UNIVERSITÀ DEGLI STUDI DI PADOVA
Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

Sviluppo di aggiornamenti per il simulatore didattico NXTSimulator

Relatore: Prof. Moro Michele

Laureando: Nicola Andreose
Matricola: 542551

Anno Accademico 2009-2010

Documento scritto in LaTeX

A Laura per avermi sostenuto,
a mio papà e mia nonna per aver creduto in me,
a mio zio Fabio.

Indice

Capitolo 1

Introduzione

Nel 2008 presso il dipartimento di Ingegneria dell'informazione dell'Università di Padova ha avuto inizio il progetto NXTSimulator, con lo scopo di produrre uno strumento in grado di simulare, con l'ausilio di un ambiente grafico, il comportamento del robot contenuto nel kit LEGO© Mindstorm© NXT[1].

In questo elaborato si descrive come sono stati risolti i diversi bug presenti nel software, in particolare quelli legati alla simulazione dei motori e dei sensori. Nel software erano presenti diversi tipi di errori da quelli logici a quelli runtime disseminati in zone diverse del programma. Sono presentati inoltre le nuove funzioni introdotte e il nuovo manuale utente del simulatore.

1.1 TERECoP

Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods (TERECoP) è un progetto didattico internazionale nel quale è inserito anche il Dipartimento di Ingegneria dell'Informazione di Padova.

Il progetto si pone come obiettivo complessivo quello di sviluppare una struttura di supporto per corsi di formazione degli insegnanti al fine di aiutarli a realizzare attività formative di tipo costruttivista con l'uso della robotica e dar loro la possibilità di divulgare attraverso questa struttura le proprie esperienze. Questo progetto trae ispirazione dalle teorie costruttiviste di Jean Piaget e dalla filosofia didattica di S.Papert.

Piaget sostiene che l'apprendimento non sia tanto il risultato di un passaggio di conoscenze quanto un processo attivo di costruzione della conoscenza

[1] Altri dettagli nei capitoli successivi

basato su esperienze ricavate dal mondo reale e collegate a preconoscenze uniche e personali (Piaget 1972) [11].

Papert invece introduce l'idea che il processo di apprendimento risulta decisamente più efficace quando vengano introdotti artefatti cognitivi, ovvero oggetti e dispositivi che si basino su concetti familiari allo studente. Il Costruzionismo (Papert, 1992) [13] è una naturale estensione del costruttivismo che enfatizza l'aspetto pratico dell'appredimento. In un ambiente cotruzionista gli studenti vengono messi in grado di realizzare da soli oggetti tangibili, significativi e condivisibili. L'obiettivo del costruzionismo è quello di fornire agli studenti buoni strumenti in modo tale che possano imparare facendo meglio di quanto potessero fare prima (Papert, 1980) [14].

La moderna ricerca nel campo dell'educazione scientifica e tecnologica ha reso possibile lo sviluppo di strategie di apprendimento e materiali che vengono incontro ai bisogni degli studenti e alle loro difficoltà di apprendimento, come ambienti didattici su computer e strumenti di laboratorio microcmputerizzati. Più di recente l'attenzione si è concentrata sulla costruzione di modelli informatici. Il Computer-aided modelling nell'ambito dell'apprendimento è considerato un valido strumento per migliorare l'apprendimento e lo sviluppo del pensiero autonomo degli studenti. Tenendo in considerazione che lo studente ottiene una migliore comprensione se si esprime attraverso inventiva e creatività [12], gli insegnanti devono essere in grado di fornirgli l'opportunità di progettare, costruire e programmare i propri modelli cognitivi. Attualmente si ritiene che la programmazione, intesa come un ambito educativo generale per la costruzione di modelli e strumenti, possa sostenere un apprendimento costruzionista lungo lo sviluppo del curriculum scolastico. Il robot della LEGO, in qualche modo derivato dall'esperienza del linguaggio di programmazione LOGO[2] creato da Papert negli anni 60, associa la tecnologia alle idee del costruzionismo.

In quest'ottica il sistema Lego Mindstorms è uno strumento flessibile per l'apprendimento costruzionista offrendo l'opportunità di progettare e costruire strutture robotiche con tempo e fondi limitati. Queste strutture programmabli rendono possibili nuovi tipi di esperimenti scientifici grazie ai quali lo studente può comprendere attraverso l'esperimento pratico i fenomeni fisici della vita di tutti i giorni.

Risulta quindi utile, in questo ambito, l'utilizzo di un simulatore semplificato che permetta di vedere il comportamento del robot senza averlo a portata di mano.

1.2 Mindstorm NXT

Il kit LEGO© Mindstorm© NXT è composto da vari elementi tra questi troviamo:

[2] linguaggio fortemente orientato alla grafica e alla geometria

- Brick NXT, la componente principale del robot.
- 4 differenti sensori, gli altri possono essere acquistati separatamente.
- 3 servomotori.

Il software NXTSimulator simula proprio il comportamento di questi elementi.

1.2.1 Brick NXT

Figura 1.1: Brick NXT

Il Brick NXT rappresenta quello che si può definire il cervello del robot: al suo interno è presente un mico computer, il quale permette l'esecuzione dell'interprete dei comandi e il controllo delle perfiriche ad esso collegate.
Specifiche Tecniche:

- Microcontrollore ARM7 a 32-bit con memorie da 256 Kbytes Flash e da 64 KByte RAM
- Microcontrollore AVR a 8-bit con memorie da 4 Kbytes Flash e 512 Byte RAM
- Bluetooth v2.0 compatibile
- Porta USB 2.0
- 4 porte di input
- 3 porte di output
- Display LCDda 100 x 64 pixel
- Speaker per la riproduzione dei suoni
- Alimentazione: 6 batterie tipo AA

1.2.2 Motori

I motori forniti con il kit sono servomotori, ovvero a differenza dei motori tradizionali possiedono bassa inerzia, linearità di coppia e velocità, rotazione uniforme e capacità a sopportare picchi di potenza.
All'interno del servomotore fornito si trova inoltre un sensore di rotazione che permette di tracciare la posizione dell'asse esterno del motore in gradi o rotazioni complete.

Figura 1.2: Servomotore

1.2.3 Sensori

I sensori permettono al robot di ricevere informazioni riguardanti l'ambiente circostante. Abbiamo diversi tipi di sensori, i quattro principali utilizzati nel simulatore NXTSimulator sono:

il sensore di luce, è in grado di rilevare le variazioni di intensità di luce e restituire il livello di grigio misurato;

Figura 1.3: Sensore di luce

il sensore di tatto, possiede un bottone che può assumere tre diversi stati: premuto, rilasciato e bumped[3];

il sensore di prossimità, o ad ultrasuoni è un dispositivo che comunica con il brick attraverso il protocollo I2C[4], esso riesce ad individuare la presenza di oggetti entro un raggio di 250cm e di restituire la distanza con una incertezza

[3]Premuto e rilasciato velocemente
[4]I2C:Inter Integrated Circuit, protocollo per la comunicazione seriale

Figura 1.4: Sensore di tatto

di circa 3 centrimetri;

Figura 1.5: Sensore ad ultrasuoni

il sensore di suono, ha la capacità di rilevare una pressione sonora fornendo come output al brick l'ampiezza misurata.

Figura 1.6: Sensore di suono

1.3 Strumenti utilizzati

Gli strumenti utilizzati per effettuare gli aggiornamenti del software sono stati molteplici: il linguaggio di programmazione del software è Java per cui è stato utilizzato l'ambiente di sviluppo gratuito NetBeans per apportare le modifiche, la parte di test ha richiesto l'utilizzo di:

- LEGO® MINDSTORMS Edu NXT per produrre file compilati in linguaggio macchina tramite il linguaggio iconografico NXT-G;
- Bricx Command Center per decompilare, nel formato NBC[5], e analizzare il codice macchina compilato con il precedente programma. Questo programma è stato utile anche per compilare i file NXC[6];
- WinMerge per trovare le differenze nei vari listati di codice macchina.

Segue quindi una breve descrizione degli strumenti utilizzati.

1.3.1 Java

Java è un linguaggio di programmazione orientato agli oggetti, derivato dallo Smalltalk (anche se ha una sintassi simile al C++) e creato da James Gosling e altri ingegneri di Sun Microsystems©. La piattaforma di programmazione Java è fondata sul linguaggio stesso, sulla Macchina virtuale Java (JVM) e sulle API Java. Java è un marchio registrato di Sun Microsystems©.

I motivi per cui è stato scelto questo linguaggio per realizzare NXTSimulator sono molteplici: la sua portabilità ovvero può essere utilizzato su diversi sistemi operativi senza dover tradurre il codice sorgente per ognuno e la relativa semplicità del linguaggio rispetto ad altri linguaggi di programmazione ha fatto sì che Java venisse scelto come linguaggio di programmazione per il progetto.
Per lo sviluppo del software è stata adottata la versione 1.6 di Java e l'eseguibile è stato testato sulle piattaforme Linux, Windows e Macintosh senza presentare problemi di compatibilità, confermando le attese iniziali.

1.3.2 NetBeans

NetBeans è un ambiente di sviluppo multi-linguaggio scritto interamente in Java. È l'ambiente scelto dalla Sun Microsystems© come IDE ufficiale, da contrapporre al più diffuso e valido Eclipse. Non vi è stato un motivo particolare per cui si è scelto utilizzare per il progetto NetBeans a Eclipse, se non per la sua tendenza a facilitare lo sviluppo e la gestione di elementi grafici. La versioni utilizzate per lo sviluppo di NXTSimulator sono state diverse ma per questo elaborato si è

[5] NeXT Byte Codes
[6] Not eXactly C

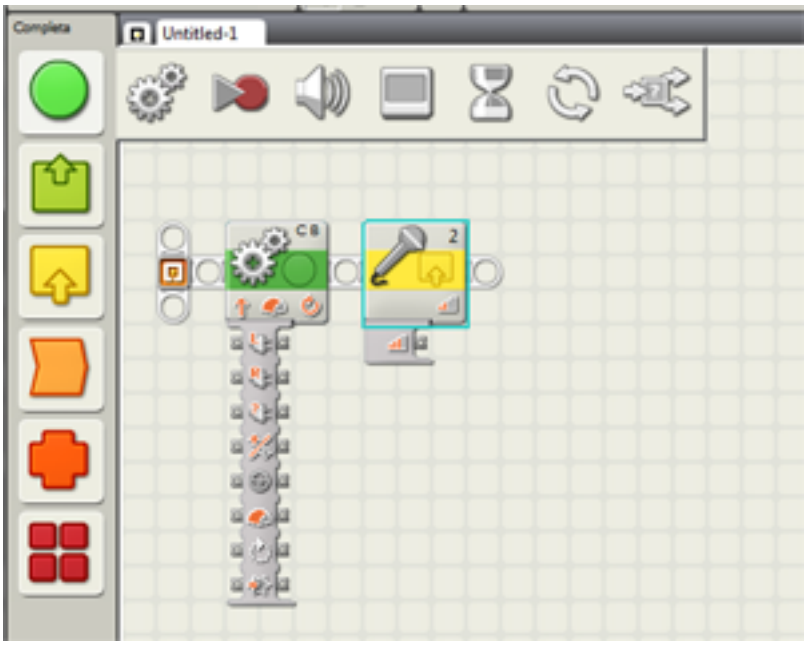

Figura 1.7: Esempio di programmazione NXT-G

utilizzata la sola versione 6.7 anche se nel ciclo di vita del programma è uscita una versione più recente la 6.9. NetBeans si è rilevato un ottimo ambiente di sviluppo soprattuto nella fase di debugging, infatti permette di modificare il codice durante l'esecuzione del programma e di vederne gli effetti, permette inoltre di controllare le variabili e i loro valori in corso di esecuzione utilizzando la tecnica JIT[7].

1.3.3 NXT-G

NXT-G è un linguaggio iconico per programmare il robot LEGO Mindstorm NXT basato sulla tecnologia LabVIEW della National Intruments, un ambiente di sviluppo grafico riadattato per rispettare le specifiche del firmware NXT. Ulteriori dettagli vengono forniti nel capitolo seguente.

1.3.4 NBC

NBC, che sta per NeXT Byte Code, è un semplice linguaggio con una sintassi molto simile a quella assembly utilizzato per programmare il brick LEGO Mindstorm NXT. Questo linguaggio ha però delle restrizioni, che non gli permettono di essere definito come un linguaggio general-purose assembly, legate alle limitazioni nel Byte-code dell'interprete LEGO.
Questo linguaggio ha permesso di analizzare il codice a basso livello e di individuare e studiare i meccanismi con cui lavora il firmware LEGO NXT.

[7] Just In Time

Codice 1.1: Esempio di codice NBC.

```
thread t000
sub      ub002A, ub0028, ub0029
not      ub002B, ub0026
mov      sw0021, ub002B
mov      uw0022, sw0021
acquire  m0004
mov      sl0003, sl0002
subcall  t001, sl0005
mov      sl0002, sl0007
release  m0004
tst      GT, ub002D, sl0002
acquire  m0009
mov      ub002E, ub0025
mov      sl0008, sl0001
mov      ub002F, ub002A
subcall  t002, sl000A
[..]
```

1.3.5 NXC

Not eXactly C è un linguaggio di programmazione simile al C come sintassi ma sviluppato solo per la creazione di file per Mindstorm NXT. Questo linguaggio ad alto livello sfrutta il compilatore NBC per produrre file RXE[8] eseguibili dal robot lego. Essendo un linguaggio Open Source vi è la possibilità di vedere, sotto forma di testo, le istruzioni del codice binario corrispondente al codice NXT.

Codice 1.2: Esempio di codice NXC.

```
task main(){
SetSensorTouch(IN_1);
SetSensorSound(IN_2);
SetSensorLight(IN_3);
SetSensorLowspeed(IN_4);
until(SensorUS(IN_4)<20)
Wait(10);
OnFwdSync(OUT_AC,75,0);
until(SENSOR_1 == 1 );
Off(OUT_AC);
until(SENSOR_2 > 50);
OnRev(OUT_B,100);
```

[8] RXE: File contentente codice binario eseguibile dal Brick NXT

```
        Wait(500);
        Off(OUT_B);
        RotateMotorEx(OUT_AC,100,-180,0,true,true);
        RotateMotorEx(OUT_AC,100,370,100,true,true);
        OnFwdSync(OUT_AC,75,0);
        until((SENSOR_3<44)||(SENSOR_2>54));
        Off(OUT_AC);
        OnFwd(OUT_B,100);
        Wait(500);
        Off(OUT_B);
}
```

1.3.6 Bricx Command Center

Bricx Command Center, versione 3.3, è un IDE per Sistemi Operativi Windows utilizzato per programmare anche il brick LEGO Mindstorm NXT usando i linguaggi NXC e NBC, ma supporta molti altri linguaggi. Per il progetto è stato molto utile, infatti aprendo un file compilato RXE lo converte in automatico nel formato NBC rendendone agevole la comprensione e lo studio del programma.

1.3.7 WinMerge

WinMerge è uno strumento open source che permette tra le varie funzioni di evidenziare le differenze lessicali tra due file. Questo programma è stato molto utile nella fase di debugging dei file RXE e NBC, infatti permetteva di individuare immediatamente quali istruzioni e/o dati cambiavano nelle diverse compilazioni. La versione utilizzata è la 2.12.

Capitolo 2

Il linguaggio NXT-G e il firmware

2.1 Il linguaggio NXT-G

Come accennato nel capitolo precedente NXT-G è un linguaggio grafico per programmare il robot LEGO Mindstorm NXT. L'ambiente di sviluppo grafico permette la programmazione definita come G-Language[1], caratterizzata dalla assenza di codice scritto sotto forma di testo. La definizione degli algoritmi del programma avviene tramite icone ed oggetti grafici, blocchi, mentre lo scambio di informazioni tra i blocchi avviene tramite delle linee che li collegano.

2.1.1 L'ambiente di sviluppo Mindstorm Educational NXT

L'interfaccia grafica di Mindstorm Educational NXT risulta essere molto semplice ed intuitiva. Possiamo individuare 4 aree fondamentali: l'area centrale di programmazione dove si dispongono e uniscono i blocchi tra di loro (contrassegnata con la lettera B in figura 2.1), l'area a sinistra contenente la barra delle funzioni in cui sono raggruppati i blocchi (contrassegnata con la lettera A in figura 2.1) in menù a scorrimento (contrassegnato con la lettera D in figura 2.1), ed infine vi è la zona in basso che mostra le caratteristiche del blocco evidenziato in quel momento (contrassegnata con la lettera C in figura 2.1).

2.1.2 I blocchi

Il software Mindstorm Edu. NXT offre circa 40 blocchi base dalle caratteristiche diverse, raggruppati in 6 categorie principali:

- Blocchi di azione, comprende motori, display e speaker.

[1] Graphic Language

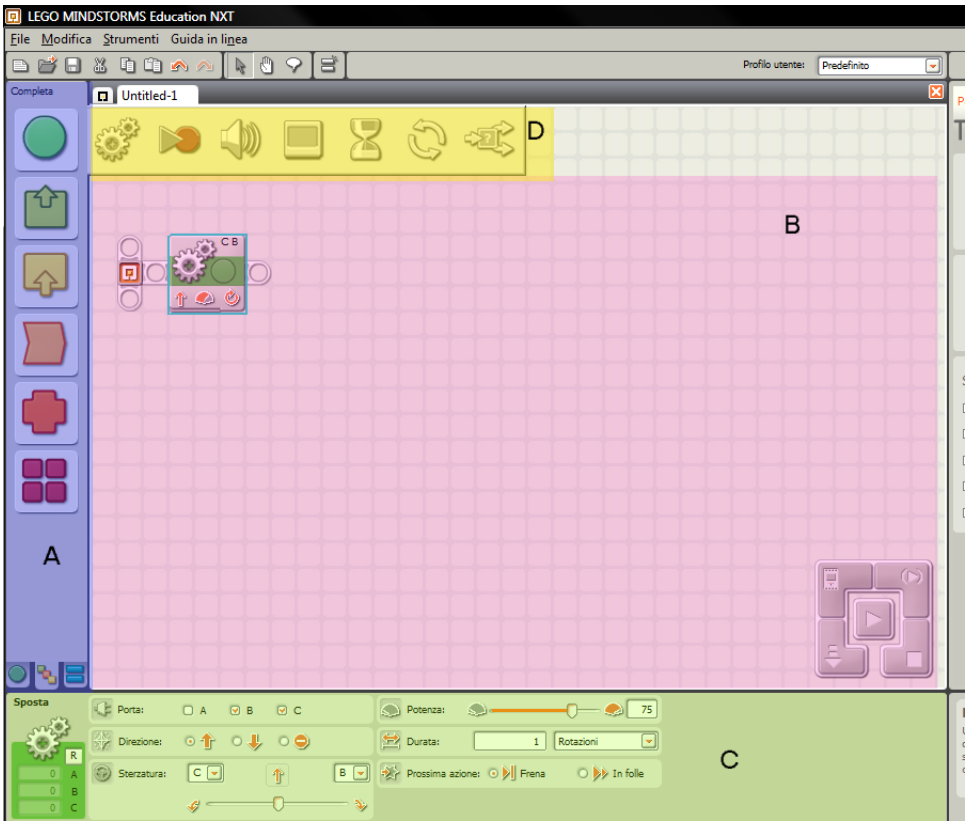

Figura 2.1: L'ambiente di sviluppo diviso in zone

- Blocchi di sensori, come sensori di luce, ad ultrasuoni e tatto.
- Blocchi di flusso, ovvero blocchi per gestire il flusso del programma con cicli, attese e arresti.
- Blocchi di dati, come blocchi di comparazione, funzioni matematiche e logiche.
- Blocchi delle funzioni avanzate, blocchi particolari per accedere a file o resettare per resettare un motore.
- Blocchi personalizzati, blocchi realizzati dall'utente.

Ogni blocco ha delle specifiche caratteristiche che vengono definite al momento della sua selezione; ad esempio, per il blocco motore, si possono definire: la porta in cui si trova collegato, la direzione, il tipo di azione che deve svolgere(se mantenere una potenza costante o ridurla/aumentarla gradualmente), la potenza di partenza, il controllo di potenza, la durata (a scelta tra gradi, secondi, rotazioni o tempo illimitato), l'eventuale attesa di completamento prima di eseguire un nuovo blocco e infine l'azione successiva da intraprendere al completamento.

Figura 2.2: Le caratteristiche del blocco Motore

2.1.3 Il diagramma di flusso

Il diagramma di flusso indica la sequenza di esecuzione dei blocchi di programmazione, solo i blocchi ad esso collegati vengono trattati, gli altri invece sono ignorati.
NXT-G permette di eseguire compiti simultanei, aggiungendo altri diagrammi di flusso a quello principale: ad esempio, se vogliamo che il nostro robot si muova in una certa direzione e nel frattempo vogliamo controllare l'azione di un braccio, non bisogna fare altro che aggiungere un secondo flusso a quello principale in cui inserire i blocchi per la gestione del braccio (un esempio in figura 2.3). Un flusso si può cominciare un nuovo flusso sia all'inizio della sequenza che all'interno.
Per creare un diagramma di flusso parallelo si sposta il puntatore del mouse sul

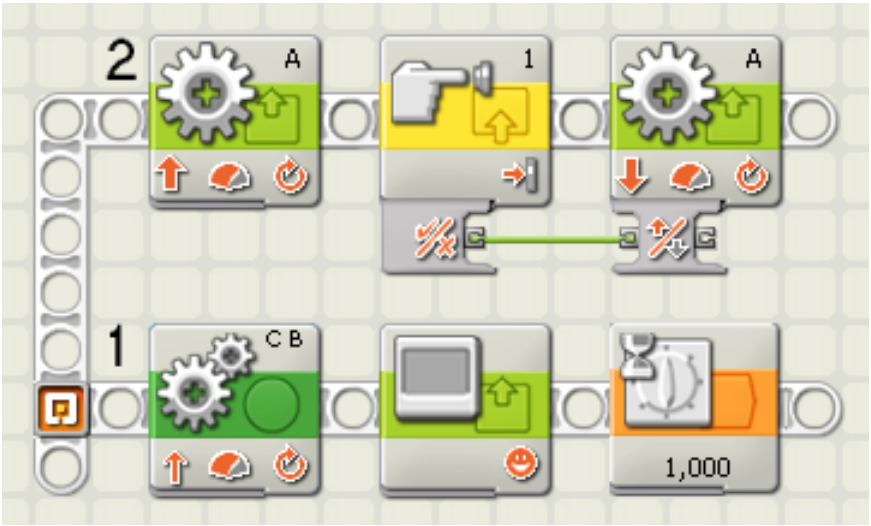

Figura 2.3: Esempio di programma NXT-G con 2 diagrammi di flusso

punto di partenza e si preme e si mantiene premuto il pulsante del mouse mentre si sposta il mouse in alto o in basso. In questo modo si crea un nuovo diagramma di flusso a cui connettere i blocchi di programmazione. Per creare un nuovo flusso all'interno invece basta premere shift mentre si clicca e si sposta il mouse.

All'esecuzione del file compilato i blocchi di entrambi i diagrammi saranno eseguiti in parallelo.

2.1.4 La compilazione

NXT-G permette di produrre il file eseguibile caricandolo direttamente nel robot collegato tramite cavo USB o tramite Bluetooth, ma non permette di default salvarlo nel computer in uso. Per poter salvare gli eseguibili nel computer si è fatto affidamento ad un plug-in[2] di terze parti con il quale è stato possibile controllare il codice eseguibile prodotto dal compilatore NXT-G, cliccando su *Download to File* dal menù *Strumenti*, senza doverlo caricare e poi recuperare dal robot.

2.1.5 Conclusioni e problematiche del linguaggio

La semplicità della programmazione in NXT-G comporta degli svantaggi, infatti un piccolo programma richiede, una strutturazione medio-alta e all'aumentare della complessità del programma si incontrano difficoltà nella gestione dell'ambiente, che diventa caotico, e nella strutturazione del programma stesso. Inoltre realizzando lo stesso programma in linguaggio NXC si è visto come la dimensione del file compilato RXE fosse di molto minore a quella del codice eseguibile prodotto da NXT-G; le righe di codice per un eseguibile compilato in NXT-G si aggirano mediamente intorno alle 450 mentre lo stesso programma compilato in NXC produce un eseguibile con 70-80 linee di codice. Tuttavia NXT-G ha permesso di individuare molti Bug presenti nel simulatore grazie alla maggiore complessità del codice prodotto.

2.2 Il Firmware

In questa sezione verrà descritto come il firmware v. 1.05 della VM[3] gestisce il file eseguibile compilato dal software LEGO© Mindstorm© NXT versione 1.5 durante l'esecuzione del programma e verrà fornita una breve descrizione di come è strutturato il file eseguibile.

2.2.1 Struttura ed esecuzione di un programma NXT

Il firmware NXT è composto da alcuni moduli tra cui quelli per gestire i sensori, i motori e la virtual machine. I file eseguibili sono per convenzione definiti dalla estensione .RXE e contengono tutte le informazioni necessarie per eseguire il programma, ovvero ogni file .RXE rappresenta un programma.

[2]File: DownloadToFile.llb liberamente scaricabile dal sito http://nxtasy.org
[3]VM: Virtual Machine

Si può dividere logicamente la struttura del programma .RXE in 3 parti: le istruzioni, le informazioni di schedulazione e i dati run-time. Ogni file .RXE contiene queste 3 componenti.

Quando la VM deve eseguire un programma, essa legge il file compilato .RXE dalla memoria flash e inizializza un banco di 32KB di RAM per il programma. Il file .RXE specifica la struttura e i valori di questo banco. Dopo l'inizializzazione il programma viene considerato nello stato *active*, o pronto. Le istruzioni contenute nel programma modificheranno ed utilizzeranno per azioni di I/O i valori delle variabili contenute in questo spazio riservato di RAM. Nella

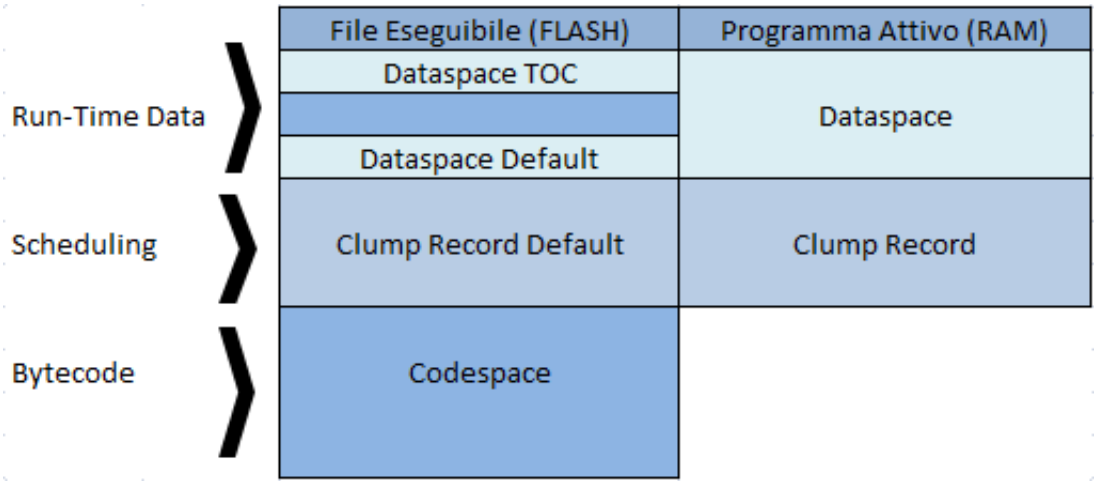

Figura 2.4: Divisione logica del programma

figura 2.4 si vede come la divisione logica del programma in 3 parti si può ulteriormente dividere e come queste sono disposte nelle 2 memorie quando il programma è attivo. Da notare in particolare che solo le istruzioni, il *codespace*, rimangono nella memoria non volatile Flash, questo perché il bytecode non cambia durante l'esecuzione, mentre le altre 2 componenti hanno una parte che rimane invariata residente nella memoria flash e una parte che può subire cambiamenti, residente nella memoria RAM.

2.2.2 Le Istruzioni

Le istruzioni costituiscono la parte principale del programma. La VM interpreta le istruzioni, le esegue operando sui dati presenti nella RAM e gestendo le operazioni di input/output.
La VM supporta le seguenti sei classi di operazioni:

- Matematiche: istruzioni come somma e sottrazione;
- Logiche: operazioni logiche del tipo and, or e not;

- Confronto: istruzioni per comparare i valori all'interno della RAM e che restituiscono un valore booleano come risultato;
- Manipolazione dati: operazioni che copiano, convertono, spostano i dati;
- Controllo: istruzioni che hanno effetto sulla schedulazione oppure che eseguono salti ad una nuova istruzione;
- Input/Output: istruzioni che permettono l'iterazione con le periferiche del brick;

2.2.3 Schedulazione delle Istruzioni

Il bytecode delle istruzioni è organizzato in uno o più pezzi di codice, questi pezzi vengono chiamati *Clump* e sono usati come subroutine per il multitasking. Ogni Clump non è altro che una sequenza di una o più istruzioni, che andranno eseguite, a meno di salti, una di seguito all'altra.
La VM decide come schedulare i vari Clump run-time; un Clump può essere richiamato più volte, oppure non può essere chiamato se prima non sono stati eseguiti altri Clump. I Clump infatti si dividono in due categorie: i Clump dipendenti e quelli indipendenti. Il tipo e l'eventuale dipendenza da altri Clump sono decisi sulla base delle informazioni contenute all'interno del file eseguibile, sotto forma di record esse descrivono ogni singolo Clump. Il record contiene informazioni del tipo: quale porzione del bytecode è assegnata ad un Clump, ordine con cui devono essere eseguiti i Clump e gli eventuali Clump dipendenti che devono essere eseguiti dopo il corrente.
I Clump dipendenti necessitano di essere eseguiti in ordine per garantire di avere, ad esempio, i dati aggiornati ed evitare di elaborare dati obsoleti. Per questo motivo, come si vede dalla figura 2.4, vi è una porzione di RAM riservata ai Clump record in modo che la VM tenga traccia di quali Clump sono stati eseguiti e di quelli da lanciare.

2.2.4 Le fasi di esecuzione di un programma

Quando un utente lancia l'esecuzione di un programma, la VM esegue le seguenti 4 fasi:

- Validazione: legge il file eseguibile e verifica che la versione e le intestazioni siano corrette;
- Attivazione: alloca e inizializza le varie strutture dati nella RAM. Al termine di questa fase il programma è pronto per l'esecuzione;
- Esecuzione: interpreta le istruzione e le esegue nell'ordine definito dal Clump record. Questa fase termina o con l'esecuzione di tutti i Clump o con la terminazione da parte dell'utente.

- Disattivazione: reinizializza tutti i sistemi input/output e rilascia l'accesso al programma.

2.2.5 Dataspace

Durante l'esecuzione del programma la VM usa uno spazio riservato della memoria RAM per salvare i dati del programma. Questo spazio contiene una parte destinata ai dati dell'utente chiamata *dataspace*. Il dataspace è organizzato in una collezione di tipi di dato.Ad ogni dato corrisponde una entry nella *dataspace table of contents* (DSTOC) che tiene traccia del tipo e della posizione di tutti i dati del dataspace della RAM. In questo modo le istruzioni si possono riferire ai dati sparsi in modo casuale nella RAM attraverso la DSTOC. Da notare, come disegnato nella figura 2.4, che la DSTOC risiede nella memoria flash, questo perché durante l'esecuzione i suoi valori non cambiano.

2.2.6 Tipi di dato

Il firmware NXT 1.03 supporta i seguenti tipi di dato:

- Interi: con segno o senza segno, di lunghezza 8, 16 o 32-bit;
- Array: di 0 o più elementi dello stesso tipo;
- Cluster: è una collezione di dati di tipo diverso, può contenere interi o array;
- Mutex record: è una struttura di dati a 32-bit usata per controllare l'accesso alle risorse condivise;

Array e cluster vengono chiamati anche tipi di dato aggregati, essi infatti contengono una aggregazione di uno o più sottotipi di dato. Possiamo avere inoltre array di array, array di cluster, o cluster contenenti array.
I tipi di dato booleani comunemente usati in programmazione non sono propriamente definiti: infatti per utilizzarli si usa un byte senza segno e per convenzione si assegna il valore 0 per indicare il valore *False* e si assegna 1 per indicare il valore *True*.
Le stringhe di testo invece sono rappresentate da array di byte senza segno. Il valore 0 alla fine dell'array indica la terminazione della stringa.
I numeri in *floating point* o frazionari non sono supportati dalla versione firmware 1.03 utilizzata.

2.2.7 Dati Statici e Dati Dinamici

Tutti i tipi di dato, compresi gli array, sono completamente specificati in fase di compilazione. Alla attivazione del programma, la VM inizializza tutti i dati con

i loro rispettivi valori di default, che sono contenuti nel file .RXE. Il programma può modificare tali valori solo run-time.
Abbiamo quindi due tipi di dati, quelli statici e quelli dinamici. I dati statici non possono essere spostati o ridimensionati dalla VM durante la fase di esecuzione e tutti i dati eccetto gli array sono considerati statici. Durante l'esecuzione la VM può muovere, ridurre o aumentare le dimensioni di un array se questo non comporta una perdita di informazione. L'unica limitazione a queste operazioni è data dalla disponibilità di memoria RAM del brick.

I dati statici e dinamici sono memorizzati in due sotto-porzioni diverse all'interno dello spazio di memoria dei dati. La porzione di dati statici è sempre situata nella parte di indirizzi più bassa della memoria rispetto ai dati dinamici, in modo che le due porzioni di dati non si sovrappongano mai.
Questa divisione inoltre separa i compiti del compilatore NXT e dell'esecutore del brick NXT: il compilatore si occupa di specificare i tipi di dato e la loro locazione iniziale nella RAM, o il loro offset, il sistema di esecuzione invece è responsabile della gestione delle locazioni dei dati dinamici durante l'esecuzione del programma. Indipendentemente dal tipo di dato, il compilatore e l'unità di esecuzione si combinano per preservare l'allineamento degli indirizzi, in modo da far iniziare i dati ad indirizzi multipli di 4.

La memorizzazione e la gestione dei dati dinamici, o array, è diversa da quella dei dati statici. Il compilatore si occupa di specificare i record della DSTOC e i valori di default di tutti i dati dinamici, decodificando i dati presenti all'interno dell'eseguibile. Quando il programma passa alla fase di attivazione, il firmware usa un componente aggiuntivo, chiamato *memory manager* (gestore della memoria), che si occupa della gestione dei dati dinamici.

Il memory manager usa uno schema di allocazione per tracciare e ridimensionare gli array all'interno della porzione di memoria dedicata ai dati dinamici. Dopo che i dati statici sono stati posizionati nella memoria la VM riserva i 32KB restanti per i dati dinamici. In questo modo, dopo la fase di attivazione e durante quella di esecuzione, il programma può muovere o ridimensionare gli array a proprio piacimento. Questa operazione di ridimensione e spostamento viene eseguita dal memory manager con l'ausilio di una struttura dati chiamata *dope vector*.
Un dope vector è una struttura che descrive un array all'interno della memoria RAM. Ogni array nella porzione della memoria dinamica è associato ad un dope vector. I dope vector non sono array, ma dei descrittori di dimensione fissa composti da 5 campi. La tabella 2.1 descrive questi campi. Poiché la dimensione e la posizione degli array può variare durante l'esecuzione del programma, il DV associato deve essere in grado di cambiare. Per questo l'insieme dei DV è memorizzato in una speciale struttura di memoria chiamata dope vector array (DVA), la quale cambia dimensione durante l'esecuzione.

Campo	**Descrizone**
Offset	Offset dall'inizio dell'array dati all'interno della RAM
Grandezza elemento	La dimensione, in byte, del singolo elemento dell'array
Numero di elementi	Il numero di elementi contenuti al momento nell'array
Puntatore di coda	Campo non utilizzato nella versione 1.03 del firmware
Link index	Indice del prossimo dope vector nella linked list del memory manager

Tabella 2.1: Struttura di un Dope Vector

La DVA è memorizzata nella stessa porzione di memoria degli array, ed ha le seguenti proprietà:

- Il DVA è un singolo oggetto, e la memoria ne contiene uno ed uno soltanto. Questo DVA contiene le informazioni su ogni DV della memoria;
- La prima entry nel DVA, all'indice 0, è un DV che descrive il DVA stesso;
- Non vi sono istruzioni che permettono la modifica del DVA, poiché è usato unicamente per la manipolazione della memoria;
- Il memory manager tratta il DVA come una linked list grazie all'ausilio del campo link index del DV.

Capitolo 3

Manuale utente del simulatore

3.1 Premessa

Il simulatore NXTSimulator è stato creato con lo scopo di fornire uno strumento che dia la possibilità di studiare la risposta del robot ad un programma senza averlo a portata di mano, affinché utenti e sviluppatori dell'ambiente Mindstorm© NXT ne traggano vantaggio nella fase di messa a punto e prima del collaudo nel robot fisico.
NXTSimulator è stato realizzato con una interfaccia bi-dimensionale, tramite la visualizzazione di immagini e controlli che rappresentano i motori e i sensori del brick, lasciando una eventuale realizzazione in un ambiente grafico 3D a sviluppi futuri.
Il tipo di simulazione è del tipo azione-reazione, ovvero il programma esegue le azioni e mostra gli effetti in tempo reale, senza però rinunciare ad una interazione con lo stesso da parte dell'utente. Infatti durante l'esecuzione l'utente può interagire variando i valori dei sensori manualmente. I sensori sono sensori virtuali e non sono fisicamente collegati al computer dell'utente.

3.2 Il pacchetto software

Il pacchetto software, composto da diversi file, non necessita di una installazione per poterlo utilizzare: infatti basta copiare tutti i file forniti in una cartella a scelta e lanciare l'eseguibile NXTSimulator.exe se si usa Windows oppure NXTSimulator.out se si usa Linux. All'interno del pacchetto troviamo le seguenti cartelle e file:

- lib: cartella contente le librerie necessarie al funzionamento del software;

- src: cartella contente le risorse, quali immagini, necessarie al funzionamento del software;
- pref.properties: file contenente le impostazioni quali la lingua del software;
- NXTSimulator.jar: archivio compresso contenente le classi del simulatore;
- NXTSimulator.exe: file eseguibile per avviare il software;
- NXTSimulator.out: file per avviare il simulatore sotto sistemi operativi Linux;
- NXTSimulator.bat: file batch per avviare il software sotto Windows ed osservare gli output di sistema nella finestra di commando;
- NXTSimulator.sh: script per sistemi Linux per avviare il simulatore e analizzare gli output di sistema nella Shell.

Dopo l'avvio del programma di simulazione e con il suo utilizzo vengono creati ulteriori file all'interno della cartella dove risiede il software; questi file servono per la manutenzione e per tenere traccia delle configurazioni salvate.
I requisiti dell'applicazione sono i seguenti:

- sistema operativo Windows XP o superiore, Mac OS X 10.5, Linux;
- risoluzione schermo 1024 per 768 pixel o superiore;
- Java Runtime Envoironment 6[1] o superiore.

Se il software non si avvia con la lingua desiderata è neccessario modificare l'opzione che si trova nel menù "Lingua"o "Language".

3.3 Guida all'utilizzo di NXTSimulator

3.3.1 L'interfaccia

Le parti principali dell'interfaccia grafica del simulatore sono:

1. menù contestuali;
2. pannello dei motori;
3. pannello dei sensori;
4. TimeLine, per modificare la scala temporale con cui viene eseguita la simulazione;

[1] http://java.sun.com/javase/downloads/index.jsp

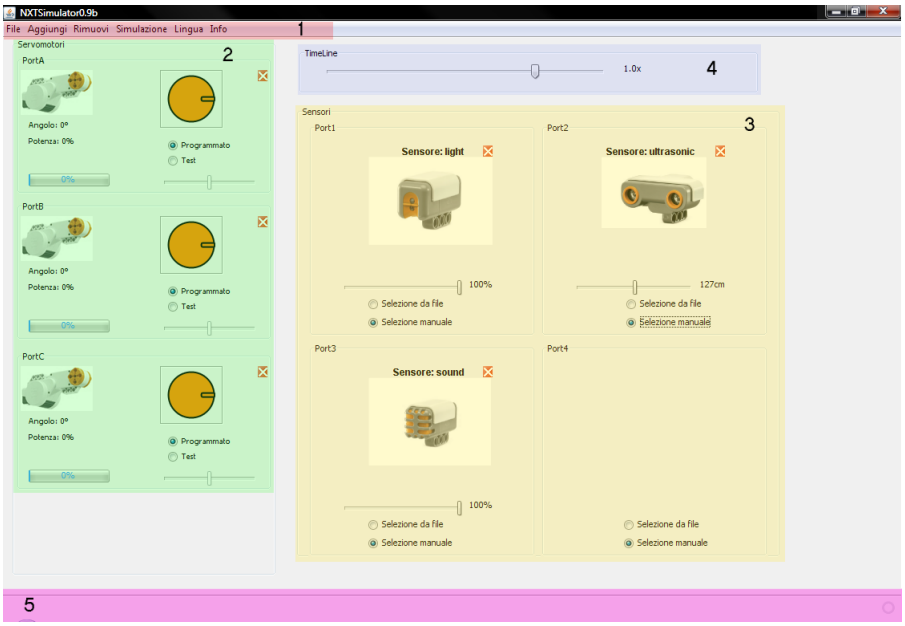

Figura 3.1: Finestra principale di NXTSimulator

5. zona stato di funzionamento del simulatore, dove vengono dati messaggi di servizio, come ad esempio: Caricare un file .RXE, simulazione avviata, simulazione terminata, ecc. .

Menù e ambiente di lavoro

L'accesso ai menù avviene tramite la barra posizionata nella parte superiore della finestra; l'uso dei menù è altamente intuitivo, anche perché ogni voce individua esattamente la sua funzione. Dal menù *File*, mostrato in figura 3.1, è possibile:

- caricare il file .RXE da utilizzare attraverso l'apertura di una finestra per la navigazione e ricerca del file all'interno del proprio file system;
- salvare la configurazione: ovvero salvare, in un file, l'attuale configurazione dei pannelli attivi nel simulatore in quell'istante; le informazioni utili al recupero della posizione dei motori e dei sensori vengono memorizzate in un file, il cui nome è definito dall'utente al momento, e ne permettono il successivo recupero;
- caricare la configurazione: permette di scegliere una configurazione precedentemente salvata, ed ottenere i motori e i sensori nelle stesse posizioni;
- visualizzare l'outputshell: questa funzione è utile solo agli sviluppatori, viene, infatti, mostrata una finestra con l'output dell'interprete dei comandi;
- uscire dal programma.

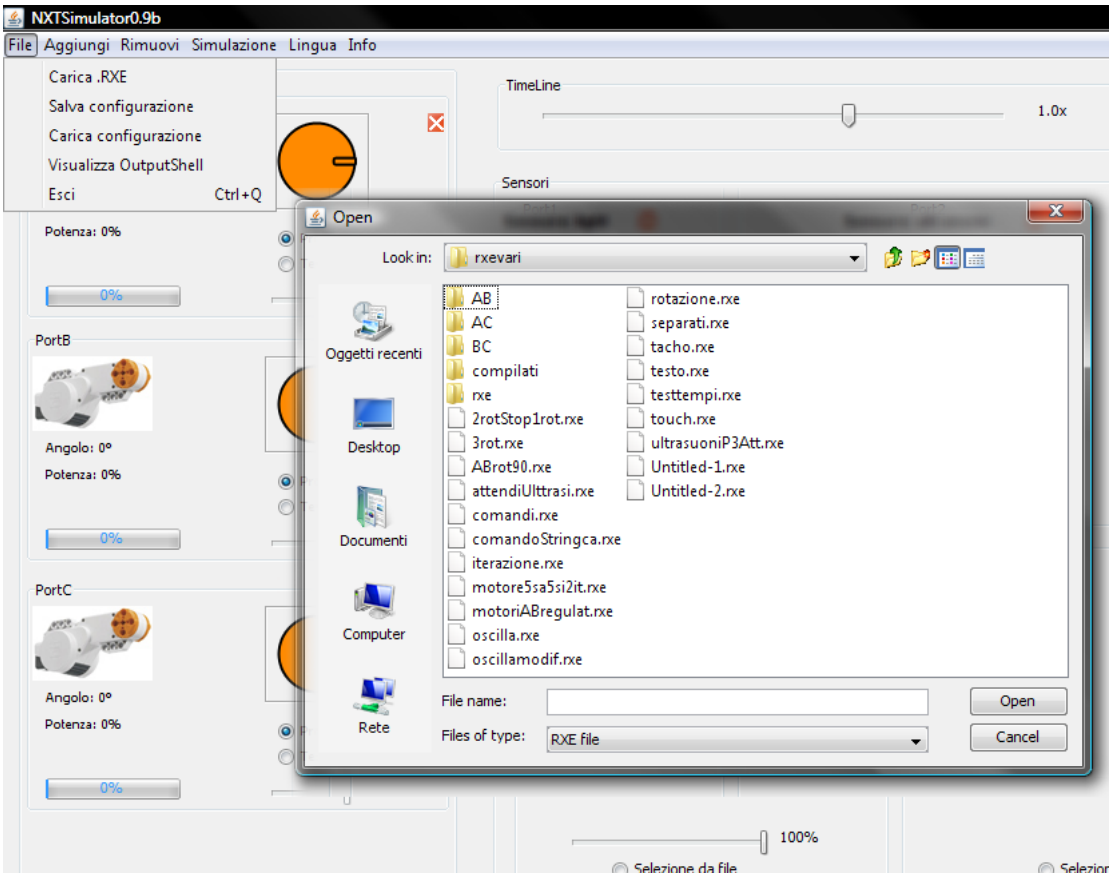

Figura 3.2: Menù File e finestra per la scelta del file RXE

Il menù *Aggiungi* è diviso in 2 parti (vedi figure 3.3 e 3.4), sensori e servomotori. Essendo i sensori di vario tipo si apre un ulteriore menù che permette di selezionarne il tipo. Cliccando sul sensore desiderato si apre una nuova finestra, figura 3.5, che permette di selezionare in quale porta "installare"il sensore.
Per i servomotori invece, essendo di un solo tipo, la selezione della porta sul quale collegarli avviene dal menù a cascata.

Tutti i componenti inseribili sono anche rimovibili sia tramite il menù *Rimuovi* (figura 3.6), in cui è possibile scegliere tra servomotori, sensori ed entrambi, sia tramite l'apposito bottone rosso presente in ogni pannello con un componente attivo.

Tramite il menù *Simulazione* (figura 3.7) è invece possibile far iniziare la simulazione oppure, se questa è già in esecuzione, terminarla. Le opzioni sono attivate/disattivate a seconda dello stato della simulazione.

Il menù Lingua, figura 3.8, permette di selezionare quale localizzazione utilizzare per l'uso dell'interfaccia.

Note sulla versione del programma e sugli autori sono raggiungibili invece tramite il menù *Info*.

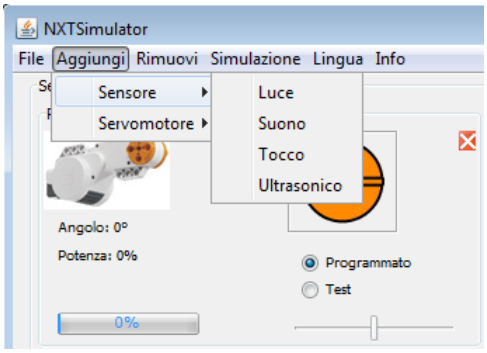

Figura 3.3: Menù Aggiungi Sensore

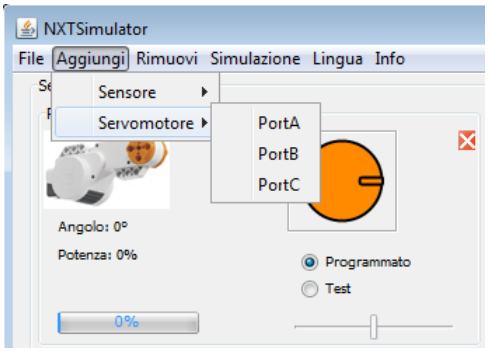

Figura 3.4: Menù Aggiungi Servomotore

Motori e Sensori

Ogni pannello motore presenta due informazioni principali:

- angolo: in cui vi è scritta la posizione in gradi del motore rispetto alla posizione di inizio simulazione;
- potenza: ovvero la potenza percentuale applicata a quel motore in quell'istante. Un valore negativo significa che la rotazione del motore sta avvenendo al contrario (il motore indietreggia invece di avanzare).

Il panello sensore invece presenta la possibilità di scegliere se muovere lo "slider" manualmente o inserire una sequenza di valori nel tempo tramite file. Se si sceglie la selezione manuale, durante l'esecuzione della simulazione, si potrà

Figura 3.5: Finestra per la selezione della porta

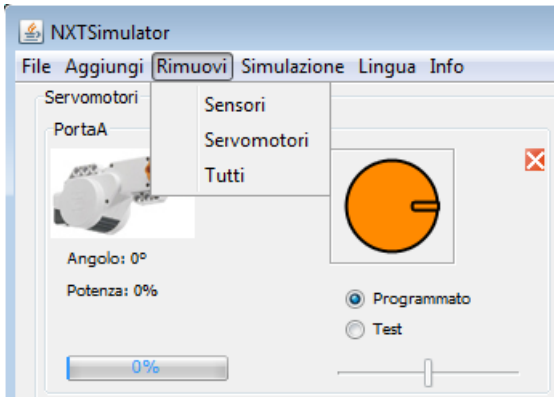

Figura 3.6: Menù Rimuovi

variare il valore cliccando sullo slider, interagendo con l'avanzamento della simulazione stessa. Per il sensore di tatto invece non è presente lo slider ma sono presenti due pulsanti Bump e Switch, il primo simula un tocco e rilascio veloce sul sensore, il secondo invece permette di passare ad uno stato "premuto"che viene mantenuto fino a quando non lo si preme ancora il pulsante.

3.3.2 Guida all'uso

Vedremo ora una normale sequenza d'uso del simulatore, ma prima è bene specificare un importante accorgimento: bisogna fare attenzione a impostare correttamente le porte ed i tipi di sensori utilizzati dalla simulazione, onde evitare

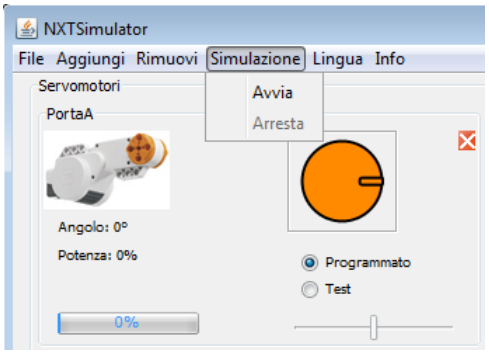

Figura 3.7: Menù Simulazione

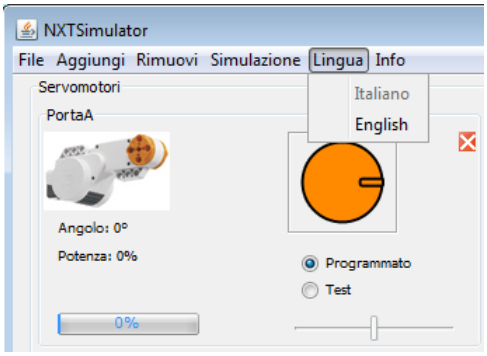

Figura 3.8: Menù Lingua

anomalie di funzionamento.
Il simulatore infatti non è in grado di determinare a priori quali porte e che tipo di sensori verranno utilizzati durante la simulazione poiché la configurazione degli stessi avviene durante l'elaborazione del file .RXE (run-time). Questo problema si verificherebbe comunque anche nel robot reale se si collegassero i sensori alle porte sbagliate.

Per eseguire una simulazione si devono seguire i seguenti passi:

1. tramite il menù File selezionare Carica .RXE, e scegliere il file precedentemente compilato da caricare;

2. inserire nelle porte corrette i motori e i sensori tramite il menù Aggiungi o

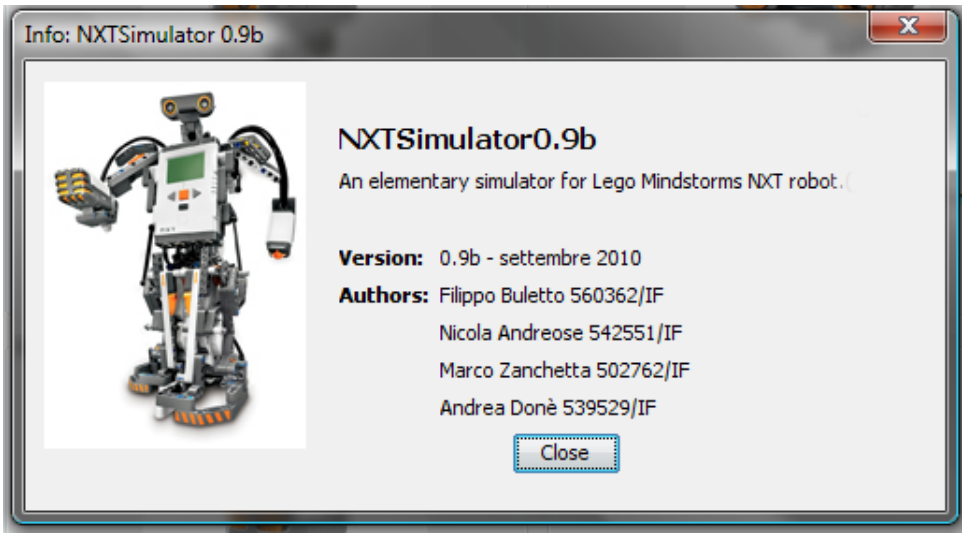

Figura 3.9: Menù Info

caricarne una precedentemente salvata;

3. salvare, eventualmente, la configurazione in un file per poterla usare in futuro;

4. se vi sono dei parametri da impostare prima, come la distanza di partenza da un oggetto del sensore ultrasonico, muovere lo slider del sensore al valore voluto;

5. avviare quindi la simulazione selezionando Avvia tramite il menù Simulazione;

6. attendere il termine della simulazione o, se si desidera terminarla prima, andare nel menù Simulazione e selezionare Arresta;

7. a simulazione terminata è possibile uscire dal programma, ripetere la simulazione ricominciando dal punto 3, oppure se si intende cambiare file .RXE andare nel menù Rimuovi e selezionare Tutti e ricominciare dal punto 1.

3.4 Sensori e selezione da file

Ad ogni Sensore sono stati associati due tipi di input: il primo ha come etichetta (il testo a lato del bottone che ha lo scopo di specificarne le modalità d'uso) *Selezione da file*, il secondo *Selezione manuale*. Come ci si può aspettare la selezione dell'input da file permette a l'utente l'associazione dell'ingresso del sensore ad un file testuale, mentre la selezione manuale porterà l'utente a far variare il valore letto dal sensore tramite il movimento dello slider.

Selezione da file

Il contenuto del file viene specificato secondo una regola semplice e ben definita. Il file, di estensione .txt, può essere pensato come diviso in sezioni: ogni sezione rappresenta un'intervallo con una determinata funzione di interpolazione ed è formata da una stringa che indica al simulatore quale metodo di interpolazione applicare ai valori successivi seguita da un certo numero di coppie (x, y(x)), dove x è un'istante temporale e y(x) è il valore della funzione in quell'istante. Dopo un numero di campioni arbitrariamente lungo (a discrezione del solo utente) il file può terminare o presentare un'altra sezione: anche qui la prima riga dovrà contenere la specifica indicante il metodo di interpolazione. Questa si suppone diversa da quella della sezione precedente, ma non impone vincoli in tal senso: la scelta spetta solo all'utilizzatore.

I metodi di interpolazione sono:

- const: rappresenta l'elaborazione costante. Una volta letto un valore questo viene mantenuto fisso fino alla lettura di una nuova coppia (x, y(x)). A questo punto il nuovo valore y(x) diventerà il valore di uscita per tutti i valori campionati fino alla lettura di una nuova coppia e così via;

- linear: elaborazione di tipo lineare. Per ogni intervallo di valori si calcola la retta che congiunge i due estremi e si ottiene il valore della funzione nel punto calcolando il valore di tale retta nell'istante desiderato;

- pol3: elaborazione tramite polinomio di terzo grado. Oltre a volere che la curva di interpolazione passi per gli estremi dell'intervallo, si impone che la sua derivata prima ivi si annulli. Questo permette di avere una funzione interpolante derivabile in tutti i suoi punti, dandole così raccordi smussati.

- sigm: elaborazione tramite sigmoide. Una curva ottenuta tramite interpolazione sigmoidale è una funzione esponenziale, simile ad un polinomio di terzo grado, che varia molto lontano dagli estremi dell'intervallo, mentre ha variazioni meno veloci in vicinanza di essi, dandole una forma a "S";

- sin: interpolazione tramite seno. Si fa in modo che una funzione del tipo A+B*sin(ω x+ φ) passi per i punti estremi dell'intervallo;

- cos: interpolazione tramite coseno. Si fa in modo che una funzione del tipo A+B*cos(ω x+ φ) passi per i punti estremi dell'intervallo;

- spline: interpolazione mediante spline. La spline è un polinomio di terzo grado, in cui si impone l'annullamento della derivata seconda nei punti estremi di ogni intervallo. In tali punti cioè si vuole avere un cambio di flesso.

In codice 3.1 è mostrato il contenuto di un file di input che muove lo slider dal valore 41 al valore 13 e viceversa, utilizzando l'elaborazione lineare.

Ulteriori approfondimenti in "Simulazione continua nel tempo dei sensori del robot didattico Mindstorm NXT"[15].

Codice 3.1: Esempio di file di input

```
linear
0 41
4 13
8 41
12 13
16 41
20 13
24 41
28 13
32 41
36 13
40 41
44 13
48 41
```

3.5 Un esempio di sessione completo

Nei prossimi paragrafi viene mostrato un esempio pratico d'uso del simulatore, partendo dalla costruzione del programma in NXT-G alla simulazione del file compilato. Il programma da utilizzare nella simulazione è realizzato in NXT-G,

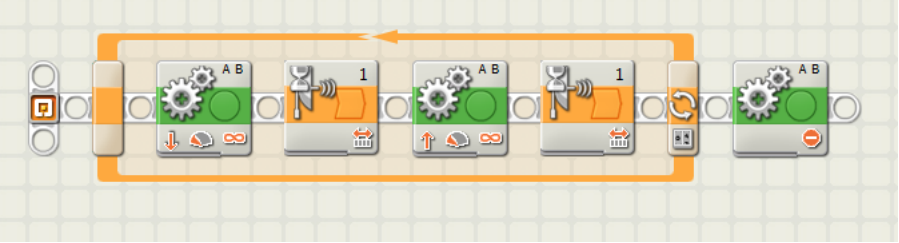

Figura 3.10: Il programma in NXT-G

ed è visibile in figura 3.10. Vediamo ora le caratteristiche dei diversi blocchi:

- il primo blocco Sposta è impostato con porte A e B, direzione indietro, potenza 40 percento e durata illimitata;
- il secondo blocco Sposta ha le stesse caratteristiche del primo tranne per la direzione che è impostata in avanti;
- il terzo ed ultimo blocco Sposta ha invece il campo direzione impostato in Stop;

- il primo blocco attendi ha come controllo il sensore ad ultrasuoni collegato alla porta 1 con l'opzione più vicino di 15 centimetri, vedi figura 3.12;
- il secondo blocco attendi ha come controllo il sensore ad ultrasuoni collegato alla porta 1 con l'opzione più lontando di 40 centimetri;
- il blocco iterazione ha come opzione di controllo Conteggio impostato al valore 6.

Figura 3.11: Le impostazioni del primo blocco Sposta

Figura 3.12: Le impostazioni del primo blocco Attendi

Una volta costruito basta andare sul menù Strumenti, selezionare la voce Download To File, scegliere il percorso dove salvare il file .RXE e premere Download per compilare il file. Il programma realizzato avvia i motori collegati alle porte A e B ed avanza ed indietreggia 6 volte per ognuna, avvicinandosi e allontanandosi dall'ostacolo immaginario.

Per l'input al sensore di ultrasuoni attraverso la modalità *Selezione da file* viene utilizzato il codice 3.1, salvato in un file di testo di nome "oscilla.txt".

Si procede ora avviando il simulatore (cliccando sul file NXTSimulator.jar) e seguendo i passi elencati precedentemente:

1. tramite il menù "File"selezionare "Carica .RXE", sfogliare il file system ed aprire il file .RXE compilato, "oscilla.rxe";
2. aggiungere alle porte A e B i due servomotori ed il sensore ad ultrasuoni alla porta 1;

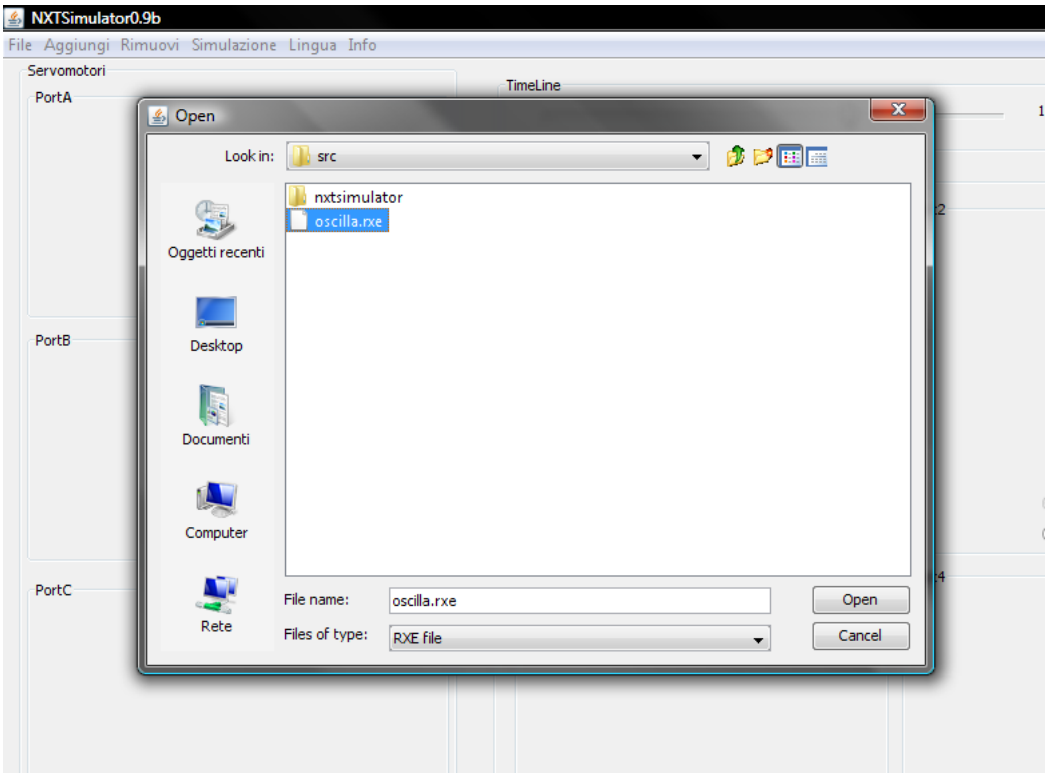

Figura 3.13: Selezione del file

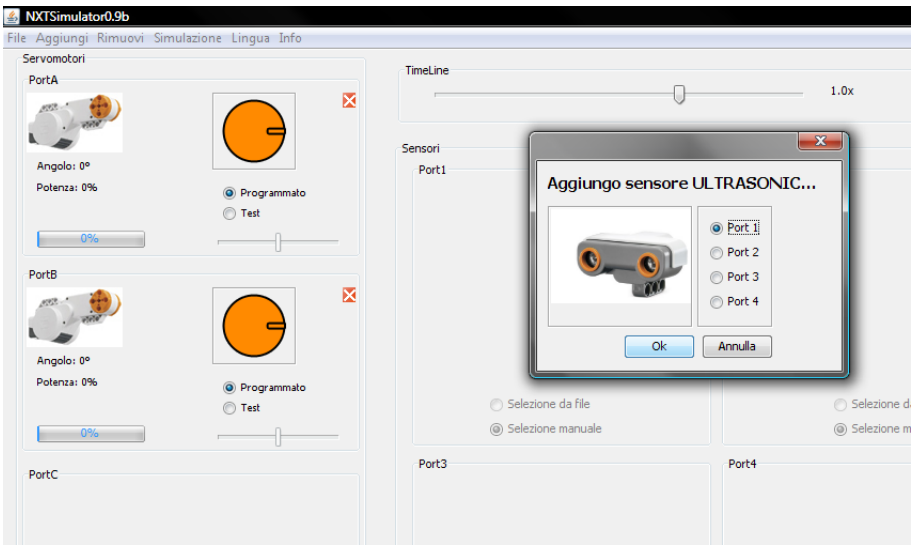

Figura 3.14: Aggiunta del sensore ad ultrasuoni alla porta 1

3. salvare la configurazione, andando su “Salva configurazione” dal menù “File”;

4. utilizzare lo slider del sensore per impostare la distanza iniziale desiderata;

5. avviare la simulazione tramite il menù "Simulazione"e cliccando su "Avvia";

6. durante la simulazione muovere lo slider avanti e indietro e vedere come ogni volta che si supera la soglia dei 40 cm o ci si avvicina di più di 15 cm i due motori cambiano direzione, cambiando il verso di rotazione e il valore della potenza diventa negativo o positivo ad indicare il cambio di direzione;

7. terminare il programma muovendo lo slider oltre le soglie per 6 volte ciascuna oppure andando nel menù "Simulazione"e cliccando su "Arresta".

Per rieseguire la stessa simulazione ma utilizzando la modalità "Selezione da file"avviare il simulatore (cliccando sul file NXTSimulator.jar):

1. tramite il menù "File"selezionare "Carica .RXE", sfogliare il file system ed aprire il file .RXE compilato, "oscilla.rxe";

2. caricare la configurazione precedentemente salvata andando in "File", selezionare "Carica configurazione"e aprire il file;

3. cliccare su "Selezione da file"nel pannello del sensore ad ultrasuoni e aprire il file "oscilla.txt";

4. avviare la simulazione tramite il menù "Simulazione"e cliccando su "Avvia";

5. durante la simulazione lo slider si muove con i tempi e i valori definiti nel file "oscilla.txt";

6. aspettare il termine della simulazione o terminarla andando nel menù "Simulazione"e cliccando su "Arresta".

Capitolo 4

Note tecniche di aggiornamento del software

In questo capitolo si descrivono gli aggiornamenti più significanti apportati al software NXTSimulator; Al fine di comprendere a fondo le modifiche è necessario fare una digressione e spiegare in dettaglio alcuni aspetti del firmware.

4.1 Formato del file .RXE

Il file RXE è diviso in 4 parti principali, come mostrato in tabella 4.1.

Segmento	**Dimensione**	**Tipo**
Header	38 Byte	Descrive gli altri segmenti del file.
Dataspace	variabile	Descrive le variabili e le strutture dati, nello specifico i tipi ed i valori di default.
Clump records	variabile	Informazioni riguardanti il momento in cui è prevista l'esecuzione di un Clump (serve per la schedulazione).
Codespace	variabile	Contiene il Bytecode delle sitruzioni.

Tabella 4.1: Struttura di un file RXE.

4.1.1 Header

Un file .RXE ha un'intestazione di 38 byte, chiamata Header, che descrive tutto il file. Questa intestazione può essere suddivisa in 4 parti come mostrato nella tabella 4.2.

Dimensione	Tipo	Descrizione
16 Byte	String	Questa stringa è caratteristica dei file RXE compilati per il firmware 1.03 e 1.05, in essa è contenuta la rappresentazione ASCII della stringa 'MindstormsNXT', seguita da un padding che indica la fine della stringa (0x0005), in altre parole tutti i file .RXE supportati dal firmware 1.03 iniziano con la seguente sequenza di bit: 4D 69 6E 64 73 74 6F 72 6D 73 4E 58 00 00 05 .
18 Byte	Dataspace header	Descrive la dimensione e la disposizione dei dati all'interno del file. Questi dati possono essere statici o dinamici.
2 Byte	Clump count	Rappresentato in memoria con un Word da 16-bit senza segno, descrive quanti Clump sono presenti all'interno del file.
2 Byte	Code word count	Rappresentato in memoria con un Word da 16-bit senza segno, descrive il numero di parole di codice (codeword) contenute nel Codespace.

Tabella 4.2: Struttura header del file RXE.

Dataspace header

Il dataspace header è la parte di intestazione che tiene conto del numero e della grandezza dei campi e degli oggetti presenti all'interno del dataspace. Questo segmento, che inizia dal sedicesimo byte, è formato da nove parole di 16 bit, in dettaglio:

- Count: Rappresenta il numero di campi presenti nella DSTOC;
- Initial Size: Grandezza, in Byte, del Dataspace completo;
- Static Size: Grandezza, in Byte, dei dati statici nel Dataspace all'interno della RAM;
- Default Data Size: Grandezza, in Byte, dei valori di default, sia statici che dinamici, memorizzati all'interno della RAM;
- Dynamic Default Offset: Offset, in Byte, dell'inizio dei valori di default dei dati dinamici;
- Dynamic Default Size: Grandezza, in Byte, dei dati dinamici;
- Memory Manager Head: Indice del primo elemento dell'array DVA della lista del memory manager;
- Memory Manager Tail: indice dell'ultimo elemento dell'array DVA della lista del memory manager;

- Dope Vector Offset: Offset, in Byte, della locazione iniziale del DV relativo all'inizio della porzione di dataspace nella RAM.

4.1.2 Dataspace

Il segmento dataspace contiene tutte le informazioni utili a inizializzare le strutture dataspace in memoria durante la fase di attivazione del programma. Il segmento è diviso in tre diversi sotto segmenti:

- tabella Dataspace of Contents2.2.5: contenente le informazioni riguardanti le veriabili del programma, come il tipo;
- Default Static Data: contiene i valori iniziali dei dati statici presenti all'interno della DSTOC;
- Default Dynamic Data: contiene i valori iniziali dei dati dinamici presenti all'interno della DSTOC;

Dataspace Table of Contents

La tabella DSTOC descrive tutti i tipi di dato e la loro locazione all'interno del programma. I suoi campi vengono utilizzati come riferimento, "puntatori", per le variabili da parte delle istruzioni.

La DSTOC non cambia durante l'esecuzione del programma, anche se i dati cambiano.

La struttura della DSTOC è mostrata nella tabella 4.3:

DSTOC Record			
Campo	Tipo	Flags	Data Descriptor
Bit	0..7	8..15	16..31

Tabella 4.3: Struttura dei record della DSTOC.

Il campo "Tipo"di 8 bit contiene un intero senza segno che identifica mediante un codice il tipo di dato; I valori riconosciuti dal firmware 1.03 sono:

0. TC_VOID: tipo di variabile non utilizzabile nel codice.
1. TC_UBYTE: intero di 8 bit, senza segno.
2. TC_SBYTE: intero di 8 bit, con segno.
3. TC_UWORD: intero di 16 bit, senza segno.
4. TC_SWORD: intero di 16 bit, con segno.
5. TC_ULONG: intero di 32 bit, senza segno.

6. TC_SLONG: intero di 32 bit, con segno.

7. TC_ARRAY: Array, che può essere composto da elementi di qualsiasi tipo ma tutti dello stesso tipo.

8. TC_CLUSTER: Struttura complessa composta da più campi anche di tipo diverso.

9. TC_MUTEX: Variabile utilizzata per lo scheduling dei Clump.

Gli 8 bit successivi del record DSTOC, il capo "Flag", indicano se la variabile è da inizializzare con il valore di default o con un valore definito a priori: se è presente il valore "1"vuol dire che la variabile deve essere istanziata con il valore zero, altrimenti se è presente il valore "0", deve essere inizializzata utilizzando il valore di default presente nel segmento dati corrispondente a questa variabile.

Le istruzioni gestiscono i dati delle variabili tramite un riferimento, indice, univoco della DSTOC, ma non tutti i dati possono essere riferiti direttamente, come, ad esempio, gli array; Infatti per gli elementi degli array è necessario prima puntare al dato TC_ARRAY e poi all'elemento che si vuole utilizzare.

Il campo data descriptor invece, non ha un unico significato, ovvero il suo contenuto e il suo scopo cambiano a seconda del tipo di dato a cui è riferito.

Descrizione dei dati nella DSTOC

La DSTOC segue regole specifiche per la descrizione di tutti i suoi possibili dati. Per le strutture complesse, quali array e cluster, viene seguita la regola secondo la quale il tipo della struttura è il primo elemento, e poi di seguito vengono elencati i tipi di elementi che la compongono, seguendo quindi una struttura di tipo top-down.

Più precisamente le regole sono:

- una variabile semplice occupa una riga di DSTOC. Il data descriptor specifica l'offset di questi valori nella RAM;

- una variabile di tipo array occupa due o più righe all'interno della DSTOC, seguendo questa disposizione: la prima riga descrive l'indice del Dope Vector a cui si riferisce l'array, le righe successive specificano il tipo di dati che compongono l'array, siano esse variabili normali o cluster, o arrta stessi;

- una variabile di tipo cluster occupa anche essa due o più righe all'interno della DSTOC, seguendo la seguente disposizione: il campo data descriptor della prima riga indica di quanti elementi è composto il cluster, mentre le righe successive specificano i tipi dei campi dello stesso. Anche quì sono ammessi tutti i tipi inclusi array e cluster.

Le regole descritte per i cluster e gli array vanno applicate ricorsivamente, ottenendo in questo modo strutture come array di array, o array di cluster, ecc.

Bisogna fare attenzione però agli offset contenuti nel data descriptor: infatti questi assumono significati diversi a seconda del tipo di dato a cui sono riferiti. I data descriptor delle variabili semplici e degli array indicano entrambi un offset senza segno di 16 bit all'interno del dataspace della RAM e sono chiamati "dataspace offset", mentre gli offset specificati nei campi degli elementi che compongono un array sono relativi all'inizio dell'array e sono chiamati "array data offset". Questa distinzione è resa necessaria dal fatto che gli array nella RAM non hanno una posizione fissa e quindi non è determinabile a priori quale sarà la posizione degli elementi che li compongono se non facendo riferimento all'inizio dell'array stesso.

Valori di default

I valori di default per i **dati statici** sono presenti subito dopo la DSTOC, e la grandezza di questi valori dipende direttamente dal tipo della variabile. I dati sono memorizzati nello stesso ordine con cui si presentano le variabili all'interno della DSTOC. Come detto in precedenza, se il valore nel campo flag delle variabili all'interno della DSTOC è pari a uno, il compilatore associa direttamente il valore di default 0 alle variabili.

La grandezza dello spazio dei valori di defualt statici è ricavata da 2 valori presenti nell' intestazione del file, nell'*header*: Default Data Size e Dynamic Default Offset. Il primo indica la dimensione dei valori di default sia statici che dinamici, ma non indica quanto è la dimensione dei dati dinamici, mentre il secondo indica l'offset dei dati dinamici e quindi permette di calcolare anche la dimensione dei dati di default statici.

Vi è solo una particolarità nel definizione dei valori di default statici per i dati di tipo *mutex*: questi infatti assumono come valore di default 0xFFFFFFFF (valore di 32bit).

I valori di default per i **dati dinamici** sono gestiti in modo diverso da quelli statici. Innanzitutto i dati dinamici sono rappresentati unicamente da array e la struttura dei valori di default dinamici deve essere "formattata"in modo da poter essere copiata direttamente nella memoria RAM senza ulteriori modifiche. Questa formattazione significa che devono essere inclusi tutti gli eventuali padding affinché la struttura rimanga bilanciata.

Infine si deve tener presente che ad ogni array nella DSTOC corrisponde un dope vector all'interno della struttura dei dati dinamici. Nella struttura dei dati dinamici troviamo all'inizio i valori di default per gli array e alla fine i dope vector che descrivono i singoli array.

4.1.3 Esempio di definizione di una struttura cluster

Supponiamo di trovare all'interno della DSTOC la seguente sequenza esadecimale:
08 00 04 00
02 01 70 00
01 01 71 00
07 00 72 00
01 00 00 00
01 01 74 00
Analizzando la sequenza, vedi tabella 4.4, si vede che la prima riga specifica una

Posizione in memoria	Valore	Tipo	Flags	Data Descriptor
0132	08 00 04 00	TC_Cluster	00	00 04
0136	02 01 70 00	TC_SByte	01	00 70
013A	01 01 71 00	TC_UByte	01	00 71
013E	07 00 72 00	TC_Array	00	00 72
0142	01 00 00 00	TC_UByte	00	00 00
0136	01 01 74 00	TC_UByte	01	00 74

Tabella 4.4: Esempio Definizione di un cluster nella DSTOC

struttura cluster composta da 4 elementi (data descriptor: 00 04), il primo elemento di questo cluster è un byte con segno, valore di default 0 e dataspace offset 00 70; il secondo elemento è un byte senza segno, con valore di default 0 e dataspace offset 00 71; il terzo elemento è un array con dataspace offset 0072, il tipo di elementi che lo compongono sono specificati nella riga succesiva ovvero, l'array contiene byte senza segno con valore di default diverso da 0 e offset rispetto all'array di 00 00 byte; il quarto elemento, che troviamo nella 5 riga, è un byte senza segno, con valore di default 0 e dataspace offset 00 74;

4.1.4 Clump Records

Prima di analizzare questo segmento è necessario riprendere il concetto di **Clump**, definito precedentemente. Ogni Clump rappresenta una porzione di codice e, se chiamato direttamente da un altro Clump, è definibile come una subroutine. I Clump sono utilizzabili per l'ambiente multitasking. La VM del Brick prende le decisioni sulla pianificazione dei vari Clump durante il l'esecuzione, basate sulle informazioni contenute nei Clump Records. Un Clump si definisce dipendente quando deve essere eseguito al termine di un altro Clump. Non essendo la VM una macchina dotata di Stack non è possibile applicare la rientranza: quindi non sono ammesse subroutine ricorsive e, per subroutine condivise, è necessario uti-

lizzare i mutex per evitare le interferenze.

Tabella 4.5: Struttura di un Clump Record.

Campo	Dimensione	Descrizione
Fire Count	1 Byte	Byte senza segno, indica quando il Clump è pronto ad essere eseguito.
Dependent Count	1 Byte	Byte senza segno, indica il numero di Clump dipendenti da questo Clump.
Code Start Offset	2 Byte	Word senza segno, indica l'offset dal Codespace rappresentante l'inizio del Clump.
Dependent List	Variabile	Array di Byte senza segno, puntano agli indici dei Clump record che dipendono da questo.

Il numero di Clump è definito nell'header. Ogni Clump record ha una lunghezza fissa di 4 Byte ed una parte variabile (la dependent list, tab. 4.5), quindi n Clump record hanno una parte fissa formata da $4 * n$ Byte ed una parte variabile che deriva dalla quantità di dipendenze.

Il numero massimo di Clump ammesso dal firmware 1.03 (e 1.05) è di 255 Clump per programma.

4.1.5 Codespace

Il segmento codespace del file eseguibile è formato da parole di 16-bit, interpretate come istruzioni di lunghezza variabile. Le istruzioni consistono in una o più codeword a seconda del tipo di codifica scelto. Le istruzioni possono essere di due tipi: long instruction e short instruction. La VM usa entrambi i tipi di codifica.
Il dodicesimo bit del primo codeword identifica il tipo di istruzione: se questo bit è pari al valore 1 allora l'istruzione appartiene alla categoria delle short instruction, altrimenti si tratta di long instruction.

Codifica long instruction

Le codifiche long sono quelle più utilizzate, anche perché la maggior parte delle istruzioni supporta solo questo tipo di codifica. La prima parola contiene il codice della operazione, la grandezza dell'istruzione, e il campo flag che viene usato solo da poche istruzioni. Le parole successive rappresentano gli operandi della istruzione, che nella maggior parte dei casi sono indici che puntano alla DSTOC (dataspace ID). La struttura di una istruzione è la seguente:

1. Word 1: 16 bit che sono scomposti nel seguente modo:

- OP_Code: byte senza segno che identifica univocamente l'istruzione;
- Size: quattro bit usati per indicare la grandezza, fornita come numero di byte, di tutta l'istruzione. Tuttavia se all'interno del campo si trova il valore 0xE, allora la dimensione è indicata nella parola successiva: questo accorgimento viene usato con istruzioni composte da molti argomenti;
- Flag: quattro bit utilizzati dalle operazioni che fanno la comparazione di 2 valori, e indicano che tipo di comparazione deve essere fatta.

2. Word 2, Word 3, ecc.: seguono la prima parola e contengono i parametri (argomenti) dell'istruzione.

Codifica short instruction

Per alcune delle istruzioni più comuni viene proposta una codifica alternativa per risparmiare spazio in memoria. Usare le istruzioni short encoding dipende dalla capacità del compilatore di fare i seguenti arrangiamenti delle istruzioni:

- per le istruzioni con un solo parametro è possibile usare la modalità short encoding se è possibile rappresentarlo con un solo byte;
- per le istruzioni con 2 parametri è possibile avere una codifica short se uno di essi può essere rappresentato da un singolo byte che è un offset rispetto all'indirizzo del secondo parametro.

Ogni istruzione che soddisfa questi criteri è possibile riorganizzarla in short encoding. Il campo *size* assume lo stesso significato e posizione delle istruzioni long encoding.
La struttura della prima parola è pertanto la seguente:

- il dodicesimo bit è impostato al valore 1;
- OP_CODE è rappresentato dai bit di posizione 13,14 e 15;
- per le istruzioni che utilizzano un solo parametro, nel primo byte dell'istruzione viene inserito l'indirizzo del parametro;
- per le istruzioni che utilizzano 2 parametri, nel primo byte della prima parola viene inserito l'offset, e nella seconda parola viene inserito l'indirizzo del secondo parametro.

Per calcolare l'indirizzo del primo parametro bisogna eseguire la seguente operazione: *Param1 = Param2 + Offset*
Bisogna tener presente che l'offset deve essere un valore a 8 bit con segno, che permetta al primo parametro di poter stare all'interno di un range di 255 indirizzi dal secondo parametro.

Nella seguente tabella 4.6 vengono riportati i valori che il campo OP_CODE assume per le istruzioni in short encoding, e in corrispondenza le istruzioni con la codifica short encoding affiancate dalle istruzioni con la codifica long encoding.

Valore OP_CODE	Short Instruction	Long Instruction
0	SHORT_OP_MOV	OP_MOV
1	SHORT_OP_ACQUIRE	OP_ACQUIRE
2	SHORT_OP_RELEASE	OP_RELEASE
3	SHORT_OP_SUBCALL	OP_SUBCALL

Tabella 4.6: Istruzioni Short Encoding e Long Encoding

La maggior parte delle istruzioni operano esclusivamente utilizzando i dati presenti all'interno del dataspace; quindi a tutti gli effetti gli operandi successivi alla istruzione sono degli indirizzi che si riferiscono al dataspace. Solo alcune istruzioni usano il valore successivo come valore immediato e quindi non si riferiscono ad alcun indirizzo di memoria. Ad esempio l'istruzione OP_JMP come argomento ha il valore di offset a cui saltare.
Per maggiori informazioni si rimanda al LEGO© Mindstorm© NXT Executable File Specification [4], dove vengono descritte in dettaglio tutte le operazioni.

4.2 Istruzioni di I/O

Come già accennato precedentemente esistono delle istruzioni di sistema (in inglese *System I/0 Instruction*) per interagire con le periferiche input/output del brick NXT.
Queste istruzioni sono mostrate e brevemente descritte nella tabella 4.7.

Chiamate a sistema

Per le chiamate a sistema si utilizza l'istruzione OP_SYSCALL, che è così strutturata:

- **OP_CODE**: 0x28;
- **Parametro 1**: SyscallID, tipo di chiamata a sistema, ne sono disponibili 33 in totale;
- **Parametro 2**: ParamCluster, cluster di memoria dove vengono memorizzati o letti i valori per elaborare la chiamata. Questa struttura, che cambia a seconda del SyscallID, serve da supporto per il dialogo con il sistema e le sue periferiche.

Valore OP_CODE	Istruzione	Descrizione
0x28	OP_SYSCALL	Invoca un chiamata a sistema, e a seconda del valore del suo primo argomento compie diverse azioni
0x30	OP_SETIN	Imposta i valori di configurazione dei sensori
0x31	OP_SETOUT	Imposta i valori di configurazione dei motori
0x32	OP_GETIN	Legge dai valori (specificati negli argomenti) dai sensori
0x33	OP_GETOUT	Legge dai valori (specificati negli argomenti) dai servomotori
0x35	OP_GETTICK	Salva in una variabile passata per argomento quanto tempo è trascorso dall'inizio dell'esecuzione del programma.

Tabella 4.7: Istruzioni System I/O

4.3 Il simulatore

Il software, realizzato in linguaggio Java, si può dividere in 3 parti logiche, ognuna delle quali ha un compito specifico:

- Analizzatore: comprende tutte le classi che si occupano di decodificare il file .RXE e di inizializzare e gestire le strutture dati per la simulazione;
- Interfaccia grafica: comprende un insieme di classi che si occupano, appunto, della interfaccia grafica, permettendo la selezione dei file da simulare, la visualizzazione delle componenti e l'iterazione i sensori;
- Connessione e supporto: a questo gruppo appartengono le classi che forniscono un collegamento tra l'analizzatore e l'interfaccia grafica, in modo da poter vedere gli effetti dell'esecuzione istruzioni a video, ma anche quelle che si occupano della conversione tra i vari tipi di dato e altre classi minori;

L'insieme di queste tre parti costituisce il simulatore NXTSimulator.

4.4 Aggiornamenti effettuati

In questa sezione vengono analizzate le problematiche del software risolte e le novità introdotte. Sono descritti solo i bug più rilevanti e tralasciati quelli banali come errori di casting o di indicizzazione errata, per non appesantire inutilmente la trattazione.

L'insieme di tutte le modifiche ha portato ad assegnare al software un nuovo numero di versione, 0.9b.

4.4.1 Schedulazione e OP_FINALCLUMPIMMED

Nella precedente versione del software vi era un errore nella schedulazione dei Clump dipendenti. In pratica quando il Clump finale di una serie di Clump dipendenti veniva eseguito e si ritornava all'inizio, il programma entrava in un ciclo senza fine, e ripeteva le istruzioni di continuo senza terminare mai.
Analizzando il file eseguibile con l'ausilio del Bricx Command Center e facendo vari debug del simulatore si è individuata l'istruzione che ne impediva il corretto funzionamento.
Questo problema si verificava, al richiamo dell'operazione OP_FINALCLUMPIMMED; la lista dei Clump dipendenti schedulati, infatti, non veniva aggiornata e i Clump che dovevano essere rieseguiti venivano saltati perché considerati già eseguiti.
Inoltre, sempre nella stessa operazione, si aggiornava solo il program counter (pc) ma non si "saltava"al nuovo Clump, questo causava un salto di istruzione nel Clump corrente e non un salto al Clump voluto.

Segue il listato di codice errato e quello corretto:

Codice 4.1: Codice errato di OP_FINALCLUMPIMMED

```
[..]
case 43:
        //OP FINALCLUMPIMMED
        coppia[0] = lett[pc + 2];
        coppia[1] = lett[pc + 3];
        source1 = conv.getPosInt(coppia);
        [..]
        pc = Clump[source1].getCodeStartOffset() * 2;
        break;
[..]
```

Codice 4.2: Codice corretto di OP_FINALCLUMPIMMED

```
[..]
case 43:
//OP FINALCLUMPIMMED
        coppia[0] = lett[pc + 2];
        coppia[1] = lett[pc + 3];
        source1 = conv.getPosInt(coppia);
        for(int i = 0; i < Clump[ClumpNow].getSize(); i++){
        scheduled[Clump[ClumpNow].getInvList(i)]=false;
        }
        [..]
        pc = Clump[source1].getCodeStartOffset() * 2;
        ClumpNow=source1;
break;
[..]
```

4.4.2 Gestione dei flussi di esecuzione

NXT-G permette di creare programmi con 1 o più flussi di esecuzione concorrenti, ma nella fase di testing dell'applicazione NXTSimulator si è visto che, se si prendevano due o tre blocchi motore e si applicavano ognuno ad un flusso di esecuzione diverso (figura 4.2), durante la simulazione uno solo di essi partiva.

Come se non bastasse, se si compilava lo stesso programma una seconda vol-

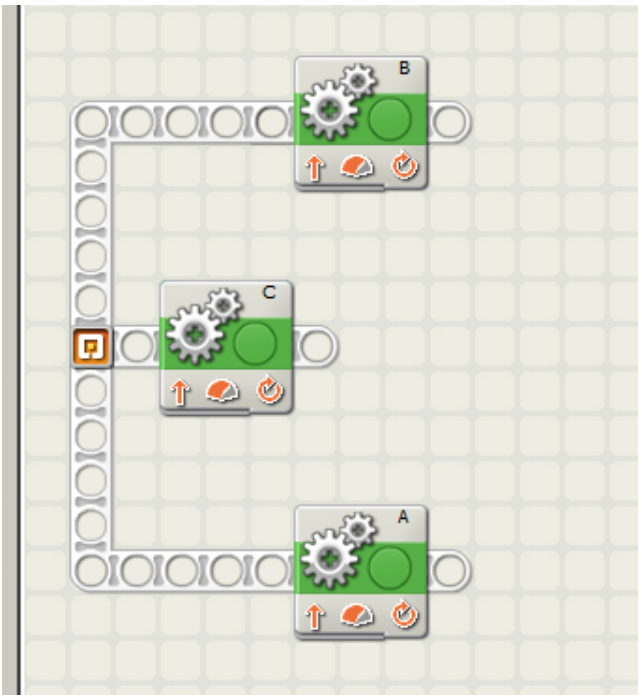

Figura 4.1: NXT-G:Tre motori in tre flussi di esecuzione distinti

ta, il motore che andava in esecuzione, durante la simulazione, non era mai lo stesso. Tramite il programma WinMerge sono state trovate le differenze dei vari eseguibili e con l'aiuto di Bricx Command Center è stata individuata la fonte dell'errore. In pratica, l'eseguibile .RXE conteneva la sequenza di Clump Record mostrata in tabella 4.8.

Da questa lista di Clump Record si vede come i primi 3 Clump abbiano il Fire Count a 0, ovvero sono tutti e tre destinati ad essere eseguiti come Clump iniziali.

Attraverso una attenta operazione di debug si è visto che la versione precedente del simulatore, iniziava la sua esecuzione solo con l'ultimo Clump che aveva Fire Count a 0 e non eseguiva mai il codice contenuto nei 2 Clump precedenti. Infatti, nel codice 4.3, si può vedere che, al termine del ciclo, la variabile ClumpNow (che definisce il primo Clump che deve essere schedulato) assume il valore dell'ultimo Clump con Fire Count uguale a 0.

Clump	Fire cnt	Dep Cnt	Code Start
000	00	00	0000
001	00	00	0049
002	00	00	0092
003	01	00	00DB
004	01	00	00E7
005	01	00	00F6
006	01	00	0140
007	01	00	017D
008	01	00	0190
009	01	00	01B9
010	01	00	01F0
011	01	00	0231
012	01	00	0249
013	01	00	0276

Tabella 4.8: Clump Record generato in NXT-G

Codice 4.3: Codice errato per l'individuazione dei Clump di partenza

```
[..]
for (int i = 0; i < head.getClumpCount(); i++) {
        if (Clump[i].getFireCount() == 0) {
                ClumpNow = i;
        }
}
[..]
```

La soluzione, visibile nel codice 4.4, è stata quella di creare una nuova classe nel pacchetto chiamata Execution.java, che estende la classe Thread, a cui far eseguire tutte le istruzioni successive, e quelle precedenti di costruzione delle strutture dati, al segmento di codice errato. Quindi si è modificato il codice errato, della classe Interpreter, con il codice 4.4, creando un thread distinto per ogni clump con Fire Count uguale a 0.

Codice 4.4: Codice corretto per l'individuazione dei Clump di partenza

```
[..]
for (int i = 0; i < head.getClumpCount(); i++) {
        if (clump[i].getFireCount() == 0){
                Execution t=new Execution(intFile,i);
                t.start();
        }
}
[..]
```

4.4.3 Sensore ad ultrasuoni I²C

Nella versione precedente del simulatore la possibilità di eseguire una comunicazione digitale tramite protocollo I^2C non era stata implementata. Tuttavia, si è visto che i programmi realizzati in NXT-G richiedono che questa funzionalità sia presente per far comunicare il brick con il sensore ad ultrasuoni.

Nella nuova versione, del software, questa comunicazione è stata realizzata, implementando i metodi OP_SYSCALL: *NXTCommLSRead* e *NXTCommLSCheckStatus*.

I tre metodi che permettono la comunicazione I^2C sono: *NXTCommLSWrite, NXTCommLSRead e NXTCommLSCheckStatus*;
Il metodo NXTCommLSWrite avvia la comunicazione con la periferica, specificando il numero di Byte che l'NXT birck si aspetta in risposta[1], NXTCommLSCheckStatus, invece, controlla che il dispositivo sia pronto a trasmettere e non sia occupato, mentre NXTCommLSRead legge i valori dal buffer e li salva in memoria.
Come accennato nel paragrafo 4.2, la struttura del parametro *ParmCluster* varia a seconda del tipo e in questo caso si è reso necessario lo studio del codice eseguibile per identificare la struttura di tali cluster. Il cluster mostrato in tabella 4.4, corrisponde a tutti gli effetti al ParmCluster usato per le due istruzioni NXTCommLSWrite e NXTCommLSRead, dove però le variabili assumono significati diversi a seconda dell'istruzione.
Per l'istruzione OP_SYSCALL NXTCommLSRead, il secondo parametro, ParmCluster, è un cluster che ha i seguenti campi:

1. Status Code, Byte con segno;
2. Port, Byte senza segno, indica la porta a cui e collegato il dispositivo;
3. Buffer, Array di Byte senza segno, contiene i dati letti dal sensore;
4. BufferLength, Byte senza segno, che indica il numero di Byte da leggere dal buffer.

L'istruzione *OP_SYSCALL NXTCommLSRead, ParmCluster*, copia il contenuto del buffer della periferica (che dovrebbe contenere il valore del sensore) nel campo buffer della struttura puntata da ParmCluster e ritorna il valore 0 memorizzandolo nel campo Status Code della struttura puntata da ParmCluster se l'operazione ha avuto successo.
L'istruzione OP_SYSCALL con primo parametro NXTCommLSCheckStatus serve solamente per controllare che la periferica sia libera e non ci siano errori di configurazione.
Il ParmCluster di NXTCommLSCheckStatus ha i seguenti campi:

[1] essendo una comunicazione asincrona di tipo Master-Slave ed il brick è il Master

1. Status Code, Byte con segno, se 0 il dispositivo è libero e non vi sono errori;
2. Port, Byte senza segno, indica la porta a cui e collegato il dispositivo;
3. BytesReady, Byte senza segno, indica il numero di byte pronti per essere letti, se ce ne sono.

Il nuovo frammento di codice inserito nel simulatore per la gestione dell'OP_SYSCALL NXTCommLSRead copia il valore, che lo slider del sensore assume in quel momento, nel campo Buffer del cluster passato come ParmCluster.

Codice 4.5: Implementazione di OP_SYSCALL NXTCommLSRead

```
[..]
case 22:
ArrayList temp2=new ArrayList();
porta=table[source2+2].getValDefault();
for(int i=0;i<array.length;i++)
        if((Integer)array[i].get(0)==source2+3)
                arNum=i;
for(int i=0;i<3;i++){
        temp2.add(i,array[arNum].get(i));
        }
int valore=NXTSView.getSensorController(porta).getValue(6);
temp2.add(3,valore);
array[arNum]=temp2;
[..]
```

L'istruzione OP_SYSCALL NXTCommLSCheckStatus è stata implementata semplicemente assegnando il valore 0 al campo Status Code del cluster indicato in ParmCluster.

4.4.4 Lettura dei valori dai sensori

Il Firmware 1.03 prevede che le porte di input (i sensori) abbiano diverse modalità di gestione dei valori di ingresso: le istruzioni *OP_GETIN e OP_SETIN*, che gestiscono le configurazioni dei sensori, hanno un parametro, *MODE*, che, passato per argomento, permette di scegliere in quale modalità effettuare la lettura del sensore.
Questo parametro MODE agisce appunto sul valore letto dal sensore scalandolo secondo le preferenze. Esistono diverse modalità: *RAWMODE, BOOLEANMODE, TRANSITIONCNTMODE, PERIODCOUNTMODE, PCTFULLSCALEMODE, CELSIUSMODE, FAHRENHEITMODE E ANGLESTEPMODE*, che applicano fattori di scalamento diversi.
Ogni volta che questo parametro di configurazione viene modificato, bisogna aggiornare anche un altro parametro del sensore, chiamato *INVALID_DATA*, ed impostarlo a valore *TRUE* (1).
Il problema, presente nella versione precedente del sensore, era che queste proprietà sebbene implementate venivano eseguite in modo errato. Nel simulatore la

proprietà MODE è rappresentata con un array di Byte, IN_MODE[ ], contenente la rappresentazione binaria del codice MODE associato.
Per individuare di quale modalità si trattava, il codice presente nella classe SensorGenericPanel, prendeva l'array "di bit"IN_MODE[ ], e controllava ad esempio se il bit alla posizione iesima assumeva il valore 1. Questo modo errato di individuare la modalità senza effettuare la dovuta conversione o il controllo di tutti i bit, provocava uno scalamento sbagliato del valore. Lo scalamento del valore veniva effettuato dall'ultima modalità che controllava il bit corrispondente a 1 in IN_MODE[ ].
Quindi le modalità che hanno più bit a uno nel vettore IN_MODE[ ], venivano "scavalcate"dalle altre modalità che avevano uno di quei bit a 1.

Un secondo errore, presente, era l'impostazione automatica del parametro *INVALID_DATA* a zero, senza che venisse verificato che gli opportuni scalamenti del valore grezzo fossero stati eseguiti. Una prima soluzione applicata al primo errore è stata la conversione del valore MODE binario in numero decimale e quindi il controllo effettuato su valori decimali.
Il secondo errore invece è stato corretto modificando la classe SensorData, realizzando una nuova funzione *setSensorValue*, che dopo aver effettuato gli scalamenti attraverso la funzione *setScaled* imposta il valore *INVALID_DATA* a zero. La funzione *setScaled* effettua anche gli scalamenti dovuti alla proprietà MODE, che sono stati rimossi dalla classe SensorGenericPanel. Seguono il frammento di codice errato e le nuove correzioni:

Codice 4.6: Codice errato della classe SensorGenericPanel

```
[..]
NXTSView.getSensorController(indice).IN_SCALED_VAL = 0;
//Modo PCTFULLSCALEMODE
if(NXTSView.getSensorController(indice).IN_MODE[8] == 1)
        NXTSView.getSensorController(indice).IN_SCALED_VAL=
                                                slider.getValue();
//Modo TRANSITIONCNTMODE o PERIODCNTERMODE o ANGLESTEPMODE
if(NXTSView.getSensorController(indice).IN_MODE[6] == 1 ||
NXTSView.getSensorController(indice).IN_MODE[4] == 1 ||
NXTSView.getSensorController(indice).IN_MODE[14] == 1)
NXTSView.getSensorController(indice).IN_SCALED_VAL =
                        (int)(slider.getValue()*655.35F);
//Modo BOOLEANMODE
if(NXTSView.getSensorController(indice).IN_MODE[2] == 1) {
        // Usata la logica inversa
        if (slider.getValue() >= 55)
        NXTSView.getSensorController(indice).IN_SCALED_VAL = 0;
        if (slider.getValue() <= 45)
        NXTSView.getSensorController(indice).IN_SCALED_VAL = 1;
        }
        NXTSView.getSensorController(indice).IN_NORMRAW =
                        (int)(slider.getValue()*10.24F);
        NXTSView.getSensorController(indice).IN_INVALID_DATA = 0;
}
[..]
```

Codice 4.7: Codice prima correzione

```
[..]
\\Recupero il valore decimale della proprietà MODE
mode=NXTSView.getSensorController(indice).getValue(1);
NXTSView.getSensorController(indice).IN_SCALED_VAL = 0;
if (mode == 128)
        NXTSView.getSensorController(indice).IN_SCALED_VAL =
                                        slider.getValue();
if(mode== 0)
        NXTSView.getSensorController(indice).IN_SCALED_VAL =
                                      (int)(slider.getValue());
// Modo TRANSITIONCNTMODE o PERIODCNTERMODE o ANGLESTEPMODE
if (mode == 96 || mode == 64 || mode == 224)
        NXTSView.getSensorController(indice).IN_SCALED_VAL =
                            (int)(slider.getValue()*655.35F);
// Modo BOOLEANMODE
if (mode == 32) {
// Usata la logica inversa
if (slider.getValue() >= 55)
NXTSView.getSensorController(indice).IN_SCALED_VAL = 0;
if (slider.getValue() <= 45)
NXTSView.getSensorController(indice).IN_SCALED_VAL = 1;
}
[..]
```

Codice 4.8: Codice sostituito a quello errato in SensorGenericPanel

```
[..]
// Applico gli opportuni scalamenti
//direttamente nella classe SensorData
NXTSView.getSensorController(indice).setSensorValue(slider.getValue());
[..]
```

Codice 4.9: Funzioni setSensorValue e setScaled

```
[..]
public void setSensorValue(int valore)
{
        VALORE_SENSORE = valore;
        setScaled();
        IN_INVALID_DATA = 0;
}
public void setScaled()
{
        int tipo=converter.arrToVal(IN_TYPE);
        //impostazione Type
        switch (tipo) {
        case 0: //no sensor
                IN_NORMRAW = VALORE_SENSORE;
                break;
        case 1: //SWITCH (TOUCH)
                IN_NORMRAW = VALORE_SENSORE * 1023;
                break;
        case 2: //Sensor Temperature
                IN_NORMRAW =
                        (int)(Math.floor((VALORE_SENSORE*0.999024375F)));
                break;
        case 3://Ligth Sensor
```

```
                IN_NORMRAW =
                        (int)(Math.floor(VALORE_SENSORE * 10.23F));
                break;
        case 4://Rotation Sensor valore grezzo 0-2147483647
                IN_NORMRAW =
                        (int)(Math.floor(VALORE_SENSORE*0.000000476F));
                break;
        case 11://I2C Sensor
                IN_NORMRAW = VALORE_SENSORE;
                break;
        case 12://I2C Sensor 9V
                IN_NORMRAW =
                (int)(VALORE_SENSORE * 4.092F) ;
                break;
        case 7:
                IN_NORMRAW =
                (int)(Math.floor(VALORE_SENSORE * 4.092F));
                break;
        case 8:IN_NORMRAW = VALORE_SENSORE;
                break;
        }
        //impostazione Mode
        int modo = converter.arrToVal(IN_MODE);
        switch(modo)
        {
        case 0: //Rawmode
                IN_SCALED_VAL = IN_NORMRAW;
                break;
        case 32:// Boolean Mode
                if (IN_NORMRAW > 560) IN_SCALED_VAL = 0;
                if (IN_NORMRAW <= 460) IN_SCALED_VAL = 1;
                break;
        case 64 ://TRANSITIONCNTMODE
                IN_SCALED_VAL =
                        (int)(Math.round(IN_NORMRAW*64.06158F));
                break;
        case 96:
                IN_SCALED_VAL =
                        (int)(Math.round(IN_NORMRAW*64.06158F));
                break;
        case 224:
                IN_SCALED_VAL =
                        (int)(Math.round(IN_NORMRAW*64.06158F));
                break;
        case 128:
                IN_SCALED_VAL =
                        (int)(Math.round(IN_NORMRAW*0.09775F));
                break;
        case 160:
                IN_SCALED_VAL =
                        (int)(Math.round(IN_NORMRAW*0.879765F)-200);
                break;
        case 192:
                IN_SCALED_VAL =
                (int)(Math.round(IN_NORMRAW*1.93548F)-400);
                break;
        }
}
[..]
```

4.4.5 Aggiunta di una Timeline

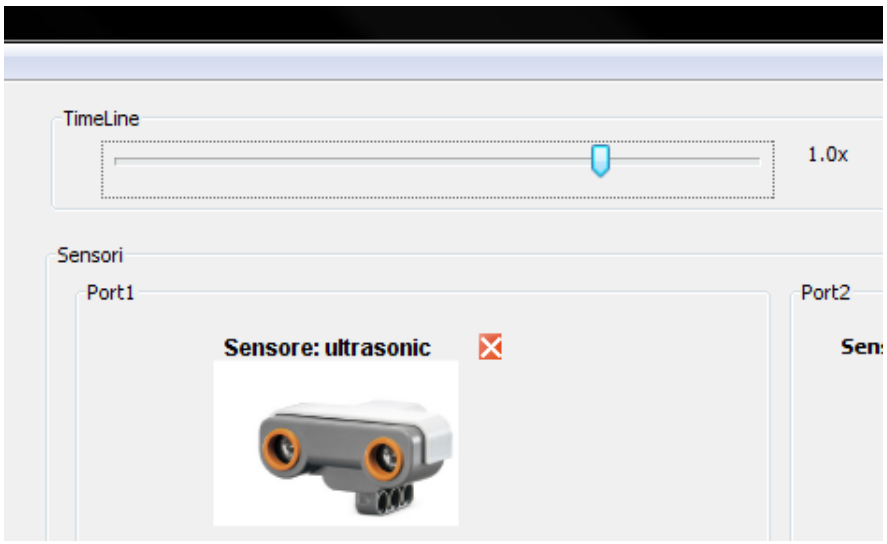

Figura 4.2: Scorcio dell'interfaccia con la Timeline

Nella nuova versione del software si è voluto dare all'utente finale la possibilità di velocizzare o rallentare la simulazione a proprio piacimento. Questo è possibile grazie al pannello TimeLine inserito nella parte superiore dell'interfaccia del simulatore. Il pannello contiene semplicemente uno slider che muovendosi influisce, utilizzando una scalamento esponenziale, su una variabile interna alla classe NXTSView chiamata *fattoreScala*. La variabile, è stata applicata come fattore di moltiplicazione a tutti i metodi che utilizzano variazioni temporali per avanzare, ad esempio nel codice 4.10, la variabile agisce sulla variabile temporale *diff* (differenza temporale dall'ultima esecuzione) per rallentare o aumentare la velocità di rotazione dei motori in relazione al tempo.

Codice 4.10: Modifica della velocità di rotazione dei motori

```
inc = speed * incPerc * diff * NXTSView.fattoreScala;
```

Codice 4.11: Aggiornamento della variabile fattoreScala

```
private void timeSliderStateChange(javax.swing.event.ChangeEvent evt) {
        fattoreScala=Math.pow(2,timeSlider.getValue());
        String temp=Double.toString(Math.pow(2,timeSlider.getValue()));
        if(temp.length()>4){
                jLabel1.setText(temp.substring(0,4)+"x");
        }else{
                jLabel1.setText(temp.substring(0,temp.length())+"x");
        }
}
```

4.4.6 Configurazioni multiple

Nella versione precedente del simulatore, era possibile salvare e caricare una singola configurazione di sensori e motori poiché le nuove configurazioni venivano salvate sempre nello stesso file "cfg.dat". Con la nuova versione, invece, è possibile salvare più file e di conseguenza caricare più configurazioni; La modifica ha interessato:

- la classe *NXTSView*, aggiungendo due funzioni *showSaveBox()* e *showLoadingCFGBox()*;
- la classe *Configuration*, modificando le tre funzioni *saveConfiguration()*, *loadConfiguration()*, e *setConfiguration()*, in modo da passare, come parametro, il file di configurazione selezionato dall'utente.

La funzione *showSaveBox()* si occupa di aprire la finestra per la selezione della cartella dove salvare il file ed esegue il salvataggio invocando il metodo *saveConfiguration2(File f)*; la funzione *showLoadingCFGBox()*, invece, carica la configurazione selezionata nella finestra di dialogo richiamando la funzione *loadConfiguration2(File f)*.

Codice 4.12: showSaveBox()

```
@Action public void showSaveBox() throws IOException{
        JFileChooser fc = new JFileChooser();
        FileFilter cfgFilter;
        cfgFilter = new FileNameExtensionFilter("File dat","dat");
        fc.addChoosableFileFilter(cfgFilter);
        fc.setCurrentDirectory (appConf.getLastOpenDir ());
        int returnVal = fc.showSaveDialog(mainPanel);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
                File f = fc.getSelectedFile();
                FileFilter selectedFilter = fc.getFileFilter();
                if (f.getName().indexOf('.') == -1) {
                        if (selectedFilter == cfgFilter) {
                                f = new File(f.getPath() + ".dat");
                        }
                }
                try {
                        cf.saveConfiguration2(f);
                }catch(IOException e) {
                JOptionPane.showMessageDialog(mainPanel, e, "ERRORE",
                                                JOptionPane.ERROR_MESSAGE);
                }
        }
}
```

Codice 4.13: showLoadingCFGBox()

```
@Action public void showLoadingCFGBox() throws IOException{
        JFileChooser fc = new JFileChooser();
        File f;
```

```
        FileFilter cfgFilter;
        cfgFilter = new FileNameExtensionFilter("File dat", "dat");
        FileFilter selectedFilter = fc.getFileFilter();
        fc.addChoosableFileFilter(cfgFilter);
        fc.setCurrentDirectory (appConf.getLastOpenDir ());
        int returnVal = fc.showOpenDialog(mainPanel);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
                f = fc.getSelectedFile();
                try {
                        cf.setConfiguration(f);
                } catch (IOException e) {
                JOptionPane.showMessageDialog(mainPanel, e, "ERRORE",
                                                JOptionPane.ERROR_MESSAGE);
                }
        }
}
```

Codice 4.14: Modifica a setConfiguration()

```
public void setConfiguration(File f) throws IOException{
if(!f.exists()){
        Message.main(SaveBeforeLoad);
}else{
        this.loadConfiguration2(f);
        for(int i=1;i<=4;i++){
                if(!(values[i-1].equals("missing"))){
                CRController.componentAdder(values[i-1], i);
                NXTSView.setEnabled(NXTSView.getFileBut(i));
                NXTSView.setEnabled(NXTSView.getManualBut(i));
                }else{
                boolean tes;
                tes = CRController.componentRemover(NXTSView.getPort(i),i);
                }
        }
        for(int i=5;i<=7;i++){
                if(!(values[i-1].equals("missing"))){
                CRController.componentAdder("motor", i);
                }else{
                boolean tes;
                tes = CRController.componentRemover(NXTSView.getPort(i),i);
                }
        }
        Message.main(ConfPortLoaded);
}
}
```

Codice 4.15: saveConfiguration2(File f)

```
public void saveConfiguration2(File f)throws IOException{
for(int i=1;i<8;i++){
        if(i>=5){
        values[i-1]=ConfController.motorConf(NXTSView.getPort(i) );
        }else{
        values[i-1]=ConfController.sensorConf(NXTSView.getPort(i) );
        }
}
raf=new RandomAccessFile(f,"rw");
raf.seek(0);
```

```
for(int i=0;i<7;i++){
        raf.writeUTF(values[i]);
}
raf.close();
Message.main(ConfPortSaved);
}
```

Codice 4.16: loadConfiguration2(File f)

```
public void loadConfiguration2(File f) throws IOException{
try{
        raf=new RandomAccessFile(f ,"r");
}catch(IOException e){
        System.out.println(e);
}
raf.seek(0);
for(int i=0;i<7;i++){
        try{
                if(raf.length()>0)
                        values[i]=raf.readUTF();
        }catch(IOException io){
                System.out.println(io);
        }
}
raf.close();
}
```

4.4.7 Altre modifiche

Reset automatico dei dati della simulazione

Nella nuova versione 0.9b di NXTSimulator, è stata rimossa la funzione che al termine di una simulazione resettava tutti i dati dei motori impedendo di vedere gli effetti di tale. Ora il reset delle variabili è situato all'inizio della simulazione, all'interno della funzione *startMotors* in modo che ad ogni avvio la simulazione riparta dalla posizione iniziale e con i valori azzerati.

Codice 4.17: Funzioni StartMotors e stopMotors prima della modifica

```
private void startMotors(){
    if(!isEmpty(5)){
        ((MotorPanel)prt5).startTimer();
    }
    if(!isEmpty(6)){
        ((MotorPanel)prt6).startTimer();
    }
    if(!isEmpty(7)){
        ((MotorPanel)prt7).startTimer();
    }
}


private static void stopMotors(){ //ferma i motori
    if(!isEmpty(5)){
        ((MotorPanel)prt5).resetMot();
```

```
    }
    if(!isEmpty(6)){
        ((MotorPanel)prt6).resetMot();
    }
    if(!isEmpty(7)){
        ((MotorPanel)prt7).resetMot();
    }

    md5.resetMotorData();
    md6.resetMotorData();
    md7.resetMotorData();

}
```

Codice 4.18: Funzione startMotors modificata

```
private void startMotors(){
     if(!isEmpty(7)){
        ((MotorPanel)prt7).resetMot();
        ((MotorPanel)prt7).startTimer();
    }
    if(!isEmpty(5)){
        ((MotorPanel)prt5).resetMot();
        ((MotorPanel)prt5).startTimer();
    }
    if(!isEmpty(6)){
        ((MotorPanel)prt6).resetMot();
        ((MotorPanel)prt6).startTimer();
        }
}
```

La rimozione del reset automatico ha portato alla luce molti bug presenti nel simulatore che precedentemente non venivano individuati.

Istruzioni per la manipolazione dei dati

Durante la ricerca dei bug presenti, sono state implementate le seguenti istruzioni, che non erano ancora supportate dal simulatore: *OP_STRCAT e OP_STRSUBSET*, la prima concatena una o più stringhe, la seconda copia un segmento della stringa passata come parametro. Il codice 4.19 ne mostra l'implementazione.

Codice 4.19: OP_STRCAT e OP_STRSUBSET

```
[..]
case 33:
        //OP_STRCAT
        coppia[0] = lett[pc + 2];
        coppia[1] = lett[pc + 3];
        size = conv.getPosInt(coppia);//dimensione dell'istruzione in # Byte
        coppia[0] = lett[pc + 4];
        coppia[1] = lett[pc + 5];
        dest=conv.getPosInt(coppia);
        dest = calcolaIndexArray(array, dest);
        ArrayList temp1 = new ArrayList();
        // Copio gli identificatori dell'array di destinazione
        temp1.add(array[dest].get(0));
```

```
        temp1.add(array[dest].get(1));
        temp1.add(array[dest].get(2));
        int f=0;
        for(int i=0; i<((size/2)-3); i++)
        {
                coppia[0] = lett[pc + 6+f];
                coppia[1] = lett[pc + 7+f];
                source1=conv.getPosInt(coppia);
                f=f+2;
                source1= calcolaIndexArray(array, source1);
                for(int c=3; c<array[source1].size()-1 ;c++)
                {
                if(Integer.valueOf(array[source1].get(3+i).toString())<127
                && Integer.valueOf(array[source1].get(3+i).toString())>31)
                        {
                        temp1.add(array[source1].get(c));
                        }
                }
        }
        temp1.add((byte)0);
        array[dest]= temp1;
        pc+=size;
        break;
case 34:
        //OP_STRSUBSET
        coppia[0] = lett[pc + 2];
        coppia[1] = lett[pc + 3];
        dest = conv.getPosInt(coppia);
        coppia[0] = lett[pc + 4];
        coppia[1] = lett[pc + 5];
        source1=conv.getPosInt(coppia);
        source1= calcolaIndexArray(array, source1);
        dest = calcolaIndexArray(array, dest);
        coppia[0] = lett[pc + 6];
        coppia[1] = lett[pc + 7];
        int indice=conv.getPosInt(coppia);
        coppia[0] = lett[pc + 8];
        coppia[1] = lett[pc + 9];
        int conta=conv.getPosInt(coppia);
        if(indice==65535)
        indice=0;
        ArrayList temp3 = new ArrayList();
        // Copio gli identificatori dell'array di destinazione
        temp3.add(array[dest].get(0));
        temp3.add(array[dest].get(1));
        temp3.add(array[dest].get(2));
        for(int i=indice; i<(conta+indice); i++){
        if(Integer.valueOf(array[source1].get(3+i).toString())<127
                && Integer.valueOf(array[source1].get(3+i).toString())>31)
                {
                temp3.add(array[source1].get(3+i));
        }else{
                break;
        }
        }
        temp3.add((byte)0);
        array[dest]=temp3;
        pc+=10;
        break;
```

```
[..]
```

Proprietà Output Port Configuration

Le proprietà Output Port Configuration riguardano i parametri delle istruzioni OP_GETOUT e OP_SETOUT. In particolare la proprietà FLAG è una sequenza di bit che indica quali valori della configurazione dei motori devono essere aggiornati. Nella precedente versione del software, questa proprietà non era implementata, perché ritenuta non necessaria. Tuttavia si è visto che molti programmi NXT-G fanno uso di una parte della proprietà FLAG, e quindi è stata implementata.

Codice 4.20: Implementazione FLAG

```
[..]
if (codeBit[13] == 1){
        System.out.println("UPDATE_TACHO_LIMIT ");
        outGen.println("UPDATE_TACHO_LIMIT ");
        outClump.println("UPDATE_TACHO_LIMIT ");
        UPDATE_TACHO_LIMIT=true;
        mc.TACH_COUNT=0;
        mc.TACH_LIMIT=TACH_LIMIT;
        if (mc1!=null){
                mc1.TACH_LIMIT=mc.TACH_LIMIT;
                mc1.TACH_COUNT=0;
        }
        if(mc2!=null){
                mc2.TACH_LIMIT=mc.TACH_LIMIT;
                mc2.TACH_COUNT=0;
        }
}
if (codeBit[12] == 1){
        System.out.println("UPDATE_RESET_COUNT ");
        outGen.println("UPDATE_RESET_COUNT ");
        outClump.println("UPDATE_RESET_COUNT ");
        mc.TACH_COUNT=0;
        if(mc1!=null){
                mc1.TACH_COUNT=mc.TACH_COUNT;
        }
        if(mc2!=null){
                mc2.TACH_COUNT=mc.TACH_COUNT;
        }
}
if (codeBit[10] == 1){
        System.out.println("UPDATE_RESET_BLOCK_COUNT ");
        outGen.println("UPDATE_RESET_BLOCK_COUNT ");
        outClump.println("UPDATE_RESET_BLOCK_COUNT ");
        mc.BLOCK_TACH_COUNT=0;
        if(mc1!=null){
                mc1.BLOCK_TACH_COUNT=mc.BLOCK_TACH_COUNT;
        }
        if(mc2!=null){
                mc2.BLOCK_TACH_COUNT=mc.BLOCK_TACH_COUNT;
        }
}
```

```
if (codeBit[9] == 1){
        System.out.println("UPDATE_RESET_ROTATION_COUNT ");
        outGen.println("UPDATE_RESET_ROTATION_COUNT ");
        outClump.println("UPDATE_RESET_ROTATION_COUNT ");
        mc.ROTATION_COUNT=0;
        if(mc1!=null)
                mc1.ROTATION_COUNT=mc.ROTATION_COUNT;
        if(mc2!=null)
                mc2.ROTATION_COUNT=mc.ROTATION_COUNT;
}
[..]
```

Rotazione dei motori e TachoLimit

Con la rimozione del reset automatico dei valori motore a fine simulazione, si è visto come i motori, nell'interfaccia grafica, in realta, non si bloccavano e continuavano a ruotare all'infinito. Con la semplice rimozione di questo errore ne è comparso un altro: i motori non si arrestavano correttamente ai gradi prestabiliti. Il problema era dovuto a due fattori:

- quando si confrontava il TachoLimit con i gradi percorsi, il motore aveva già superato la soglia;
- la grafica, a cui i valori del motore sono dipendenti, veniva bloccata al controllo successivo e non immediatamente.

Attualmente a questo nuovo errore si è mediato con una soluzione provvisoria, che prevede che una volta raggiunto il valore TachoLimit, che rappresenta la distanza in gradi da percorrere con il motore, venga bloccato l'avanzamento dei motori nel punto esatto del TachoLimit.
La soluzione proposta tuttavia non è ancora stata testata a dovere e potrebbe contenere ulteriori errori.

Codice 4.21: Soluzione attuale per il TachoLimit

```
public void actionPerformed(ActionEvent e){
actualTime = actualTimeStatic;
actualTimeStatic = System.currentTimeMillis();
diff = actualTimeStatic - actualTime;
if (!NXTSView.startMI.isEnabled()) {
//Se è attiva la simulazione eseguo l'aggiornamento automatico della grafica;
//interazione con MotorController
MotorData mc = NXTSView.getMotorController(portIndex);
if(programmed){
if (NXTSView.getMotorController(portIndex).RUN_STATE == 32) {
//modalità Running
        if (!started) {
        NXTSView.getMotorController(portIndex).ACTUAL_SPEED =
        NXTSView.getMotorController(portIndex).SPEED;
        speed = NXTSView.getMotorController(portIndex).ACTUAL_SPEED;
        started = true;
}
```

```
speed=NXTSView.getMotorController(portIndex).SPEED;
inc = speed * incPerc * diff * NXTSView.fattoreScala;
int Tach;
Tach = (int) (NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
if (NXTSView.getMotorController(portIndex).SPEED > 0) {
        if (Tach >= NXTSView.getMotorController(portIndex).TACH_LIMIT) {
                NXTSView.getMotorController(portIndex).RUN_STATE = 0;
                inc = NXTSView.getMotorController(portIndex).TACH_LIMIT -
                 NXTSView.getMotorController(portIndex).TACH_COUNT;
        }
} else if (NXTSView.getMotorController(portIndex).SPEED < 0) {
if (Math.abs(Tach) >= NXTSView.getMotorController(portIndex).TACH_LIMIT) {
        NXTSView.getMotorController(portIndex).RUN_STATE = 0;
        inc = 0 - NXTSView.getMotorController(portIndex).TACH_LIMIT +
        NXTSView.getMotorController(portIndex).TACH_COUNT;
        }
}
angle = (inc + NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT);
NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT = (int) angle;
NXTSView.getMotorController(portIndex).TACH_COUNT = (int)
        (NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
}
else if (NXTSView.getMotorController(portIndex).RUN_STATE == 64) {
//modalità Rumpdown
if (!started) {
        speed = NXTSView.getMotorController(portIndex).ACTUAL_SPEED;
        started = true;
}
inc = speed * incPerc * diff * NXTSView.fattoreScala;
int Tach;
Tach = (int) (NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
if (NXTSView.getMotorController(portIndex).SPEED >= 0) {
        if (Tach >= NXTSView.getMotorController(portIndex).TACH_LIMIT) {
        NXTSView.getMotorController(portIndex).RUN_STATE = 0;
inc = NXTSView.getMotorController(portIndex).TACH_LIMIT -
        NXTSView.getMotorController(portIndex).TACH_COUNT;
Tach = (int) (NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
}
} else if (NXTSView.getMotorController(portIndex).SPEED < 0) {
if (Math.abs(Tach) >= NXTSView.getMotorController(portIndex).TACH_LIMIT) {
NXTSView.getMotorController(portIndex).RUN_STATE = 0;
inc = 0 - NXTSView.getMotorController(portIndex).TACH_LIMIT +
         NXTSView.getMotorController(portIndex).TACH_COUNT;
Tach = (int) (NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
}
}
speed = NXTSView.getMotorController(portIndex).ACTUAL_SPEED =
 NXTSView.getMotorController(portIndex).ACTUAL_SPEED -
        ((NXTSView.getMotorController(portIndex).ACTUAL_SPEED -
        NXTSView.getMotorController(portIndex).SPEED) *
        (Tach / NXTSView.getMotorController(portIndex).TACH_LIMIT));
angle = (inc + NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT);
NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT = (int) angle;
NXTSView.getMotorController(portIndex).TACH_COUNT =
(int) (NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
} else if (NXTSView.getMotorController(portIndex).RUN_STATE == 16) {
//modalità Rumpup
if (!started) {
speed = NXTSView.getMotorController(portIndex).ACTUAL_SPEED;
```

```
        started = true;
 }
inc = speed * incPerc * diff * NXTSView.fattoreScala;
int Tach;
Tach = (int) (NXTSView.getMotorController(portIndex).TACH_COUNT+inc);
if (NXTSView.getMotorController(portIndex).SPEED >= 0) {
        if (Tach >= NXTSView.getMotorController(portIndex).TACH_LIMIT) {
        NXTSView.getMotorController(portIndex).RUN_STATE = 0;
        inc = NXTSView.getMotorController(portIndex).TACH_LIMIT -
         NXTSView.getMotorController(portIndex).TACH_COUNT;
        Tach = (int)(NXTSView.getMotorController(portIndex).TACH_COUNT+inc);
}
} else if (NXTSView.getMotorController(portIndex).SPEED < 0) {
        if (Math.abs(Tach) >=
                NXTSView.getMotorController(portIndex).TACH_LIMIT) {
        NXTSView.getMotorController(portIndex).RUN_STATE = 0;
        inc = 0 - NXTSView.getMotorController(portIndex).TACH_LIMIT +
        NXTSView.getMotorController(portIndex).TACH_COUNT;
        Tach = (int)(NXTSView.getMotorController(portIndex).TACH_COUNT+inc);
        }
}
speed = NXTSView.getMotorController(portIndex).ACTUAL_SPEED =
        NXTSView.getMotorController(portIndex).SPEED * (Tach /
        NXTSView.getMotorController(portIndex).TACH_LIMIT);
angle =(inc + NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT);
NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT = (int) angle;
NXTSView.getMotorController(portIndex).TACH_COUNT = (int)
(NXTSView.getMotorController(portIndex).TACH_COUNT + inc);
} else if (NXTSView.getMotorController(portIndex).RUN_STATE == 0) {
        //modalità Idle
        inc = speed * incPerc * diff * NXTSView.fattoreScala;
        angle = (inc + imagePanel.getAngle());
        NXTSView.getMotorController(portIndex).TACH_COUNT = (int)angle;
}else{
        inc = speed * incPerc * diff * NXTSView.fattoreScala;
        angle = (inc + imagePanel.getAngle());
        NXTSView.getMotorController(portIndex).BLOCK_TACH_COUNT =(int)angle;
}
setIntAngle(((int) angle) % 360);
serie[1] = serie[0];
serie[0] = Math.floor(angle / 360);
if (!isEqual(serie[0], serie[1])) {
        if (isIncreasing(serie[0], serie[1])) {
        if (increasedLast) {
        setRounds(getRounds() + 1);
        } else {
        setRounds(0);
        increasedLast = !increasedLast;
        }
} else {
if (increasedLast) {
        setRounds(0);
        increasedLast = !increasedLast;
} else {
        setRounds(getRounds() - 1);
}
}
}
NXTSView.getMotorController(portIndex).ROTATION_COUNT = getRounds();
```

```
imagePanel.setAngle(angle);
angleLab.setText(angleVar + ((getIntAngle()) % 360) + "°")
powerLab.setText("Power: " + speed + "%");
powerBar.setValue(Math.abs(speed));
rotationLab.setText(rotation + getRounds());
}
}
}
```

Rimozione motori

Dopo l'introduzione dei pulsanti *Selezione da File e Selezione manuale* nel pannello dei sensori, la rimozione dei motori tramite l'apposito comando sulla barra dei menù non veniva più effettuata.

Durante l'inserimento di tali pulsanti è stata modifica la funzione che si occupa-

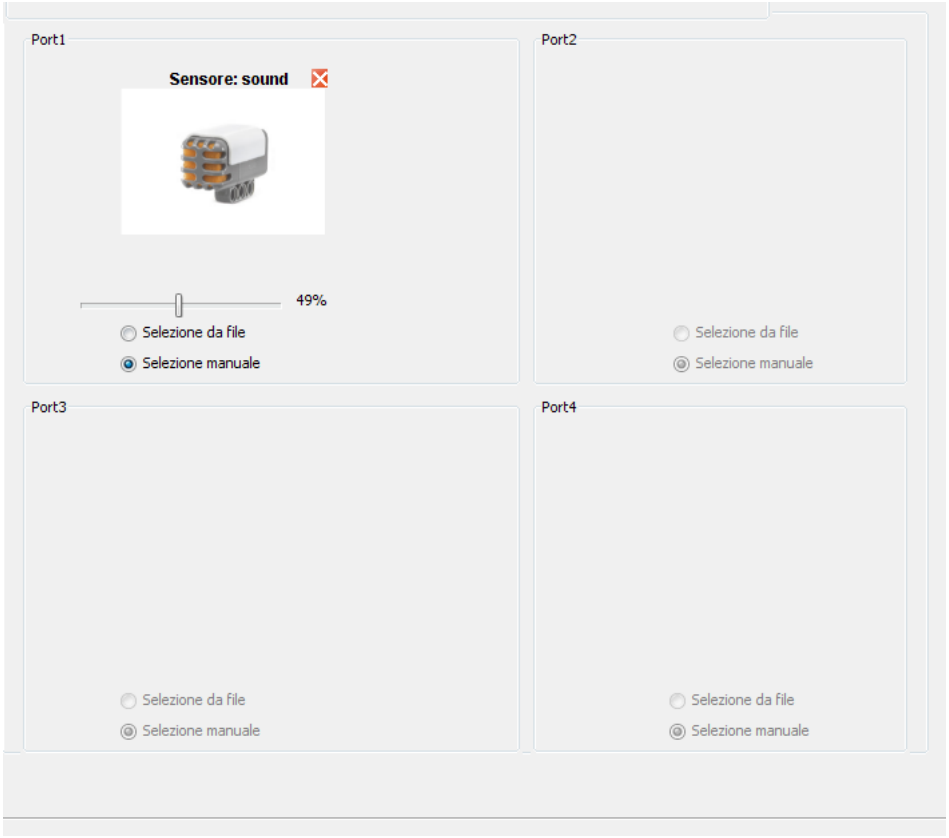

Figura 4.3: Pannello Sensori: pulsanti textitSelezione da File e Selezione manuale

va, appunto, della rimozione dei sensori e dei motori (nella classe CRController), influendo negativamente nella rimozione degli ultimi. Per risolvere il problema è stata creata una nuova funzione destinata alla rimozione dei soli motori, in quanto la rimozione dei sensori avveniva comunque. Il codice è visibile in 4.22.

Codice 4.22: Aggiunta di componentRemover nella classe CRController

```
public static boolean componentRemover2(java.awt.Container cont,int pos){
        if(cont.getComponentCount()!=0   ){//se la componente non è vuoto
                cont.removeAll();//rimuovi componente presente
                cont.repaint();//ridisegna il pannello
                if(pos>=0&&pos<=3)
                {
                NXTSView.setDisabled(NXTSView.getFileBut(pos));
                NXTSView.setDisabled(NXTSView.getManualBut(pos));
                }
                return true;
        }
//disabilitazione pulsanti
return false;
}
```

Codice 4.23: Modifica di clearSerPanel() nella classe NXTSView

```
private void clearSerPanel() {
        for(int i=5;i<=7;i++){
        @SuppressWarnings("static-access")
        boolean tes = CRController.componentRemover2(this.getPort(i),i);
        }
}
```

4.4.8 Bug noti

Al momento della stesura di questo elaborato è nota solo la presenza di un bug che causa in certe occasioni la terminazione anticipata della simulazione. Per eliminare l'errore si suggerisce di procedere in una fase di controllo, tramite una operazione di debug, delle variabili dei motori e delle strutture dati come array.

Conclusioni

Il software, dopo le diverse modifiche apportate, funziona correttamente e si può definire completo dal lato di esecuzione delle istruzioni principali, ma mancano ancora alcune funzionalità per supportare tutti i programmi NXT-G.

L'aggiornamento del software NXTSimulator ha richiesto, oltre ad una buona conoscenza di programmazione e di teoria degli elaboratori, un notevole lavoro di analisi dei codici eseguibili .RXE, di comprensione delle dinamiche del firmware [4] e del codice sorgente del simulatore.Le informazioni preventivamente acquisite, grazie alle relazioni derivanti dai precedenti studi per lo sviluppo del sw [7] [8], si sono rivelate molto utili nell'iniziare una fase di analisi sulla struttura della VM; ma solo lo studio prolungato del codice ha permesso di raggiungere una buona conoscenza del simulatore.

Le modifiche dell'applicazione NXTSimulator sono state apportate senza un seguire un ordine prefissato. La maggior parte, infatti, erano legate alla correzione di bug non noti. Ogni volta che veniva individuato un errore, il codice veniva corretto. A seguito, si attuava una nuova fase di testing, al fine di individuarne altri. La difficoltà maggiore è stata riscontrata nella rilevazione degli errori di casting mascherati con delle strutture *try and catch*.

L'attuale versione, la 0.9b,nonostante siano stati corretti molti bug ed implementate nuove funzioni, non può ancora definirsi quella finale, in quanto manca ancora l'implementazione di alcune istruzioni e funzionalità.

Per raggiungere la versione definitiva è necessario completare, quindi, le istruzioni mancanti ed eseguire una profonda fase di testing per individuare ulteriori bug non noti.

Gli sviluppi futuri del software possono essere svariati, ma si suggerisce l'aggiunta della simulazione del display, dei pulsanti e dell'altoparlante del brick nell'interfaccia grafica del simulatore. UN'altra possibilità di espansione futura potrebbe essere l'aggiornamento alla versione 2.0 del firmware ed, in un futuro più remoto, la realizzazione del simulatore in un ambiente 3D.

Lista degli Acronimi

Abbreviazione	Acronimo	Pagina
TERECoP	Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods	pag.1
JVM	Java Virtual Machine	pag.6
NBC	NeXT Byte Codes	pag.6
NXC	Not eXactly C	pag.6
JIT	Just In Time	pag.7
I2C	Inter Integrated Circuit	pag.4
VM	Virtual Machine	pag.14
DSTOC	Dataspace Table of Contents	pag.17
DV	Dope Vector	pag.18
DVA	Dope Vector Array	pag.18

Elenco dei Codici

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] Project TERECoP, http://www.terecop.eu/, (2009)

[2] NXT-G MINDSTORM NXT SOFTWARE,
http://mindstorms.lego.com/overview/NXT_Software.aspx

[3] John Hansen. Not eXactly C Programmer's Guide,
http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf

[4] LEGO. MINDSTORMS NXT Executable File Specification.

[5] John Hansen. Sourceforge NBC & NXC,
http://bricxcc.sourceforge.net/nbc/index.html

[6] Microsoft Robotic Studio,
http://msdn.microsoft.com/en-us/robotics/default.aspx

[7] Mauro Donadeo, Realizzazione di un simulatore elementare del sistema robotico Lego NXT in Java, 2008 Università degli Studi di Padova

[8] Andrea Donè, Realizzazione di un semplice simulatore per il robot didattico Lego Mindstorms NXT, 2008 Università degli Studi di Padova

[9] Filippo Buletto, Realizzazione di un simulatore semplificato del rovot educativo Mindstorm NXT, 2009 Università degli Studi di Padova

[10] NXT Firmware Open Source, The Open Source files include all the source files needed for the ARM7 ATMEL microcontroller and the 8-bit AVR ATMEL microcontroller..
http://mindstorms.lego.com/Overview/OpenSource.aspx

[11] Piaget, J. The Principles of Genetic Epistemology. N.Y.:Basic Books (1972)

[12] Piaget, J. To understand is to invent. N.Y.: Basic Books (1974)

[13] Paper, S. The Children's Machine. N.Y.: Basic Books (1992)

[14] Paper, S. Mindstorms:Children, Computers, and Powerful Ideas. N.Y.: Basic Books (1980)

[15] Marco Pierobon, Simulazione continua nel tempo dei sensori del robot didattico Mindstorm NXT, 2010 Università degli Studi di Padova