliberativo ed una possibile sintassi proposta a questo fine.

Capitolo 1

I Sistemi Embedded

Quando si parla di embedded system si intende un un apparato elettronico di elaborazione a microprocessore, progettato per il controllo di funzioni specifiche di un sistema attraverso una combinazione di hardware, software, sensori, attuatori ed altre parti meccaniche. Caratterizzati spesso da vincoli temporali real-time, i sistemi embedded sono, nella maggioranza dei casi, parte di un ambiente più esteso. Al contrario dei computer general purpose (come ad esemio un PC), questi device altamente integrati assolvono funzioni specifiche – note durante la fase di sviluppo – e sono, a tal fine, appositamente programmati ed ottimizzati: grazie a questo l'hardware può essere ridotto ai minimi termini per diminuire lo spazio occupato, riducendo così anche i consumi, i tempi di elaborazione (maggiore efficienza) ed il costo di fabbricazione. L'esecuzione del software inoltre è spesso in tempo reale (real-time) per consentire un controllo deterministico dei tempi di esecuzione. Possiamo trovare l'utilizzo di questi apparati in molti oggetti di uso comune quali: forni a microonde, telefoni cellulari, orologi digitali, missili teleguidati, ricevitori GPS, device per il monitoraggio cardiaco, stampanti, centraline di automobili, fax, fotocamere e in molti altri ambiti.

Questi sistemi altamente integrati contengono chip di elaborazione costituiti tipicamente da microcontrollori o Digital Signal Processor (DSP). In generale non è definibile un sistema integrato in termini di hardware utilizzato, di potenza computazionale o di memoria; sono infine considerati sistemi embedded anche dispositivi portatili dotati

di sistema operativo col quale eseguire diversi task.

1.1 Le Caratteristiche

I sistemi embedded sono sistemi di calcolo, nel significato più generale del termine: la maggior parte di questi è progettata per eseguire ripetutamente un'azione a costo contenuto, soddisfando vincoli temporali, come la necessità di operare in tempo reale. Può anche accadere che un sistema debba essere in grado di eseguire molto velocemente alcune funzioni, ma possa tollerare velocità inferiori per altre attività. Tipico esempio di quanto detto è un dispositivo mobile per il controllo del ritmo cardiaco: mentre abbiamo specifiche temporali stringenti sul monitoraggio del battito, possiamo ritenere di minore priorità – e dunque di frequenza di esecuzione – il controllo della batteria residua dello strumento.

Risulta difficile caratterizzare la velocità o i costi di un sistema embedded generico, anche se, soprattutto per sistemi che devono processare una grande quantità di dati, il progetto stesso assorbe la maggior parte dei costi. Per molti sistemi embedded le prestazioni richieste possono essere soddisfatte con una combinazione di hardware dedicati e una quantità limitata di software ottimizzati. Questo permette all'architettura di un sistema embedded di essere semplificata rispetto a quella di un computer generico che deve eseguire le stesse operazioni, usando, ad esempio, una CPU più economica che tutto sommato, si comporta discretamente anche per queste funzioni secondarie.

Come in ogni progetto, la riduzione dei costi diventa una priorità: si rivela dunque di fondamentale importanza la scelta dell'hardware che si andrà ad utilizzare, dal processore alla memoria, fino ad arrivare ai componenti che fanno da contorno al sistema, quali i sensori e gli attuatori, senza dimenticare il fattore affidabilità. A tale proposito infatti, è necessario specificare che i sistemi embedded sono utilizzanti negli ambiti più diversi, e soprattutto devono essere attivi e funzionali continuativamente 24/7.

1.1.1 Reliability

I sistemi embedded sono per loro natura nascosti all'utente, resi inaccessibili, e per questo collocati ed integrati all'interno degli apparati con cui lavorano a stretto contatto. L'inaccessibilità che si riscontra più frequentemente è quando il sistema non è fisicamente raggiungibile da nessun operatore (si pensi ad esempio ai moduli lanciati nello spazio). La criticità di questi sistemi è data da due fattori: oltre alla più evidente affidabilità hardware, è necessaria anche un certo grado di sicurezza a livello software; mentre il primo fattore può essere preventivato a tempo di design, consultando le specifiche del produttore, per i fault software è necessaria un'accurata fase di testing e debugging, in modo tale da ridurre la probabilità di crash; i device in oggetto, inoltre, devono essere capaci di auto-resettarsi autonomamente in caso di malfunzionamenti, perdita o corruzione dei dati. Questa funzionalità è molto spesso ottenuta con l'inserimento di un componente elettronico chiamato watchdog, attuato tramite temporizzatore hardware, in modo tale da rilevare loop infiniti o situazioni di deadlock ripristinando opportunamente il processore. Per migliorare l'attendibilità hardware, invece, si può agire in due direzioni: aumentare la sicurezza dei componenti migliorandone i processi produttivi ed assicurando un tempo di vita maggiore (come ad esempio eliminando le parti meccaniche, più soggette ad usura, etc...), oppure apportare un certo grado di ridondanza in componenti chiave del sistema (aggiungendo ad esempio una seconda fonte di alimentazione, circuiti di backup, etc...). Si vuole precisare che, in questo tipo di sistemi, la ridondanza non è vista come un errore di progettazione, piuttosto come una possibile soluzione a basso costo per garantire la sicurezza ed il funzionamento continuo dell'applicazione.

1.1.2 Le periferiche

I sistemi embedded dialogano col mondo esterno attraverso periferiche: le più comuni sono i componenti discreti, detti anche General Purpose Input/Output (GPIO) ed i componenti Analogici, per i quali si necessita di una conversione (ADC – Analog to Digital, DAC – Digital to Analog Conversion) per la corretta interpretazione da parte del microprocessore. Oltre a questi componenti c'è un'ampia gamma di

apparati che svolgono le proprie funzioni in autonomia, comunicando col sistema attraverso bus e protocolli di comunicazione ad hoc – quali interfacce seriali, parallele, fieldbuses, debugging protocol – e comunicazioni da e verso memorie ausiliari quali EEPROM o Multi Media Cards – SD, CompactFlash etc...

1.1.3 User interface

I sistemi embedded possono spaziare dall'avere nessuna interfaccia – per i device dedicati ad un solo task – fino a complesse GUI: tra i due estremi c'è un insieme di soluzioni intermedie per l'interazione uomo-device rappresentato da semplici bottoni, LED, joystick, display di caratteri o LCD, con semplici menù; gli strumenti più avanzati possono fare uso di schermi grafici e persino di touchscreen. Altri sistemi espongono interfacce remote attraverso connessioni seriali (RS-232, USB, I2C, etc.) o utilizzando la rete (la connessione Ethernet è l'esempio più comune) riducendo così costi per hardware addizionale (cavi, bottoni, led) estendendo le capacità del sistema tramite le cosiddette "rich application" disponibili mediante l'uso di un PC. Queste poi diventano indispensabili quando i dispositivi realizzano applicazioni ove si renda necessario trasmettere a distanza le immagini delle telecamere (come ad esempio la videosorveglianza).

1.2 Architetture Software

Esistono diversi tipi di architetture software di uso comune. Esaminiamo quelle maggiormente utilizzate

1.2.1 Simple Control Loop o Round Robin

È l'architettura in assoluto più semplice, il programma consiste in un ciclo infinito. Le operazioni in esso contenute vengono eseguite in modo sequenziale, eseguendo subroutine che hanno il compito di gestire l'hardware ed il controllo del sistema

1.2.2 Sistema a Interrupt

Alcuni sistemi embedded sono controllati in modo predominante a interrupt: un sistema ibrido a Control Loop affiancato da ulteriori task che l'apparato esegue in reazione a differenti tipi di eventi che occorrono. Un interrupt può essere generato, ad esempio, da un timer ad una certa frequenza, da una porta seriale alla ricezione di un byte, al cambiamento di fronte di un bit GPIO, etc... Questi tipi di sistemi sono usati se le funzioni che devono rispondere agli interrupt sono relativamente semplici e necessitano di un breve tempo di esecuzione: questo è un importante vincolo in quanto, un'errata gestione di un interrupt bloccherebbe il normale ciclo di esecuzione del programma, andando a minare la stabilità del sistema.

1.2.3 Cooperative multitasking

Un tipo di multitasking non preemptive è molto simile ad un Simple Control Loop ad eccezione del fatto che il ciclo viene nascosto attraverso le API del sistema. Il programmatore definisce una serie di task, oguno dei quali ha il proprio ambiente protetto e sicuro dove essere eseguito. Quando un task diventa inattivo lascia libero processore e risorse ad altri task, mettendosi in uno stato di pausa o attesa. I vantaggi e svantaggi di questo approccio sono quelli collegati al Simple Control Loop da cui eredita la struttura, anche se apparentemente mascherata; inoltre permette agilmente l'aggiunta di nuovo software, semplicemente scrivendo il codice per nuovi task ed aggiungendoli alla coda di quelli già esistenti.

1.2.4 Preemptive Multitasking o Multithreading

Fondamentale di questo approccio è l'esecuzione di più task (o thread), alternandoli, come se apparentemente ognuno stesse operando su un processore dedicato. Questo può essere realizzato attraverso uno strato software di basso livello in grado di eseguire lo "switching" dei task, dedicando a ciascuno un determinato tempo di processore (dato da un timer). Il sistema così programmato può essere generalmente considerato come in possesso di un "operating system kernel",

mostrando un certo grado di concorrenzialità tra le applicazioni, pertanto il codice di ogni task deve essere opportunamente progettato e testato, soprattutto per quanto riguarda l'accesso a dati comuni: si rende necessario dunque l'utilizzo di specifiche strategie di sincronizzazione come semafori, medium di coordinazione, etc... A causa della sua complessità, è pratica comune l'uso di sistemi operativi real time, come già accennato nei paragrafi precedenti, permettendo al programmatore di concentrarsi sulle funzionalità dei device, piuttosto che la coordinazione e lo switching fra task. È altresì vero che, per sistemi molto piccoli questo non è possibile, in quanto le ridotte specifiche di memoria, potenza di elaborazione o vincoli di consumo di energia (per sistemi mobili o a basso consumo) introdurrebbero un overhead insostenibile

1.3 Linguaggi di programmazione

Il software redatto per sistemi embedded, chiamato firmware, è scritto ed adattato per hardware special purpose a cui è destinato, ed è il principale supervisore dell'elaborazione dei dati provenienti dal mondo fisico, controllandone le variabili ed effettuando modifiche attraverso gli attuatori a cui ha accesso; è inoltre responsabile delle comunicazioni fra i sistemi attraverso l'uso di bus e protocolli standard open o proprietari. Tuttavia, quando si parla di firmware, l'associazione più comune è quella di software non modificabile: pur non essendo totalmente errato, è più corretto associare al termine la semantica di piccolo programma e strutture dati associate per il controllo interno di un device elettronico. Ciò nonostante il confine tra firmware e software non è definibile in termine di dimensione o complessità: il termine fu originariamente coniato per mettere in contrasto il software scritto a basso livello appositamente per sistemi embedded con quello scritto ad alto livello per macchine general purpose.

L'utente finale vede questo software come "built in", a cui non ha né accesso né possibilità di modifica. I firmware di più basso livello risiedono infatti in strutture come PLA o ROM (e successive) la cui possibilità di aggiornamento non è prevista; altri tipi di strutture dedicate al mantenimento del firmware implicano l'uso di programma-

tori ad hoc, software compilati in un file immagine binario – spesso rilasciato dal costruttore.

Come per tutti i software in ambito general purpose, anche quelli utilizzati per i sistemi embedded sono sviluppati attraverso l'uso di compilatori e linker per poi essere perfezionati mediante l'aiuto di appositi debugger. In questa fase bisogna dunque fare perno sul passo successivo, cioè il livello fisico: il software, sviluppato su apposita piattaforma (come ad esempio un pc), deve poi essere immesso in maniera semi-definitiva sul dispositivo a microprocessore che ne eseguirà fisicamente l'algoritmo presente.

Il produttore, a tal fine, dovrà fornire appositi strumenti per il trasferimento del codice tra la piattaforma di sviluppo e quella di esecuzione/produzione, in maniera trasparente per il programmatore. Sarà inoltre cura dello stesso produttore mettere a disposizione un valido strumento di debugger con cui il programmatore potrà controllare lo stato del sistema in determinati istanti per verificare il corretto funzionamento dell'algoritmo. Ulteriori strumenti per la risoluzione di problemi durante lo sviluppo sono gli emulatori che consentono un ulteriore fase di debug ancora prima di caricare il software sul sistema.

Una possibile soluzione alternativa, è quella di aggiungere un ulteriore strato (middleware), interponendo al programma in oggetto un sistema operativo real-time, il quale funge da intermediario tra il processore ed il software applicativo. Tre dei più famosi sistemi operativi real-time per sistemi embedded sono QNX[26], TinyOS[31] e FreeRTOS[14]. Questo strato software è progettato per essere compatto, efficiente ed affidabile, innalzando il livello di astrazione. Principale compito è fornire un insieme di API standard – ed il più possibile condivise a fronte di hardware eterogenei – implementando alcune delle funzioni più comuni – tra cui possiamo annoverare multitasking, multithreading, condivisione delle risorse ... A differenza dei sistemi operativi desktop, gli RTOS sono staticamente collegati in una singola immagine eseguibile: un apparato embedded dunque non può caricare ed eseguire diversi programmi allo stesso tempo, bensì sempre e solo una immagine alla volta

Nel mondo dei sistemi embedded è difficile elencare quali siano tutti i linguaggi di programmazione utilizzati, in quanto ogni produttore di hardware mette a disposizione strumenti propri per la realizzazione

del software di controllo. Possiamo tuttavia categorizzare le soluzioni più utilizzate in ambiti professionali, descrivendo in maniera generale l'approccio seguito anche in relazione a quanto già descritto finora.

1.3.1 Linguaggio C

Il linguaggio C, incluse le sue declinazioni più specifiche per questo ambiente, è sicuramente il linguaggio più utilizzato in ambito embedded. I compilatori C, forniti dai produttori di hardware, sono costruiti sullo standard ANSI-C ed in seguito personalizzati per supportare concetti essenziali per la progettazione dei sistemi quali: general purpose I/O, gestione di interrupt, gerarchie comportamentali e strutturali, concorrenza, comunicazioni, protocolli, sincronizzazione, transizioni di stato, gestione delle eccezioni, timing, etc. Spesso il progettista è assistito da ulteriori strumenti quali debugger con cui può osservare lo stato interno del sistema di elaborazione, verificandone la correttezza semantica del software prodotto.

1.3.2 Diagrammi Statecharts

Nato in ambito professionale, il diagramma Statecharts mostra gli stati che il sistema può assumere, ciascuno in risposta ad eventi, sia esterni che interni. Il concetto di stato è dunque parte fondante di questo tipo di programmazione, spesso messo in relazione al ciclo di vita stesso del sistema: lo spazio degli stati del diagramma coincide con lo spazio degli stati che il sistema ammettere. Inoltre lo StateChart Diagram è un diagramma previsto dallo standard UML, a riconferma delle potenzialità, praticità e semplicità di utilizzo, senza alterare la potenza espressiva del linguaggio. Gli elementi rappresentati da uno StateChart Diagram sono lo stato - distinguendo tra iniziale, intermedio e stato finale - l'evento, l'azione e la guardia. Lo stato può descrive sia una qualità dell'entità o classe che si sta rappresentando, chè un determinato momento all'interno del ciclo di vita del sistema; l' "evento" è la descrizione dell'azione che comporta il cambiamento di stato mentre "azione" è l'evento che ne consegue; la "guardia" è un'eventuale ulteriore condizione che si deve verificare perché si possa compiere l'azione.

1.3.3 VHDL e Verilog

VHDL è l'acronimo di VHSIC (Very High Speed Integrated Circuits) Hardware Description Language ed è, insieme al Verilog, il linguaggio più usato per la progettazione di sistemi elettronici digitali. È lo strumento fondamentale per la progettazione dei moderni circuiti integrati digitali e le sue applicazioni spaziano dai microprocessori (DSP, acceleratori grafici), alle comunicazioni (Cellulari, TV satellitare) al settore automobilistico (navigatori, controllo stabilità) e molte altre. Nonostante il VHDL presenti costrutti di un normale linguaggio di programmazione (come ad esempio if...then...else) è essenzialmente utilizzato come un modello per la descrizione della struttura dei componenti hardware del sistema. Uno dei punti che lo differenzia da un applicativo software è la concorrenzialità intrinseca: con questo termine si indica il fatto che diverse parti di un codice scritto in VHDL devono essere immaginate come funzionanti contemporaneamente, in quanto effettivamente implementate simultaneamente.

Il VHDL permette di modellare facilmente l'interazione tra i vari blocchi funzionali che compongono un sistema. Queste interazioni sono essenzialmente lo scambio di segnali di controllo e di dati tra i vari oggetti che costituiscono il sistema. In un sistema hardware infatti ogni oggetto da modellare, che sia esso una semplice porta logica o un complesso microprocessore, reagisce istantaneamente agli stimoli ai suoi ingressi producendo dei cambiamenti sulle sue uscite. Ogni blocco funzionale, a sua volta, è descritto nella relazione ingresso-uscita, usando i classici costrutti del linguaggi di programmazione (if, for, while).

Il linguaggio (o anche Verilog HDL) supporta la progettazione, la verifica, e l'implementazione di circuiti digitali attraverso una sintassi simile al C. Allo stato attuale può considerarsi l'unico linguaggio, assieme al VHDL utilizzato nel mondo della progettazione e simulazione digitale industriale. La somiglianza del linguaggio con lo standard di riferimento ANSI-C è evidente: dalla presenza di un preprocessore, alla condivisione delle più importanti parole chiave. Il linguaggio differisce da quelli convenzionali nell'esecuzione delle istruzioni, dato che essendo un linguaggio che descrive processi paralleli e concorrenti, l'esecuzione non è strettamente lineare. Un progetto Verilog consiste di

una gerarchia di moduli. Ciascuno è definito da un insieme di ingressi e uscite e porte bidirezionali.

La sintesi di processi paralleli e sequenziali ne definiscono il comportamento costituendo le relazioni attraverso porte, register e wire. Le istruzioni sequenziali sono poste all'interno di un blocco begin/end in ordine sequenziale all'interno del blocco. Tutti le istruzioni concorrenti e tutti i blocchi begin/end sono eseguiti in parallelo. Un modulo contiene una o più istanze di un altro modulo per definire sotto comportamenti. La lista di connessioni (netlist) può essere per esempio in una forma che descrive un circuito integrato di tipo gate array, molto più raramente uno standard cells, ovvero degli ASIC. Più comunemente l'uscita è un bitstream utilizzata per un dispositivo programmable logic device (ad esempio, una FPGA).

1.3.4 PLC

Negli ultimi 50 anni i progressi nell'ambito dell'automazione industriale hanno fatto si che emergessero diversi stili di progettazione e programmazione dei sistemi imponendo nuovi standard sul mercato. Anche se gli impianti industriali non fanno parte propriamente della categoria dei sistemi embedded, è doveroso citarli in quanto parte fondamentale dello sviluppo e della penetrazione informatica in ambito altamente professionale. La programmazione del PLC è effettuata normalmente con un PC sul quale un software specializzato permette di creare programmi da caricare nella memoria interna della CPU del PLC. Questi software di programmazione possono leggere il programma direttamente dalla memoria della CPU, e visualizzare l'output sul PC mediante un'interfaccia grafica elaborata. Nel 1993 l'entità preposta ha standardizzato 5 linguaggi di programmazione, di cui 3 grafici e 2 testuali:

- *Ladder diagram (LD)* anche detto linguaggio a contatti, era in assoluto uno dei più utilizzati in quanto era quanto di più vicino alla rappresentazione cartacea presente fino alla sua introduzione.

- *Sequential function chart (SFC)* rappresenta il diagramma sequenziale funzionale del sistema molto simile alla rappresentazione di una macchina a stati finiti.

- *Function Block Diagram (FBD o FUP)* è un particolare linguaggio grafico analogo ai diagrammi circuitali.

- *Instruction List (IL)* si compone di istruzioni di basso livello consentendo di avere il pieno controllo del PLC. Proprio per questa sua caratteristica è un linguaggio complesso da utilizzare, paragonabile all'assembler di un normale PC.

- *Structured Text (ST)* è una versione di più alto livello rispetto l'IL, molto simile al Pascal per quanto riguarda la sintassi, incorporando strutture dati e metodi per un più veloce apprendimento e guadagno in termini di tempo nello sviluppo vero e proprio.

1.3.5 Altri linguaggi

Su tutti linguaggi desktop che si sono sviluppati nel corso della storia dell'informatica, molti di questi sono stati portati, attraverso apposite limitazioni ed opportune ottimizzazioni, su piattaforme embedded. Tra i maggiori, è opportuno segnalare il linguaggio Java, che si è di volta in volta adattato alle caratteristiche delle piattaforme, grazie alla sua particolare struttura in grado di produrre un bytecode interpretato da un'apposita macchina virtuale. Scrivendo cioè un'apposita macchina virtuale per una specifica piattaforma si è in grado di eseguire un discreto numero di applicativi con tutti i pregi e difetti che la sintassi/semantica di java offre. In particolare, sono 2 le varianti del linguaggio che hanno portato Java nei sistemi embedded:

1. *Java ME* è una versione specializzata ed ottimizzata della JVM per dispositivi a ridotte capacità, tra cui possiamo annoverare anche i sistemi embedded. I requisiti in termini di capacità computazionale e di memoria sono ridotti rispetto alla Standard Edition permettendo una più ampia diffusione tra i device meno performanti. Inoltre sono state estese le piattaforme hardware

utilizzabili, aumentando le implementazioni della JVM per dispositivi con architettura non prettamente x86 (come ad esempio ARM, PowerPC o architetture MIPS).

2. *Java Real Time* è in realtà una combinazione di tecnologie che permette ai programmatori di scrivere programmi che rispettino determinati vincoli per applicazioni real-time. Oltre ai vantaggi tipici della Java Virtual Machine – come la sofisticata gestione della memoria, costrutti per gestire la concorrenza, forte tipizzazione dei dati – sono stati inoltre implementati ulteriori meccanismi per una migliore modellazione dei thread, basandosi su uno schema di priorità. Le specifiche derivate (Real-Time Specification for Java – RTSJ) risultano dunque come un set di interfacce comportamentali che permettono una programmazione real-time: a fronte di questa specifica (RTSJ 1.0) un'insieme di aziende nell'ambito IT del calibro di IBM, Sun Microsystems ed altre, hanno sviluppato la propria JVM Real Time per sistemi embedded.

1.4 Cyber-physical system

Un sistema di tipo Cyber-Physical (CPS) è un sistema in cui si implementano un combinazione e collaborazione strettamente integrata tra componente elaborativa ed elementi fisici; derivano direttamente dai sistemi embedded, con cui condividono buona parte dell'architettura hardware e software: mentre questi ultimi sono fortemente votati ad una componente elaborativa, i CPS possiedono un'accezione maggiorata alla parte di interfaccia microprocessore / componenti esterni.

I Cyber-Physical Systems devono possedere le seguenti caratteristiche:

- *Availability*: la probabilità che un sistema funzioni in un determinato tempo t' deve essere costante e pari a quella al tempo t=0. Questo deve comportare un'alta affidabilità sia hardware chè software.

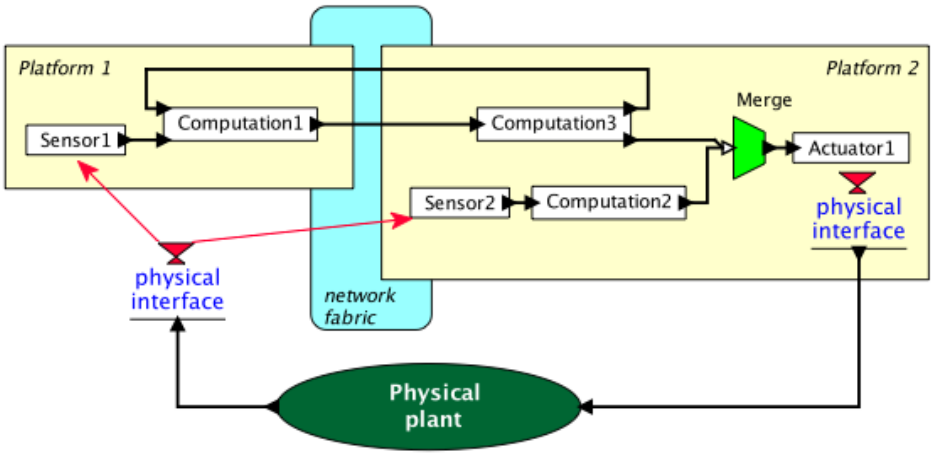

Figura 1.1: Schema concettuale di un Cyber-Physical System (CPS)

- *Maintainability*: il corretto funzionamento anche dopo l'occorrenza di una situazione di errore. Ciò può essere riferito sia alla parte hardware (il default di uno o più componenti) che alla parte software del sistema (ad esempio il crash del sistema).
- *Safety*: il sistema non deve causare errori o malfunzionamenti nel sistema controllato
- *Security*: le comunicazioni ed i dati utilizzati per poter rimanere confidenziali ed autentiche.

Dal punto di vista prettamente ingegneristico il sistema deve avere i seguenti requisiti funzionali:

- *Sistemi Real time o sistemi Reattivi*
- *Dimensioni contenute in termini di spazio e peso*
- *Basso costo*
- *Capacità di limitare la potenza assorbita dal sistema*
- *Capacità di evitare, se possibile, il raffreddamento*: parti mobili sono più soggette ad usura e quindi a malfunzionamenti

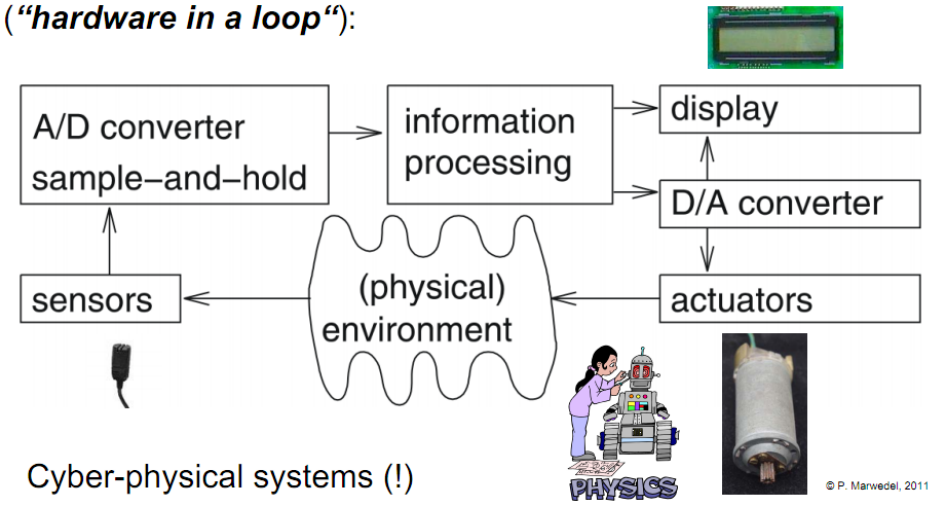

Figura 1.2: Tipico esempio di Cyber-Physical System (CPS)

I vincoli temporali sui sistemi fisici sono spesso molto stringenti, pertanto i CPS devono saper agire e reagire agli stimoli che provengono dagli oggetti controllati nei tempi e modi stabiliti dall'ambiente e dalle specifiche di progetto. Azioni corrette che però arrivano troppo tardi sono, e devono essere, considerate come sbagliate. Inoltre un sistema che possa garantire sicurezza in condizioni cruciali non deve dipendere da variabili statiche: a tal fine il tempo di risposta deve essere provato sperimentalmente ed essere minore del tempo massimo concesso per le azioni.

Edward A. Lee[20] conduce un'ottima analisi e riflessione sui sistemi Cyber-Physical, focalizzando il fulcro del suo discorso intorno al modello concettuale dei linguaggi di programmazione attualmente utilizzati che si interfacciano con il livello hardware sottostante. I sistemi fisici introducono requisiti di affidabilità e sicurezza qualitativamente molto differenti da quelli generalmente trattati da una macchina general purpose. Inoltre, i componenti fisici sono – anche intuitivamente – molto diversi da da componenti software object-oriented. Vengono così a crollare le astrazioni standard di base, come le chiamate a metodo oppure i thread, che ora non possono più funzionare nell'integrazione col mondo esterno. I sistemi CPS porteranno, già nel breve

termine, una potenziale rivoluzione in ogni settore, dalla medicina al settore automobilistico, al controllo di processo, risparmio energetico, sistemi di difesa, edifici intelligenti... Tuttavia, la rivoluzione starà nel fatto che tutti i sistemi saranno collegati alla rete, in un futuro scenario di pervasive computing estremo. I problemi legati all'affidabilità e sicurezza saranno dunque amplificati, come in un processo sequenziale concatenato, ove il fallimento di un sistema può portare il default di altri ad esso collegato. Come già visto, anche i sistemi operativi stanno muovendo i primi passi in questa direzione, anche se tuttavia gli RTOS rimangono ancora tecnologie best-effort. Applicare politiche di timing sull'esecuzione di programmi, infatti, inducono il programmatore e a fare un passo indietro, ad esempio tramite system calls o system interrupt. Analoghi problemi li possiamo trovare per quanto riguarda la concorrenza, fondamentale quando si parla di CPS essendo loro stessi sistemi concorrenti. Già oggi i sistemi devono reagire, in tempo reale, a stimoli provenienti da sensori multipli attraverso un set di attuatori in modo concorrente l'uno all'altro. Timing e concorrenza non sono certamente astrazioni di primo livello ben rappresentati nei linguaggi fin'ora sviluppati per sistemi embedded o CPS; gli RTOS in questa ottica appaiono solo come una mera patch, temporanea. Inoltre il concetto di interrupt è sottovalutato nei linguaggi di programmazione, considerati come astrazione di basso livello e lasciati gestire al sistema operativo, spesso in modo errato. Quello che si vuole ottenere dalla prossima generazione di sistemi cyber-physical è un modello elaborativo concorrente, deterministico, predicibile, eliminando il concetto di thread che rende i programmi decisamente imprevedibili.

Ovviamente questi problemi non sono del tutto nuovi, e ricerche in questo ambito hanno già contribuito a risolvere parzialmente i problemi. Strumenti avanzati di verifica formale, tecniche di emulazione e simulazione, metodi di certificazione e l'ingegnerizzazione dei processi attraverso design pattern e componenti software standardizzati hanno aiutato notevolmente. Oltretutto rimane senza soluzione di alto livello la problematica riguardante il timing come astrazione base di linguaggio: una soluzione completa per sistemi CPS non può esserne sprovvista. Raggiungere la precisione temporale non è compito arduo se potessimo accantonare le performance appesantite dall'overhead introdotto per lo scopo; la sfida ingegneristica sarà quella di avere sia

l'una che l'altra soluzione, senza però dimenticare gli ultimi 40 anni di evoluzione tecnologica, a partire da cache e pipeline fino a compilatori e sistemi operativi.

Per realizzare completamente il potenziale dei CPS dunque bisogna ripensare alle astrazioni centrali dei sistemi. Miglioramenti incrementali porteranno alla semplificazione nella realizzazione dei sistemi altamente integrati nel mondo reale, senza dimenticare che la vera sfida consisterà nell'affrontare la progettazione non più come l'unione di due sistemi separati – quello elaborativo e quello fisico – ma come un unico sistema interagente.

1.4.1 Un nuovo paradigma

In questo nuovo scenario, il software gioca un ruolo di fondamentale importanza, imponendosi come gestore e coordinatore di dispositivi hardware e del sistema nel suo insieme, talvolta operando in situazioni critiche, senza margine di errore. Per tale motivo lo sviluppatore è chiamato alla programmazione di piattaforme il cui compito esula dal perseguimento di un singolo obbiettivo, ma di molteplici finalità ove l'interazione e lo scambio di informazioni con altri dispositivi è un requisito fondamentale. Compito dell'ingegneria del software è quello di mettere a disposizione validi strumenti che permettano un utilizzo semplice, ma allo stesso tempo potente, di tali piattaforme, sollevando lo sviluppatore dai compiti di più basso livello innalzando le astrazione di prim'ordine. Ne risulterà così un tempo di sviluppo inferiore, maggiore stabilità del codice ottenuto e migliori strumenti per la rilevazioni di eventuali errori concettuali. In questo quadro, vengono considerati quindi nuovi paradigmi di programmazione, tra i quali quello ad agenti da me sperimentato in questa tesi, in quanto perfettamente calzante alla metafora che ne viene rappresentata: ogni CPS infatti può essere interpretato come l'ambiente in cui è immerso l'agente (la nostra entità attiva), il quale dispone di opportune strumenti di percezione ed attuazione, il cui compito è il controllo globale o locale del sistema.

Capitolo 2

Agenti BDI

2.1 Caratteristiche degli Agenti

Negli ultimi anni, nel campo dell' Information Tecnology, si è assistito all'affermarsi di un nuovo paradigma computazionale la cui astrazione di base è l'agente, i cui concetti tipici hanno trovato riscontro nell'ambito di diverse discipline nella società dell'informazione, tra cui l'ingegneria del software, intelligenza artificiale, sistemi distribuiti e concorrenti, reti di calcolatori, sistemi mobili.

Secondo la nota definizione di N. R. Jennings, M. Wooldridge e K. Sycara:

Un agente è un sistema computazionale situato in un ambiente e capace di azioni flessibili e autonome per raggiungere i suoi obiettivi di progetto. [32]

Analizzando i singoli concetti espressi nella definizione, possiamo riconoscere diversi punti chiave: il contesto ambientale è una parte fondamentale della vita di un agente a cui deve rispondere e con cui interagire, ricevendo input sensoriali ed allo stesso tempo compiendo azioni che ne modificano lo stato in diversi modi; per "azioni flessibili" si intende la tipologia di comportamento che è mostrata sia nei confronti dell'ambiente che degli obiettivi (o task). La prima caratteristica permette di poter aumentare il livello di complessità del contesto applicativo, mentre quella rispetto agli obiettivi indica la possibilità di poterli modificare, aggiungerne di nuovi o rimuoverli, rimanendo nell'ottica di una concorrenza fra task contemporanei.

Questi tipi di flessibilità possono essere raggiunti attraverso i seguenti comportamenti:

- *Autonomia*: l'attitudine naturale di un agente di prendere decisioni, non soltanto riguardo alla sequenza di azioni da svolgere, ma anche a quali obbiettivi perseguire, secondo una logica di massimizzazione delle performance (siano essi in termini di tempo impiegato o di risorse utilizzate) mirando al raggiungimento dello stato desiderato.

- *Reattività*: indica la capacità di osservare l'ambiente per riconoscere eventi di interesse specifico, ed avere la possibilità (ma allo stesso tempo opportunità) di poter reagire favorevolmente a tali accadimenti ponendo in atto opportune azioni.

- *Proattività (o goal oriented)*: indica la capacità di porre in atto un comportamento che porti al raggiungimento di un dato obiettivo. Fondamentale a questo scopo è la capacità di sintetizzare un piano di azioni e di eseguirlo in funzione dell'obiettivo e dello stato. Rientra nel comportamento proattivo anche la capacità di modificare, se possibile, i propri piani, adattandoli alle mutevoli condizioni di un ambiente dinamico.

- *Capacità sociale*: è la proprietà intrinseca di un agente di poter comunicare con altri suoi simili in un contesto di pervasive computing distribuito. Non è raro infatti voler modellare un sistema come un insieme di agenti che vivono e interagiscono nello stesso ambiente, con l'abilità di poter scambiare informazioni l'un l'altro. Il livello di interazione, tuttavia, differisce da caso a caso: a cominciare da una relativa autosufficienza fino a forme complesse di competizione e/o cooperazione in vista di obiettivi comuni o divergenti.

Altre caratteristiche che stanno emergendo dallo studio di questi sistemi, e che la tecnologia sta rendendo possibili sono:

- *Mobilità*: la possibilità di un agente di muoversi attraverso la rete, spostandosi tra i nodi del sistema distribuito.

- *Razionalità*: indica la capacità di un agente di perseguire una molteplicità di obiettivi, senza ostacolare se stesso in questo compito.

- *Apprendimento / Adattatività*: possibilità da parte di un agente di imparare nuove nozioni derivanti dalla sua attività precedente o dall'osservazione dell'ambiente. Inoltre, un agente dotato di questa proprietà può modificare il proprio comportamento in base al reperimento di nuove informazioni o per adattarsi ad eventuali cambiamenti nell'ambiente circostante.

La caratteristica distintiva di tali sistemi è sicuramente un superiore livello di autonomia, mantenendo il controllo sul proprio stato interno e sulle proprie strutture dati relative. La genericità della definizione data fa comunque sì che molti sistemi (da puri programmi software fino ai controllori di sistemi fisici più avanzati) possano essere classificati come agenti: in particolare un sistema di controllo automatico è in grado di garantire il comportamento desiderato anche in presenza di disturbi, ossia sotto condizioni di variabilità ambientale: vediamo così la stretta relazione tra i due settori, in quanto la desiderabilità di uno stato è anche il task principale di un agente. Il contesto applicativo può essere un ambiente fisico reale, come nel caso di un robot, oppure un ambiente puramente software (come una simulazione o lo stesso Internet); in quest'ultimo caso si parla di agenti software. Le azioni e le percezioni ovviamente differiscono da situazione a situazione: è facile pensare come in un ambiente fisico l'agente avrà diretto accesso ad un ambiente reale, mentre in un contesto applicativo software questo si interfaccerà ad altri componenti della stessa natura.

Strettamente legato all'astrazione di agente è il concetto di "azione" in un dato ambiente (environment) indipendentemente dal fatto che esso sia fisico o software; questa consiste infatti nel'atto di poter modificare il contesto che avvolge l'agente in modo diretto (mediante l'uso di attuatori) o indiretto (attraverso lo scambio di messaggi in un sistema multi agente, modificando la conoscenza di altri suoi simili). L'enfasi sulla proprietà di integrazione in un ambiente ha dato luogo allo specifica caratteristica di essere "situato": con questo termine si vuole sottolineare che il processo di deliberazione dell'agente

è fortemente influenzato non solo dal contesto dell'ambiente in cui è immerso (e da cui riceve percezioni ed in cui agisce modificandolo) ma anche dalle informazioni ottenute da altre entità con cui può interagire. Inoltre gli agenti si definiscono intelligenti quando oltre a presentare comportamenti flessibili, hanno la capacità di adattarsi ai cambiamenti dell'ambiente, perseguendo obiettivi complessi in maniera autonoma ed in modo non precedentemente codificato a causa del processo di apprendimento di cui esso è dotato.

2.2 Multi Agents Systems

L'infrastruttura moderna dei sistemi non può prescindere da una una natura distribuita sia delle informazioni che delle risorse computazionali: le reti, come collegamento tra punti di elaborazioni, sono penetrate in tutti gli ambiti applicativi, dal cellulare all'automobile, ai sistemi di controllo, automazione etc... In questo scenario dunque non è possibile definire una nuova architettura computazionale come gli agenti che non comprenda l'interazione – oltre all'ambiente come fin qui descritto – con entità omogenee: si parla cioè di Sistemi Multi Agenti (i cosiddetti MAS). La ricerca sui MAS è incentrata sullo studio del comportamento di un insieme di agenti autonomi, eterogenei, che mirano a risolvere un dato problema e/o che competono per massimizzare il vantaggio individuale. Tale comportamento è determinato dalla capacità degli stessi di agire, interagire e comunicare: tale abilità non può manifestarsi solo attraverso l'uso mediatore dell'ambiente in cui sono immersi, ma deve anche poter essere supportato attraverso un'opportuna astrazione di comunicazione diretta inter-agente. L'apparato "sociale", cioè la possibilità di interazione fra una moltitudine di entità, è fonte di un nuove forme di autonomia: ora gli agenti possono avere la capacità di supportare scambi di informazioni dinamici con partner non definiti staticamente a tempo di progetto, ma durante l'esecuzione, in maniera impredicibile. Questo rappresenta uno schema di interazione di più alto livello che supera il tradizionale pattern client-server. Tipicamente i sistemi di controllo per domini fisici critici, come i sistemi embedded, non possono essere terminati, o sospesi nella loro esecuzione; inoltre solitamente non possono neanche essere

rimossi dall'ambiente in cui sono integrati; ciò nonostante, questi sistemi hanno bisogno di manutenzione ed aggiornamenti, sia software che hardware come qualsiasi altro componente, il tutto mentre l'ambiente in cui vivono cambia costantemente. Questi sistemi (denominati "aperti") devono presentare complessi dispositivi che possiedono alti livelli di riorganizzazione ed interazione – a causa delle caratteristiche dalla struttura del sistema stesso appena elencate – permettendo loro di svolgere determinate mansioni e collaborare in modo efficiente.

Inoltre, le problematiche riguardanti la scarsità di risorse, come ad esempio l'alimentazione, o più semplicemente l'irraggiungibilità nella comunicazione possono far si che i nodi (o parti di rete) rimangano isolati in modo impredicibile: si può facilmente intuire che il requisito di dinamicità rispetto all'organizzazione e alla ristrutturazione degli schemi di comunicazione è fondamentale. Queste problematiche sono ulteriormente aggravate nell'ambito di reti mobili, in cui l'interazione deve essere fruibile e controllabile a prescindere dalla mancanza di componenti essenziali e di dinamiche di connettività. Considerazioni simili si possono fare per computazioni distribuite aperte e basate su Internet.

I sistemi naturali sono spesso fonte di ispirazione per i progetti MAS. Molte forme di insetti sociali o animali come colonie di insetti, termiti, formiche, mostrano complesse forme di coordinazione, senza una diretta comunicazione tra loro e senza una particolare intelligenza, se non quella minima di un comportamento razionale, limitata a certo numero di azioni. Il sistema evolve naturalmente, in stato di equilibrio auto organizzante. La forma di comunicazione indiretta – mediata dall'ambiente – tra le varie entità del sistema prende il nome di "stigmergia". I sistemi naturali presentano molte analogie con i MAS in quanto sono composti da molte entità autonome in grado di coordinarsi attraverso l'ambiente per raggiungere uno stato di equilibrio. Anche gli agenti come gli insetti possono utilizzare l'ambiente per il raggiungimento del loro scopo sociale.

2.3 Practical Reasoning Agents

Tra le varie architetture proposte in letteratura e ricerca, ci focalizziamo su quella che più si avvicina al pensiero filosofico del Practical Reasoning, proposta dal filosofo Michael Bratman nel 1987 [9]: l'agente viene indotto attraverso un processo decisionale ricorrente, partendo dalle credenze iniziali sull'ambiente e del suo stato interno, attraverso la scelta dapprima degli obbiettivi da perseguire, ed in un secondo tempo delle azioni per raggiungere lo stato desiderato. Concetto fondamentale del practical reasoning è quindi la scelta dei propri obiettivi: questo procedimento presuppone la conoscenza sia delle opzioni a disposizione, sia dei propri desideri. Le decisioni prese hanno differenti gradi di complessità e d'importanza: si possono fare scelte fondamentali, come decidere di sfruttare una nuova opportunità totalmente nuova e inaspettata presentatasi a causa di un'evoluzione non prevista a priori, o decidere semplicemente il modo migliore di perseguire un determinato stato finale. Una volta scelto un obiettivo non rimane che determinare la miglior sequenza di azioni a disposizione dell'agente per conseguirlo finché non lo si raggiunge; l'obiettivo viene abbandonato solo se la possibilità di ottenerlo è nulla. Nella teoria del practical reasoning questo passaggio è descritto come la trasformazione delle opzioni scelte in intenzioni, le quali possiedono la caratteristica di persistenza finché le nostre credenze (Belief) fanno si che siano ancora ragionevoli. Nel processo descritto si possono identificare sia il comportamento proattivo (persistenza delle intenzioni) sia quello reattivo (abbandono delle intenzioni perché non raggiungibili). Il comportamento reattivo ci permette di verificare le nostre scelte alla luce delle ultime informazioni, invece di concentrarci ciecamente su ciò che abbiamo precedentemente deciso. La caratteristica fondamentale delle architetture Belief-Desire-Intention (BDI) è quella di utilizzare strutture dati che corrispondono ai beliefs(credenze), desires(desideri) ed intentions(intenzioni) cui si riferisce la teoria del practical reasoning; i processi decisionali sono realizzati da funzioni di selezione che agiscono su queste strutture. Per meglio focalizzare:

- i *Beliefs* corrispondono alle informazioni, o meglio le sue credenze, che l'agente ha sull'ambiente in cui esso è situato: queste possono essere incomplete, non corrette, o non aggiornate;

- i *Desires* rappresentano lo stato del sistema che l'agente vorrebbe raggiungere, desiderato;
- le *Intentions* rappresentano i desideri che l'agente si è impegnato a realizzare, per i quali sono già in atto processi decisionali/deliberativi sulle azioni che sta compiendo.

Affrontiamo ora in dettaglio il processo deliberativo/decisionale di un agente: possiamo partire da un'analogia con il ragionamento pratico umano applicandolo poi al contesto applicativo in ambito agenti BDI. Come abbiamo già discusso precedentemente esistono due fasi principali:

1. *Deliberation*, la decisione di quale tipo di attività o task portare a compimento
2. *Means-end Reasoning* l'atto di scelta delle azioni da utilizzare per il raggiungimento dello scopo al punto appena deliberato.

É tuttavia necessaria una puntualizzazione sul termine Intentions, in quanto fonte di ambiguità nell'interpretazione: esso infatti può essere interpretato sia nel senso di uno state mentale (ho intenzione di fare qualcosa nel futuro ad uno scopo), oppure nel senso della caratterizzazione di un'azione che si sta eseguendo. In questo contesto la prima interpretazione è quella che useremo, con l'accezione di intenzioni future (future-directed intention), in quanto col passare del tempo l'adozione di certe intenzioni in determinati istanti deve forzatamente influenzare il ragionamento pratico da quel momento in poi: le intenzioni dunque, sono strettamente legate ai belief correnti e futuri su cui l'agente fonda le proprie ipotesi.

Il risultato finale prima della deliberation ed in seguito del means-end reasoning è un piano (o un insieme di traguardi intermedi) da effettuare (o raggiungere) per il task selezionato. A sequito di ciò l'agente procederà alla fase esecutiva, in cui tenterà di eseguirlo: l'uso del condizionale in questa affermazione sta a significare che in un mondo reale, l'agente sarà inevitabilmente soggetto a vincoli (siano essi temporali o di risorse limitate/condivise), e a cambiamenti nelle proprie credenze, a seguito sia dell'evoluzione del sistema a causa diretta delle proprie azioni, che indirettamente data la naturale evoluzione

temporale dell'ambiente e/o data la presenza di altri agenti coesistenti. L'agente sarà così spinto in una fase continua di aggiornamento del proprio stato, e del processo deliberativo e decisionale per determinare in ciascun istante temporale quale siano:

1. possibili goal raggiungibili;
2. il goal eletto ad essere conseguito;
3. il piano più adatto allo scopo;
4. la prossima azione da eseguire.

Vediamo come le fasi di un agente possono essere formalizzate, partendo dalla fase di aggiornamento dei beliefs: un'apposita funzione, raccogliendo le percezioni dall'ambiente, andrà a modificare le credenze interessate. Per formalizzare quanto appena descritto:

$$update : \gamma(Beliefs) \times Perception \rightarrow \gamma(Beliefs)$$

Fase deliberativa: in un primo tempo vengono generate opzioni – cioè una lista di desires – a partire dai belief correnti e dalle intenzioni proprie dell'agente nel determinato istante.

$$options : \gamma(Beliefs) \times \gamma(Intentions) \rightarrow \gamma(Desires)$$

In seguito, al fine di scegliere una di queste opzioni che competono per essere eseguite, l'agente utilizza un'apposita funzione di filtro. Intuitivamente questa funzione semplicemente dovrà scegliere la miglior opzione da perseguire nel resto della sua vita; possiamo dunque rappresentare questa funzione filtro come segue:

$$filter : \gamma(Belies) \times \gamma(Desires) \times \gamma(Intentions) \rightarrow \gamma(Intentions)$$

Fase decisionale: è il processo che porta alla selezione di uno specifico piano da attuare per arrivare allo scopo ottenuto dalla fase precedente, utilizzando le azioni disponibili. Se con Actions definiamo tutte le possibili azioni di cui un agente è dotato la formalizzazione del means-end reasoning può essere formalizzato come segue:

$$plan : \gamma(Belies) \times \gamma(Intentions) \times \gamma(Actions) \rightarrow Plan$$

dove Plan è un sottoinsieme delle azioni.

2.4 Cicli di ragionamento / esecuzione di un agente

Con riferimento ai linguaggi di programmazioni esistenti, ne prenderò in considerazione uno dei riferimenti più importanti che in questi anni abbia dato prova della bontà dell'approccio che implementa. Allo scopo di questa tesi quindi, vediamo come sono stati effettivamente implementati i cicli di ragionamento pratico da AgentSpeak e dalla sua implementazione Jason.

2.4.1 AgentSpeak

Il linguaggio AgentSpeak(L), sviluppato da A. S. Rao nel 1996 [27], consente di definire la sintassi e semantica di agenti PRS, consentendo inoltre di essere eseguiti mediante interprete; le operazioni di base degli agenti AgentSpeak(L) si basano su belief, desideri e intenzioni, avendo cura, inoltre, di mantenere memorizzati i possibili piani disponibili – la cosiddetta libreria dei piani (plan library). Oltre ai tre concetti base BDI, AgentSpeak introduce un'ulteriore astrazione chiamate Evento: questo si presenta quando un agente modifica la belief base o i propri goal a seguito di percezioni dell'ambiente o dell'esecuzione di un piano. Gli agenti reagiscono a tali eventi seguendo un modello formalmente ben definito e predicibile che porta all'eventuale trasformazione degli stessi in successive intenzioni da voler soddisfare mediante un opportuno piano di azioni.

Il modello formale di ragionamento di un agente AgentSpeak(L) può essere così riassunto (fig. 2.1) :

1. Seleziona un evento da elaborare.
2. Scorre la libreria di tutti i piani che abbiamo come evento di attivazione quello appena percepito, le cui precondizioni siano soddisfatte.

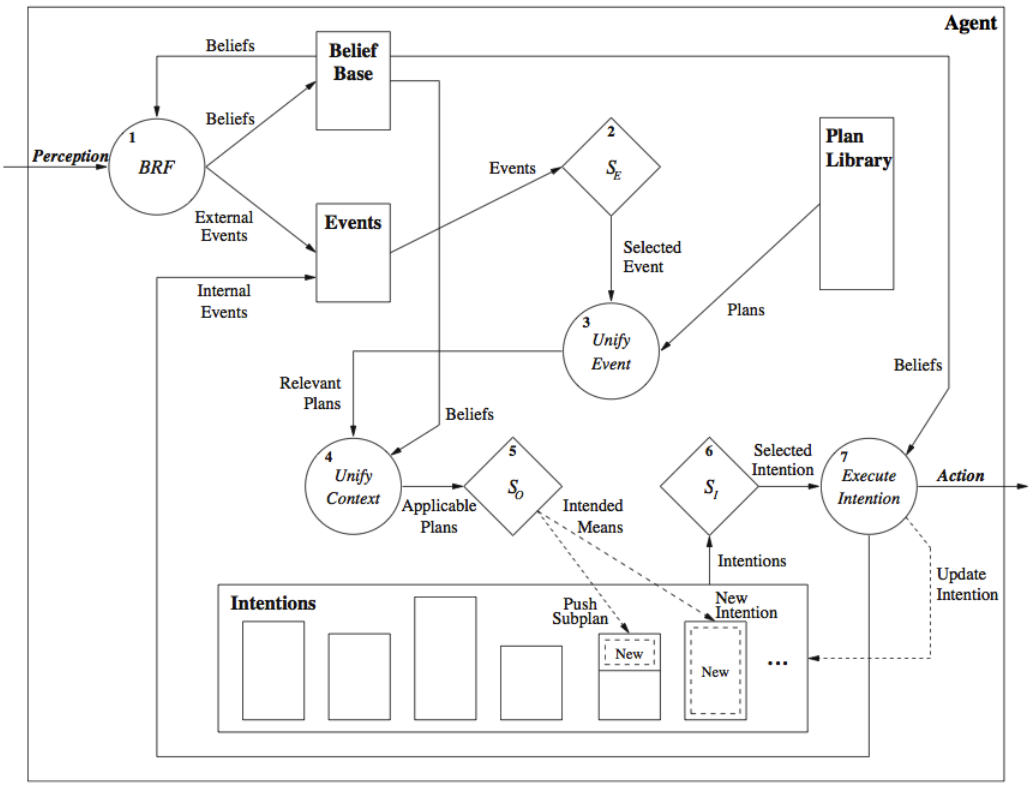

Figura 2.1: Ciclo di ragionamento di un agente in AgentSpeak(L)

3. Seleziona uno dei piani generati al punto precedente, etichettandolo col nome dell'evento che lo ha generato.

4. Crea una nuova entry nella lista delle intenzioni, ed mettendo in cima il piano appena definito.

5. Seleziona un'intenzione e considerare il prossimo passo del piano al suo top per l'esecuzione.

6. Se il piano al top è completo, si considera il piano successivo; se l'intenzione è vuota la si può considerare eseguita con successo, e dunque rimuoverla.

2.4.2 Jason

Jason è un interprete progettato e realizzato come una versione estesa in Java di AgentSpeak(L) aggiungendo al modello originale costrutti per il supporto alla comunicazione inter-agente basata sugli speech-act consentendo di avere agenti distribuiti sulla rete attraverso l'uso di SACI [16]. È stato realizzato da R. H. Bordini e J. F. Hubner [7] e rilasciato sotto licenza opensource; la sua specifica è stata formalizzata mediante una semantica operazionale fornita per AgentSpeak(L).

Oltre a interpretare il linguaggio predecessore, di cui possiede tutte le caratteristiche, Jason offre ulteriori funzionalità:

- negazione forte
- gestione del fallimento di piani
- supporto per lo sviluppo di environments programmabili
- supporto per una libreria di "azioni interne"
- possibilità di eseguire un sistema multi agente distribuito su una rete usando SACI
- funzioni di selezione, funzioni di fiducia e architettura complessiva dell'agente (percezioni, revisione di belief, comunicazione inter-agente e funzionamento) completamente modificabili secondo esigenze specifiche

- protocollo KQML per lo scambio di informazioni e comunicazioni tra agenti

Seguendo la definizione degli autori [8]:

- I Beliefs rappresentano le informazioni disponibili agli agenti (riguardanti l'ambiente o gli altri agenti); essi sono formalmente descritti secondo la semantica operazionale con termini di primo grado.

- I Goals rappresentano lo stato del sistema che l'agente vuole raggiungere – i cosiddetti "Achievement Goals", ovvero i fatti che vorrà che fossero asseriti a seguito delle azioni intraprese – oppure l'atto di reperire informazioni dalla BeliefBase – "Test Goals".

- Un agente reagisce ad eventi eseguendo i Piani.

- Gli eventi scaturiscono come conseguenza di cambiamenti nelle credenze o nei goal dell'agente.

- I Piani sono l'insieme delle azioni per raggiungere il determinato scopo per i quali sono stati definiti; rappresentano in un certo senso il "know-how" dell'agente;

- I Piani sono strutturati come segue:

$$triggering\ event : context <- body$$

  dove: "triggering event" denota l'evento che il piano andrà a gestire; "context" rappresenta le circostanze per cui il piano può essere utilizzato; "body" è la lista delle azioni che devono essere eseguite al fine di raggiungere l'obbiettivo; un piano si dice "applicabile" se risponde al triggering event corrente e context risulta un'espressione vera nella Belief Base corrente.

L'interprete di Jason è costituito da un motore (engine) che determina le transizioni tra gli stati del sistema in esecuzione. Uno stato del sistema è costituito da diverse componenti: le componenti citate sono in parte elementi dell'agente come l'insieme dei belief e quello dei

piani, le funzioni di selezione e di fiducia mentre per il resto queste componenti sono date dalle circostanze e dall'ambiente specificato nel file di configurazione del MAS. Una circostanza, formalmente definita come una tupla, non è altro che una classe Java rappresentante un insieme eterogeneo di strutture dati adatte per la memorizzazione di vari elementi quali:

- la coda degli eventi (o event queue);
- l'insieme delle intenzioni attive;
- l'insieme dei messaggi ricevuti (o mailbox);
- l'insieme dei piani rilevanti;
- l'insieme dei piani applicabili;
- l'azione correntemente in esecuzione;
- l'evento scelto per essere trattato;
- l'intenzione correntemente in esecuzione;
- l'opzione scelta per l'esecuzione, costituita da una coppia i cui elementi sono un piano e l'unificatore che ne determina l'applicabilità;
- una lista di azioni eseguite con l'indicazione del risultato di tale esecuzione (insieme dei feedback delle azioni);
- l'insieme delle azioni pendenti che sono state mandate in esecuzione e per le quali si attende un feedback.

Tutte le code e gli insiemi sopra elencati sono implementati con liste, gestite attraverso la politica di inserimenti e cancellazioni sono gestite con il criterio FIFO (First In-First Out). *Un evento è definito da una coppia* $< triggering\ event, intenzione >$: mentre il triggering event è l'evento vero e proprio (o azione interna) che viene scatenato da una modifica alla belief base o ai goal, l'intenzione associata è quella per cui verrà in seguito creato un piano. La creazione dell'Environment prevede la definizione delle percezioni iniziali positive (e se

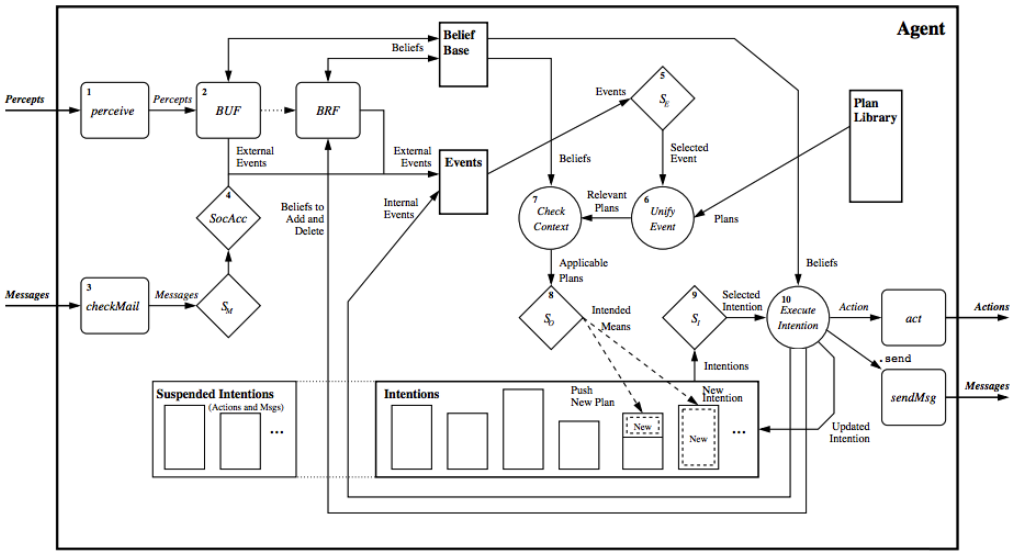

Figura 2.2: Ciclo di ragionamento di un agente in Jason

necessario negative) dell'agente. Esistono due tipi di azioni – quelle "di base" (che modificano lo stato dell'ambiente) e le cosiddette "azioni interne": queste non sono mirate a cambiare lo stato dell'ambiente (al contrario delle precedenti), ma rappresentano porzioni di codice che deve essere eseguito come parte del ciclo vitale dell'agente; inoltre, è anche un modo elegante di poter invocare codice legacy – cioè non direttamente legato al MAS – anche in altri linguaggi, eseguendo funzioni secondarie.

Il motore vero e proprio definisce il processo di esecuzione del "Reasoning Cycle" dell'agente, definito come in fig. 2.2, avviene in diverse fasi:

1. Controlla l'ambiente, percependo le eventuali modifiche dallo stato precedente

2. In seguito l'agente aggiorna la Belief Base, ponendola in consistenza con l'ambiente in quel determinato istante

3. Riceve le eventuali comunicazioni che sono arrivate da altri agenti
4. Seleziona un evento su cui focalizzare la sua attenzione – sia esso un cambiamento della belief base o un messaggio ricevuto
5. Ricerca tutti i possibili piani che soddisfano tale triggering event
6. Determina i piani applicabili – cioè quelli il cui contesto risulta vero
7. Seleziona un piano, mettendolo in cima allo stack dell'intenzione ad esso associata
8. Questa intenzione è inserita nella lista delle intenzioni
9. Seleziona un'intenzione per la futura esecuzione
10. Preleva ed esegue l'azione in cima allo stack del piano per l'intenzione appena selezionata

Il ciclo riparte così in un continuo avvicendarsi tra le fasi decisionali e quella esecutiva.

La rimozione dell'intenzione è subordinata al suo completamento. Se il corpo dell'istanza di piano in cima all'intenzione è vuoto e l'intenzione è vuota questa viene rimossa dall'insieme delle intenzioni attive e il processo di applicazione delle regole si arresta. Se invece l'intenzione contiene ancora delle istanze di piano, si rimuove quello appena completato, si seleziona il successivo dal cui corpo si preleva un elemento e si aggiorna l'unificatore dell'istanza di piano corrente componendolo con quello dell'istanza di piano appena rimossa. A questo punto si procede controllando se ora l'intenzione può essere rimossa. Se il corpo dell'istanza di piano non è vuoto il processo di applicazione delle regole si arresta.

Caratteristica peculiare di Jason è quella di poter gestire il fallimento nell'esecuzione di un piano, come una sorta di rollback o di clean-up. Questo processo infatti è eseguito in maniera molto elegante, rappresentato dall'esecuzione di un ulteriore piano di recovery, al fallimento del precedente attraverso l'uso della cancellazione di un goal: anche se sintatticamente definito, in pratica esso non ha un senso

logico, o in altre parole, una semantica associata. Concettualmente, il fallimento di un piano legato ad un goal scatena un nuovo evento nell'agente, il quale è definito come la negazione del goal stesso: il processo di ragionamento così ripartirà, con la ricerca di piani che soddisfino tale task.

Capitolo 3

Arduino

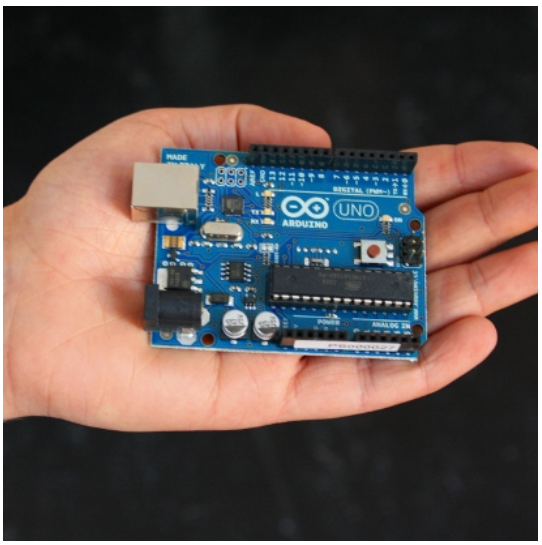

Figura 3.1: Arduino Uno

Arduino è una piattaforma di prototipazione elettronica open-source basata su una combinazione hardware-software flessibile e facile da usare[2]. Fondatori del progetto furono Massimo Banzi e David Cuartielles, presso l'Interaction Design Institute di Ivrea, nel 2005. Il progetto prese avvio con lo scopo di rendere disponibile, a progetti di Interaction design realizzati da studenti, un dispositivo per il controllo

che fosse più economico rispetto ai sistemi di prototipazione allora disponibili. I progettisti riuscirono a creare una piattaforma di semplice utilizzo ma che, al tempo stesso, permetteva una significativa riduzione dei costi rispetto ad altri prodotti disponibili sul mercato. Ad ottobre 2008 in tutto il mondo ne erano già stati venduti più di 50.000 esemplari. Il suo intento primario è quello di aiutare artisti, designer, hobbisti e chiunque altro sia interessato a creare oggetti interattivi con l'ambiente, apprendendo facilmente i principi fondamentali dell'elettronica e della programmazione. L'idea vincente che ha portato al successo Arduino è il rilascio dell'intero progetto con licenza Creative Commons, con cui chiunque, con le adeguate conoscenze, può utilizzare gli schemi ed il software prodotto modificandolo, adattandolo senza pagare nessuna royalty.

Arduino può acquisire determinate informazioni dall'ambiente circostante, attraverso input da una vasta gamma di sensori (come temperatura, luminosità, posizione, networking...), influenzandolo, a sua volta, mediante l'utilizzo di attuatori (luci, motori...). È costituito da una piattaforma di prototipazione rapida per il physical computing su cui trova spazio un microcontrollore, il quale può essere programmato usando l'Arduino Development Enviroment mediante l'apposito linguaggio di programmazione.

3.1 Hardware

Una scheda Arduino consiste in un controllore ad 8 bit Atmel AVR con l'aggiunta di componenti complementari per facilitare la programmazione e l'integrazione con altri circuiti. Un importante aspetto di Arduino consiste nello standard con cui i connettori sono esposti, permettendo alla scheda principale di essere collegata ad una moltitudine di moduli add-on (conosciuti come Shields) con funzionalità aggiuntive. Le specifiche ufficiali delle schede Arduino prevedono l'uso di microprocessori ATmega8, ATmega168, ATmega328, ATmega1280 e ATmega2560, dipendentemente dalla versione utilizzata. Oltre a questo la scheda è dotata di un regolatore lineare di tensione a 5 volt, un oscillatore al cristallo a 16 MHz, ed altri circuiti atti alla comunicazione seriale con il PC per il caricamento dei software. La scheda di

prototipazione espone la maggior parte dei propri pin di I/O per essere usati dal programmatore attraverso connettori: ad esempio Arduino Uno mette a disposizione quattordici porte di Input/Output digitali, di cui sei possono produrre segnali in modulazione di frequenza (PWM) e sei input analogici, mediante connettori di tipo femmina. Esistono in commercio i cosiddetti shield, schede aggiuntive plug-in che estendono le funzionalità base di Arduino (come ad esempio connettività ethernet, wireless oppure gsm, display a 7 segmenti o lcd, motori stepper o passo-passo etc...).

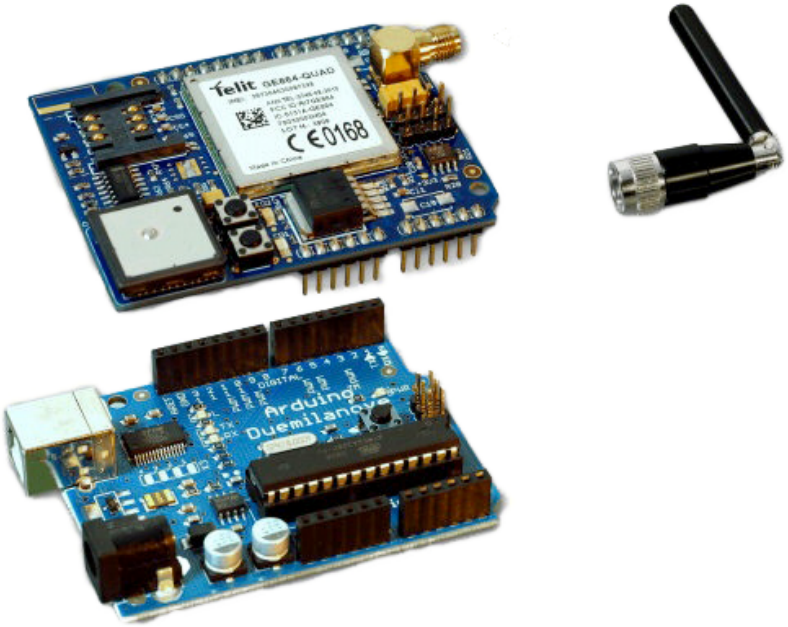

Figura 3.2: Esempio di Shield GSM-GPS per Arduino

3.2 Software

Per quanto riguarda la parte software possiamo analizzare Arduino su tre visuali: il linguaggio utilizzato, l'ide e il sistema di bootloader/caricamento dello sketch sulla scheda.

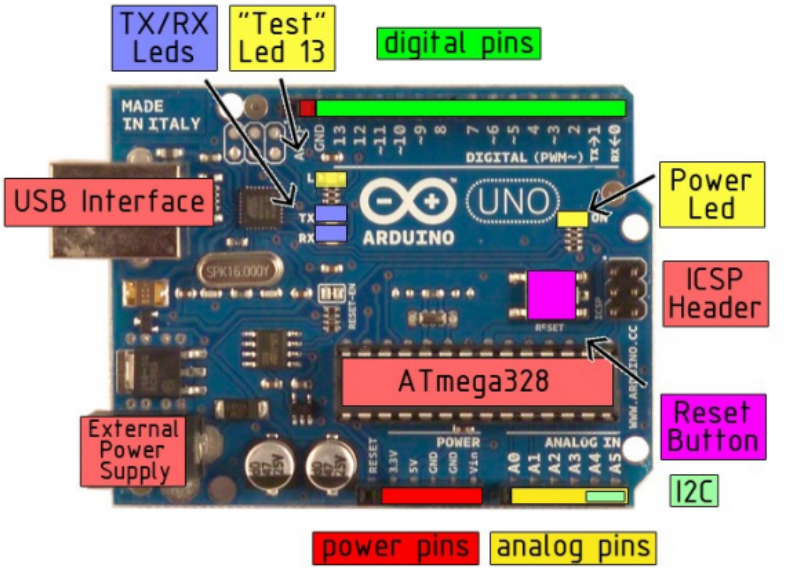

Figura 3.3: Pinout di Arduino Uno

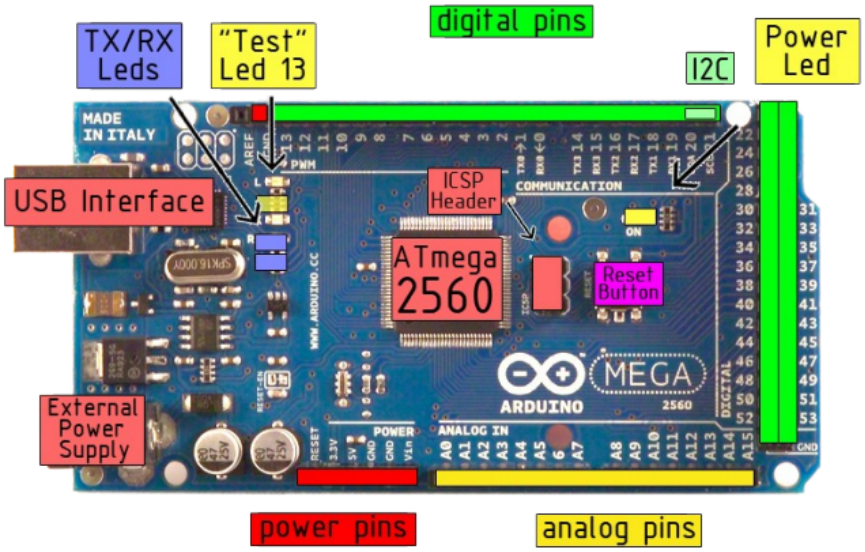

Figura 3.4: Pinout Arduino Mega 2560

3.2.1 Il Linguaggio

Arduino si basa su Wiring, un linguaggio di programmazione derivato da C/C++ ed una piattaforma hardware per applicazioni elettroniche, grafiche ed artistiche. Nato nel 2003 da Hernando Barragán presso l'Universidad de Los Andes si è presto orientato al comparto elettronico grazie ai progetti strettamente collegati quali Processing (da cui eredita la sintassi) e lo stesso Arduino. Wiring è organizzato come uno Sketchbook, ovvero un insieme di piccoli programmi – detti sketch – i quali realizzano determinate funzioni; questi poi vengono scaricati all'interno della memoria del microprocessore per essere eseguiti. L'utilizzo di Wiring da parte di Arduino permette agli sviluppatori di fare comuni operazioni di input/output facilmente, attraverso un ambiente C/C++ realizzato appositamente per sistemi a microprocessore. Per quanto riguarda il modo con cui si programma Arduino, rispetto alla classificazione fatta nel primo capitolo, possiamo affermare che sia attualmente utilizzato l'approccio a Single Control Loop. Analizzando infatti la documentazione del progetto, ci accorgiamo di come i programmi debbano definire almeno due metodi:

1. *void setup()*, funzione chiamata una sola volta alla partenza del programma. Spesso si utilizza questo metodo per inizializzare l'ambiente, come il modo di funzionamento dei pin – input o output – aprire una connessione seriale etc., prima che il processo di loop() abbia esecuzione.

2. *void loop()*, esecuzione continua delle linee di codice contenute dentro a questo blocco, fino a quando il programma non viene terminato. Il numero di volte con cui il metodo loop() viene eseguito al secondo, dipende dalla complessità del codice e può essere controllato tramite funzioni quali delay() e delayMicroseconds().

Il preprocessore incluso in Wiring avrà il compito di integrare il codice così realizzato in uno scheletro che favorirà l'esecuzione dello sketch in maniera corretta, secondo la semantica dei metodi appena definita. Il programma così ottenuto viene compilato con le apposite librerie AVR rilasciate dal produttore; in seguito – attraverso l'uso combinato

di IDE e del bootloader – viene caricato nella memoria del processore di Arduino: il processo viene definito "upload" del firmware.

Listing 3.1: Esempio di codice Arduino

```
void setup(){
    // Metodo invocato una sola volta
}

void loop() {
    // Invocazione continua del metodo loop
}
```

3.2.2 Il Bootloader

Il bootloader è un piccolo firmware che i progettisti di Arduino hanno inserito nei chip che vengono forniti con la board il cui scopo principale è quello di caricare gli sketches nel microprocessore senza nessun altro hardware esterno (programmatore). Ogni qual volta si accende la scheda, viene eseguito il bootloader – lo si può notare perché il led posizionato sul pin 13 lampeggia – il cui funzionamento è quello di rimanere in stand-by qualche istante per verificare una eventuale comunicazione da parte dell'IDE: stabilita una connessione tra pc e scheda, si effettua il trasferimento dello sketch nella flash memory del microprocessore; qualche secondo dopo il bootloader procede al lancio dello sketch appena caricato. Se, in fase di start up, nessun comando arriva dal pc, il bootloader di default fa partire l'ultimo sketch caricato durante l'ultima sessione di comunicazione con il pc; se nessuno sketch è mai stato caricato sul chip, il bootloader sarà l'unico programma che verrà eseguito e continuerà a resettarsi in attesa di un ciclo di programmazione. L'uso di un bootloader ha permesso ai progettisti della scheda di evitare l'utilizzo di un programmatore esterno montato sulla scheda, risparmiando quindi spazio fisico ed ulteriori costi; d'altra parte si introduce un ritardo durante la fase di start up del sistema e un minore spazio disponibile per lo sketch vero e proprio: seppur di minime dimensioni, il bootloader ha un peso da non sottovalutare in sistemi come questi, ove la memoria è una risorsa scarsa.

3.2.3 L'ambiente di sviluppo

L'Integrated Development Enviromnet (IDE) utilizzato da Arduino è un'applicazione cross-platform scritto in Java, derivante dall'IDE di Processing, che include un editor di codice, ed un meccanismo semplice per l'interfacciamento con la board. Tanto minimale dal punto di vista

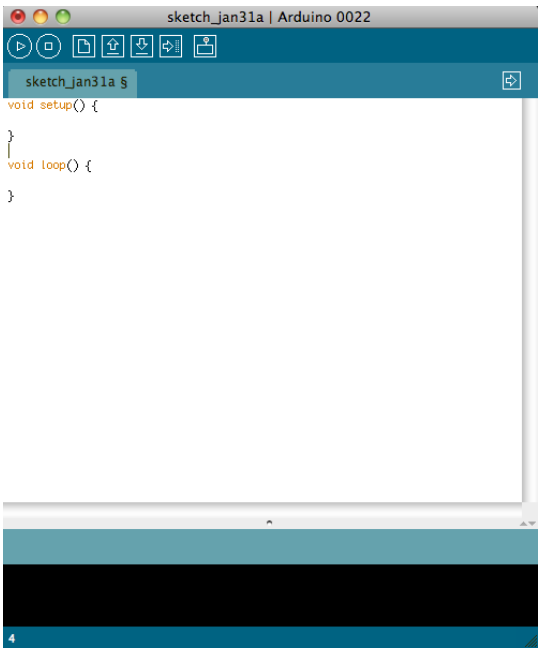

Figura 3.5: Ide Arduino 1.0

grafico quanto semplice e potente per un utente con poca esperienza con compilatori e programmatori hardware: dopo aver scritto il codice dell'applicazione – avendo cura di aver precedentemente selezionato la porta seriale (creata quando si collega Arduino al PC tramite cavo USB) su cui trasmettere ed il tipo di scheda in proprio possesso – premendo il pulsante Upload si innesca il processo per il quale l'IDE dapprima compilerà il programma alla ricerca di errori sintattici, e in seguito si occuperà del trasferimento dello sketch verso il processore. A tal fine per prima cosa effettuerà un riavvio della scheda, in modo da permettere al bootloader di mettersi in attesa di comunicazione per un eventuale caricamento di un nuovo firmware. Stabilendo così

la comunicazione seriale, lo sketch compilato precedentemente viene trasferito alla flash memory del processore con conseguente avvio del programma in esso contenuto.

3.2.4 Android Device Kit

In occasione della conferenza Google I/O di maggio 2011, Google ha presentato ADK (Android Device Kit), un kit di sviluppo basato su Arduino e dedicato alla produzione di accessori in collaborazione col suo nuovo sistema operativo per smartphone, Android. Attualmente è disponibile il solo supporto per USB ma nei prossimi mesi arriverà anche il supporto per il Bluetooth. L'ADK, basato attualmente sulla scheda Arduino Mega2560, consente l'interfacciamento di un dispositivo dotato del sistema operativo Android verso una scheda Arduino per la creazione di applicazioni integrate tra i due device. Lo spettro di applicazione di questa nuova interazione è ampiamente immaginabile, e può andare dalla creazione di accessori per lo smartphone ad ambiti domotici e persino di controllo industriale. La versione di An-

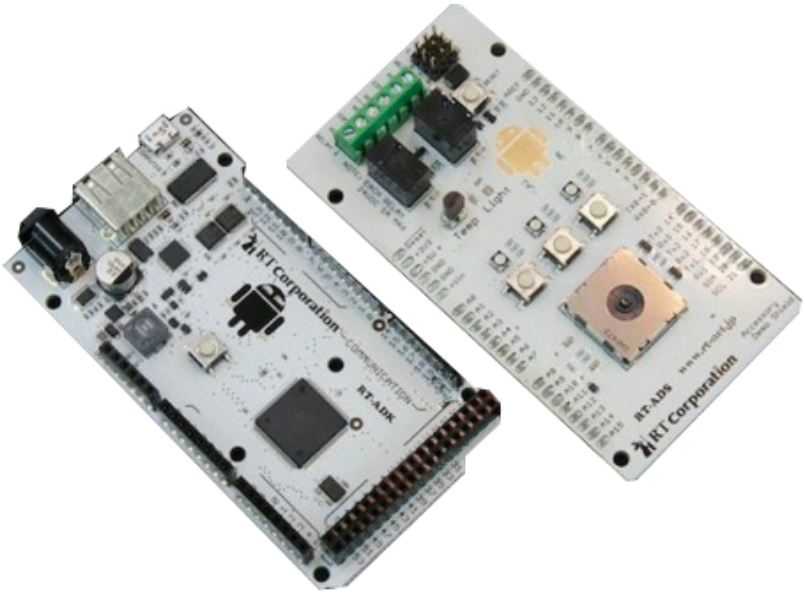

Figura 3.6: Google Android Device Kit

droid supportata è la 3.1 [1] introducendo la nozione di Android Open Accessory che permette ad hardware esterno l'interazione mediante porta USB: quando un accessorio è collegato questo diviene automaticamente "l'host" – il quale alimenta il bus e enumera i devices – e il device Android agisce come un normale dispositivo USB. Gli accessori devono essere specificatamente progettati per collegarsi ai dispositivi Android aderendo al semplice protocollo che permetta loro la corretta individuazione reciproca; inoltre comporta che l'host alimenti il device secondo gli attuali standard che prevede il protocollo (vale a dire 500mA a 5V)

3.3 Alternative

A seguito di Arduino, e del suo enorme successo, alcune industrie e semplici ingegneri hanno cercato di proporre soluzioni concorrenti. Tra decide di piattaforme per il rapid prototyping vogliamo citarne in particolar modo due: l'MBED e il Texas Instruments Launchpad.

3.3.1 MBED

Mbed è un'azienda start-up nata nel 2005 in seno ad ARM (di cui è anche la principale sponsor insieme ad NXP) che progetta e realizza schede per lo sviluppo rapido di sistemi embedded a microcontrollori. Come Arduino, lo scopo didattico/hobbistico è stato il principale obbiettivo, introducendo la piattaforma performante ARM in questo ambito. L'attuale terza generazione della scheda è equipaggiata con un controllore Cortex-M0 che unisce alte prestazioni a un ridotto consumo; la particolare conformazione permette di collocare la piattaforma direttamente su breadboard, semplificando le connessioni a componenti esterni, accelerando ulteriormente i tempi di sviluppo. Vediamo in breve le maggiori caratteristiche hardware:

- NXP LPC11U24 MCU - 32 bit
- ARM® Cortex(TM)-M0 Core a basso consumo
- Frequenza di clock 48MHz, 8KB RAM, 32KB FLASH per mantenere dati e programma

- Dispositivo dotato di collegamento USB, 2xSPI, I2C , UART, 6xADC, GPIO
- Form-factor a 40-pin DIP package, 54x26mm
- 32 pin, di cui 6 utilizzabili come I/O analogici
- 5V USB, tensione di alimentazione 4.5-9V oppure batteria da 2.4-3.3V
- Built-in USB drag'n'drop FLASH programmer

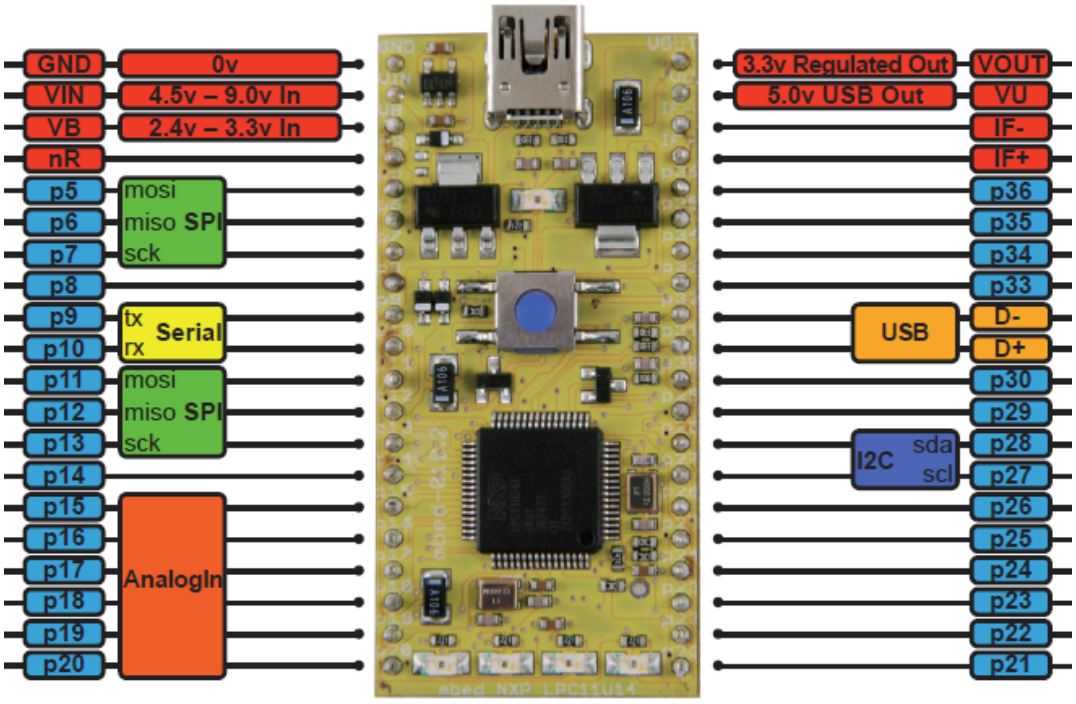

Figura 3.7: Microcontrollore MBED (NXP LPC11U24) e relativo pinout

Il top delle prestazioni viene invece espresso dalla scheda NXP LPC1768 che monta un controllore Cortex-M3, il quale raddoppia la frequenza di funzionamento e raggiunge 512KB di memoria flash; ma

ancora più sorprendente è la capacità di integrare al suo interno una quantità enorme di dispositivi addizionali sia per il controllo dell'I/O, che delle comunicazioni. Infatti esso ha supporto pieno per lo standard Ethernet, Usb, Can bus, I2C e SPI, oltre a 3 linee per la comunicazione seriale; inoltre esso presenta 6 porte controllabili in PWM (Pulse Width Modulation).

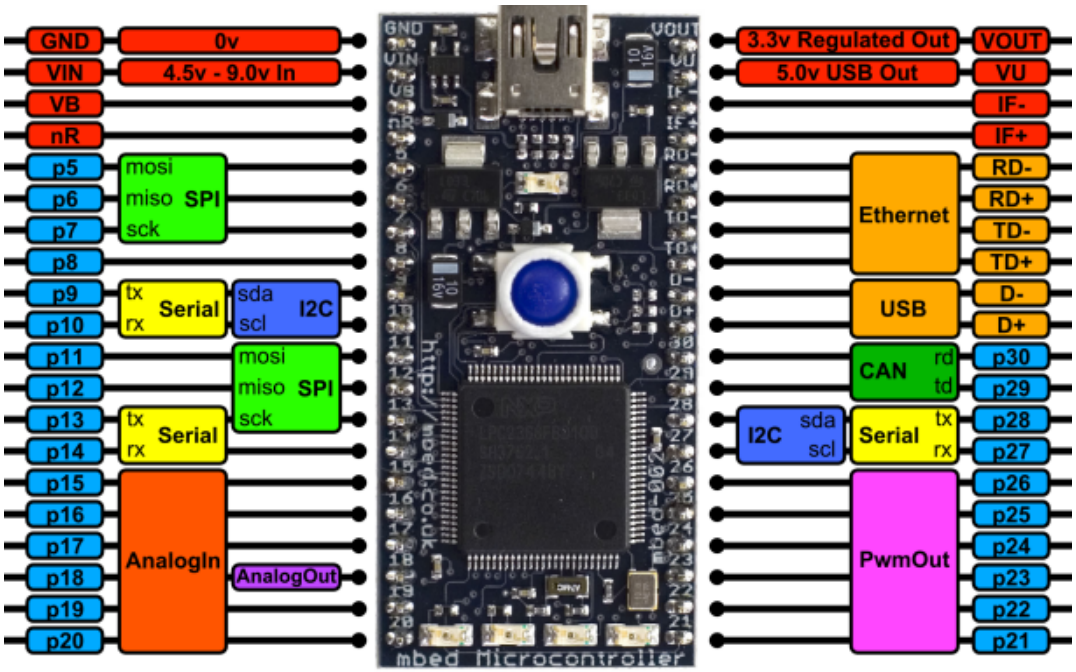

Figura 3.8: Microcontrollore MBED (NXP LPC1768) e relativo pinout

Caratteristica peculiare di queste piattaforme proviene dall'IDE: questo infatti è completamente online, unificato tra le varie tipologie di schede, disponibile sul sito della casa produttrice, come anche lo stesso compilatore: in questo modo lo sviluppo del software astrae dal sistema operativo utilizzato, in quanto tutto il processo di compilazione avviene in remoto; attraverso un semplice drag'n'drop sarà poi possibile scaricare il programma all'interno del microcontrollore. Il codice viene scritto in linguaggio C/C++ di alto livello, compilato in un ambiente web AJAX-style, integrando una fortissima componente

sociale, vale a dire un luogo dove gli utilizzatori possono scambiarsi informazioni, discussioni, codice e documentazione.

3.3.2 Launchpad

Texas Instrument lanciò questo nuovo prodotto nel Luglio 2010, con un prezzo decisamente accattivante, 4.30$, (dalle 5 alle 10 volte inferiore a quello degli attuali rivali) promuovendo così i suoi microcontrollori MSP430: infatti su tale scheda è possibile montare qualsiasi controllore di questa linea, fino a quelli a 20 pin. Questi sono caratterizzati da un'architettura a 16 bit, un bassissimo consumo, specialmente in Low Power Mode, un sistema di debugging molto efficiente ed avanzato incluso in un IDE – il Code Composer Studio – derivato dalla piattaforma Eclipse.

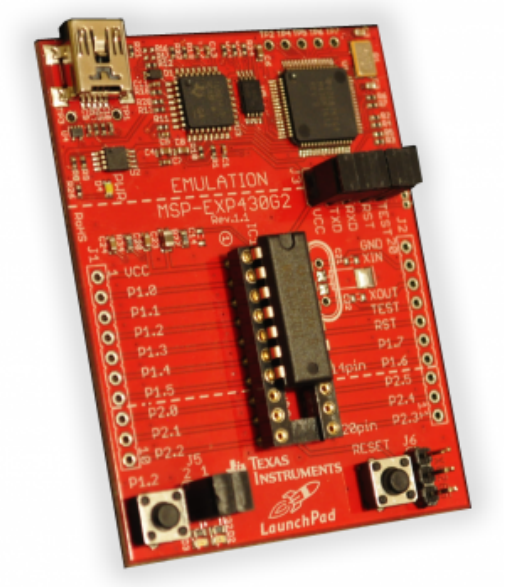

Figura 3.9: Microcontrollore MSP430 Launchpad

La scheda è costituita da 3 parti, logicamente indipendenti: oltre al microprocessore e relative porte di I/O, è presente una parte per la comunicazione, programmazione, emulazione e debug ed una terza in

cui sono presenti due led ed un pulsante per lo sviluppo di applicazioni in modo rapido; la configurazione della piattaforma ricorda quella di Arduino, con la possibilità di montare shield (tra i quali quelli realizzati dalla stessa TI, come il Capacitive Touch BoosterPack). La tensione operativa è di 1,8-3,6V (al contrario dei 5V del concorrente) che unita ai bassissimi consumi dell'architettura 430 fanno di questo un'ottimo strumento per dispositivi di nuova generazione volti alla riduzione della potenza consumata: tale piattaforma infatti, può essere alimentata facilmente da piccoli pannelli solari o addirittura da materiale organico ed essere totalmente indipendente (si pensi infatti a scenari come reti di sensori wireless o ridotte applicazioni aerospaziali). Una caratteristica importante è la possibilità di poter utilizzare interrupt hardware su tutta la gamma di porte di input dall'esterno: come vedremo poi questa potrebbe essere fonte di importanti ottimizzazioni software.

Capitolo 4

Un DSL ad agenti BDI per Arduino

Innalzare il livello di astrazione di un linguaggio di programmazione significa innanzi tutto introdurre nuovi concetti e principi, dando così la possibilità allo sviluppatore di non preoccuparsi degli aspetti basilari. Incapsulare proprietà e comportamenti complessi, attraverso l'uso di funzioni e librerie è sempre stato l'approccio più adottato e perseguito: in questa tesi invece si vuole riformare alla base il meccanismo di scrittura di un programma per la piattaforma Arduino, rivoluzionando il concetto di controllo, mascherandolo al punto giusto. L'entità attiva, l'agente, come punto focale e del quale lo sviluppatore ne dovrà specificare task ed eventi al quale è interessato e piani da mettere in pratica.

Tra i diversi modi con cui sviluppare un processo che porti alla definizione dell'astrazione di agenti su questa piattaforma hardware, ho scelto quella di articolare un Domain Specific Language corredato da un apposito generatore di codice: in questo modo è possibile definire la sintassi e semantica dell'agente una ed una sola volta, creando di fatto un modello comportamentale ben definito. Basterà in seguito adattare di volta in volta il generatore di codice alle varie piattaforme utilizzate, estendendo il concetto di agente non solo ad Arduino, ma anche virtualmente a tutte quelle esistenti. Inoltre si amplierà in questo modo il concetto di portabilità del software, in quanto a partire dallo stesso codice sorgente del comportamento di un dato agente, que-

sto potrà essere eseguito su altre piattaforme, soltanto ri-generando il codice appropriato.

In questo capitolo descriverò il processo che mi porterà alla definizione di un modello di agenti BDI per la piattaforma Arduino. A tale scopo, analizzerò in primo luogo le limitazioni hardware e software che la stessa piattaforma impone in un ambito così complesso quale può essere la specifica del comportamento di un agente.

In questa parte definirò più specificatamente le nozioni di Belief, Desire, Intention adattandole al contesto applicativo, introducendo ulteriormente il concetto di Event come già discusso in precedenza; come termine di paragone, riassumerò in breve l'implementazione BDI in stile JASON.

4.1 Vincoli di Arduino

Nonostante sia una piattaforma molto evoluta, Arduino presenta notevoli limitazioni che possiamo suddividere in:

- *Memoria*
- *Capacità di elaborazione*
- *Numero di Porte I/O*
- *Numero di Interrupt*

Vediamo nei paragrafi seguenti in cosa consistono queste limitazioni rispetto all'ultima piattaforma disponibile al momento della scrittura di questa tesi – cioè prendendo come riferimento Arduino Uno, equipaggiato di microcontrollore Atmega 328. In alcuni casi riporteremo anche le caratteristiche di Arduino Mega 2560, che rappresenta la massima espressione di Arduino per prestazioni, mettendo a confronto gli ordini di grandezza in gioco.

4.1.1 Memoria

Indubbiamente la memoria, sia Flash (in cui risiedono bootloader e programma) che RAM, rappresenta la restrizione maggiore nei contesti

embedded: disponendo infatti di soli 32KB di memoria flash – di cui 0,5KB utilizzati dal bootloader – e di 1KB di EEPROM (memoria dati non volatile in cui possono essere mantenuti l valori delle variabili, disponibili anche dopo un reset del sistema) possiamo facilmente capire come l'ottimizzazione del codice e la minimizzazione del numero di variabili sia di fondamentale importanza in questo ambito.

Per quanto riguarda Arduino Mega, il contesto memoria è leggermente migliore – 256KB di flash e 4KB di EEPROM – ma non ancora sufficiente a permettere un utilizzo spregiudicato della stessa. Mantenere complesse strutture dati particolareggiate, o che descrivono complesse funzioni, è dunque una soluzione non praticabile, pertanto il contenimento dello spazio sarà la strategia da intraprendere, mettendo in gioco variabili o piccole strutture solo ove strettamente necessario.

4.1.2 Capacità di elaborazione

Il primo fattore che determina la capacità di elaborazione di un microprocessore embedded è sicuramente la frequenza di clock del sistema. Entrambe le piattaforme discusse presentano una velocità massima di 16MHz, di molto inferiore – almeno due ordini di grandezza – rispetto un normale PC di ultima generazione. Inoltre, non tutte le operazioni vengono eseguite in un unico ciclo di clock: ad esempio la lettura (o scrittura) di pin analogici possono impiegare fino a 13 cicli[3]. Inoltre l'infrastruttura che verrà generata partendo dal codice dell'agente, influirà enormemente sulle prestazioni, portando con sé un notevole overhead. Aumentare infatti il livello di astrazione, passando da codice C-like (quale è Wiring) a un'infrastruttura ad agenti implicherà un diverso livello di complessità del programma rispetto al loop standard, come poi evidenzierò nei paragrafi successivi. Tuttavia la capacità di elaborazione deve essere messa in relazione al sistema da realizzare: in ultima istanza sarà dunque necessario effettuare prove a posteriori valutando le prestazioni ottenute, come indice delle performance, per dare un'indicazione di massima guidando il futuro sviluppatore nella scelta di questa astrazione in condizioni di ambiente fisico critico.

4.1.3 Porte di Input/Output

Le porte (pin) di Input/Output rappresentano il modo con cui l'agente può ricevere informazioni dall'esterno mediante sensori ed agire sull'ambiente attraverso attuatori. Quante più variabili presenta il sistema, quante più porte saranno necessarie al microcontrollore per esaminarle e poterle controllare. Anche in questo caso è impossibile definire a priori quale sia il numero di pin adatto: ogni problema richiede un diverso adattamento. Tuttavia, possiamo affermare che maggiori interfacce consentono un quantitativo maggiore di applicazioni realizzabili; inoltre sono anche fonte di dinamicità nello sviluppo del codice: il sistema di controllo infatti può crescere via via, aumentando le proprie interazioni con il mondo, durante la progettazione e realizzazione, in modo incrementale. Detto questo, vediamo come le due piattaforme Arduino siano molto diverse: mentre la più comune presenta 20 pin I/O (di cui 6 analogici e 14 digitali), ArduinoMega ha un notevole quantitativo di pin supplementari (in totale 16 analogici e 54 digitali).

4.1.4 Interrupt

Gli interrupt sono di fondamentale importanza nello sviluppo di codice ottimizzato ed efficiente: essi infatti permettono di interrompere il normale flusso di controllo del processore, richiamando l'attenzione dello stesso verso un evento accaduto all'esterno. Grazie a questa tecnica si evita il cosiddetto "polling", la procedura per cui si è necessario controllare continuamente il valore di una variabile o, come in questo caso, di una porta. Gli interrupt sono anche alla base degli agenti: possiamo notare, infatti, che questo concetto è molto vicino a quello di evento, cioè il cambiamento di stato dell'ambiente a cui possiamo essere interessati. In ambito elettronico/embedded è possibile settare interrupt su determinate porte, in modo da poter eseguire codice (spesso una routine) all'avvenimento di determinate condizioni sul pin specificato, su:

- una transizione da valore logico Alto a Basso
- una transizione da valore logico Basso ad Alto

- una qualsiasi transizione di valore logico

Questo meccanismo è presente in Arduino, ma purtroppo con una forte limitazione sulle porte di ingresso, in quanto non è possibile sfruttare gli interrupt su tutti i pin del microcontrollore, ma solo su determinate porte stabilite dal costruttore. Si tratta infatti solamente dei pin 2 e 3 sulla piattaforma Arduino Uno (2 pin digitali su un totale di 14) – 6 pin per Arduino Mega su 54 disponibili; nessun interrupt invece è presente su porte analogiche.

4.2 Il modello ad agenti proposto

4.2.1 Reasoning Cycle

A partire dalle limitazioni sopra riportate, descriverò ora un modello comportamentale per lo sviluppo di agenti su piattaforma Arduino. È evidente come una mappatura 1:1 dei concetti espressi da un linguaggio già presente ed evoluto come JASON, non sia una soluzione riproducibile, in quanto mancano tutta una serie di meccanismi ed automatismi (quale il matching, l'unificazione, etc) la cui realizzazione sia poco performante. È dunque necessario un adattamento del modello ad agenti classico, utilizzando strumenti di più basso livello offerti da Arduino/Wiring, che però catturino gli elementi strutturali di questa astrazione.

Il processo di ragionamento dell'agente è strutturato in un loop continuo, che esegue i seguenti passi:

1. L'agente percepisce beliefs ambientali

2. Per ogni variazione di interesse rilevata si crea un nuovo evento nell'opportuna coda

3. Si sceglie un evento nella coda di eventi, attraverso la funzione di selezione del piano

4. L'agente cerca nella Plan Library i piani che soddisfano le condizioni al contesto

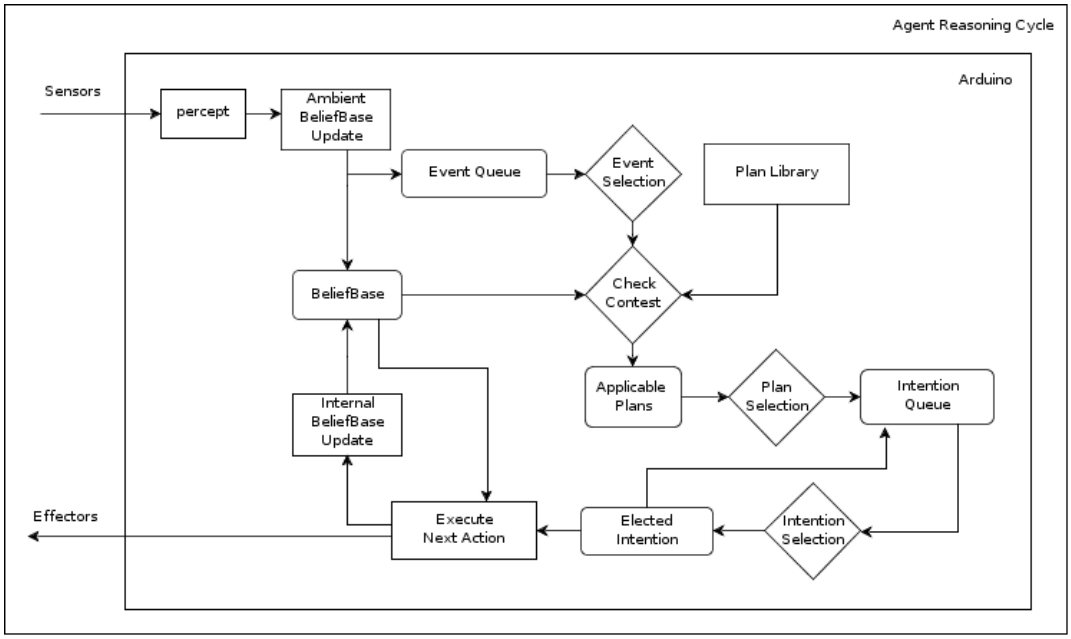

Figura 4.1: Ciclo di update dei Beliefs e identificazione di un Evento specifico

5. Tra i piani rilevati ne assegna uno all'evento, attraverso la funzione di selezione del piano

6. L'evento associato al piano diventa un'intenzione e viene messa nell'apposita coda

7. Una intenzione viene prelevata dalla coda, mediante opportuna funzione di selezione

8. L'agente procede con la prima istruzione non ancora eseguita del piano, aggiornando se necessario i beliefs interni all'agente

9. L'intenzione viene rimessa nella Intention Queue

4.2.2 Beliefs

I beliefs sono le credenze dell'agente, le sue conoscenze riguardanti l'ambiente esterno. Esiste dunque, uno stretto collegamento tra i beliefs ed i pin d'interfaccia della scheda Arduino; mentre in JASON questi vengono memorizzati attraverso l'aggiunta o la rimozione di predicati del primo ordine, in Arduino verranno realizzati mediante la creazioni di apposite variabili. Possiamo inoltre distinguere due tipi di beliefs: quelli il cui riferimento è l'ambiente esterno, ove è presente una stretta correlazione tra variabile e pin, e quelli interni, come una sorta di stato interno non affetto direttamente dal mondo, ma conseguenza della storia dell'agente. Li possiamo definire, rispettivamente beliefs ambientali e beliefs interni: entrambi verranno mappati in codice come variabili, con la differenza che i primi saranno soggetti a continui aggiornamenti, mentre i secondi no. Tra i beliefs ambientali possiamo anche distinguere tra i segnali analogici e quelli digitali: in un'ottica di ottimizzazione del codice ogni pin digitale verrà mappato su una variabile di tipo byte mentre ogni pin analogico verrà memorizzato su una variabile di tipo int (si veda la Reference di Arduino su digitalRead - digitalWrite - analogRead - analogWrite). Per i beliefs interni invece darò la possibilità di poter utilizzare i tipi di dato che la piattaforma mette a disposizione: void, boolean, char, unsigned char, byte, int, unsigned int, word, long, unsigned long, float, double, string(char array), (String - object), array.

4.2.3 Events

Gli eventi sono definiti come il cambiamento di stato dell'ambiente o, come in questo caso, dei beliefs che lo rappresentano; il problema dunque si divide in due: la cattura di una o più variazioni e la rappresentazione della stessa. Per quanto riguarda la cattura si pro-

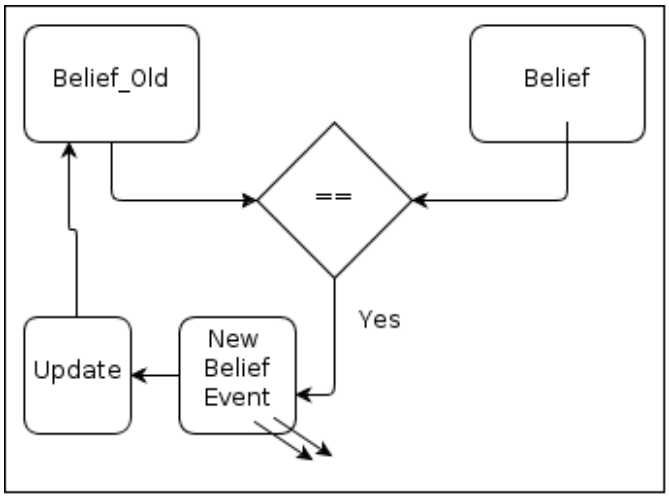

Figura 4.2: Ciclo di update dei Beliefs e identificazione di un Evento

pone il mantenimento temporaneo di una copia di ciascuna variabile identificata come belief, confrontandola col nuovo valore letto ad ogni ciclo dell'agente: una variazione identificata dalle regole di Event (sotto definite)specificate nel codice dell'agente porterà alla generazione di un evento, posto a sua volta in un'apposita coda di elaborazione. Per quanto riguarda segnali digitali, analizzando la natura degli stessi, possiamo identificare un insieme di variazioni a cui l'agente può rispondere:

- La variazione da stato logico alto a basso (falleding edge)
- La variazione da stato logico basso ad alto (rising edge)
- Una qualunque variazione dello stato logico (rising or falleding edge)

Per quanto quanto riguarda invece segnali analogici possiamo riproporre un ragionamento analogo, sostituendo ai valori HIGH e LOW un valore di soglia, catturando così i seguenti eventi:

- Il segnale attraversa un valore di soglia dall'alto verso il basso (falleding edge)
- Il segnale attraversa un valore di soglia dal basso verso l'alto (rising edge)
- Una qualunque variazione delle precedenti (rising or falleding edge)
- Un valore entra in un range limitato di valori (ranged value)

Per cui è bene diversificare i comportamenti dell'agente, a seconda del tipo di evento a cui si vuole che crei un evento. Un evento sarà definito da una struttura atta a mantenere i seguenti dati:

- Timestamp dell'occorrenza
- Nome dell'evento accaduto

Ad ogni ciclo di percezione più eventi possono verificarsi contemporaneamente: è dunque necessario un meccanismo per il mantenimento di questi, creando così una "coda". Quest'ultima si realizzerà mediante un array dinamico di strutture di eventi; data la ridotta capacità di memoria della piattaforma sarà necessario limitare ad un certo numero la dimensione massima della coda di elementi. Successivamente un opportuno modulo provvederà a selezionare un evento alla volta e trasformalo in una Intention.

4.2.4 Tasks

Un'astrazione di agente in grado di rispondere ed agire solamente in seguito ad eventi occorsi all'ambiente non incapsulerebbe il concetto di soddisfacimento di un goal (o task) a lungo termine, ma soltanto un comportamento reattivo. In questi termini è dunque necessario introdurre la nozione di task, senza però aggiungere una struttura dati apposita (cioè senza creare un Goal Base): i desideri sono considerati

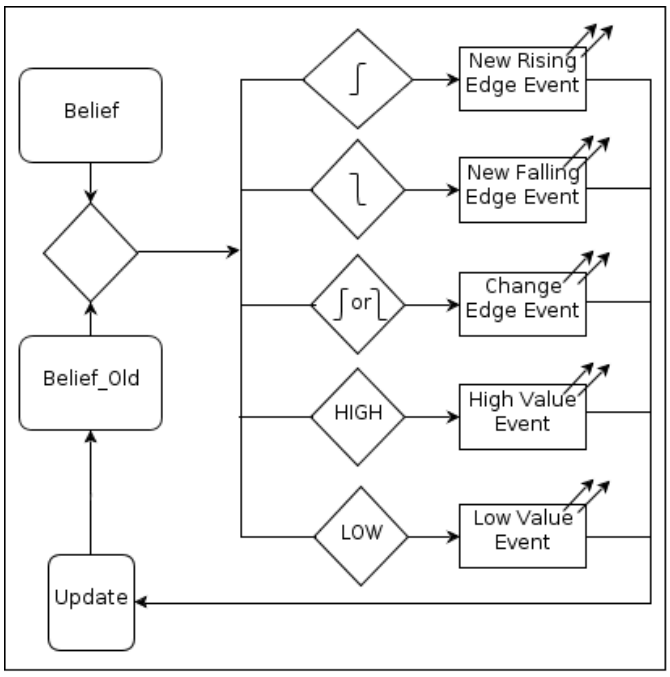

Figura 4.3: Ciclo di update dei Beliefs e identificazione di un Evento specifico

alla stregua di eventi, automaticamente attivati al momento della loro aggiunta, seguendo il medesimo percorso nel ciclo di ragionamento sopra riportato. Così come per gli eventi, che sono un insieme numerabile, è necessario definire a priori i tutti i task, specificando quali siano disattivi al momento dell'inizializzazione del sistema. Al fine di ottenere una migliore gestione ed un livello maggiore di flessibilità sarà possibile l'attivazione e la disattivazione degli stessi; queste due azioni interverranno dunque sulla coda degli eventi, aggiungendone di nuovi:

- All'attivazione di un goal, verrà creata una nuova intenzione, associando un opportuno piano;
- All'atto di disattivazione verrà eliminata dalla coda delle intenzioni quella associata al desire in oggetto.

4.2.5 Intentions ed IntentionsBase

Al fine di realizzare gli obbiettivi dell'agente, ad ogni evento a cui esso è interessato, è bene specificare un opportuno comportamento in reazione a tale occorrenza. Così al presentarsi di un evento, quale possa essere una variazione di un pin digitale, un attraversamento della soglia di un segnale analogico o in risposta ad un goal, verrà associato a questo un piano di esecuzione, il quale conterrà una lista di istruzioni da eseguire. Più in generale, nel medesimo tempo t possono essere generati più eventi: è necessario dunque un'apposita funzione che elegga uno di essi a diventare prossima intenzione. Tale funzione però non è oggettiva, ma può variare in base all'ambito applicativo: dalla più semplice politica FIFO (First In - First Out) oppure associando un valore di priorità a ciascun evento, scegliendo successivamente quello a maggior priorità. Nel corso dello sviluppo di questo progetto ho adottato selezione FIFO per l'elaborazione della coda di eventi. Selezionatone uno, si passa alla scansione della libreria dei piani, alla ricerca di un possibile candidato all'esecuzione: viene eletto quello che risponde all'attuale esigenza (cioè quello che ha come oggetto il suddetto evento) e che ulteriormente soddisfa i criteri di validità del contesto (vale a dire un'espressione booleana) che devono risultare senza impedimenti al momento della selezione e che ne convalidano l'uso del piano in tale

situazione. Così come per gli eventi, diversi piani possono soddisfare la stessa coppia $< evento, condizioni >$ allo stesso tempo, e analogamente occorre una funzione di selezione del piano opportuno ad essere eseguito: nel contesto di questa tesi, sceglierò una politica piuttosto rigida, eleggendo come piano volto ad essere eseguito il primo che soddisfa i criteri sopra descritti. Nel caso in cui un agente non trovi un piano, o che nessuno di questi soddisfi il contesto, l'evento sarà ignorato. Nello specifico l'intention non possiede una struttura fisica ma coincide con l'istanziazione di un piano da parte della Intention Base: quest'ultima infatti manterrà l'elenco dei piani attivi, facendo da tramite con l'agente per l'attivazione, reset ed esecuzione delle singole azioni. Si occuperà inoltre di implementare una politica di round robin tra le esecuzioni dei piani, così da permetterne l'interleaving degli stessi, mostrando così un certo grado di contemporaneità dei task.

4.2.6 Piani

Ogni piano, come ampiamente descritto nei paragrafi precedenti, sarà costituito non solo da un elenco di istruzioni, ma anche dall'evento oggetto per cui verrà eseguito ed una condizione al contesto. Tale condizione, rappresentata come una funzione con parametro di ritorno booleano, viene valutata al tempo della generazione dell'evento e con le credenze dell'agente in quell'istante e serve a descrivere le precondizioni per cui tale piano è abilitato all'attuazione. I piani possono essere eseguiti in due modi:

- *Blind execution* consiste nell'esecuzione del piano senza nessun interruzione da parte di altri eventi provenienti dall'esterno. In questo modo l'attuazione del piano verrà considerata atomica, a scapito però della percezione dell'ambiente: infatti in tale situazione l'agente sarà come cieco nei confronti del sistema, inerte nel rispondere ad altri eventi.

- *Purely Reactive execution* è la tecnica per cui ad ogni istruzione del piano l'agente torna ad eseguire il proprio ciclo di controllo di eventi esterni; una volta che un piano è assegnato ad un certo evento, rimane istanziato finché non termina la sua esecuzione,

mentre la condizione di contesto deve essere valida solo all'atto dell'assegnazione e può diventare false in qualsiasi momento senza alcun effetto.

Ciascun approccio ha i suoi pregi e difetti, ognuno in maniera duale all'altro: mentre la blind execution contiene l'inevitabile overhead introdotto dall'astrazione, impedisce all'agente di catturare eventi che possono verificarsi nel mondo durante l'esecuzione; d'altra parte un sistema puramente reattivo introduce un alto overhead, mantenendo però un maggior livello di coerenza e aggiornamento dei suoi beliefs, eseguendo sempre il miglior piano per le condizioni ambientali ad ogni istante. Nel corso di questa tesi abbiamo creduto opportuno realizzare un sistema di esecuzione puramente reattivo, intervallando alla fase di sense l'esecuzione di una sola istruzione di un determinato piano. Altra forte assunzione è quella concorrenzialità tra task: essi infatti devono poter essere eseguiti parallelamente, o in maniera tale da avere un certo grado di concorrenza. Ovviamente il microprocessore fornito con Arduino non è in grado di eseguire parallelamente più istruzioni né di gestire un sistema multithreading come ad esempio un PC dotato di adeguato Sistema Operativo. La soluzione sarà dunque data da un'interleaving tra le istruzioni non solo dei piani, ma anche quelle che permettono all'agente funzioni sensoriali ed esecutive.

4.3 Sviluppo di un DSL in Xtext

Definite così le astrazioni di prim'ordine con cui vogliamo profilare l'agente possiamo fornire, attraverso una grammatica EBNF-like, una sua descrizione formale come segue:

Listing 4.1: Definizione di un agente tramite il nome

```
1
Agent:
  'agent' ID '{' AgentBody? '}'

5 AgentBody:
  'beliefs{' Belief* '}'
  ('tasks{' Goal (',' Goal)* '}')?
```

```
  ('events{' Event+ '}')?
  Plan*;

Belief:
  ( 'belief' |'sensor' | 'effector' ) ID 'as' Type

Event:
  ID ':' ('raised'|'falled'|'changed'|'between')
  BeliefTypes (INT (INT)?)?;

Goal:
  ID ('disabled')?;

Plan:
  'plan' ID '{'
    (('event' Event) | ('task' [Goal]))
    'context' BoolExpr
    'body {'
      Action*
    '}'
  '}';
```

Volendo creare un Domain Specific Language è stato in primo luogo necessario scegliere un opportuno strumento di modellazione del linguaggio. Xtext (giunto alla versione 2.1) è un ottimo framework/tool con cui descrivere DSL testuali personalizzati facendo uso della grammatica EBNF ed, attraverso uno specifico generatore, creerà un parser, un metamodello dell' Abstract Sintax Tree ed un funzionale editor per la scrittura in linguaggio, con un'ottima funzione di autocompletamento Eclipse-like. Il tutto viene fornito come un plugin, anch'esso progettato per Eclipse, di facile installazione ed utilizzo. Inoltre con Xtext è fornito anche Xtend, un linguaggio template-based che permette lo sviluppo agile di generatori di codice ed altri programmi che fanno uno di concatenazione di stringhe. Il suo output è un generatore di codice in linguaggio Java il quale sarà automaticamente applicato al linguaggio sviluppato terminando così il ciclo di scrittura del codice nel nuovo linguaggio, generazione dell'albero associato, e traduzione dello

stesso in un'ulteriore linguaggio compilabile. Vediamo così che quest'insieme di strumenti è perfetto per lo scopo di questa tesi, la quale punta a descrivere un nuovo linguaggio agent-like, traducendolo poi in linguaggio C/C++ Wiring compilabile su piattaforma Arduino. Per quanto riguarda lo sviluppo al codice generato sono state passate in rassegna numerose librerie di supporto esistenti, tra cui anche diversi sistemi operativi, che fornissero strumenti al fine di realizzare un certo grado di concorrenzialità tra i task; di notevole interesse è stata scelta la libreria "Protothreads" [10] [11](http://www.sics.se/ adam/pt/ rilasciata con licenza BSD-like open source) con la quale è possibile specificare una forma di concorrenzialità simile a threads, ottimizzando le prestazioni e l'uso di memoria su sistemi a scarse risorse quali, appunto, gli embedded systems.

4.4 Sintassi e semantica associata

Specificato il modello dell'agente che andrò a realizzare, è necessario definire una sintassi dello stesso, in modo da permettere allo sviluppatore di creare nuovi agenti. Nella stesura di regole sintattiche mi sono liberamente ispirato ai linguaggi di programmazione più utilizzati (in particolare Java e C++) combinando nozioni provenienti dall'ambiente Jason-like; rimane da dire che tuttavia la sintassi di un DSL è altamente soggettiva e context-depending: in ogni caso cercherò di non portare ambiguità nel modello, proponendo la mia personale visione.

Prima di tutto si procede con identificazione dell'agente attraverso un nome:

Listing 4.2: Definizione di un agente tramite il nome

```
agent NomeAgente {
  // Specifiche
}
```

Questo aiuterà in fase di generazione del codice, associando il nome dell'agente al nome del progetto Arduino che verrà creato

Passiamo ora alla definizione degli statement per ogni struttura che abbiamo sopra definito, partendo dalle credenze dell'agente:

- *belief* indica una credenza interna, come una sorta di proprietà dell'agente come frutto della sua esperienza
- *sensor* è il modo con cui lo sviluppatore indica quali sono i pin che vengono utilizzati per aggiornare la belief base
- *effector* specifica quali sono i pin che, associati ad una credenza, fungeranno da attuatori

Listing 4.3: Sintassi dei beliefs

```
agent NomeAgente {

  beliefs {
    belief internalBel int
    sensor analogBel as A1
    effector digitalBel as D1
  }

}
```

Ho ritenuto che anche lo stato degli attuatori sia associabile, in un certo senso, ad un belief: l'agente infatti deve poter mantenere traccia delle operazioni che sta eseguendo, e lo fa utilizzando la belief base come strumento per tale scopo. Grazie a ciò è anche possibile specificare agilmente forme di conflittualità tra piani: una condizione che imponga un effettore spento inibirà il piano stesso quando l'attuatore risulterà in funzione.

Anche se non esiste una Goal Base vera e propria, lo sviluppatore deve essere in grado di specificare i goal che l'agente deve perseguire: a tal fine ho ritenuto opportuno creare un blocco di codice ove sono specificati i goal, indicando per ognuno se essi siano attivi o meno allo start del sistema.

Listing 4.4: Impostazione dei goal da raggiungere

```
agent NomeAgente {

  beliefs { ... }
  goals {
```

```
    goal1 enabled, goal2 disabled
  }

}
```

Sono principalmente due i momenti in cui viene creato un evento a partire da un goal: all'atto dell'inizializzazione del sistema – per ogni goal attivo – e quando passa dallo stato disabilitato a quello abilitato; l'evento creato avrà lo stesso nome del goal e potrà attivare un piano identificato dallo stesso.

Listing 4.5: Sintassi degli eventi

```
agent NomeAgente {

  beliefs { ... }
  goals { ... }
  events {
    evento1: raised digitalBel
  }
}
```

Gli eventi occupano un loro spazio nel codice dell'agente: vediamo come siano identificati da un nome (univoco), il tipo di evento ed il belief a cui è applicato. Vediamo qual'è la sintassi e semantica delle tipologie di eventi:

- $<$ **raised** *beliefName* $(threshold - optional, foranalogsignal)$ $>$ quando il valore di una credenza su un pin di ingresso digitale passa dallo stato logico basso ad alto, oppure quando una credenza su una porta analogica sale al di sopra del valore di soglia specificato

- $<$ **falled** *beliefName* $(threshold - optional, foranalogsignal)$ $>$ quando il valore di una credenza su un pin di ingresso digitale passa dallo stato logico alto a basso, oppure quando una credenza su una porta analogica scende al di sotto del valore di soglia specificato

- < **between** *beliefName* *thresholdLow* **and** *thresholdHigh* > evento per segnali analogici, attivato quando il valore entra nel range stabilito dalle due soglie
- < **changed** *beliefName* > una qualsiasi variazione del valore di una credenza, valido per segnali analogici e digitali

Listing 4.6: Definizione dei Piani

```
agent NomeAgente {

  beliefs { ... }
  goals { ... }
  plan {
    event goal1
    context analogBel <= internalBel
    body {
      //Instruction List
    }
  }
  plan {
    goal goal1
    context analogBel != internalBel
    body {
      //Instruction List
    }
  }
}
```

Per ultimo rimane la compilazione dei piani che gli agenti potranno eseguire in risposta ad eventi o task: infatti come prima clausola abbiamo proprio la keyword event o goal seguito dal nome. Interessante funzionalità offerta da Xtext è il completamento automatico ed il riconoscimento degli errori: infatti in questo punto potranno essere selezionati solo eventi o goal precedentemente specificati, riducendo così gli errori che possono essere commessi da parte dell'utilizzatore. Per quanto riguarda la clausola context essa rappresenta una funzione booleana che, come ho già detto, deve risultare vera nel momento in cui si presenta l'evento affinché il piano possa essere ritenuto eleggibile

a soddisfare la situazione; verrà dunque valutata all'istante opportuno come descritto nel ciclo di ragionamento dell'agente.

Il blocco **body**, scritto in un linguaggio di alto livello agent-like, specifica quali siano le azioni che devono essere eseguite; le istruzioni che preliminarmente vengono implementate sono

- L'assegnamento di un valore ad un effector (es. myAnalogEffector = 512), comporta automaticamente la scrittura del pin;
- Operazioni matematiche su belief;
- Tutte le principali operazioni di controllo algoritmiche quali blocco if - then - else, cicli while, do ... while, for;
- delay milliSec, permette al task di poter aspettare un determinato tempo prima di eseguire la prossima istruzione; al contrario della corrispondente istruzione arduino, questa è non bloccante, e quindi non comporta il blocco nell'esecuzione di altri task;
- serialOut, istruzione per la comunicazione seriale con l'IDE di Arduino.

4.5 Struttura del generatore di codice

In primo luogo è stato necessario, a partire dall'analisi del progetto appena descritta, creare una opportuna grammatica EBNF per la definizione delle principali strutture del nuovo linguaggio. Così lo sviluppo si è diretto verso una progressiva specifica di regole per il parsing di Agentino, a partire dai blocchi base (beliefs, events, goals e plans) fino poi alla definizione delle operazioni aritmetico/booleane e delle istruzioni di controllo. Molta cura è stata data nell'uso del cross reference, feature chiave di Xtext, la quale permette un'identificazione, ancora prima della compilazione, degli errori più comuni che un utente può commettere; in questa maniera il tool ci permette inoltre di fornire all'utente una funzionalità di autocompletamento del codice sicuramente interessante e pratica. Il tool infine suggerisce automaticamente quali sono i blocchi e le parole chiave da utilizzare segnalando preventivamente errori sintattici.

Listing 4.7: Regole di definizione dell'agente e dei Piani

```
Agent:
    'agent' name=ID '{'
    (agentBody=AgentBody)?
'}';

AgentBody:
      'beliefs {' beliefsList+=BeliefTypes* '}'
      'goals {' ((goalsList+=Goal)
       (',' goalsList+=Goal)*)? '}'
      ('events {' (eventList += Event)+ '}')*
      (plansList+=Plan)+;

Plan:
      'plan' name=ID '{'
       (('event' eventName=[Event]) |
    ('goal' goalName=[Goal]))
  'context' contextExpr=BoolExpr
  'body {'
        planBody+=Action*
  '}'
'}';
```

Vediamo un frammento di codice Xtext dove si può notare come vengano specificati i blocchi fondamentali del linguaggio:

- Elencazione delle parole chiave del linguaggio,rappresentate da stringhe racchiuse tra apici (come ad esempio *agent*, *beliefs*, *plans* ...)
- Associazione di regole di parsing ad oggetti, i quali saranno poi disponibili al generatore di template (es. name=ID o beliefsList+=BeliefTypes)
- Cross reference che viene identificato da un riferimento alla classe, come al punto precedente, racchiudendolo tra parentesi quadre (taskName=[Task] o eventName=[Event])

Possiamo così notare come nella regola Plan, ci siano due cross reference (in alternativa l'una all'altra), una per gli eventi ed una per

i goal: questo ci permette di referenziare i goal o gli eventi definiti in precedenza.

Qui sotto viene rappresentato un altro frammento di codice Xtext ove possiamo vedere alcune delle definizioni delle funzioni di controllo di flusso

Listing 4.8: Regole per la specifica delle istruzioni di controllo

```
Action:
    {Expression} action=BoolExpr |
    {ActionRule_If} action=Rule_If |
    {ActionRule_Do} action=Rule_Do |
    {ActionRule_While} action=Rule_While |
    {ActionRule_For} action=Rule_For |
    {ActionDelay} action=Delay;

Delay returns SingleAction:
    'delay' microSec=INT;

Rule_If returns SingleAction:
    'if' '(' expr=BoolExpr ')' '{'
  ifActions+=Action*
     '}' ('else' '{'
  elseActions+=Action*
    '}')*
;

Rule_Do returns SingleAction:
    'do' '{'
  doActions+=Action*
    '}' 'while' '(' expr=BoolExpr ')'
;


.
.
.
```

Oltre alla grammatica Xtext, che definisce la sintassi del linguaggio, è necessaria la creazione di un generatore, il cui compito è la tradu-

zione Model to Model da Agentino a C/C++ Wiring: ad effettuare il processo sarà il plug-in Xtend, il quale avrà a disposizione l'intero albero sintattico prodotto dal parser (Xtext) su cui agire. A questo punto rimane da definire una struttura del codice di output, come un template che verrà riempito con i dati dell'agente che lo sviluppatore avrà inserito, quali beliefs, task, event e plan. Vedremo, nella prossima sezione, come sarà strutturato il codice generato da Agentino e come tutte le varie parti interagiranno tra loro per ottenere il comportamento desiderato.

4.6 Struttura del codice generato

Possiamo suddividere il codice prodotto in output da Agentino in due parti: una parte strutturale invariante, composta dalle classi principali, ed una parte variabile nella quale vengono definiti i piani. Al fine di comprendere come vengano generati diversi files contenenti classi e funzioni di supporto del ciclo di ragionamento dell'agente si rimanda al Class Diagram di fig.4.4. La struttura di un normale programma Arduino non è stata modificata: nel file nominato "agentName".ino sono presenti i metodi setup() e loop() descritti nel capitolo 3 propri dell'astrazione della piattaforma; inoltre i sorgenti generati sono direttamente compilabili senza errori con la versione 1.0 dell'IDE. "agentName".ino contiene dunque la struttura fondante dell'agente, al cui interno possiamo ritrovare chiaramente il ciclo $sense, plan, act$ a cui abbiamo fatto riferimento per la maggior parte della tesi. La fase di set-up inizia con la creazione ed inizializzazione delle strutture dati fondamentali (quali la beliefs base, l'intention base e la coda di eventi); in seguito l'agente andrà a settare fisicamente quali siano i pin di input ed output mediante l'apposita funzione (pinMode) e a generare, posizionandoli nell'opportuna cosa, gli eventi corrispondenti ai task iniziali. A questo punto l'agente è pronto per compiere il ciclo vitale vero e proprio:

1. *sense()* è la funzione per la quale, in corrispondenza dei beliefs definiti nella struttura del programma Agentino, vengono aggiornati i parametri delle variabili interne associate al valore dei sensori. In questa fase vengono anche generati i possibili eventi

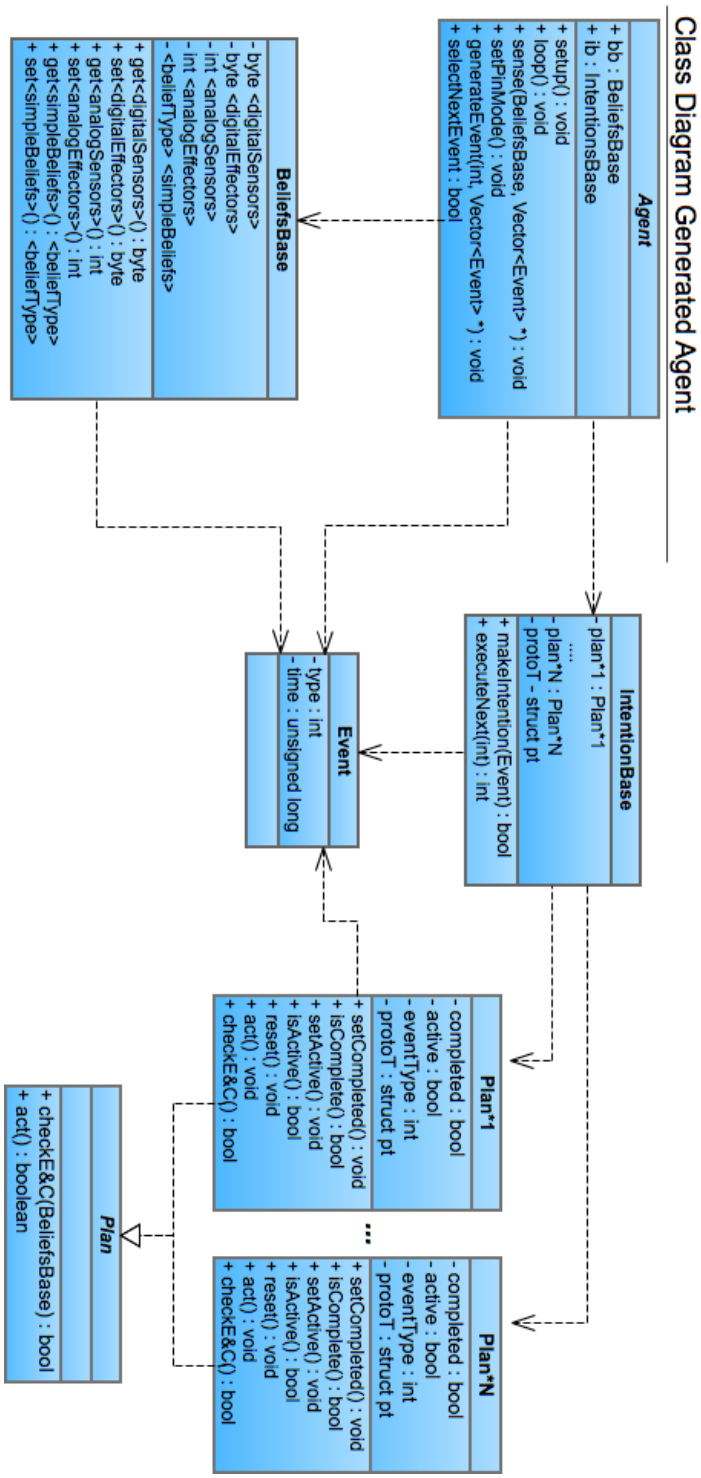

Figura 4.4: Class diagram generato da Agentino

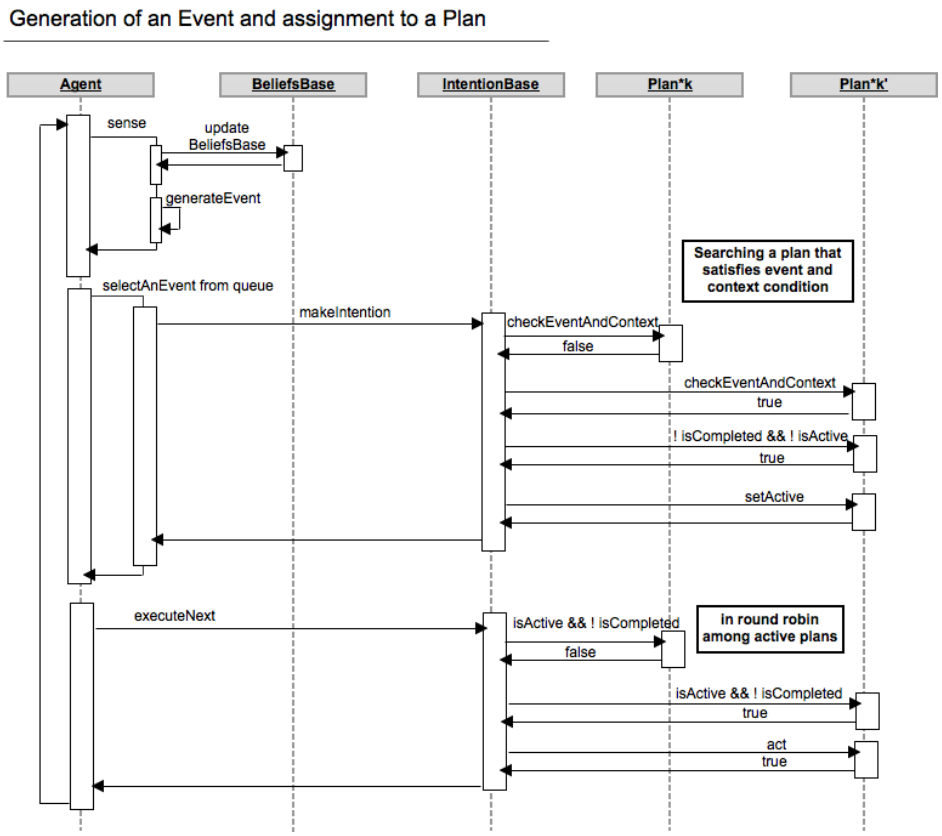

Figura 4.5: Diagramma del comportamento dell'agente

che possono essersi scatenati al seguito di una modifica del valore dei sensori. Ogni evento dunque sarà tradotto in un insieme di condizioni tali che, prendendo in considerazione il valore precedente e l'attuale del sensore di volta in volta considerato, se verificate daranno origine ad una variazione significativa a cui l'agente è interessato.

2. *selectNextEvent(eventQueue, nextEvent)* e *ib.makeIntention(nextEvent)*: la prima rappresenta la funzione di selezione del prossimo evento che nel nostro caso è stata semplicemente implementata come una politica FIFO applicata alla eventQueue, mentre la seconda chiama in causa l'oggetto IntentionsBase (ib) creando l'intenzione opportuna dato il prossimo evento da processare. A causa delle limitazioni di memoria di Arduino, si è preferito mantenere solo concettualmente l'idea di intenzione, senza creare un oggetto fisico in RAM, ma andando ad inizializzare direttamente il piano che soddisfa l'evento le cui condizioni di contesto siano verificate. All'interno della IntentionsBase dunque saranno mantenuti tutti i riferimenti ai piani attualmente disponibili, andando ad agire di volta in volta al momento in cui si presenta un evento, attivandoli o disattivandoli . Questo porta un'ulteriore forte assunzione in quanto due eventi distinti che accadono in breve tempo non portano all'istanziazione di due piani concorrenti: la semantica di compromesso trovata in questo caso porta all'esecuzione di un solo piano, creato al momento dell'elaborazione del primo fatto accaduto, mentre viene poi lasciato allo sviluppatore, se lo ritiene opportuno e necessario al caso di studio in a cui è applicato, il compito di controllare eventuali successivi eventi dello stesso tipo.

3. *ib.executeNext()* con tale dichiarazione indichiamo all'intentions base di eseguire una istruzione del piano eletto ad essere messo in atto in tale momento.

L'intentions base è una classe di grande importanza nella struttura dell'agente: come abbiamo già visto è quella reputata all'elaborazione degli eventi e alla gestione dei piani nonché della loro esecuzione. Essa infatti mantiene i riferimenti ai piani istanziati, permettendo di

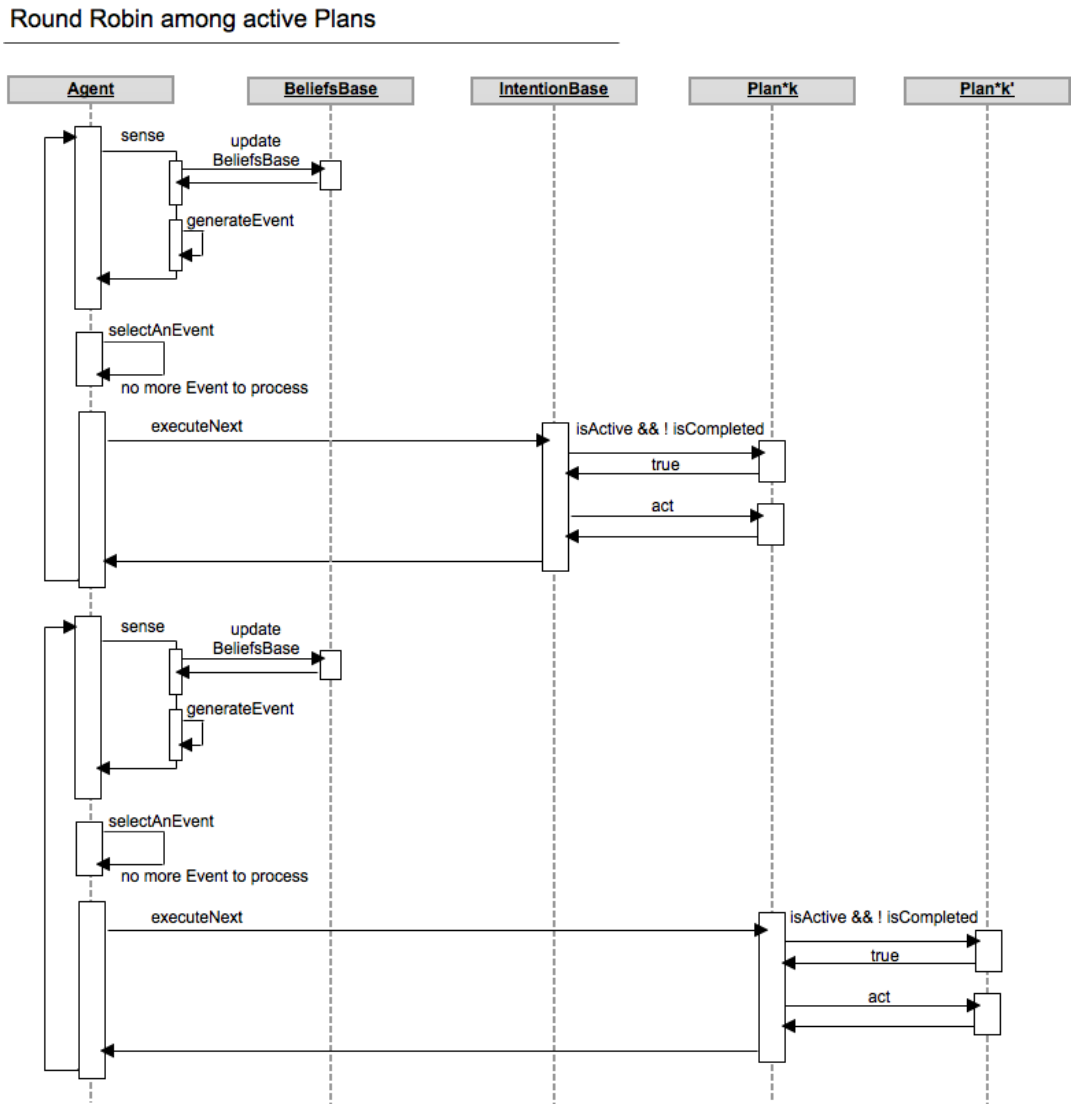

Figura 4.6: Diagramma del comportamento dell'agente

controllarli con le istruzioni di avviamento, stop, e reset; inoltre implementa la politica di Round Robin 4.6 tra i piani, permettendo loro l'esecuzione concorrente.

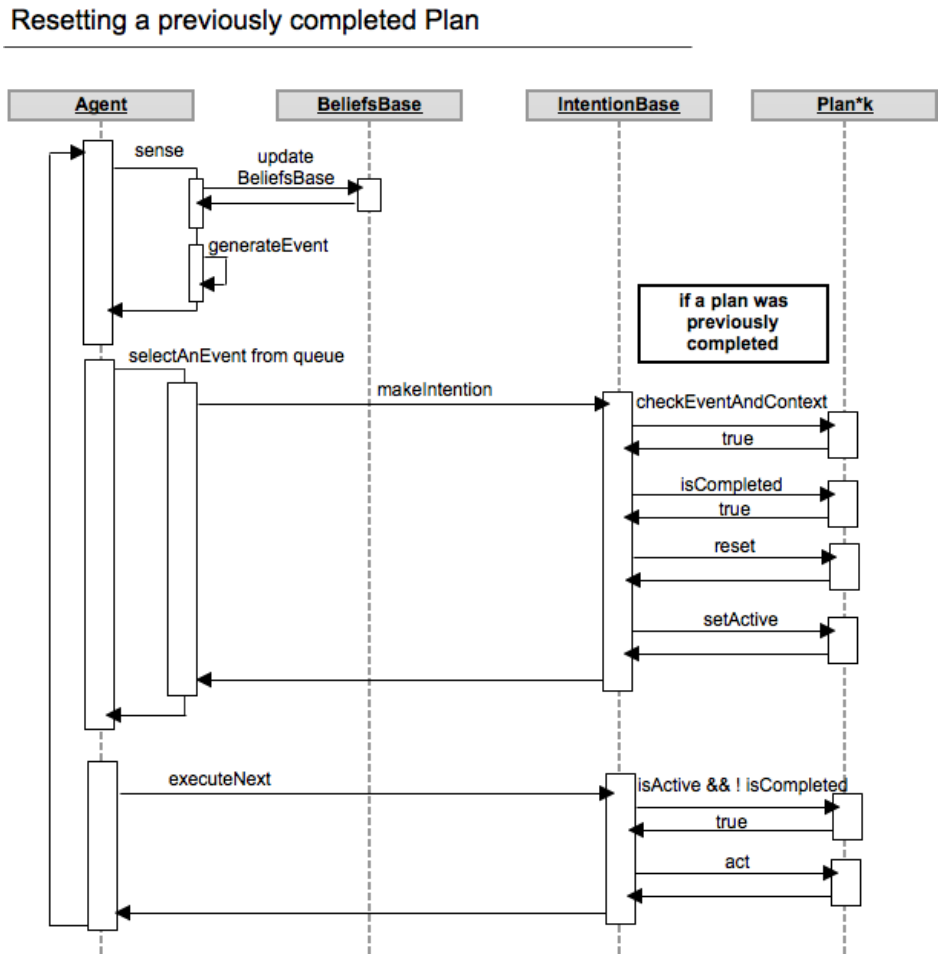

Figura 4.7: Diagramma del comportamento dell'agente

Proprio in questo caso ci viene in aiuto la libreria Protothreads: grazie infatti alla sua utilizzazione è possibile definire staticamente un ciclo di istruzioni di esecuzione dei piani, intervallate da direttive (cosiddette Wait) che permettono la sospensione della funzione come se si fosse in presenza di un context switch, ritornando il flusso di con-

trollo al ciclo loop di arduino, ma ricordandosi l'ultimo piano eseguito (vedi fig. 4.8). La nuova esecuzione della stessa funzione riporterà l'instruction pointer all'istruzione successiva alla wait, come una sorta di go-to interna. Inoltre, tale classe avrà il compito di controllare se, a fronte di un evento, il piano corrispondente risulta completato: in questo caso è possibile soddisfare l'Event resettando il Plan ad esso associato e re-inizializzarlo per la nuova esecuzione (fig.4.7)

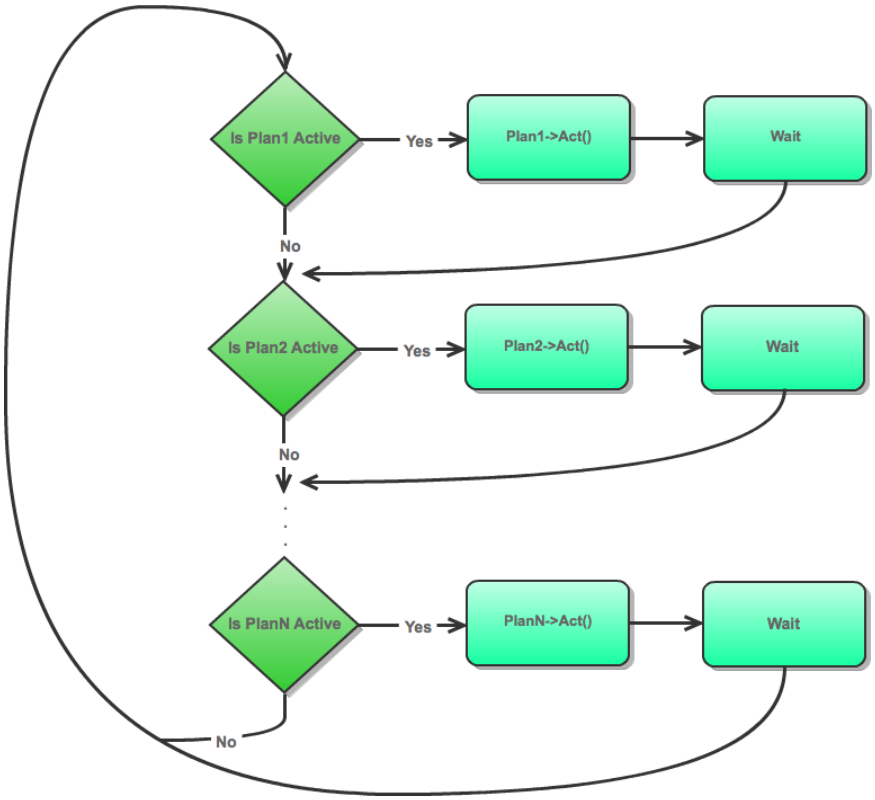

Figura 4.8: Diagramma del comportamento dell'agente

La BeliefsBase è una classe wrapper attorno ai belief, dotata di funzioni getter e setter per ciascuna di essi: le funzioni Get sono realizzate su misura in base al componente dai cui restituire il valore con il corretto tipo, definito in fase di scrittura del codice; discorso analogo per le funzioni di Set con la differenza che esse stesse sono in

grado, a fronte di un assegnamento ad un belief effettore, di pilotare la rispettiva porta di uscita associata. La beliefs base è dunque il componente che implementa fisicamente l'accesso al mondo esterno, come una sorta di interfaccia tra il sistema ed Arduino stesso. Possiamo notare che la lettura delle porte di input non viene effettuata da questo componente, ma dalla funzione sense(), principalmente per due ragioni: in primo luogo vogliamo incapsulare concettualmente la lettura e la generazione degli eventi in un'unico metodo e tempo di esecuzione; in secondo luogo, vogliamo che il valore del sensore letto rimanga invariato durante tutto il ciclo di ragionamento dell'agente, evitando così letture fantasma che potrebbero verificarsi. Questo è un punto fondamentale in quanto letture successive del medesimo pin, a fronte di cambiamenti repentini del sistema, potrebbero causare il non corretto funzionamento dell'agente; leggere e mantenere il valore della porta di ingresso rende il modello coerente e corretto.

Listing 4.9: Esempio di funzioni Getter e Setter della BeliefsBase

```
int BeliefsBase::getMyBeliefValue() {
  return myBeliefValue;
}
byte BeliefsBase::getPushButtonValue() {
  return pushButtonValue;
}
byte BeliefsBase::getLedValue() {
  return ledValue;
}

void BeliefsBase::setMyBeliefValue(int temp) {
  myBeliefValue=temp;
}
void BeliefsBase::setLedValue(byte temp) {
  ledValue=temp;
  digitalWrite(8, temp);
}
```

Infine rimane da descrivere Plan e le classi da essa derivate: Plan definisce in modo astratto le due principali funzioni di qualsiasi piano,

quali il controllo della condizione di contesto ed il metodo contenente l'elenco delle istruzioni da eseguire. L'implementazione del metodo *checkEventAndContext* in ciascuna classe derivata, permette al piano, utilizzando la beliefs base per recuperare i dati necessari, di valutare la condizione specificata dalla clausola context in linguaggio Agentino; il parametro di ritorno booleano è indicativo della corretta valutazione, e quindi anche della possibilità di poter applicare il piano a seguito del determinato evento occorso. Il secondo metodo invece implementa il corpo (body) del piano vero e proprio: ogni azione definita in linguaggio Agentino viene tradotta in linguaggio C/C++ Wiring, non senza modifiche. Anche se a primo sguardo le sintassi dei due linguaggi, per quanto riguarda il corpo del piano, sono molto simili (a parte l'assenza del ";" finale classico, a significare il fine istruzione) non avviene una traduzione 1:1 delle istruzioni: infatti di ciascuna ne viene effettuato la verifica sintattica, il parsing, e l'albero di derivazione tempo di scrittura del codice Agentino, generando poi l'insieme delle funzioni intervallate da direttive di Wait della libreria Protothreads (come già visto per la classe IntentionsBase). Questo è il punto focale che distingue un'esecuzione Blind da una Reactive di un piano: intervallando così istruzioni di piano con la funzione di sense avremo un sistema pienamente reattivo agli eventi esterni; inoltre in questo modo avremo la possibilità di eseguire più piani concorrentemente, cosa impossibile se un piano avesse avuto il pieno controllo del microcontrollore fino al suo termine. Così facendo, un frammento di codice contenente

$$while(true)\{\ \}$$

Il risultato così ottenuto è un insieme di file, in linguaggio C/C++ Wiring, direttamente compilabili dall'IDE di Arduino: come possiamo vedere dalla fig.4.9) sulla destra in alto compare il file creato in linguaggio Agentino (Counter.agentino) mentre all'interno della directory src-gen sono presenti i file generati dal motore di Model to Model implementato in Xtexd. La finestra che compare sulla destra dell'immagine è invece l'editor del linguaggio, ed i plugin necessari dal sintax highlighting, alla segnalazioni di errori, fino all'autocompletamento ove previsto dalla funzione di cross reference. Così, data la definizione di un agente, in automatico sarà disponibile uno sketch Arduino: aprendo infatti il file Counter.ino mediante l'uso dell'IDE

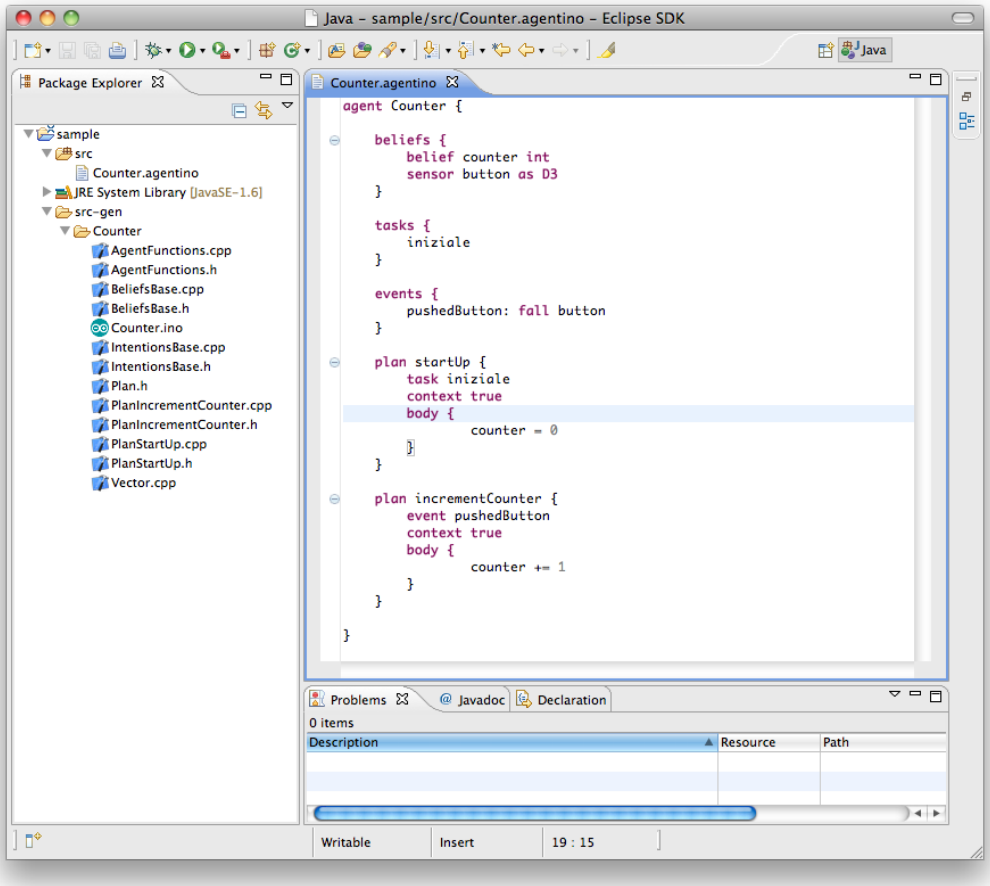

Figura 4.9: Editor Eclipse con plugin Agentino: a destra il File Manager, a destra il codice di esempio di un contatore

Arduino e utilizzando l'apposito tasto upload, lo sviluppatore avrà direttamente a disposizione un agente funzionante sulla propria scheda di prototipazione.

4.7 Test e caso di studio

Durante la fase di sviluppo del generatore Model to Model, gran parte dello sforzo è stato dedicato al testing del codice generato a partire dalle istruzioni del codice del piano. Xtext ed Xtend sono due ottimi strumenti che mi hanno permesso, nel tempo dedicato a questa tesi, di sviluppare agevolmente una gran quantità di soluzioni non triviali da risolvere anche con linguaggi evoluto come, ad esempio Java. La sintassi e semantica sono dunque state testate in un discreto numero di combinazioni, creando di volta in volta risoluzioni a problemi di ordine pratico, oltre che teorico. Il passo successivo è stato quello di progettare e realizzare una piccola dimostrazione pratica che sottolinei la facilità di progettazione e scrittura del codice di un agente, comparandolo poi alla programmazione classica della piattaforma. Cercherò di evidenziare i punti chiave, mostrandone pregi e difetti di ciascuna.

Il modello consiste in un piccolo sistema di automazione, in cui è presente un rullo trasportatore, su cui sono posizionati contenitori che devo essere riempiti da un apposito erogatore ed un sistema a fotocellula per la rilevazione del corretto posizionamento del contenitore in corrispondenza dell'erogatore.

Identifichiamo quindi prima di tutto quali sono i sensori e gli effettori del nostro sistema:

- Rullo trasportatore, comandato da un motore: un effettore digitale, il cui stato può essere acceso o spento.
- Fotocellula, sensore analogico indicante la presenza di un oggetto che interrompe il fascio di luce puntato su di essa.
- Erogatore, analogamente al rullo, un effettore digitale, il cui stato può essere acceso o spento.

In questo modo è già possibile scrivere il corpo dell'agente per quanto riguarda i beliefs:

Listing 4.10: Definizione di un agente tramite il nome

```
agent Demo {

  beliefs {
    effector motore as D3
    sensor fotocellula as A0
    effector erogatore as D4
  }
  ...
```

L'unico evento è identificato dalla variazione del di stato della fotocellula, il cui valore (con range da 0 a 1023) attraverserà una determinata soglia (la quale dipenderà dallo specifico componente hardware utilizzato) quando il contenitore si troverà nella posizione di riempimento.

Listing 4.11: Definizione di un agente tramite il nome

```
tasks {
      init
}
events {
      contenitorePosizionato: falled fotocellula 70
}
...
```

Ora analizziamo la dinamica del funzionamento del sistema:

1. Allo startup del sistema il motore verrà avviato, facendo così girare il nastro trasportatore.

2. Quando il contenitore è posizionato in corrispondenza della fotocellula (e quindi sotto all'erogatore) l'agente disattiverà il motore ed accenderà l'erogatore per un certo tempo T necessario al riempimento del contenitore

3. Trascorso tale tempo l'agente spegnerà l'erogatore e riattiverà il motore, in attesa di un nuovo contenitore in posizione corretta.

Questa può essere una delle tante semantiche applicate al sistema da controllare: nonostante la semplicità ci permette di evidenziare

molti punti a favore del linguaggio Agentino. Vediamo come è possibile tradurre ciascun comportamento in un piano:

Listing 4.12: Definizione di un agente tramite il nome

```
plan Start {
  task init
  context true
  body {
    //Accensione del motore
    motore = 1
  }
}

plan RiempiContenitore {
  event contenitorePosizionato
  context true
  body {
    // Spegnimento del motore
    motore = 0
    // Accensione dell'erogatore
    erogatore = 1
    delay 2000
    // Dopo 2 secondi
    // Spegnimento dell'erogatore
    erogatore = 0
    // Accensione del motore
    motore = 1
  }
}
```

Vediamo ora come sarebbe scritto il codice Arduino per la gestione dello stesso sistema, attraverso un utilizzo della piattaforma come macchina a stati:

Listing 4.13: Definizione di un agente tramite il nome

```
#define pinFotocellula A0
#define pinMotore 3
#define pinErogatore 4
```

```
#define SPOSTAMENTO 0
#define EROGAZIONE 1

int fotocellula;
int stato;
int soglia = 70;
unsigned long tempoInizio;
int tempoErogazione=2000;

void setup() {
    pinMode(pinFotocellula, INPUT);
    pinMode(pinMotore, OUTPUT);
    pinMode(pinErogatore, OUTPUT);

    digitalWrite(pinMotore, HIGH);
    stato = SPOSTAMENTO;
}

void loop() {
    switch(stato) {
      case SPOSTAMENTO:
        fotocellula = analogRead(pinFotocellula);
        if (fotocellula < 70) {
          digitalWrite(pinMotore, LOW);
          digitalWrite(pinErogatore, HIGH);
          tempoInizio=millis();
        stato = EROGAZIONE;
        }
        break;
      case EROGAZIONE:
        if ((millis()-tempoInizio)
                          > tempoErogazione) {
          digitalWrite(pinErogatore, LOW);
      digitalWrite(pinMotore, HIGH);
          tempoInizio=millis();
        stato = SPOSTAMENTO;
```

```
            }
        }
}
```

Come possiamo facilmente notare il livello di astrazione è profondamente diverso: mentre in Arduino dobbiamo preoccuparci direttamente di settare correttamente i pins (in modalità input o output) della lettura e scrittura a basso livello degli stessi, e della gestione del tempo, in Agentino queste operazioni sono effettuate dal framework che ne sta alla base e di cui abbiamo già discusso; inoltre avviene un ragionamento event-driven che semplifica notevolmente lo sviluppo del codice. *L'effetto primario risultante è un codice molto più pulito e leggibile, con conseguente facilità di estensione, manutenzione e debugging.*

Capitolo 5

Conclusioni

Lo sviluppo di questa tesi mi ha portato allo studio e ricerca di un modello, computazionalmente corretto e coerente, per un nuovo paradigma di programmazione orientata agli agenti applicato all'ambito degli embedded systems ed in particolare alla piattaforma Arduino. Tale modello punta ad innalzare il livello di astrazione che lo sviluppatore si trova innanzi al momento di prendere confidenza con la piattaforma in oggetto, non solo a primo impatto, ma anche nel lungo periodo: infatti un linguaggio più evoluto permette di astrarre da dettagli implementativi legati alla specifica piattaforma, concentrando gli sforzi dell'utilizzatore sulla risoluzione del problema. Il codice così scritto risulta maggiormente leggibile, di facile comprensione, rendendo più agevoli le operazioni di debug e test, estendendone così il livello di manutenzione, estendibilità ed efficienza.

La programmazione in linguaggio Agentino, come descritta nel capitolo precedente, è orientata dalla proattività del modello ad agenti a cui fa riferimento: quindi non più oggetti, metodi e flusso di controllo quali nozioni base per la descrizione di una sistema. Ora la persecuzione di un task a lungo termine unita alla reattività verso eventi esterni dimostra la complessità del modello stesso e del lavoro svolto nella progettazione del sistema, attraverso uno strumento altamente personalizzabile e scalabile nel numero di task. Infatti l'estensione di un agente con ulteriori funzionalità mantiene assolutamente inalterato il processo di analisi, progettazione e sviluppo appena completato, aggiungendo solamente la parte richiesta.

Si introduce così il concetto di concorrenza tra task (caratteristica intrinseca di piattaforme estremamente più sofisticate, costose e tipicamente dotate di sistema operativo) anche su sistemi a microcontrollori accessibili a tutti: la capacità interna di gestire molteplici task contemporaneamente semplifica la progettazione in maniera eccezionale, nonché la descrizione finale dell'agente. Inoltre il codice risultante in linguaggio Agentino è totalmente privo di qualsiasi istruzione che gestisca esplicitamente tale concorrenza, rendendo il codice estremamente chiaro e comprensibile anche ad un utente non esperto di programmazione.

Come abbiamo potuto notare dal caso pratico, la facilità con cui si passa dall'analisi del problema alla scrittura del codice, è strabiliante, con una traduzione quasi 1:1 delle nozioni espresse in linguaggio naturale verso costrutti Agentino: i concetti così espressi sono immediati, senza fronzoli aggiunti dal linguaggio C, focalizzando subito l'attenzione sui principi realmente importanti nel comportamento dell'agente. In questo modo si esalta ancora di più l'eleganza del linguaggio nella sua forma sintattica e semantica, portando in modo naturale lo sviluppatore ad una programmazione estremamente semplice e pulita.

Il linguaggio descritto in questo modo, inoltre, può essere oggetto di test sistematici, tra cui anche un'analisi del modello a priori tramite Model Checking, potendo così garantire determinate condizioni di sicurezza ed affidabilità del sistema da controllare: un aspetto non banale quando il software prodotto sarà fisicamente non accessibile e non aggiornabile. Garantire tali condizioni con un'analisi statica a priori significa certificare che il programma è privo di errori semantici, e l'applicazione sarà funzionante (a meno di fallimenti hardware) assicurando l'operatività in qualsiasi condizione.

La netta separazione tra sintassi/semantica e generazione del codice (per la specifica piattaforma) che viene introdotta attraverso l'uso di Xtext/Xtend rende virtualmente possibile il porting del linguaggio su di una qualsiasi piattaforma embedded, implementando il codice dell'agente una sola volta (così come il concetto introdotto a suo tempo da Java "Write once, run anywhere"). In teoria è come se introducesse un nuovo livello di astrazione intermedio, quasi una sorta di interfaccia posta tra il software e l'hardware, senza però appesantirne le performance (come farebbe ad esempio una macchina virtuale) in

quanto il codice generato è direttamente compilabile mediante gli stessi strumenti forniti dai produttori delle piattaforme hardware. Questo grazie al fatto che i generatori di codice possono essere dinamicamente cambiati, ed ottimizzati di volta in volta sulle piattaforme target. In quest'ottica un produttore di sistemi embedded non sarebbe più legato ad una specifica soluzione hardware solo perché "sa programmare quella", ma potrebbe spaziare su tutti i microprocessori per i quali è stato ottenuto il porting del linguaggio, in modo uniforme.

L'inevitabile overhead computazionale introdotto a favore di un nuovo livello di astrazione può essere tuttavia ottimizzato. La gestione a polling delle letture una ad una dei sensori, è pesante ed inefficiente. La soluzione migliore rimane quella che fa riferimento ad interrupt hardware, che però Arduino dispone solo in piccolissima parte. Per quanto riguarda le performance del sistema, non sono stati eseguiti test specifici, in quanto si tratta di una versione del software non del tutto ottimizzata; inoltre sarebbe possibile, a rigor di logica, eliminare del tutto il software legacy che attualmente viene fornito con Arduino (come le librerie per la facilitazione dell'input/output, la gestione dei timer, delle eccezioni, etc.) sostituendolo con codice C/C++ ottimizzato per la sinergia con il generatore di codice. Le maggiori dimensioni dei programmi e le minori performance di esecuzione (rispetto ad un software progettato ad hoc), vengono senza dubbio giustificate per i pregi appena elencati. Tuttavia, per quanto detto finora, possiamo definire le performance come un parametro relativo intrinseco alla scheda e, in quanto tale, decidere in qualsiasi momento di valutare l'adozione di una diversa piattaforma, senza modificare il lavoro svolto e riutilizzando il codice Agentino già prodotto, rigenerando solamente il codice in modo opportuno, in base alla scelta effettuata.

5.0.1 Ottimizzazioni e lavori futuri

Come già anticipato nel paragrafo precedente, questa tesi può essere il primo passo di un lavoro più ampio e che si ramifica in diversi ambiti: dall'estensione del linguaggio, ad un'ottimizzazione del codice generato, fino alla creazione di un insieme di tool di supporto per il debug, l'emulazione e il model checking dell'intero sistema, che per ovvie ragioni di tempo non è stato possibile effettuare.

Lo scopo primario sarebbe quello di fornire un ambiente integrato per lo sviluppo di MAS veri e propri, identificando quindi un protocollo e primitive per lo scambio di informazioni a messaggi, nonché un'apposita casella ove ricevere questi dati: l'arrivo di questi sarebbe trattato alla stregua di un nuovo evento accaduto, abbracciando interamente la filosofia event-driven adottata.

Per quanto riguarda l'estensione del linguaggio si possono introdurre nuove istruzioni utilizzabili nel blocco $< body >$ del piano, quale la chiamata, da parte di un piano "padre" di uno o più task "figli" in diretta concorrenza, o come più semplici esecuzioni di funzioni sequenziali. Risulterebbe utile anche l'importazione di librerie esterne (come la comunicazione con particolari periferiche quali LiquidCrystal, MotorStepper, I2C...) in modo da poterle adattare ed armonizzare alla struttura del linguaggio finora utilizzato.

Xtext inoltre permette funzionalità di validazione avanzata del codice a tempo di scrittura, come ad esempio il type check, oppure la segnalazione di errore nel caso in cui il nome di un belief sia duplicato. Sarebbe dunque consigliabile un maggiore utilizzo di questa feature al fine di evitare possibili errori rilevabili prima della compilazione del codice.

La parte di ottimizzazione potrebbe portare ottimi risultati in fattore performance per quanto riguarda la piattaforma Arduino: in primo luogo bypassando la struttura a funzioni setup - loop, in seguito creando istruzioni ad hoc per la lettura e scrittura di molteplici pin digitali direttamente dal registro del micocontrollore (eliminando così l'overhead introdotto dalle libreria di sistema, come provato da test di programmatori indipendenti [17], infine riducendo al minimo le strutture dati utilizzate.

Inoltre lo sviluppo di un emulatore a livello software sarebbe assolutamente l'applicazione definitiva per il debugging ed il model checking che attualmente manca a questa piattaforma. Emulare il comportamento di un agente, via software, permetterebbe allo sviluppatore un rapido controllo del corretto funzionamento; un sistema integrato di model checking, di non facile applicazione ed efficienza su codice C/C++, porterebbe alla formulazione, in fase di analisi, di regole LTL(Linear Temporal Logic) o CTL (Computational Tree Logic) che il sistema deve rispettare al fine di essere idoneo alle specifiche date.

Bibliografia

[1] Android-ADK. http://developer.android.com/guide/topics/usb/adk.html.

[2] Arduino. http://www.arduino.cc/.

[3] Arduino. http://www.arduino.cc/.

[4] ATMEL. http://www.atmel.com/images/doc2559.pdf.

[5] M. Banzi. *Getting started with Arduino*. Make, 2008.

[6] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, and P. Friese. Xtext user guide. *Dostupné z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html*, 2008.

[7] R. Bordini, J. Hübner, et al. Jason–a java-based agentspeak interpreter used with saci for multi-agent distribution over the net. *On-line at http://jason. sourceforge. net*, 2004.

[8] R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2008.

[9] M. Bratman. Intention, plans, and practical reason. 1987.

[10] A. Dunkels and O. Schmidt. Protothreads-lightweight stackless threads in c. *SICS Research Report*, 2005.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.

[12] S. Efftinge and M. Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[13] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[14] FreeRTOS. http://www.freertos.org/.

[15] S. Heath. *Embedded systems design*. Newnes, 2003.

[16] J. Hübner and J. Sichman. Simple agent communication infrastructure. 2003.

[17] JeeLabs. Pin input output performance, http://jeelabs.org/2010/01/06/pin-io-performance/.

[18] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38, 1998.

[19] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*, volume 25. Springer-Verlag New York Inc, 2011.

[20] E. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[21] E. Lee and S. Seshia. *Introduction to Embedded Systems-A Cyber-Physical Systems Approach*. Lee & Seshia, 2011.

[22] P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer Verlag, 2010.

[23] MBed. http://mbed.org/.

[24] D. Mellis, M. Banzi, D. Cuartielles, and T. Igoe. Arduino: An open electronic prototyping platform. *EA CHI'07*, 2007.

[25] J. Noble. *Programming Interactivity: A Designer's Guide to Processing, Arduino, and OpenFrameworks*. O'Reilly Media, 2009.

[26] QnxOS. http://www.qnx.com/.

[27] A. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. *Agents Breaking Away*, pages 42–55, 1996.

[28] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 1995.

[29] D. Simon. *An embedded software primer*. Addison-Wesley Professional, 1999.

[30] TexasInstruments. Ti's wikipedia website, http://processors.wiki.ti.com/index.php/msp430.

[31] TinyOS. http://www.tinyos.net/.

[32] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152, 1995.