UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

PROGETTAZIONE DI UN'ARCHITETTURA MODULARE, APERTA ED IN TEMPO REALE PER UN ROBOT MOBILE

Relatore:
Chiar.mo Prof. STEFANO CASELLI

Correlatori:
Dott. Ing. MONICA REGGIANI
Dott. Ing. FRANCESCO ZANICHELLI

Tesi di laurea di:
FRANCESCO MONICA

29 Aprile 2003

Ai miei genitori,
Marcello e Natalia.

Lo svolgimento della tesi è senza dubbio il periodo più bello nel corso degli studi universitari, e mettere assieme nei pochi minuti che mancano alla mia "deadline" i nomi di tutti quelli che mi hanno aiutato durante questi ultimi mesi intensi e piacevoli non è facile.

Ringrazio prima di tutto il mio relatore, il Prof. Caselli, che mi ha permesso di realizzare questo progetto, fornendomi in numerose occasioni i consigli indispensabili per andare avanti e sopportando pazientemente la mia capacità innata di sottostimare i tempi per fare le cose...

Ringrazio l'ing. Zanichelli per l'aiuto che mi ha dato durante il lavoro con il robot, facendomi conoscere la sua profonda esperienza nel campo della robotica mobile e non solo.

Ringrazio l'ing. Reggiani per tutte le ore passate discutendo (animatamente!) su quale fosse il modo migliore per sviluppare quello che mi veniva in mente, e ringrazio la Monica per l'amicizia.

Ringrazio Pietro, che ha condiviso con me le gioie e i dolori dell'operazione di messa a punto del Nomad, e con lui Nicola, Davide, Francesco e tutti quelli che ci hanno aiutato, anche solo per stringere una vite!

Ringrazio Daniele e l'ing. Rimassa, che mi hanno dato un sacco di buoni consigli (ed un'ottima libreria!) per realizzare il framework "ad oggetti".

Ringrazio Massimo, il regista del filmato "le mirabolanti imprese di un robot", e poi Jacopo, Michele, Stefano e tutti gli studenti e i laureandi del laboratorio di robotica e della palazzina 1.

Saluto infine tutti gli amici con cui ho condiviso i momenti di "pausa", visto che anche loro hanno contribuito alla mia tesi risollevandomi il morale quando le cose proprio non ne volevano sapere di funzionare e tenendomi sveglio con i tanti caffè!

“I love deadlines. I like the whooshing sound they make as they fly by.”

Douglas Adams

Indice

1 Introduzione **1**

1.1 Evoluzione delle architetture 2
1.2 Stato dell'arte 4
1.3 Organizzazione della tesi 10

2 Il Nomad 200 **12**

2.1 Il robot originale 12
2.1.1 Movimento 13
2.1.2 Sensorialità *propriocettiva* 14
2.1.3 Sensorialità *eterocettiva* 15
2.1.4 Elaborazione 20
2.2 Limiti dell'hardware 23

3 Aggiornamento del robot **27**

3.1 Sostituzione dell'hardware 27
3.1.1 Sistema di elaborazione 28
3.1.2 Hardware di interfacciamento 29
3.1.3 Connessione di rete 30
3.1.4 Sintesi vocale 31
3.1.5 Alimentazione 32
3.2 Adeguamento del software 34
3.3 Limiti del software *robotd* 35

4 Progetto del framework **37**

4.1 Obiettivi 37

4.2 Strumenti disponibili 39
4.2.1 Moduli 39
4.2.2 Attività 41
4.2.3 Comunicazioni 45

5 Realizzazione del framework **60**
5.1 Scheduling Real Time 60
5.2 Comunicazioni 66
5.3 Evoluzione 69

6 Sperimentazione **71**
6.1 Interfaccia sensomotoria 71
6.1.1 Sensori Sonar 71
6.1.2 Sensori Infrarossi 73
6.1.3 Motori 73
6.2 Test del comportamento 74
6.2.1 Verifica della reattività 74
6.2.2 *Wall Following* 76

7 Conclusioni **81**

Appendici **83**

A Circuiti di accensione e spegnimento **83**

B Codice dei moduli per il Nomad 200 **85**
B.1 Interfacce sensomotorie 86
B.2 *Behaviour* per il robot 89

Bibliografia **91**

Capitolo 1

Introduzione

La rapida evoluzione dei dispositivi elettronici, che permettono di costruire strumenti di calcolo sempre più potenti e versatili, ha consentito all'informatica di progettare sistemi complessi ed articolati in cui è possibile identificare a più livelli numerose entità indipendenti ed autonome, hardware o software, che collaborano per l'esecuzione di un compito comune. Lo sviluppo dell'*Ingegneria del Software*, che nasce dalla necessità di dominare la complessità intrinseca dei sistemi di grandi dimensioni, enfatizza in modo particolare l'esigenza di produrre software caratterizzato da un alto grado di riuso e interoperabilità, per consentirne l'utilizzo in contesti molteplici ed in continua evoluzione.

Anche la *Robotica* mostra un crescente interesse verso la ricerca di strumenti che, favorendo lo sviluppo di applicazioni aperte e modulari, possono evolvere grazie all'apporto progressivo di hardware e software di differente provenienza. La crescita dei settori legati alla robotica di servizio, ed alla *domotica* in particolare, ha sancito l'utilizzo della robotica *intelligente* al di fuori del campo accademico, dirigendo l'attenzione verso nuovi problemi diventati recentemente di importanza cruciale. La necessità di realizzare soluzioni a basso costo che siano nel contempo facili da utilizzare per l'utente finale, ha avuto l'effetto di spingere le aziende ad abbandonare progressivamente l'utilizzo di protocolli proprietari, largamente impiegati nell'hardware e nel software fino a pochi anni fa, in favore dell'uso di standard aperti che agevolano l'interconnessione delle diverse componenti del sistema e ne facilitano la sostituzione quando nuove versioni le rendono obsolete.

Questa tesi è stata realizzata con l'intento di aggiornare il robot mobile *Nomad 200* disponibile presso il Laboratorio di Robotica del Dipartimento di Ingegneria dell'Informazione, per poter usufruire di un livello software di supporto che consenta di facilitare lo sviluppo di applicazioni specifiche per il robot, garantendo nel contempo un buon grado di generalità. Oltre al progetto di un nuovo sistema software di controllo si è provveduto a sostituire parte dell'hardware elettronico a bordo del robot per migliorare la potenza computazionale e l'aderenza ai nuovi standard di comunicazione wireless.

1.1 Evoluzione delle architetture

Il contesto in cui viene progettato un sistema robotico è per sua natura eterogeneo e complesso: la necessità di operare direttamente sul mondo reale impone vincoli temporali stringenti sui tempi di risposta del sistema, e la percezione dell'ambiente è affetta da errori di misura che devono essere accuratamente considerati durante la realizzazione degli algoritmi. Per realizzare compiti complessi, o elaborare informazioni percettive provenienti dai sensori più evoluti, sono inoltre necessari algoritmi dalla forte componente computazionale che, imponendo lunghi tempi di elaborazione, mal si adattano alla natura dinamica dell'ambiente di utilizzo. Per dominare i problemi in questione e far convivere più componenti dissimili per realizzare lo scopo comune, sono state proposte numerose architetture che hanno seguito un'evoluzione continua nel corso degli ultimi decenni.

Le prime proposte, provenienti per lo più dall'intelligenza artificiale, vanno sotto il nome di *architetture deliberative* [1], e tentano di scomporre la procedura di controllo in tre fasi distinte, che saranno eseguite in modo sequenziale:

- *Sense*: percezione dell'ambiente, che viene rappresentato internamente mediante un modello logico-matematico;
- *Plan*: decisione dei compiti che il robot dovrà eseguire, in genere ricavata tramite algoritmi di pianificazione e inferenza logica;
- *Act*: intervento sul mondo reale.

La rappresentazione simbolica dell'ambiente, unita alle tecniche di ragionamento logico, è un potente strumento per realizzare compiti complessi e per consentire l'utilizzo di tecniche di apprendimento, ma presenta alcuni limiti importanti che hanno fanno naufragare il puro approccio deliberativo applicato alla robotica. La natura fortemente dinamica del mondo reale tende a fare decadere velocemente la validità del modello, che deve essere continuamente ricostruito perché le fasi di pianificazione possano portare a risultati corretti: gli algoritmi di *sensing* e *planning* sono però piuttosto onerosi, e impediscono al robot di compiere rapidamente i compiti più semplici, riducendone in modo eccessivo la reattività.

Verso la metà degli anni '80, rompendo nettamente con il passato, ha riscosso molto successo l'introduzione di *architetture reattive* per il controllo di sistemi robotici [2]: per garantire una veloce risposta agli stimoli sensoriali si rinuncia ad una rappresentazione complessa dell'ambiente e vengono cablate nel codice una serie di regole di *stimolo-risposta* che consentono al robot di rispondere immediatamente alle situazioni più significative che vengono riconosciute dall'apparato sensoriale. Attraverso un insieme di moduli, intrinsecamente concorrenti, che operano a differenti livelli di priorità, viene portato avanti uno scopo comune, garantendo potenzialmente un ampio margine di robustezza alle situazioni impreviste che si possono presentare. Pur offrendo notevoli vantaggi legati alla semplicità ed alla scarsa necessità di risorse computazionali, le architetture di tipo puramente reattivo sono fortemente limitate nella complessità dei compiti che possono svolgere dall'assenza quasi totale di uno stato interno, e dall'incapacità di pianificare azioni complesse e di apprendere.

Uno sviluppo interessante di questa tecnica di progetto è rappresentato dall'architettura *subsumption*, presentata per la prima volta in [3]: il sistema è costituito da un certo numero di *comportamenti* (in inglese *behaviours*) che, eseguiti in parallelo, leggono i dati sensoriali e comandano gli attuatori in modo indipendente l'uno dall'altro. Lo sviluppo dell'architettura avviene con una tecnica *bottom-up*, partizionando i comportamenti in una serie di livelli numerati che saranno sviluppati separatamente: i comportamenti di livello 0 svolgono i compiti più elementari, che possono essere realizzati e collaudati autonomamente, ed i comportamenti dei livelli superiori eseguono via via operazioni più evolute, che sono in grado di ini-

bire o utilizzare dove serve (sussumere) le funzioni dei livelli inferiori. I progetti sviluppati facendo uso di questo nuovo approccio hanno mostrato come sia possibile fare eseguire al robot compiti evoluti anche senza l'utilizzo di rappresentazioni complesse dell'ambiente, ma sfruttando la nascita di comportamenti che emergono dall'interazione tra i moduli contenuti nei singoli livelli [4].

Per superare i problemi ed unire i vantaggi degli approcci deliberativo e reattivo sono state presentate tecnologie miste in cui coesistono comportamenti separati ma interconnessi che mostrano le caratteristiche dell'una o dell'altra tecnica: i sistemi che vengono realizzati prendono il nome di *architetture ibride* [5, 6]. In esse in genere è presente un *livello inferiore*, composto da processi reattivi che realizzano i compiti più semplici, ed un *livello superiore* che, mediante algoritmi complessi ed in genere più lenti, decide la strategia di lungo termine che il robot deve attuare per raggiungere lo scopo. L'attivazione dei comportamenti reattivi viene controllata dal livello superiore, che vede il livello inferiore come un insieme di moduli elementari da utilizzare per svolgere delle attività più complesse. Molto spesso, per facilitare la coesistenza dei due sistemi, viene introdotto un *livello intermedio* che ha il compito di ricevere comandi dal sottosistema deliberativo e gestire la sequenzializzazione dei componenti reattivi che devono essere attivati. In pratica, questo livello ha il compito di costruire dinamicamente un grafo di azioni elementari che vengono eseguite in sequenza o in concorrenza parziale tramite il livello inferiore [7].

1.2 Stato dell'arte

La crescita di complessità delle architetture software per la robotica ha seguito di pari passo la crescita dei sistemi a cui corrispondono; in generale ogni gruppo di ricerca tende a proporre soluzioni proprie che riflettono visioni personali e innovative pur modellando i medesimi concetti di base discussi nel paragrafo precedente. La maggior parte dei sistemi proposti in lettaratura si basano su strutture di tipo *behaviour-based* che evolvono verso un modello ibrido (suddiviso in tre macro livelli) quando la complessità dei compiti che si vuole svolgere diventa elevata.

Per comprendere meglio la struttura tipica di un moderno sistema robotico viene illustrata brevemente l'architettura progettata presso il *Laboratory for Analysis*

and Architecture of Systems (LAAS/CNRS) [8] per la robotica mobile. Lo schema generale, illustrato in figura 1.1, mette in evidenza la rigida stratificazione che viene compiuta sia sulle componenti software che sul resto del sistema, in particolare sono presenti i tre livelli tipici delle architetture ibride:

- *Functional Level*. Include un insieme di azioni elementari che il robot è in grado di compiere (*image processing*, *obstacle avoidance*, *motion control*, etc.), incapsulate in unità autonome e controllabili chiamate *Moduli*. Per garantire a questo livello un certo grado di indipendenza rispetto all'hardware e rendere i moduli portabili verso altri robot, l'architettura inserisce uno strato software di interfaccia verso il sistema fisico: il *logical robot level*.

- *Execution control Level*. Controlla e coordina l'esecuzione della funzioni distribuite nei moduli del livello inferiore in accordo con le sequenze di comandi che vengono inviate dal livello superiore. Questo livello è composto di un unico sistema, l'*Executive*, che funge da interfaccia tra il sistema reattivo, operante a frequenze elevate ($10 \div 100$ Hz), ed il sistema deliberativo, i cui processi hanno tempi di elaborazione tipicamente superiori a 100 ms.

- *Decision Level*. Contiene gli algoritmi necessari alla pianificazione delle operazioni che il robot deve svolgere ed alla supervisione della loro esecuzione. In base alla complessità dei compiti per i quali il sistema è realizzato questo livello può essere suddiviso in più strati che, operando a frequenze diverse, forniscono funzioni via via più evolute ed astratte: il livello superiore ha il compito di dialogare con l'operatore.

Per facilitare lo sviluppo delle componenti dell'architettura, realizzata prevalentemente in linguaggio *C*, vengono utilizzati alcuni strumenti che provvedono a costruire il codice necessario partendo da una descrizione di alto livello realizzata mediante linguaggi definiti appositamente. Il codice associato ai singoli moduli viene infatti prodotto tramite un tool, denominato *Genom*, a partire da una descrizione formale dei servizi che esporta e dall'insieme degli algoritmi che li rendono realizzabili (chiamati *codel*). Per descrivere il comportamento dell'Executive è stato sviluppato un altro linguaggio formale (*Kheops*) ed un tool che provvede a tradurne

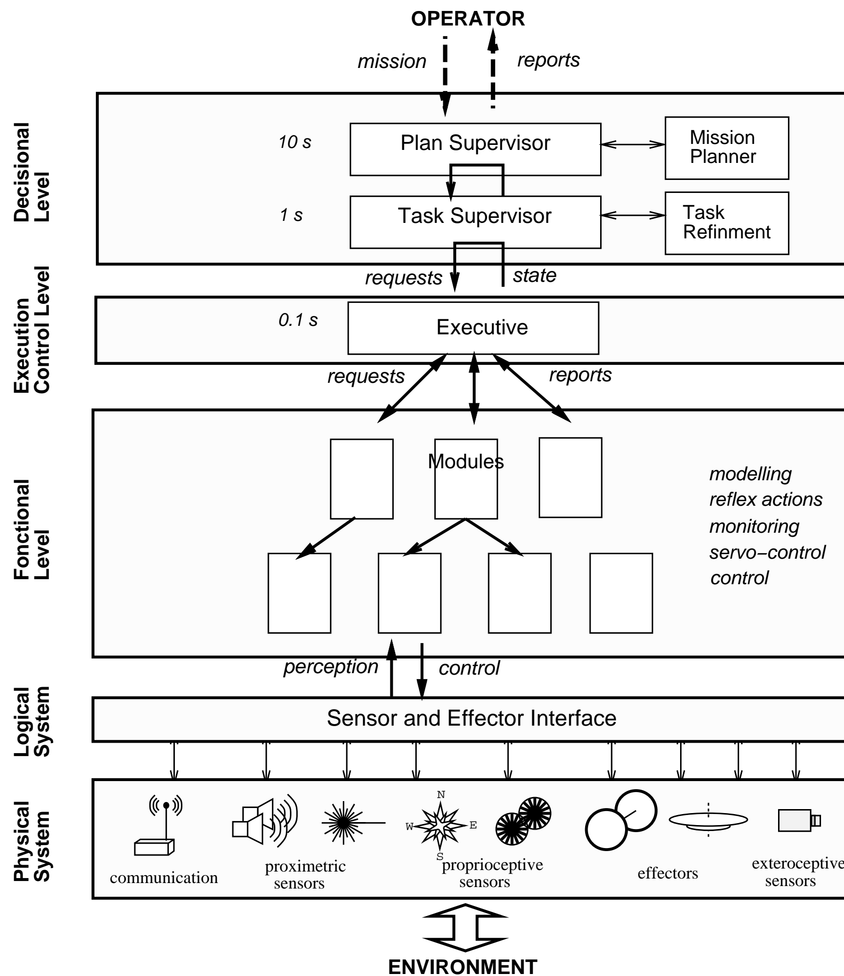


Figura 1.1: L'architettura sviluppata presso i laboratori del *LAAS/CNRS*.

le regole in codice *C* compilabile.

Un altro sistema attentamente studiato durante questo lavoro di tesi è il framework in corso di sviluppo nell'ambito del progetto *OROCOS* [9], nato con l'intento di produrre un'architettura *Open Source* che fornisca una base comune per lo sviluppo di applicazioni sia per robot mobili che per manipolatori. Molte delle architetture esistenti vengono realizzate all'interno di un solo gruppo di ricerca, pertanto è piuttosto comune che la struttura del sistema sia molto influenzata dal contesto originale di utilizzo e poco portabile verso altri sistemi. Al contrario, OROCOS ha come scopo principale la ricerca di soluzioni quanto più possibile generali ed aper-

te, che sfruttano fin dove è possibile gli standard a disposizione. Si è ricorso quindi al linguaggio *C++* per sviluppare le varie parti del sistema, favorendo l'uso di librerie portabili come *POSIX* ed *ACE* per mantenere un alto grado di indipendenza dalla piattaforma di esecuzione, e si è scelto di realizzare in *CORBA* la maggior parte degli schemi di comunicazione, per permettere l'interoperabilità tra componenti eterogenei.

Il progetto, che si trova attualmente nelle prime fasi di sviluppo, è suddiviso in sotto-sistemi che vengono portati avanti separatamente da alcuni gruppi di ricerca. Il nucleo del sistema, realizzato presso la *Katholieke Universiteit Leuven*, è rappresentato dal *Real-time motion kernel*: si tratta di un insieme di librerie che consentono l'inserimento dei moduli di controllo delle componenti hardware del robot all'interno di un ambiente che ne gestisce l'attivazione periodica in tempo reale. La libreria realizza in pratica uno strato di interfaccia verso il sistema operativo utilizzato (attualmente *RTLinux* e *RTAI*), fornendo gli strumenti di base per la programmazione concorrente in ambiente real-time, come primitive di sincronizzazione e task ad attivazione periodica. A questo livello la comunicazione tra le componenti, per garantire un'efficienza elevata, avviene mediante meccanismi a memoria condivisa.

Il *Task execution sequencing*, che ha lo scopo di coordinare l'esecuzione delle componenti di livello *kernel*, è realizzato dal laboratorio *LAAS/CNRS* [10], che può contare sull'esperienza proveniente da numerosi progetti, tra cui l'architettura descritta in precedenza. In particolare il gruppo francese prevede di riutilizzare all'interno di OROCOS il tool di sviluppo real-time *Genom* [11].

Un'enfasi particolare è stata riservata per il sottosistema che gestisce le comunicazioni tra i componenti dell'architettura: presso l'istituto di ricerca *FAW* (Monaco) è stato condotto un minuzioso lavoro di analisi dei requisiti che un'applicazione robotica presenta in termini di primitive di comunicazione, producendo un documento che descrive l'insieme di *pattern di comunicazione* che possono diventare utili in un sistema complesso [12]. Da questo progetto sono nate separatamente le due librerie *Orocos::SmartSoft* [13] e *Orocos@KTH* [14], sviluppate rispettivamente presso il già citato *FAW* e l'istituto svedese *KTH*, che forniscono due differenti soluzioni concrete, entrambe realizzate al di sopra di un *middleware* CORBA.

Il progetto OROCOS è tuttora incompleto, ed una scarsa coesione tra i gruppi che lo compongono sembra allontanare progressivamente risultati significativi a

breve termine; tuttavia lo sforzo progettuale che è stato effettuato fino ad ora ha messo in luce alcuni aspetti di notevole importanza, che sono stati attentamente considerati durante questo lavoro di tesi. In particolare l'apertura verso la filosofia di sviluppo *Open Source*, specialmente se viene favorita da un sistema che esalta la granularità dei componenti, consente un'evoluzione progressiva che può sfruttare a vari livelli l'apporto di persone che non sono direttamente legate al progetto. L'enfasi sulla *granularità* del sistema e delle sue componenti è del resto ben visibile in molte architetture moderne: è forte l'esempio di *CLARAty* [15, 16], che, pur seguendo un approccio di tipo ibrido, propone di eliminare il livello intermedio, monolitico e difficile da modificare, per favorirne la scomposizione all'interno dei moduli che compongono gli altri livelli. È ugualmente interessante la soluzione adottata nella realizzazione di *OSCAR* [17] che promuove lo sviluppo di moduli distribuibili anche in forma binaria, per formare una sorta di database di comportamenti che può crescere progressivamente.

La ricerca di una fine scomposizione dei componenti del sistema robotico ha avuto l'effetto ulteriore di far crescere l'interesse del settore per le tecniche di design e programmazione orientate agli oggetti: l'approccio *object-oriented* si presta particolarmente a rappresentare entità autonome che collaborano in un ambiente complesso ed eterogeneo, quindi può essere particolarmente efficace per realizzare le componenti di un'applicazione robotica (è immediato il confronto con il concetto di *behaviour*) in cui devono convivere problemi legati alla concorrenza, alla distribuzione su di un sistema piuttosto vasto, all'esecuzione in tempo reale, ed in cui un approccio rigoroso è il più delle volte indispensabile [18, 19, 20, 21]. I linguaggi di programmazione orientati agli oggetti possiedono inoltre una maggiore potenza espressiva rispetto ai linguaggi procedurali: questo consente molto spesso di sfruttare direttamente i costrutti del linguaggio per costruire l'architettura del sistema, riducendo la necessità di linguaggi creati ad hoc per descrivere le componenti dell'applicazione. In questo senso è emblematico l'esempio di *Genom*: nonostante i singoli algoritmi di controllo (codel) vengano realizzati in linguaggio *C* per le specifiche formali del modulo è stato introdotto un ulteriore linguaggio descrittivo, delegando ad un tool automatico la produzione del codice *C* definitivo che contiene alcune parti di tipo generale, provenienti da un *module skeleton*, e il codice dei

codel. Un linguaggio maggiormente espressivo come il *C++* può essere utilizzato *a due livelli* [22] per sviluppare l'architettura, permettendo di costruire un insieme di componenti di libreria molto più versatili ed estendibili, che possono essere utilizzati durante lo sviluppo dell'applicazione concreta.

La ricerca di un possibile riuso del software che la programmazione ad oggetti intende facilitare può essere realizzata mediante numerose tecnologie che sono state sviluppate nel corso degli anni, ma la robotica si è mostrata molto sensibile in particolare all'utilizzo di *Framework* come strumento per costruire sistemi aperti ed estendibili: OROCOS ne è un esempio, ed ulteriori riferimenti si possono trovare in [16] e [23].

Delle possibili definizioni di *framework* presenti in letteratura, vengono citate le seguenti (provenienti da [24]):

> *A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.*
>
> *A framework is the skeleton of an application that can be customized by an application developer.*

Sfruttando le funzionalità tipiche dei linguaggi object-oriented, come l'ereditarietà e la programmazione generica, è possibile realizzare una libreria che contiene tutte le parti immutabili dell'architettura (*frozen spots*) ma che lascia la possibilità all'utente di realizzare codice specifico da inserire in alcuni punti, lasciati aperti dalla struttura (*hot spots*). È quindi possibile pensare al framework come ad uno scheletro sul quale si può realizzare con maggiore facilità un'applicazione specifica, sfruttando i pattern di design [25] che mette a disposizione. Per molti versi un framework è simile come potenza espressiva ad un *Application generator* [26], che costruisce un'applicazione concreta a partire da un linguaggio di descrizione di alto livello.

Un altro aspetto importante per le architetture di controllo riguarda l'utilizzo di specifiche di tempo reale per i processi che operano all'interno delle applicazioni. Gli algoritmi di controllo dell'hardware di attuazione e la necessità intrinseca nel problema di lavorare nel mondo reale fanno di questo elemento un punto centrale

per garantire un buon comportamento dinamico del robot e il necessario grado di sicurezza. L'esistenza contemporanea di numerosi task che devono operare in tempo reale, cioè soggetti ad un tempo massimo di esecuzione, è in generale un problema complesso, la cui soluzione, per fornire garanzie formali di schedulabilità dei processi, richiede un approccio matematico rigoroso e impone alcuni vincoli sulle componenti hardware e software del sistema.

Quasi tutti i sistemi analizzati durante il lavoro di tesi, pur accennando all'utilizzo di processi real-time, non forniscono una documentazione dettagliata delle scelte effettuate; nei pochi casi in cui le informazioni sono presenti l'uso di politiche di scheduling evoluto, come *Rate Monotonic (RMA)* o *Earliest Deadline First (EDF)*, è per lo più ignorato in favore di un assegnamento fisso delle priorità che viene delegato a chi realizza l'applicazione concreta. Solo l'architettura *Ethnos* [27], sviluppata presso l'Università di Genova, fornisce nativamente uno scheduler dinamico con politica di tipo *RMA* per i processi che vengono eseguiti all'interno del *kernel* dell'architettura.

1.3 Organizzazione della tesi

Partendo dalle motivazioni che hanno dato spunto a questo lavoro di tesi, dopo lo studio della letteratura del settore, mi accingo a descrivere le fasi del progetto realizzato, che saranno illustrate nei prossimi capitoli.

Nel *capitolo 2* è illustrato il robot *Nomad 200* nella sua dotazione originale di hardware e software; vengono messi in luce i pregi e i difetti oggettivi dell'architettura, alla luce dell'evoluzione degli strumenti hardware a disposizione e dell'esperienza maturata con la realizzazione delle precedenti applicazioni per il robot presso il nostro Dipartimento.

Nel *capitolo 3* viene descritto il lavoro di *upgrade* effettuato sul robot che ha coinvolto in modo particolare il sistema di elaborazione di bordo e parte del software di controllo. Questo ha consentito di disporre di una piattaforma aggiornata agli standard attuali per realizzare applicazioni di robotica mobile.

I *capitoli 4 e 5* espongono l'analisi e la realizzazione di un framework per la costruzione di applicazioni *behaviour-based* che fornisce un insieme di primitive di alto livello per la creazione dei comportamenti, la gestione delle comunicazioni

tra essi e le problematiche di scheduling real-time. Il sistema è stato progettato in previsione di una sua possibile estensione verso architetture di tipo ibrido.

Nel *capitolo 6* è riportato il risultato ottenuto costruendo un comportamento di *WallFollowing* per il Nomad con il framework progettato in precedenza. L'applicazione ha lo scopo di verificare il funzionamento di tale libreria in un contesto di utilizzo reale.

La tesi si conclude (*capitolo 7*) con una breve discussione sui risultati ottenuti e sulle possibili evoluzioni del progetto.

Capitolo 2

Il Nomad 200

2.1 Il robot originale

Il *Nomad 200* è un robot mobile che la società statunitense *Nomadic Technologies Inc.* produceva durante la prima metà degli anni '90 assieme ad una vasta gamma di supporti per la robotica mobile. Recentemente la società è stata assorbita da una importante compagnia operante nel settore delle telecomunicazioni, ed ha perciò interrotto ricerca e produzione dei propri prodotti per la robotica.

La famiglia di robot mobili *Nomad* ha indubbiamente giocato un ruolo rilevante nella didattica e nella ricerca dell'ultimo decennio: lo dimostrano l'ampio utilizzo dei modelli *Scout*, *200* e *XR4000* nelle università europee e d'oltre oceano, come spesso si evince dalla letteratura specifica del settore robotico e dell'intelligenza artificiale, e la palese influenza che questi modelli hanno avuto sull'evoluzione dei supporti per la robotica mobile progettati dalle altre aziende negli anni seguenti.

Il Nomad 200 [28] è un robot di medie dimensioni che presenta una dotazione abbondante di sensori ed attuatori particolarmente adatti ad un utilizzo in ambienti chiusi e strutturati[1]. Il robot, di cui è presentata un'immagine in figura 2.1, è stato realizzato in modo tale da contenere un'unità autonoma di percezione, elaborazione e movimento in un ingombro complessivo relativamente ridotto, avente forma cilindrica di 50 cm di diametro e 80 cm di altezza (fino al piano superiore).

[1] In gergo si usa spesso il termine inglese *indoor environment*.

Figura 2.1: Il robot mobile Nomad 200.

2.1.1 Movimento

Il robot è sostenuto da tre ruote di identica dimensione, illustrate nello schema di figura 2.2, fissate in posizione simmetrica alla base della struttura. Il moto traslazionale del robot è prodotto dal movimento solidale delle ruote, a cui viene trasmessa l'energia di un unico motore elettrico mediante un sistema di cinghie. La direzione lungo cui si sposta il robot viene modificata da un secondo motore che ruota contemporaneamente l'asse di ogni ruota in modo sincrono. La parte superiore del robot (*torretta*) può ruotare in modo completamente indipendente rispetto alla base mediante un terzo motore posto nella parte inferiore.

Gli organi meccanici descritti, controllati da una scheda dedicata *Galil DMC-630* [29], consentono al robot di raggiungere velocità massime di spostamento e

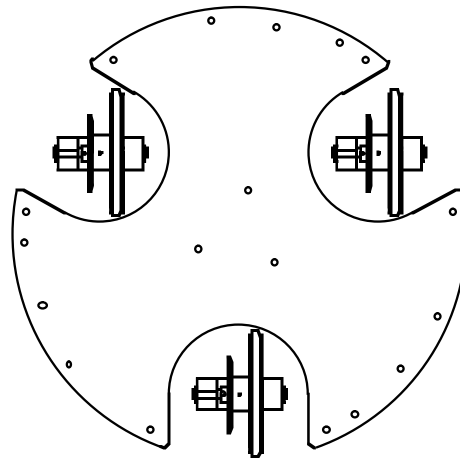


Figura 2.2: Le ruote (vista dal basso).

rotazione pari rispettivamente a 50 $\mathrm{cm/s}$ e 45 $^{\circ}/\mathrm{s}$.

Il sistema di locomozione, che evidentemente non può essere considerato *olonomo*[2], consente comunque al Nomad di ruotare attorno al proprio centro geometrico senza che sia necessario spazio aggiuntivo oltre al normale perimetro del robot.

La potenza necessaria al robot per muoversi è fornita da due batterie a 12 V collegate in serie ed alloggiate all'interno della base che forniscono in totale 288 Wh. L'assorbimento di corrente dei motori è stato misurato nel corso di questa tesi: alla massima velocità degli assi, mossi singolarmente, le batterie forniscono 300 mA per consentire la rotazione della torretta, 900 mA per la rotazione delle ruote e 3.5 A per il moto traslatorio; a robot fermo la corrente assorbita sull'alimentazione a 24 V scende a 150 mA.

2.1.2 Sensorialità *propriocettiva*

2.1.2.1 Odometria

L'elettronica di controllo dei servomeccanismi fornisce un insieme di primitive fondamentali necessarie al calcolo dell'odometria sulla posizione del robot: il sistema di elaborazione centrale contenuto all'interno del Nomad si avvale delle informa-

[2] Si può notare come la direzione di rotazione delle ruote, istante per istante, impone un vincolo alla direzione del moto traslatorio.

zioni provenienti dagli organi di movimento per produrre una stima in tempo reale delle velocità istantanee di traslazione e rotazione del robot. Tali grandezze vengono ulteriormente combinate ed integrate nel tempo per determinare il percorso compiuto sul piano durante il normale ciclo di funzionamento. La posizione corrente del robot, calcolata con una risoluzione temporale di 18.2 Hz, ne individua le coordinate (x, y) sul piano e l'orientazione della base rispetto all'asse delle ascisse con una precisione di circa 2.5 mm sugli assi traslazionali e 0.1° per quelli rotazionali.

Contemporaneamente alle misure odometriche il robot fornisce alcune informazioni sullo stato attuale del proprio funzionamento, che comprendono la verifica di eventuali condizioni di stallo dei motori e lo stato di carica delle batterie.

2.1.2.2 Bussola

Sulla sommità del robot è posizionata una bussola elettronica *KVH C100* che rappresenta un sensore assoluto di orientazione rispetto al nord magnetico. Il principio di funzionamento dello strumento consiste nella misura elettronica dell'angolo formato tra un piccolo nucleo magnetico posto in fluttuazione libera all'interno del suo contenitore e la sua posizione a riposo. Un piccolo microprocessore dedicato interpreta le informazioni provenienti dall'ago magnetico e produce una stima dell'angolazione della torretta del Nomad con una precisione dell'ordine di 0.3°. La correttezza della misura è comunque fortemente influenzata dalla presenza di campi magnetici indesiderati nelle vicinanze del robot.

2.1.3 Sensorialità *eterocettiva*

2.1.3.1 Sensori di contatto

Lo sviluppo geometrico del Nomad 200 è realizzato in modo tale da proteggere quanto più possibile le componenti più delicate dalle eventuali collisioni che possono avvenire con gli ostacoli incontrati durante il movimento. Per questo motivo la base del robot presenta un diametro maggiore rispetto alla torretta ed è rivestita da due anelli sporgenti di gomma antiurto, visibili nel dettaglio riportato in figura 2.3. All'interno dello strato di gomma sono inseriti 20 micro interruttori (10 per ogni anello) che individuano la presenza di un oggetto che si trovi a contatto del

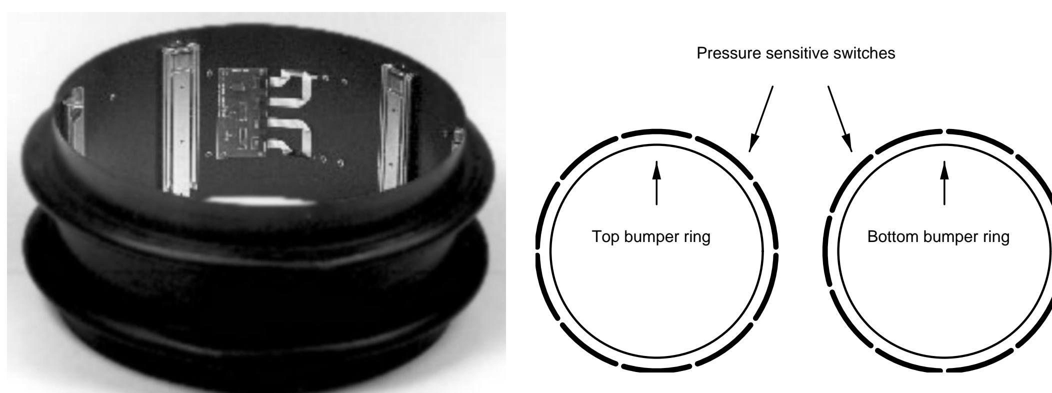


Figura 2.3: Sensori di contatto: *bumper*.

robot; la disposizione di tali sensori, detti *sensori di pressione* o più semplicemente *bumper*, consente di rilevare la presenza di oggetti lungo tutto il perimetro del robot e può essere efficacemente utilizzata per realizzare operazioni che ne coinvolgano lo spostamento (*object pushing*). Le specifiche della Nomadic per l'anello di bumper, denominato *Sensus 100*, riportano per ogni switch una copertura di 18° ed una sensibilità pari a circa 2N.

2.1.3.2 Sensori di prossimità

Nella sommità della torretta è alloggiato un sistema sensoriale in grado di rilevare la presenza di ostacoli nello spazio di movimento che si basa su un totale di 16 sensori sonar ultrasonici *Polaroid* [30] controllati dalla scheda dedicata *Polaroid 6500*; il modulo[3] *Sensus 200*, illustrato in figura 2.4, consente di misurare correttamente le distanze nell'intervallo $30 \div 600$ cm con una precisione stimata pari all'1%. Il *beam pattern* riportato in figura 2.4 (realizzato ad una frequenza di 49.4 kHz) mostra come un singolo trasduttore sonar sia in grado di illuminare e rilevare efficacemente gli oggetti che si trovano entro un cono massimo di circa 25°: i 16 sensori, che risultano sfasati tra loro di 22.5°, riescono quindi a coprire l'intero perimetro del robot.

I sonar utilizzati sfruttano la misura del *tempo di volo* di un treno di impulsi per

[3] Si può notare come le componenti sensoriali del Nomad 200 siano organizzate in *strati* separati e modulari: in effetti la Nomadic forniva il robot sotto forma di piattaforma base a cui è possibile applicare separatamente i vari gruppi di sensori nella misura necessaria all'applicazione che si desidera realizzare.

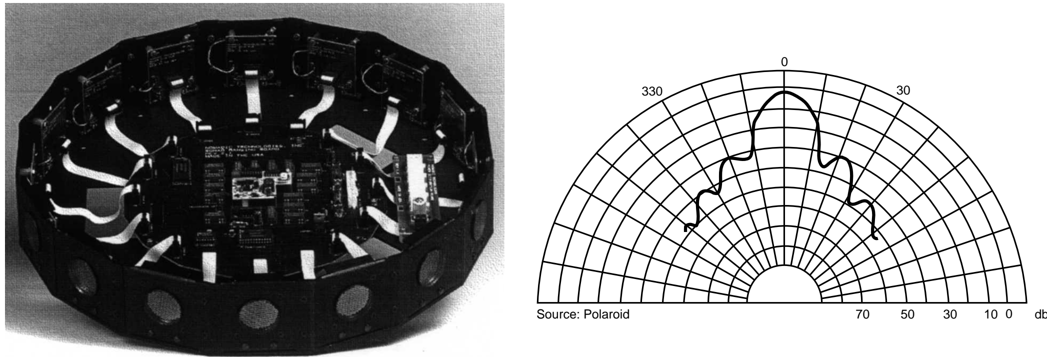


Figura 2.4: Sensori di prossimità: *sonar*.

identificare la distanza di un eventuale ostacolo. Dal momento che il medesimo trasduttore elettrostatico viene utilizzato sia per produrre il segnale che per riceverne l'eco di ritorno, diventa impossibile rilevare oggetti molto vicini perché gli echi prodotti si sommano ai fenomeni di risonanza propri del sensore. Il circuito stampato a cui sono connessi direttamente i 16 sensori realizza un *multiplexing* che consente di collegare alternativamente ogni sonar con un solo circuito di comando Polaroid posto al centro della scheda: da software è possibile abilitare selettivamente i singoli sensori e decidere l'ordine progressivo di sparo che viene ripetuto ciclicamente dall'hardware dedicato. Assieme al *firing pattern* è possibile specificare anche la durata del ciclo di ricezione che viene eseguito prima della commutazione del circuito di comando sul sonar successivo: valori bassi di tale intervallo consentono di eseguire un numero maggiore di misure nell'unità di tempo, ma l'utilizzo di tempi dilatati permette di rilevare echi provenienti da oggetti distanti, che altrimenti non possono tornare alla sorgente prima dell'invio dell'impulso successivo. L'intervallo di sparo può variare da un minimo di 4 ms ad un massimo di 1 s; il valore minimo consente di individuare ostacoli ad una distanza massima di circa 70 cm.

L'utilizzo di un solo circuito di comando per l'intero anello di sonar impedisce di attivare contemporaneamente più di un sensore alla volta: questa soluzione però, oltre a consentire un notevole risparmio di hardware, è di fatto implicita nell'uso di sensori di questo tipo, che, soffrendo fortemente di problemi di interferenza reciproca, sono generalmente inadatti ad un uso simultaneo.

Per misurare la distanza di oggetti particolarmente vicini al corpo del robot la

Nomadic mette a disposizione il modulo *Sensus 300*, inserito nella parte inferiore della torretta, che dispone di 16 emettitori/ricevitori di radiazione infrarossa in grado di rilevare ostacoli fino ad una distanza massima di circa 60 cm (figura 2.5). Ogni sensore è formato da due diodi LED emettitori ed un fotodiodo rilevatore: la

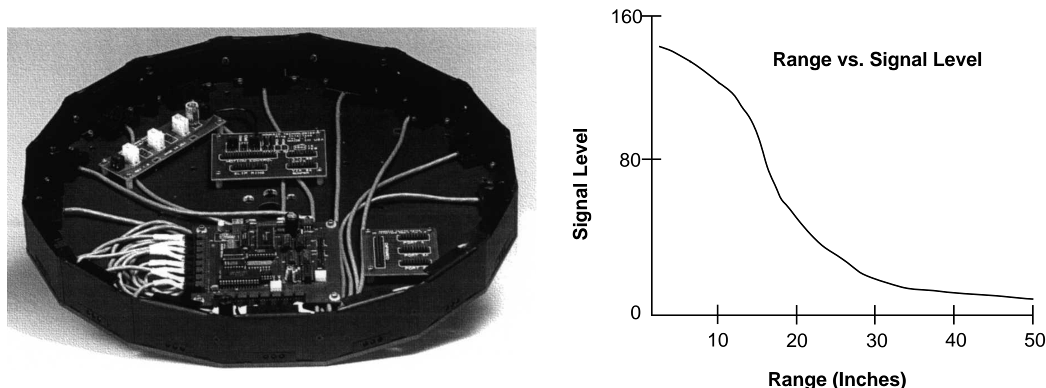


Figura 2.5: Sensori di prossimità: *infrarossi*.

distanza viene misurata a partire dall'intensità della radiazione riflessa che viene opportunamente modulata dai diodi emettitori per ridurre l'effetto del rumore presente nell'ambiente.

L'intensità della radiazione misurata dal fotodiodo è funzione della distanza della superficie riflettente dell'ostacolo, come si evidenzia dal grafico di figura 2.5, ma dipende fortemente anche dal grado di illuminazione infrarossa dell'ambiente e dal grado di riflettività delle superfici. Per rendere significativo il valore acquisito, ogni ciclo di lettura prevede due misure consecutive effettuate alternativamente con il diodo di emissione acceso e spento; questo consente di produrre una misura differenziale che riduce l'effetto del rumore. Una fase di calibrazione dei sensori in funzione dell'ambiente di utilizzo resta comunque di particolare importanza.

La misura analogica dell'intensità di radiazione viene convertita in un segnale digitale mediante un VCO (*Voltage Controlled Oscillator*) che genera un'onda quadra di frequenza proporzionale al valore misurato. Un contatore ad alta frequenza provvede a misurare il periodo di questa oscillazione per determinare una stima digitale del valore desiderato. La misura complessiva richiede un tempo pari a circa 2.5 ms.

2.1.3.3 Sistema di visione

La Nomadic nel corso degli anni ha prodotto due differenti sistemi di visione da applicare ai propri robot: seguendo la numerazione tradizionale, i due apparati sensoriali vengono identificati rispettivamente come *Sensus 400* e *Sensus 500*.

Sul robot disponibile presso il Laboratorio di Robotica (RIMLab) del Dipartimento di Ingegneria dell'Informazione è montata una unità Sensus 400 (figura 2.6), che comprende una videocamera CCD monocromatica con risoluzione di 768x494 pixel (compatibile con il formato NTSC) ed un frame grabber CX100 capace di digitalizzare frame monocromatici in scala di grigio alle risoluzioni comprese tra 243x256 e 512x512 pixel. La videocamera è montata sulla sommità di una unità

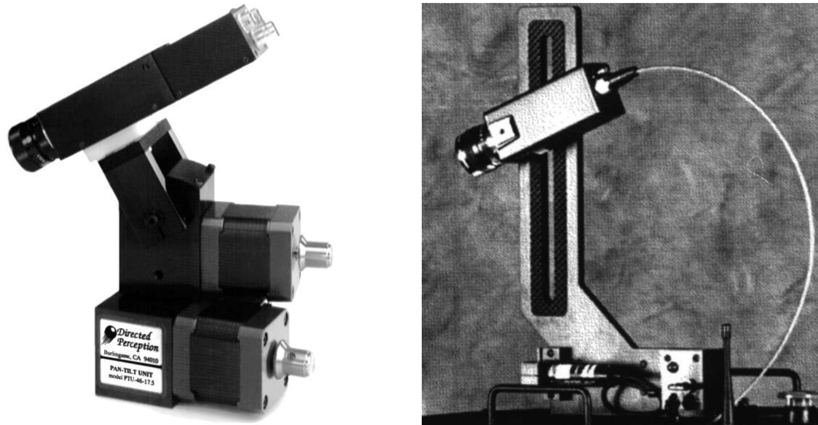


Figura 2.6: Sistema di visione: *Sensus 400* (a sinistra) e *Sensus 500* (a destra).

PTU (*Pan Tilt Unit*) fornita dalla *Directed Perception* [31], che consente un ampio movimento della visuale lungo i due assi: le specifiche meccaniche riportano una capacità di movimento compresa tra $\pm 136^\circ$ sull'asse orizzontale (asse *pan*) e $[-46^\circ, +31^\circ]$ su quello verticale (*tilt*). Mediante una semplice connessione seriale l'unità PTU può essere comandata dall'unità centrale del robot.

Il Sensus 500 (figura 2.6) costituisce un sistema percettivo strutturato, disponibile opzionalmente per i robot della Nomadic, che coniuga l'utilizzo di una videocamera CCD monocromatica ed un diodo laser utilizzato come sorgente luminosa collimata: mediante una scansione progressiva dello spazio antistante il robot, effettuata tramite il diodo laser, unitamente ad un algoritmo di triangolazione applicato

all'output della videocamera, il Sensus 500 è in grado di ricostruire la profondità delle superfici che incontra in un intervallo di distanze compreso tra 45 e 300 cm.

2.1.4 Elaborazione

2.1.4.1 Hardware

L'apparato sensoriale e di movimento del Nomad 200 viene controllato dall'unità di elaborazione del robot, illustrata in figura 2.7, che occupa la parte centrale della torretta.

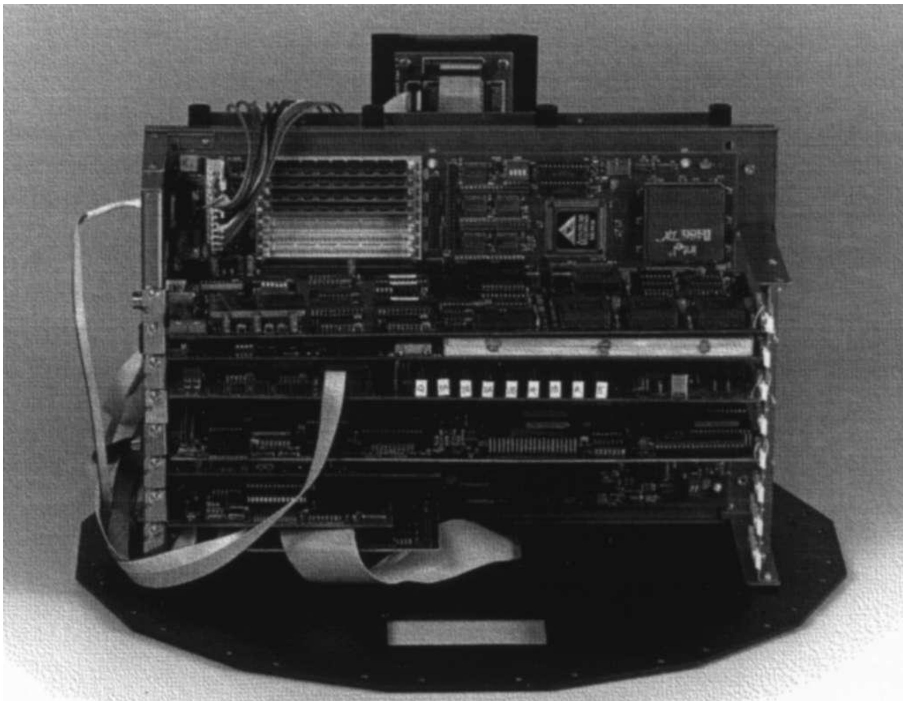


Figura 2.7: Il sistema di elaborazione del Nomad 200.

I componenti del sistema di elaborazione, schematizzati in figura 2.8, verranno ora descritti nel dettaglio. La scheda madre ospita un processore *Intel 486DX2* con frequenza di clock pari a 66 MHz, dotato di 16 MB di RAM, di un Hard Disk da 1.2 GB e del normale corredo di porte di input/output disponibili in una architettura PC AT. Sul bus ISA della motherboard sono inserite alcune schede di controllo che forniscono al PC la necessaria interfaccia verso le periferiche hardware del ro-

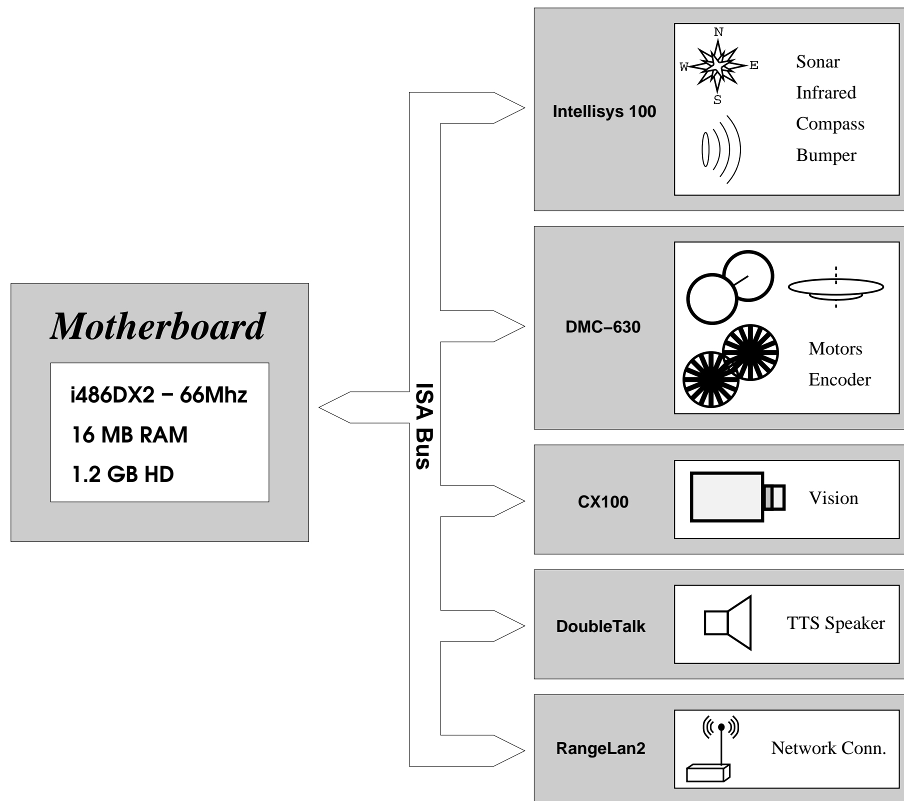


Figura 2.8: Schema a blocchi del sistema di elaborazione del Nomad.

bot: oltre alle già citate schede DMC-630 (per il controllo dei motori) e CX100 (frame grabber) è presente una scheda *Intellisys 100*, prodotta appositamente dalla Nomadic, che controlla contemporaneamente i sensori di contatto, di prossimità e la bussola magnetica. L'hardware della scheda comprende un microcontrollore *Motorola MC68HC11*, operante a 16 MHz, ed una memoria *Dual Ported Ram* che, utilizzata come buffer di interfaccia sul bus, realizza un meccanismo di comunicazione a *memoria condivisa* tra il processore centrale ed il microcontrollore della scheda. Quest'ultimo, occupandosi delle complesse fasi di lettura e controllo dei sensori, consente una notevole riduzione del carico di lavoro richiesto al PC.

Completano la dotazione hardware una scheda di sintesi vocale *DoubleTalk PC* [32], che realizza la conversione TTS (Text To Speech) avvalendosi di hardware dedicato, ed un modulo *Proxim RangeLan2* [33] che, assieme ad un *Ethernet Access Point*, fornisce un link radio per la connessione senza fili ad una rete locale con una

banda di trasmissione pari a circa 500 Kbps.

Il sistema di elaborazione ed i sensori posti a bordo del Nomad sono alimentati da due batterie alloggiate nella parte inferiore della torretta, che forniscono ognuna una energia pari a 204 Wh, più una terza batteria da 144 Wh posta alla base del robot, per un totale di 552 Wh. Le tensioni necessarie al funzionamento delle schede elettroniche vengono realizzate da un alimentatore *switching* integrato all'interno del robot, presentato in figura 2.9.

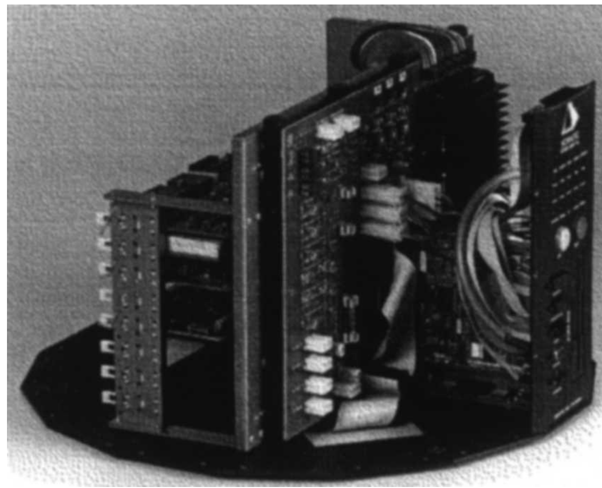


Figura 2.9: La scheda di alimentazione Nomadic.

2.1.4.2 Software

Le periferiche hardware presenti sul Nomad vengono rese disponibili per lo sviluppo di applicazioni robotiche mediante uno strato software di interfaccia, realizzato dalla Nomadic, che consente di imporre comandi agli attuatori e acquisire lo stato dei sensori senza conoscerne architettura e temporizzazioni nel dettaglio. Nelle prime versioni il PC del Nomad 200 ospitava un sistema operativo di tipo *MSDOS* ed un software ad hoc realizzato in forma di applicazione eseguibile. In seguito si è preferito equipaggiare il Nomad con il sistema operativo *Linux* che fornisce indubbiamente numerosi vantaggi come il supporto nativo per il multithreading e un set completo di primitive per la gestione delle connessioni di rete. Questo ha reso

necessario riprogettare parzialmente il programma di controllo *robotd* che ha acquisito il paradigma tipico dei *Demoni UNIX*: il server viene lanciato durante lo start up del sistema operativo per consentire ai client, che eseguono codice custom, di connettersi localmente o da macchina remota mediante un meccanismo di *IPC* basato su *UNIX Socket*. La Nomadic fornisce un set completo di API, utilizzabili da linguaggio C, che permettono di scrivere le proprie applicazioni ignorando di fatto il meccanismo di trasporto dei comandi.

Assieme al Nomad è disponibile il software di simulazione *NServer* che, sostituendosi a robotd, è in grado di interfacciare le applicazioni client con un ambiente virtuale che simula l'interazione tra il robot ed una serie di ostacoli collocati sulla scena mediante un ambiente grafico di controllo. Nonostante l'uso di un ambiente di simulazione possa svolgere un importante compito di supporto per la didattica, l'impiego intensivo di questi strumenti per verificare le funzionalità delle applicazioni realizzate è in genere sconsigliato: le semplificazioni che il simulatore introduce nella modellazione dell'ambiente tendono infatti a produrre errori, dovuti per lo più all'eccessiva predicibilità e ripetibilità delle misure, che portano ad ottenere risultati ottimistici nelle fasi di test che non si riscontrano nella realtà. Il simulatore della Nomadic, in particolare, fornisce un supporto che include soltanto la sensorialità di basso livello del robot ed utilizza modelli matematici molto semplificati per descrivere l'interazione con l'ambiente, risultando di fatto di limitiata utilità per l'esecuzione di applicazioni non banali.

2.2 Limiti dell'hardware

Nonostante il Nomad 200 costituisca una unità robotica completa, robusta e versatile, che ha consentito la sperimentazione di compiti robotici complessi presso il nostro Dipartimento [34, 35, 36, 37], la rapida evoluzione dell'hardware per l'informatica, che mette a disposizione strumenti sempre più evoluti a basso costo, ha fatto affiorare alcuni limiti importanti dell'architettura che rendono problematico l'utilizzo del Nomad 200 per realizzare attività di ricerca in alcuni recenti settori di interesse. Alcuni di questi problemi, giudicati insormontabili sino a pochi anni fa, diventano ora oggettivamente superabili se si riesce a trarre beneficio dall'evoluzione tecnologica.

In particolare, durante il progetto di un software di teleoperazione per il Nomad [38] sono emersi alcuni ostacoli alla realizzazione di un sistema sufficientemente reattivo ed affidabile, di cui riporto brevemente i principali motivi:

- Il PC contenuto nel Nomad fornisce la capacità computazionale sufficiente per l'esecuzione del software di controllo robotd ma consente di ospitare processi aggiuntivi di complessità molto ridotta: questo obbliga di fatto all'utilizzo di un calcolatore remoto per la realizzazione di applicazioni di dimensioni significative e diminuisce di conseguenza l'efficacia dei meccanismi di sicurezza, che si attivano quando viene riconosciuta una situazione di pericolo, dal momento che tutti i comandi di attuazione vengono inviati al robot mediante una connessione radio-ethernet;

- La connessione di rete fornita con il robot presenta spesso valori di latenza piuttosto elevati che si sommano alla necessità di aggregare le letture sensoriali per ridurre il traffico di informazioni, pessimizzando di fatto il tempo di reazione agli stimoli;

- All'interno di un ambiente chiuso e fortemente strutturato la connessione radio soffre visibilmente degli svantaggi legati alla modulazione ad alta frequenza (2.4 GHz), riducendo il più delle volte la zona di copertura a poche decine di metri: l'uso di una rete di punti di accesso opportunamente dislocati nei locali di maggior interesse potrebbe risolvere questo problema, ma il protocollo proprietario utilizzato per implementare la connessione radio impedisce l'interoperabilità dell'hardware di produttori differenti;

- La telecamera on board può essere utilizzata per realizzare algoritmi di visione artificiale (eseguiti preferibilmente sul PC locale) ma è in genere inadatta per fornire un output video di dimensioni e *frame rate* sufficienti per applicazioni teleoperate: il motivo è ancora da ricercarsi nella ridotta banda a disposizione della connessione di rete.

Esaminando le attività di ricerca che negli anni passati hanno fatto uso del robot presso il nostro Dipartimento è possibile individuare altri importanti limiti di carattere generale che tendono a ridurre sensibilmente l'operatività del robot a causa della ridotta capacità computazionale:

- La navigazione guidata dalle letture sensoriali che il robot rileva durante il cammino richiede un certo grado di sicurezza che consenta di contenere i rischi di collisione con gli ostacoli dell'ambiente; se il sistema di elaborazione non è in grado di elaborare in tempo reale le informazioni provenienti dai sensori è necessario ridurre la rapidità dei movimenti o disabilitare parte della sensorialità: complessivamente questo produce una diminuzione della reattività e l'introduzione implicita di un insieme di vincoli che l'ambiente deve rispettare affinchè il compito venga svolto in modo corretto[4];

- La pianificazione del moto e la localizzazione possono coinvolgere attività computazionali di una certa complessità che il robot non può svolgere durante la navigazione: questo impone di separare nettamente le operazioni di movimento dalle fasi di elaborazione più onerose, che saranno eseguite preferibilmente a robot fermo. I vincoli imposti tendono ad impedire l'utilizzo di algoritmi che producono correzioni progressive della traiettoria e rendono perciò il movimento del robot meno fluido ed in genere inadatto ad ambienti dinamici;

- L'utilizzo di sensori di alto livello, che producono generalmente un'informazione ricca ma sovrabbondante, richiede quasi sempre l'utilizzo di algoritmi di *post-processing* che, applicati a quantità di dati rilevanti, tendono a saturare le capacità computazionali del Nomad: ancora una volta emerge la necessità di separare nel tempo l'esecuzione dei processi percettivi dalle fasi di movimento e dalla navigazione. Ad esempio, l'uso della visione artificiale sul Nomad 200 si è spesso limitato all'acquisizione di pochi frame, a robot fermo, effettuata quando la sensorialità attiva durante la navigazione (sonar o infrarossi) mette in evidenza una situazione di possibile interesse, come un ostacolo dal profilo noto [36].

Un altro limite dell'architettura, che emerge assieme al crescente interesse del mondo informatico per i problemi di *standardizzazione* e *interoperabilità* delle componenti hardware e software, è legato all'utilizzo di protocolli proprietari, comune

[4] Ad esempio, se la navigazione è guidata esclusivamente dai sensori sonar frontali e laterali si esclude a priori ogni forma di controllo su eventuali ostacoli mobili che possono raggiungere il robot da dietro.

fino a pochi anni fa, nella realizzazione dei moduli di interfaccia: l'utilizzo della robotica al di fuori del contesto accademico, ed in particolare la recente crescita del settore della *domotica*, accresce l'importanta dell'interazione delle componenti eterogenee, realizzate spesso da produttori differenti, all'interno del sistema in esame. È facile individuare i limiti che il Nomad 200 mostra in questa direzione: si pensi ad esempio al metodo di comunicazione utilizzato da robotd per scambiare messaggi con i client o alla connessione radio fornita dalla scheda RangeLan.

Per ridurre i limiti riscontrati nell'architettura del robot e consentire così l'utilizzo del Nomad in un contesto di ricerca più ampio, si è deciso di avviarne un aggiornamento tramite la sostituzione per quanto possibile dell'hardware elettronico che vi è contenuto.

Capitolo 3

Aggiornamento del robot

3.1 Sostituzione dell'hardware

Da un esame dettagliato delle componenti che costituiscono il PC contenuto nel robot, emerge che il sistema è stato realizzato utilizzando per lo più schede di tipo standard disponibili per il mercato consumer. La scheda madre è una *Super Voyager VLB* realizzata dalla *American Megatrends* per la famiglia di processori Intel 486: si tratta di una scheda di dimensioni piuttosto elevate (33x22 cm) che soddisfa lo standard AT e che fornisce 8 slot ISA a 16 bit, due dei quali provvisti di modulo di estensione *VESA Local Bus*. La lista delle schede ISA inserite negli slot disponibili è riportata nella tabella 3.1.

Card	**Slot**
Multi I/O Adapter (VGA + IDE)	VLB 32 bit
Frame grabber CX100	ISA 8 bit
Scheda di rete wireless RangeLan	ISA 16 bit
Sintetizzatore DoubleTalk	ISA 8 bit
Controller Galil DMC-630	ISA 8 bit
Scheda Intellisys 100	ISA 8 bit

Tabella 3.1: Schede utilizzate dal PC del Nomad 200.

3.1.1 Sistema di elaborazione

Il passaggio ad un processore *Pentium* di ultima generazione implica necessariamente la sostituzione della scheda madre e con essa della maggior parte delle schede ISA utilizzate sul robot, che vengono rese inevitabilmente obsolete dal nuovo bus PCI. Fortunatamente molte schede madri disponibili sul mercato forniscono alcuni slot ISA per connettere dispositivi *legacy* di difficile sostituzione: questo consente di riutilizzare le schede *Galil DMC-630* e *Intellisys 100*, fortemente legate all'hardware del robot.

Per conciliare i vincoli di ingombro, le necessità di porte di I/O, e ridurre per quanto possibile il consumo di potenza, è stato scelto di utilizzare una motherboard *Soyo SY-7VEM* [39], illustrata in figura 3.1, che, disponendo di un ampio corredo di periferiche a bordo, consente di ridurre considerevolmente il numero di schede aggiuntive. Oltre al corredo standard di connettori, sostanzialmente equivalente a

Figura 3.1: La nuova scheda madre del Nomad 200.

quello disponibile in precedenza, la nuova scheda ATX fornisce due canali IDE

Ultra ATA/66, due canali USB, una scheda video *Trident Blade 3D* e un codec audio compatibile con lo standard *AC97*. Sono infine presenti tre slot di espansione *PCI* ed un solo slot *ISA*.

L'installazione attuale include infine un processore *Pentium III* con frequenza di clock pari a 1 GHz, un modulo di RAM *PC133* da 256 MB ed un HardDisk da 40 GB.

3.1.2 Hardware di interfacciamento

L'hardware sensoriale e di attuazione disponibile sul Nomad utilizza complessivamente due schede di interfaccia: la scheda Intellisys 100 e il controller Galil DMC-630. Entrambi i moduli sono espressamente realizzati e tarati per il robot e non possono essere facilmente eliminati dal sistema senza un impegnativo lavoro di reingegnerizzazione dell'hardware. In un primo tempo si è pensato di utilizzare una scheda madre che fornisse almeno due slot ISA per collegare le schede necessarie: purtroppo le schede ATX che soddisfano questo vincolo hanno dimensioni troppo elevate per essere alloggiate nello spazio disponibile all'interno del robot, ed in ogni caso l'ingombro decisamente elevato delle schede ISA da inserire limita fortemente le soluzioni possibili.

Esaminando le caratteristiche del bus ISA si può notare che i singoli slot sono collegati in parallelo tra loro, quindi possono essere duplicati a partire da un solo connettore: sfruttando questa possibilità abbiamo inserito le due schede in un *bus replicator*[1], che è stato collegato allo slot ISA della motherboard mediante un cavo flat a 64 poli (figura 3.2).

I due slot PCI della scheda madre sono rimasti inutilizzati: questo consente di sostituire agevolmente il frame grabber CX100 con un modello più recente o di inserire una scheda firewire per utilizzare le periferiche di acquisizione video digitale di ultima generazione [40].

[1]Queste schede venivano spesso utilizzate per variare di 90° l'inclinazione degli slot nei PC di tipo *slim*.

Figura 3.2: Scheda di controllo dei motori.

3.1.3 Connessione di rete

La connessione radio-lan disponibile sul robot tramite la scheda RangeLan2 soffre, oltre che di una ridotta banda di trasmissione rispetto agli standard attuali, del difetto di essere realizzata tramite un protocollo proprietario, impiegato per trasmettere i pacchetti, che impedisce l'utilizzo di hardware di produttori alternativi per le schede di rete e gli *access point*. I protocolli di comunicazione wireless hanno subito una recente standardizzazione, definita dall'IEEE con la sigla *802.11*, che consente l'interoperabilità di hardware eterogeneo. Lo standard 802.11, comunemente indicato con l'acronimo *WiFi*, ha riscontrato una rapida diffusione, e sta diventando lo standard di riferimento anche per molte applicazioni di domotica.

All'interno del Dipartimento di Ingegneria dell'Informazione è stata recentemente realizzata una copertura wireless che può essere sfruttata per garantire al Nomad una connessione veloce ed affidabile che ne consenta il movimento in tutti i locali disponibili. Per garantire al robot l'accesso alla rete è stato montato sulla sommità della torretta un modulo *802.11b USB Wireless Adapter* prodotto dalla *ActionTec* [41] (figura 3.3).

Il dispositivo wireless utilizzato fornisce una banda massima di trasmissione di 11 Mbps; in realtà, a causa dei limiti introdotti dal bus USB, dell'uso della cifratura

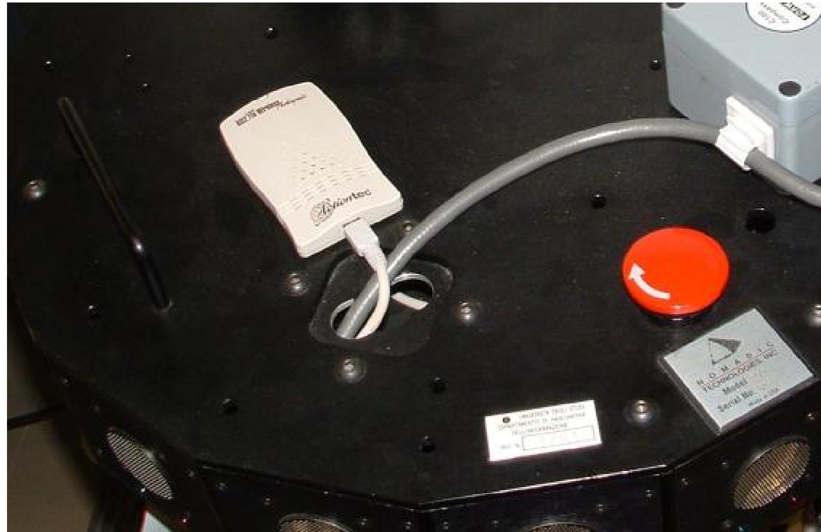


Figura 3.3: WiFi adapter collegato al bus USB.

WEP del canale e del traffico di rete, le velocità misurate non sono mai salite oltre i 6 Mbps. È da evidenziare, tuttavia, che trattandosi di tecnologia standard essa è destinata ad evolvere e migliorare nel rapporto costo/prestazioni, di cui anche il robot potrà beneficiare: ad esempio, durante la redazione di questa tesi, si sono resi disponibili sul mercato componenti conformi al nuovo standard *802.11g* che prevede una banda di 54 Mbps.

3.1.4 Sintesi vocale

La scheda di sintesi vocale DoubleTalk, compatibile con il bus ISA, realizzava la conversione *Text-To-Speech* facendo uso di un microprocessore dedicato *Intel 80C188EB* a 10 MHz. La potenza di calcolo fornita dal nuovo processore Pentium III, a differenza del precedente 486DX2, consente di eseguire direttamente gli algoritmi di sintesi senza necessità di hardware aggiuntivo; inoltre la nuova scheda madre fornisce direttamente un codec audio integrato; diventa quindi molto più semplice delegare ad un software dedicato la sintesi vocale.

Dopo aver esaminato alcuni progetti disponibili in rete, è stato deciso di utilizzare il *Festival Speech Synthesis System* [42], sviluppato presso l'Università di Edinburgo. Il programma, disponibile gratuitamente per il sistema operativo Linux,

fornisce una interfaccia interattiva che ne consente l'esecuzione da console e un set di librerie utilizzabili nei propri programmi per importarne le funzionalità. Il sistema è ampiamente estendibile; consente in particolare di personalizzare la lingua utilizzata mediante una serie di file contenenti le regole lessicali e le librerie di campioni vocali digitalizzati.

3.1.5 Alimentazione

La scheda di potenza progettata dalla Nomadic per alimentare il PC a bordo del robot non è sufficiente per fornire le tensioni necessarie alla nuova scheda madre: il motivo risiede nei cambiamenti avvenuti con il passaggio dallo standard AT al nuovo standard ATX. Oltre alle modifiche apportate ai connettori di alimentazione, che sono in sè facilmente superabili, le schede madri di tipo ATX richiedono all'alimentatore una tensione continua di 3.3 V, da cui viene assorbita una notevole potenza, che non era presente sul connettore precedente e che quindi diventa difficile realizzare mediante un qualche circuito adattatore. Per non rischiare di sovraccaricare il circuito di potenza già presente sul robot, grazie anche allo spazio rimasto libero dalla differente disposizione delle schede ISA, si è deciso di utilizzare un alimentatore DC-DC da 250 W [43] collegato direttamente alle batterie del Nomad. La disposizione dell'alimentatore all'interno del robot è illustrata in figura 3.4. Dalla misura del consumo di potenza del sistema di elaborazione è emerso un assorbimento di corrente dall'alimentazione a 12 V (fornita dalle batterie) che varia tra i 6.7 e i 7 A, consentendo al robot di operare per più di 5 ore consecutive. Contrariamente alle previsioni il consumo complessivo si è nettamente ridotto (presumibilmente grazie alla riduzione del numero di schede a bordo del robot), e non è stato necessario inserire batterie aggiuntive.

L'uso di un'architettura ATX per il sistema di elaborazione presenta un altro problema legato al fatto che l'alimentatore, contrariamente al comportamento dei modelli AT, non fornisce corrente alla scheda madre fino a che non riceve da essa un comando di accensione sul pin *Power Supply On*. Il pannello di accensione del Nomad (illustrato in figura 3.5) fornisce due pulsanti per l'accensione e lo spegnimento del robot internamente collegati ad un relè bistabile che interrompe e ripristina completamente l'alimentazione delle batterie quando vengono premuti (si

Figura 3.4: Alimentatore ATX all'interno del robot.

veda [28] per gli schemi elettrici del Nomad): di conseguenza la semplice pressione del tasto *On* non ha più l'effetto di accendere il PC. Un altro inconveniente legato all'alimentazione del robot, che si presentava anche prima dell'upgrade, si ha nel momento dello spegnimento: il tasto *Off* del pannello di figura 3.5 toglie immediatamente tensione al robot senza consentire nessuna fase di *shutdown* al PC; per un uso corretto del robot lo spegnimento dovrebbe quindi essere sempre preceduto da un comando di *halt* eseguito dal super user tramite console remota.

Per rendere più agevole l'accensione e lo spegnimento sono stati realizzati alcuni semplici circuiti elettrici (si veda l'appendice A per una descrizione degli schemi) che, cooperando con alcuni script che sfruttano l'interfaccia software *ACPI* del sistema operativo, consentono di accendere completamente il Nomad con una semplice pressione del tasto On o scatenare la fase di shutdown del PC quando viene premuto Off.

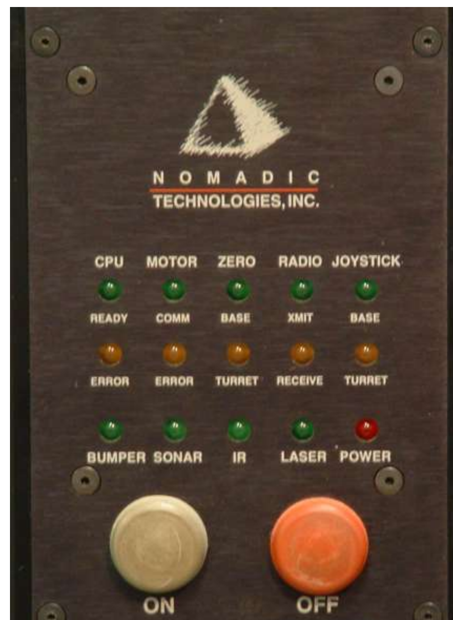


Figura 3.5: Pannello di controllo del Nomad.

3.2 Adeguamento del software

Nelle scelte effettuate durante l'aggiornamento dell'elettronica di bordo si è cercato di minimizzare le possibili differenze dell'interfaccia software verso l'hardware del robot, ciò nonostante si sono rese necessarie alcune modifiche alla configurazione del sistema operativo installato e al programma di controllo robotd, di cui fortunatamente la Nomadic ha reso disponibili i sorgenti.

Approfittando dello spazio ampliato dell'hard disk è stata installata una distribuzione recente del sistema operativo Linux (*SuSE 8.1*) ed un kernel dell'ultima generazione (versione 2.4.20). Il nuovo dispositivo wireless viene controllato da un kernel-driver che ne rende trasparente l'utilizzo, fornendo al sistema una connessione TCP/IP analoga a quella disponibile tramite la scheda RangeLan.

Per disporre del sistema di sintesi vocale è stato installato il software Festival assieme ai file necessari per la sintesi del parlato in lingua inglese: il programma, che interpreta i comandi provenienti dallo standard input, viene avviato in background durante lo startup del sistema impostando la redirezione dell'input verso la socket UNIX `/dev/festival`: in questo modo l'utilizzo delle funzionalità di TTS da programma richiede semplicemente l'invio di stringhe verso tale file.

Il programma di controllo del robot fornito dalla Nomadic è ancora in grado di comunicare con le schede Intellisys 100 e Galil DMC-630 che si collocano agli stessi indirizzi di I/O anche nella nuova architettura. Sono state comunque necessarie alcune modifiche al codice preesistente per consentire la lettura della bussola magnetica, che ora si trova collegata ad una differente porta seriale, e per integrare il supporto del nuovo software di sintesi vocale. A seguito delle modifiche apportate il programma robotd, derivato dalla versione 2.6.7 (l'ultima che supporta la scheda di controllo DMC-630), diventa pienamente compatibile con il nuovo hardware installato sul robot ed esporta tutte le funzionalità della versione originale.

3.3 Limiti del software *robotd*

L'architettura del Nomad 200, e con essa il software fornito dalla Nomadic, mostra le tendenze tipiche della robotica del periodo in cui è stato costruito; in particolare la ridotta capacità computazionale a bordo del robot ha orientato i progettisti di *robotd* verso soluzioni che privilegiano l'accesso remoto all'uso locale. Il nuovo robot al contrario, disponendo di un processore dell'ultima generazione, consente di svolgere algoritmi di notevole complessità senza la necessità di un calcolatore aggiuntivo.

Il software di controllo in realtà viene fornito anche in forma di libreria, che espone le stesse API del client remoto, che può essere linkata al codice aggiuntivo per realizzare un programma che viene eseguito localmente sul robot. Purtroppo però i tentativi effettuati di sfruttare questa possibilità per produrre programmi più efficienti e funzionali hanno avuto scarsi risultati, principalmente a causa dell'interfaccia poco amichevole della libreria, che spesso differisce dalle specifiche e non implementa tutte le funzioni disponibili remotamente.

Un altro limite che può ridurre sensibilmente la reattività delle applicazioni è la totale assenza di meccanismi di comunicazione di tipo *push*: le letture dei sensori avvengono infatti tramite una richiesta effettuata dal client sul server, introducendo un tempo di latenza significativo e scarsamente misurabile tra l'istante in cui il dato viene letto dall'hardware e quando diventa disponibile per il client. Questo intervallo può essere ridotto soltanto se si aumenta la frequenza con cui il dato

viene richiesto (quindi introducendo overhead sulle comunicazioni), e comunque non può mai essere annullato completamente.

Il software di controllo, che non utilizza nessuna funzionalità real-time del sistema operativo, non garantisce il funzionamento ottimale, con prestazioni che decadono progressivamente quando l'unità di elaborazione è sovraccarica.

Infine risulta difficile integrare nuovo hardware sensoriale nell'architettura preesistente, che tra l'altro non fornisce un'interfaccia omogenea neppure per l'hardware originale del robot (ad esempio la telecamera ed il pantilt non possono essere controllati se non localmente).

Capitolo 4

Progetto del framework

4.1 Obiettivi

La libreria *object-oriented* prodotta durante questo lavoro di tesi ha come scopo la costruzione di un sistema software che fornisca allo sviluppatore gli strumenti necessari per realizzare di un'applicazione di robotica mobile ad un livello sufficientemente elevato di astrazione, nascondendo i dettagli legati allo scheduling dei task ed alla realizzazione delle comunicazioni tra di essi, fornedo nel contempo i mezzi per garantirne il controllo ove necessario.

Nonostante questo progetto abbia avuto inizio con l'obiettivo di costruire un'architettura più moderna e funzionale per il Nomad 200, la realizzazione del framework si è svolta ad un più alto livello di astrazione, che ha permesso di mantenerne la completa indipendenza da ogni specifica architettura hardware e aumentarne quindi la generalità. L'uso del framework per realizzare applicazioni specifiche per il Nomad (come viene descritto nel capitolo successivo) è una prima applicazione, ed è auspicabile un suo utilizzo in futuro per altri sistemi robotici.

Focalizzando l'attenzione sulla robotica mobile sono state analizzare alcune architetture di publico dominio (descritte nel capitolo 1) con lo scopo di individuare le problematiche e le necessità di cui il progetto dovrà occuparsi. L'analisi dei *pattern di comunicazione* effettuata dai membri del progetto *OROCOS* si è rivelata particolarmente utile per identificare le possibili forme di comunicazione che l'architettura deve poter fornire ai propri componenti: una breve descrizione delle primitive

evidenziate in [12] è riportata nella tabella 4.1.

Pattern	**Descrizione**
Send	Trasferisce dati dal client al server senza la ricezione di una conferma dal server. Rappresenta una comunicazione *monodirezionale* utile per inviare comandi e settare configurazioni.
Query	Il client effettua una richiesta al server ed attende la risposta. Rappresenta una comunicazione *bidirezionale* tra un solo client ed un solo server. Può essere utile quando un'informazione viene utilizzata ad una frequenza molto bassa rispetto alla velocità con cui viene prodotta: è più ragionevole che il client la richieda mediante una query piuttosto che venga prodotta una quantià eccessiva di aggiornamenti non necessari.
Autoupdate	Uno o più client si registrano presso un server richiedendo un'informazione che viene inviata non appena nuovi dati sono disponibili. Rappresenta una comunicazione di tipo *push*. È possibile ridurre il traffico, se il client richiede l'informazione ad una frequenza minore rispetto a quella con cui viene prodotta, mediante una richiesta che prevede l'invio dei dati soltanto ad ogni ennesimo aggiornamento (*autoupdate timed*).
Event	Il server individua un evento avvenuto sullo stato del sistema ed informa in modo asincrono il client che ne assicura la gestione. Gli eventi sono utilizzati principalmente per notificare modifiche nello stato che sono rilevanti per coordinare i task in esecuzione.
Configuration	Supporta una relazione di tipo *Master-Slave* tra i moduli che consente l'attivazione selettiva delle loro attività. Il modulo slave, quando un'attività viene disattivata, deve impedire che il proprio stato interno possa diventare inconsistente, deve proteggere l'esecuzione delle sezioni critiche e gestire la terminazione delle comunicazioni pendenti con gli altri moduli.

Tabella 4.1: Pattern di comunicazione identificati in *OROCOS*.

L'evoluzione progressiva dei sistemi robotici ha segnato l'affermazione delle *architetture ibride*, che consentono la cooperazione di processi reattivi, dal ridotto carico computazionale ma dall'elevata frequenza di attivazione, e processi deliberativi, che richiedono tempi di elaborazione lunghi e difficilmente quantificabili. Il design dei singoli livelli dell'architettura enfatizza molto spesso l'importanza della

modularità del progetto, che favorisce la costruzione progressiva delle applicazioni ed il parziale riuso dei componenti.

Alla luce di queste considerazioni, dopo l'analisi dello stato dell'arte, si è deciso di progettare il framework in modo tale da consentire lo sviluppo di applicazioni per la robotica mobile secondo uno schema *behaviour-based*, limitando l'architettura al solo livello reattivo ma studiandone la struttura in modo da favorirne l'integrazione con i livelli superiori. Verranno messi a disposizione strumenti avanzati per lo scheduling real-time dei processi ed un insieme completo di primitive di comunicazione di alto livello. In particolare le interfacce per lo scambio di dati e comandi dovranno garantire la trasparenza rispetto al meccanismo che le implementa, per garantire la possibilità di estendere il supporto verso la comunicazione con dispositivi remoti e la teleoperazione.

Lo sviluppo del software coinvolto nelle applicazioni robotiche che si desidera progettare può avvenire a diversi livelli di astrazione. In primo luogo è necessario realizzare il codice che costituisce il framework vero e proprio, quindi occorrerà costruire i componenti necessari per il funzionamento del sistema (sfruttando l'interfaccia messa a disposizione dal framework), infine i singoli componenti, che possono rappresentare behaviour di uso generale o possono essere legati ad un particolare hardware robotico, devono essere scelti e riuniti per ottenere l'applicazione desiderata. Le funzionalità ed i punti di estensione messi a disposizione dal framework verranno discussi nel paragrafo successivo, mentre il capitolo 5 si occuperà di illustrare nel dettaglio la realizzazione della libreria.

4.2 Strumenti disponibili

4.2.1 Moduli

L'elemento fondamentale per il progetto guidato dal framework è il *Modulo*, inteso come componente attivo ed autonomo che opera all'interno dell'architettura: il sistema in esecuzione è composto da un insieme di moduli che svolgono attività indipendenti e cooperano tra loro scambianDo informazioni e comandi (figura 4.1).

I moduli concreti, che possono essere di uso generale o legati ad un robot specifico, svolgono compiti elementari che sarà possibile riunire per realizzare l'applica-

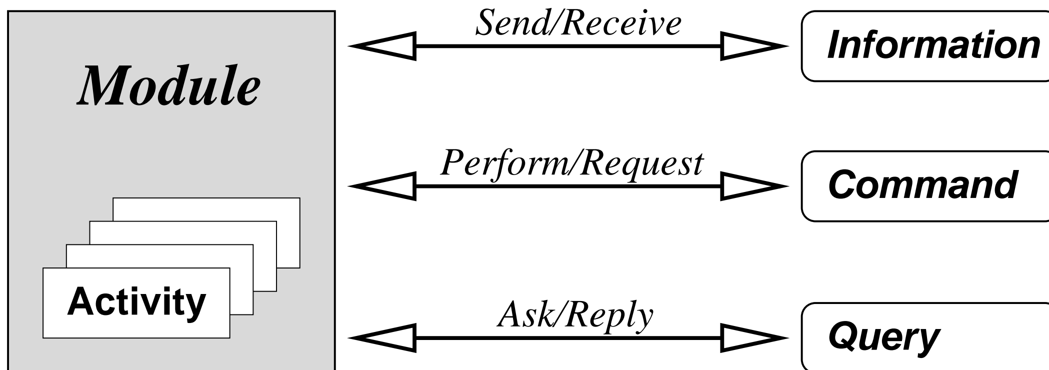


Figura 4.1: Strumenti per lo sviluppo: il *Modulo* ed i suoi componenti.

zione complessiva: un singolo modulo può rappresentare la versione concreta di un *behaviour*, oppure può rendere disponibile all'interno del sistema alcune interfacce verso l'hardware sensomotorio del robot.

Il modulo può essere visto come un componente che possiede un insieme di algoritmi, eseguiti sulla medesima struttura dati, che rispondono alle differenti sollecitazioni provenienti dall'esterno per portare avanti un obiettivo comune (si noti l'analogia con il concetto di *object* della OOP). La libreria consente di realizzare i moduli concreti dell'applicazione come classi C++ derivate dalla classe astratta `Module`, che rappresenta il principale *hot-spot* del framework. Ogni modulo possiede un certo numero di *attività* che, eseguendo il codice specificato nel corpo della classe, possono operare a basso livello sull'hardware del robot e dialogare con gli altri moduli del sistema mediante le primitive di comunicazione fornite dal framework. La figura 4.2 illustra il meccanismo con un semplice esempio: il modulo *Sonar* si occupa della lettura dei sensori e provvede ad inviarne periodici aggiornamenti al modulo *AvoidCollision* (un behaviour del robot) che controlla lo stato dei sensori per identificare le condizioni di pericolo. Quando viene rilevato un ostacolo eccessivamente vicino al robot *AvoidCollision* invia un comando di *STOP* al modulo *Motor*, che ne interrompe il moto.

Il costruttore della classe `Module` richiede un solo parametro di tipo stringa che consente di associare un nome ad ogni oggetto delle classi derivate; questo permette al framework di indicizzare i moduli del sistema per gestirne l'attivazione: tramite le funzioni membro `turnOn()` e `turnOff()` (che richiedono un parametro di

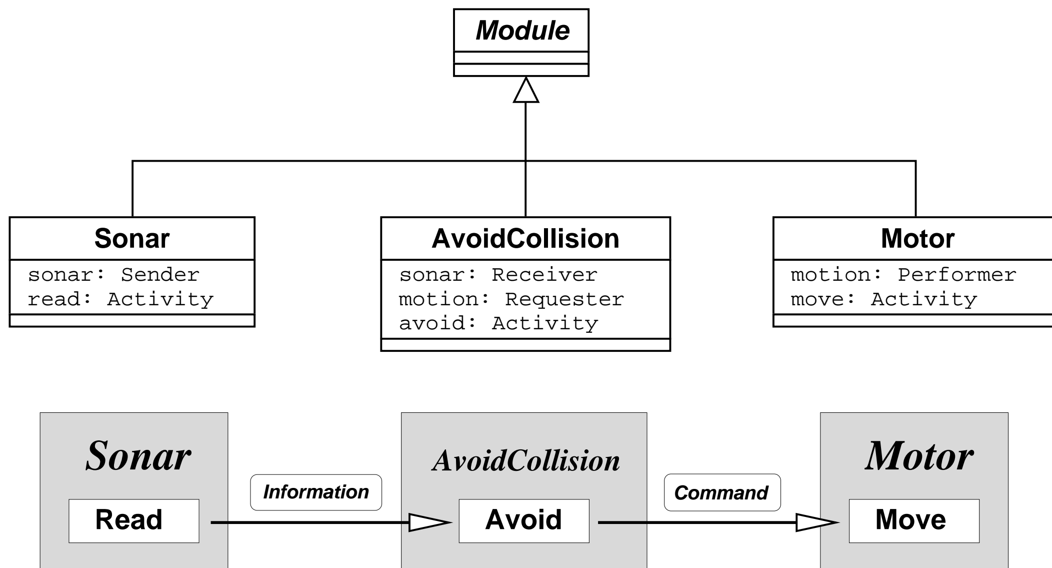


Figura 4.2: Una semplice applicazione di esempio.

tipo stringa) ogni modulo può attivare o disattivare l'esecuzione dei processi relativi al modulo specificato. Questa interfaccia garantisce le funzionalità espresse dal pattern *Configuration* indicato nella tabella 4.1, anche se in questo caso non viene costruita una gerarchia che consenta solo ai moduli di categoria superiore di manopolare le attivazioni. Quando un modulo viene sospeso, il framework posticipa la disattivazione fino a che le attività pendenti vengono portate a termine. Questa caratteristica semplifica la realizzazione dei singoli moduli, che non devono occuparsi di risolvere i problemi legati all'interruzione forzata della attività in esecuzione.

4.2.2 Attività

I moduli concreti arricchiscono la classe `Module` con un insieme di *data-member* che ne rappresentano lo *stato* ed alcuni *metodi* che elaborano su di essi. Dati e metodi sono componenti *privati* della classe `Module` e non possono essere utilizzati direttamente all'esterno di essa: questa proprietà deriva dalla considerazione che il modulo non si comporta come un oggetto *passivo*, che esporta funzionalità che altri oggetti possono richiedere eseguendo i suoi metodi, bensì possiede una propria trama di esecuzione che realizza autonomamente le *attività* proprie del modulo, che

possono essere risvegliate periodicamente o scatenate da un evento esterno. Di tutte le funzioni membro definite all'interno di un modulo, alcune vengono utilizzate solo localmente, mentre altre rappresentano il corpo delle attività che il framework provvederà a porre in esecuzione al momento opportuno: la classe template `Activity` consente di specificare le attività del modulo e fornisce i metodi necessari al loro utilizzo da parte degli altri componenti della libreria.

Ogni attività associata ad un modulo viene posta in esecuzione tramite i meccanismi interni del framework: l'esecuzione può essere periodica, nel qual caso il metodo che vi è associato viene lanciato ad intervalli regolari, oppure può dipendere dall'interazione con gli altri moduli presenti nel sistema (ad esempio può essere attivata quando viene inviata un'informazione). Ad ogni attività, tramite il costruttore[1] è assegnato un parametro che, espresso in secondi, ne indica il periodo *nominale* di funzionamento: esso rappresenta l'esatta cadenza temporale dell'esecuzione periodica e l'intervallo minimo che deve intercorrere tra due attivazioni aperiodiche consecutive. In entrambe le situazioni, operando in ambito real-time, il valore indicato avrà anche il significato di *deadline* per l'attività, cioè il tempo massimo entro cui l'esecuzione del metodo deve giungere a termine per essere giudicata corretta.

L'esecuzione di una attività equivale formalmente all'esecuzione di una funzione membro del modulo corrispondente: per questo motivo i parametri template della classe
`Activity` consentono di specificare il tipo di dato che sarà passato come parametro all'atto dell'esecuzione ed il tipo del valore di ritorno. Per semplicità il numero di parametri di una `Activity` non può essere superiore ad uno; resta comunque possibile specificare un numero maggiore di parametri facendo uso di una struttura che li racchiude.

Il codice che realizza concretamente le operazioni legate all'attività viene specificato fornendo al costruttore dell'oggetto `Activity` il puntatore alla funzione membro corrispondente; quest'ultima dovrà essere dichiarata nel corpo della classe derivata da `Module` compatibilmente con i parametri template della `Activity`

[1] Alcuni oggetti del framework che vengono dichiarati normalmente all'interno di una classe di tipo `Module` (tra cui `Activity`), richiedono per esigenze implementative il puntatore all'oggetto che li racchiude (il **this** di Module); questo valore viene di norma passato come primo parametro al costruttore della classe contenuta, ma per semplicità non verrà più descritto nella trattazione.

a cui viene associata. Il listato 4.1 chiarisce con un esempio il modo corretto di dichiarare un'attività all'interno di un modulo.

```
class ConcreteModule : public Module
  {
  private:
    void   act1Body(int param);
    float  act2Body();

  public:
    Activity<void,int>    act1;
    Activity<float,void>  act2;

    ConcreteModule() : Module( "MyName" ),
      act1 ( *this, &ConcreteModule::act1Body , 0.1 ),
      act2 ( *this, &ConcreteModule::act2Body , 0.02 )
      {}
  };
```

Listato 4.1: Dichiarazione di attività all'interno di un modulo.

L'attivazione aperiodica di un oggetto `Activity` viene controllata dal sistema di comunicazione del framework, che sarà descritto nel paragrafo successivo; al contrario la schedulazione periodica di una attività può essere invocata direttamente mediante il metodo `startPeriodicRun()` della classe `Activity`, o interrotta con `stopPeriodicRun()`. Lo scheduling periodico, eseguito con il periodo specificato nel costruttore, è limitato alle attività che non presentano parametri e valore di ritorno, dal momento che in questo caso non sono particolarmente utili.

È possibile pensare alla classe `Module` come ad una particolare versione del pattern *Active Object* illustrato in [44]: mentre l'esecuzione del metodo di un oggetto passivo sfrutta il *processo* dell'oggetto chiamante, che resta bloccato in attesa che l'operazione venga portata a termine, un oggetto attivo possiede un processo proprio che provvede all'esecuzione dei metodi quando ne viene fatta richiesta. In pratica la chiamata a metodo ha l'effetto di inserire la richiesta in una coda che il processo dell'active object provvede a servire in ordine *FIFO*; il chiamante può decidere se

procedere con l'esecuzione di altro codice o attendere il risultato dell'esecuzione del metodo richiesto.

Esistono molte varianti del pattern Active Object, ma è interessante in modo particolare quella che emerge dalla discussione sulla concorrenza nei sistemi object-oriented fatta da B.Meyer in [45]. L'autore propone una soluzione semplice, quanto rigorosa, al problema della programmazione concorrente introducendo la keyword "**separate**" nel linguaggio *Eiffel* da lui realizzato: se un oggetto di tipo "T" è accessibile in un punto del codice mediante un reference di tipo "**separate** T", allora l'esecuzione dei metodi applicati a tale oggetto avviene su di un processore diverso da quello del chiamante (chiamata *asincrona*), altrimenti l'esecuzione del metodo avviene in modo convenzionale, sfruttando lo stesso processo del chiamante (chiamata *sincrona*). È importante notare che nè il codice che chiama il metodo nè il codice dell'oggetto separato dipendono dal meccanismo utilizzato per gestire la comunicazione, sia esso realizzato mediante primitive di sincronizzazione in un ambiente a memoria condivisa, scambio di messaggi tra processi UNIX o chiamate a funzione su calcolatori remoti (RPC).

Un altro punto chiave della filosofia di Meyer è l'assenza totale di sincronizzazioni esplicite nel codice che viene realizzato: l'autore del *Design by Contract* sostiene che per garantire la validità delle proprietà formali in ambito concorrente, descritte mediante *precondizioni*, *postcondizioni* ed *invarianti* che caratterizzano i metodi, è necessario che in ogni momento in un dato oggetto (attivo) si trovi in esecuzione al più uno solo dei suoi metodi, rendendo di fatto superflo l'utilizzo di sezioni critiche all'interno dei metodi per modificare lo stato dell'oggetto. Nella sostanza quello che viene affermato è che l'esecuzione contemporanea di più processi sui metodi di uno stesso oggetto può portarne lo stato in una condizione inconsistente rispetto a quello che è stato specificato nelle loro precondizioni e postcondizioni, rendendo impossibile una verifica formale della correttezza degli algoritmi. Conviene quindi consentire l'accesso ai dati dell'oggetto ad un solo ad un processo alla volta, facendo diventare i singoli metodi delle unità atomiche di esecuzione.

La presenza di processi eseguiti con obblighi di tempo reale aggiunge complessità ulteriore al sistema in esame. Infatti i vincoli sui tempi di esecuzione introducono un forte legame tra gli oggetti dell'applicazione e rendono necessario l'utilizzo di un'entità software complessa (lo *scheduler*) che si occupa di assegnare la CPU

in modo da consentire il rispetto di tutte le deadline. Ne deriva che l'esecuzione del codice associato alle `Activity` non viene mai effettuata direttamente. La richiesta (periodica o aperiodica) viene inoltrata allo scheduler integrato nel framework, che ne gestisce l'attivazione coordinandola con tutte le altre attività che sono in esecuzione.

Alla luce dai concetti proposti da Meyer si è deciso di imporre che per ogni modulo attivo nel sistema una sola attività alla volta possa modificarne lo stato, accodando le richieste che arrivano contemporaneamente. In realtà è possibile effettuare una distinzione sulle attività in esecuzione che consente di ottimizzare leggermente questo vincolo: il C++, mediante il modificatore "**const**", consente di specificare nella dichiarazione di un metodo se questo potrà o meno modificare le variabili membro degli oggetti su cui viene eseguito; associando una attività ad un metodo *const* si ottiene che l'attività stessa potrà essere eseguita in più istanze contemporanee sullo stesso oggetto senza pregiudicarne lo stato. L'uso di attività costanti, che svologono solo una funzione di lettura dei data-member dell'oggetto, è particolarmente utile nella realizzazione del meccanismo di comunicazione *Query* (illustrato nel paragrafo successivo), quindi l'interfaccia della classe `Activity` è stata opportunamente modificata per consentire la specifica di questa proprietà. Il listato 4.2 illustra come sia possibile definire attività costanti e non costanti ponendo rispettivamente *CONST* e *NONCONST* il terzo parametro template di `Activity`; per semplicità, quando viene definita una attività non costante, il parametro *NONCONST* può essere omesso.

L'uso di attività costanti su di un modulo può allora trarre vantaggio da una politica di attivazione di tipo *lettori/scrittori*: in uno stesso modulo saranno poste in esecuzione contemporanea al più un'attività non costante (scrittore) o un numero variabile di attività costanti (lettori).

4.2.3 Comunicazioni

I moduli realizzati per l'applicazione vivono contemporaneamente nello stesso spazio di indirizzamento, pertanto è possibile realizzare i meccanismi di cooperazione tra essi mediante un insieme condiviso di strutture dati in cui le attività dei singoli moduli leggono e scrivono in modo concorrente. La struttura dati comune utilizzata

```
class ConcreteModule : public Module
  {
  private:
    int value;

    void setValue(int v) { value = v; }
    int getValue() const { return value; }

  public:
    Activity<void, int> set;
    Activity<int, void, CONST> get;

    ConcreteModule(): Module( "MyName" ), value(),
      set( *this, &ConcreteModule::setValue , 0.02 ),
      get( *this, &ConcreteModule::getValue , 0.02 )
      {}
  };
```

Listato 4.2: Attività *CONST* e *NONCONST*.

per scambiare le informazioni prende il nome di *blackboard* in molte architetture robotiche [46], e presenta alcuni vantaggi legati principalmente al basso overhead che viene introdotto dalle comunicazioni ed alla facilità con cui è possibile realizzare un modello del mondo reale con questo sistema. Se cresce la complessità dell'architettura l'approccio diventa però ben presto inadeguato; vediamone le cause:

- L'uso intensivo di memoria condivisa e dei meccanismi di sincronizzazione per gestirne l'utilizzo rendono più difficile la realizzazione del codice, che diventa sensibile alle problematiche tipiche della programmazione concorrente: rischio di deadlock, modifiche inconsisteti dei dati, errori non deterministici e non ripetibili, ecc.

- Se aumenta il numero di processi che lavorano sulla blackboard, che deve essere modificata all'interno di sezioni critiche, diventano via via maggiori i tempi in cui i thread restano sospesi sulle primitive di sincronizzazione; separare le strutture dati condivise per garantire un accesso più efficiente è

spesso una soluzione solo parziale, dato che in genere è difficile deciderne a priori la granularità, ed a posteriori diventa difficile modificare il codice già scritto per garantirne la correttezza.

- L'architettura non consente di integrare direttamente componenti remoti: se alcuni dati sensoriali vengono prodotti da un altro calcolatore (ad esempio l'elaborazione di una telecamera eseguita da un PC dedicato) occorre inserire localmente un *proxy* che provvede a interrogare il sistema remoto e riportare nella blackboard i valori restituiti. In generale, se l'elaborazione è distribuita su più macchine, ognuna di esse deve avere una rappresentazione locale della blackboard, ed occorre attivare una serie di processi che provvedono a mantenerne coerente lo stato sulle differenti macchine.

- L'architettura che viene progettata impone in modo implicito una forte separazione tra processi e strutture dati, che tende a indirizzare lo sviluppo verso sistemi *orientati agli algoritmi*, in cui i processi in esecuzione svolgono compiti piuttosto complessi e monolitici, e sono fortemente legati ai dati comuni che manipolano. La modifica della struttura della blackboard richiede in genere di modificare quasi tutto il codice dei processi che ne fanno uso, per cui i componenti del sistema diventano fortemente legati all'applicazione e scarsamente riusabili.

Alla luce di queste considerazioni, ricordando anche quanto detto nel paragrafo 4.2.2 a proposito delle sincronizzazioni, si è deciso di integrare nel framework un insieme di primitive di alto livello che consentono ai moduli di comunicare senza l'utilizzo esplicito di dati condivisi. La realizzazione concreta delle primitive utilizza solo internamente memoria condivisa e sincronizzazioni, così da esporre un'interfaccia uniforme che consente di ignorare (almeno in prima approssimazione) la "posizione reciproca" dei due moduli comunicanti, e ne rende molto può facile l'utilizzo nei sistemi che integrano componenti remoti. In questo modo il framework potrà evolvere in modo sufficientemente agevole per consentire la comunicazione di moduli situati su calcolatori differenti e permettere la realizzazione di architetture distribuite che possono svolgere anche compiti in teleoperazione.

Information

Il primo pattern di comunicazione che viene descritto prende il nome di *Information*, e consente ai moduli di inviare e ricevere dati di tipo generale che serviranno per lo più a rappresentare misure sensoriali effettuate dall'hardware del robot. Mediante questa primitiva di comunicazione i singoli moduli sono in grado di spedire informazioni nel sistema, che ne descrivono un cambiamento dello stato; ad esempio, in linguaggio naturale, il significato di una comunicazione di tipo *Information* può corrispondere a:

> "la *velocità* vale *10* $\mathrm{cm/s}$"

Le classi template `Sender` e `Receiver`, illustrate nel diagramma UML di figura 4.3, possono essere utilizzate all'interno dei moduli concreti come *data-member* (nel disegno sono riportati a titolo di esempio i moduli già visti in figura 4.2): ogni modulo cioè può possedere uno o più oggetti di questo tipo, che utilizza come strumento con cui *registrarsi* per l'invio o la ricezione di un dato, il cui tipo viene specificato mediante il parametro template `Type`[2]. Il parametro `name`, di tipo stringa, presente nel costruttore di entrambe le classi, consente al framework di assegnare un nome simbolico[3] ai differenti dati che i moduli intendono scambiarsi: due moduli M_1 e M_2 che si registrano rispettivamente come *sender* e *receiver* del medesimo dato vengono collegati virtualmente dal framework, che provvede ad adeguare la copia locale di M_2 quando M_1 ne produce un aggiornamento ed esegue il metodo `update()` dell'oggetto sender. Il modulo M_2 può accedere al dato ricevuto mediante il metodo `get()` dell'oggetto receiver che ne ritorna una copia locale a cui il modulo può liberamente accedere in lettura.

Affinchè il modulo ricevente possa individuare situazioni anomale di funzionamento in cui il dato non viene più inviato (ad esempio perché l'attività del sender è stata sospesa) si è deciso di assumere che la chiamata al metodo `get()` *consumi* il dato ricevuto. Se tra due letture successive non viene effettuato nessun aggiornamento, il metodo `get()`, quando viene lanciato per la seconda volta, solleva una eccezione di tipo `EmptySlot` che il chiamante dovrà manipolare. Un esempio di

[2] Corrisponde ai "$\mathrm{cm/s}$" dell'esempio in linguaggio naturale.
[3] Corrisponde alla parola "velocità" dell'esempio in linguaggio naturale.

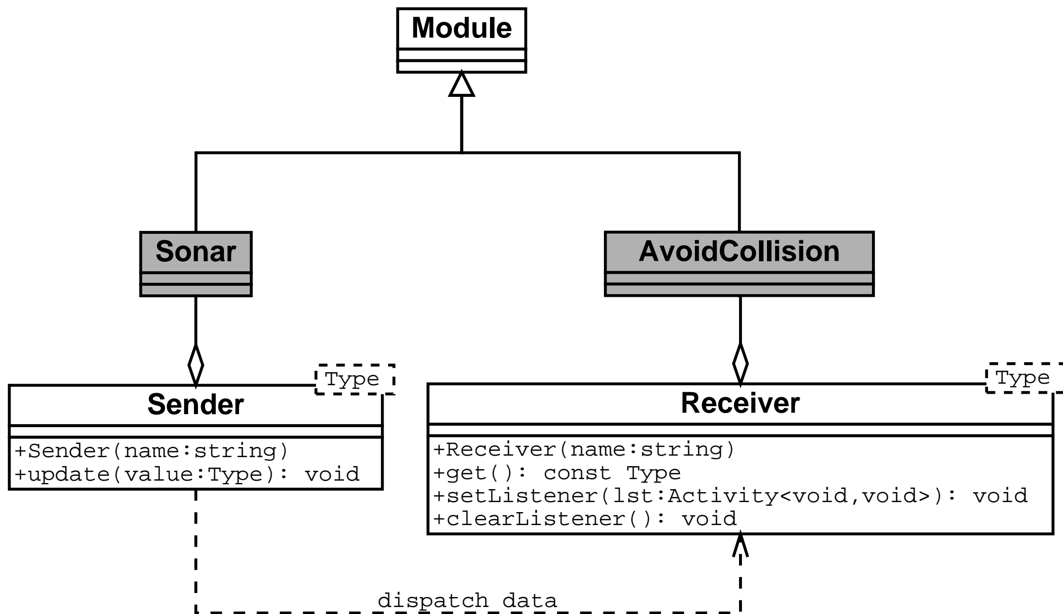


Figura 4.3: *Information* pattern: class diagram.

utilizzo delle classi `Sender` e `Receiver` all'interno di un modulo è illustrato nei listati 4.3 e 4.4.

Per garantire una maggiore efficienza nell'ambiente a memoria condivisa in cui si trovato i moduli, il framework provvede a ridurre il numero di copie dei dati di grandi dimensioni che vengono scambiati mediante il pattern information. Per fare questo il metodo `get()`, anzichè ritornare una copia dell'oggetto, ritorna uno *smart pointer* che punta ad un oggetto condiviso tra tutti i moduli receiver; questi ultimi possono utilizzarlo per l'elaborazione ma non possono modificarlo, dato che il puntatore è di tipo *const*. Mediante un meccanismo di *reference-count* sincronizzato, lo smart pointer provvede inoltre a liberare la memoria occupata dall'oggetto quando più nessun modulo ne possiede il riferimento.

È possibile pensare al sistema come ad una versione *implicita* della blackboard, in cui ogni modulo può leggere e scrivere in una porzione di memoria condivisa mediante i meccanismi forniti dalle classi `Sender` e `Receiver`; in questo caso però la comunicazione è completamente controllata dal framework, e la sua realizzazione fisica può essere modificata a piacere per supportare schemi differenti,

```
class SendModule : public Module
  {
  private:
    void measureVel()
      {
      float vel;
      // Low level read ...
      velocity.update(vel);
      }

  public:
    Sender<float> velocity;

    SendModule() : Module( "VelSender" ),
      velocity( *this, "velocity" )
      {}
  };
```

Listato 4.3: Utilizzo di un oggetto *Sender*.

senza intaccare il codice dei moduli concreti.

Il meccanismo illustrato consente una comunicazione di tipo *broadcast* tra i moduli: un solo modulo si può infatti registrare come *sender* di un dato, ma molti moduli possono esserne *receiver*. Il framework provvederà automaticamente ad aggiornare lo stato di ognuno di essi. Durante la creazione degli oggetti `Sender` e `Receiver` viene eseguita una verifica di correttezza che segnala la presenza di dati condivisi aventi lo stesso nome ma tipo differente e che impedisce a più di un modulo alla volta di registrarsi come *sender* dello stesso dato In entrambi i casi i costruttori delle classi sollevano un'eccezione nel codice del modulo chiamante di tipo `NameConflict`.

È importante notare che la creazione di un oggetto `Receiver` eseguita quando nessun modulo ha creato un oggetto `Sender` con il medesimo nome, è perfettamente lecita. In caso contrario infatti diventa necessario introdurre un ordine di inizializzazione dei moduli che, oltre ad essere scomodo da rispettare, rende impossibile l'avvio dell'applicazione se sono presenti dipendenze circolari che coinvolgono moduli e dati condivisi. Questa modalità di funzionamento però può essere causa di

```
class RecvModule : public Module
  {
  private:
    void readVel()
      {
      try
        {
        float vel = velocity.get();
        // Do something...
        }
      catch ( EmptySlot & e )
        {
        // Do something else ...
        }
      }

  public:
    Receiver<float> velocity;

    RecvModule() : Module( "VelReceiver" ),
       velocity ( *this , " velocity " )
      {}
  };
```

Listato 4.4: Utilizzo di un oggetto *Receiver*.

malfunzionamenti del sistema poco visibili, dovuti ad errori fatti durante la definizione dei moduli. Se ad esempio viene scritto il nome del dato in modo differente nella definizione del `Sender` e del `Receiver` (es.: “`sonar`” e “`Sonar`”), quest’ultimo non riceve nessun aggiornamento e non segnala nessun errore. Per mettere in evidenza queste situazioni è previsto un servizio di *log* in cui il framework riporta le azioni che vengono effettuare e segnala le situazioni che possibilimente indicano un’anomalia (ad esempio un receiver e nessun sender o un sender e nessun receiver).

Nell’interfaccia della classe `Receiver` (figura 4.3) restano ancora da descrivere i metodi `setListener()` e `clearListener()`, che consentono di *agganciare* l’esecuzione di una attività alla ricezione di un nuovo dato. Quando il dato

richiesto dal modulo ricevente viene aggiornato, se è stata specificata una attività *listener* mediante il metodo corrispondente, il framework esegue lo scheduling aperiodico del metodo che vi è associato. Il sistema provvede inoltre a registrare l'istante dell'ultima attivazione per garantire che la successiva non avvenga in un istante più ravvicinato del periodo nominale associato all'attività. Se il dato viene aggiornato con frequenza troppo elevata, alcune attivazioni vengono tralasciate.

I componenti descritti, grazie alla capacità di risvegliare le attività del modulo, permettono di realizzare tramite il framework i pattern di comunicazione *Event* e *Autoupdate* (anche nella versione *timed*) descritti nella tabella 4.1.

L'ultima caratteristica delle primitive di tipo *Information* che viene illustrata riguarda la specializzazione esplicita della classi template `Sender` e `Receiver` per i dati *multipli*. Molto spesso alcuni apparati sensoriali montati a bordo dei robot sono formati da un insieme di sensori uguali tra loro che vengono letti in successione dal sistema di elaborazione: è il caso ad esempio dei sensori sonar ed infrarossi del Nomad 200 (si veda il paragrafo 2.1.3). Per utilizzare agevolmente questo tipo di informazione all'interno delle comunicazioni tra i moduli è stata definita la seguente struttura template che consente di definire un dato *multiplo*, corrispondente come significato ad una lista di N elementi di tipo T:

```
template < class T, size_t N >
struct multiple
  {
  typedef T value_type;
  const static size_t size = N;
  };
```

Infine è stata scritta una specializzazione esplicita delle classi `Sender` e `Receiver` per il tipo `multiple<T,N>` (dove `T` e `N` sono ancora una volta parametri template); queste classi presentano un'interfaccia leggermente modificata (si veda il diagramma UML di figura 4.4) che consente ai moduli di operare su *vettori* di dati di tipo `T` e lunghezza `N`, i cui elementi possono essere aggiornati singolarmente dal modulo sender, risvegliando attività dei moduli receiver che possono essere agganciate alle singole posizioni del vettore.

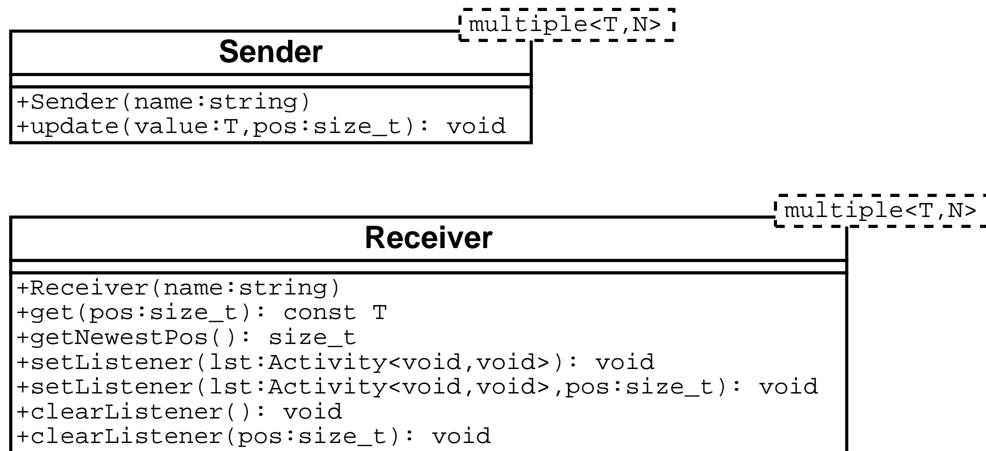


Figura 4.4: *Information* pattern: specializzazione esplicita per dati multipli.

Command

Le primitive di tipo *Command* consentono di realizzare comunicazioni tra i moduli simili come semantica al pattern *Send* indicato nella tabella 4.1. Il significato di un messaggio di questo tipo può essere chiarito con un esempio in linguaggio naturale:

> "imposta una *velocità* pari a *20* $\mathrm{cm/s}$"

Il meccanismo di comunicazione è duale rispetto a quello fornito da *Information*, infatti consente a molti moduli *requester* di inviare comandi contemporaneamente allo stesso modulo *performer*.

Le classi coinvolte nel pattern Command, illustrate nel diagramma UML di figura 4.5, ricordano lo schema già adottato per la comunicazione Information: il parametro `name`, indicato nei costruttori di `Performer` e `Requester`, consente al framework di etichettare i tipi di comandi e *collegare* virtualmente i moduli che li inviano e li ricevono, in modo del tutto analogo a quanto visto per `Sender` e `Receiver`.

L'esecuzione di un comando corrisponde nella programmazione imperativa al meccanismo di chiamata a funzione, pertanto è in generale importante avere la possibilità di specificare un parametro quando il comando viene richiesto (nell'esempio riportato sopra è tassativo specificare il valore della velocità): per questo motivo tutte le classi coinvolte forniscono il parametro template `Param` che consente

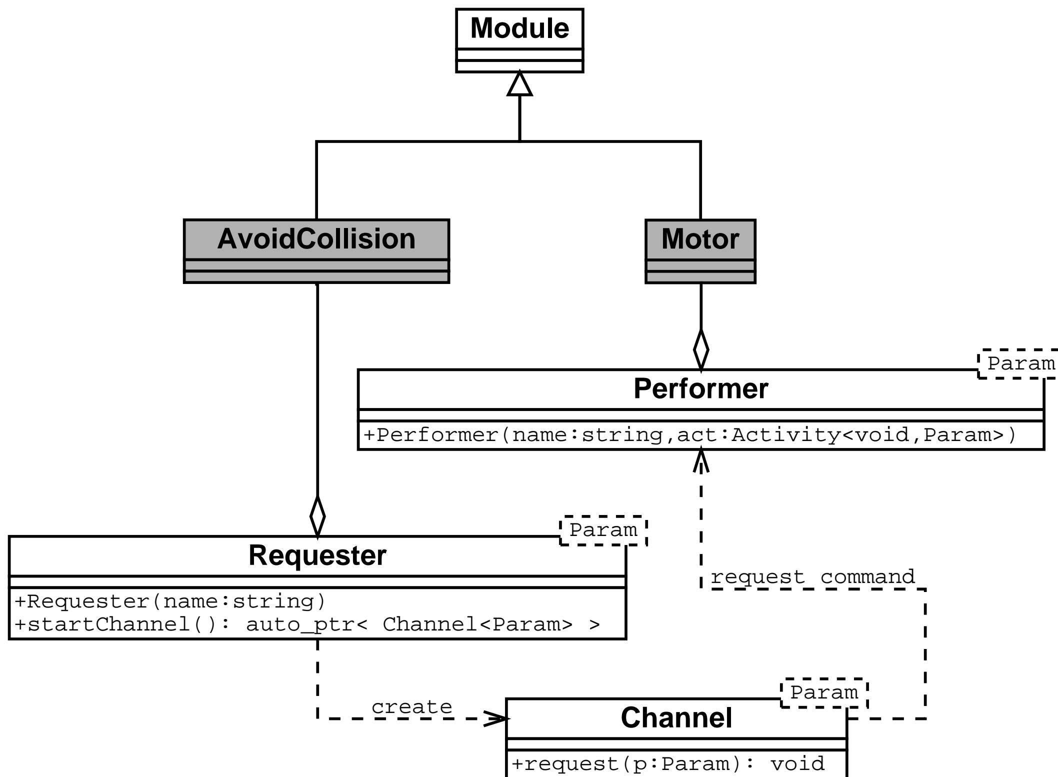


Figura 4.5: *Command* pattern: class diagram.

di specificarne il tipo. Il framework provvederà a trasportare il parametro effettivo dal modulo che fa la richiesta al modulo che, specificando tra i propri membri un oggetto di classe `Performer`, intende portarlo a termine. Assieme al nome, nel costruttore di `Performer` viene indicata l'attività che deve essere schedulata per eseguire il comando: anche quest'ultima deve accettare un parametro di tipo `Param`. Un semplice esempio di utilizzo di `Performer` è riportato nel listato 4.5.

I moduli che intendono richiedere l'esecuzione del comando dovranno dichiarare un oggetto di tipo `Requester` avente `Param` e `name` compatibili con quelli dichiarati nel modulo *performer* corrispondente. La classe `Requester` non fornisce però direttamente l'interfaccia per inviare comandi, bensì consente di aprire un *canale* attraverso cui farne richiesta. Il metodo `startChannel()` restituisce infatti uno smart pointer ad un oggetto di classe `Channel` tramite cui è possibile inviare un comando mediante il metodo `request()`. Il canale viene chiuso quando

```
class Motor : public Module
  {
  private:
    void setVelocity ( float value )
      {
      // Low level code ...
      }

  public:
     Activity<void, float> setVel;
    Performer<float> vel;

    Motor () : Module( "Motor" ),
      setVel ( *this, &Motor::setVelocity , 0.01 ),
      vel ( *this , " velocity " , setVel )
      {}
  };
```

Listato 4.5: Utilizzo di un oggetto *Performer*.

l'oggetto viene distrutto (listato 4.6).

La ragione per l'introduzione del *canale* deriva dal fatto che il concetto di comando, nella visione del framework, implica la sua *persistenza*: il modulo che richiede un comando desidera cioè (in modo implicito) che l'effetto duri per un certo tempo. Il problema nasce quando moduli differenti inviano comandi potenzialmente discordanti allo stesso performer; in questa situazione occorre un meccanismo di arbitrazione che decida quale comando deve essere eseguito. Se ad esempio il modulo *AvoidCollision*, utilizzato negli esempi precedenti, richiede di fermare il robot imponendo una velocità nulla, è importante che i comandi di velocità inviati dagli altri moduli vengano ignorati. In caso contrario il robot continuerebbe ad avanzare, alternando accelerazioni e brusche interruzioni di movimento dovute all'intervento di *AvoidCollision*. Al contrario, aprendo un canale per inviare il comando di stop, *AvoidCollision* può bloccare il movimento del robot finchè non giudica sicuro il suo movimento: solo allora il canale verrà chiuso.

La scelta di eseguire o meno i comandi che giungono ad un oggetto `Performer` quando più canali sono aperti contemporaneamente viene realizzata aggiungendo

```
class AvoidCollision : public Module
  {
  private:
    void lockRobot()
      {
      velChannel = vel.startChannel();
      velChannel -> request (0.0);
      }

    void unlockRobot()
      {
      velChannel.reset();
      }

  public:
    Requester<float> vel;
    auto_ptr< Channel<float> > velChannel;

    AvoidCollision () : Module( "AvoidCollision" ),
      vel( *this, "velocity" )
      {}
  };
```

Listato 4.6: Utilizzo di un oggetto *Requester*.

un ulteriore parametro template alla classe: il tipo `Arbiter`. La classe concreta che viene passata come parametro template svolge una funzione di *Policy* [47] per la classe `Performer`, poichè consente di modificarne il comportamento, in modo controllato, a tempo di compilazione. Per un corretto funzionamento del sistema anche le classi `Requester` e `Channel` richiedono un parametro template di tipo `Arbiter`.

Il framework supporta due diverse classi *Arbiter*, che realizzano due importanti strategie:

- *Priority*. Questa strategia di soluzione delle contese, associata alla classe `PriorityArbiter`, prevede di dotare ogni oggetto requester di una priorità. Essa permette al Performer di rendere attivo in ogni momento soltanto il canale aperto che ha priorità più alta; l'esecuzione di `request()` su tut-

ti gli altri canali avrà l'effetto di generare un'eccezione di tipo `Locked` nel codice del chiamante. Questo metodo è particolarmente utile per realizzare architetture reattive di tipo *Subsumption* [3].

- *Merge*. La classe `MergeArbiter` è stata introdotta principalmente per consentire la realizzazione di applicazioni che possano fare uso delle tecniche di navigazione *potential field* descritte da Arkin in letteratura [48]. In sostanza la strategia consiste nel fondere i vari comandi che provengono dalle differenti sorgenti, producendo ad ogni istante un valore che ne rappresenta la somma. Tutti i canali aperti forniscono un insieme di comandi differenti che vengono sommati per costruire il parametro passato all'attività che il performer ha registrato.

Query

Le primitive di tipo *Query*, che realizzano le funzionalità del pattern omonimo indicato nella tabella 4.1, servono per supportare un meccanismo di richiesta esplicita di informazioni tra i moduli. Ancora una volta il colloquio tra i moduli può essere chiarito con un semplice esempio:

D: "Quanto vale la *velocità*?"
R: " *42* cm/s."

Rispetto agli altri meccanismi illustrati, il pattern *Query* introduce una comunicazione bidirezionale tra due moduli, inviando una richiesta e ponendosi in attesa del risultato. Rispetto al pattern *Information*, in cui i moduli inviano dati in modo autonomo, in questo caso, la domanda è fondamentale affinchè venga prodotta una risposta significativa che sarà inoltrata al modulo interessato.

Il meccanismo di comunicazione è in questo caso piuttosto semplice. La query si comporta infatti da *wrapper* attorno all'attività che risponde alle richieste, garantendo l'invio e la ricezione dei dati attraverso i due moduli coinvolti.

Il modulo che intende rispondere alle query di un certo tipo deve dichiarare un oggetto `Activity` adatto allo scopo, quindi deve inizializzare un oggetto `Replier` a cui vengono passati come parametri il nome della query e l'attività. La

classe Replier, illustrata in figura 4.6, possiede due parametri template che consentono di specificare un parametro per l'invio della richiesta (Param) ed il tipo del risultato (Val). La activity richiesta dal costruttore di Replier è dichiarata con

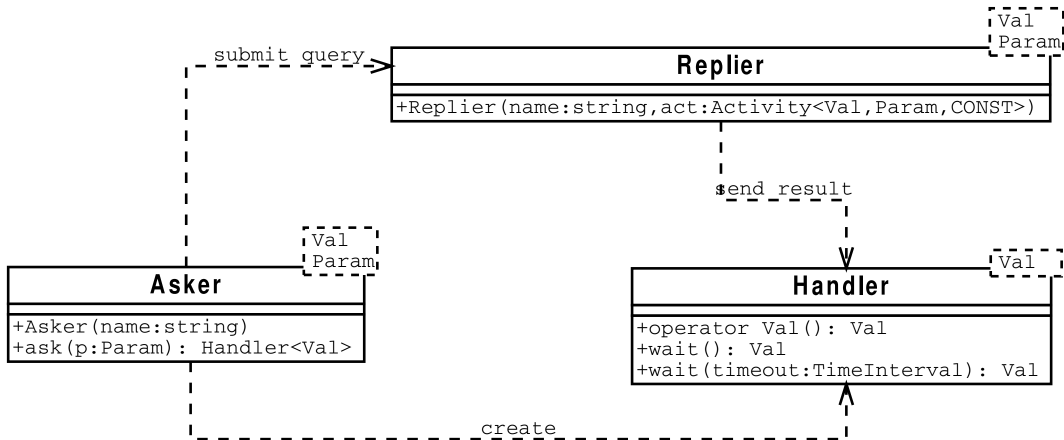


Figura 4.6: *Query* pattern: class diagram.

gli stessi parametri template Param e Val. L'uso corretto del pattern Query consiste nel di ricavare il risultato richiesto esplorando lo stato del modulo *replier*, senza però modificarlo. Per questo motivo l'activity associata alla query è di tipo *const*: questo vincolo permette una maggiore efficienza del codice, dato che le activity costanti possono avere accesso concorrente ai dati del modulo.

Il modulo che desidera sottomettere delle richieste tramite il pattern *Query* deve dichiarare un oggetto di tipo Asker che il framework provvederà a collegare con il Replier corrispondente mediante il nome indicato nel costruttore; la query può essere richiesta eseguendo il metodo ask(), che ritorna un oggetto di tipo Handler. L'interfaccia di Handler, mediante il metodo wait() e l'operatore di cast verso il tipo Val, consente di usare questo meccanismo per realizzare la query sia come chiamata sincrona con la sintassi convenzionale del C++ per la chiamata a funzione, sia come chiamata asincrona, in cui il chiamante continua l'esecuzione dopo aver inoltrato la richiesta e si sospende in attesa della risposta solo quando ne ha bisogno immediato. Il listato 4.7 riporta un esempio di utilizzo della classe Asker; come è possibile intuire il pattern *Query* verrà utilizzato per comunicazioni prevalentemente di alto livello.

```
class MapUser : public Module
  {
  private:
    void doSomething()
      {
      Handler<float> distR = distance.ask( "Lab.Robotica" );
      Handler<float> distW = distance.ask( "Lab.Workstation" );
       //...
      int minDist = min( distR.wait (), distW.wait () );
      }

  public:
    Asker<float, string > distance;

    MapUser() : Module( "MapUser" ),
      distance ( *this , "roomDistance" )
      {}
  };
```

```
class MapServer : public Module
  {
  private:
    float computeBody( string room ) const
      {
      float value;
      // computation ...
      return value;
      }

  public:
    Replier<float, string > distance;
    Activity<float, string ,CONST> computeDistance;

    MapServer () : Module( "MapServer" ),
      computeDistance( *this, &MapServer::computeBody, 10.0 ),
      distance ( *this , "roomDistance", computeDistance )
      {}
  };
```

Listato 4.7: Utilizzo di oggetti *Asker* e *Replier*.

Capitolo 5

Realizzazione del framework

5.1 Scheduling Real Time

Il generale si definisce *sistema real-time* un sistema la cui correttezza non dipende soltanto dal risultato logico della computazione, ma anche dall'istante temporale in cui il risultato viene prodotto; devono cioè essere soddisfatti un certo insieme di vincoli temporali (*deadline*) che caratterizzano le attività del sistema. In generale parleremo di *hard real-time* quando il fallimento delle deadline è causa di severe inconsistenze, che possono generare situazioni rischiose se il sistema controlla un apparato di grande importanza, e di *soft real-time* quando il rispetto dei vincoli temporali è descrivibile in termini probabilistici, e mancare alcuni di essi non degrada sensibilmente l'applicazione.

La definizione che è stata introdotta copre in realtà una gamma molto vasta di sistemi software, che vanno dai dispositivi di controllo di impianti industriali ai programmi che visualizzano flussi multimediali audio/video. Spesso un task real-time viene considerato semplicemente come un processo che "gira" ad una priorità molto elevata sul calcolatore; in effetti lo scheduler di molti sistemi operativi *general purpose* (che decide quale processo porre in esecuzione) fornisce strumenti per attivare task a priorità più elevata degli stessi processi del kernel, che permettono di ridurre i tempi di latenza dei task che operano in tempo reale. In realtà realizzare un sistema in tempo reale richiede uno sforzo maggiore della semplice assegnazione delle priorità, infatti chi lo sviluppa deve in genere far collaborare numerosi task

eseguiti in concorrenza, e verificarne il corretto funzionamento in modo formale.

Le caratteristiche del sistema operativo (*preemtive scheduler*, *dispatch latency*, *preemtible kernel*, *timer resolution*, ecc.) pongono un limite alla "qualità" del supporto per il real-time che viene fornito, per cui la classificazione hard/soft real-time viene estesa anche ad essi. Tra i sistemi che consentono di realizzare applicazioni *soft real-time* ricordiamo Linux e Solaris SunOS, mentre per disporre di garanzie *hard real-time* sono necessari kernel specifici, come quelli di VxWorks, LynxOS e QNX.

Molte applicazioni eseguite in tempo reale, per garantire un funzionamento corretto, fanno un uso diretto del meccanismo di priorità fornito dal sistema operativo, assegnando priorità via via più elevate ai processi più importanti del sistema che si sta realizzando. Al crescere del numero di processi diventa però sempre più complesso costruire manualmente un'assegnamento di priorità che sia tale da garantire l'eseguibilità dell'applicazione nel pieno rispetto delle deadline. Per questo motivo negli ultimi decenni si sono evoluti alcuni approcci formali al problema dello scheduling real-time, che hanno lo scopo di determinare il modo ottimo di porre in esecuzione le attività richieste e garantire così l'eseguibilità dell'interno sistema nel rispetto dei vincoli. La *schedulabilità* dell'applicazione, mediante alcuni strumenti di analisi, può essere dichiarata anche a priori se è possibile conoscere la deadline, la periodicità ed il valore massimo del tempo di esecuzione (*WCET*) di ogni task che l'applicazione richiede.

La realizzazione concreta di questi concetti consiste nella costruzione di algoritmi di scheduling evoluti che decidono istante per istante quale task porre in esecuzione sulla CPU in base ai loro vincoli di tempo reale, quali periodo e deadline. Lo scheduler che viene realizzato agisce quindi da wrapper su quello del sistema operativo, assegnando i task ad alcuni processi di sistema (ad esempio thread o LWP) e variandone le priorità. Le tecniche di scheduling più utilizzate sono: *Rate Monotonic Assignment (RMA)* ed *Earliest Deadline First (EDF)*:

- Lo scheduler di tipo *RMA* [49], dato un insieme di task periodici, assegna priorità crescenti ai task da eseguire con frequenza maggiore. Alcuni teoremi consentono di stabilire a priori se l'insieme di task è schedulabile secondo questo algoritmo, dato il periodo, la deadline e il WCET di ognuno di essi. Il *Rate Monotonic* è un assegnamento *statico* delle priorità, valido se i task

vengono attivati contemporaneamente in fase di start-up del sistema; nonostante questo è possibile realizzare politiche di *acceptance* che consentono di aggiungere nuovi task ad un sistema real-time in esecuzione, ripetendo ogni volta i test di schedulabilità e riassegnando le priorità ai processi. RMA nasce per porre in esecuzione un insieme di task periodici, con la possibilità di stabilire a priori il rispetto di tutte le deadline, esistono però alcune estensioni (*Sporadic Server*, *Deferred Server*, *Cyclic Pool*) che permettono di utilizzare task aperiodici all'interno del sistema.

- L'approccio EDF consiste nello schedulare in ogni istante il task che possiede la deadline più vicina, assegnando priorità decrescente ai task con deadline più lontana. Lo scheduling di tipo EDF richiede un assegnamento dinamico delle priorità e non opera distinzioni tra task periodici ed aperiodici, dato che l'esecuzione di ognuno di essi dipende soltanto dalla deadline relativa all'attivazione corrente. L'algoritmo è particolarmente adatto per realizzare scheduler dinamici che consentano di attivare i task durante l'esecuzione dell'applicazione real-time, valutando di volta in volta la schedulabilità della nuova attività che viene sottomessa.

Per schedulare dinamicamente il codice associato alle *Activity*, il sistema utilizza il framework *TODS (Timed Object for Distributed Systems)*, descritto in [50], che fornisce un'interfaccia orientata agli oggetti per supportare la programmazione in tempo reale all'interno di un'applicazione C++ per il sistema operativo Linux[1]. TODS, sviluppato presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma, consente di schedulare task *periodici* e *one-shot* assegnando ad ognuno di essi un thread di sistema, prelevato da un *thread-pool*, che viene sottoposto allo scheduler *FIFO* del sistema operativo Linux. La libreria permette di realizzare un algoritmo di scheduling real-time variando dinamicamente la priorità dei thread posti in esecuzione. La politica di scheduling che viene fornita di default con TODS e di tipo EDF.

Un obiettivo fondamentale del framework per la robotica che si sta realizzando è il mantenimento di un alto grado di indipendenza dall'architettura fisica su cui

[1] L'uso di Linux viene imposto dalla necessità di impiegare il framework con il Nomad 200, il cui software originale, che sarà riutilizzato parzialmente nella nuova architettura (si veda il capitolo 6), è disponibile solo per questo sistema operativo.

verrà utilizzato, per cui un legame stretto con il sistema operativo sottostante non è accettabile. TODS, in questo senso, consente un buon grado di estendibilità, essendo sviluppato a partire da API *POSIX*, che sono supportate dalla quasi totalità dei sistemi operativi UNIX-like, e consentendo la modifica del codice sorgente, garantita dalla licenza *GPL*. Il framework che ne fa uso mantiene comunque un buon grado di incapsulazione delle componenti legate allo scheduling, che potranno essere sostituite, qualora questo divenga necessario, operando modifiche solo in alcuni punti del codice complessivo.

Il sistema operativo Linux non fornisce a livello di kernel un set soddisfacente di primitive per la realizzazione di applicazioni real-time; in particolare mancano strumenti per la gestione delle regioni critiche che permettano di risolvere i problemi di *priority inversion*. Questo fenomeno avviene quando un task τ_a viene bloccato tentando di acquisire l'accesso esclusivo ad una risorsa già posseduta dal task τ_b a minore priorità; se a questo punto un task τ_c, avente priorità maggiore di τ_b ma minore di τ_a, diventa pronto per l'esecuzione, lo sheduler del sistema operativo lo pone immediatamente in esecuzione, interrompendo τ_b e impedendo di conseguenza il proseguimento del task τ_a, che pure ha la priorità più alta. Esistono alcuni metodi che consentono di risolvere questa categoria di problemi, tra cui il più semplice è il *Priority Inheritance Protocol*:

> La priorità del task che acquisisce una risorsa condivisa, per tutta la durata della sezione critica, sarà pari al massimo tra le priorità dei task che vengono bloccati su di essa e la priorità propria del processo.

TODS fornisce un insieme di strumenti di sincronizzazione (mutex, semafori, monitor) che realizzano il protocollo di *priority inheritance* a livello di libreria, garantendo il corretto funzionamento dello scheduling EDF anche in presenza di task che utilizzano risorse condivise.

Il framework messo a disposizione da TODS, sfruttando efficacemente la potenza espressiva del linguaggio C++, permette di realizzare oggetti dotati di attività eseguite in tempo reale mediante la derivazione dalla classe `RTObject`, come viene mostrato nell'esempio del listato 5.1. Le attività real-time dell'oggetto, rappresentate dalle istanze della classe template `RTMethod`, possono essere eseguite simulando la consueta sistassi di chiamata del C++. Nell'esempio seguente il corpo

```
class myRTobj : public RTObject
{
protected:
   // the body of the RT method
   int body( float )
   {
      ...   // the code that performs task
   }

public:
  RTMethod< myRTobj, int, float > method;

   // ctor
  myRTobj( shared_ptr< RequestManager > rm ) :
    RTObject( rm ),
    method( this , &( myRTobj::body ) )
  {}
};
```

Listato 5.1: Dichiarazione di un oggetto real-time in TODS.

del metodo `method` viene eseguito su di un thread separato, la cui priorità sarà determinata dallo scheduler in base alla deadline che viene specificata nel punto di chiamata:

```
ResultType r = rtObject .method( param, deadline );
```

La libreria consente di effettuare chiamate sincrone, in cui il thread chiamante si sospende in attesa del valore di ritorno, o asincrone, ritornando un *handle* mediante il quale sarà possibile valutare tale valore in un secondo momento. Viene fornita la possibilità di eseguire task periodici ed aperiodici, ed è possibile richiedere l'esecuzione di un metodo a bassa priorità per segnalare una *deadline miss* avvenuta durante l'esecuzione di un task.

Ad ogni sottomissione di un nuovo task real-time la richiesta viene inoltrata ad un modulo di *acceptance*, che decide se il task è schedulabile in base alle condizioni di carico attuali. In caso di rifiuto, il modulo solleva un'eccezione che viene propagata fino al codice che ha sottomesso il task. L'accepter, per decidere la schedulabilità dei task, sfrutta il calcolo dei WCET, che viene realizzato automaticamente

dagli oggetti `RTMethod` durante l'esecuzione.

All'interno del framework che si vuole realizzare, TODS è stato utilizzato per consentire lo scheduling real-time dei task rappresentati dagli oggetti `Activity`. In effetti tale classe deriva da `RTObject`, e definisce al proprio interno un metodo privato `activityBody` ed un oggetto `RTMethod` che ne funge da wrapper. Le richieste di scheduling periodico o aperiodico che vengono indirizzate all'activity hanno quindi l'effetto di schedulare, tramite TODS, un task real-time che esegue il corpo del metodo `activityBody`; a sua volta, quest'ultimo provvederà a lanciare il metodo che è stato associato all'attività tramite il suo costruttore. Per garantire la mutua esclusione nell'accesso ai dati del modulo è stato utilizzato un oggetto monitor, fornito da TODS, che realizza una politica di tipo lettori/scrittori: esso viene acquisito e rilasciato da `activityBody` rispettivamente prima e dopo l'esecuzione del metodo associato all'attività; l'acquisizione avviene in modalità *reader* per le activity di tipo *const*, ed in modalità *writer* per le activity *non-const*.

All'interno della classe `Module` sono stati definiti i metodi virtuali:

```
void DeadlineMissHandler( ActivityHandler& act );
void LoadExceededHandler(ActivityHandler& act);
```

Essi vengono eseguiti quando un'attività del modulo subisce una deadline miss e quando il carico del sistema è troppo elevato per schedulare una nuova attività. Il comportamento predefinito di entrambi consiste nella produzione di una stringa di errore sullo standard error, ma ridefinendo il metodo nei moduli concreti è possibile agire sull'attività che ha provocato l'errore. Il parametro `act` indica infatti l'oggetto `Activity` che ha registrato il malfunzionamento e ne fornisce la stessa interfaccia, limitatamente ai metodi che consentono di modificarne la periodicità. È importante notare che l'esecuzione di questi metodi avviene senza l'acquisizione del monitor associato al modulo, quindi nel codice che li realizza le modifiche ai dati membro del modulo avvengono in modo concorrente.

5.2 Comunicazioni

La natura dinamica del sistema che si intende realizzare, in cui il numero di componenti deve poter variare liberamente durante l'esecuzione, richiede schemi di comunicazione che consentano di gestire l'avvicendamento di moduli differenti per la fornitura di un servizio (*sender*, *performer*, *replier*), o la sua non disponibilità temporanea. Le possibili evoluzioni del framework, descritte nel dettaglio nel paragrafo successivo, includono la possibilità di utilizzare comunicazioni remote, e renderanno perciò ancora più importanti queste considerazioni.

Le scelte definitive, per le quali sono stati ampiamente sfruttati i pattern di design e di implementazione illustrati in [25, 44] e in [47], propongono l'utilizzo di una classe accessoria per ogni protocollo di comunicazione, che svolge il compito di *mediatore* (*Broker*) tra l'oggetto che deve inviare il dato e l'oggetto che desidera riceverlo (figura 5.1). Incapsulare i dettagli della comunicazione all'interno di que-

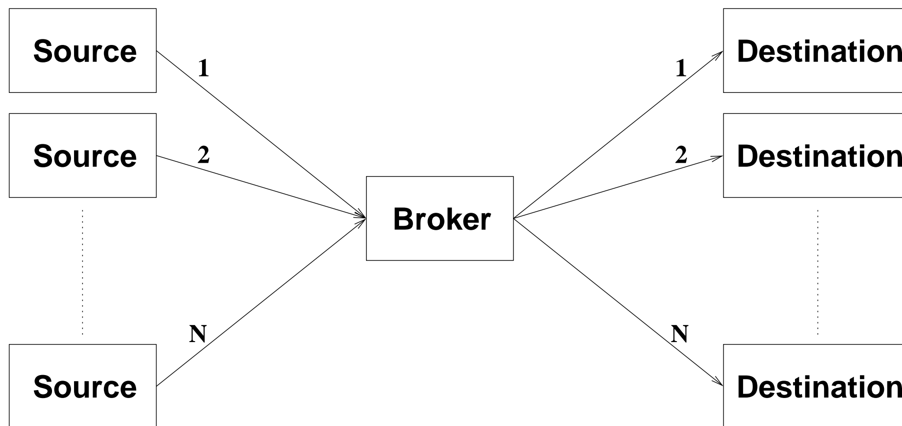


Figura 5.1: Schema generale di funzionamento delle comunicazioni.

sto oggetto, che rappresenta in un certo senso il *canale*, permette di disaccoppiare le classi sorgente e destinazione dei dati, sia come codice che come legame a tempo di esecuzione; l'oggetto *Broker* potrà infatti decidere istante per istante a chi deve essere inviata l'informazione in base ad un meccanismo di registrazione che le classi di destinazione eseguono su di esso.

Ogni canale di comunicazione, che collega tra loro due o più oggetti sorgente e destinazione, viene rappresentato dal nome simbolico che identifica il dato scambiato (la "velocità" degli esempi del capitolo precedente). L'oggetto broker sarà

unico per il canale, e gli oggetti che si trovano ai due estremi ne ricavano il riferimento a partire dal nome simbolico associato dalla comunicazione. Per rendere utilizzabile il canale quando almeno un oggetto ne ha fatto richiesta, i broker dei vari pattern di comunicazione vengono gestiti con un metodo di *reference count*: vengono creati quando il primo oggetto (sorgente o destinazione) richiede il canale e vengono distrutti quando l'ultimo di essi smette di utilizzarlo.

I singoli oggetti di tipo broker sono indicizzati in una tabella globale (*repository*) mediante il nome del canale che rappresentano; per fare questo viene utilizzata la classe `Repository` che consente di memorizzare dati eterogenei in un contenitore associativo, utilizzando stringhe alfanumeriche per indicizzare gli elementi. L'accesso ai dati che vi sono contenuti avviene mediante i metodi `create()` e `get()`, che funzionano in modo corretto anche durante l'accesso concorrente di più thread.

Il framework prevede tre repository separati per i pattern *Information*, *Command* e *Query*, i cui broker vengono realizzati mediante classi differenti; ciò nonostante è ancora indispensabile che tali contenitori possano includere oggetti eterogenei, infatti le istanze concrete delle classi di tipo broker, anche all'interno del medesimo protocollo, devono gestire la comunicazione di dati di tipo differente l'uno dall'altro, perciò, per sfruttare i vantaggi del type-checking statico del compilatore, saranno classi template. Per lo stesso motivo, anche *create* e *get* sono in realtà *template method* della classe `Repository`, ed il tipo del dato che viene richiesto tramite esse deve essere specificato esplicitamente quando vengono lanciati. Supponendo di voler richiedere al Repository un dato di tipo `DataType`, identificato dalla stringa `DataName`, i metodi descritti dovranno essere utilizzati come segue:

```
Repository_ptr<DataType,owner> ptr = create<DataType>( DataName );
Repository_ptr<DataType,user> ptr = get<DataType>( DataName );
```

Il valore di ritorno di entrambi i metodi è uno *smart pointer*, che si comporta come un puntatore all'oggetto di tipo `DataType`, ma gestisce la *lifetime* dell'oggetto a cui è collegato, garantendone la distruzione quando l'ultimo smart pointer che lo referenzia viene distrutto. Il secondo parametro template della classe `RepositoryPtr` consente di definire un reference di tipo *owner* o *user*: del primo, per ogni oggetto contenuto nel repository, può esistere una sola copia alla volta,

mentre del secondo può essere creato contemporaneamente un numero qualsiasi di istanze. Il reference di tipo *owner* rappresenta un accesso privilegiato all'oggetto, e verrà utilizzato dalle classi di tipo *sender*, *performer* e *replier*, il cui numero di istanze per ogni comunicazione realizzata non deve mai superare l'unità. Al contrario, le classi di tipo *receiver*, *requester* e *asker* utilizzeranno reference di tipo *user* per comunicare con i broker a cui sono associate.

Il metodo *create* alloca un nuovo oggetto nel Repository e ne ritorna un reference di tipo *owner*, se l'oggetto esiste già, ed è di possesso di un altro smart pointer di tipo *owner*, viene sollevata un'eccezione di tipo `NameConflict`. Il metodo *get* esplora in modo analogo il contenitore alla ricerca dell'oggetto con il nome richiesto e provvede a fornirne un reference di tipo *user* quando questo esiste. In realtà anche questo metodo può creare un oggetto quando non esiste; in questo caso il nuovo elemento viene dichiarato *not-owned* all'interno del repository ed una successiva chiamata al metodo *create* che ne faccia richiesta ritornerà l'oggetto creato in precedenza. Il motivo di questo comportamento deriva dalle considerazioni discusse nel paragrafo 4.2.3 sulla volontà di evitare l'introduzione di un ordine di inizializzazione per i moduli dell'applicazione.

Qualora venga richiesto un oggetto già presente nel repository, ma avente tipo differente da `DataType`, entrambi i metodi *create* e *get* sollevano un'eccezione di tipo `NameConfict`.

Nonstante le classi descritte fino ad ora possano essere utilizzate per realizzare tutti i meccanismi di comunicazione introdotti nel capitolo 4, fino ad ora il framework mette a disposizione le funzionalità del solo pattern *Information*, dato che *Command* e *Query* non sono ancora stati portati a termine.

Il pattern *Information* realizza la comunicazione tra i moduli che dichiarano al proprio interno due oggetti di classe `Sender` e `Receiver`, aventi lo stesso parametro template e lo stesso nome. Il broker per questo protocollo è rappresentato dalla classe `Dispatcher`, illustrata in figura 5.2. Entrambi i costruttori degli oggetti ne richiedono un'istanza al repository corrispondente: l'oggetto sender ne possiede il reference in modalità *owner*, e gli oggetti receiver in modalità *user*. Questi ultimi, mediante i metodi `Attach` e `Detach` del Dispatcher, si registrano per ricevere gli aggiornamenti dei dati che vengono inviati.

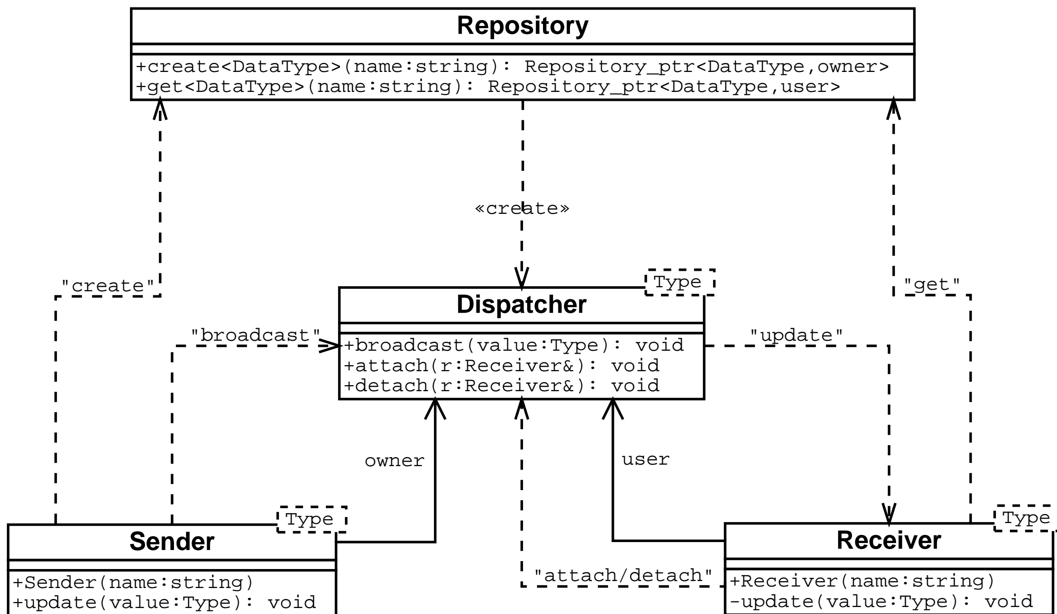


Figura 5.2: Realizzazione del pattern *Information*.

Il corpo del metodo `update()` della classe Sender esegue `broadcast()` sul Dispatcher, che provvede ad inviare la nuova informazione ai Receiver che si sono registrati eseguendo il metodo `update()` di ognuno di essi. Quest'ultimo, oltre a modificare la copia locale del dato, se necessario richiede allo scheduler l'esecuzione dell'attività che è stata registrata come *listener* dell'evento.

5.3 Evoluzione

Il framework che è stato presentato negli ultimi due capitoli si presta a numerose possibilità di estensione, a cui spesso si è già accennato durante la discussione. In particolare, nella realizzazione attuale, il caricamento dei moduli viene eseguito manualmente nella funzione principale dell'applicazione: sarebbe auspicabile introdurre la possibilità di caricare dinamicamente il codice oggetto dei singoli componenti, che possono essere utilizzati come elementi precompilati da utilizzare al momento opportuno.

Per quanto rigurada il pattern di comunicazione *Information*, si sta studiano l'u-

tilità di inserire meccanismi di attivazione delle attività *listener* che siano più evoluti rispetto alla semplice ricezione di un evento, legati all'arrivo di più informazioni differenti, o al valore che esse trasportano.

Il progetto delle componenti di comunicazione è stato studiato in modo da favorire una possibile evoluzione dei meccanismi di trasporto al fine di supportare l'invio e la ricezione di dati remoti, che consentano la ripartizione del carico su più macchine e la teleoperazione del robot. Il middleware CORBA rappresenta in questo senso la soluzione ideale, sposandosi in modo ottimale con le tecniche di programmazione orientate agli oggetti e fornendo recenti estensioni per il real-time. Lo standard definito nel documento *Real-Time CORBA* [51], in particolare, ha proposto l'introduzione di meccanismi di scheduling real-time all'interno delle applicazioni, che saranno valutati al momento della loro disponibilità all'interno degli *ORB*.

Infine, per consentire la realizzazione di applicazioni di elevata complessità, l'architettura deve garantire la possibilità di integrare nel sistema alcune attività di tipo deliberativo, eseguite su processi che non presentano scheduling real-time. L'introduzione di algoritmi che realizzano compiti deliberativi di alto livello tende a favorire l'utilizzo delle tecniche tipiche dell'Intelligenza Artificiale: a questo proposito, esaminando i pattern di comunicazione che sono stati progettati, si possono notare numerose analogie con i *FIPA Communicative Acts* [52], che regolano i protocolli di comunicazione nei sistemi ad agenti; è quindi auspicabile una favorevole integrazione dei due mondi.

Capitolo 6

Sperimentazione

I capitoli 4 e 5 hanno illustrato nel dettaglio gli strumenti per lo sviluppo del software che il framework mette a disposizione. Il passo successivo nella costruzione di un'architettura robotica consiste nella realizzazione un insieme di moduli che fungono da interfaccia verso l'hardware del robot. Sfruttando le funzionalità messe a disposizione da questi moduli sarà infine possibile costruire uno o piú behaviour che ne controllano il comportamento.

Questo capitolo descrive il progetto di alcuni moduli per il Nomad 200 che sono stati realizzati con lo scopo di testare il funzionamento del framework e porre le basi per la costruzione di un'architettura *behaviour-based* per il robot. Le parti più significative del codice sorgente sono disponibili nell'appendice B.

6.1 Interfaccia sensomotoria

I meccanismi di comunicazione che il framework intende fornire, come descritto nel capitolo 5, sono stati realizzati solo parzialmente; nonostante questo è stato possibile sviluppare alcuni moduli elementari che consentono l'utilizzo delle principali componenti dell'hardware del robot all'interno di semplici applicazioni.

6.1.1 Sensori Sonar

Il modulo *Sonar* (listato B.2) comanda l'hardware di controllo dell'anello di sensori sonar del robot (descritti nel paragrafo 2.1.3) e fornisce agli altri moduli del sistema

le misure di distanza che vengono rilevate. Le funzioni di basso livello (listato B.5), realizzate sfruttando il codice di *robotd*, dialogano con l'hardware sfruttando una zona di memoria che viene condivisa tra il processore ed il microcontrollore della scheda *Intellisys 100* grazie alla *Dual Ported RAM (DPR)* motata su di essa. Le distanze misurate dal microcontrollore vengono fornite in pollici, che il modulo provvede a convertire in millimetri prima di inviare.

Le misure prodotte ad ogni ciclo di sparo dei sensori vengono inviate nel sistema tramite il meccanismo di comunicazione *Information* fornito dal framework: la classe `SonarModule` definisce perciò un oggetto `Sender` al suo interno che invia dati aventi il nome simbolico "`sonar`". L'anello è composto di 16 sensori che vengono numerati in senso antiorario a partire dal sonar frontale, perciò, per consentire al modulo *Sonar* di aggiornare singolarmente le misure rilevate, il dato che verrà scambiato tra i moduli viene definito come segue:

```
typedef multiple<unsigned int,16> SonarData;
```

La lettura e l'aggiornamento dei valori misurati sono stati trattati con particolare attenzione, i singoli sensori infatti, a causa dei vincoli imposti dal principio di funzionamento, richiedono tempi di misura piuttosto lunghi e non possono essere utilizzati contemporaneamente. Per garantire una maggiore reattività ai comportamenti del robot diventa importante ridurre il più possibile il tempo di latenza che intercorre tra l'istante in cui la misura viene effettuata dal microcontrollore e l'istante in cui il modulo ne produce l'aggiornamento all'interno del sistema. In questo modo i *behaviour* del robot possono disporre di misure molto recenti dello stato dell'ambiente e prendere decisioni più velocemente a fronte di rapidi cambiamenti del mondo reale.

La classe `SonarModule` definisce al proprio interno un attività, schedulata ciclicamente, che ad ogni esecuzione spedisce l'ultimo valore misurato dal microcontrollore. Purtroppo l'hardware di controllo dei sonar non dà nessuna informazione sull'istante in cui le misure di distanza vengono rilevate, per cui il modulo *Sonar* mantiene una copia locale dei valori letti dai sensori e li confronta con i nuovi valori disponibili sulla *DPR* ad ogni esecuzione dell'attività. In questo modo il modulo può stimare quale il sonar è stato letto per ultimo (l'unico valore che ha subito una modifica), ed inviarne la misura agli altri moduli con frequenza pari allo sparo dei

sonar e latenza non superiore al periodo dell'attività.

La lunghezza del ciclo di lettura di un sensore sonar può essere modificata agendo sull'interfaccia verso l'hardware; il modulo *Sonar* imposta per tale intervallo un valore pari a 24 ms, che sarà pari al periodo dell'attività che viene schedulata al suo interno. A causa della granularità dei timer di sistema la frequenza dell'attività di lettura può essere leggermente ridotta rispetto al valore impostato: questo non pregiudica la correttezza dell'algoritmo.

6.1.2 Sensori Infrarossi

L'anello di sensori infrarossi presente sul Nomad 200 è simile per molti aspetti all'anello di sonar: è composto da 16 sensori numerati in senso antiorario, che sono controllati ancora una volta dall'elettronica della scheda *Intellisys 100* (i valori misurati sono in pollici); i singoli sensori non possono effettuare cicli di misura contemporanei.

Il modulo *Infrared*, realizzato dalla classe `InfraredModule` (listato B.3), rende disponibili le letture dei sensori (in millimetri) mediante il dato "`infrared`" di tipo seguente:

```
typedef multiple<unsigned int,16> InfraredData;
```

Contrariamente a quanto visto per il modulo *Sonar*, in questo caso non è ragionevole aggiornare separatamente i valori misurati, infatti la lettura di un sensore infrarosso è molto più rapida del ciclo di sparo dei sonar (si veda il paragrafo 2.1.3) e la granularità dei timer del sistema operativo impedisce di schedulare un'attività alla frequenza necessaria. Il modulo *Infrared* provvede pertanto a spedire tutte le letture dei sensori infrarossi nel medesimo istante, ad una frequenza tale da consentire al microcontrollore di aggiornare tutte le misure che vengono riportate sulla *DPR*: se la lettura di un sensore impiega al massimo 2.5 ms, la fequenza minima di aggiornamento dell'interno anello sarà pari a 25 Hz.

6.1.3 Motori

Il pattern di comunicazione più adatto per la realizzazione dei moduli di gestione degli attuatori è senza dubbio *Command*, dato però che attualmente il codice che

lo realizza non è ancora stato inserito nel framework, il modulo *Motor* sfrutta in via provvisoria le funzionalità di una classe `Receiver` per ricevere le richieste ed impostare la velocità degli assi di movimento. Per applicazioni di dimensioni abbastanza ridotte (si ricorda che non possono esistere contemporaneamente due moduli “sender” dello stesso dato) il meccanismo è comunque soddisfacente.

La classe `MotorModule`, illustrata nel listato B.4, si registra nel costruttore per la ricezione di informazioni di tipo `VelocityCommand`, che specificano contemporaneamente un valore di velocità per il moto traslatorio del robot (in millimetri al secondo), un valore per la rotazione delle ruote ed uno per la torretta (entrambi in decimi di grado al secondo):

```
struct VelocityCommand { int trasl ; int rotat ; int turret ; };
```

Un valore positivo del parametro `trasl` farà avanzare il robot, mentre un valore positivo di `rotat` e `turret` gli farà compiere una rotazione in senso antiorario. Assegnando valori negativi alle variabili si ottiene lo spostamento in senso inverso. L’informazione scambiata ha il nome simbolico “`velocity`”.

Ad ogni ricezione di un aggiornamento di velocità viene posta in esecuzione una attività che provvede a inviare il comando alla scheda di controllo dei motori tramite l’interfaccia di basso livello (listato B.5) derivata dal codice di *robotd*.

6.2 Test del comportamento

6.2.1 Verifica della reattività

Il primo test che è stato costruito ha lo scopo di misurare la reattività che è possibile ottenere da un semplice comportamento realizzato con la nuova architettura e confrontare infine i risultati sperimentali con un’applicazione analoga che fa uso di *robotd* per connettersi al robot.

La realizzazione del test richiede la costruzione di un *behaviour* di *Collision Avoidance* che tiene sotto controllo la misura dei sensori sonar frontali del robot ed invia un comando di stop ai motori quando viene rilevata la presenta di un ostacolo a distanza ravvicinata. L’applicazione che viene eseguita sfrutta perciò i moduli: *Sonar*, *Motor* e *AvoidCollision*.

La classe `AvoidCollision` (listato B.6) è stata realizzata in modo tale da porre in esecuzione un'attività quando il modulo *Sonar* aggiorna la misura dei sonar frontali; il codice concreto legge la distanza rilevata e ferma il robot quando questo valore diventa inferiore a 35 cm. Lo stesso comportamento è stato realizzato facendo uso delle API esportate da *robotd*; il programma, linkato con la libreria fornita dalla Nomadic, esegue un ciclo infinito in cui vengono eseguite in sequenza le seguenti operazioni:

1. Lettura delle distanze misurate sull'intero anello di sonar;
2. Controllo della misura dei sonar frontali ed eventuale comando di stop;
3. Attesa per il tempo necessario a rinnovare la misura di tutto l'anello[1].

Il test finale si è svolto facendo muovere il robot ad una velocità di 25 $\mathrm{cm/s}$ verso un ostacolo piano (un muro di cartone) con lo scopo di misurare la distanza a cui il comportamento di *Collision Avoidance* ne interrompe il moto: l'intervento può considerarsi tanto più reattivo quanto più la distanza finale dall'ostacolo aumenta. Dato che numerosi fattori casuali possono incidere sul risultato, l'esperimento è stato eseguito varie volte, facendo uso alternativamente delle due applicazioni realizzate; le misure di distanza rilevate sono riportate in tabella 6.1.

Il confronto delle due colonne della tabella mette in evidenza come l'applicazione realizzata con il framework riesca a fermare il robot ad una distanza mediamente maggiore, a parità di parametri, rispetto al programma realizzato mediante robotd: il miglioramento è pari al 37% della distanza totale. Un altro aspetto positivo si registra nella riduzione della la varianza, anche se il fenomeno è di minore entità.

La causa del miglioramento che si è ottenuto è imputabile alla strategia realizzata dal modulo *Sonar* per sincronizzare le letture effettuate dall'hardware di controllo con l'aggiornamento dell'informazione che viene spedita al modulo *AvoidCollision*. Se si esamina il funzionamento del programma realizzato con *robotd* si nota infatti che l'assenza di una sincronizzazione di questo tipo può causare un tempo di latenza tra la fase di sparo del sensore e la sua lettura che induce l'algoritmo realizzato ad analizzare dati mediamente più vecchi di quelli disponibili.

[1] Questo valore, pari a 384 ms, è calcolato moltiplicando il tempo di sparo di un sensore (24 ms) per il numero di sensori presenti sul robot (16).

robotd [cm]	*framework* [cm]
28	32
13	22
26	19
10	23
23	32
21	24
15	31
26	29
18	29
25	28
22	25
12	32
$\eta = 19,92$	$\eta = 27,17$
$\sigma = 5,89$	$\sigma = 4,26$

Tabella 6.1: Distanze misurate nel test di reattività.

6.2.2 *Wall Following*

Per mostrare che il framework consente di realizzare comportamenti di interesse concreto per il robot è stata sfruttata l'architettura disponibile per costruire un'applicazione elementare che consenta al Nomad di muoversi in un ambiente chiuso mediante la sola navigazione sensoriale, accostandosi alla parete più vicina e mantenendone costante la distanza durante il moto.

6.2.2.1 Progetto

L'applicazione che si vuole realizzare consiste nel controllo continuo dell'allineamento del robot con la superficie verticale del muro, che per semplicità supporremo alla sua sinistra. Il sistema, schematizzato in figura 6.1, è retto dal seguente modello matematico:

$$\begin{cases} \dot{\alpha} = \omega \\ \dot{d} = v \sin \alpha \end{cases} \tag{6.1}$$

Dove v e ω sono rispettivamente la velocità lineare ad angolare del robot, α rappresenta l'angolo tra il muro e la direzione istantanea di movimento e d misura la distanza che si vuole controllare. La velocità lineare del robot viene mantenu-

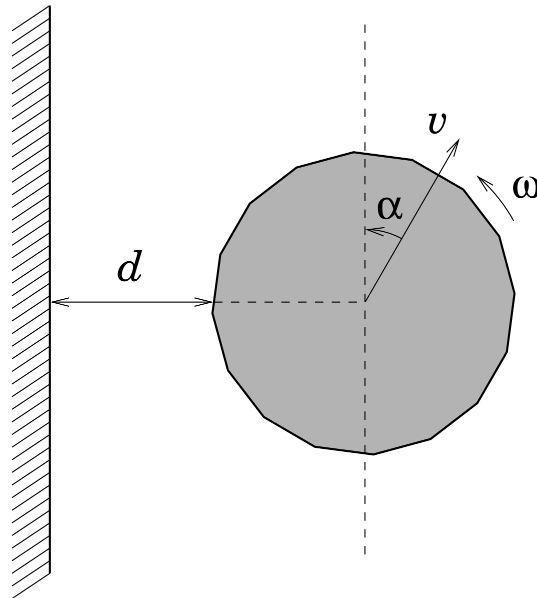


Figura 6.1: Modello geometrico del sistema.

ta costante durante il moto, per cui l'unico *ingresso manipolabile* del sistema è la velocità angolare ω.

Per regolare dinamicamente la distanza del robot da muro viene utilizzata una tecnica di *controllo in retroazione* realizzata in due due fasi.

Il primo anello di retroazione (figura 6.2) consente di regolare l'angolo α impostando una velocità di rotazione ω proporzionale alla differenza tra il *set-point* istantaneo α^* e il valore misurato $\hat{\alpha}$. Se vale il modello proposto, il *regolatore proporzionale* (avente guadagno pari a K_α) assicura la convergenza verso l'esatto valore del set-point grazie al comportamento integrale del sistema. Da semplici considerazioni geometriche si deduce che dovrà essere: $K_\alpha < 0$.

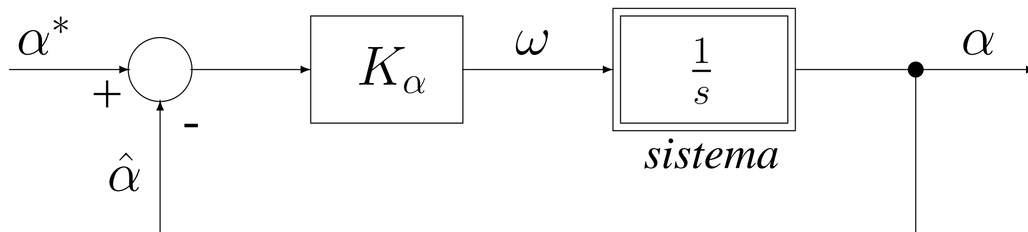


Figura 6.2: Regolatore dell'angolo α.

Il secondo anello di retroazione (figura 6.3) può a questo punto sfruttare il set-point α^* per impostare una correzione dinamica che consenta di regolare la distanza d dal muro in base ad un valore d^* deciso a priori. Come si vede dallo schema, anche per questo regolatore è stato utilizzato un semplice controllo di tipo proporzionale (in questo caso caso avremo $K_d > 0$), infatti il sistema controllato presenta ancora un comportamento pressochè integrale per valori di α vicini allo zero, presso cui vale l'approssimazione: $\sin\alpha \simeq \alpha$.

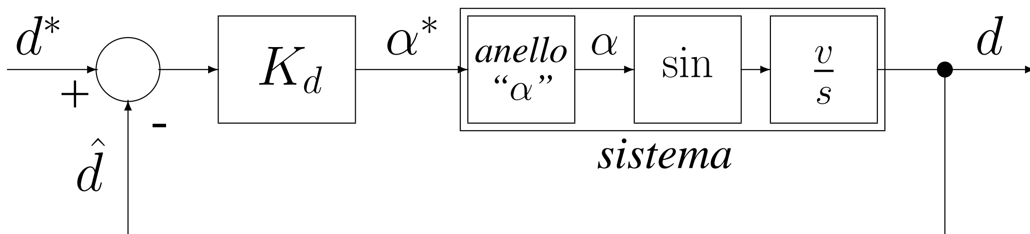


Figura 6.3: Regolatore della distanza d.

Per realizzare concretamente il controllo serve infine un metodo che consenta di risalire alla misura istantanea di α e d a partire dai sensori del robot. Per misurare le distanze dagli ostacoli circostanti, il Nomad 200 fornisce due anelli separati di sensori, sonar ed infrarossi, che presentano la medesima geometria; per questo motivo le considerazioni seguenti saranno valide per entrambe le categorie di sensori.

Siano $\{d_i\}_{i=0..8}$ le misure di distanza rilevate dai sensori posti lungo il lato sinistro del robot, allora definiamo m come la posizione del sensore che ha misurato la distanza minore, esclusi quello frontale e quello posteriore:

$$m : d_m = \min_{j=1..7}\{d_i\}; \tag{6.2}$$

I valori di α e d sono dati da:

$$\begin{cases} \alpha = \alpha_\Delta + \alpha_\delta \\ d = d_m \end{cases}; \tag{6.3}$$

Dove i due contributi all'angolo α derivano dalla geometria dell'anello di sensori (figura 6.4):

- α_Δ dipende dall'indice m del sensore che ha rilevato la minore distanza:

$$\alpha_\Delta = \frac{\pi}{8}(m-4); \tag{6.4}$$

- α_δ viene ricavato dalle misure dei sensori $m-1$ ed $m+1$:

$$\sin \alpha_\delta = \frac{d_{m-1} - d_{m+1}}{l}. \tag{6.5}$$

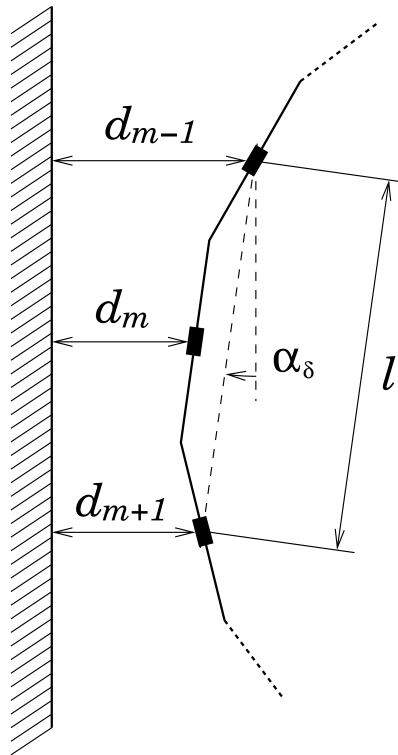


Figura 6.4: Stima di α_δ e d mediante i sensori del robot.

6.2.2.2 Realizzazione

Per determinare la posizione reciproca tra il robot e la parete si è deciso di utilizzare i sensori infrarossi, poichè questi ultimi forniscono misure più precise quando il Nomad si trova molto vicino agli ostacoli; i moduli utilizzati nell'applicazione sono allora: *Infrared*, *Motor* e *WallFollowing*.

Il modulo `WallFollowing`, illustrato nel listato B.7, realizza il controllo descritto in precedenza eseguendo un'attività periodica con frequenza pari a 10 Hz: ad ogni iterazione il codice del modulo provvede a stimare i valori correnti di α e d (sulla base dei dati sensoriali forniti dal modulo *Infrared*), calcolare il nuovo valore di ω mediante il controllo di tipo proporzionale, ed inviare infine al modulo *Motor* i nuovi valori di velocità. Per garantire l'allineamento delle ruote con la torretta il moto rotatorio sarà sempre lo stesso per entrambi gli assi.

Impostando una velocità linerare di 25 $\mathrm{cm/s}$ ed un set-point di distanza pari a 20 cm, il test è proseguito effettuando numerose prove che hanno consentito di individuare i valori dei parametri K_α e K_d che consentono al robot di avanzare senza oscillazioni pur seguendo correttamente l'andamento della parete. Durante quest'ultima fase è emersa l'impossibilità di controllare adeguatamente il robot quando lo spazio frontale diventa molto ridotto, ad esempio perchè il muro forma un angolo: in situazioni di questo tipo il behviour progettato è insufficiente per consentire al Nomad di muoversi senza investire la parete, ed il motivo è dovuto al fatto che la velocità traslatoria impedisce al robot di compiere raggi di curvatura abbastanza stretti. Per risolvere questo problema si è deciso di aggiungere all'interno dell'applicazione una semplice regolazione della velocità di avanzamento all'interno del modulo WallFollowing: ad ogni iterazione del controllo la velocità v viene impostata in modo proporzionale al valore di distanza rilevato dal sensore infrarosso n.1, se la misura supera i 35 cm la velocità non sale oltre i 25 $\mathrm{cm/s}$.

Una serie di test condotti all'interno degli edifici del Dipartimento ha mostrato la correttezza del sistema realizzato. Il robot ha percorso tutto lo spazio a disposizione senza la necessità di supervisione, mantenendosi a distanza costante dalla parete senza oscillazioni apprezzabili. Il comportamento si adegua velocemente alle variazioni graduali dell'angolo del muro ed è in grado di superare senza problemi le rientranze fino a circa 20 cm. Per discontinuità di entità maggiore il robot tende a curvare repentinamente e correggere in seguito la direzione con qualche oscillazione residua. Se lo spazio si riduce eccessivamente la velocità cala ed il comportamento riesce a riportare il Nomad nella corretta direzione di moto, evitando le collisioni con le pareti.

Capitolo 7

Conclusioni

In questa tesi è stato affrontato il problema della realizzazione di un'architettura robotica, composta di hardware e software, che permetta di costruire applicazioni concrete in modo agevole e robusto, garantendo nel contempo la possibilità di sostituirne agevolmente le componenti ed estendere il supporto per nuove funzionalità, grazie anche all'apertura verso gli standard che sono disponibili nel settore informatico. Il progetto è stato realizzato adottando una granularità fine dei componenti, affinché sia possibile trarre profitto dai moduli che lo costituiscono, nei molteplici contesti che possono presentarsi durante l'utilizzo.

La prima fase del lavoro è consistita nell'aggiornamento del *Nomad 200*, al fine di disporre di un robot versatile e funzionale con cui effettuare verifiche significative. L'hardware originale del robot presentava alcuni limiti, emersi durante i precedenti progetti che lo hanno coinvolto, che sono stati superati grazie alla sostituzione del sistema di elaborazione con componenti di nuova generazione. Il miglioramento nella versatilità del Nomad è stato notato immediatamente, anche se un riscontro significativo si potrà ottenere solo con la realizzazione di applicazioni complesse che possano trarre reale beneficio dall'aumento di potenza di calcolo.

Il contributo principale della tesi è stato il progetto e la realizzazione di un framework che consenta di agevolare il lavoro di sviluppo di architetture robotiche di tipo reattivo, composte da un insieme di moduli indipendenti, mettendo a disposizione un set completo di primitive per la comunicazione e lo scheduling realtime. La progettazione della libreria è avvenuta facendo riferimento alle architetture

sviluppate dai maggiori centri di ricerca internazionali, esaminando le scelte e le motivazioni che emergono dalle soluzioni proposte.

L'ultima fase del lavoro di tesi ha riguardato la creazione di moduli compatibili con l'architettura del Nomad 200 per analizzare e verificare concretamente il funzionamento del framework sviluppato. In particolare i behaviour che sono stati prodotti hanno permesso lo sviluppo di alcune applicazioni di navigazione che hanno mostrato buone prestazioni pur mantenendo una sufficiente semplicità, astrazione e rapidità di sviluppo nella realizzazione del codice.

Le probabili estensioni di questo lavoro di tesi sono le seguenti:

- L'evoluzione dei protocolli di comunicazione del framework, nei quali è auspicabile l'inserimento di meccanismi per lo scambio di informazioni tra moduli remoti, che sarà basato presumibilmente su *middleware CORBA*.

- L'introduzione di un supporto per l'esecuzione di task a priorità non real-time, che sarà necessario per realizzare attività dalla forte componente deliberativa. Queste estensioni consentiranno di utilizzare il framework per la costruzione di sistemi robotici basati su architetture di tipo *ibrido*.

- La ricerca di un possibile miglioramento del supporto realtime mediante l'utilizzo di un sistema operativo più completo, come *Solaris SunOS*. Questa necessità deriva dal fatto che il sistema operativo *Linux*, almeno nella versione corrente del kernel, fornisce strumenti per l'esecuzione di processi in tempo reale che si sono dimostrati piuttosto limitati.

- La costruzione di un insieme completo di moduli che forniscano l'accesso a tutto l'apparato sensomotorio del Nomad 200, dato che attualmente è disponibile un'interfaccia solo per i sensori sonar, infrarossi e per il controllo dei motori, ed, eventualmente, di altri robot mobili.

Appendice A

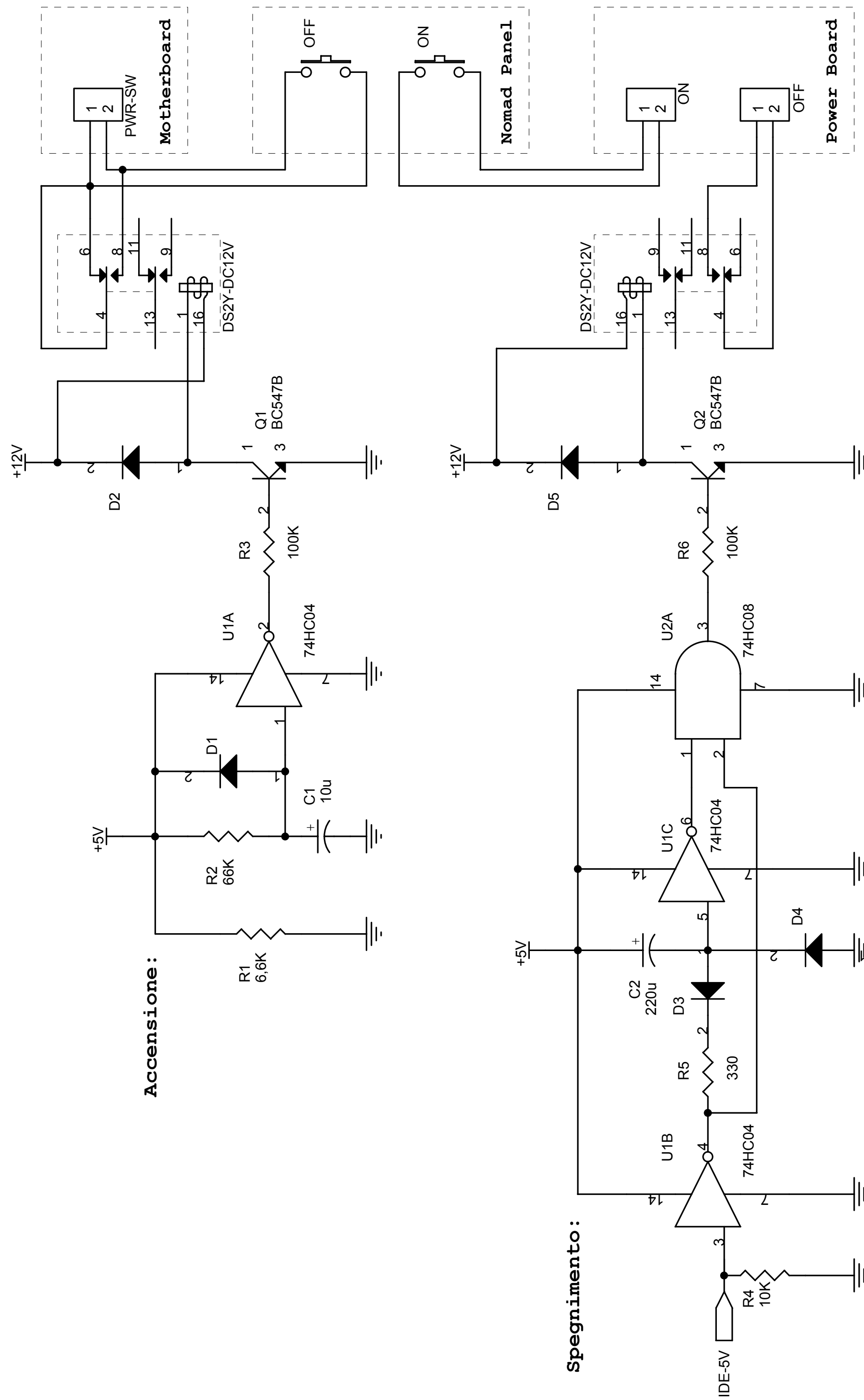
Circuiti di accensione e spegnimento

Il circuito illustrato nella pagina successiva è stato realizzato per consentire una corretta fase di accensione e spegnimento del PC che si trova a bordo del Nomad quando vengono premuti i pulsanti *On* e *Off* del pannello di controllo del robot:

- Quando viene premuto il tasto *On*, di cui è stato mantenuto il collegamento originale, la scheda di alimentazione inizia a fornire le tensioni $+5V$ e $+12V$; il circuito di *accensione* chiude brevemente il contatto *PWR-SW* sulla scheda madre del PC, assicurandone lo start-up.

- Quando viene premuto il taso *Off* il PC inizia una fase di shut-down, che termina con lo spegnimento delle tensioni fornite dall'alimentatore ATX, tra cui la tensione *IDE-5V*; il circuito di *spegnimento*, quando tale segnale presenta un fronte di discesa, chiude il connettore *OFF* della scheda di alimentazione del robot, che si spegne completamente.

Gli impulsi di chiusura dei relè presenti nel circuito sono generati dalla carica dei condensatori C_1 e C_2, quindi è importante utilizzare porte logiche della famiglia *74HCxx*, che non assorbono corrente sugli ingressi, per mantenere corretti i valori dei ritardi. I diodi D_1 e D_4 consentono ad entrambi i condensatori di scaricarsi (tramite la resistenza R_1) quando l'alimentazione viene spenta. Entrambe le tensioni $+5V$ e $+12V$ possono essere prelevate dai connettori di alimentazione dei dispositivi IDE che vengono forniti sulla scheda di alimentazione della Nomadic.



Motherboard
PWR-SW
Nomad Panel
OFF
ON
Power Board
DS2Y-DC12V
+12V
D2
Q1
BC547B
R3
100K
U1A
74HC04
D1
C1
10u
+5V
R2
66K
R1
6,6K
Accensione:
D5
Q2
BC547B
R6
100K
U2A
74HC08
U1C
74HC04
D4
C2
220u
D3
R5
330
U1B
74HC04
R4
10K
IDE-5V
Spegnimento:

Appendice B

Codice dei moduli per il Nomad 200

In questa appendice viene illustrato il codice sorgente dei moduli che sono stati realizzati per il Nomad 200. I listati riportano soltanto le parti più significative del codice, che è stato parzialmente ridotto per aumentarne la chiarezza. Per la descrizione degli algoritmi si veda il capitolo 6.

Listato B.1: Definizioni di uso comune nei moduli.

```
// Sonar:
const string SONAR_NAME = "sonar";
typedef multiple<unsigned int, 16> SonarData; // mm

// Infrarossi :
const string INFRARED_NAME = "infrared";
typedef multiple<unsigned int, 16> InfraredData ; // mm

// Motori:
const string VELOCITY_NAME = "velocity";
struct VelocityCommand
  {
  int trasl ;  // mm/sec
  int rotat ;  // decimi di grado/sec
  int turret ; // decimi di grado/sec
  };
```

B.1 Interfacce sensomotorie

Listato B.2: Modulo di controllo dei sensori sonar.

```
class SonarModule : public Module
  {
  const double SONAR_FIRERATE = 0.024; // secondi

  private:
     Activity<void,void> read;
    Sender< SonarData > sonar;

     // valori in decimi di pollice
    SonarData::value_type values[SonarData::size];
     size_t startPoint;

    void readBody()
       {
       for ( size_t i = startPoint;
          i != circleDec( startPoint );
          i = circleInc (i) )
          {
          SonarData::value_type new_val = get_sonar(i);
          if ( new_val != values[i] )
             {
             values[i] = new_val;
              startPoint = i;
             break;
             }
          }
       // NB: 0.1 inch = 2.54 mm
       sonar.update ( 2.54 * values[ startPoint ], startPoint );
        startPoint = circleInc ( startPoint );
       }

     static size_t circleInc ( size_t v )
       { return ( v + 1 ) % SonarData::size; }
     static size_t circleDec ( size_t v )
       { return ( v + SonarData::size - 1 ) % SonarData::size; }

  public:
```

```
    SonarModule() :
      Module( "Sonar" ),
      read ( * this, &SonarModule::readBody, SONAR_FIRERATE ),
      sonar ( * this , SONAR_NAME ),
      startPoint ( 0 )
      {
      init_sonar ( SONAR_FIRERATE );

      for ( size_t i = 0; i < SonarData:: size ; ++i )
        values [ i ] = 0;

      read . startPeriodicRun ();
      }
  };
```

Listato B.3: Modulo di controllo dei sensori infrarossi.

```
class InfraredModule : public Module
  {
  private:
    Activity <void,void> read;
    Sender< InfraredData > infrared ;

    void readBody()
      {
      for ( size_t i = 0; i != InfraredData :: size ; ++i )
        {
        InfraredData :: value_type value = get_infrared ( i );
        infrared .update ( 2.54 * value , i ); // dec . inch −> mm
        }
      }
  public:
    InfraredModule () :
      Module( " Infrared " ),
      read(* this, &InfraredModule :: readBody, 0.0025* InfraredData :: size ),
      infrared ( * this , INFRARED_NAME )
      {
      init_infrared ();
      read . startPeriodicRun ();
      }
  };
```

Listato B.4: Modulo di controllo dei motori.

```
class MotorModule : public Module
  {
  private:
     Activity <void,void> setVel;
     Receiver< VelocityCommand > velocity;

     void setVelBody()
       {
       VelocityCommand vel = velocity.get();
       int inch_vel = vel.trasl / 2.54;
       dmc_set_vel( inch_vel, vel.rotat, vel.turret );
       }

  public:
     MotorModule() :
       Module( "Motor" ),
       setVel ( *this, &MotorModule::setVelBody, 0.02 ),
       velocity ( *this, VELOCITY_NAME )
       {
       init_dmc();
       velocity.setListener( setVel );
       }
  };
```

Listato B.5: Interfaccia di basso livello verso l'hardware.

```
// Sonar (misure in pollici)
void init_sonar( double firerate );
unsigned int get_sonar( size_t pos );

// Infrarossi (misure in pollici)
void init_infrared();
unsigned int get_infrared( size_t pos );

// Motori ( velocita in dec. di pollice/sec e dec. di grado/sec)
void init_dmc();
void dmc_set_vel( int trasl, int rotat, int turret );
```

B.2 *Behaviour* per il robot

Listato B.6: Comportamento “*Avoid Collision*” per il robot.

```
class AvoidCollision : public Module
  {
  private:
    const static unsigned int SONAR_THRESHOLD = 350; //mm
    const static VelocityCommand STOP = { 0, 0, 0 };

    Activity <void,void> check;
    Receiver< SonarData > sonar;
    Sender< VelocityCommand > velocity;

    void collChecker ()
      {
      size_t pos = sonar.getNewestPos();

      if ( sonar.get(pos) <= SONAR_THRESHOLD )
        velocity .update ( STOP );
      }

  public:
    AvoidCollision () :
      Module( "AvoidCollision" ),
      check( *this, &AvoidCollision :: collChecker , 0.02 ),
      sonar ( *this , SONAR_NAME ),
      velocity ( *this , VELOCITY_NAME )
      {
      sonar. setListener ( check,  0 );
      sonar. setListener ( check,  1 );
      sonar. setListener ( check , 15 );
      }
  };
```

Listato B.7: Comportamento “*Wall Following*” per il robot.

```
class WallFollowing : public Module
  {
  private:
     // Set point di distanza dal muro (mm)
    const static double desiredDistance = 200.0;

     // Distanza geometrica tra due sensori in posizione i e i+2
    const static double SENSOR_DISTANCE = 170.0; // Distanza in mm

     Activity <void,void> controller ;
    Receiver< InfraredData > infrared ;
    Sender< VelocityCommand > velocity;

    double wallDistance ; // distanza stimata dal muro (mm)
    double wallAngle;      // angolo stimato con il muro(rad)
    double freeSpace ;     // spazio rilevato davanti al robot ( mm)

     InfraredData :: value_type values [ InfraredData :: size ];

    void readSensors ()
       {
        // Aggiornamento del valore degli infrared
       for ( size_t i = 0; i < 9; ++ i )
          {
          try { values [ i ] = infrared . get( i ); }
          catch ( EmptySlot& e ) { /* old value used */ }
          }
       freeSpace = values [1];

        // Calcolo della posizione che da ' il valore minimo:
        //    se tutti i sensori segnalano distanza massima
        //    l ' angolo wallAngle risulta pari a 0.
       int minPos = 4; // ( grazie a questa istruzione )
       for ( size_t i = 1; i < 8; ++ i )
          if ( values [ i ] < values [minPos] )
             minPos = i ;
       wallDistance = values [minPos];

       double alpha_DELTA = M_PI / 8.0 * ( minPos - 4 );
```

```
    double sin_alpha_delta =
      ( values[minPos-1] - values[minPos+1] ) / SENSOR_DISTANCE;

    wallAngle = alpha_DELTA + asin( sin_alpha_delta );
    }

  void motorController ()
    {
    readSensors ();

    // rad
    double desiredAngle =
      P_control ( 0.006, desiredDistance , wallDistance , 1.05, 0.05 );

    // decimi di grado/s
    double rotatSpeed =
      P_control ( -300.0, desiredAngle , wallAngle , 450.0, 0.0 );

    // 250 mm/sec quando il muro dista 350 mm
    double traslSpeed = freeSpace * 250.0 / 350.0;
    if ( traslSpeed > 250.0 )   // saturazione
      traslSpeed = 250.0;

    VelocityCommand vel = { traslSpeed , rotatSpeed , rotatSpeed };
    velocity .update( vel );
    }

  // Regolatore proporzionale con soglia massima e minima
  static double P_control ( double K, double setPoint , double measure,
    double maxVal, double minVal );

public:
  WallFollowing () :
    Module( "WallFollowing" ),
    controller ( * this, &WallFollowing:: motorController , 0.1 ),
    infrared ( * this , INFRARED_NAME ),
    velocity ( * this , VELOCITY_NAME )
    { controller . startPeriodicRun (); }
};
```

Bibliografia

[1] N. Nilsson and R.E. Fikes. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2), 1971.

[2] R. Arkin. *Behavior-based Robotics*. 1998.

[3] R.A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[4] R.A. Brooks. A Robot that Walks; Emergent Behavior from a Carefully Evolved Network. In *IEEE International Conference on Robotics and Automation '89*, pages 292–296, May 1989.

[5] E. Gat. Integrating Planning and Reaction in a Heterogeneous Asynchronous Architecture for Controlling Mobile Robots. In *Tenth National Conference on Artificial Intelligence (AAAI)*, 1992.

[6] D. Lyons, A. Hendriks, and S. Mehta. Achieving robustness by casting planning as adaptation of a reactive system. In *IEEE conference on Robotics and Automation*, 1991.

[7] M. Schoppers. A Software Architecture for Hard Real-Time Execution of Automatically Synthetized Plans or Control Laws. In *Conf. on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS '94)*, March 1994.

[8] R. Alami et al. An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315–337, April 1998.

[9] Open RObot COntrol Software, OROCOS. `http://www.orocos.org`.

[10] OROCOS at LAAS. `http://www.laas.fr/~mallet/orocos`.

[11] S. Fleury, M. Herrb, and R. Chatila. GenoM: a Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. Technical report, Laboratory for Analysis and Architecture of Systems (LAAS).

[12] C. Schlegel. Communications Patterns for OROCOS. Hints, Remarks, Specifications. Technical report, Research Institute for Applied Knowledge Processing (FAW), February 2002.

[13] OROCOS at FAW. *`http://www1.faw.uni-ulm.de/orocos`*.

[14] OROCOS at KTH. *`http://cogvis.nada.kth.se/orocos`*.

[15] R. Volpe et al. The CLARAty Architecture for Robotic Autonomy. Technical report, Jet Propulsion Laboratory (JPL), 2001.

[16] I.A.D. Nesnas. Toward Developing Reusable Software Components for Robotic Applications. Technical report, Jet Propulsion Laboratory (JPL), 2001.

[17] S.A. Blum. Towards a Component-based System Architecture for Autonomous Mobile Robots. Technical report, Institute for Real-Time Computer Systems, Technische Universität München.

[18] L.B. Becker and C.E. Pereira. SIMOO-RT: An Object-Oriented Framework for the Development of Real-Time Industrial Automation Systems. *IEEE Tansactions on Robotics and Automation*, 18(4):421–430, 4 August 2002.

[19] N.R.S. Raghavan and T. Waghmare. DPAC: An Object-Oriented Distributed and Parallel Computing Framework for Manufacturing Applications. *IEEE Tansactions on Robotics and Automation*, 18(4):431–443, 4 August 2002.

[20] D. Brugali and M.E. Fayad. Distributed Computing in Robotics and Automation. *IEEE Tansactions on Robotics and Automation*, 18(4):409–420, 4 August 2002.

[21] G. Butler, A. Gantchev, and P. Grogono. Object-oriented design of the subsumption architecture. *Software Practice and Experience*, 31:911–923, 2001.

[22] C. Pescio. C++, Java, C#: qualche considerazione. *C++ Informer*, (12), 12 October 2000. http://www.eptacom.net.

[23] M. Salichs et al. Pattern-Oriented Implementation for Automatic and Deliberative Skills of a Mobile Robot. In *1st Int'l Workshop on Advances in Service Robotics (ASER 03)*, 2003.

[24] R.E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[25] E. Gamma, R. Helm, R. Jhonson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[26] J.C. Cleaveland. Building application generators. *IEEE Software*, (4):25–33, July 1988.

[27] M. Piaggio, A. Sgorbissa, and L. Tricerri. *Expert Tribe in a Hybrid Network Operating System*. Università di Genova - Dip. di Informatica Sistemistica e Telematica, 4.2.1 edition, 29 January 1999.

[28] Nomadic Technologies Inc. *The Nomad 200 User Guide*, December 1993.

[29] Galil Motion Control Inc. Multi Axis Motion Controllers. http://www.galilmc.com.

[30] SensCorp Inc. Electrostatic Sonar Transducers. http://www.senscomp.com.

[31] Directed Perception Inc. Pan-Tilt Units. http://www.dperception.com.

[32] RC System Inc. DoubleTalk PC Card. http://www.rcsys.com.

[33] Proxim Inc. RangeLan Tools. http://www.proxim.com.

[34] D. Diemmi. Navigazione di Robot Mobili: Sviluppo di Strumenti di Simulazione e Realizzazione di Esperienze Applicative. Tesi di Laurea in Ingegneria Elettronica, Università degli Studi di Parma, 1995.

[35] C. Bertani. Navigazione di Robot Mobili basata su Apprendimento con Rinforzo. Tesi di Laurea in Ingegneria Elettronica, Università degli Studi di Parma, 1996.

[36] A. Calafiore. Navigazione di Robot Mobili in Ambiente Strutturato Basata su Visione Artificiale. Tesi di Laurea in Ingegneria Elettornica, Università degli Studi di Parma, 1996.

[37] L. Capuzzello. Sviluppo di Compiti di Navigazione mediante Supporti Evoluti di Programmazione Concorrente. Tesi di Laurea in Ingegneria Elettronica, Università degli Studi di Parma, 1997.

[38] F. Monica and D. Pallastrelli. Controllo remoto per il robot Nomad, 2001.

[39] SOYO Computer Inc. SY-7VEM Motherboard. `http://www.soyousa.com/products`.

[40] Videre Design. MEGA-D Megapixel Digital Stereo Head. `http://www.videredesign.com/products.htm`.

[41] ActionTec Electronics Inc. 802.11b USB Wireless Adapter. `http://www.actiontec.com/products`.

[42] University of Edinburgh. The Festival Speech Synthesis System. `http://www.cstr.ed.ac.uk/projects/festival`.

[43] Key Power Inc. DC-DC Power Supply. `http://www.keypower.com`.

[44] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, volume 2. 2000.

[45] B. Meyer. *Object-Oriented Software Construction*. 2nd edition, 1997.

[46] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[47] A. Alexandrescu. *Modern C++ Design, Generic Programming and Design Pattern Applied*. 2001.

[48] R. Arkin and T. Balck. AuRA: Principles and Practice in Review. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:175–189, 1997.

[49] L.P. Briand and D.M. Roy. *Meeting Deadlines in Hard Real-Time Systems - The Rate Monotonic Approach*. 1999.

[50] D. Pallastrelli. Studio e Realizzazione di un Framework Orientato agli Oggetti per Applicazioni Real-time. Tesi di Laurea in Ingegneria Informatica, Università degli Studi di Parma, 2002.

[51] Object Management Group. Real-Time CORBA Specification, August 2002.

[52] Foundation for Intelligent Physical Agents. Fipa Communicative Act Library Specification. `http://www.fipa.org`.

[53] J.J. Borrelly et al. The ORCCAD Architecture. *The International Journal of Robotics Research*, 17(4):338–359, April 1998.

[54] S.A. Schneider et al. ControlShell: A Software Architecture for Complex Electromechanical Systems. *The International Journal of Robotics Research*, 17(4):360–380, April 1998.