

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



**Un approccio per
l'aggiornamento dinamico sicuro
di controllori a stati finiti**

Relatore: Prof. Carlo Ghezzi
Correlatori: Dott. Valerio Panzica La Manna
Dott. Joel Greenyer

Laureando:
Donato CLUN
Matricola 798027

Anno Accademico 2014-2015

Ringraziamenti

Ringrazio il professor Carlo Ghezzi per tutte le opportunità e gli stimoli che mi ha dato, e le esperienze che mi ha consentito di fare. Grazie a lui ho potuto lavorare su un progetto che mi ha appassionato molto, ho potuto presentare il risultato del lavoro al workshop MiSE e partecipare alla conferenza ICSE, ed in generale ho potuto avere un piccolo assaggio di cosa significa fare ricerca nel campo dell'ingegneria del software, un'esperienza per me di valore inestimabile.

Ringrazio Valerio Panzica La Manna e Joel Greenyer per il supporto che mi hanno dato nello svolgimento di questa tesi. In particolare ringrazio Valerio anche per tutti i suggerimenti su come presentarlo nel migliore dei modi al workshop MiSE, e per tutti gli altri consigli che mi ha dato nel corso di quest'ultimo anno, l'utilità dei quali va ben oltre questo lavoro di tesi.

Un grazie speciale a tutte le persone che hanno creduto in me e che mi sono state vicine. Questo lavoro lo dedico a voi.

Grazie a tutti i miei amici, sia quelli di Trieste, sempre pronti ad accogliermi con gioia ad ogni mio ritorno, sia agli amici di Milano, con cui ho condiviso questo percorso.

Se sono arrivato a questo punto lo devo ovviamente anche ai miei genitori, che mi hanno sempre supportato.

Grazie di cuore a tutti.

Sommario

Sono molti i sistemi software che devono garantire lo svolgimento dei loro compiti senza interruzioni per lunghi periodi di tempo. Si pensi ai sistemi di controllo del traffico aereo, ai sistemi informativi finanziari, alle applicazioni industriali o spaziali, in cui fermare il sistema è pericoloso, troppo costoso, o semplicemente impossibile. Spesso i requisiti che questi sistemi devono soddisfare, o gli ambienti in cui tali sistemi funzionano, sono soggetti a dei cambiamenti non previsti che impongono la necessità di un aggiornamento del software. In questi casi l'aggiornamento deve necessariamente essere eseguito *dinamicamente*, a runtime, senza interrompere il funzionamento del sistema. Tale aggiornamento deve essere anche *sicuro*: durante e dopo l'aggiornamento il sistema deve continuare a soddisfare i requisiti di funzionamento, perchè in qualunque momento un comportamento erraneo potrebbe causare danni economici o mettere in pericolo la vita di persone. L'aggiornamento deve anche avvenire rapidamente: il cambiamento dei requisiti potrebbe infatti consistere nell'introduzione di nuovi requisiti di importanza critica che devono essere soddisfatti prima possibile; d'altra parte se l'ambiente in cui un sistema critico è in funzione cambia in un modo che compromette lo svolgimento corretto delle sue funzioni è fondamentale che l'adattamento al cambiamento avvenga prima possibile.

Questa tesi si focalizza sul problema dell'aggiornamento dinamico di un automa a stati finiti che controlla un reactive system, utilizzando l'approccio descritto in [8] che si basa sulla relazione

tra l'evoluzione delle specifiche del sistema e l'aggiornamento del software per i sistemi in cui il controllore è stato sintetizzato automaticamente a partire da tali specifiche, ed apporta i seguenti contributi.

Sviluppa un approccio sistematico alla realizzazione di un'architettura che consente l'esecuzione e l'aggiornamento dinamico sicuro degli automi a stati finiti utilizzati come controllori di reactive system. Tale architettura garantisce che l'aggiornamento avvenga dinamicamente non appena il controllore è in uno stato che lo consente, senza che il sistema mostri un cambiamento osservabile del comportamento.

Presenta un'implementazione di tale infrastruttura realizzata utilizzando Java sul sistema Lego Mindstorm NXT. Attualmente questa infrastruttura consente di eseguire ed aggiornare dinamicamente i controllori generati con ScenarioTools [24], uno strumento model-driven che consente di sintetizzare automaticamente degli automi a stati finiti che garantiscono il rispetto delle specifiche espresse tramite diagrammi MSD, ma è estendibile anche a qualsiasi altro strumento di sintesi che produca questo genere di controllore. L'esperienza acquisita realizzando questa implementazione evidenzia le potenzialità, i limiti, gli aspetti critici, e l'effettiva applicabilità dell'approccio quando esso è utilizzato in un contesto reale.

Indice

Elenco delle figure	5
Elenco delle tabelle	7
1 Introduzione	9
1.1 Il problema	12
1.2 L'obiettivo	14
1.3 Contributi	14
1.4 Lavori correlati	15
1.5 Struttura della tesi	16
2 Il problema	19
2.1 Il sistema d'esempio	19
2.1.1 I nuovi requisiti	20
2.2 Campo di applicazione: i reactive systems	21
2.2.1 Specifiche del reactive system d'esempio	21
2.3 Fondamenti teorici	23
2.3.1 Formalizzazione del problema	23
2.3.2 Fondamenti teorici dell'aggiornabilità	25
2.4 Definizione delle specifiche tramite MSD	28
2.4.1 Semantica degli MSD	29
2.4.2 Specifiche MSD dell'esempio	30
2.5 La sintesi automatica del controllore	31
2.5.1 L'algoritmo di play-out	33
2.5.2 La sintesi del primo controllore	33

2.5.3	La generazione del dynamically updating controller	34
2.6	Il Lego Mindstorm NXT Robot System	36
3	Analisi del problema	39
3.1	Gli eventi	39
3.1.1	L'alfabeto degli eventi	40
3.1.2	Requisiti relativi al software che gestisce gli eventi	41
3.1.3	Controllable event, uncontrollable event	42
3.1.4	Eventi del robot, eventi delle specifiche	43
3.1.5	Eventi legati alla comunicazione con altri sistemi	44
3.1.6	La modellazione del tempo	44
3.2	L'interpretazione e l'aggiornamento del controller	45
3.2.1	La struttura del controller	45
3.2.2	Esecuzione del controller	46
3.2.3	Aggiornamento del controller	47
3.2.4	Eliminazione del vecchio controller	48
3.3	Il sistema di comunicazione	48
4	Architettura della soluzione	51
4.1	Introduzione	51
4.2	Il sistema di gestione degli eventi	53
4.2.1	Gestione degli uncontrollable event	54
4.2.2	Gestione dei controllable event	58
4.3	Rappresentazione del controller	59
4.3.1	Gestione dell'accesso concorrente	61
4.4	Executor thread	63
4.4.1	Gestione della concorrenza	64
4.4.2	Eliminazione della vecchia versione del controllore	65
4.5	Deployer thread	67
4.5.1	Comandi	68
4.6	Logging	70
4.7	Software di gestione remota del robot	71

5 Implementazione	73
5.1 Software utilizzato	73
5.1.1 LeJOS	73
5.1.2 Piattaforma Java SE	74
5.2 Gestione degli eventi	74
5.2.1 Gestione degli uncontrollable event	74
5.2.2 Gestione dei controllable event	75
5.3 Rappresentazione del controller	75
5.3.1 La struttura dati dinamica	77
5.4 Gestione dell'accesso concorrente	79
5.5 Librerie in comune tra computer e robot	79
5.6 Executor thread	81
5.6.1 Dettagli dell'algoritmo di esecuzione	81
5.6.2 Implementazione	83
5.7 Deployer thread	83
5.7.1 Protocollo di comunicazione	84
5.8 Software di gestione remota	88
5.9 Sistema di logging	88
5.9.1 Esempio di log	90
6 Conclusioni	93
7 Sviluppi futuri	95
Bibliografia	97

Elenco delle figure

2.1	Il robot descritto nell'esempio	20
2.2	Le specifiche iniziali del comportamento del robot . . .	31
2.3	Le specifiche aggiornate che il robot deve rispettare dopo l'aggiornamento	32
2.4	Il controller sintetizzato automaticamente che soddisfa le specifiche iniziali	34
2.5	Il dynamically updating controller	35
2.6	Il componente principale del sistema Lego NXT con alcuni sensori e motori collegati	37
4.1	L'idea alla base dell'architettura	52
4.2	Le classi che rappresentano gli uncontrollable event nel sistema Lego NXT	55
4.3	La struttura dell'UncontrollableEventManager	58
4.4	Struttura da utilizzare nella rappresentazione del con- trollore	60
5.1	Implementazione dell'UncontrollableEventManager . . .	76
5.2	Classi che rappresentano i controllable robot event . .	77
5.3	Clasi utilizzate nella rappresentazione del controller . .	78
5.4	Alcune classi presenti nelle librerie comuni	80
5.5	La classe FiniteStateMachineExecutor che implementa l'executor thread	82
5.6	La classe DeployerThread	83
5.7	Il software di gestione remota	89
5.8	La classe Logger che implementa il sistema di logging .	90

Elenco delle tabelle

2.1	Eventi in ingresso	22
2.2	Eventi in uscita	22
2.3	Caratteristiche hardware del sistema Lego NXT	37
5.1	Struttura del comando FSMC	85
5.2	Struttura del comando PING	86
5.3	Struttura del comando EXIT	86
5.4	Struttura del comando SMSG	86
5.5	Struttura del comando RMSG	86
5.6	Struttura del comando GLOG	87
5.7	Struttura del comando MEMO	87
5.8	Struttura del comando GETC	87
5.9	Struttura del comando STAT	88

Capitolo 1

Introduzione

Dal suo esordio, non ha mai conosciuto declino l'utilizzo del computer come strumento di automazione nella gestione di macchine, processi e sistemi di complessità crescente in ambito industriale e scientifico, fino a diventare lo strumento centrale del funzionamento autonomo di sistemi sempre più complessi ed applicati a processi sempre più eterogenei. Parallelamente, l'evoluzione dell'applicazione delle tecnologie dell'informazione nella vita di tutti i giorni sta andando verso un modello in cui i computer sono talmente pervasivi e ben integrati negli oggetti che ci circondano da diventare sostanzialmente irrinconoscibili, e concetti attuali come pervasive computing, internet of things e smart objects rappresentano bene questa tendenza.

La conseguenza è che l'affermazione “i computer sono dovunque” sta diventando giorno per giorno sempre più vera e di conseguenza lo sta diventando anche l'affermazione “il software è dovunque”, ed in effetti il software è ormai un elemento essenziale di molti aspetti della vita e della società, che fa funzionare il sistema economico globale, le reti di telecomunicazione, e gran parte degli apparecchi che utilizziamo ogni giorno.

Chi ha a che fare con lo sviluppo di software sa però molto bene che i requisiti che i sistemi devono rispettare non sono quasi mai statici, ma al contrario evolvono nel tempo, molto spesso in ma-

niera completamente imprevedibile. Questa costante nello sviluppo del software è appesantita dal fatto che la natura stessa di questi cambiamenti è varia: in alcuni casi è l'ambiente in cui il software funziona che subisce un cambiamento a causa del quale è necessario modificare il comportamento del software in modo che possa continuare ad offrire lo stesso servizio, mentre in altri casi sono i nostri requisiti che subiscono un'evoluzione. Ad ogni modo l'unica via per poter continuare a soddisfare le specifiche è procedere all'aggiornamento del software, ovvero la realizzazione di una nuova versione in grado di soddisfare le nuove specifiche e la sostituzione del vecchio software con quello nuovo.

Sono sempre più frequenti, tuttavia, sistemi che devono obbligatoriamente continuare a svolgere il proprio compito senza interruzioni, magari perché un'interruzione del loro funzionamento potrebbe mettere in pericolo la vita di esseri umani (si pensi ai mezzi di trasporto a guida autonoma, o ai sistemi di controllo che regolano il funzionamento di apparati che possono produrre disastri come centrali nucleari o dighe), oppure perché l'interruzione del servizio porterebbe danni economici enormi (ad esempio i sistemi informativi finanziari), o ancora perché il sistema presenta delle caratteristiche che rendono semplicemente impossibile l'interruzione del funzionamento. In questi casi è quindi necessario che l'aggiornamento del software avvenga a runtime, ovvero mentre il sistema è in funzione e senza interromperne il funzionamento.

Il fatto che non vi siano interruzioni nella continuità del funzionamento del sistema è un vincolo che già di per sé può introdurre delle difficoltà nello sviluppo. Purtroppo però nella maggior parte dei casi tale vincolo non è nemmeno sufficiente, perché non esclude che la transizione dalla vecchia versione del software a quella nuova il funzionamento violi dei requisiti che è necessario che il sistema rispetti. Tale caso verrà ora illustrato meglio per mezzo di un esempio.

Si immagini una linea di produzione di autoveicoli composta da molte fasi in ciascuna delle quali dei macchinari devono eseguire

un certo insieme di operazioni su un veicolo, e poi inviarlo alla fase successiva. Dopo un certo tempo si decide di aggiornare il software che gestisce tutta la linea di produzione perché ne è stata sviluppata una nuova versione in grado di abbreviare il tempo di produzione ottimizzando l'ordine delle operazioni eseguite in ciascuna delle fasi. Si vuole però che tale aggiornamento venga eseguito senza “spegnere” l'intera linea di produzione perché tale operazione avrebbe un costo enorme, essendo necessario attendere che l'intera linea si svuoti completando tutti i veicoli su cui sta lavorando, ed essendo poi necessario attendere anche il suo nuovo riempimento dopo che il software è stato aggiornato. Si immagini quindi che in un certo istante di tempo il vecchio software di controllo venga rimpiazzato istantaneamente da quello nuovo.

È intuitivamente evidente che il nuovo software non può limitarsi a funzionare soddisfacendo i nuovi requisiti, ovvero eseguendo le operazioni nel nuovo ordine in ciascuna fase, ma deve anche gestire correttamente la transizione da un metodo di funzionamento all'altro, tenendo conto di quali operazioni il vecchio software aveva eseguito su ciascun veicolo, e trovando quindi una sequenza di operazioni che permetta di completare tutti i veicoli ancora incompleti. Se questo non fosse correttamente fatto, infatti, la linea di produzione potrebbe infatti subire gravi danni ad esempio tentando di montare il motore su un veicolo su cui in realtà era già stato montato.

Questo esempio dimostra la necessità di quello che viene definito *aggiornamento sicuro*, una garanzia essenziale del sistema sviluppato nell'ambito di questo lavoro di tesi, che consiste nel fatto che anche durante e dopo l'aggiornamento del software il sistema continui a soddisfare tutti i requisiti di funzionamento.

Garantire l'aggiornamento sicuro del software tuttavia non è ancora sufficiente: è infatti necessario tenere in considerazione anche la rapidità con cui ha luogo l'aggiornamento, magari perché i nuovi requisiti che devono essere soddisfatti sono di importanza critica, oppure perché è necessario adeguarsi rapidamente ad un cambia-

mento dell'ambiente in cui il sistema funziona. È quindi necessario garantire anche che il passaggio alla nuova versione del software avvenga prima possibile.

In conclusione, dato che la pervasività del software sta crescendo e ad esso vengono assegnati compiti sempre più critici in un ambiente in rapida evoluzione come quello moderno, è indispensabile sviluppare un approccio sistematico che consenta di affrontare il problema evidenziato in questa introduzione, fornendo le necessarie garanzie di corretto funzionamento, rapidità dell'aggiornamento, e - per quanto possibile - facilità di utilizzo. Il lavoro illustrato in questa tesi si propone di andare in questa direzione, concentrandosi su un componente essenziale di questo approccio: l'architettura del sistema di esecuzione.

1.1 Il problema

Il problema dell'aggiornamento dinamico del software citato nell'introduzione è in realtà estremamente vasto, perché il concetto stesso di software si presta a diversi strumenti in differenti contesti di utilizzo: il semplice automa a stati finiti che controlla il funzionamento di un tornello può infatti essere considerato software tanto quanto l'algoritmo che controlla il funzionamento dell'ABS su un veicolo, ma l'approccio al problema dell'aggiornamento dinamico è evidentemente diverso nei due casi. Anche il problema di garantire che durante l'aggiornamento non siano violate le specifiche può essere affrontato con una varietà di metodi; ad esempio alcuni di essi prevedono che il sistema o le sue parti vengano aggiornate nel momento in cui non sono coinvolte in nessuna interazione, mentre altri si concentrano sulla trasformazione dello stato dell'esecuzione attuale in uno stato dell'esecuzione della nuova versione del software.

Nell'ambito di questo lavoro il problema è stato preso in considerazione relativamente ai reactive systems, ovvero sistemi costituiti da componenti che interagiscono tra di loro e con l'ambiente tramite degli scambi di eventi, una vasta categoria di sistemi che tro-

va applicazione in molti campi, dall'automazione nella produzione industriale, alla logistica, ai più disparati sistemi di controllo automatico: anche un semplice termostato può essere visto come un reactive system, dato che riceve dall'ambiente degli eventi che corrispondono alle variazioni di temperatura, e reagisce inviando degli eventi che corrispondono all'apertura o chiusura di un circuito. In questo genere di sistemi il software di controllo è spesso un automa a stati finiti, le cui transizioni sono associate ad eventi in ingresso ed in uscita, che definisce il comportamento del sistema, e che quindi nel caso dell'aggiornamento dinamico costituisce il software che deve essere aggiornato.

Questa tesi si focalizza proprio sul problema dell'aggiornamento dinamico di un automa a stati finiti che controlla un reactive system, utilizzando un approccio che si basa sulla relazione tra l'evoluzione delle specifiche del sistema e l'aggiornamento del software, per i sistemi in cui il controllore è stato sintetizzato automaticamente a partire da tali specifiche.

Tale approccio è basato su Sscenario tools [24], un ambiente di sviluppo basato su Eclipse che consente di modellare le specifiche di funzionamento di un reactive system, sintetizzare in maniera automatica un controllore che garantisce la soddisfazione delle specifiche assegnate, modellare eventuali cambiamenti alle specifiche di funzionamento e a partire da essi sintetizzare un nuovo controllore in grado di eseguire l'aggiornamento dinamico sicuro, ovvero passare dall'esecuzione del vecchio controllore a quello nuovo, mentre il sistema è in funzione, garantendo che durante l'aggiornamento il sistema continui a soddisfare i nuovi requisiti.

Tale strumento si limita alla sintesi del vecchio e del nuovo controllore, mentre il processo grazie al quale nell'ambiente di esecuzione sia possibile passare dall'esecuzione della prima versione del controllore a quella più aggiornata rappresenta il problema aperto che è stato affrontato nell'ambito di questa tesi.

1.2 L'obiettivo

Per consentire la soluzione del problema descritto è necessario progettare un'infrastruttura che consenta innanzitutto l'esecuzione di un controllore generato da ScenarioTools, ovvero un automa a stati finiti privo di stati finali, le cui transizioni corrispondono a degli eventi in ingresso (ad esempio provenienti da un pulsante che invia un evento ogni qual volta viene premuto) e in uscita (per esempio degli eventi che corrispondono all'impostazione della velocità di un motore) che progredisce nella sua esecuzione ricevendo ed inviando eventi, e che in questo modo determina il comportamento del sistema controllato. La parte principale del problema tuttavia consiste nel supportare l'operazione di aggiornamento dinamico, ovvero supportare la capacità di far passare l'esecuzione dal controllore attuale al nuovo controllore non appena il sistema raggiunge uno stato che lo consente, e fare tutto questo mentre il sistema è in funzione e sta eseguendo il controller, senza che questo interferisca con l'esecuzione in maniera diversa da quanto previsto (ad esempio rallentando il tempo di risposta del controllore).

Parte dell'obiettivo consiste anche nel realizzare concretamente utilizzando il sistema Lego Mindstorm NXT, un kit contenente i componenti hardware necessari alla costruzione di semplici robot a scopo ludico ed educativo. Tale kit include un microcontrollore a cui è possibile collegare sensori ed attuatori di vario tipo inclusi nel kit, e varie componenti meccaniche con le quali costruire la parte fisica dei robot. Tale microcontrollore infatti è sufficientemente potente da eseguire una macchina virtuale Java semplificata, ed è possibile comunicare con esso, sia per caricare un nuovo programma, sia durante l'esecuzione, tramite USB e Bluetooth.

1.3 Contributi

Le più diffuse tecniche sviluppate per aggiornare sistemi software a runtime si basano sul fatto che il sistema non sia impegnato in

alcuna interazione, oppure sul fatto che il nuovo componente continui ad offrire lo stesso servizio della vecchia versione. La teoria su cui si basa l'approccio sviluppato in questo lavoro al contrario è innovativa poiché valuta l'evoluzione del software dal punto di vista dell'evoluzione delle specifiche, e quindi valuta quando l'aggiornamento è corretto rispetto alla variazione delle specifiche. Fino ad ora, però, non è mai stata messa in pratica su un'infrastruttura di esecuzione reale e questa tesi, puntando ad analizzarne l'applicabilità su un problema reale e le problematiche che tale applicazione presenta, porta i seguenti contributi:

- Propone un'infrastruttura che consente l'esecuzione e l'aggiornamento dinamico di controllori a stati finiti, attualmente complementare al sistema di sintesi ScenarioTools ma estendibile anche ad altri strumenti. Tale infrastruttura può essere eseguita su qualsiasi sistema in grado di eseguire una macchina virtuale Java, ed è facilmente riproducibile anche su altre architetture.
- Evidenzia i limiti, le potenzialità, gli aspetti critici e l'effettiva applicabilità dell'approccio quando esso è utilizzato in un contesto reale.

1.4 Lavori correlati

Sono stati proposti diversi approcci al problema dell'aggiornamento dinamico del software, che si differenziano in base ai contesti di utilizzo, e presentano caratteristiche diverse.

Alcuni approcci si basano sul fatto che il sistema al momento dell'aggiornamento non sia coinvolto in interazioni con altri sistemi. Quest'idea, che oltre ad essere stata utilizzata nei primi approcci all'aggiornamento dinamico [7, 13] è stata utilizzata anche su sistemi distribuiti [21, 27, 22], limita la possibilità di aggiornamento se il sistema da aggiornare è coinvolto in una interazione. Rispetto a tale approccio, quello utilizzato nell'ambito di questa tesi consente di

effettuare l'aggiornamento in un numero maggiore di casi, e quindi consente di ottenere un aggiornamento più tempestivo.

Nell'area dei linguaggi di programmazione sono state proposte diverse tecniche per aggiornare alcune componenti dell'ambiente di esecuzione per applicazioni in Java [11, 20] ed in C [18, 23]. L'approccio proposto in questa tesi d'altra parte è applicabile a qualsiasi sistema il cui comportamento sia controllato dall'interpretazione di un automa a stati finiti, a prescindere dal linguaggio e dall'architettura utilizzati.

In [14] viene proposto un framework formale per studiare l'aggiornamento dinamico del software, nel quale un aggiornamento dinamico è ritenuto valido se lo stato del vecchio programma è anche uno stato raggiungibile del nuovo programma. La nozione di stato aggiornabile utilizzata in questa tesi è simile, ma in questo caso si prendono in considerazione automi a stati finiti in relazione alle specifiche da cui sono stati generati in maniera automatica.

1.5 Struttura della tesi

La struttura della tesi è la seguente:

- Nel capitolo 2 viene proposta una descrizione dettagliata del problema e vengono introdotte tutte le informazioni che sono significative per affrontarlo: in quali casi si presenta, qual'è la teoria usata per affrontarlo, ed in generale tutti i vincoli che influenzano la soluzione.
- Nel capitolo 3 vengono analizzati in dettaglio diversi aspetti del problema in modo da evidenziare dei requisiti specifici che dovranno essere soddisfatti dalla soluzione.
- Nel capitolo 4 viene elaborata l'architettura della soluzione del problema: vengono illustrate le idee alla base del funzionamento del sistema e le scelte architetturali fatte, e viene fornita una visione d'insieme dei componenti del sistema e del ruolo di ciascuno di essi.

- Nel capitolo 5 viene descritta l'implementazione del sistema: le tecnologie utilizzate, i componenti sviluppati ed il loro funzionamento.
- Nel capitolo 6 vengono tratte le conclusioni sul lavoro svolto.
- Nel capitolo 7 vengono proposti dei possibili sviluppi futuri.

Capitolo 2

Il problema

In questa sezione verranno presentate tutte le informazioni che sono significative per capire ed affrontare il problema: i campi di applicazione in cui si presenta, i requisiti da soddisfare, la teoria che consente di affrontarli, i vincoli che derivano dalla piattaforma hardware utilizzata, e via dicendo. Per capire meglio il problema il resto di questo capitolo verrà sviluppato con l'ausilio di un esempio che presenta la necessità di un aggiornamento dinamico.

2.1 Il sistema d'esempio

Immaginiamo quindi di aver lanciato sul pianeta Marte una missione di esplorazione composta da un robot alimentato a batterie che si muove su ruote, dotato degli strumenti necessari per compiere gli esperimenti, ed una base di ricarica alimentata ad energia solare. Concentriamoci ora su una versione semplificata del comportamento che il robot deve rispettare quando si dirige alla stazione per ricaricare le batterie, ed ipotizziamo che il robot sia in grado di dirigersi alla stazione, e sia in grado di misurare la distanza da essa. Affinché l'avvicinamento avvenga in sicurezza è necessario che quando il robot si sta avvicinando alla base, non appena arriva ad una certa distanza rallenti la sua velocità. Per completare la procedura di ricarica è inoltre necessario che quando raggiunge la stazione di ri-

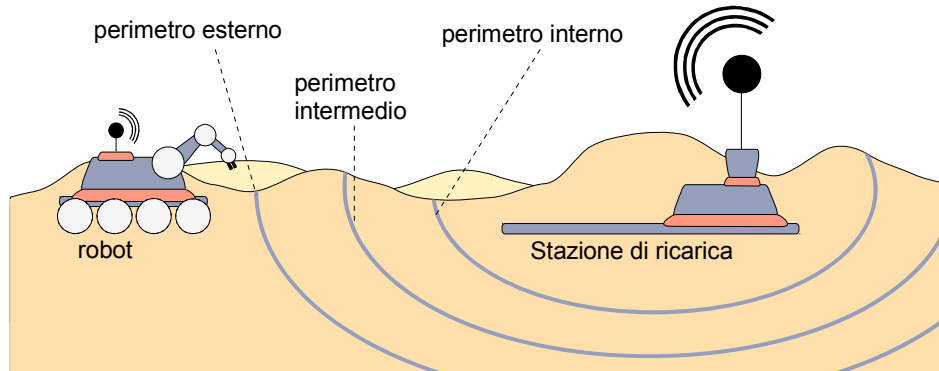


Figura 2.1: Il robot descritto nell'esempio

carica si fermi, si colleghi, e inizi a caricare le batterie. Appena il robot rileva che le batterie sono cariche il robot deve abbandonare la stazione per dedicarsi ai suoi esperimenti.

2.1.1 I nuovi requisiti

Ipotizziamo ora che a distanza di qualche anno venga inviato su Marte un altro robot, dotato di strumenti diversi, ma sempre alimentato a batteria. Per risparmiare peso però assieme al nuovo robot non è stata mandata una seconda stazione di ricarica, e quindi i due robot si trovano a dover condividere l'unica stazione presente.

Per evitare collisioni è necessario che durante la fase di avvicinamento alla base ogni robot si fermi appena viene raggiunta una certa distanza di sicurezza, chieda alla base l'autorizzazione ad avvicinarsi, e resti in attesa fin quando la base non autorizza l'avvicinamento. Per effettuare questa modifica al comportamento del vecchio robot è necessario aggiornare il suo software, ma ipotizziamo che tale operazione debba essere effettuata a runtime perché a causa di scelte progettuali non è possibile spegnere il robot. Ovviamente anche durante l'aggiornamento le specifiche di funzionamento devono essere rispettate, perché se così non fosse il robot potrebbe subire dei danni permanenti, ad esempio per non aver rallentato prima di raggiungere la base, oppure per non essersi fermato dopo aver-

la raggiunta. In altri termini l'aggiornamento deve essere sicuro: è necessario avere la garanzia che il sistema continui a rispettare le specifiche durante e dopo l'aggiornamento.

Nelle prossime sezioni con l'ausilio di questo esempio verranno descritti vari aspetti del problema.

2.2 Campo di applicazione: i reactive systems

Il problema affrontato nell'ambito del lavoro presentato in questa tesi è relativo ai cosiddetti reactive systems. In questa sezione essi verranno meglio definiti, e si mostrerà che l'esempio precedentemente descritto costituisce un sistema di questo tipo.

Un reactive system è un sistema che funziona rispondendo a degli stimoli esterni, provenienti dall'ambiente o dall'interazione con altri sistemi, producendo degli altri stimoli diretti verso l'ambiente o verso altri sistemi. Questa definizione molto astratta lascia intendere che essa è applicabile a diversi contesti, ed in effetti questa classe di sistemi trova ampia applicabilità pratica a patto però di chiarire in ciascun contesto cosa si intende per stimolo.

Nel contesto di questo lavoro di tesi, relativo alla realizzazione di robot o altri apparati mecatronici, il concetto di stimolo è legato sia ai sensori e gli attuatori di cui il sistema dispone, sia alle possibili interazioni non fisiche con altri sistemi (ad esempio l'invio e la ricezione di un messaggio attraverso un canale di comunicazione).

Il sistema che gestisce l'avvicinamento alla stazione di ricarica nell'esempio del robot introdotto in precedenza può certamente essere visto come un reactive system: esso riceve dall'esterno degli eventi di vario tipo (Tabella 2.1) e risponde ad essi con i possibili eventi in uscita (Tabella 2.2).

2.2.1 Specifiche del reactive system d'esempio

Si assuma che il robot sia in grado di identificare la propria posizione rispetto alla base di ricarica, e che sia in grado di determina-

Evento	Sorgente
Attraversamento perimetro	Sensore di distanza
Arrivo alla stazione di ricarica	Sensore di contatto
Batteria carica	Gestione della batteria
Autorizzazione all'avvicinamento	Stazione di ricarica

Tabella 2.1: Eventi in ingresso

Evento	Destinazione
Rallentamento, stop, avvicinamento	Motori collegati alle ruote
Richiesta avvicinamento	Stazione di ricarica
Inizio carica	Gestione della batteria
Collegamento a stazione	Sistema di collegamento

Tabella 2.2: Eventi in uscita

re l'attraversamento di differenti perimetri di sicurezza attorno alla base.

Le specifiche iniziali

Le specifiche iniziali sono date dai seguenti requisiti:

- Il robot, durante la fase di avvicinamento alla base, dopo aver attraversato il perimetro di sicurezza esterno deve necessariamente rallentare, e deve farlo prima di aver attraversato il perimetro di sicurezza interno.
- Appena arrivato alla stazione di ricarica, il robot deve fermarsi, collegarsi alla stazione e avviare la procedura di ricarica.
- Dopo aver concluso la ricarica il robot deve abbandonare la stazione.

e dalle seguenti assunzioni:

- Durante la fase di avvicinamento il robot, dopo aver attraversato il perimetro di sicurezza esterno oltrepasserà anche il perimetro di sicurezza interno e arriverà alla stazione.

- Il processo di ricarica delle batterie si conclude sempre con le batterie completamente cariche.

Le specifiche aggiornate

Le specifiche del sistema aggiornato sono costituite dalle specifiche iniziali a cui è stato aggiunto il seguente requisito:

- Dopo aver attraversato il perimetro intermedio, il robot deve chiedere alla stazione di ricarica l'autorizzazione per avvicinarsi, e deve arrivare alla stazione di ricarica solo se essa ha risposto affermativamente.

e le seguenti assunzioni:

- La base di ricarica risponde sempre a tutte le richieste ricevute.
- Il perimetro di sicurezza intermedio si trova tra il perimetro esterno e quello interno.

2.3 Fondamenti teorici

Questo lavoro di tesi si basa sull'approccio all'aggiornamento dinamico descritto in [8]. In questa sezione verranno introdotti i concetti teorici sufficienti a capire il problema

2.3.1 Formalizzazione del problema

Si consideri un sistema di oggetti che scambiano messaggi. Per semplicità si consideri che i messaggi sono sincroni, ovvero l'invio e la ricezione di un messaggio costituiscano un unico evento.

Definizione 1. (*Sistema, evento, alfabeto, run*). Un **sistema** è costituito da un insieme di oggetti O che scambiano messaggi. Ogni messaggio ha un nome, un oggetto mittente ed un oggetto destinatario. L'invio e la ricezione di un messaggio è un **evento**. L'**alfabeto** Σ è l'insieme dei possibili eventi che possono accadere nel sistema. Una sequenza infinita di eventi $\pi \in \Sigma^\omega$ è chiamata **run** del sistema.

Quale sia l'alfabeto degli eventi nell'ambito di un generico problema verrà affrontato nella sezione 3.1. Nel frattempo si possono prendere in considerazione gli eventi dedotti intuitivamente dalle specifiche presentate nella sezione 2.2 e più precisamente nelle tabelle 2.1 e 2.2;

Gli oggetti del sistema sono controllati da un *controller*, di seguito definito.

Definizione 2. (*Controller, linguaggio*). Un **controller** è un automa a stati finiti senza stati finali: un automa a stati finiti è la quadrupla (Σ, Q, q_0, T) , dove $Q = q_0, \dots, q_n$ è un insieme finito di stati, q_0 è lo stato iniziale, e $T \subseteq Q \times \Sigma \times Q$ è una relazione di transizione. Per un controller c , $L(c) \subseteq \Sigma^\omega$ è il **linguaggio** di c . Un run $\pi = (m_0, m_1, \dots)$ è un elemento di $L(c)$ se e solo se esiste una sequenza di stati che parte dallo stato iniziale del controllore $(q_0, q_1, \dots) \in Q^\omega$ tale che $\forall i \geq 1 : (q_i, m_i, q_{i+1}) \in T$.

Un controller può anche consistere nella composizione di controller diversi che controllano sottoinsiemi disgiunti di oggetti. I controller composti si sincronizzano grazie ad eventi che coinvolgono oggetti controllati da entrambi i controller. Per una definizione più rigorosa di composizione si rimanda a [8]. A questo punto si definisce cosa sia una specifica, e quando un sistema implementa una specifica.

Definizione 3. (*Specifica, soddisfare una specifica*). La **specifica** S è una tupla (A, R) in cui le **assunzioni** A ed i **requisiti** R sono insiemi di run. Un run π **soddisfa** le specifiche S , scritto $\pi \models S$, se e solo se $\pi \in A \implies \pi \in R$, ovvero se il run è nelle assunzioni deve anche essere nei requisiti. Un run è anche detto **ammissibile** rispetto ad una specifica se e solo se soddisfa tale specifica. Un controllore c soddisfa S , scritto $c \models S$ se e solo se ogni run in $L(c)$ soddisfa S .

Definizione 4. (*Oggetti di sistema e dell'ambiente*). Gli oggetti del sistema possono essere **oggetti di sistema** che possono essere controllati oppure **oggetti dell'ambiente** che non possono essere controllati. Per ogni controller degli oggetti dell'ambiente è necessario

che in ogni stato vi siano transizioni in uscita tramite le quali esso può ricevere qualsiasi evento mandato dagli oggetti di sistema agli oggetti dell'ambiente. È necessario che il controller degli oggetti di sistema sia infinitamente spesso in uno stato con transizioni in uscita che gli consentono di ricevere qualsiasi evento inviato dagli oggetti dell'ambiente agli oggetti di sistema.

In altri termini:

- L'ambiente non può mai impedire che un qualsiasi evento accada nel sistema.
- Il sistema può eseguire qualsiasi numero finito di passi, ma alla fine deve poter ricevere qualsiasi evento proveniente dall'ambiente.

Nell'esempio del robot per l'esplorazione di Marte il robot costituisce l'unico oggetto di sistema. La stazione di ricarica in questo caso viene considerata parte dell'ambiente.

Definizione 5. (Implementazione). *Un controller c per gli oggetti di sistema **implementa** o **realizza** S se e solo se c composto con ogni possibile controller e per gli oggetti dell'ambiente soddisfa S , più formalmente $\forall e, e \parallel c \models S$.*

2.3.2 Fondamenti teorici dell'aggiornabilità

Supponiamo che aggiornare il sistema tramite un aggiornamento offline, ovvero spegnere il sistema, sostituire il controller con la versione che soddisfa le nuove specifiche ed avviare il sistema con il nuovo controller nello stato iniziale, sia una possibilità accettabile. Senza considerare i dettagli relativi al come spegnere il sistema, assumiamo che un sistema controllato da un controller c che implementa la specifica S possa essere spento e riavviato con lo stesso controllore c nel suo stato iniziale senza violare le specifiche S . Da queste considerazioni si deduce che spegnere e riavviare

il sistema produce un comportamento equivalente a quello che si ottiene quando il controller visita lo stato iniziale.

Un aggiornamento online del controller c che implementa la specifica S verso una nuova specifica S' è quindi ammissibile se produce un run che è equivalente ad un run di un sistema controllato da c che raggiunge lo stato iniziale e continua l'esecuzione dallo stato iniziale di un nuovo controller c' che implementa le nuove specifiche. Un aggiornamento online del controller è quindi sempre possibile se il controller si trova nello stato iniziale. Possono però esservi anche altri stati in cui è possibile modificare il controller ed ottenere lo stesso comportamento. Questi stati sono chiamati *stati aggiornabili*.

Intuitivamente uno stato del controller c è aggiornabile se la sequenza degli eventi con cui si arriva ad esso partendo dallo stato iniziale può essere completata in un run che soddisfa le nuove specifiche. Tuttavia se il sistema è nello stato q_c e vi sono diverse sequenze che raggiungono questo stato a partire dallo stato iniziale, vi sono diverse sequenze di eventi che potrebbero aver condotto a q_c . Queste possibili sequenze sono le *possibili storie recenti* dello stato q_c .

Affinché lo stato q_c sia aggiornabile è necessario che:

- Ogni possibile storia recente possa essere completata in un run che soddisfa la nuova specifica S' .
- Una continuazione che completa una possibile storia recente soddisfacendo la specifica S' completi anche ogni altra possibile storia recente soddisfacendo S' .

Per determinare quali sono le possibili storie recenti di uno stato del controller è necessario tenere in considerazione anche gli eventi dovuti all'ambiente, quindi non è sufficiente considerare solo il controller del sistema, ma anche il comportamento dell'ambiente, comportamento che è descritto dalle assunzioni relative all'ambiente.

Se le possibili storie recenti debbano essere valutate utilizzando le assunzioni A delle specifiche S , o le nuove assunzioni A' relative alle nuove specifiche S' dipende dal fatto se il sistema sta funzionando in un ambiente che rispetta già le nuove assunzioni, oppure l'ambiente modificherà il suo comportamento durante o dopo l'aggiornamento. Nel seguito di questo capitolo verrà preso in considerazione il primo caso, ovvero quello in cui l'ambiente si sta già comportando secondo le nuove specifiche.

Definizione 6. (*Possibili storie, possibili storie recenti*). Consideriamo la composizione di c con ogni possibile controller dell'ambiente e' che soddisfi le assunzioni di S' tale che $L(e' \| c) \subseteq A'$. Le **possibili storie** $\Pi_{past}(c, q_{c_{cur}})$ sono i percorsi dallo stato iniziale di $e' \| c$ allo stato $q_{c_{cur}}$, $\Pi_{past}(c, q_{c_{cur}}) = \{(m_1, \dots, m_n) \in \Sigma^* \text{ tale che } \exists (q_0, \dots, q_n) \in (Q_{e'} \times Q_c)^*, q_0 = (q_{e'_0}, q_{c_0}), q_n = (q_{e'_n}, q_{c_{cur}}) \text{ e } \forall i \in \{0, \dots, n\} : (q_i, m_i, q_{i+1} \in T_{e' \| T_c})\}$. Le **possibili storie recenti** $\Pi_{past}^<(c, q_{c_{cur}})$ sono quelle possibili storie in cui lo stato di partenza non è visitato una seconda volta, ovvero $\forall i \neq 0 : q_0 \neq q_i$.

Definizione 7. (*Stato aggiornabile, aggiornamento corretto*). Uno stato $q_{c_{cur}}$ del controller c del sistema è **aggiornabile** alle specifiche S' se e solo se esiste un controller c' che implementa S' in cui la composizione di ogni possibile controller dell'ambiente e' ha linguaggio $L(e' \| c')$ in cui

1. per ogni possibile storia recente $\pi_{past}^< \in \Pi_{past}^<(c, q_{c_{cur}})$ ci deve essere un run $\pi \in L(e' \| c')$ in cui $\pi_{past}^<$ è un prefisso di π . In altre parole, deve esistere una continuazione di ogni possibile storia recente, che chiamiamo π_{futura} che concatenata con $\pi_{past}^<$ forma π
2. Se π_{futura} è una continuazione per una possibile storia recente, deve anche essere una continuazione per qualsiasi altra storia recente, $\forall \pi_{past}^{<1}, \pi_{past}^{<2} \in \Pi_{past}^<(c, q_{c_{cur}}), \pi_{past}^{<1} \neq \pi_{past}^{<2} : \pi_{past}^{<1} \cdot \pi_{futura} \in L(e' \| c') \implies \pi_{past}^{<2} \cdot \pi_{futura} \in L(e' \| c')$

*Un sistema effettua un **aggiornamento corretto** per soddisfare le nuove specifiche S' se si comporta in accordo al controller corrente c fin quando non viene raggiunto uno stato aggiornabile. A quel punto la sequenza di eventi che ha avuto luogo dal momento in cui il controller è stato nello stato iniziale per l'ultima volta viene completata in un run che soddisfa le specifiche S' .*

Nel seguito di questo capitolo sarà possibile vedere questo principio applicato al problema del robot per l'esplorazione di Marte: verranno mostrati il controller che implementa le specifiche iniziali, il controller che implementa le nuove specifiche, e le transizioni tra gli stati aggiornabili del controllore iniziale ed i corrispondenti stati del nuovo controllore. Prima però è necessario introdurre il funzionamento della sintesi automatizzata del controllore a partire dalle specifiche.

2.4 Definizione delle specifiche tramite MSD

I Modal Sequence Diagram (MSD) sono un elemento essenziale dell'approccio descritto in questa tesi, perché costituiscono il formato in cui sono definite le specifiche di funzionamento del sistema all'interno di ScenarioTools. In questa sezione si vedrà come le specifiche del sistema di esempio possono essere tradotte in MSD.

Le specifiche in questo formato consistono in un insieme di diagrammi MSD in cui ogni linea verticale (lifeline) rappresenta un oggetto del sistema (in tal caso l'etichetta associata è rettangolare) oppure l'ambiente (etichetta a forma di nuvola). Gli scambi di messaggi sono sempre considerati sincroni, nel senso che l'invio e la ricezione di un messaggio costituiscono un evento unico, e sui diagrammi sono rappresentati come frecce orizzontali tra due lifeline: ogni freccia parte dalla lifeline dell'oggetto che manda il messaggio e arriva alla lifeline dell'oggetto che riceve il messaggio, ed è associata al nome del messaggio inviato.

Per comprendere la semantica degli MSD è necessario introdurre il concetto di unificazione dei messaggi: un evento che avviene nel sistema è unificabile con un evento rappresentato con una freccia su un diagramma se l'oggetto che invia l'evento è lo stesso da cui parte la freccia, l'oggetto che riceve l'evento è lo stesso a cui arriva la freccia, ed il messaggio ha lo stesso nome.

2.4.1 Semantica degli MSD

In ScenarioTools, e di conseguenza in questo lavoro di tesi, vengono utilizzati soltanto universal MSD, ovvero diagrammi che formulano requisiti per tutti i run del sistema. Ogni messaggio su un MSD è caratterizzato da una temperatura e da un tipo. La temperatura può essere *calda* o *fredda*: i messaggi caldi sono colorati di rosso ed i messaggi freddi di blu. Il tipo può essere *eseguito* o *monitorato*: i messaggi eseguiti sono costituiti da una linea continua, i messaggi monitorati da una linea tratteggiata. Il primo messaggio di un diagramma è sempre freddo e monitorato.

Ogni MSD accetta un insieme di run del sistema. Se in un run del sistema avviene un evento che è unificabile con il primo evento di un MSD, viene creata una *copia attiva dell'MSD* (per brevità *MSD attivo*). Se il primo messaggio di un MSD si ripresenta in seguito nel diagramma non vengono create ulteriori copie attive dell'MSD.

Con il progredire degli eventi nel run l'MSD attivo progredisce, e per catturare questo progresso si segna l'insieme delle locazioni passate tramite un *taglio*. Se il taglio raggiunge la fine del diagramma il relativo MSD attivo viene scartato. Se il taglio di un MSD attivo è immediatamente prima di un messaggio, tale messaggio è abilitato. Se uno dei messaggi abilitati in un MSD attivo è caldo, il taglio è caldo. Se tutti i messaggi abilitati sono freddi, il taglio è freddo. Analogamente, se uno dei messaggi abilitati è eseguito, il taglio è eseguito. Se tutti i messaggi abilitati sono monitorati, il taglio è monitorato. I messaggi abilitati sono anche chiamati messaggi attivi o eventi attivi.

Le regole che determinano se un run sia accettato o meno sono le seguenti:

- Se l'MSD attivo è in un taglio caldo non è consentito che avvengano eventi non abilitati. Qualora questo dovesse accadere, si avrebbe una violazione di sicurezza, o violazione calda. Un MSD non accetta qualsiasi run in cui avviene una violazione di sicurezza.
- Se l'MSD attivo è in un taglio freddo ed avviene un evento che è unificabile con un evento attualmente non abilitato, si ha una violazione fredda dell'MSD. Le violazioni fredde sono concesse e fanno scattare l'MSD attivo.
- Se un taglio è eseguito ma non avanza mai, si ha una liveness violation. Un MSD non accetta run in cui ci sia una liveness violation. Se un MSD attivo è in un taglio monitorato, al contrario, potrebbe rimanere in questo taglio per sempre.

2.4.2 Specifiche MSD dell'esempio

Le specifiche iniziali

La figura 2.2 mostra la specifica iniziale del sistema composta da tre diagrammi MSD, due dei quali esprimono assunzioni sul comportamento dell'ambiente, ed uno esprime il requisito del sistema. Sui diagrammi sono rappresentati il robot, l'ambiente e la stazione di ricarica. Sul diagramma MSD che rappresenta il requisito di funzionamento, il primo messaggio (outPerim, che avviene quando il robot attraversa il perimetro esterno) è un messaggio freddo e monitorato, il che significa che lo scenario inizia quando il robot attraversa il perimetro. A quel punto il robot deve obbligatoriamente ridurre la sua velocità (messaggio reduceSpeed, di tipo caldo eseguito).

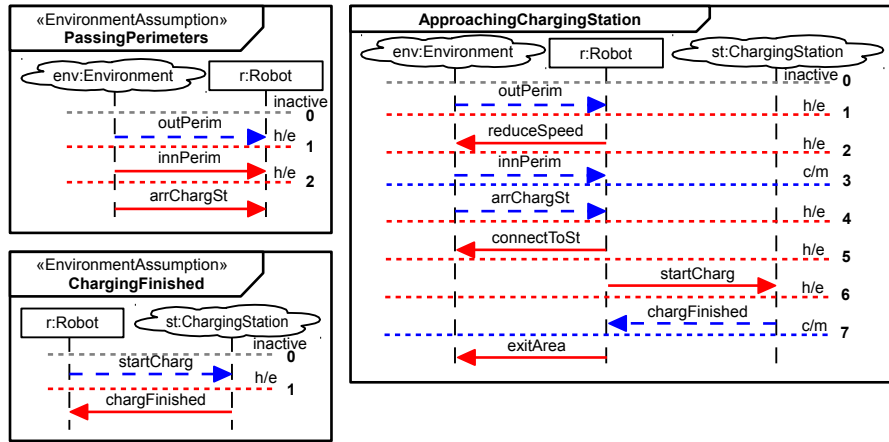


Figura 2.2: Le specifiche iniziali del comportamento del robot

Le specifiche modificate

La figura 2.3 mostra invece le specifiche che il robot deve rispettare dopo l'aggiornamento. Tali specifiche sono state ottenute aggiungendo dei diagrammi MSD che descrivono i nuovi requisiti e le nuove assunzioni.

2.5 La sintesi automatica del controllore

In questa sezione verranno illustrati gli elementi del processo di sintesi, integrato in ScenarioTools, che sono significativi per affrontare il problema. Tale processo può essere distinto in due parti: la sintesi del controllore a partire dalle specifiche, e la sintesi del dynamically updating controller a partire dal controllore iniziale e dalle nuove specifiche. Entrambe queste funzionalità sono integrate in ScenarioTools e corrispondono alle due fasi del ciclo di vita dell'esempio proposto: la sintesi del controllore a partire dalle specifiche genera un controllore che soddisfa i requisiti della versione iniziale in cui un solo robot deve accedere alla stazione di ricarica, mentre la sintesi del dynamically updating controller genera il controllore che realizza l'aggiornamento sicuro e fa funzionare il sistema quando vi sono due robot che condividono la stazione di ricarica.

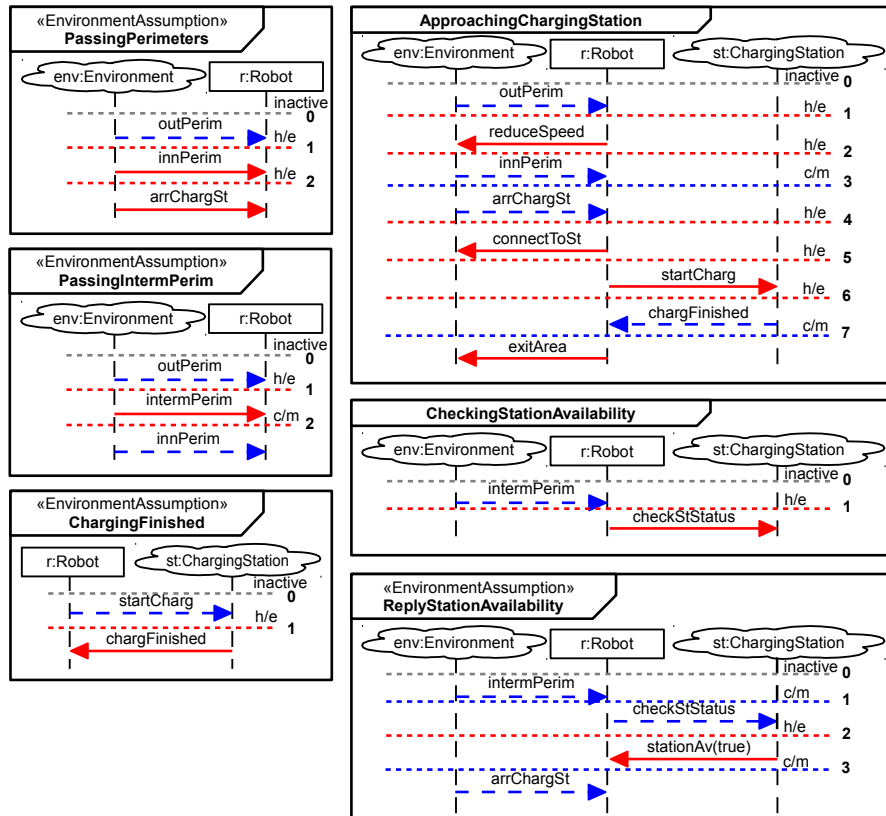


Figura 2.3: Le specifiche aggiornate che il robot deve rispettare dopo l'aggiornamento

2.5.1 L'algoritmo di play-out

Gli MSD descritti costituiscono un formalismo intuitivo ma preciso che consente di catturare i requisiti del sistema e le assunzioni riguardanti l'ambiente, tanto che possono essere utilizzati direttamente per far funzionare il sistema utilizzando il seguente algoritmo di play-out [17]: all'inizio il sistema non fa nulla, e attende che avvenga un evento dell'ambiente ed a quel punto, come descritto nella sezione 2.4.1, se l'evento ricevuto è unificabile con il primo evento di uno degli MSD vengono create le relative copie attive degli MSD coinvolti. A quel punto, se ci sono MSD attivi in un taglio eseguito, esiste un insieme di eventi attivi (abilitati ed eseguiti). L'algoritmo sceglie quindi di eseguire uno di questi eventi possibili che non violi uno degli altri MSD. In questo modo il sistema progredisce facendo avanzare il taglio di alcuni diagrammi attivi, ed eventualmente causando l'attivazione di altri MSD. Questo processo viene ripetuto fin quando non vi è alcun MSD attivo con ancora un taglio eseguito, e quando questo accade il sistema torna ad attendere l'arrivo di un evento dall'ambiente che attivi degli MSD.

L'esecuzione di questo algoritmo, tuttavia, può portare a una situazione in cui esiste un insieme di eventi attivi, ma tutti questi eventi sono vietati perchè porterebbero alla violazione di un altro MSD, ed in questo caso l'algoritmo non può procedere.

2.5.2 La sintesi del primo controllore

Per sintetizzare un controllore che garantisca il rispetto delle specifiche, ScenarioTools utilizza una tecnica basata sul mappare il problema di sintesi in un gioco a due giocatori che può essere risolto utilizzando gli algoritmi che sono implementati in Uppaal Tiga [4], un'estensione del model checker Uppaal sviluppata per risolvere questo tipo di problemi. Tale strumento viene utilizzato per calcolare una *strategia vincente* grazie alla quale il sistema può sempre reagire ad eventi del sistema in modo tale che le specifiche siano sempre rispettate.

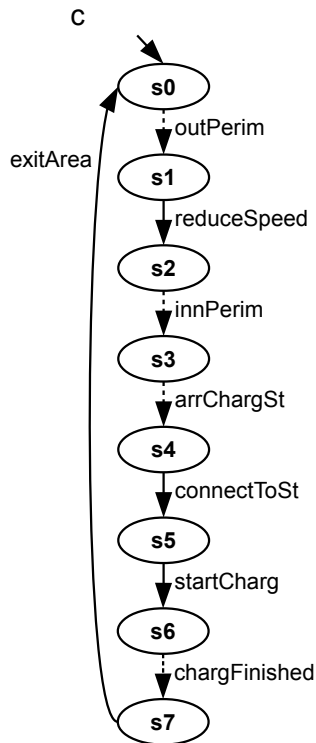


Figura 2.4: Il controller sintetizzato automaticamente che soddisfa le specifiche iniziali

Utilizzando questo approccio, contenuto in ScenarioTools, è possibile quindi sintetizzare un controllore che soddisfi le specifiche illustrate nella figura 2.2, ottenendo quindi il primo controllore rappresentato in figura 2.4 (per una spiegazione più dettagliata si rimanda a [10]).

2.5.3 La generazione del dynamically updating controller

Il *dynamically updating controller* è un controller che si comporta come il controller attuale e, appena viene raggiunto uno stato aggiornabile, si aggiorna dinamicamente al nuovo comportamento.

Ipotizzando che il controller *c* che implementa la specifica *S* sia già noto e correntemente utilizzato per far funzionare il sistema,



Figura 2.5: Il dynamically updating controller

data una nuova specifica S' il dynamically updating controller viene costruito nel seguente modo:

- Utilizzando il processo di sintesi introdotto nella sezione precedente si sintetizza il nuovo controllore c' che soddisfa la nuova specifica S' .
- Si individuano gli stati aggiornabili del controllore c , e per ciascuno di essi si trova il corrispondente stato del nuovo controllore c' .
- Si aggiungono delle ϵ -transition che partono dagli stati aggiornabili del controllore c ai corrispondenti stati di c' .
- Si eliminano le altre transizioni che partono dagli stati aggiornabili di c .

Durante l'esecuzione di questa procedura tutti gli stati del controllore attuale c vengono mantenuti e soprattutto lo stato attuale non viene modificato, di conseguenza la sua esecuzione può continuare normalmente.

Per l'esempio del robot per l'esplorazione di Marte il risultato di questo processo è illustrato nella figura 2.5: gli stati colorati di verde sono gli stati aggiornabili, e da essi partono le ϵ -transition verso i corrispondenti stati del nuovo controller.

2.6 Il Lego Mindstorm NXT Robot System

Il Lego Mindstorm NXT è un kit per la costruzione di robot a scopo ludico e didattico costituito da un componente principale programmabile (d'ora in poi chiamato mattone NXT), dei sensori di vario tipo, alcuni motori, e da varie componenti meccaniche per la costruzione dei meccanismi e della struttura del robot. Il mattone NXT è essenzialmente un computer con delle potenzialità molto ridotte, le cui caratteristiche sono riassunte in tabella 2.3

CPU principale	ARM 32bit @ 48 MHz (AT91SAM7S256)
Coprocessore	AVR 8bit @ 8 MHz (ATmega48)
RAM	64 Kb
Flash	256 Kb
Connessioni	Bluetooth USB 2.0 4 per sensori Lego NXT (connessione proprietaria) 3 per motori Lego NXT (connessione proprietaria)
Input	4 pulsanti
Output	Display LCD b/n 100x64 pixel Speaker

Tabella 2.3: Caratteristiche hardware del sistema Lego NXT



Figura 2.6: Il componente principale del sistema Lego NXT con alcuni sensori e motori collegati

Il kit include tre motori e diversi tipi di sensori che possono essere collegati al mattone NXT attraverso la connessione proprietaria: sensore di contatto, sensore di luminosità, sensore di intensità sonora, e sensore di distanza ad ultrasuoni. I motori sono in realtà dei servomeccanismi in grado di garantire una precisione di un grado.

La configurazione software originale prevede che sul mattone NXT venga eseguito un sistema operativo proprietario su cui vengono caricati dei programmi sviluppati in NXT-G, un linguaggio di programmazione grafico piuttosto semplice e non adatto alle necessità del lavoro da svolgere in questa tesi.

Per questa ragione è stato utilizzato LeJOS, un sistema alternativo costituito da una macchina virtuale Java semplificata ed adattata per essere eseguita sul mattone NXT (nonostante le caratteristiche hardware molto ridotte), ed un insieme di API che facilitano l'utilizzo dei sensori, dei motori, e delle altre caratteristiche del mattone NXT.

Capitolo 3

Analisi del problema

In questa sezione verranno analizzati in dettaglio i vari aspetti del problema in modo da evidenziare i requisiti relativi a ciascuno di essi. Gli eventi, essendo centrali nel funzionamento del sistema ed alla base degli aspetti trattati successivamente, saranno i primi ad essere analizzati. In seguito ci si concentrerà sui requisiti relativi all'interpretazione e all'aggiornamento dinamico del controller, ed infine alla comunicazione con il computer remoto che invia gli aggiornamenti e viene usato per gestire il sistema.

3.1 Gli eventi

Gli eventi rappresentati nelle specifiche MSD illustrate nella sezione 2.4, le capacità di interazione messe a disposizione da un certo hardware illustrate tramite un esempio nella sezione 2.2, e la nozione di evento introdotta nella formalizzazione del problema presentata nella sezione 2.3 rappresentano in realtà la stessa entità, ovvero l'interazione che è alla base del funzionamento di un reactive system, e sono quindi strettamente legati. La relazione che lega queste tre facce dello stesso aspetto del problema è il primo elemento del sistema che deve essere analizzato, dato che tutto il resto della trattazione si fonda sul concetto di evento, su come l'evento

deve essere gestito, rappresentato in memoria, inviato all'ambiente, e via dicendo.

In questa sezione quindi si valuterà dapprima qual è l'alfabeto degli eventi in relazione ad un qualsiasi problema, e successivamente quali sono i requisiti associati al software che mappa gli eventi dell'alfabeto in operazioni per sensori ed attuatori.

3.1.1 L'alfabeto degli eventi

L'alfabeto degli eventi è di fondamentale importanza perché il sistema che si vuole sviluppare deve essere in grado di gestire, memorizzare, inviare, ed effettuare operazioni di altro genere su tutti i possibili eventi che si possono presentare nell'ambito di un problema. Partendo quindi dalla considerazione che la rappresentazione delle specifiche di funzionamento di un sistema espresse tramite MSD non può essere considerata sufficiente a definire l'alfabeto degli eventi, dato che per definizione gli MSD possono venire modificati anche aggiungendo nuovi tipi di messaggi, è necessario individuare un altro modo per definire tale insieme.

La prima cosa da osservare è che dato che un sensore di per sé non può sempre essere considerato un reactive system l'insieme degli eventi che può essere ricevuto da un certo tipo di sensore dipende essenzialmente dal software che lo controlla. Un sensore di distanza, ad esempio, può scatenare un evento in diversi casi: quando la distanza supera una soglia, oppure quando va sotto una soglia, oppure quando la velocità di avvicinamento supera o scende sotto un certo valore. Questo esempio del sensore di distanza evidenzia anche che una singola tipologia di evento possa prevedere dei parametri (il valore soglia che fa scattare l'evento) a causa dei quali i possibili eventi associati ad una certa tipologia di sensore possano essere addirittura infiniti. Tutte queste considerazioni possono inoltre essere facilmente estese anche agli attuatori.

Per quanto riguarda i messaggi scambiati con altri reactive system, non è possibile porre un limite all'insieme dei possibili eventi,

ed è quindi necessario ipotizzare che il sistema possa scambiare messaggi caratterizzati da qualsiasi nome, con qualsiasi parametro, verso un qualsiasi oggetto del sistema.

Queste considerazioni definiscono dei requisiti che dovranno essere soddisfatti dall'implementazione del sistema e che influenzeranno aspetti come la rappresentazione in memoria o il formato utilizzato per trasmettere queste informazioni.

3.1.2 Requisiti relativi al software che gestisce gli eventi

Spesso l'associazione tra i sensori e gli eventi deve tenere in considerazione alcuni aspetti inerenti l'hardware. Prendiamo in considerazione un semplice sensore di contatto. Questo tipo di sensore è generalmente realizzato tramite un interruttore meccanico (ed infatti anche nel kit Lego NXT è così), ma gli interruttori meccanici sono spesso afflitti dal problema del *contact bounce*: essendo costituiti da parti meccaniche, quando queste si muovono il loro momento e l'elasticità dei materiali fanno sì che prima di raggiungere una situazione di contatto stabile vi possano essere diverse oscillazioni tra lo stato di contatto aperto e contatto chiuso. Pur trattandosi di un transitorio estremamente breve è comunque sufficientemente lungo da poter essere osservato via software, che quindi (se ogni passaggio del circuito da aperto a chiuso viene considerato un evento) potrebbe presentare un comportamento erraneo perché a fronte di una sola pressione del sensore ad essa corrisponderà più di un evento, essendo facile immaginare condizioni in cui questo può produrre problemi. Nel caso del lego mindstorm questo fenomeno è amplificato dal fatto che i sensori di contatto sono collegati a strutture plastiche piuttosto elastiche di massa non trascurabile.

Una soluzione a questo problema consiste nel generare l'evento solamente dopo che il circuito si è stabilizzato per un certo intervallo di tempo, sufficiente ad esaurire l'eventuale transitorio. Nel caso del sistema utilizzato in questo lavoro, il problema deve essere risolto

a livello software programmando opportunamente la routine che si occupa di lanciare un evento a seconda dell'input dei sensori.

A queste considerazioni bisogna aggiungere il fatto che in alcuni casi potrebbero essere necessari degli eventi complessi che coinvolgono più sensori: ad esempio per migliorare l'affidabilità di un certo sensore potrebbe essere utilizzato un sistema ridondante composto in realtà da diversi sensori ed un software che decide in quali casi scatenare l'evento ed in quali no.

Le associazioni tra gli eventi e le informazioni fornite dai sensori possono quindi essere molto più complicate e variegate di quanto si possa inizialmente immaginare, ed è quindi evidente come esse debbano essere gestite da uno strato software estendibile e configurabile in grado di:

- Riconoscere, per ogni tipologia di sensore, i diversi eventi che esso può generare (che possono addirittura essere infiniti, se gli eventi sono parametrici)
- Effettuare tutte le elaborazioni necessarie sulle informazioni grezze fornite dai sensori.

3.1.3 Controllable event, uncontrollable event

Una distinzione importante che ricorrerà in tutto il resto della tesi è quella tra gli eventi che sono provocati dal sistema e quelli provocati dall'ambiente.

I *controllable event* sono gli eventi che corrispondono a messaggi inviati dagli oggetti di sistema verso l'ambiente, ad esempio l'impostazione della velocità di un motore oppure l'invio di un messaggio ad un oggetto che fa parte dell'ambiente. Le transizioni del controllore che danno luogo a tale tipo di evento sono chiamate *controllable transition*.

Analogamente, gli *uncontrollable event* sono gli eventi che corrispondono a messaggi inviati dall'ambiente verso il sistema, ad esempio la pressione di un sensore di contatto o la ricezione di un

messaggio proveniente da un oggetto dell'ambiente. Le transizioni del controllore associate ad eventi di questo tipo sono chiamate *uncontrollable transition*.

Controllable e uncontrollable event possono essere entrambe ulteriormente separati secondo la distinzione proposta nella prossima sezione.

3.1.4 Eventi del robot, eventi delle specifiche

L'esempio utilizzato nella sezione 3.1.2 è utile per evidenziare un fatto significativo nella definizione delle specifiche: esse in genere sono espresse in termini di eventi di più "alto livello" rispetto agli eventi che sono legati ai sensori ed agli attuatori, e questo viene fatto per diverse ragioni.

La ragione più intuitiva è che in questo modo le specifiche diventano più facili da comprendere e modificare qualora ve ne sia la necessità, tuttavia va anche osservato che in ogni problema esistono degli aspetti fondamentali che difficilmente sono soggetti ad evoluzione e che si ripetono spesso nel problema, ed appare quindi sensato considerarli come sottointesi e risolti ad un livello inferiore. Ad esempio in un sistema mecatronico come il robot citato nell'esempio del capitolo precedente, è verosimile che nelle specifiche ricorra spesso un evento in uscita che sta ad indicare che il robot deve fermarsi, evento che corrisponde all'impostazione a zero della velocità di tutte le ruote. Un'ulteriore ragione è che il processo di sintesi del controllore presenta una complessità computazionale elevata, e di conseguenza la semplificazione delle specifiche oltre a determinare la sintesi di un controllore con un numero inferiore di stati (che già di per sé è vantaggioso in un contesto limitato come quello del Lego NXT), abbrevia il processo di sintesi e consente di affrontare problemi più complessi. La complessità temporale del problema, infatti, è tale che l'applicazione dell'approccio alla sintesi automatica a partire dalle specifiche descritta nella sezione 2.5 è tutt'ora una sfida notevole.

È necessario quindi che il sistema sviluppato tenga in considerazione queste problematiche e permetta quindi di definire una corrispondenza tra degli eventi di alto livello utilizzati nelle specifiche e gli eventi di basso livello legati ai sensori e gli attuatori collegati al robot. Anche se gli eventi di alto livello generalmente nascondono degli aspetti statici del sistema è comunque auspicabile che l'associazione tra i due tipi di eventi possa essere modificata dinamicamente a runtime.

Nel resto di questa tesi questi due tipi di eventi verranno chiamati *eventi del robot* ed *eventi delle specifiche*.

3.1.5 Eventi legati alla comunicazione con altri sistemi

Oltre agli eventi che dipendono da sensori ed attuatori, il sistema deve prevedere la possibilità di inviare eventi diretti verso altri sistemi e riceverne di provenienti da altri sistemi. Un esempio di messaggio di questo tipo è la richiesta di autorizzazione all'avvicinamento alla base di ricarica presente nelle specifiche del problema d'esempio presentato nel capitolo precedente.

Questa categoria di eventi è particolare perché non essendo legata ai sensori ed agli attuatori non è possibile porre a priori dei vincoli all'insieme dei possibili messaggi scambiati durante il funzionamento del sistema, e questo fatto deve ovviamente riflettersi nel modo in cui il sistema di gestione è realizzato. La restrizione che può essere fatta considerando quali sono gli eventi scambiati con altri sistemi presenti nelle specifiche infatti non può essere considerata sufficiente perché le specifiche possono variare nel tempo. Questo costituisce un requisito che deve essere tenuto in considerazione nello sviluppo del sistema.

3.1.6 La modellazione del tempo

I reactive system devono spesso reagire al passaggio del tempo, ed infatti anche utilizzando il sistema Lego NXT e l'approccio proposto in questa tesi per costruire dei semplici robot di prova ci si è trovati

a formulare frequentemente requisiti come "dopo dieci secondi il robot deve fermarsi".

Una prima ipotesi di soluzione consiste nell'inserire nel controller delle transizioni che richiedono un certo tempo per essere eseguite. Per quanto la semplice realizzabilità di questo approccio lo renda attraente, esso è essenzialmente sbagliato perché è in contrasto con la synchrony hypothesis [17], un'astrazione generalmente utilizzata nello sviluppo di reactive systems (e non solo), secondo la quale il tempo scorre solamente tra due eventi, ed in un intervallo di tempo finito il sistema può compiere un insieme infinito di operazioni.

Il modo corretto per modellare il passaggio del tempo consiste invece nell'utilizzo di timer che periodicamente generano degli eventi che rappresentano il passaggio di una certa unità di tempo. Questi eventi possono quindi essere utilizzati nelle specifiche (quindi anche nei diagrammi MSD) per definire requisiti che coinvolgono il passaggio del tempo.

3.2 L'interpretazione e l'aggiornamento del controller

Nella sezione 2.3 sono state presentate le definizioni degli elementi che definiscono a livello puramente teorico molti aspetti del sistema. Questa sezione si concentrerà su alcuni di essi relativi all'interpretazione e l'aggiornamento del controllore. Verranno anche valutati i requisiti relativi al processo di eliminazione della vecchia versione del controllore, che avviene dopo che la transizione alla nuova versione è avvenuta, un aspetto che invece è interamente nuovo perché non preso in considerazione nella teoria utilizzata.

3.2.1 La struttura del controller

Dalla sezione 2.3 si possono evidenziare le caratteristiche del controller che deve essere eseguito:

- Il controllore è costituito da un automa a stati finiti con uno stato iniziale e senza stati finali. Tra gli stati possono esistere transizioni di tre tipi:
 - ϵ -transition
 - Controllable transition, associate ad eventi in uscita
 - Uncontrollable transition, associate ad eventi in ingresso
- Da ogni stato del controllore non può uscire più di una ϵ -transition, poiché le tale tipo di transizione esiste solo in qualità di update transition, e per costruzione ve ne può essere una sola per ogni stato.
- Se da uno stato parte una ϵ -transition, da esso non partono altre transizioni, perché l'aggiunta di una update transition è accompagnata dall'eliminazione delle vecchie transizioni.

3.2.2 Esecuzione del controller

Le caratteristiche dell'esecuzione del controller, derivate in larga parte dalla formalizzazione teorica del problema indicata in precedenza, sono di seguito elencate:

- L'esecuzione del controller deve partire dallo stato indicato come iniziale, e progredisce inviando un evento quando viene seguita una controllable transition, e ricevendo un evento quando viene seguita una uncontrollable transition.
- Se da uno stato parte solo una controllable transition, il sistema esegue la transizione inviando l'evento associato ad essa.
- Se da uno stato partono più controllable transition, il sistema sceglie in maniera non deterministica quale delle transizioni seguire. Tutte le possibilità garantiscono che le specifiche di funzionamento del sistema vengano soddisfatte.
- Se da uno stato parte solo una uncontrollable transition, il sistema attende di ricevere l'evento associato a tale transizione.

- Se da uno stato partono più uncontrollable transition il sistema deve rimanere in tale stato fin quando non viene ricevuto uno degli eventi associati alle transizioni presenti. Quando questo accade il sistema deve seguire la transizione associata all'evento ricevuto.
- Avendo assunto che le azioni compiute dal sistema siano istantanee, e che il sistema sia sufficientemente veloce da eseguire un insieme infinito di transizioni prima dell'arrivo del prossimo evento, se da uno stato escono sia controllabile che uncontrollable transition, il sistema esegue una delle controllabile transition.

3.2.3 Aggiornamento del controller

Il requisito più importante relativamente all'operazione di aggiornamento del controller è già stato evidenziato nella descrizione teorica del problema, e consiste nel fatto che le operazioni di aggiornamento non devono interferire con l'esecuzione del controller.

Dalla teoria inoltre si deduce che l'aggiornamento consiste nell'aggiungere il controller che soddisfa le nuove specifiche, ed aggiungere le update transition che partono dagli stati aggiornabili del vecchio controller ai corrispondenti stati del nuovo controller. Appare quindi ragionevole considerare l'aggiornamento come l'esecuzione di una sequenza di operazioni di questi tipi:

- Aggiunta di un nuovo stato al controller
- Aggiunta di una transizione (controllabile, uncontrollable o update)

In un sistema reale, dato che tali operazioni non possono essere considerate istantanee, l'ordine con cui queste vengono eseguite deve essere tenuto in considerazione assicurandosi che in nessun caso la sequenza delle operazioni eseguite lasci il controllore in una situazione inconsistente, anche quando la sequenza di operazioni è stata eseguita solo parzialmente.

Intuitivamente questo requisito si tradurrà nell'eseguire le operazioni di aggiornamento seguendo il seguente ordine:

- Aggiunta dei nuovi stati
- Aggiunta delle transizioni tra i nuovi stati
- Aggiunta delle update transition dagli stati del vecchio controllore ai corrispondenti stati del nuovo controllore

3.2.4 Eliminazione del vecchio controller

L'eliminazione del vecchio controllore è un'operazione che in un ambiente reale deve necessariamente essere realizzata, ed il requisito principale relativo ad essa è che non deve interferire con l'esecuzione del controllore attuale. Essa è possibile perché dopo che l'esecuzione passa al nuovo controllore, la vecchia versione diventa completamente inutile, dato che non esistono transizioni dagli stati della nuova versione agli stati della vecchia.

Un ulteriore requisito è che l'eliminazione avvenga appena possibile, in modo da liberare le risorse occupate dal vecchio controller. Questo è particolarmente significativo visti i vincoli estremamente stringenti che derivano dalla piattaforma hardware utilizzata.

La teoria di riferimento utilizzata nell'ambito di questo lavoro non prende in considerazione la problematica dell'eliminazione del vecchio controllore, di conseguenza a questo proposito non vi sono altri requisiti da evidenziare.

3.3 Il sistema di comunicazione

Nell'ambito del problema che si sta affrontando, il sistema di comunicazione ha diversi ruoli.

- Consentire lo scambio di messaggi con altri sistemi.
- Supportare il processo di aggiornamento dinamico, quindi mentre il controllore è in esecuzione deve poter ricevere in qualsiasi

si momento un nuovo controllore, e confermare se l'aggiornamento ha avuto buon fine.

- Fornire le informazioni che possono essere utili per verificare lo stato del problema, capire la ragione di eventuali comportamenti anomali, ed in generale aiutare ad individuare e correggere eventuali errori.

Relativamente ad esso è possibile evidenziare i seguenti requisiti:

- Per lasciare la massima libertà al robot è opportuno che il collegamento avvenga tramite senza fili, quindi nel caso del Lego Mindstorm NXT il collegamento deve avvenire tramite Bluetooth.
- In qualsiasi momento il computer preposto deve potersi collegare al robot ed inviare un nuovo controllore, ed il robot deve rispondere confermando o meno il corretto aggiornamento del controller.
- È opportuno che il sistema fornisca la possibilità di accedere alle seguenti informazioni utili ai fini di debug:
 - Il controllore attualmente in esecuzione.
 - Lo stato in cui attualmente si trova l'esecuzione del controllore.
 - Un log contenente la traccia dell'esecuzione fino all'istante corrente, che includa gli eventi ricevuti ed inviati, le transizioni eseguite, ed eventuali errori avvenuti durante l'esecuzione.
 - Quantità di memoria disponibile

Capitolo 4

Architettura della soluzione

4.1 Introduzione

L'idea alla base del funzionamento dell'architettura consiste nella suddivisione tra più thread dei compiti evidenziati durante l'analisi del problema e nella rappresentazione in memoria del controller utilizzando una struttura dati a cui i vari thread possono accedere secondo necessità. In particolare verranno evidenziati due thread principali:

- L'*executor thread*, il cui ruolo principale è l'esecuzione del controller.
- Il *deployer thread*, il cui ruolo principale consiste nell'esecuzione dell'aggiornamento del controller.

Quest'idea di base è rappresentata nella figura 4.1: dovendo eseguire il controller, l'*executor thread* ha ovviamente accesso alla struttura dati che lo rappresenta, ed inoltre invia e riceve messaggi sia utilizzando direttamente i sensori e gli attuatori, sia comunicando con altri sistemi. Il *deployer* resta in attesa della nuova versione del controllore, che gli viene inviata da un computer remoto, ed ha

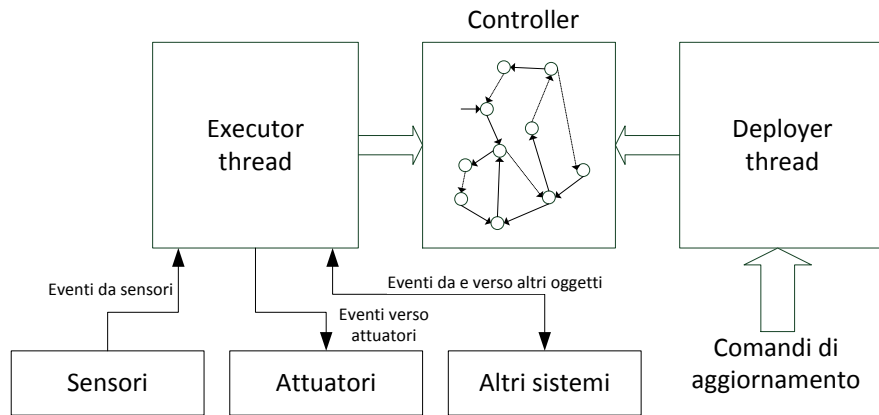


Figura 4.1: L'idea alla base dell'architettura

quindi accesso alla struttura dati che rappresenta il controller per poter installare la nuova versione.

Per adattarsi alle ridotte capacità computazionali offerte dal sistema Lego NXT è opportuno demandare al computer remoto che elabora ed invia la nuova versione del controllore la maggior parte delle operazioni di calcolo che devono essere eseguite per aggiornare il controllore. Dall'analisi del problema effettuata nel capitolo precedente è emerso che la trasformazione del controller attuale nel dynamically updating controller consiste in una sequenza di semplici operazioni di modifica che devono essere applicate al controller attuale. Nell'architettura da sviluppare sul sistema Lego NXT l'elaborazione della sequenza di comandi è completamente demandata al computer remoto, ed il ruolo del mattone NXT è di mera esecuzione.

Questo comporta che:

- Il computer remoto che calcola la sequenza di comandi che realizza l'aggiornamento deve essere a conoscenza del controller in esecuzione sul robot, in modo da identificare correttamente gli stati da cui devono partire le update transition. Se il controller in esecuzione non corrisponde a quello noto al computer il robot andrà in contro ad una situazione di errore.

- Tutti i controlli necessari a garantire la correttezza delle operazioni devono essere fatti preventivamente dal computer che elabora la sequenza di comandi. Se il computer invia al robot un comando che, ad esempio, inserisce nel controllore in esecuzione una transizione verso uno stato inesistente, il robot, eseguendo alla cieca il comando ricevuto, andrà in contro ad una situazione di errore.

Una delle caratteristiche del problema è quella di prevedere la possibilità che possano avvenire scambi di messaggi anche con altri oggetti che, pur essendo considerati parte dell'ambiente, sono di fatto degli altri sistemi con cui si comunica direttamente tramite scambio di messaggi. L'esempio del robot per l'esplorazione di Marte ne è un esempio, il robot infatti comunica con la base di ricarica (che è considerata parte dell'ambiente) tramite uno scambio di messaggi.

Il sistema che si vuole utilizzare per la realizzazione pratica dell'infrastruttura dispone soltanto di due canali di comunicazione: un collegamento Bluetooth ed uno USB. Scartato il sistema USB per le limitazioni che derivano dall'utilizzo di cavi, rimane disponibile solo il collegamento Bluetooth, che però deve essere utilizzato per la comunicazione con il computer remoto che invia le nuove versioni del controller e verifica lo stato del robot.

La soluzione consiste quindi nell'utilizzare il computer remoto anche come "gateway" per tutti gli scambi di messaggi con altri sistemi, nel senso che tutti i messaggi inviati dal robot vengono ricevuti dal computer remoto, ed è compito di quest'ultimo inoltrarli ai sistemi di destinazione, ed analogamente tutti i messaggi destinati al robot sono inviati dal computer remoto, che li ha eventualmente ricevuti da fonti diverse.

4.2 Il sistema di gestione degli eventi

Prima di discutere i dettagli del sistema di gestione degli eventi è necessario introdurre delle distinzioni che riguardano le diverse ti-

pologie di eventi che il sistema deve gestire. Le diverse categorie qui individuate saranno utilizzate in tutto il resto di questa tesi. Dalla formalizzazione del problema e dall'analisi dei requisiti si possono individuare due tipi di eventi:

- Controllable event, che corrispondono all'esecuzione di una controllable transition e quindi un evento in uscita (ad esempio l'impostazione della velocità di un motore, o l'invio di un messaggio ad un altro sistema).
- Uncontrollable event, che corrispondono all'esecuzione di una uncontrollable transition e quindi un evento in ingresso (ad esempio la segnalazione che un pulsante è stato premuto, o la ricezione di un messaggio proveniente da un altro sistema).

Una ulteriore distinzione che è opportuno ricordare è quella tra:

- Robot event, ovvero gli eventi di basso livello legati a sensori, attuatori, ed alla comunicazione con altri sistemi
- Problem event, ovvero gli eventi di alto livello utilizzati nella definizione delle specifiche

4.2.1 Gestione degli uncontrollable event

Per rappresentare tutti gli eventi che possono essere ricevuti dal robot è necessario creare una classe astratta che sarà chiamata `UncontrollableEvent` da cui derivano tutte le sottoclassi che rappresentano ciascuno dei possibili tipi di evento. La figura 4.2 mostra ad esempio la gerarchia delle classi che rappresentano tutti i possibili uncontrollable event nell'implementazione per il sistema Lego NXT.

Dall'analisi dei requisiti svolta in precedenza, si rileva che l'insieme di eventi che devono essere gestiti può essere molto vario. Si è quindi deciso di sviluppare un componente unico, chiamato `UncontrollableEventManager`, in cui è unificata la gestione di tutte le varie tipologie di sorgenti di uncontrollable event e la traduzione

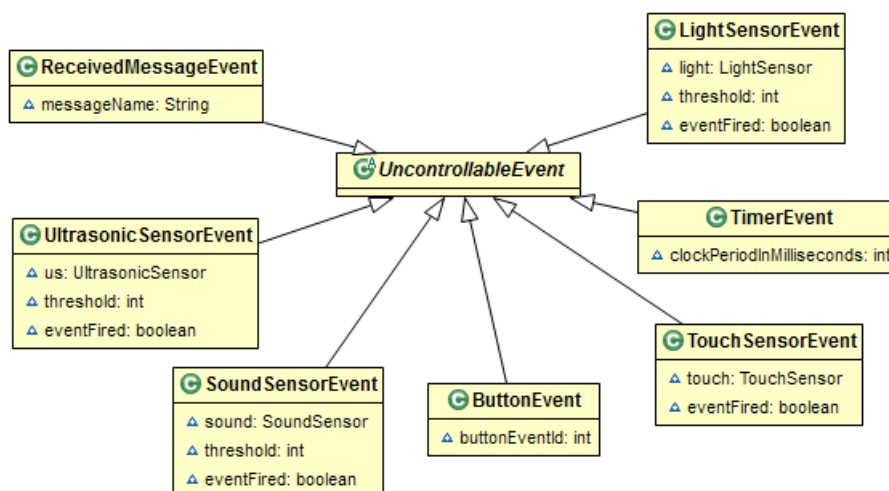


Figura 4.2: Le classi che rappresentano gli uncontrollable event nel sistema Lego NXT

tra uncontrollable robot event ed uncontrollable problem event. Sarà questo componente ad offrire all'executor thread la possibilità di attendere l'arrivo di un evento.

L'insieme degli eventi che deve essere gestito da questo componente è ovviamente variabile a seconda del problema, ed a seconda di come le specifiche evolvono nel tempo. Di conseguenza tale componente, dato un insieme qualunque di UncontrollableEvent, deve essere in grado di eseguire tutte le operazioni che sono necessarie per predisporre il sistema a ricevere eventi di tale tipo su una particolare architettura. Ad esempio se è necessario l'utilizzo di un uncontrollable robot event che si verifica quando la distanza misurata da sensore di distanza scende sotto una certa soglia, l'UncontrollableEventManager dovrà inizializzare il sensore, predisporre un thread che legga periodicamente la distanza misurata, e di volta in volta elaborare la distanza per determinare se l'evento si è verificato oppure no, ed infine (qualora la condizione sia verificata) segnalare all'executor thread che si è verificato l'evento.

L'UncontrollableEventManager ha anche il compito di effettuare la traduzione tra uncontrollable robot event e uncontrollable problem event, ed a questo scopo si è deciso di utilizzare una traduzio-

ne basata su automi a stati finiti. Per ogni uncontrollable problem event esiste un automa le cui transizioni sono costituite da uncontrollable robot events, e sono associate ad un valore vero o falso che indica se l'esecuzione di tale transizione comporta la generazione del problem event associato a quell'automa. Ogni volta che l'UncontrollableEventManager rileva un robot event in ingresso lo propaga a tutti gli automi di traduzione, ed alcuni di questi genereranno un problem event. L'architettura presentata in questa tesi prevede che la scelta degli uncontrollable robot event verso i quali il sistema deve restare in ascolto, e la definizione degli automi che traducono gli eventi ricevuti producendo problem event, vengano predisposti nella fase di configurazione iniziale del sistema, prima che inizi l'esecuzione del controllore. Questo significa che deve essere cura di chi utilizza questa infrastruttura per risolvere uno specifico problema individuare tutti gli eventi che possono essere significativi, anche se non sono immediatamente utilizzati, tenendo conto di possibili evoluzioni future delle specifiche di funzionamento.

Questo limite deriva dal fatto che supportare il cambiamento dinamico degli uncontrollable robot event pone due problematiche:

- Per ogni inserimento di un nuovo robot event è necessario eseguire a runtime la fase di inizializzazione necessaria a predisporre sua ricezione. Questa operazione in alcuni è piuttosto onerosa, e quindi non adatta ad essere eseguita su una piattaforma hardware dalle potenzialità ridotte come quella utilizzata in questo lavoro.
- L'architettura del sistema viene notevolmente complicata.

L'UncontrollableEventManager include anche una coda FIFO bloccante nella quale vengono messi in coda tutti gli uncontrollable problem events generati. Questa scelta è giustificata da due ragioni:

- A causa del meccanismo di traduzione robot/problem event è possibile che l'arrivo di un unico robot event generi più problem event contemporaneamente. Siccome in nessun caso è

ammissibile che l'esecuzione del controller "perda" degli eventi è necessario fare in modo che questo non accada, ed inserirli in una coda da cui l'esecuzione del controller può prenderli è una soluzione.

- Il funzionamento del sistema si basa sulla ipotesi di sincronia [17], secondo la quale l'invio di un messaggio non consuma tempo, e quindi nel tempo che intercorre tra la ricezione di due eventi il sistema può eseguire infinite operazioni. Ovviamente questa ipotesi non può sussistere in un sistema reale, in cui l'invio di un evento richiede un certo tempo, ed è quindi necessario prendere in considerazione l'ipotesi che il sistema possa ricevere un evento mentre è impegnato ad inviare una sequenza (eventualmente molto lunga) di eventi.

La scelta di utilizzare una coda bloccante risulta particolarmente comoda nell'ottica dell'esecuzione in un ambiente a più thread:

- I vari sottosistemi dell'UncontrollableEventManager che gestiscono le varie sorgenti di eventi ed il sistema di traduzione fanno parte del vengono gestiti nei rispettivi thread e da essi aggiungono eventi alla coda.
- Il thread di esecuzione "consuma" gli eventi aggiunti alla coda se ve ne sono di disponibili, altrimenti viene bloccato fin quando non arriva un nuovo uncontrollable problem event.

L'implementazione dell'UncontrollableEventManager dovrà essere strettamente legata all'insieme dei possibili eventi che possono essere inviati e ricevuti. Tale insieme è definito da tutte le possibili istanze delle sottoclassi della classe UncontrollableEvent precedentemente citata, e quindi l'UncontrollableEventManager deve essere in grado di individuare il verificarsi degli eventi comunicando con l'hardware dei sensori, con il sistema di comunicazione, o con le altre fonti di eventi (ad esempio i timer). Compito di questa componente del sistema è anche la soddisfazione dei requisiti individuati nella sezione 3.1.2

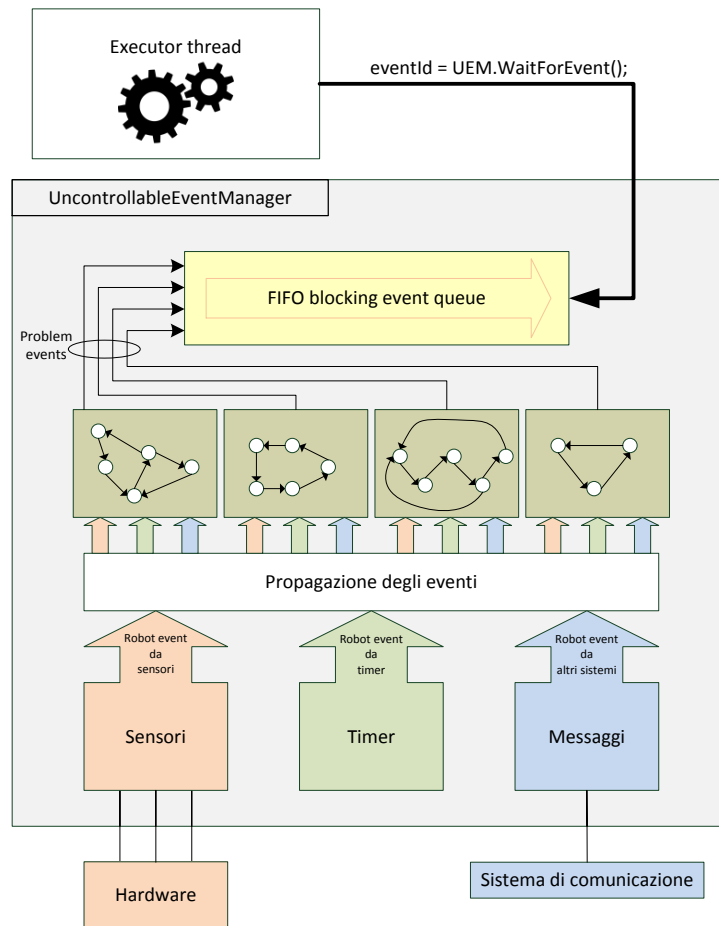


Figura 4.3: La struttura dell'UncontrollableEventManager

Anche la soddisfazione dei requisiti compito dell'implementazione del riconoscimento degli eventi presente nell'UncontrollableEventManager.

La figura 4.3 rappresenta gli aspetti finora descritti dell'UncontrollableEventManager.

4.2.2 Gestione dei controllable event

I controllable events sono eventi che scaturiscono dall'esecuzione del controller, e pertanto, vista la loro natura diversa, l'architettura di gestione di questi eventi è completamente diversa rispetto a quel-

la degli uncontrollable events. In questa sezione se ne vedranno le caratteristiche.

Un aspetto da tenere in considerazione, che ha avuto un impatto notevole sulla progettazione del sistema di gestione di questi eventi, consiste nel fatto che questo tipo di eventi corrisponde semplicemente all'esecuzione di certe operazioni, quindi non è necessaria la fase di setup necessaria nel caso della ricezione degli eventi. La conseguenza è che i controllable events possono essere liberamente modificati a runtime.

Anche nel caso dei controllable events sussiste la differenziazione tra i problem event di più alto livello ed i robot event di più basso livello. In questo caso però ad ogni controllable problem event corrisponde semplicemente una sequenza di controllable robot event.

L'esecuzione dei controllable event in particolare è completamente demandata all'executor thread: ogni volta che, durante l'esecuzione del controller, viene eseguita una transizione corrispondente ad un controllable problem event, l'executor esegue la sequenza dei controllable robot event associata a tale evento del problema.

4.3 Rappresentazione del controller

Il controller è l'elemento centrale del sistema ed in questa sezione verrà descritta la sua architettura. Dalla precedente analisi dei requisiti e dall'idea alla base dell'architettura del sistema descritta all'inizio di questo capitolo, si deduce la necessità che il controller sia rappresentato in modo da poter essere letto e modificato da più thread contemporaneamente, e sia possibile modificarlo con un impegno computazionale ridotto in modo che le operazioni di modifica non interferiscano con l'esecuzione del controller.

Per la rappresentazione del controller sono state quindi prese in considerazione diverse possibilità.

La prima idea valutata consiste nella sua rappresentazione per mezzo di una matrice di transizione unica, una struttura dati estremamente semplice da realizzare ed utilizzare, ma che purtroppo

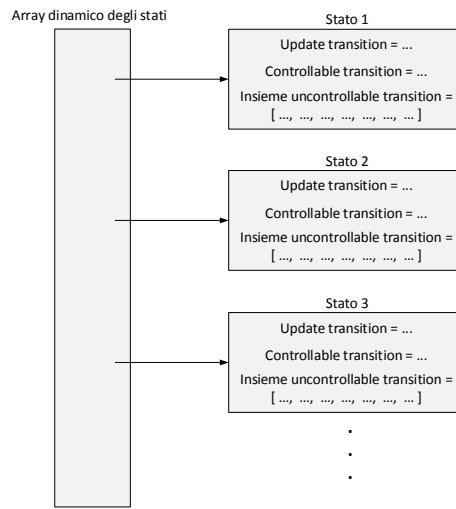


Figura 4.4: Struttura da utilizzare nella rappresentazione del controllore

presenta delle caratteristiche che sono in conflitto con i requisiti. Modificare tale struttura per aggiungere o eliminare nuovi stati, infatti, nella maggior parte delle architetture richiede di creare una nuova matrice di dimensione differente e copiare il contenuto (o parte di esso) della vecchia matrice in quella nuova, un'operazione onerosa dal punto di vista computazionale, con una complessità proporzionale alla dimensione del controller, durante la quale l'executor thread è obbligato ad attendere fin quando tale operazione non è conclusa.

Per questa ragione la struttura della rappresentazione del controllore è suddivisa per stati, e l'insieme degli stati è memorizzato in una struttura la cui dimensione può essere modificata con uno sforzo computazionale ridotto. La scelta della particolare struttura da utilizzare viene lasciata come dettaglio dell'implementazione, ma in generale strutture come liste concatenate, tabelle hash o altri tipi di array dinamico possono essere adeguate allo scopo.

Dall'analisi dei requisiti si determina che ogni stato deve poter contenere:

- Una update transition, eventualmente vuota. L'analisi dei requisiti aveva infatti evidenziato che da uno stato può partire al

più una update transition.

- Una controllable transition, eventualmente vuota. L'analisi dei requisiti aveva evidenziato che da uno stato possono partire più uncontrollable transition, e che in tal caso l'esecuzione può proseguire lungo una qualunque delle alternative, poiché in ogni caso è garantito il rispetto delle specifiche, tuttavia per semplificare il controllore tale scelta viene fatta a priori, durante la procedura di aggiornamento dinamico.
- Un array di lunghezza pari al numero di problem event definiti nell'UncontrollableEventManager, in cui ogni elemento corrisponde allo stato di destinazione della transizione che corrisponde all'evento del problema, se tale transizione esiste. Questo array permette di rappresentare tutte le possibili uncontrollable transition.

La figura 4.4 rappresenta in maniera molto schematica la struttura dati risultante.

4.3.1 Gestione dell'accesso concorrente

L'operazione di trasformazione del controller attuale nel dynamically updating controller descritto nella sezione 2.5.3 include anche delle operazioni sugli stati del controller attuale, più precisamente l'aggiunta delle update transition verso i corrispondenti stati del nuovo controller. Non essendo possibile escludere che questo accada proprio mentre lo stato attuale del controller è lo stato che il deployer thread sta modificando, è necessario prevenire il verificarsi di race conditions.

La soluzione adottata è la seguente: ad ogni stato del controller è associato ad un monitor che garantisce la mutua esclusione. Ogni qual volta l'esecuzione del controller entra in uno stato l'executor thread deve acquisire il monitor associato allo stato prima di potervi entrare, e non lo rilascerà fin quando l'esecuzione non uscirà da

quello stato. Ogni qual volta il `deployer thread` deve eseguire un'operazione su uno stato (inclusa l'aggiunta o la rimozione di transizioni) prima di poter eseguire qualsiasi operazione deve prima acquisire il `monitor`.

Questo meccanismo di `locking` a livello di stato consente al `deployer thread` di eseguire operazioni di aggiornamento su qualsiasi stato tranne lo stato corrente, che è mantenuto bloccato dall'`executor thread`. Quando un'operazione di `update` è richiesta sullo stato corrente, il `deployer thread` attenderà l'uscita dell'`executor thread` da tale stato, e poi eseguirà l'operazione richiesta.

Questo approccio garantisce che, se vengono soddisfatti i requisiti relativi all'ordine delle operazioni evidenziati in precedenza, l'esecuzione del controller non venga influenzata dal processo di aggiornamento, e che il controller in esecuzione sia sempre valido.

Immaginiamo un caso particolarmente sfortunato in cui l'`executor thread` sta per eseguire una transizione verso uno stato che il `deployer thread` sta attualmente modificando per aggiungervi una `update transition`. In questo caso il thread di esecuzione dovrà necessariamente attendere che il `deployer` completi l'operazione, ma grazie al design della struttura dati contenente il controller questa operazione si riduce ad una semplice scrittura di una variabile, un'operazione che provoca un ritardo certamente trascurabile. A conclusione di quest'operazione inoltre, abbiamo la certezza che l'`update transition` conduca l'esecuzione verso uno stato del nuovo controller, che era già stato completamente installato nel sistema in precedenza.

Per il corretto funzionamento del sistema è necessario che anche la struttura dati che contiene l'insieme degli stati possa essere letta e modificata senza provocare `race conditions`, infatti da un lato le operazioni di aggiornamento del controller eseguite dal `deployer thread` aggiungono anche dei nuovi stati, dall'altro le operazioni di esecuzione del controller includono l'accesso a tale struttura ad ogni transizione per accedere al nuovo stato.

4.4 Executor thread

Come già evidenziato, l'executor thread ha il compito di eseguire il controllore.

Per definire l'algoritmo di esecuzione del controller è utile stabilire una priorità con cui l'executor thread valuta le varie tipologie di transizione che partono da uno stato. Dalle specifiche individuate in precedenza, è emerso che l'aggiunta di una update transition al controllore è accompagnata dall'eliminazione delle vecchie transizioni che partivano da quello stato. Per semplificare il funzionamento del sistema è possibile semplificare la procedura di aggiornamento evitando di dover eliminare le vecchie transizioni, a patto di considerare le update transition "prioritarie" rispetto agli altri tipi di transizione. Dai requisiti individuati nella sezione 3.2.2 si deduce inoltre che le controllable transition hanno priorità maggiore rispetto alle uncontrollable transition. Da questo insieme di informazioni si deduce che l'ordine con cui l'executor thread prende in considerazione le possibili transizioni da uno stato deve essere il seguente:

- Update transition
- Controllable transition
- Uncontrollable transition

Nella discussione della rappresentazione del controller è già stata evidenziata la scelta di lasciare che da ciascuno stato possa partire al massimo una controllable transition, in modo da semplificare il processo di esecuzione (che altrimenti dovrebbe scegliere non deterministicamente una delle possibilità) e la struttura del controller stesso.

Per poter iniziare l'esecuzione lo stato corrente dell'executor thread deve essere inizializzato con lo stato iniziale del controller, una informazione che deve fare parte della struttura dati che descrive il controller. A quel punto il thread può entrare nel loop principale di esecuzione.

Parallelamente al suo compito principale di esecuzione del controller, l'executor thread ha anche il compito di raccogliere le informazioni relative all'esecuzione che potrebbero esser significative per indagare su eventuali problemi sopravvenuti durante l'esecuzione. Tali informazioni dovrebbero indicare gli eventi ricevuti dal sistema, le transizioni effettuate, le operazioni eseguite, ed eventuali errori individuati avvenuti durante l'esecuzione. È ragionevole che questo compito sia demandato all'executor thread perché le informazioni necessarie sono ad esso facilmente accessibili.

4.4.1 Gestione della concorrenza

L'executor thread deve utilizzare appropriatamente gli elementi architetturali introdotti nella rappresentazione del controller per gestire correttamente l'accesso concorrente alla struttura dati che lo rappresenta.

La prima operazione eseguita consiste nell'“entrare” nello stato corrente: in questa fase avviene immediatamente l'acquisizione del monitor associato a questo stato, quindi qualora lo stato sia attualmente in fase di modifica da parte del deployer thread, il thread di esecuzione attenderà la conclusione dell'operazione di modifica. Vale la pena ricordare anche che, per costruzione, non vengono mai eseguite eliminazioni di stati raggiungibili dall'esecuzione, quindi non vi è alcun rischio che lo stato verso cui si sta entrando venga eliminato.

Appena acquisito il monitor, l'executor thread verifica se dallo stato attuale parte una epsilon transition, ed in tal caso la esegue. In caso contrario verifica se dallo stato attuale parte una controllable transition: se si esegue la sequenza di uncontrollable robot event ad essa associati ed esegue la transizione verso il nuovo stato, altrimenti l'esecuzione entra nel ciclo che attende l'arrivo di un uncontrollable event che dia luogo ad una transizione verso un altro stato. In tale ciclo l'executor thread si avvale dell'uncontrollable event manager precedentemente descritto per ricevere ogni even-

to e verificare sulla matrice di transizione se corrisponde ad una transizione.

All'uscita dallo stato attuale corrisponde il rilascio del monitor associato a tale stato, ed a questo punto il ciclo si ripete. Durante tutto questo processo, l'executor thread aggiunge al log di esecuzione delle informazioni che consentono di ricostruire ciò che è avvenuto durante l'esecuzione: le transizioni seguite, i controllabile event eseguiti, gli uncontrollable event ricevuti, ed eventuali errori.

Da questo insieme di considerazioni si determina lo pseudocodice presentato nell'algoritmo 1.

4.4.2 Eliminazione della vecchia versione del controllore

L'eliminazione del controllore può essere demandata all'executor thread. La vecchia versione del controllore diventa inutile nel momento in cui l'esecuzione segue una update transition che porta il controllore in uno stato del nuovo controllore. Dato che, per costruzione, non esistono transizioni che partono dagli stati della versione nuova del controllore ed arrivano a stati della versione vecchia, l'esecuzione non può più raggiungere il vecchio controller che quindi può essere eliminato in maniera sicura.

Se ad ogni stato del controllore è associato un numero di versione tale che:

- Tutti gli stati di una versione del controllore hanno lo stesso numero.
- Due stati appartenenti a due controllori di versioni diverse hanno numeri diversi.

allora l'executor thread, ad ogni esecuzione di una update transition (che è sempre tra due versioni diverse del controllore) conosce qual'è il numero di versione associato al vecchio controllore, e quindi può facilmente procedere all'eliminazione dei suoi stati.

Un aspetto di questa soluzione che dovrà essere valutato durante l'implementazione è il carico di lavoro necessario all'operazione di

Algorithm 1 Pseudocodice dell'executor thread

```
statoCorrente = statoIniziale;
while (true) do
    statoCorrente.monitor.acquisisci();
    if èAggiornabile(statoCorrente) then
        updateTransition =
            statoCorrente.getUpdateTransition();
        nuovoStato = updateTransition.getStatoFinale();
    else if haControllableTransition(statoCorrente) then
        transizione = statoCorrente.getContrTransition();
        eseguiControllableOperation(transizione.getOperation());
        nuovoStato = transizione.getStatoFinale();
    else
        while (true) do
            eventoRicev = attendiUncontrollableProblemEvent();
            uncontrTransitions =
                statoCorrente.getUncontrollableTransitions();
            if uncontrTransitions.contiene(eventoRicev) then
                nuovoStato = uncontrTransitions.
                    get(eventoRicev).getStatoFinale();
                break;
            end if
        end while
    end if
    statoCorrente.monitor.rilascia();
    statoCorrente = nuovoStato;
end while
```

eliminazione degli stati. Un controllore può infatti essere costituito da un numero notevole di stati, ed in tal caso anche il tempo necessario alla loro eliminazione potrebbe non essere più trascurabile, e quindi potrebbe portare ad un cambiamento del comportamento del controllore.

4.5 Deployer thread

Il compito primario svolto dal deployer thread consiste nella ricezione della nuova versione del controllore da un computer remoto, l'esecuzione dell'aggiornamento, e l'invio della risposta contenente l'esito dell'operazione. Si è deciso di porre tale thread come elemento centrale dell'architettura del sistema di comunicazione, integrando in esso anche tutte le altre funzionalità inerenti la comunicazione individuate nella sezione 3.3, quali: la lettura del log di esecuzione (vedi sezione 4.6), la comunicazione con altri sistemi per l'invio e la ricezione di eventi e le altre funzionalità di diagnostica remota.

Il protocollo studiato per gestire la comunicazione tra il robot ed il computer è un protocollo di tipo richiesta-risposta in cui è sempre il computer remoto ad inviare una richiesta utilizzando un set di comandi predefinito, ed il robot si limita a rispondere alla richiesta ricevuta per poi mettersi in attesa della ricezione di un nuovo comando. Ogni comando è identificato univocamente da una sequenza di caratteri, grazie alla quale il deployer thread è in grado di determinare la struttura dei dati che seguono il comando, struttura che è determinata a priori per ciascuno dei comandi possibili.

A causa dell'utilizzo di un protocollo di questo tipo non è possibile che il deployer thread invii immediatamente un messaggio in uscita prodotto dal controller e diretto verso un altro sistema attraverso il canale di comunicazione, perché il deployer non può inviare una comunicazione di sua iniziativa, ma deve attendere di essere interrogato dal computer remoto. Per questo motivo il deployer mantiene in memoria una coda in cui vengono memorizzati i messaggi in uscita, ed è compito del computer remoto interrogare

periodicamente il sistema per verificare se vi sono dei messaggi che devono essere inviati. La classe che implementa il deployer thread deve quindi esporre un metodo che consenta all'executor thread di aggiungere un messaggio alla coda dei messaggi in uscita.

Per quando riguarda la gestione della concorrenza nell'accesso alla struttura dati che rappresenta il controller, il deployer thread deve fare uso del sistema illustrato nella sezione 4.4.1.

Oltre al controller, il deployer thread si interfaccia con il sistema di logging, in modo da poter leggere il contenuto del log di esecuzione ed inviarlo al computer remoto quando questo lo richiede. Il sistema di logging costituisce una componente separata del sistema, e sarà descritto nella sezione 4.6; la gestione dell'accesso concorrente a tale sistema (dato che l'executor thread aggiunge informazioni al log, ed il deployer thread lo legge) sarà risolta all'interno del componente di logging stesso.

Il suo funzionamento è quindi definito nel seguente modo. Una volta avviato rimane in attesa di una comunicazione, e dopo averla stabilita resta in attesa di comandi da eseguire. Qualora il comando ricevuto richieda l'esecuzione di una operazione sul controllore, per garantire l'assenza di race condition tale operazione passerà attraverso il sistema di locking a livello di stato descritto nella sezione 4.3.1.

4.5.1 Comandi

In base all'analisi dei requisiti, sono stati individuati i seguenti comandi:

- Operazione di modifica del controllore. Essendo stato deciso che l'aggiornamento del controllore viene suddiviso in una serie di semplici comandi di modifica del controllore da parte del computer remoto che invia i comandi, questa operazione contiene un solo parametro che corrisponde ad una struttura dati che rappresenta l'operazione di modifica da eseguire. Dalla sezione 3.2.3 è emerso che tale comando può essere

Algorithm 2 Pseudocodice del ciclo principale del deployer thread

```
while (true) do
    connessione = attendiConnessione();
    while (connessione.aperta()) do
        idComando = connessione.leggiStringa();
        if idComando = "comando1" then
            eseguiComando1(connessione);
        else if idComando = "comando2" then
            eseguiComando2(connessione);
        else if idComando = "comando3" then
            eseguiComando3(connessione);
        end if
    end while
end while
```

l'aggiunta di un nuovo stato o l'aggiunta di una transizione di uno qualunque dei tre tipi possibili (update, controllable, uncontrollable).

- Ping del robot. Una semplice operazione utile a verificare il corretto funzionamento della comunicazione tra il computer ed il robot, che consiste nell'invio al robot di una stringa predefinita a cui il robot deve rispondere.
- Termine dell'esecuzione del programma sul robot. Questa operazione è necessaria in un contesto reale per poter terminare correttamente l'esecuzione del sistema.
- Invio di un messaggio al robot. Vista la decisione di utilizzare il computer come elemento centrale anche per la comunicazione con altri sistemi, l'architettura del sistema di comunicazione deve prevedere anche un comando che consenta l'invio di un messaggio al robot. Alla ricezione del messaggio il deployer thread lo inoltrerà all'UncontrollableEventManager.

- Ricezione dei messaggi dal robot. Per la stessa ragione è necessario un comando che consenta al computer di ricevere tutti i messaggi che il controller ha inviato dall'ultima operazione di lettura.
- Lettura del log di esecuzione. Questa operazione consente al computer remoto di leggere il contenuto del log di esecuzione, in modo da poterlo presentare all'utente, che quindi lo può utilizzare per verificare lo stato del sistema ed analizzare eventuali errori.
- Download del controller attuale. Questa operazione consente al computer remoto di farsi inviare dal robot l'intero controller attualmente in esecuzione. Questa operazione può essere utile nelle prime fasi di sviluppo del sistema per valutare situazioni in cui il comportamento del sistema non corrisponde a quello previsto.

4.6 Logging

L'idea alla base del sistema di logging consiste nel memorizzare in un file di log in un formato direttamente leggibile da un essere umano, sia una traccia dell'esecuzione del controller proveniente dall'executor thread (le transizioni, le operazioni eseguite, gli eventi ricevuti e gli eventuali errori incontrati), sia eventuali informazioni significative provenienti da altre componenti del sistema (ad esempio eccezioni nel polling dei sensori).

L'architettura del sistema è dunque la seguente. È stato sviluppato un componente separato a cui l'executor thread accede per aggiungere informazioni al log ogni qual volta sia necessario: ad ogni ingresso in uno stato, ad ogni esecuzione di un controllabile event, ad ogni uncontrollable event ricevuto, l'executor thread utilizzerà l'interfaccia fornita dal sistema di logging per aggiungere al log le informazioni significative.

Il sistema di logging infatti è stato sviluppato per essere utilizzato indifferentemente da più thread contemporaneamente. La classe che implementa il logger deve fornire dei metodi per leggere e scrivere il log che possono essere chiamati da qualunque thread, gestendo internamente eventuali problemi di concorrenza.

In questo modo qualunque componente del sistema può aggiungere componenti al log, e questa caratteristica è particolarmente utile se si considera la molteplicità dei componenti coinvolti nel sistema.

Tramite l'utilizzo del design pattern singleton si garantisce l'esistenza di un'unica istanza della classe Logger, e si fornisce a tutti i componenti del sistema un modo per accedervi.

4.7 Software di gestione remota del robot

Il compito principale del software di gestione remota consiste nell'aggiornamento a runtime del controllore; per eseguire tale operazione il software ha a disposizione la sequenza di comandi in grado di trasformare il controller attualmente in esecuzione nel dynamically updating controller e, dopo essersi connesso al robot, la invia, e mostra il risultato dell'operazione. L'architettura di questo software è complementare a quella del deployer thread: l'interfaccia deve consentire di eseguire ciascuno dei comandi individuati in precedenza, e presentare all'utente l'esito dell'invio.

Capitolo 5

Implementazione

In questo capitolo verrà descritta l'implementazione, sul sistema Lego Mindstorm NXT, dell'infrastruttura di esecuzione ed aggiornamento dinamico sicuro di controllori basati su automi a stati finiti.

5.1 Software utilizzato

5.1.1 LeJOS

Il software eseguito sul robot, che ha il compito di eseguire ed aggiornare il controllore a stati finiti e comunicare con il sistema di gestione remoto è basato su LeJOS, un firmware sviluppato per il sistema Lego Mindstorm NXT che include una macchina virtuale Java grazie alla quale è stato possibile utilizzare tale linguaggio per programmare il robot. Il fatto che Java sia un linguaggio attualmente ampiamente utilizzato e funzionante su una grande varietà di architetture garantisce che l'infrastruttura sviluppata nell'ambito di questo lavoro possa essere facilmente adattata a diverse piattaforme. Inoltre LeJOS ha il pregio di includere una serie di API che facilitano notevolmente l'utilizzo dell'hardware del sistema Lego NXT.

5.1.2 Piattaforma Java SE

Per sviluppare il software di gestione remota che viene utilizzato per collegarsi al robot, verificarne il funzionamento ed aggiornare il controllore, la piattaforma Java SE è stata una scelta favorita dal fatto che LeJOS include anche delle librerie Java che facilitano la comunicazione con il robot. Grazie al fatto che sia il robot e sia il computer remoto eseguono programmi scritti in Java, inoltre, è stato possibile sviluppare delle librerie in comune ad entrambe le parti, che hanno facilitato ulteriormente lo sviluppo del sistema di comunicazione.

5.2 Gestione degli eventi

5.2.1 Gestione degli uncontrollable event

Come stabilito nella definizione dell'architettura, la gestione degli uncontrollable event è demandata al componente UncontrollableEventManager. Nella figura 5.1 è possibile osservare alcuni dettagli della sua implementazione.

La relazione tra la classe UncontrollableEventManager e la classe UncontrollableEvent definisce l'insieme dei possibili uncontrollable robot event che tale componente deve gestire. In accordo con il progetto architetturale del sistema, questo insieme è definito in fase di inizializzazione del sistema, ed infatti si può osservare che insieme è presente nel costruttore, assieme alle istanze della classe EventTranslationMachine che definiscono gli automi a stati finiti che effettuano la traduzione tra uncontrollable robot event ed uncontrollable problem event.

L'implementazione del sistema ha anche richiesto lo sviluppo di alcuni thread necessari a gestire certe sorgenti di eventi, ad esempio il SensorHandlerThread effettua l'interrogazione periodica dei sensori che richiedono il polling periodico allo scopo di individuare il verificarsi degli eventi.

Come specificato nel progetto dell'architettura, tutta la fase di preparazione alla ricezione degli eventi tramite sensori è svolta dall'`UncontrollableEventManager`, che quindi inizializza e gestisce tutti gli eventuali thread di supporto.

Le funzionalità messe a disposizione dalle librerie di LeJOS sono una porzione estremamente ridotta rispetto a quelle messe a disposizione dal normale Java Runtime Environment, e non includono alcuna struttura che possa essere utilizzata efficientemente come coda FIFO bloccante per la gestione degli eventi. È stato quindi necessario sviluppare una coda adatta all'utilizzo che ne viene fatto nell'`UncontrollableEventManager`.

5.2.2 Gestione dei controllable event

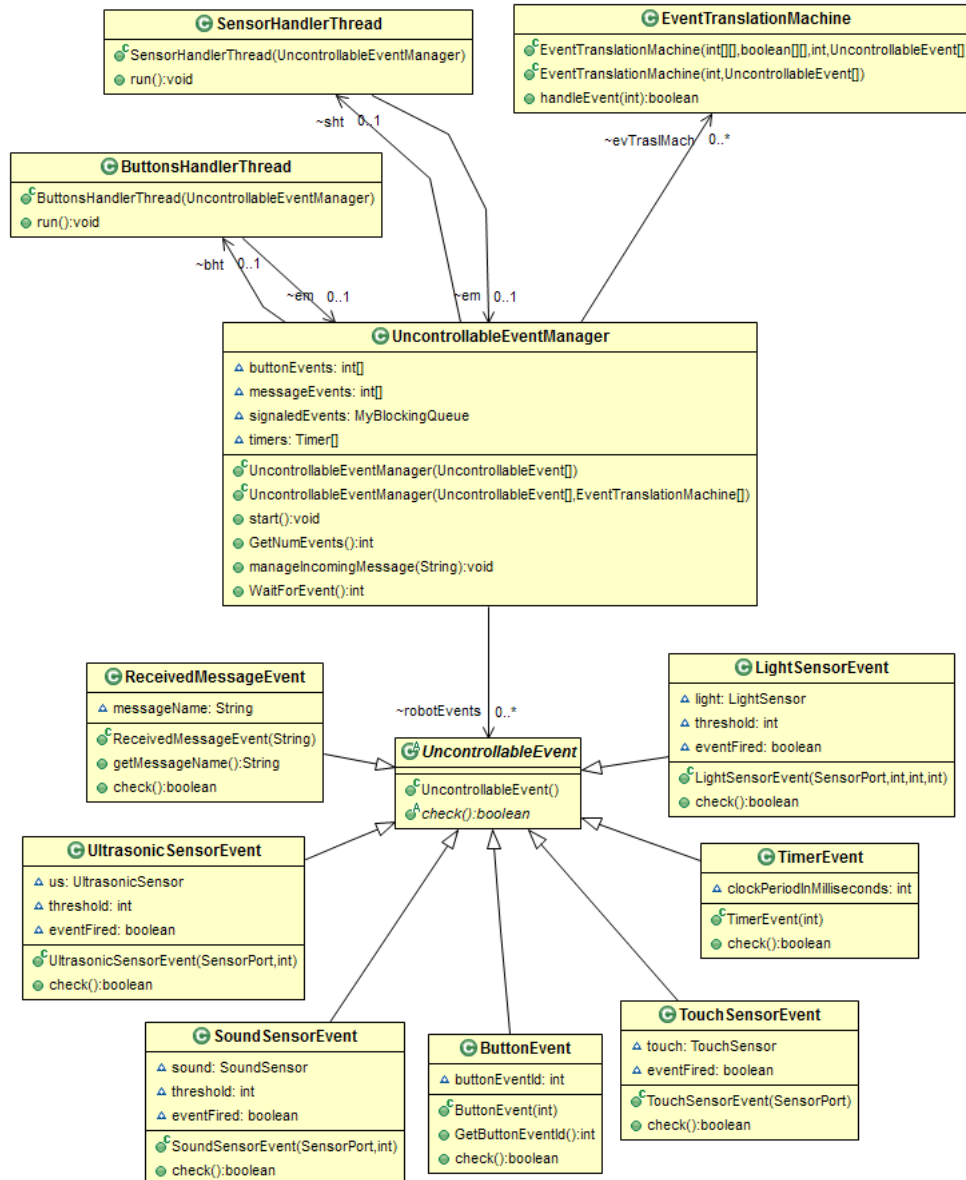
Anche per rappresentare i possibili controllable robot event è stata implementata una gerarchia di classi analoga a quella sviluppata per gli uncontrollable event. Tale gerarchia è illustrata in figura 5.2.

Si può anche osservare che le classi implementano dei metodi che consentono la serializzazione e deserializzazione attraverso uno stream. Questa caratteristica consente di facilitare la trasmissione dei controllable event attraverso la connessione utilizzata con il computer remoto che, dovendo inviare la nuova versione del controllore, deve inviare i controllable event associati alle controllable transition.

5.3 Rappresentazione del controller

La rappresentazione del controller rispecchia quanto è stato stabilito nella fase di progettazione architetturale, e di conseguenza è stata creata una classe `State` che contiene:

- Un array di controllable robot event, che rappresenta la sequenza di operazioni che deve essere eseguita quando viene seguita tale transizione.

Figura 5.1: Implementazione dell'`UncontrollableEventManager`

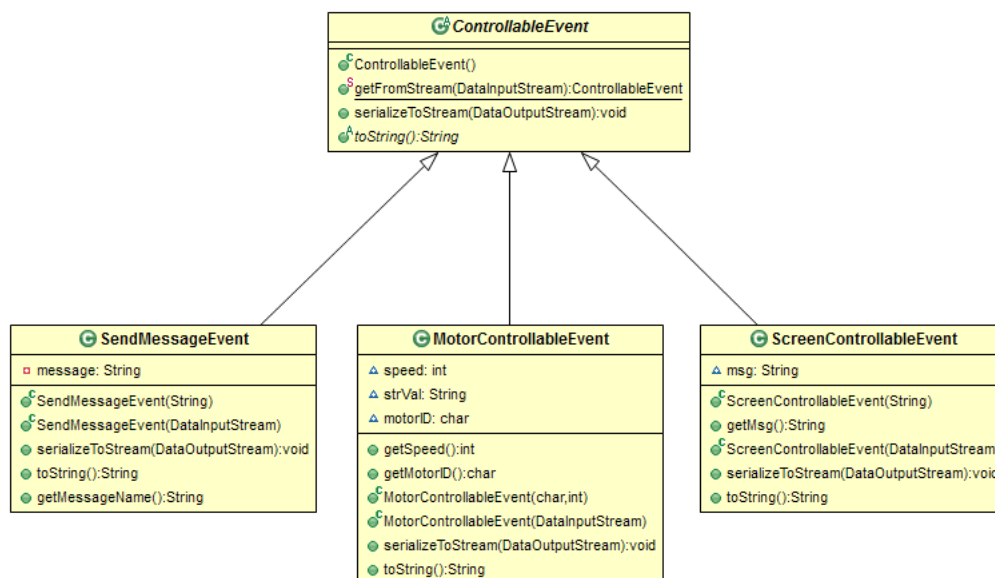


Figura 5.2: Classi che rappresentano i controllable robot event

- Un array di dimensione pari al numero di uncontrollable problem event definiti nell'UncontrollableEventManager, che indica verso quale stato ciascuno degli eventi scatena una transizione.
- L'identificatore dello stato di destinazione dell'update transition, se tale transizione esiste.
- Un oggetto che viene utilizzato per la gestione dell'accesso concorrente, tramite gli strumenti per la sincronizzazione offerti da Java.

5.3.1 La struttura dati dinamica

Ogni stato del controller è identificato univocamente da un numero intero, e tutti gli stati sono memorizzati in una struttura creata appositamente: un array sparso basato su una hash table che utilizza come chiave un numero intero. Questa soluzione è stata ritenuta valida sia perché, come richiesto dai requisiti, la chiave che identifica gli stati è un numero intero (la posizione nell'array sparso), ma

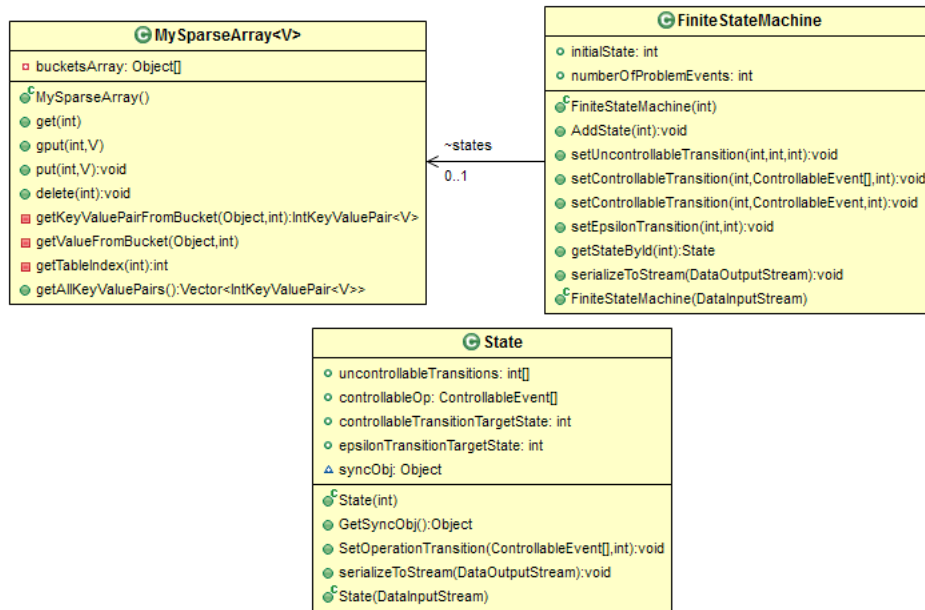


Figura 5.3: Classi utilizzate nella rappresentazione del controller

soprattutto perché se il numero di collisioni di hash è trascurabile, la complessità computazionale delle operazioni di lettura, inserimento, ed eliminazione è data solamente dal calcolo dell'hash e da un accesso ad un array.

Purtroppo LeJOS non offre alcuna struttura dati di questo tipo, pertanto è stato necessario svilupparne una da zero. Nonostante l'impegno richiesto è stata un'occasione per sviluppare una struttura che si adatti al meglio alle caratteristiche del problema.

La funzione di hash utilizzata sulla chiave è il resto della divisione per il numero di bucket. Questa funzione pur essendo estremamente semplice si adatta molto bene al modo in cui la struttura dati è utilizzata: man mano che i nuovi stati vengono creati, infatti, le chiavi utilizzate per identificarli sono contigue, essendo derivate da un contatore crescente, quindi fin tanto che il numero di stati è inferiore al numero di bucket non si verificano collisioni di hash.

5.4 Gestione dell'accesso concorrente

La gestione della concorrenza è stata sviluppata facendo uso del costrutto Java *synchronized*. Ad ogni stato del controller è associato un oggetto che tramite tale costrutto viene utilizzato per garantire che il deployer thread non possa modificare lo stato in cui l'executor thread si trova attualmente. Anche l'array sparso che contiene l'insieme degli stati è stato sviluppato in modo tale da garantire che non si verifichino race condition quando il deployer thread tenta di aggiungere uno stato mentre l'executor thread tenta di eseguire una transizione.

5.5 Librerie in comune tra computer e robot

Molti delle classi citate fin'ora in questa descrizione dell'implementazione sono state racchiuse in un package che viene utilizzato sia nell'ambiente di esecuzione del robot, sia nel sistema di controllo remoto, in questo modo entrambe le parti possono condividere le stesse strutture dati. A queste classi sono state aggiunte delle funzionalità che consentono alle istanze di essere serializzate e deserializzate attraverso uno stream, che nel caso di questa implementazione è fornito dalla connessione Bluetooth. Questa caratteristica ad esempio consente al computer remoto di ottenere una copia esatta del controller attualmente in esecuzione, dato che questo può essere serializzato, inviato attraverso la connessione, e deserializzato. A questo punto il computer possiede una copia esatta del controller in esecuzione.

La figura 5.4 mostra alcune classi che sono state incluse nelle librerie comuni allo scopo di facilitare l'implementazione del sistema di comunicazione. Le classi mostrare in particolare rappresentano le operazioni di aggiornamento che possono essere eseguite sul controller.

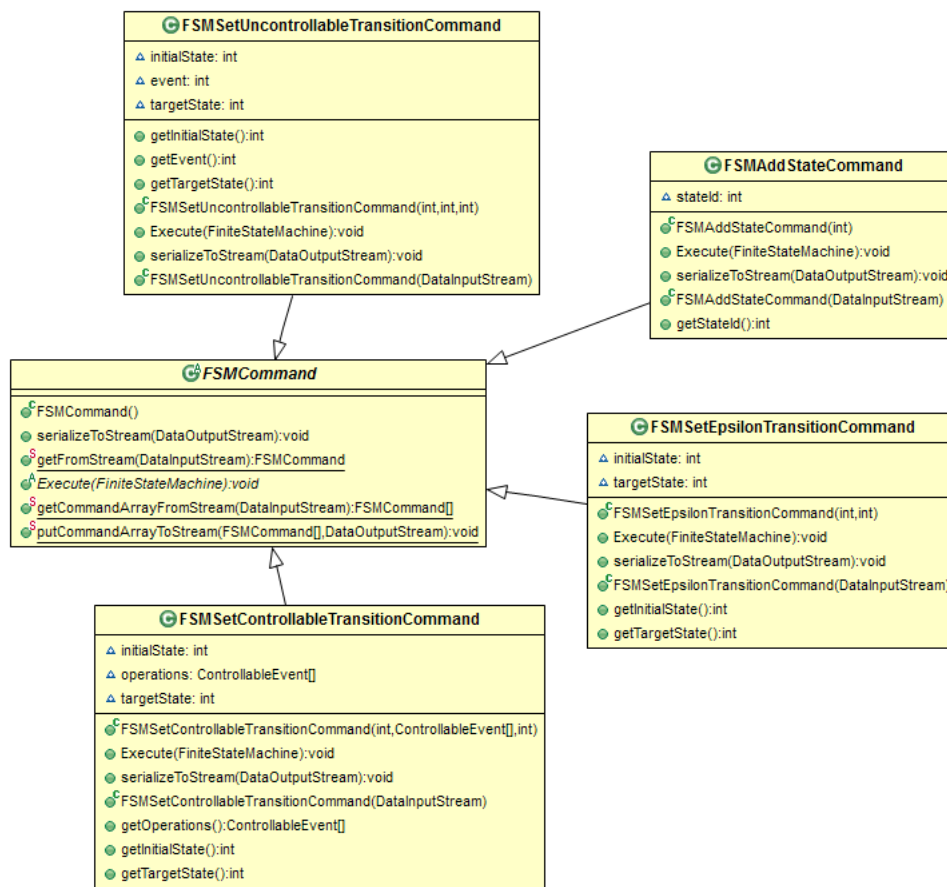


Figura 5.4: Alcune classi presenti nelle librerie comuni

5.6 Executor thread

Dall'analisi dei requisiti e dalla progettazione dell'architettura è necessario che tale thread sia sviluppato in modo da rispettare i requisiti precedentemente individuati riguardo l'esecuzione del controller: deve rispettare le specifiche relative necessarie a gestire correttamente la concorrenza, deve cercare i diversi tipi di transizione nell'ordine corretto, ed infine deve raccogliere nel log di sistema le informazioni che possano essere significative per indagare su eventuali problemi sopravvenuti durante l'esecuzione.

5.6.1 Dettagli dell'algoritmo di esecuzione

Lo stato corrente del controller deve essere inizializzato con lo stato iniziale del controller, una informazione che, come visto in precedenza, fa parte della struttura dati che descrive il controller. A quel punto il thread può entrare nel loop principale di esecuzione del controller.

La prima operazione eseguita consiste nell'entrare nello stato corrente: in questa fase avviene immediatamente l'acquisizione del monitor associato a questo stato, quindi qualora lo stato sia attualmente in fase di modifica da parte del deployer thread, il thread di esecuzione attenderà la conclusione dell'operazione di modifica. Vale la pena ricordare anche che, per costruzione, non vengono mai eseguite eliminazioni di stati raggiungibili dall'esecuzione, quindi non vi è alcun rischio che lo stato verso cui si sta entrando venga eliminato.

Appena acquisito il monitor, l'executor thread verifica se dallo stato attuale parte una epsilon transition, ed in tal caso la esegue. In caso contrario verifica se dallo stato attuale parte una controllable transition: se si esegue la sequenza di uncontrollable robot event ad essa associati ed esegue la transizione verso il nuovo stato, altrimenti l'esecuzione entra nel ciclo che attende l'arrivo di un uncontrollable event che dia luogo ad una transizione verso un al-

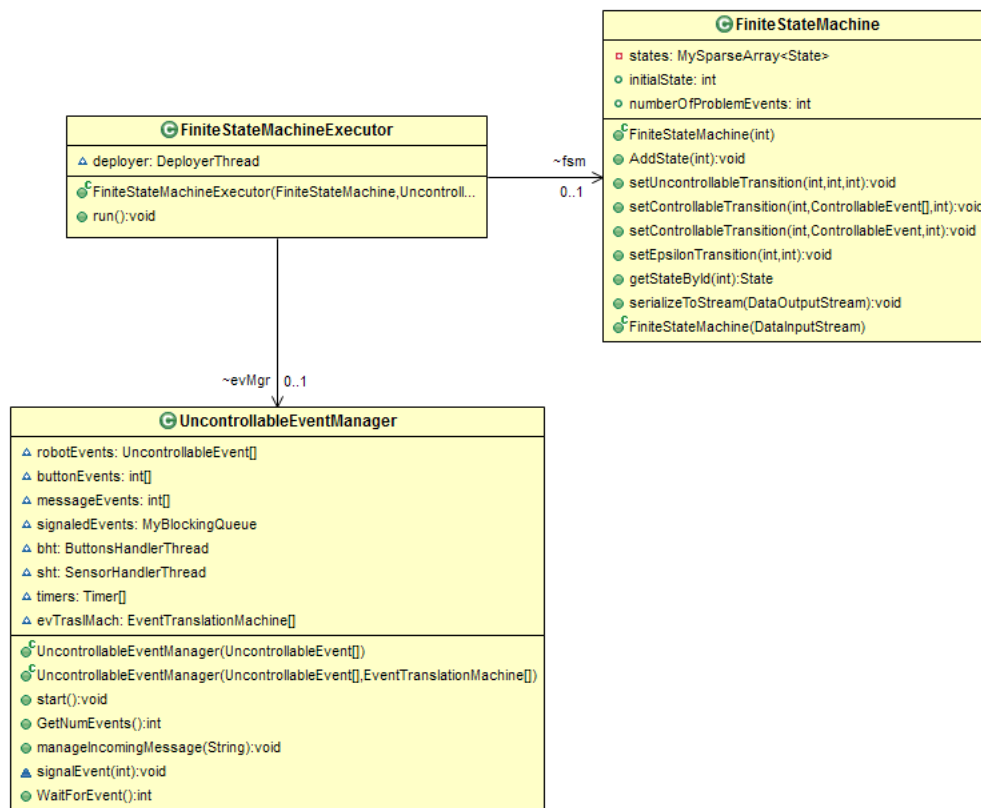
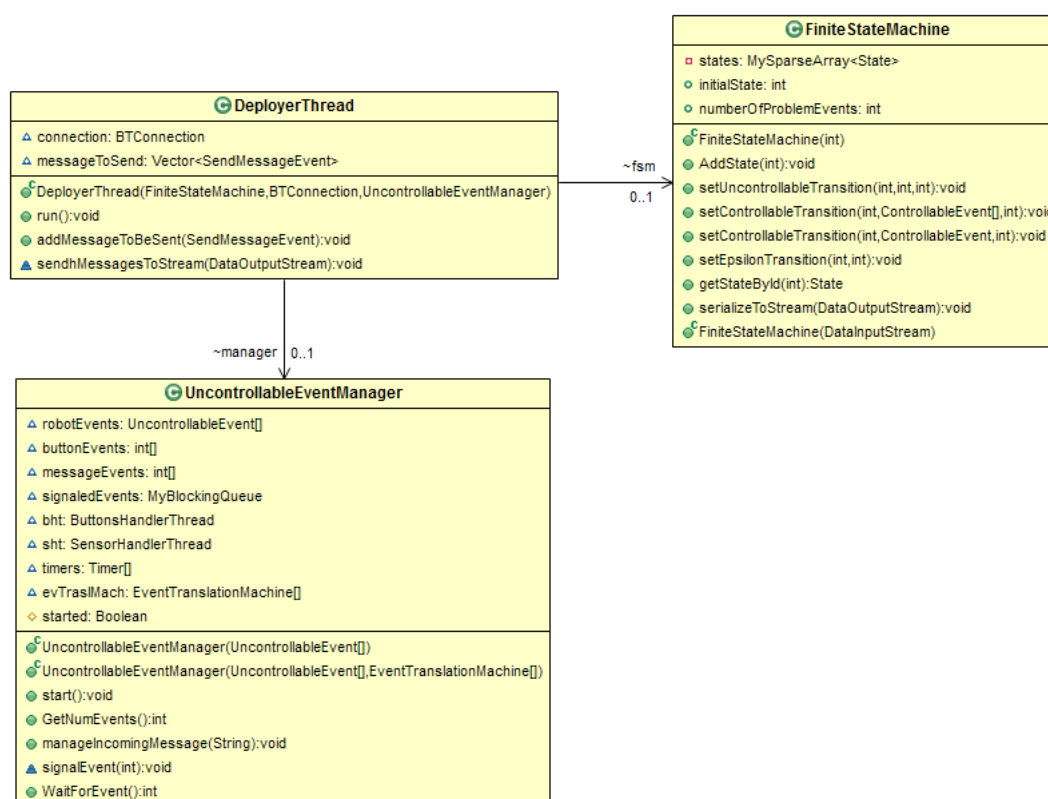


Figura 5.5: La classe *FiniteStateMachineExecutor* che implementa l'executor thread

tro stato. In tale ciclo l'executor thread si avvale dell'uncontrollable event manager precedentemente descritto per ricevere ogni evento e verificare sulla matrice di transizione se corrisponde ad una transizione.

All'uscita dallo stato attuale corrisponde il rilascio del monitor associato a tale stato, ed a questo punto il ciclo si ripete.

Durante tutto questo processo, l'executor thread aggiunge al log di esecuzione delle informazioni che consentono di ricostruire ciò che è avvenuto durante l'esecuzione: le transizioni seguite, i controllabile event eseguiti, gli uncontrollable event ricevuti, ed eventuali errori.

Figura 5.6: La classe *DeployerThread*

5.6.2 Implementazione

La figura 5.5 mostra la classe `FiniteStateMachineExecutor` che implementa l'executor thread. Tale classe contiene il riferimento all'istanza della classe `FiniteStateMachine` contenente il controller da eseguire, e contiene il riferimento all'istanza dell'`UncontrollableEventManager` che viene utilizzato per gestire gli uncontrollable event in arrivo.

5.7 Deployer thread

Come specificato nella progettazione architetturale, il deployer thread ha due compiti fondamentali.

- Gestire la comunicazione con il computer remoto per:

- Inviare gli aggiornamenti del controller.
 - Verificare lo stato del sistema tramite le funzionalità di diagnostica remota (lettura del log, download del controller, verifica della memoria disponibile, ecc...).
 - Ricevere i controllabile robot event che il robot invia verso altri sistemi (ad esempio il messaggio che il robot dell'esempio del capitolo 2 manda alla stazione di ricarica per chiedere l'autorizzazione all'avvicinamento), ed inviare al robot eventuali uncontrollable robot event provenienti da altri sistemi.
- Eseguire le operazioni di aggiornamento del controllore.

Il deployer thread una volta avviato rimane in attesa di una connessione via Bluetooth, ed una volta stabilita la connessione resta in attesa dei comandi da eseguire.

5.7.1 Protocollo di comunicazione

In accordo con quanto stabilito nella sezione 4.5 il protocollo sviluppato per fornire queste funzionalità è un protocollo di tipo request-response: è sempre il computer remoto ad inviare per primo uno dei comandi definiti nel protocollo, ed il robot risponde con le informazioni richieste o con l'esito dell'esecuzione dell'operazione richiesta.

Il computer remoto invia una sequenza di 4 caratteri che identifica il comando richiesto, seguita dagli eventuali parametri, ed ogni comando prevede che la risposta inviata dal robot segua una struttura predefinita, nota al computer remoto che quindi può interpretarla. Per facilitare la programmazione, rendere più semplice la manutenzione ed il debug, e migliorare l'estendibilità futura, gran parte delle informazioni che devono transitare attraverso il canale di comunicazione sono strutturate come istanze di classi che supportano la serializzazione e la deserializzazione attraverso uno stream. Ogni comando di aggiornamento di uno stato, ad esempio, corrisponde ad un'istanza di una delle possibili sottoclassi della classe

Operazione di modifica del controller	
Questo comando consente di inviare al robot un'operazione di modifica del controllore attualmente in esecuzione, ed ottenere una risposta sull'esito della sua esecuzione.	
Identificatore	FSMC
Richiesta	• Istanza della classe FSMCommand serializzata
Risposta	Se il comando è stato eseguito senza problemi: • "C-OK" Se il comando ha dato luogo ad un errore: • "C-KO" • Lunghezza del messaggio di errore • Messaggio di errore

Tabella 5.1: Struttura del comando FSMC

FSMCommand (visibili nella figura 5.4) che contiene l'identificatore dello stato interessato dalla modifica, e le altre informazioni eventualmente necessarie, quali le controllabile transition (quindi anch'esse serializzabili). Per inviare al robot un comando di aggiornamento, quindi, il computer invia l'identificatore del comando di aggiornamento, seguito dalla serializzazione dell'istanza della classe FSMCommand.

Questo approccio implica che tutte le classi utilizzate da entrambi i lati della comunicazione (robot e computer) siano contenute in un package apposito, condiviso da entrambi i progetti (l'ambiente di esecuzione sul robot, ed il programma di gestione sul pc).

Qualora il comando ricevuto richieda l'esecuzione di una operazione sul controllore, l'esecuzione di tale comando passerà attraverso il sistema di locking a livello di stato, per evitare race conditions.

Le tabelle dalla 5.1 alla 5.9 mostrano i comandi che sono previsti dal protocollo di comunicazione, ed i parametri necessari a ciascuna di esse.

Ping del robot	
Invia al robot un semplice messaggio a cui il robot risponde con una risposta, per testare la connessione	
Identificatore	PING
Richiesta	Nessun parametro
Risposta	"PONG"

Tabella 5.2: Struttura del comando PING

Termina esecuzione	
Provoca il termine dell'esecuzione del programma sul robot	
Identificatore	EXIT
Richiesta	Nessun parametro
Risposta	Nessuna risposta

Tabella 5.3: Struttura del comando EXIT

Invio di un messaggio	
Invia al robot un evento corrispondente al messaggio indicato	
Identificatore	SMSG
Richiesta	• Lunghezza del messaggio • Contenuto del messaggio
Risposta	Nessuna risposta

Tabella 5.4: Struttura del comando SMSG

Leggi i messaggi in arrivo dal robot	
Legge tutti i messaggi in uscita destinati a sistemi esterni al robot.	
Identificatore	RMSG
Richiesta	Nessun parametro
Risposta	• Numero di messaggi in uscita per ciascuno di essi: • Lunghezza del messaggio • Contenuto del messaggio

Tabella 5.5: Struttura del comando RMSG

Lettura del log di esecuzione	
Legge la porzione del log di sistema generata dopo l'ultima lettura.	
Identificatore	GLOG
Richiesta	Nessun parametro
Risposta	• Flag che indica l'overflow • Lunghezza della porzione del log • Contenuto della porzione del log

Tabella 5.6: Struttura del comando GLOG

Stato della memoria	
Verifica la quantità di memoria totale e disponibile nel robot	
Identificatore	MEMO
Richiesta	Nessun parametro
Risposta	• Memoria totale in byte • Memoria disponibile in byte

Tabella 5.7: Struttura del comando MEMO

Download del controller attuale	
Richiede al robot di inviare l'intero controller attualmente in esecuzione	
Identificatore	GETC
Richiesta	Nessun parametro
Risposta	• Istanza della classe <code>FiniteStateMachine</code> serializzata

Tabella 5.8: Struttura del comando GETC

Stato generale del sistema	
Interroga il robot sullo stato generale del sistema, accorpando più interrogazioni in una sola.	
Identificatore	STAT
Richiesta	Nessun parametro
Risposta	• Memoria totale in byte • Memoria disponibile in byte • Porzione del log di esecuzione (vedi comando GLOG) • Messaggi in uscita dal robot (vedi comando RMSG)

Tabella 5.9: Struttura del comando STAT

5.8 Software di gestione remota

In figura 5.7 è possibile vedere la schermata principale del software di gestione remota sviluppato. Tale software consente di:

- Caricare da file la sequenza di comandi che aggiorna il controllore attualmente in esecuzione sul robot.
- Inviare al robot la sequenza di comandi caricata.
- Leggere il log di esecuzione del robot.
- Inviare al robot uncontrollable robot event di tipo *messaggio*.
- Ricevere gli eventi di tipo controllable robot event di tipo *messaggio*.
- Visualizzare la quantità di memoria disponibile.

5.9 Sistema di logging

La realizzazione del sistema di logging definito nel disegno architetturale del sistema ha presentato alcune sfide. Gli stretti vincoli

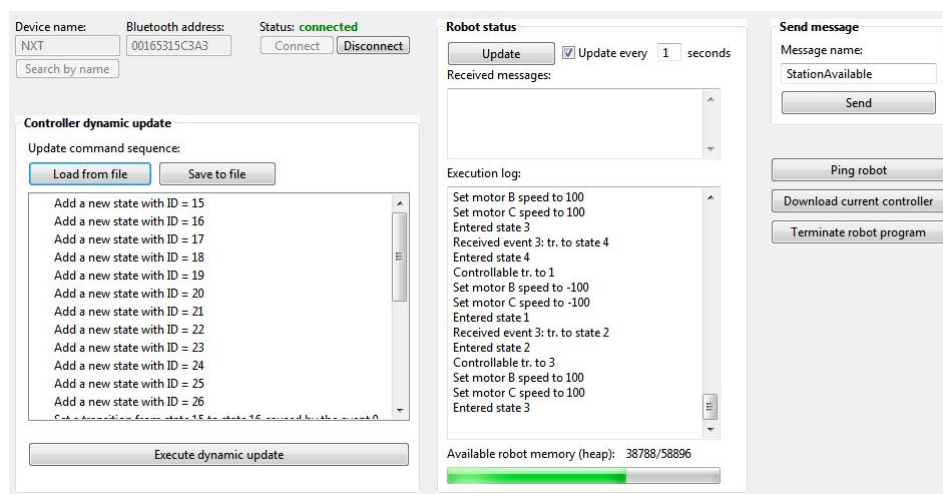


Figura 5.7: Il software di gestione remota

dovuti all'architettura hardware, ed in particolare la poca memoria disponibile, hanno reso necessario l'utilizzo di accorgimenti particolari; tale log infatti tende a crescere rapidamente in dimensione, rischiando di utilizzare tutta la poca memoria disponibile. Il fatto che diversi thread possano aggiungere informazioni al log, ed un thread possa leggerlo, ha reso necessario un sistema per gestire l'accesso concorrente.

Il sistema di logging sviluppato consiste quindi in un buffer circolare di dimensione fissa, con un sistema che consente una sola operazione di scrittura o lettura alla volta. Il fatto che il buffer sia circolare consente di assegnargli una certa quantità di memoria, che deve essere periodicamente vuotata dal computer remoto che legge il log. Esiste quindi la possibilità che, se il computer remoto non vuota sufficientemente spesso il contenuto del log, una parte del suo contenuto venga perso.

Tale sistema è contenuto nella classe `Logger`, presentata in figura 5.8.

La quantità di memoria che è opportuno assegnare al log deve essere valutata situazione per situazione: i problemi che prevedono un gran numero di eventi e transizioni in un breve periodo di tempo tenderanno a riempire rapidamente il buffer, ed è quindi ne-

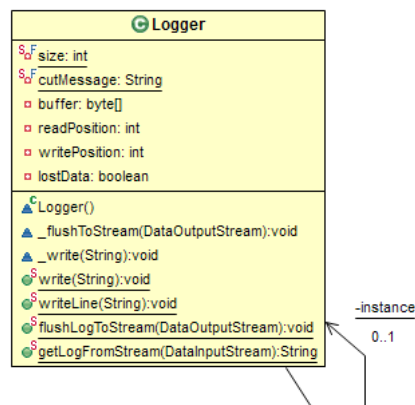


Figura 5.8: La classe *Logger* che implementa il sistema di logging

cessario dedicare ad esso più memoria, oppure scaricarne spesso il contenuto. A titolo di esempio, sono stati eseguiti dei test basati sul problema presentato nel capitolo 2, in cui il robot deve avvicinarsi alla base di ricarica rispettando un certo comportamento, un problema che genera un numero modesto di eventi: in tale situazione un buffer di solo 1kByte letto una volta al secondo si è rivelato sufficiente a coprire le necessità del sistema.

5.9.1 Esempio di log

Di seguito viene proposta una porzione del log di esecuzione dell'esempio presentato nel capitolo 2.

```

1 Received event 0: tr. to state 6
2 Entered state 6
3 Controllable tr. to 7
4 Set motor B speed to 100
5 Set motor C speed to 100
6 Entered state 7
7 Received event 3: no transition
8 Received event 1: tr. to state 8
9 Entered state 8
10 Controllable tr. to 9
11 Set motor B speed to 0
12 Set motor C speed to 0
13 Entered state 9
14 Received event 3: tr. to state 10
15 Entered state 10
16 Received event 3: tr. to state 11
  
```

```
17      Entered state 11
18      Epsilon tr. to 23
19      Entered state 23
20      Controllable tr. to 24
21      Set motor B speed to -200
22      Set motor C speed to -200
23      Entered state 24
24      Received event 3: tr. to state 25
25      Entered state 25
26      Received event 3: tr. to state 26
27      Entered state 26
28      Controllable tr. to 15
29      Set motor B speed to 200
30      Set motor C speed to 200
31      Entered state 15
32      Received event 3: no transition
33      Received event 0: tr. to state 16
34      Entered state 16
35      Controllable tr. to 17
36      Set motor B speed to 0
37      Set motor C speed to 0
38      Send message: AskingAvailability
39      Entered state 17
40      Received event 3: no transition
41      Received message: StationAvailable
42      Received event 2: tr. to state 18
43      Entered state 18
44      Controllable tr. to 19
45      Set motor B speed to 100
46      Set motor C speed to 100
47      Entered state 19
48      Received event 3: no transition
49      Received event 3: no transition
50      Received event 1: tr. to state 20
51      Entered state 20
52      Controllable tr. to 21
53      Set motor B speed to 0
54      Set motor C speed to 0
```

Il comportamento iniziale del robot prevede che appena viene raggiunta una certa distanza dalla base di ricarica il robot rallenti, e le prime sei linee del log mostrano questo comportamento: l'arrivo dell'evento 0 (generato dal sensore di distanza) produce una transizione del sistema verso lo stato 6, dal quale parte una controllable transition verso lo stato 7 che rallenta i motori collegati alle ruote.

Il robot quindi prosegue lentamente verso la base, fin quando, alla linea 8 del log, viene ricevuto l'evento 1 (generato dal sensore di contatto) che indica che il robot è arrivato alla stazione, e provoca

una transizione verso lo stato 8.

Appena il sistema entra nello stato 8 esegue immediatamente una controllable transition verso lo stato 9 che ferma entrambi i motori (linee 10, 11 e 12 del log).

Il robot si trova quindi ad essere fermo alla base di ricarica, mentre sta caricando le batterie. Il progresso del processo di ricarica è simulato dall'arrivo dell'evento 3 alle righe 14 e 16 del log, evento che in realtà proviene da un timer.

A questo punto è possibile osservare il processo di aggiornamento dinamico in azione: mentre il sistema era in esecuzione gli è stato inviato tramite Bluetooth un nuovo controller, e sono state aggiunte al controller attuale le update transition dagli stati aggiornabili del vecchio controller ai corrispondenti stati della nuova versione. Entrando nello stato 11 (linea 17 del log) l'executor thread osserva che da tale stato parte una ϵ -transition verso lo stato 23 (linea 18) e la esegue immediatamente (linea 19).

Da questo punto in poi l'esecuzione procede nel nuovo controller, il cui comportamento è quello che consente la condivisione della base di ricarica tra più robot: quando il sensore di distanza rileva l'avvicinamento alla base a causa dell'evento 0 (linea 33) viene eseguita una transizione che ferma i motori e invia alla stazione di ricarica l'autorizzazione ad avvicinarsi (linee 36, 37 e 38).

Il robot rimane fermo in tale stato fino a quando riceve dalla stazione l'autorizzazione ad avvicinarsi (linee 41 e 42), e quindi il comportamento prosegue in maniera analoga a quanto accadeva con la vecchia versione: il robot procede lentamente (linee 45 e 46), arriva alla stazione (linea 50) e si ferma (linee 53 e 54).

Capitolo 6

Conclusioni

In questa tesi è stato sviluppato un approccio sistematico alla realizzazione di un'architettura che consente l'esecuzione e l'aggiornamento dinamico sicuro degli automi a stati finiti utilizzati come controllori di reactive system. L'architettura che risulta dall'applicazione di tale approccio garantisce che l'aggiornamento avvenga dinamicamente non appena il controllore è in uno stato che lo consente, senza che il sistema mostri un cambiamento osservabile del comportamento.

È stata anche presentata un'implementazione di tale infrastruttura realizzata utilizzando Java sul sistema Lego Mindstorm NXT, che consente di eseguire ed aggiornare dinamicamente i controllori generati con ScenarioTools, uno strumento model-driven che consente di sintetizzare automaticamente degli automi a stati finiti che garantiscono il rispetto delle specifiche espresse tramite diagrammi MSD. Tale implementazione è facilmente riutilizzabile su qualsiasi sistema che utilizzi Java.

L'implementazione realizzata ha dimostrato le potenzialità che risultano dall'utilizzo dell'approccio teorico descritto in [8] unito all'infrastruttura di esecuzione descritta in questo lavoro: dopo la definizione o la modifica delle specifiche di funzionamento del sistema l'intero processo (sintesi del controllore, sintesi del dynamically updating controller e aggiornamento del controller in esecu-

zione) è completamente automatizzato, e questo consente di reagire rapidamente ai cambiamenti delle specifiche; come evidenziato già nell'introduzione questo fatto è particolarmente significativo.

Capitolo 7

Sviluppi futuri

Gli sviluppi futuri di questo lavoro possono essere i seguenti:

- Valutazione dell'applicabilità dell'approccio su sistemi più complessi. La complessità del sistema pone infatti delle sfide riguardanti la complessità computazionale del processo di sintesi del controllore e l'effettiva applicabilità dell'approccio proposto al crescere della dimensione del controllore.
- Estensione dell'approccio ai sistemi distribuiti. Questo possibile sviluppo presenta infatti delle sfide notevoli: l'aggiornamento dinamico di un insieme di componenti controllati da dei controllori separati è un problema estremamente complesso sia dal punto di vista della sintesi del controllore, sia dal punto di vista della realizzazione pratica del processo di aggiornamento.
- Valutazione dell'utilizzo dell'approccio descritto in questa tesi allo sviluppo di sistemi auto adattativi.

Bibliografia

- [1]
- [2] R. Adler, I. Schaefer, T. Schuele, and E. Vecchié. From model-based design to formal verification of adaptive embedded systems. In *Proc. 9th Intl. Conf. on Formal methods and software engineering*, ICFEM'07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.
- [3] S. Ajmani. *Automatic software upgrades for distributed systems*. PhD thesis, MIT, Cambridge, MA, USA, 2004.
- [4] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Timed games for everyone. In *Nordic Workshop on Programming Theory (NWPT'06)*, 2006.
- [5] S. Bouveret, J. Brunel, D. Chemouil, and F. Dagnaty. Towards a categorical framework to ensure correct software evolutions. In *Proc. 27th International Conference on Data Engineering Workshops*, ICDEW '11, pages 139–144, Washington, DC, USA, 2011. IEEE.
- [6] S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *Proc. 3rd Workshop on specification and verification of component based systems (SAVCBS)*, pages 55–61, Oct. 2004.
- [7] R. P. Cook and I. Lee. Dymos: A dynamic modification system. In *Proc. Software engineering symposium on High-level debug-*

- ging, SIGSOFT '83, pages 201–202, New York, NY, USA, 1983. ACM.
- [8] C. Ghezzi, J. Greenyer, and V. P. La Manna. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 145–154. IEEE Press, 2012.
- [9] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proc. 12th Intl. Symp. on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 179–188, New York, NY, USA, 2004. ACM.
- [10] J. Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, 2011.
- [11] A. R. Gregersen and B. N. Jørgensen. Dynamic update of java applications—balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21(2):81–112, Mar. 2009.
- [12] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu. Javelus: A low disruptive approach to dynamic software update. In *19th Asia-Pacific Software Engineering Conf. (APSEC 2012)*, pages 527–537, Hong Kong, China, dec. 2012. IEEE Computer Society.
- [13] D. Gupta and P. Jalote. On line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23:949–964, Sept. 1993.
- [14] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Software Engineering*, 22(2):120–131, Feb. 1996.
- [15] D. Hare and A. Pnueli. On the development of reactive systems. 1985.

- [16] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.
- [17] D. Harel and R. Marelly. *Come, let's play: scenario-based programming using LSCs and the play-engine*, volume 1. Springer Science & Business Media, 2003.
- [18] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, Nov. 2005.
- [19] M. U. Iftikhar and D. Weyns. Activforms: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014*, pages 125–134, New York, NY, USA, 2014. ACM.
- [20] J. Kabanov. JRebel tool demo. *Electron. Notes Theor. Comput. Sci.*, 264(4):51–57, Feb. 2011.
- [21] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16:1293–1306, Nov. 1990.
- [22] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proc. 19th Symp. and 13th European Conf. on Foundations of software engineering, ESEC/FSE '11*, pages 245–255, New York, NY, USA, 2011. ACM.
- [23] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN Not.*, 41(6):72–83, June 2006.
- [24] U. of Paderborn. Scenariotools project homepage, 2015.
- [25] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates deri-

- ved from specification changes. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 63–72. IEEE Press, 2013.
- [26] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceeding of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)*, 2013.
- [27] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33:856–868, Dec. 2007.
- [28] D. Weyns, S. Shevtsov, and S. Pllana. Providing assurances for self-adaptation in a mobile digital storytelling application using activforms. In *Self-Adaptive and Self-Organizing Systems (SASO), 2014 IEEE Eighth International Conference on*, pages 110–119, Sept 2014.
- [29] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proc. 28th Intl. Conf. on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.
- [30] J. Zhang, H. J. Goldsby, and B. H. Cheng. Modular verification of dynamically adaptive systems. In *Proc. 8th Intl. Conf. on Aspect-oriented software development*, AOSD '09, pages 161–172, New York, NY, USA, 2009. ACM.