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

PROGETTAZIONE DI UN'ARCHITETTURA SOFTWARE RICONFIGURABILE E IN TEMPO REALE PER UN MANIPOLATORE ROBOTICO EVOLUTO

Relatore:
Chiar.mo Prof. Stefano Caselli

Correlatore:
Dott. Ing. Francesco Zanichelli
Dott. Ing. Giuseppe Beccari

Tesi di laurea di:
Monica Reggiani

Anno Accademico 1996-1997

Indice

Capitolo 1

Introduzione

Come ogni creatura biologica richiede la coordinazione di centinaia di muscoli per generare un comporamento utile, anche la più semplice macchina o robot richiede la coordinazione di diversi componenti per compiere un lavoro utile. È pertanto necessario progettare un'architettura per il coordinamento e la cooperazione.
Lo scopo di questa tesi è fornire al progettista gli strumenti per realizzare con maggiore facilità queste architetture di governo rivolgendosi in particolare al campo dei manipolatori robotici evoluti.

1.1 Motivazioni

La ricerca ha sviluppato architetture per guidare la costruzione di sistemi robotici che devono operare in un ambiente reale, cioè in un ambiente inaccessibile (i sensori sono imperfetti e percepiscono solo gli stimoli nelle vicinanze del robot), non deterministico (un robot deve considerare l'incertezza nei valori su cui basa le sue decisioni) e dinamico (un robot deve sapere quando è meglio deliberare e quando deve reagire immediatamente). Le proposte per i sistemi di governo possono essere raggruppate in tre categorie: *architetture deliberative*, *architetture reattive* e *architetture ibride*.
La *tecnica deliberativa* [1] rappresenta l'approccio classico ed è attualmente l'approccio più consolidato nelle applicazioni perché sono disponibili precise metodologie per aiutare il progettista nello sviluppo del sistema di controllo. Un'architettura deliberativa contiene, esplicitamente rappresentato, un modello simbolico del mondo. L'architettura di controllo prende delle decisioni

attraverso un ragionamento logico, basato sulla conformità a determinate condizioni e/o su una manipolazione simbolica. Poter intraprendere azioni basandosi sul solo ragionamento logico è un'idea molto attraente, ma con due problemi ancora lontani dall'essere completamente risolti: quali aspetti del mondo reale rilevare perché la descrizione accurata consenta di prendere decisioni adeguate e come rappresentare le informazioni e porle in relazione tra loro in modo da costruire il modello simbolico su cui si basa il ragionamento logico.

Nella metà degli anni '80 sono state proposte *architetture reattive* per rendere il governo adeguato anche in sistemi soggetti a vincoli temporali. Nelle tecniche deliberative, infatti, l'azione di deliberazione è insensibile alle variazioni in tempo reale che inevitabilmente avvengono in un ambiente dinamico, rendendo il piano a lungo termine obsoleto, indipendentemente dalla complessità del modello che lo supporta.

Le *architetture reattive* [2],[3] comprendono una speciale categoria di architetture che non possiedono un modello simbolico dell'ambiente in cui operano e che non usano un ragionamento simbolico complesso; le decisioni vengono prese in risposta a stimoli che ricevono dal mondo in cui si trovano. Solitamente le architetture reattive sono composte da un insieme di moduli raggruppati e posti in livelli (che operano in modo asincrono) in modo tale che i moduli a livelli più bassi possano inibire quelli a livelli più alti. Ogni livello ha uno scopo o comportamento: ai livelli più bassi si trovano i comportamenti più primitivi.

Pur offrendo vantaggi come la semplicità e una limitata necessità di risorse computazionali, queste architetture sono una valida risposta solo per sistemi di governo che devono perseguire scopi semplici. Quando è richiesto un comportamento complesso la cui esecuzione richiede conoscenze sull'ambiente non immediatamente disponibili attraverso la percezione, non si potrà più progettare il compito con un'architettura che risponde ai soli stimoli.

Le esperienze sulle architetture completamente reattive o completamente deliberative hanno spinto diversi ricercatori ad approntare tecniche che consentissero un approccio ibrido al problema sposando gli approcci classici a quelli alternativi.

Le *architetture ibride* [4] [5] sono costituite da due sottosistemi: uno deliberativo, contenente un modello simbolico del mondo, che sviluppa piani e prende decisioni con le tecniche di ricerca proposte dal filone principale dell'intelligenza artificiale simbolica; uno reattivo, che è in grado di reagire ad

un evento in un ambiente senza dover far intervenire un ragionamento complesso.
L'esperienza porta a cercare di realizzare un'architettura ibrida gerarchica composta da moduli di controllo intelligenti, ognuno con funzioni specifiche. Questi moduli di controllo sono eterogenei. Alcuni hanno la responsabilità di filtrare le informazioni sensoriali, stimare lo stato delle variabili o rilevare eventi. Altri hanno quella di generare piani, allocare risorse o compiere attività di scheduling. Altri ancora devono settare priorità, calcolare costi e benefici, mantenere un inventario delle risorse utilizzate, correggere errori o agire sui malfunzionamenti. Alcuni moduli hanno azioni che si concludono in millisecondi, altre che si concludono in secondi o minuti, altre ancora che non si concludono se non con la conclusione del compito stesso.
Ogni modulo di controllo opera concorrentemente ed indipendentemente eseguendo le proprie funzioni in relazione allo stato dell'ambiente. Quindi, il sistema non è soggetto all'inflessibilità dei sistemi centralizzati completamente deliberativi. Inoltre, non occorre avere un modulo separato per ogni comportamento e nemmeno che tutti i comportamenti siano continuamente attivi come nelle architetture reattive, perché un sistema di controllo centrale provvederà a scegliere i moduli di controllo necessari a conseguire lo scopo. Le architetture ibride, sono un campo attivo di ricerca perché sono in grado di esprimere sia la flessibilità della componente deliberativa, determinando comportamenti "intelligenti" anche in domini dinamici e complessi, sia la componente reattiva, garantendo caratteristiche di predicibilità. Resta peraltro aperto il problema di come realizzare i sottosistemi deliberativi e reattivi e di come farli interagire in modo coerente in una organica struttura di controllo. Uno dei problemi maggiori nel combinare componente deliberativa e componente reattiva è la complessità. Un sistema in tempo reale, che realizza la componente reattiva, deve garantire il rispetto di vincoli temporali che l'ambiente impone, anche nel caso peggiore. Questa richiesta mal si sposa con i sistemi deliberativi poiché molte delle tecniche di intelligenza artificiale non consentono un'analisi per individuare i tempi di risposta peggiori. Inoltre, le tecniche di intelligenza artificiale spesso presentano una forte varianza nei tempi di risposta che rende un sistema progettato per garantire l'esecuzione nel caso peggiore notevolmente sottoutilizzato durante le operazioni normali.
Per questi problemi, le metodologie di programmazione ed i meccanismi di sistema operativo in grado di supportare in modo adeguato le due componenti

rimangono un tema di ricerca ancora aperto e di vivo interesse.

1.2 Approcci e contributi

Per cercare di armonizzare la componente di intelligenza artificiale con i metodi di calcolo in tempo reale, la ricerca ha proposto diverse architetture in cui componente reattiva e componente deliberativa collaborano fra di loro. A partire da un'analisi di questi lavori, nell'ambito della tesi, è stata progettata un'architettura organizzata su tre livelli:

- un livello di pianificazione, che realizza il sottosistema di intelligenza artificiale,
- un livello di esecuzione, che realizza il sottosistema in tempo reale,
- un livello di controllo, che rende autonomi i due sottosistemi precedenti.

Lo scopo principale del lavoro di tesi è stato di automatizzare la progettazione, la realizzazione e le modifiche del sistema di controllo in tempo reale. In particolare, è stata progettata e realizzata una libreria di moduli di controllo astratti che possono essere specificati per la costruzione di architetture di governo. Attraverso la selezione e configurazione degli opportuni moduli di controllo estratti dalla libreria, il progettista può specificare l'architettura di governo di un sistema robotico astraendo dai problemi di gestione delle priorità, dei vincoli temporali e della comunicazione tra trame di controllo.

1.3 Il dominio di esempio

Utilizzando gli strumenti disponibili presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma, è stata realizzata un'applicazione con sorgenti di incertezza che richiedono un sistema di governo in grado di adattare all'ambiente le azioni di un sistema robotico. In particolare, un manipolatore robotico Puma 560 dotato di sensori ad ultrasuoni, infrarossi e di un sistema di visione industriale interagisce con un robot mobile che trasporta degli oggetti verso una zona di scarico. Quando il sensore di visione trasmette l'informazione di presenza di oggetti presenti nella zona di scarico, il Puma li preleva depositandoli in un'area temporanea. Il sistema di controllo, utilizzando un algoritmo di pianificazione, determina il percorso ottimale

tra il punto di deposito momentaneo di ciascun oggetto ed una scatola dove esso trova la sua collocazione finale.
Per realizzare questa applicazione, il sistema di controllo del manipolatore deve fornire risposte in tempo reale agli eventi del dominio (arrivo di parti) e contemporaneamente deve utilizzare tecniche di ricerca complesse e con tempi di risposta variabili (calcolo della traiettoria da parte dell'algoritmo di pianificazione).
Il controllo del manipolatore è basato sull'architettura di riferimento a tre livelli ed è stato realizzato attraverso la libreria di moduli di controllo astratti descritta nella tesi.

1.4 Organizzazione della tesi

Dopo aver fornito le motivazioni che hanno guidato il lavoro di tesi, si descrive brevemente il contenuto dei capitoli successivi.
I capitoli seguono una sequenza logica: partendo dai concetti di base e da una prima proposta (capitolo 2), sono stati analizzati gli strumenti a disposizione (capitolo 3) per realizzare una libreria di moduli (capitolo 4) su cui basarsi per la realizzazione dell'architettura di governo di un manipolatore robotico evoluto (capitolo 5). La tesi si conclude con una puntualizzazione dei risultati ottenuti ed una discussione di proposte di continuazione del lavoro (capitolo 6).
Con maggiore dettaglio, i capitoli sono organizzati come segue:

- Il capitolo 2 presenta una descrizione delle caratteristiche dei sistemi di controllo in tempo reale e delle tecniche per la deliberazione. Dopo aver sottolineato la necessità di una cooperazione paritetica tra sistema deliberativo e sistema in tempo reale, viene proposta un'architettura che realizza tale cooperazione integrando le tendenze ed i risultati delle ricerche nel settore.

- Il capitolo 3 presenta il sistema operativo Solaris, lo strumento di base utilizzato nella tesi, descrivendo in dettaglio le estensioni inserite per renderlo conforme allo standard IEEE POSIX per il tempo reale.

- Il capitolo 4 descrive la realizzazione della libreria di moduli. La libreria offre al progettista un livello di astrazione sufficiente a costruire un'architettura di governo senza doversi preoccupare della progettazione degli strumenti necessari all'interfaccia tra i moduli ed alla gestione del tempo.

- Nel capitolo 5 è descritta la realizzazione di un'istanza di architettura di controllo per il governo di un manipolatore robotico in grado di gestire le sorgenti d'incertezza che provengono da un ambiente dinamico e parzialmente non strutturato.

- La tesi si conclude con il capitolo 6, in cui vengono suggeriti possibili sviluppi dell'architettura e discusse ipotesi di estensione della libreria.

- Diverse appendici danno maggiori dettagli sulla realizzazione di alcune parti dell'architettura.

Capitolo 2

Requisiti delle architetture di governo dei sistemi robotici evoluti

Un sistema robotico evoluto deve la sua autonomia all'ampia dotazione sensoriale, alle capacità di reazione e ragionamento, alla destrezza della struttura di attuazione del movimento.
L'esperienza ricavata dagli studi sulle architetture deliberative e reattive ha spinto numerosi ricercatori ad approntare tecniche che consentano un approccio ibrido al problema della progettazione dell'architettura di governo di un sistema robotico.
Benché l'idea di architettura ibrida sia ormai chiara, la sua realizzazione richiede il difficile sviluppo di un sistema che combini *flessibilità* e rispetto di *vincoli temporali*.

2.1 Sistemi in tempo reale

In molteplici applicazioni è necessario progettare un sistema di controllo che sappia reagire agli stimoli che riceve dall'ambiente esterno, rispettando nel contempo precisi vincoli di tempo di risposta, in modo che la velocità di adattamento del sistema corrisponda alla velocità di variazione dell'ambiente.
Questi sistemi, in cui la correttezza di funzionamento non dipende soltanto dalla validitá dei risultati ottenuti ma anche dal tempo in cui i risultati sono prodotti [6], sono detti sistemi in tempo reale. Essi hanno ormai un ruolo

vitale nella nostra societá, comprendendo un largo spettro di applicazioni, dal controllo degli esperimenti di laboratorio al controllo di impianti nucleari, dal controllo dei sistemi di volo aereo alla robotica.
Le tecniche di programmazione dei sistemi in tempo reale sono purtroppo molto costose. Infatti, per la progettazione di moduli in grado di rispettare i limiti di tempo richiesti sono necessarie molte ore di programmazione e molte altre per la fase di collaudo.
Per questo motivo i sistemi in tempo reale trovano spazio in ambienti statici e ben conosciuti. In ambienti dinamici ed incerti, infatti, sono richieste caratteristiche di adattabilità e di ragionamento che sono proprie delle tecniche di intelligenza artificiale ma che non sono previste dalle tecniche in tempo reale. Nelle applicazioni complesse, inoltre, diventa estremamente difficile per il programmatore riuscire a prevedere tutte le possibili situazioni in cui il dominio può trovarsi, ma questa completa copertura è necessaria per garantire il rispetto dei vincoli temporali [7].

2.2 Sistemi di controllo flessibili

Quando un sistema robotico opera in un ambiente dinamico, incerto e non completamente conosciuto è necessario che il controllo sia flessibile per consentire al sistema di adattarsi agli eventi inaspettati. In questo modo il sistema godrà di robustezza e mostrerà un comportamento capace di rispondere a tutte le situazioni in cui si potrà trovare.
Le tecniche tradizionali utilizzate per assicurare flessibilità ad un sistema di controllo sono basate sulla ricerca nello spazio degli stati del problema. All'aumentare delle informazioni sensoriali e delle azioni che il sistema può intraprendere lo spazio degli stati cresce esponenzialmente, pertanto la ricerca dell'azione da intraprendere diventa un problema intrattabile. Ovviamente queste tecniche non sono direttamente utilizzabili nei sistemi robotici autonomi poiché non consentono di introdurre vincoli temporali dettati dall'ambiente in cui operano.

2.3 Sistemi flessibili ed in tempo reale

Per affrontare un problema reale, cioè il controllo di un sistema robotico operante in un ambiente dinamico, complesso e soggetto a vincoli temporali,

è necessario utilizzare tecniche e principi che provengono sia dall'intelligenza artificiale sia dalla programmazione in tempo reale.
Un sistema di controllo costruito sulla base di questi principi dovrebbe:

1. essere attivo continuamente per lunghi periodi di tempo
2. comunicare con l'ambiente attraverso l'uso di sensori ed attuatori
3. occuparsi dell'incertezza e della perdita di dati
4. dedicare maggiori risorse al controllo degli eventi più urgenti o prioritari
5. gestire, in modo prevedibile, eventi sincroni o asincroni garantendo risposte entro limiti temporali dettati dall'ambiente
6. non bloccarsi

Il sistema robotico non deve bloccarsi terminando in uno stato non noto. Esso deve essere in grado di avvertire quando le risorse non sono sufficienti per il compito previsto, provvedendo a modificare i parametri di alcuni componenti, e, quando si verificano degli errori, eseguendo delle azioni di recupero per portarsi in uno stato di sicurezza.
Per realizzare un sistema di governo con queste caratteristiche sono stati proposti tre metodi: racchiudere la componente di intelligenza artificiale nel sistema in tempo reale, racchiudere le reazioni in tempo reale in un sistema di intelligenza artificiale, oppure unire i due sistemi in modo che agiscano come componenti parallele e cooperanti (fig. 2.1).

2.3.1 Intelligenza artificiale racchiusa in un sistema in tempo reale

Il modo più semplice per amalgamare le tecniche del tempo reale (RT) e dell'intelligenza artificiale (AI) consiste nell'associare ad ogni processo di AI un limite di tempo entro cui esso deve deliberare. Pertanto la sfida diventa quella di riuscire a prendere decisioni "intelligenti" entro i termini imposti dall'applicazione.
Sfortunatamente i processi di ricerca del comportamento ottimale sono problemi intrattabili, per cui la richiesta di risorse nel caso peggiore è estremamente più grande rispetto alla media, rendendo considerevole la varianza

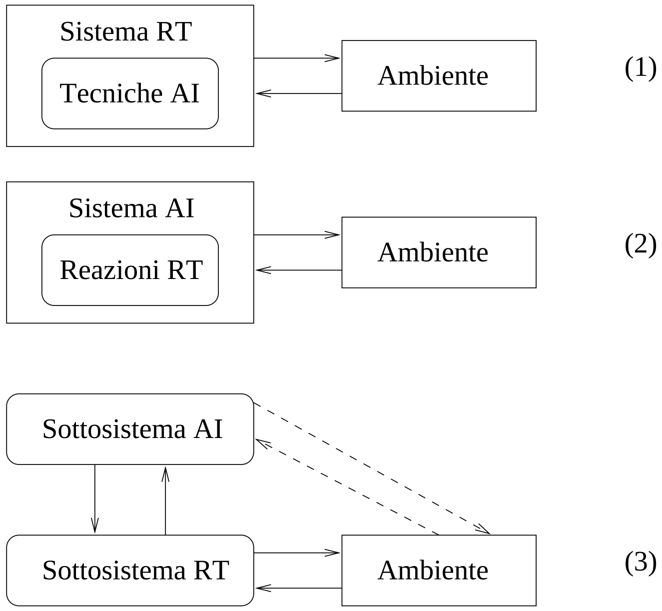


Figura 2.1: Approcci all'integrazione di tecniche di intelligenza artificiale (AI) e di tempo reale (RT)

nei tempi di esecuzione. Questa considerazione, unita alla necessità di poter quantificare con precisione i tempi di esecuzione dei compiti rende indispensabile trovare metodi che avvicinino i due approcci.
Per risolvere il contrasto tra i processi di intelligenza artificiale e gli schedulatori in tempo reale si può agire in due modi: semplificare i processi deduttivi e realizzare metodi di inferenza che riducano la varianza dei sistemi AI [8] o progettare i processi di AI come algoritmi iterativi che ad ogni ciclo rendono disponibili risultati sempre più precisi e che pertanto possono essere interrotti quando risultano esauriti i tempi assegnati fornendo un risultato comunque utile.
Tra gli studi che si sono mossi in questa direzione si possono ricordare:

1. Architetture deliberative come il *Procedural Reasoning System* (PRS) [9]. Ingrand e Georgeff hanno dimostrato che è possibile assicurare un'attenzione ad ogni evento del mondo in un termine definito, se si fissa una certa frequenza al sistema decisionale. Il problema di questo sistema è che garantisce solo l'attenzione, e non una risposta all'evento, perché l'informazione sui tempi di reazione non è prevista dal sistema. Nel PRS, inoltre, non viene stabilita alcuna priorità tra le azioni, per

cui una risposta ad un particolare evento può essere interrotta in modo opinabile dall'arrivo di un altro evento. Pertanto PRS non sceglie l'azione da svolgere compiendo una selezione tra le azioni appropriate per l'ambiente a cui si interfaccia in un determinato momento; al contrario, sceglie come e quando invocare dei piani parziali chiamati "procedural knowledge areas".

2. Architetture AI esclusivamente reattive [10] composte da numerosi moduli che procedono in parallelo senza l'uso di memoria condivisa e che si scambiano informazioni attraverso dei canali di comunicazione. I moduli sono *reattivi* poiché producono una risposta dipendente dai soli stimoli che ricevono senza ricorrere ad un ragionamento sullo stato del mondo. Per perseguire il comportamento desiderato dal progettista esiste una gerarchia tra i diversi componenti. Tuttavia questi sistemi non possiedono tecniche di intelligenza artificiale per scegliere l'insieme di azioni ottimali per una determinata situazione e non prestano attenzione alla quantità di risorse necessarie perché possa essere eseguito efficientemente l'insieme di moduli.

3. Algoritmi "any time" [5] che realizzano metodi di ricerca incrementale e rilasciano un risultato in qualsiasi momento vengano interrotti, senza garantire che il processo di ricerca sia completato o che la risposta superi una soglia minima di qualità. Questa tecnica garantisce il rispetto dei limiti temporali ma la qualità del risultato può essere sacrificata.

2.3.2 Reazioni in tempo reale per un sistema di intelligenza artificiale

Le architetture robotiche tradizionali appartengono a questa impostazione. Ad una fase di pianificazione, in cui a partire da una descrizione simbolica sia del mondo che del goal desiderato si cerca di costruire, sfruttando algoritmi di ricerca tipici dell'AI, una sequenza di azioni per giungere allo scopo, segue una fase di esecuzione controllata, in tempo reale del piano ottenuto.
Tra gli approcci di questo tipo si può ricordare STRIPS/PLANEX [11] in cui per uno stesso piano l'attività di pianificazione ed esecuzione non si sovrappongono. Una volta prodotto un piano, se l'esecutore in tempo reale non riesce a portarlo a termine il pianificatore viene distolto dall'elaborazione del compito successivo e investito del compito di produrre una nuova soluzione.

L'approccio che cerca di inserire reazioni in tempo reale in un sistema di intelligenza artificiale si può utilizzare con efficacia in ambienti dove è usuale impiegare azioni deliberative e si deve ricorrere raramente a comportamenti reattivi con vincoli temporali.
Ad esempio in Hero-Soar [12], modificato per gestire problemi in tempo reale, produzioni particolari in tempo reale evitano la normale fase deliberativa; al posto di essa, l'azione di una produzione in tempo reale viene eseguita appena la produzione é adeguata.
Quando si cerca di collegare e coordinare i comportamenti reattivi con il comportamento deliberativo dell'intero sistema si devono affrontare alcune difficoltà. Infatti, se le azioni reattive saltano il normale meccanismo di deliberazione diventa difficile per il processo deliberativo riconoscere gli effetti dell'azione in tempo reale e si deve quindi procedere ad una fase di messa a punto del sistema.
Il problema centrale è quindi la perdita delle proprietà di tempo reale da parte delle azioni reattive, poiché esse vengono invocate da sistemi di intelligenza artificiale con comportamenti temporali non prevedibili.

2.3.3 Cooperazione tra tempo reale ed intelligenza artificiale

Questo approccio cerca di conservare le capacità dei sistemi in tempo reale e dell'intelligenza artificiale dividendoli in due sottosistemi separati che collaborano per raggiungere uno scopo. I sottosistemi devono essere isolati, perché i meccanismi dell'intelligenza artificiale non devono interferire con il rispetto dei vincoli temporali da parte delle operazioni che stanno avvenendo nel sottosistema in tempo reale, ma è anche necessario che i due sottosistemi comunichino per poter influenzare le scelte l'uno dell'altro.
Essendo una linea di ricerca recente, i sistemi cooperativi proposti si differenziano per la complessità dei processi che regolano i sottosistemi e per le relazioni tra di essi.
Tra le proposte formulate si può ricordare:

- Sistemi, come *DR/Maruti* [13], che estendono l'architettura completamente reattiva. In essi un certo numero di comportamenti reattivi e in tempo reale sono eseguiti in concorrenza ed un processo di intelligenza artificiale a più alto livello modifica i loro parametri perchè il comportamento reattivo riesca a perseguire uno scopo complesso. Questo modello suppone che le risorse siano sufficienti perché le azioni reattive in tempo reale riescano a rispettare i limiti temporali. Questo porta alla definizione di un sistema sovrastimato, necessario quando tutte le azioni devono essere eseguite contemporaneamente ma inefficiente quando è richiesta solo l'esecuzione di alcuni moduli.

- *SSS*. Questa architettura proposta da Connell [14] integra una componente deliberativa ed una reattiva. La componente deliberativa abilita/disabilita i comportamenti della parte reattiva. La componenete deliberativa non è però in grado di generare nuovi comportamenti.

- L'architettura *ATLANTIS* [15] si compone di tre livelli. Il livello più basso fornisce un controllo prettamente reattivo a cui può essere richiesto di attivare od inibire alcuni suoi moduli, di variare parametri o di rendere osservabile ai livelli superiori la propria attività. Il livello più alto mantiene modelli del mondo e produce piani. Nel mezzo, si trova un sequenziatore/monitor in grado di rendere attivi o inattivi l'insieme dei comportamenti reattivi del livello più basso. Questa architettura è incompleta poichè non è previsto che il livello più alto, il pianificatore, possa inserire nel livello più basso, il reattore, nuovi comportamenti.

- $\mathcal{RS}$. Lyons [16] ha proposto la rappresentazione del compito di un manipolatore robotico attraverso l'uso del *Robot Schemas* ($\mathcal{RS}$). Nel modello $\mathcal{RS}$, i compiti del robot sono rappresentati come processi comunicanti e concorrenti. Sulla base dell'$\mathcal{RS}$ è stato sviluppato un ambiente di esecuzione (*reactor*) che contiene una rete di reazioni specificate attraverso l'$\mathcal{RS}$. Il reactor garantisce l'esecuzione delle azioni entro limiti di tempo predicibili. In modo asincrono ed indipendente dal *reactor* un pianificatore *planner* modifica incrementalmente lo schema di comportamento reattivo in corso di esecuzione nel reactor. Il planner determina continuamente se la risposta del *reactor* alle sollecitazioni dell'ambiente è corretta per lo scopo che si vuole perseguire modificando, se necessario, la configurazione del reactor per renderlo più in linea con gli obiettivi [17].

- *CIRCA* [5]. In questa architettura si possono individuare tre sottosistemi. Il sottosistema in tempo reale (RTS) esegue un insieme di azioni

(TAP) di cui si conosce il tempo di esecuzione nel caso peggiore. Finché l'RTS esegue solo l'insieme delle azioni che gli sono state passate dalla componente di intelligenza artificiale viene garantito il rispetto dei vincoli temporali. La parte di intelligenza artificiale (AIS) cerca di trovare un insieme di TAP che permettano di avvicinarsi al conseguimento dello scopo del sistema robotico. In cooperazione con il sottosistema di intelligenza artificiale uno schedulatore verifica che l'insieme di TAP proposte dalla componente di intelligenza artificiale rispetti i vincoli temporali. Poiché l'AIS e l'RTS operano in modo asincrono, l'AIS non deve rispondere ai rigidi vincoli temporali della RTS, quindi può utilizzare metodi euristici con tempi di risposta non predicibili e con grande varianza senza compromettere le prestazioni del sistema.

2.4 Proposta per una architettura flessibile ed in tempo reale

2.4.1 Requisiti

Per i motivi esposti nei paragrafi precedenti, affinché il sistema robotico sia autosufficiente in un ambiente reale, è necessario progettare una architettura che consenta la collaborazione tra la componente in tempo reale e la componente di intelligenza artificiale.
Pertanto si deve realizzare un sottosistema software che abbia non solo la possibilitá di modificare i parametri dei processi in tempo reale ma che sia anche in grado di riprogrammare dinamicamente l'intero sottosistema scegliendo quali azioni compiere e quali parametri assegnar loro affinché il sistema possa raggiungere lo scopo finale.
Per poter realizzare queste premesse è necessario avere a disposizione un sottosistema in tempo reale che sappia adattarsi alle richieste provenienti dal livello di intelligenza artificiale, fornendo nel contempo delle procedure che consentano al sottosistema di AI di modificare i parametri di esecuzione. Il livello di AI deve inoltre essere sensibile al carico a cui é sottoposto il sistema hardware per modificare i parametri delle azioni in tempo reale quando le risorse disponibili non sono più sufficienti a garantire la sicurezza del comportamento globale del sistema.

Un'architettura conforme a questi requisiti consente di attribuire al sistema i seguenti vantaggi:

1. il sottosistema in tempo reale non deve essere riprogrammato quando si cambia l'ambiente e lo scopo dell'architettura;

2. il sottosistema in tempo reale può essere basato su un sistema operativo semplice, in grado di gestire processi che condividano la risorsa CPU ed eventualmente operanti a differenti frequenze;

3. la componente di intelligenza artificiale può riprogrammare il sottosistema in tempo reale per evitare sovraccarichi delle risorse o per disporre di maggior tempo per prendere una decisione quando il sistema è fermo in attesa di un nuovo piano;

4. la componente di intelligenza artificiale conserva le caratteristiche di scelta "intelligente" dell'azione da svolgere e di flessibilitá.

2.4.2 Descrizione dell'architettura

In fig. 2.2 è mostrata l'architettura che è stata progettata sulla base dei requisiti presentati nel paragrafo precedente. Tale architettura tiene in conto delle tendenze e dei risultati delle ricerche nel settore, collocandosi tra quelle in cui si ha una cooperazione tra attività deliberative e di tempo reale. Non dissimilmente da altre proposte [5] [4] [18], l'architettura si compone di tre livelli: *livello di pianificazione*, *livello di controllo* e *livello di esecuzione*. Le interazioni tra i livelli e l'attribuzione dei ruoli presentano tuttavia alcuni elementi di originalità.

A *livello di pianificazione* un pianificatore decompone il problema in un insieme di sottoproblemi, cercando quindi di individuare per ognuno di essi l'insieme di processi necessari per raggiungere lo scopo. Poiché il sistema in tempo reale deve essere garantito nel rispetto dei vincoli temporali, viene verificato da un *controllore di schedulabilità* che le risorse computazionali del sistema robotico siano sufficienti. Dalla collaborazione tra pianificatore e controllore di schedulabilità al livello di pianificazione produce un insieme di moduli di cui è garantita l'esecuzione nei vincoli temporali e altri moduli non critici sotto il profilo del tempo reale, o comunque eseguiti in modalità "best-effort".

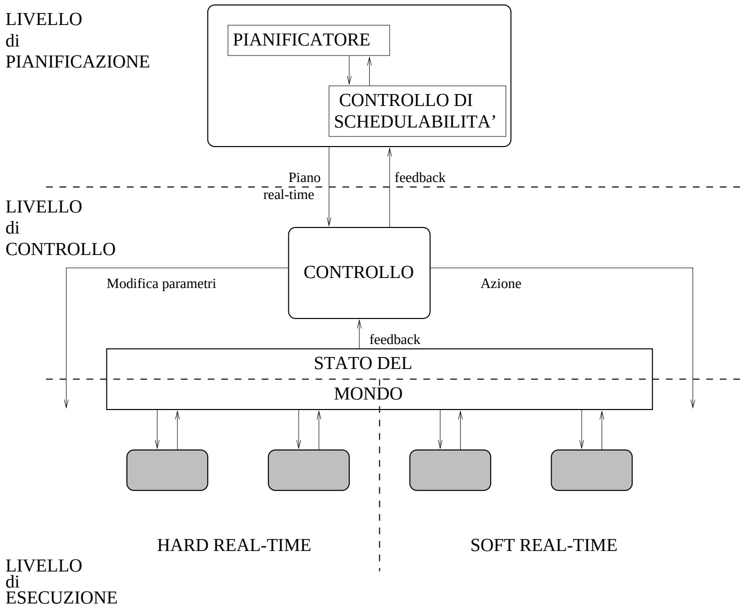


Figura 2.2: Diagramma dell'architettura

Il *livello di controllo* sorveglia l'esecuzione dei moduli che, collaborando, portano al conseguimento dello scopo. Esso deve essere in grado di rilevare stati del sistema robotico controllato che richiedono la modifica dei parametri dei moduli. Questo può voler dire, non solo diminuire la frequenza di esecuzione per evitare il sovraccarico del sistema ma anche, rallentare le attività del livello esecutivo per consentire alla parte di intelligenza artificiale di avere a disposizione maggiori risorse computazionali. Queste consentono al livello deliberativo di trovare un nuovo insieme di processi la cui esecuzione consentià al sistema robotico di procedere, con maggiore efficienza, al conseguimento dello scopo finale. Poiché il livello di controllo seleziona parametri di procedure predefinite, il tempo massimo di risposta può essere determinato.
Il *livello di esecuzione* concretizza l'insieme dei moduli che sono stati indicati dal livello di pianificazione. L'esecutore deve essere in grado di controllare completamente l'esecuzione dei moduli verificando che essi rispettino i propri vincoli temporali. L'autonomia del livello di esecuzione attenua le richieste sui tempi di risposta del livello di pianificazione che può continuare a progettare il nuovo sistema di moduli per la parte esecutiva.
Poiché l'ambiente in cui il sistema robotico opera è dinamico e complesso, può succedeere che il verificarsi di eventi inaspettati porti il sistema in una

situazione non prevista. Per questo motivo è necessario che insieme al flusso di informazioni verso il basso per l'istruzione del livello di esecuzione sia presente un flusso di retroazione che confermi il buon funzionamento del sistema.
Il livello di controllo conosce lo stato del mondo che viene continuamente aggiornato dai moduli che gestiscono i sensori, quindi è in grado di decidere quando far compiere al livello esecutivo azioni la cui attuazione è richiesta solo sporadicamente come conseguenza di eventi predefiniti.
Il livello di controllo deve avere inoltre accesso alle informazioni sui comportamenti temporali dei moduli per rendersi conto se il sistema è sovraccarico e di consequenza, diminuendo la frequenza dei moduli, diminuire le richieste di risorse computazionali.
Rispetto alle architetture in cui una delle due componenti viene incapsulata nell'altra, il sistema reattivo, cioè il livello esecutivo, non è definito nel momento in cui il sistema è progettato ma varia dinamicamente e la parte di intelligenza artificiale (livello di pianificazione) può essere bloccata al verificarsi di particolari condizioni.

2.4.3 Condizioni di sicurezza per il livello di esecuzione

Mentre il livello esecutivo esegue il programma mantenendo il mondo dell'applicazione in un insieme conosciuto di stati, il livello di pianificazione deve poter prepare il nuovo programma senza vincoli di tempo e senza dover preoccuparsi della sicurezza della componente in tempo reale. Questo è possibile se la parte in tempo reale riesce a controllare il sistema mantenendolo sempre all'interno di un insieme di stati per un periodo di tempo indefinito.
Mutuando l'idea dalla teoria dei sistemi di controllo si definisce questa proprietà *stabilità ad anello chiuso*: un sistema che gestisce un insieme di eventi discreti è stabile ad anello chiuso nei confronti di un insieme di stati finali, se il controllore è in grado di guidare il sistema controllato in modo che esso raggiunga uno stato finale in un tempo finito e, quindi, resti in un insieme di stati finali per un tempo indefinito.
Un'altra condizione per assicurare la sicurezza é l'intersezione tra gli stati finali del programma che é in esecuzione sulla parte in tempo reale e gli stati del programma che il sottosistema di AI sta preparando (fig. 2.3). In questo caso, i due insieme X e Y condividono lo stato S. La parte in tempo reale ha inizialmente un programma che mantiene il mondo in uno degli stati dell'in-

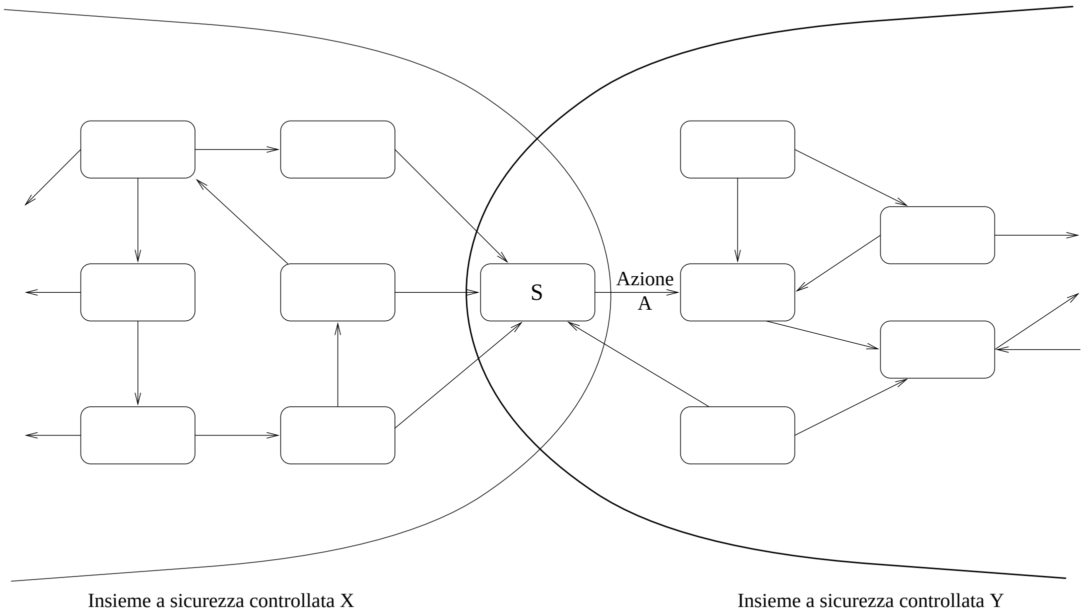


Figura 2.3: Il sovrapporsi di uno stato richiesto per il trasferimento da un insieme dalla sicurezza garantita ad un altro

sieme X. Quando un nuovo piano è disponibile per l'insieme Y e il mondo è nello stato S, allora la componente in tempo reale può trasferire il controllo al nuovo programma, che eseguirà l'azione A per progredire nel nuovo insieme di stati.
Non ci può essere una transizione da uno stato condiviso ad uno stato non condiviso, perché occorre garantire che il sistema, quando riceve un nuovo programma, si trovi in una situazione in grado di gestirlo.

2.4.4 Benefici e problemi

Schoppers [18] individua tre concetti che spingono la ricerca di una architettura in cui AI e tempo reale collaborano parallelamente: *sicurezza*, *generalità* ed *economia*.

- *Sicurezza* perché i vincoli temporali dell'applicazione vengono completamente rispettati e perché, grazie alla retroazione dell'informazione sul carico computazionale dalla parte in tempo reale alla parte di intelligenza artificiale è possibile modificare i parametri dei diversi compiti per evitare un blocco del sistema.

- *Generalità* perché il sistema contiene una componente di pianificazione che può modificare dinamicamente il programma in tempo reale e controllare l'uso della potenza di calcolo.

- *Economicitá* perché la risorse disponibili sono condivise e distribuite e quindi l'architettura può scalare i parametri dei diversi compiti per renderli consoni alle possibilità della parte hardware.

Se sviluppati, questi concetti portano a un sistema che gode dei seguenti benefici:

- Sincronizzazione spaziale. L'abilità di modificare dinamicamente la frequenza di esecuzione di un processo in tempo reale consente di sincronizzare la elaborazione con il moto nello spazio. Ad esempio, per un robot che si muove in un ambiente non noto, a frequenza costante, riducendo la velocità diminuisce lo spazio percorso tra due letture; oppure si può diminuire la frequenza, riducendo così il carico di elaborazione a parità di spazio tra due letture. Se il controllo di collisione è compiuto ad intervalli regolari, minore è la velocità minore saranno i tempi richiesti per la computazione. Se un sistema in tempo reale è programmato perché conosca questa sua proprietà può ridurre la velocità per diminuire la richiesta di potenza di calcolo da parte dei componenti di controllo del moto.

- Degradazione graduale. L'abilità di scalare le operazione per assicurare i vincoli temporali elimina il bisogno di progettare un immaginario caso peggiore, perché non c'è più una netta separazione tra ciò che il sistema è in grado di reggere e ciò che invece lo fa cadere in uno stato impredicibile.

- Riconfigurabilità. Un sistema può disporre di diversi modi per raggiungere lo stesso risultato. Quando il sensore utilizzato normalmente non funziona, un altro potrà essere usato al suo posto. Poiché i tempi di elaborazione richiesti per tali sensori sono diversi, è necessario prevedere una riconfigurazione del sistema perché sia considerata la variazione nella potenza computazionale richiesta.

2.5 Requisiti per il sottosistema in tempo reale

L'architettura di controllo di un sistema robotico autonomo presentata nei paragrafi precedenti richiede che il livello di esecuzione sia costituito da una componente in tempo reale che presenti le seguenti caratteristiche:

1. capacità di determinare il tempo di CPU utilizzato da ogni trama di controllo;
2. capacità di verificare se l'insieme dei processi che verranno eseguiti su un determinato sistema riuscirà a rispettare tutti i vincoli temporali;
3. capacità di modificare i parametri temporali dei diversi processi, continuando a rispettare tutti i vincoli prima, dopo e durante le modifiche;
4. capacità di gestire sia processi periodici che aperiodici.

Per rispettare questi requisiti è necessario realizzare il sottosistema in tempo reale con sistemi operativi real-time.
Se, infatti, si volesse realizzare il livello di esecuzione in ambiente Unix BSD, non sarebbe possibile garantire le frequenze di processo e l'utilizzo di tempo del CPU a causa dell'elevato numero di processi concorrenti e della politica di schedulazione di Unix. Anche attribuendo la più alta priorità al sottosistema in tempo reale, questo rimarrebbe sempre a livello utente, quindi processi di I/O, da disco o da rete, continuerebbero ad avere precedenza sullo stesso.
Nei sistemi operativi in tempo reale si cerca, invece, di garantire la predicibilità. Se vengono fornite informazioni sufficienti su ogni processo e sullo stato del mondo, un sistema predicibile tenta di trovare una schedulazione in cui i processi possano rispettare i limiti temporali imposti. Le informazioni necessarie al sistema per ogni processo sono il tempo di esecuzione nel peggiore dei casi, il periodo di esecuzione, il tempo di partenza dall'istante di attivazione, la priorità e la deadline. Un sistema predicibile utilizza queste informazioni per tentare di conciliare la competizione nella domanda di risorse da parte dei diversi processi.
Diverse sono le proposte di sistemi operativi real-time. Con riferimento al dominio applicativo, essi possono essere classificati secondo il loro grado di specificità per le applicazioni robotiche in:

- sistemi operativi *real-time* dedicati ad applicazioni robotiche di ricerca (es. HARTIK e CHIMERA)

- sistemi operativi *real-time general-purpose* dedicati ad applicazioni industriali (es. VxWorks)

- sistemi operativi *general-purpose* che incorporano anche funzionalità in tempo reale (es. Solaris 2.x).

Molti dei sistemi operativi in tempo reale tra cui iRMX II, OS-9, pSOS+ offrono i propri ambienti di sviluppo. Questi ambienti, che contengono editor, compilatori, file system e strumenti grafici, sono generalmente inferiori agli strumenti offerti dai sistemi operativi UNIX. Purtroppo fino a qualche anno fa, nessuno dei sistemi operativi della famiglia UNIX offriva supporti per funzionalità real-time.
La recente introduzione in Solaris 2.x di tecniche per aderire allo standard POSIX P1003.4 ha reso disponibile un sistema operativo a basso costo, con un uso diffuso negli ambienti di ricerca e con caratteristiche sia general-purpose sia real-time.
Le principali caratteristiche di interesse di Solaris rilevanti per il lavoro di questa tesi sono riassunte nel capitolo successivo.

2.6 Requisiti per il sottosistema deliberativo

La progettazione del livello di pianificazione deve essere guidata dai requisiti a cui si deve attenere un sistema deliberativo:

1. capacità di rappresentare le informazioni e porle in relazione tra loro per costruire un modello simbolico del mondo;

2. capacità di scegliere quali informazioni del mondo reale rilevare perché la descrizione sia sufficientemente accurata per prendere decisioni adeguate;

3. capacità di suddividere il compito da perseguire in sottoproblemi;

4. capacità di gestire tutte le operazioni dell'architettura di controllo, per prendere decisioni che coinvolgono non solo il livello reattivo ma anche il livello deliberativo;

5. capacità di riceve informazioni in retroazione dai livelli sottostanti, per modificare la pianificazione in corso se le informazioni ricevute lo richiedono.

Capitolo 3

Il sistema operativo Solaris

Solaris 2 è una valida alternativa per realizzare la componente in tempo reale, infatti è completamente SVR4, inoltre è conforme a numerosi requisiti dello standard Posix P1003.4: è preemptive, consente il caricamento dinamico dei moduli del kernel, come le chiamate di sistema, i driver per i dispositivi del sistema, infine supporta le classi di priorità real-time e rende disponibili thread e processi leggeri.

3.1 Processi e thread

Nella maggior parte dei sistemi operativi tradizionali, ciascun processo dispone di uno spazio di indirizzamento ed un singolo flusso di controllo. In alcuni problemi si avverte però la necessità di avere thread (fili, trame) di controllo multiple che condividano lo stesso spazio di indirizzamento, ma che eseguano codici sorgenti diversi, come se fossero processi separati. Nei sistemi tradizionali (fig. 3.1 (a)) su una macchina possono essere eseguiti diversi processi, ogni processo possiede un proprio contatore di programma, un proprio stack, un proprio insieme di registri ed un proprio spazio di indirizzamento. Essi sono completamente indipendenti e possono comunicare tra di loro solo attraverso le primitive di comunicazione pipe, o segnali per il modello ad ambiente locale.
Nei sistemi multithreading (fig. 3.1 (b)) è stata realizzata l'astrazione delle thread. Come i processi, ciascuna thread opera in modo strettamente sequenziale ed ha un proprio contatore di programma ed uno stack per mantenere traccia dell'attuale stato di esecuzione. Le thread, come fanno i proessi,

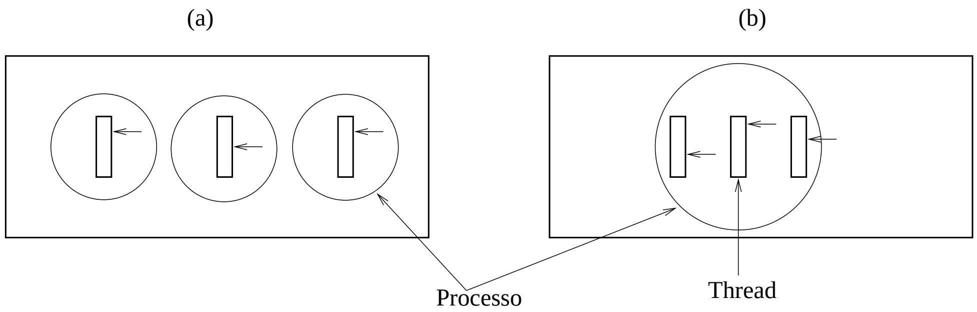


Figura 3.1: Confronto tra struttura tradizionale a processi (a) e struttura multithread (b)

condividono la stessa CPU in multiprogrammazione, possono creare thread figli, si possono bloccare in attesa che venga completata una chiamata, così da permettere ad una thread sospesa di essere riattivata. In un sistema multithread ad ogni processo, in generale, possono essere associate una o più thread (trame di esecuzione). Tuttavia, le thread di uno stesso processo non sono così indipendenti come lo sono processi diversi, infatti queste condividono lo stesso spazio di indirizzamento e quindi ciascuna di esse può accedere ad un qualunque indirizzo virtuale. Meccanismi di protezione tra thread appartenenti allo stesso processo non esistono ma non dovrebbero essere necessari, poiché il processo appartiene ad un solo utente e quindi le thread dovrebbe essere create per collaborare.
Oltre a condividere lo spazio di indirizzamento, le thread appartenenti allo stesso processo condividono lo stesso insieme di file aperti, di processi figli, di timer, di segnali (fig. 3.2).

3.1.1 Stati delle thread

Le thread possono trovarsi in diversi stati (fig. 3.3):

1. *in esecuzione*: la thread ha il controllo della CPU;
2. *pronta*: la thread è stata schedulata per l'esecuzione e verrà eseguita appena sarà il suo turno;
3. *bloccata*: la thread sta aspettando su una variabile di sincronizzazione;

Per ogni thread	Per ogni processo
Program counter	Spazio degli indirizzi
Stack	Variabili globali
Insieme dei registri	File aperti
Thread figli	Processi figli
Stato	Timer
	Segnali
	Semafori
	Informazioni di accredito

Figura 3.2: Dati propri delle thread e dei processi

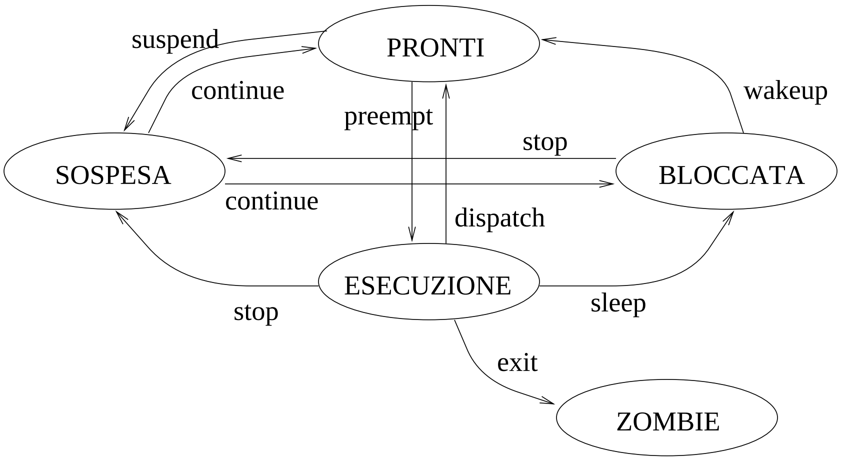


Figura 3.3: Transizione tra gli stati per le thread

4. *sospesa*: la thread è sospesa (in Solaris questo avviene con la chiamata alla funzione *thr_suspend()*) e rimarrà in questo stato fino a quando un'altra thread la risveglierà;

5. *zombie*: una thread ha concluso l'esecuzione del suo codice e sta aspettando che le sue risorse vengano liberate.

3.1.2 Thread utente e thread kernel

In generale sono possibili due tecniche per gestire le thread: all'interno dello spazio di indirizzamento dell'utente o nel kernel.

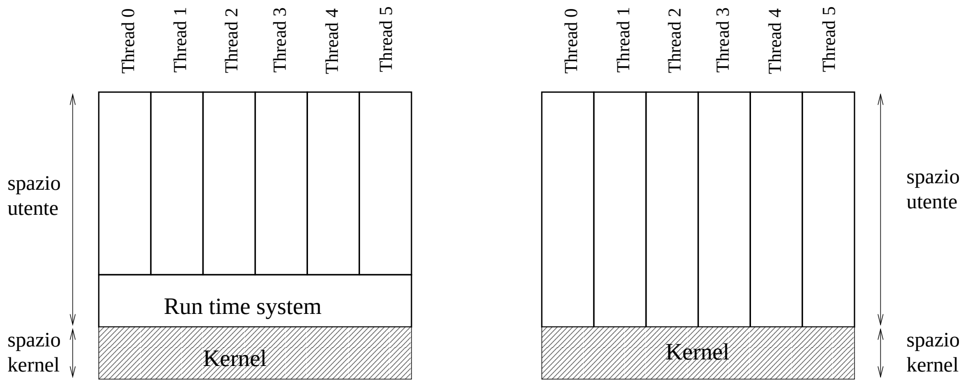


Figura 3.4: (a) Thread utente (b) Thread kernel

Thread utente
In questo caso l'insieme delle funzioni è interamente nello spazio di indirizzamente dell'utente, e pertanto il kernel continua a gestire processi a trama singola.
Il vantaggio di questa scelta è che una libreria di thread utente può essere realizzata anche su sistemi operativi che non supportano le thread come l'UNIX tradizionale. La struttura dei sistemi con thread a livello utente è rappresentata in fig. 3.4 (a). La gestione delle thread avviene tramite un run time system, cioé un insieme di procedure.
Quando una thread esegue una chiamata di sistema essa in realtà invoca una procedura del supporto, che controlla se la thread deve essere sospesa. In questo caso, la procedura memorizza i registri della thread in una tabella, cerca una thread non bloccata che può essere messa in esecuzione e carica i registri di macchina con i valori salvati della nuova thread. Non appena lo stack pointer o il program counter sono stati scambiati, la nuova thread entra in esecuzione.
Nel caso in cui l'architettura hardware metta a disposizione un'istruzione per la memorizzazione e il caricamento dei registri, lo scambio di contesto fra le thread avviene con poche istruzioni. Poiché queste istruzioni avvengono a livello utente, lo scambio tra thread è molto più veloce dell'esecuzione di una chiamata al kernel.

Le thread utente realizzano il modello "Many to One" (fig. 3.5 (a)), in cui molte thread possono esistere in un processo, ma solo una può essere eseguita in ogni momento.

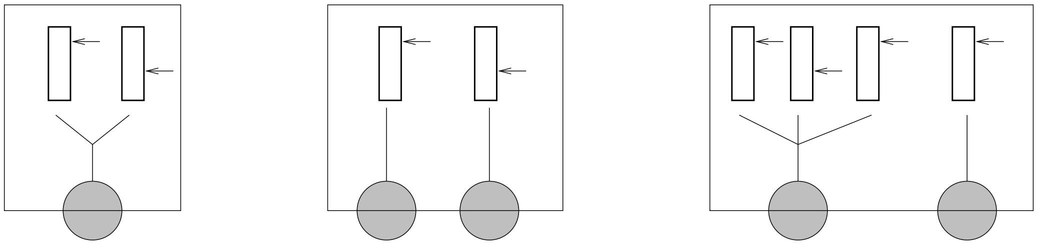


Figura 3.5: (a) Many to One (b) One to One (c) Many to Many

Thread kernel
Se il sistema operativo gestisce le thread non occorre nessun run time system (fig. 3.4 (b)). Per ogni processo, il kernel ha una tabella con un elemento per ciascuna thread, che contiene i registri, lo stato, la priorità ed altre informazioni; le informazioni sono le stesse memorizzate per le thread a livello utente, ma in questo caso si trovano all'interno del nucleo.
Tutte le chiamate che potrebbero bloccare una thread vengono realizzate come chiamate di sistema, con un costo considerevolmente più alto rispetto ad una chiamata al supporto run time. Quando una thread si blocca, il nucleo può mandare in esecuzione, a sua discrezione, una thread dello stesso processo oppure una thread di un processo diverso.
Ci sono due forme di thread a livello kernel e quindi due modelli: one to one e many to many
Nel modello "One to One" (fig. 3.5 (b)), c'è una thread kernel per ogni thread utente del sistema. In questo modello, le thread e non i processi sono le entità schedulabili ed ogni thread all'interno di un processo può essere eseguita su processori differenti. Quindi questo modello consente l'elaborazione parallela.
Nel modello "Many to Many" ((fig. 3.5 (c)), un numero qualsiasi di thread utente sono gestite da alcune (in numero minore o uguale) thread kernel.

3.1.3 I vantaggi delle thread

L'introduzione delle thread porta a una maggiore efficienza ed una maggiore semplicità nella codifica per programmi che possono essere facilmente espressi con l'uso di numerose trame di controllo che compiono diverse azioni.
A parte la possibilità di esecuzioni parallele, le thread utente e le thread

kernel offrono diversi vantaggi rilevanti per le applicazioni di controllo dei robot.

Risorse condivise Le thread condividono molte delle risorse del processo: file, memoria e lo spazio degli indirizzi virtuali. L'uso delle thread permette pertanto al programmatore di conservare delle risorse di sistema. Inoltre, le prestazioni del sistema migliorano perchè l'uso di risorse condivise è meno costoso, in tempo ed in spazio, rispetto alla condivisione di risorse tra processi.

Velocità delle operazioni sulle thread Le operazioni di creazione, distruzione e cambio di contesto sulle thread sono generalmente più veloci rispetto a quella tra processi.

Operazione	*microsecondi*	*Rapporto*
Create unbound thread	52	-
Create bound thread	350	6.7
fork()	1700	32.7

Tabella 3.1: Tempi per la creazione di thread in Solaris 2.x

Struttura del programma In alcune applicazioni, il programmatore ha bisogno di salti incondizionati o altri metodi di questo tipo per aggirare le limitazioni della programmazione sequenziale. Le thread offrono invece la possibilità di utilizzare gli approcci paralleli più vicini al problema per applicazioni intrinsecamente concorrenti.

3.2 Processi, LWP e thread in Solaris 2.x

3.2.1 Processi

Il sistema operativo Solaris presenta una struttura per il processo diversa dalla struttura dei sistemi UNIX tradizionali (fig. 3.7).
All'interno della struttura di processo è presente un nuovo campo, chiamato *lightweight process* (LWP), che sostituisce le statistiche di esecuzione, lo stato della CPU e il kernel stack. Gli LWP che appartengono a un processo condividono pertanto informazioni della struttura processo.

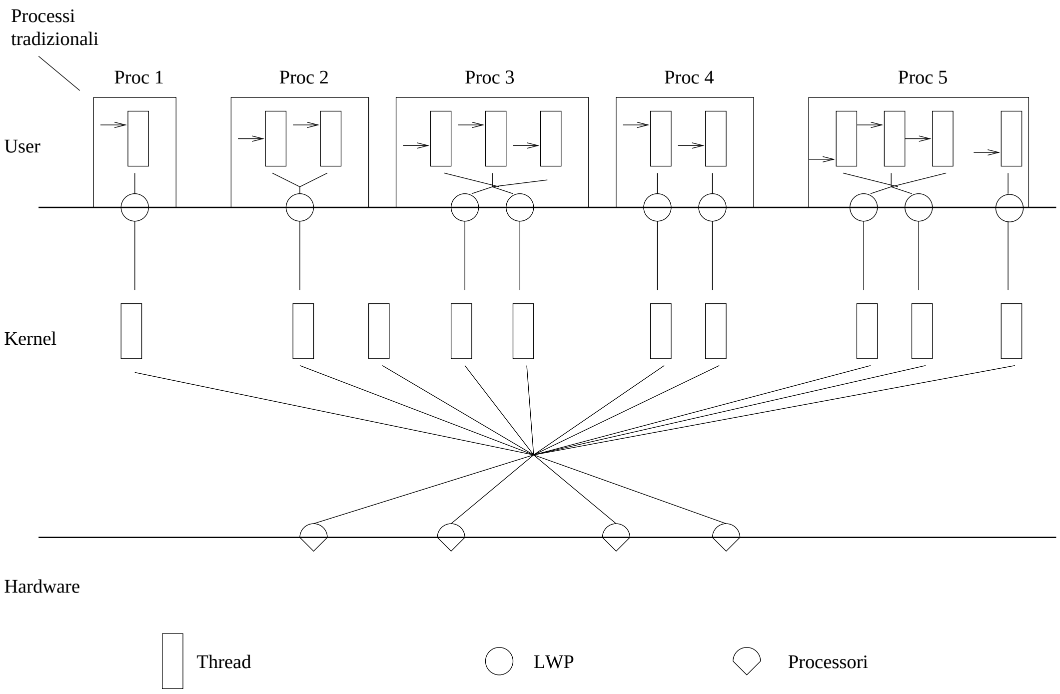


Figura 3.6: Architettura del sistema multithread Solaris

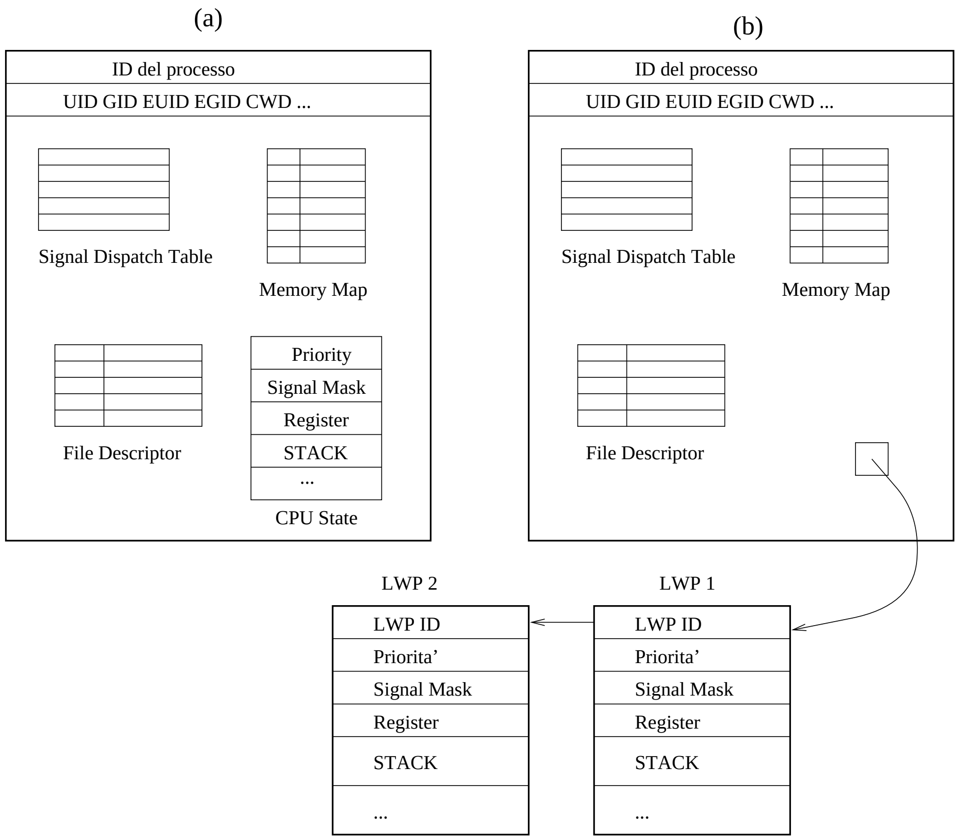


Figura 3.7: Struttura dei processi negli UNIX tradizionali (a) e in Solaris 2 (b)

La struttura del processo in Solaris 2.x si trova nello spazio kernel, e quindi il codice dell'utente vi può accedere solo attraverso l'uso di chiamate di sistema.

3.2.2 Thread

In Solaris 2.x le thread sono realizzate in modo da condividere larga parte della struttura del processo di cui fanno parte. Le informazioni che sono proprie di ogni thread sono:

- identificatore di thread
- stato dei registri (includendo anche il program counter e il puntatore dello stack)
- stack

- maschera dei segnali
- priorità
- zona di memoria privata della thread ([19] pag. 5)

Poiché le thread condividono le istruzioni del processo e molti dei suoi dati, un cambiamento nei dati condivisi può essere visto da tutte le thread del processo. Pertanto, quando una thread deve interagire con le altre thread del processo può farlo senza dover coinvolgere il sistema operativo.
Se non viene specificato diversamente (par. 3.2.5), le thread Solaris sono a livello utente, e quindi sono utilizzate nello spazio utente evitando i costi di un cambio di contesto a livello kernel. Un'applicazione può quindi essere organizzata attraverso centinaia di thread e continuare a non consumare molte risorse di kernel.
La libreria Solaris offre le seguenti tecniche di sincronizzazione, che essendo a livello utente, sono più veloci delle corrispondenti operazioni di sincronizzazione tra processi:

- mutua esclusione (mutex)
- semafori
- variabili condizione
- lock lettori-scrittore.

La tabella 3.2 riporta i tempi di sincronizzazione tra due thread utilizzando due operazioni p e v sul semaforo. Si può osservare che la sincronizzazione tra thread unbound, avvenendo a livello utente è 3 volte più veloce rispetto alla sincronizzazione tra processi. Per sincronizzazione tra thread bound è richiesto un tempo circa sei volte superiore alle thread unbound perché vengono coinvolti meccanismi a livello kernel.

3.2.3 Processi Lightweight

Un processo lightweight rappresenta una CPU virtuale disponibile per eseguire del codice utente. Ogni LWP è gestito dal kernel in modo indipendente, e quindi governa le proprie chiamate di sistema, può incorrere in page fault indipendenti e può essere eseguito in parallelo su un sistema multiprocessore.
Gli LWP sono un ponte tra il livello utente ed il livello kernel. Ogni processo

Operazione	*Microsecondi*	*Ratio*
Unbound thread	66	-
Bound thread	390	5.9
Tra processi	200	3

Tabella 3.2: Tempi per la sincronizzazione utilizzando due operazioni p e v (su una SPARCstation 2 Sun 4/75 utilizzando il timer in microsecondi)

contiene uno o più LWP, ognuno dei quali esegue una o più thread a livello utente. La libreria delle thread a livello utente, con l'aiuto di alcune direttive da parte del programmatore e del sistema operativo, assicura che il numero di LWP disponibili sia adeguato per il numero di thread nel sistema.
Il sistema operativo non ha conoscenza sulle thread utente che sono attive in ogni processo. La schedulazione da parte del kernel avviene perciò basandosi sulla classe di appartenenza e sulla priorità degli LWP(par. 3.4).
Gli LWP offrono un livello di parallelismo e concorrenza a livello kernel che supporta l'interfaccia delle thread. Il programmatore dovrebbe tuttavia evitare di usare direttamente l'interfaccia degli LWP poichè, al contrario dell'interfaccia standard delle thread, essa dipende dalla realizzazione, e quindi l'uso degli LWP compromette la portabilità del sistema.

3.2.4 Thread unbound

Le thread che condividono l'insieme degli LWP sono chiamate thread unbound. Generalmente le thread sono di questo tipo e possono quindi utilizzare qualsiasi LWP.
La libreria delle thread genera il numero necessario di LWP ed assegna ad essi alcune delle thread pronte per l'esecuzione. L'LWP prende lo stato della thread ed esegue le sue istruzioni. Se la thread si blocca su un meccanismo di sincronizzazione, o se un'altra thread deve essere eseguita, lo stato della thread viene salvato nella memoria del processo e la libreria assegna un'altra thread all'LWP.

3.2.5 Thread bound

Una thread che è assegnata permanentemente ad un LWP è chiamata thread bound. Questa scelta viene compiuta per:

- far sì che la thread sia schedulata globalmente (par. 3.4)
- dare alla thread uno stack alternativo per i segnali
- dare alla thread un proprio timer (par. 3.5.2)

Spesso è necessario utilizzare nell'applicazione sia thread bound che unbound. Ciò accade, ad esempio, in una applicazione in tempo reale in cui alcune thread devono avere una priorità di sistema e una schedulazione in tempo reale, mentre altre thread devono essere eseguite solo negli intervalli di tempo non utilizzati dalle thread precedenti.

3.3 Scheduling

Esistono due metodi di base per la schedulazione delle thread: schedulazione *locale* al processo e schedulazione *globale* del sistema.
La schedulazione *locale* significa che tutti i meccanismi di schedulazione per le thread sono locali al processo e la libreria delle thread ha il pieno controllo su quali thread saranno schedulate sull'LWP. Questo significa che in queste operazioni non sono coinvolte chiamate di sistema.
La schedulazione *globale* è compiuta al livello kernel e possono quindi essere realizzate delle operazioni di time-sharing tra gli LWP. In Solaris, lo scheduling globale è realizzato associando una thread permanentemente ad un LWP. Questo LWP (e quindi la thread ad esso collegata) è schedulata dal kernel seguendo i normali algoritmi di scheduling.
Lo scheduling locale è compiuto dalla librearia delle thread. Essa sceglie quali thread unbound devono essere assegnate ai diversi LWP disponibili. La schedulazione degli LWP è ancora globale ed indipendente dallo scheduling locale.
A livello di scheduling locale non c'è il concetto di time-slicing come nei sistemi operativi che operano in multiprogrammazione. Una volta che una thread inizia la sua esecuzione su un LWP esso lo rilascerà solo per:

- *sincronizzazione*
- *sospensione* con l'operazione *thr_suspend()* in attesa dell'arrivo di un *thr_continue()* proveniente da un'altra thread

- *preemption*, quando una thread in esecuzione modifica lo stato determinando la necessità di eseguire una thread a più alta priorità. In questo caso la thread in esecuzione viene revocata e la seconda thread prende il suo posto nell'LWP.

- *yielding*, se il programmatore pone un esplicita chiamata a *thr_yield* nel codice. In questo caso lo schedulatore controlla se c'è una thread con la stessa priorità pronta per l'esecuzione. Se essa esiste, la nuova thread prende il controllo dell'LWP, altrimenti la thread continua la propria esecuzione.

3.4 Classi di priorità in Solaris 2.x

Il kernel di Solaris è dotato di tre classi di scheduling: *time-sharing*, *sys*, *realtime*.
La politica di scheduling della classe *time-sharing* distribuisce le risorse di CPU tra gli LWP, cercando di fornire una buona risposta in termini di tempo per gli LWP e un buon throughput per la CPU.
Le thread che appartengono alla classe time-sharing sono schedulate dinamicamente con un intervallo di tempo di alcune decine di millisecondi. Lo schedulatore time-sharing cambia velocemente il contesto con una tecnica round-robin per assicurare ad ogni thread l'opportunità di utilizzare il tempo di CPU.
Gli LWP che appartengono alla classe time-sharing possiedono un attributo di priorità. Lo scopo di questo valore è di fornire un qualche grado di controllo sulla schedulazione degli LWP nella classe time-sharing. Non è comunque garantito che un LWP con una priorità più alta verrà eseguito prima di un LWP a più bassa priorità. Il motivo di questo comportamento sta nel fatto che anche altri fattori determinano la politica di scheduling degli LWP time-sharing; infatti il sistema può dinamicamente cambiare la priorità degli LWP sulla base del tempo di CPU già utilizzato e dell'ultimo uso della CPU.
La classe *sys* consente di schedulare l'esecuzione di LWP di sistema. Le thread nella classe sys hanno una priorità fissata stabilita dal kernel quando vengono create.
La classe *real-time* fornisce una politica di scheduling preemptive con priorità statiche da utilizzare per gli LWP che richiedono una risposta più veloce e deterministica ed un controllo assoluto sulle priorità per la schedulazione.

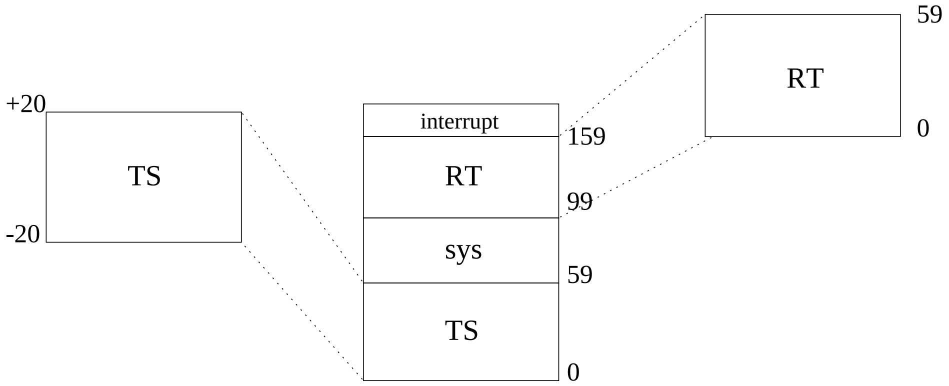


Figura 3.8: Classi di priorità in Solaris

Gli LWP di questa classe hanno diritto al tempo di CPU prima di LWP appartenenti ad ogni altra classe.
Il valore di priorità associato agli LWP della classe real-time determina la priorità dell'LWP rispetto agli altri della classe. Infatti l'LWP con il più alto valore di priorità manterrà il controllo della CPU.
Oltre al valore di priorità è possibile specificare un controllo sull'intervallo di tempo concesso ad ogni LWP nella classe. L'intervallo specifica il valore massimo di tempo che un LWP può avere a disposizione assumendo che entro questo tempo non termini o si sospenda o venga interrotto da un LWP a più alta priorità.

3.5 Strumenti utilizzati

3.5.1 Tempo ad alta risoluzione

Solaris 2.x offre, oltre alle chiamate di sistema *gettimeofday()* e *settimeofday()*, che presentano una risoluzione in secondi, due nuove API *gethrtime()* e *gethrvtime()* che hanno una risoluzione in nanosecondi. La chiamata di sistema *gethrtime()* restituisce il valore in alta precisione del tempo reale. Il tempo è espresso in nanosecondi partendo da un valore arbitrario nel passato che non è in nessuna relazione con il tempo del giorno e che quindi non può essere modificato con l'uso di chiamate di sistema quali *settimeofday()*.
La chiamata *gethrvtime()* restituisce il valore in alta precisione del tempo virtuale di LWP.
Prima di poter utilizzare la chiamata *gethrvtime()* è necessario abilitare il

conteggio temporale dei microstati.
Ogni LWP infatti può trovarsi in NMSTATE microstati. Il significato dei microstati ed il loro numero vengono definiti in *sys/msacct.h*:

```
/* LWP microstates */
#define LMS_USER 0 /* running in user mode */
#define LMS_SYSTEM 1 /* running in sys call or page fault */
#define LMS_TRAP 2 /* running in other trap */
#define LMS_TFAULT 3 /* asleep in user text page fault */
#define LMS_DFAULT 4 /* asleep in user data page fault */
#define LMS_KFAULT 5 /* asleep in kernel page fault */
#define LMS_USER_LOCK 6 /* asleep waiting for user-mode lock */
#define LMS_SLEEP 7 /* asleep for any other reason */
#define-LMS_WAIT_CPU 8 /* waiting for CPU (latency) */
#define LMS_STOPPED 9 /* stopped (/proc, jobcontrol, or lwp_stop) */

/*
 * NMSTATES must never exceed 17 because of the size restriction
 * of 128 bytes imposed on struct siginfo (see <sys/siginfo.h>).
 */
#define NMSTATES 10 /* number of microstates */
```

Le informazioni sui tempi di ogni microstato vengono memorizzate nella seguente struttura definita in *sys/kwlp.h*:

```
        struct mstate {
                int ms_prev; /* previous running mstate */
                hrtime_t ms_start; /* lwp creation time */
                hrtime_t ms_term; /* lwp termination time */
                hrtime_t ms_state_start; /* start time of this mstate */
                hrtime_t ms_acct[NMSTATES]; /* per mstate accounting */
        } lwp_mstate;
```

L'abilitazione al conteggio temporale dei microstati viene compiuta con l'uso della funzione:

```
int ms_set(int onoff)
{
    char buffer[80];
```

```
    int fd;
    int state = PR_MSACCT;

    sprintf(buffer, "/proc/%d", getpid());

    if ((fd = open(buffer, O_RDWR))== -1) {
        perror("open:");
        return(-1);
    }

    if (ioctl(fd, (onoff?PIOCSET:PIOCRESET), &state) == -1) {
        perror("PIOCSET/PIOCRESET");
        close(fd);
        return(-1);
    }
    close(fd);
    return(0);
}
```

La funzione *ms_set()* setta semplicemente il flag PR_MSACCT (definito in *sys/procfs.h*) per il processo che esegue la funzione, consentendo in tal modo l'aggiornamento dei tempi dei microstati.
La sperimentazione pratica delle funzioni *gethrtime()* e *gethrvtime()* ha mostrato che la risoluzione effettiva è dell'ordine del mezzo microsecondo e non di nanosecondi come indicato dalla pagina *gethrtime(3c)*.
In Solaris 2.x troviamo inoltre la API *nanosleep()* che consente di specificare i tempi di sospensioni con risoluzione fino ai nanosecondi, al contrario della *sleep* che specifica i tempi in secondi. Inoltre la funzione nanosleep è "MT safe", cioè la sospensione interessa solo la thread che ha compiuto la chiamata di sistema.

3.5.2 Timer POSIX

Solaris, recependo le proposte del comitato POSIX.1b per l'interfaccia dei sistemi operativi real-time, offre delle chiamate di sistema per la gestione di timer.

I timer POSIX.1b sono più flessibili e potenti rispetto ai timer UNIX per i seguenti motivi:

- la risoluzione è in nanosecondi;
- l'utente può specificare il segnale che deve essere inviato quando il timer scade;
- l'intervallo del timer può essere specificato sia in tempo assoluto che in tempo relativo.

Il numero massimo di timer POSIX che possono essere creati per processo è definito dalla costante_POSIX_DELAYTIMER_MAX nell'header *limits.h*. Nella versione del sistema operativo utilizzata il valore di _POSIX_DELAY-TIMER_MAX è 32.
I timer POSIX vengono gestiti dalle funzioni: *timer_create*, *timer_settime*, *timer_gettime*, *timer_delete*.
La API *timer_create(clockid_t clock, struct sigevent* spec, timer_t* timer_hdrp)* crea dinamicamente un timer. L'argomento *clock* specifica quale clock di sistema il nuovo timer dovrà utilizzare. In Solaris è possibile specificare tre tipi di clock: CLOCK_REALTIME per un timer basato sul tempo reale, CLOCK_VIRTUAL per un timer basato sul tempo di utilizzo del singolo LWP, CLOCK_PROF per un timer basato sul tempo di utilizzo del singolo LWP più il tempo utilizzato dal sistema. L'argomento *spec* definisce l'azione che deve essere intrapresa quando il timer scade. Esso è definito di tipo *sigevent*:

```
struct sigevent {
        int sigev_notify; /* notification mode */
        int sigev_signo; /* signal number */
        union sigval sigev_value; /* signal value */
};
```

Il campo *sigev_signo* specifica il numero del segnale che deve essere inviato allo scadere del timer. Esso è valido solo se il campo *sigev_notify* contiene il valore SIGEV_SIGNAL. Se esso contiene invece *SIGEV_NONE*, nessun segnale viene inviato quando il timer scade. Poiché lo stesso segnale puó essere generato da diversi timer, il campo *sigev_value* è utilizzato per contenere dei dati che consentono di identificare il timer che ha generato il segnale. La struttura di questo campo è:

```
union sigval {
        int sival_int; /* integer value */
        void *sival_ptr; /* pointer value */
};
```

Per catturare il segnale prodotto dal timer quando scade è necessario installare un gestore di segnali. Questa operazione viene compiuta attraverso la chiamata di sistema *sigaction* che indicherà la funzione da compiere ogni volta che viene catturato il segnale del timer.
Il prototipo della funzione che si occupa del segnale deve essere:

```
void signal_handler (int signo, siginfo_t* evp, void* ucontext);
```

Quando viene invocato il gestore di segnali, il campo *evp->si_value* contiene i dati del *spec->sigev_value* del timer che ha generato il segnale.
L'ultimo argomento della API *timer_create* è l'indirizzo di una variabile di tipo *timer_t* (un intero) contenente l'identificatore del timer appena generato.
La API *timer_settime(timer_t timer_hdr, int flag, struct itimrspec* val, struct itimerspec* old)* fa partire o terminare un timer.
Gli argomenti *val* ed *old* contengono rispettivamente il nuovo ed il vecchio valore del timer. Essi sono di tipo *struct itimerpspec*:

```
typedef struct itimerspec { /* definition per POSIX.4 */
        struct timespec it_interval; /* timer period */
        struct timespec it_value; /* timer expiration */
} itimerspec_t;
```

con la struttura *timespec* definita come:

```
typedef struct timespec { /* definition per POSIX.4 */
        time_t tv_sec; /* seconds */
        long tv_nsec; /* and nanoseconds */
} timespec_t;
```

Ogni timer contiene nella variabile *itimerspec::it_value* il tempo rimanente nel timer, mentre la variabile *itimerspec::it_interval* specifica il nuovo tempo che deve essere inserito nel timer. Tutti i tempi vengono specificati in secondi (con il campo *timespec::tv_sec*) ed in nanosecondi (con ilcampo *timespec::tv_nsec*).

L'argomento *flags* può contenere il valore TIMER_RELTIME, se il tempo in cui scadrà il timer è relativo al momento della chiamata del timer_settime, oppure TIMER_ABSTIME, se la scadenza è relativa al 00:00 Universal Coordinated Time, 1 Gennaio 1970.
La API *timer_gettime* è utilizzata per conoscere il valore del timer al momento della chiamata.
Infine la chiamata di sistema *timer_delete(timer_t timer_hdr)* è utilizzata per distruggere un timer creato dalla API *timer_create*.

Capitolo 4

Progettazione di una libreria di moduli di controllo

Spesso la progettazione dell'architettura di controllo di un sistema robotico evoluto è un processo *ad hoc*, in cui larga parte del software viene sviluppata partendo da zero anche se il codice prodotto è simile a quello di altre applicazioni.
Questa mancanza di riuso e di strutturazione nella programmazione richiede le competenze dello sviluppatore su tutti gli aspetti che concorrono a definire le funzionalità del sistema: la competenza dello sviluppatore nella teoria dei sistemi di controllo, nella meccanica dei manipolatori, nell'ingegneria del software, nella teoria dei sistemi in tempo reale, nella programmazione dei sistemi operativi. Pertanto la progettazione e la realizzazione del sistema finale avrà richiesto un considerevole investimento finanziario e umano.
Per ridurre i costi di questi sistemi e contemporaneamente ottenere sistemi più robusti è necessario fornire allo sviluppatore un livello di astrazione tra il sistema operativo in tempo reale e gli algoritmi di controllo. Questa infrastruttura deve fornire le linee guida che facilitino la modifica e l'espandibilità del codice, consentendo una programmazione efficiente per la realizzazione di sistemi riconfigurabili.

4.1 Sistemi riconfigurabili

Un *sistema riconfigurabile* è un sistema di controllo che supporta diverse applicazioni e, all'interno di ognuna di esse, è in grado di gestire diverse organizzazioni dell'hardware. Ogni configurazione è composta da diversi moduli che interagiscono tra di loro e che possono essere condivisi da configurazioni diverse.
Quando si fa l'analisi di un sistema che presenta diverse possibili configurazioni la riconfigurabilità può essere:

- *statica*, se si assicura, all'interno di ogni configurazione, la correttezza della comunicazione tra moduli e il rispetto dei vincoli temporali e delle richieste di risorse di ogni modulo;

- *dinamica*, se si assicura l'integrità del sistema quando si opera una transizione da una configurazione ad un altra.

Un modulo software si definisce riconfigurabile solo se la sua progettazione e programmazione è indipendente dall'applicazione e dalla particolare configurazione hardware del sistema. Il secondo requisito cerca l'indipendenza dalla configurazione e non dall'hardware, infatti un modulo software può anche essere dipendente dall'hardware, ma tutte le sue dipendenze devono essere nascoste all'interno del modulo stesso.

Anche se la progettazione di sistemi riconfigurabili richiede un tempo maggiore rispetto ad una programmazione "ad hoc" quando si progetta una singola applicazione per un hardware ben definito, essa richiede molto meno tempo nel caso in cui si voglia affrontare il problema dello studio di un'architettura di controllo per una sistema dinamico.
Alcuni dei motivi che spingono alla programmazione di sistemi riconfigurabili sono:

- Il software riconfigurabile può supportare diverse applicazioni in un sistema hardware fisso.

- Il software è riusabile. Per definizione, il software riconfigurabile può essere inglobato in un sistema senza modificare il software esistente. Quando il modulo riusabile è stato costruito esso diventa parte di una libreria di moduli. Le nuove applicazioni non devono quindi sviluppare per intero ogni parte del nuovo sistema. Come conseguenza, il tempo di sviluppo dell'applicazione è ridotto significativamente.

- Il sistema è espandibile poiché l'hardware esistente può essere aggiornato, o nuovo hardware o software può essere aggiunto, senza dover riprogettare l'applicazione.
- un modulo può essere facilmente trasferito in ambiti di ricerca diversi che utilizzano la stessa struttura per la progettazione di software riconfigurabile.

4.2 Descrizione dei moduli per un sistema di controllo

Le applicazioni di controllo di un sistema robotico evoluto, come un manipolatore dotato di sensorialità, presentano una gerarchia di livelli di controllo. Comunemente essa è composta da tre categorie: *livello servo*, *livello supervisore*, *livello di pianificazione*.
A livello servo si trova la lettura dei dati dai sensori, l'analisi dei dati, il controllo delle parti elettromeccaniche. I tempi di questo livello sono critiche e richiedono cicli con frequenze che vanno da 1Hz a 1000Hz.
A livello supervisore si trovano azioni di alto livello, come specifiche di un compito, gestione di comandi al robot del tipo "muovi il giunto x di y gradi" o "muoviti nella posizione z". Il tempo a questo livello è un fattore importante, ma non è critico come avviene al livello servo.
A livello di pianificazione vi sono le azioni di controllo delle prestazioni del sistema, la simulazioni di compiti, la programmazioni di nuovi compiti che il sistema dovrà compiere. Nel livello di pianificazione il tempo solitamente non è un fattore critico. Un'architettura di un sistema di controllo deve fornire un livello d'astrazione sufficiente a consentire, in fase di progettazione, di realizzare le trame di controllo evitando di specificare le interfacce fra i moduli e la gestione temporale. Perché l'architettura sia utile allo sviluppatore di un nuovo sistema di controllo essa deve fornire un livello d'astrazione sufficiente per rappresentare queste azioni senza che il progettista si debba preoccupare di progettare gli strumenti necessari all'interfaccia tra i moduli ed alla gestione del tempo.

4.3 Funzioni della libreria proposta

Nella parte rimanente di questo paragrafo, verrà descritta una libreria studiata per rendere più efficace la progettazione e la riconfigurazione di sistemi di controllo robotici.
In particolare la libreria offre:

- la gestione di trame di controllo periodiche, aperiodiche e sporadiche, necessarie ai tre livelli delle applicazioni di controllo di un sistema robotico evoluto (cap. 4.4);
- meccanismi di comunicazione asincrona tra sensori e trame di controllo (cap. 4.5);
- l'individuazione di errori temporali per il non rispetto di vincoli sulla deadline delle trame di controllo periodiche e la possibilità di specificare le azioni da intraprendere in tal caso (par. 4.6);
- un gestore unico dei segnali (par. 4.7);
- la rilevazione automatica dei tempi di esecuzione delle singole thread e del carico sul processore (par. 3.5.1).

4.4 Proposta per un sistema riconfigurabile

4.4.1 Terminologia

In fig. 4.1 è rappresentata la proposta per un'architettura software per un sistema di controllo riconfigurabile. Un *modulo di controllo* è una istanza dell'oggetto modulo che descriveremo nei prossimi paragrafi.
Una *compito di controllo* è il codice con vincoli di tempo reale in esecuzione su un modulo di controllo. Poiché il codice eseguito su ogni modulo di controllo è unico si utilizzerà i termini *modulo* e *compito* alternativamente.
I moduli di controllo possono essere *periodici* e *aperiodici*, e possono realizzare funzioni in tempo reale, tra cui il controllo del moto, il trattamento dei dati, la gestione degli eventi, l'input/output dei dati. I moduli periodici si bloccano in attesa di segnali temporali, mentre i moduli aperiodici si bloccano su eventi asincroni cioè su semafori, messaggi o interrupt.
Una *libreria di moduli* è un insieme di moduli di controllo disponibili per la

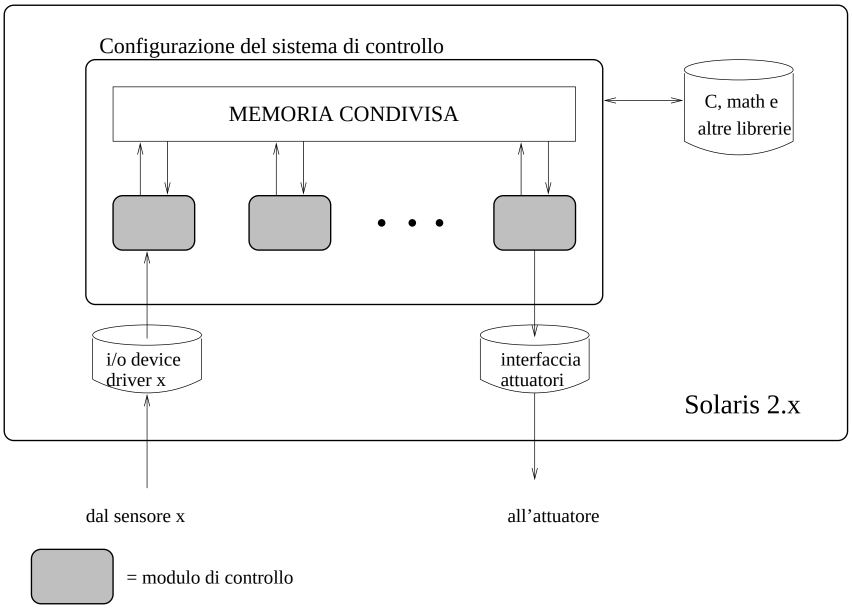


Figura 4.1: Architettura software per un sistema riconfigurabile

costruzione del sistema, l'insieme dei moduli è composto da diverse sottoclassi che specificano le loro particolarità.
Una *configurazione* è formata dall'integrazione di oggetti provenienti dalla libreria di moduli per compiere un compito particolare. Gli oggetti che provengono dalla libreria vengono collegati tra di loro grazie alla condivisione di una memoria comune.
Un'*applicazione* è l'esecuzione in concorrenza dei moduli che costituiscono una configurazione.

4.4.2 Descrizione delle strutture dati del modulo di controllo

Un modulo di controllo possiede informazioni private ed informazioni che condivide con gli altri moduli dell'architettura (fig. 4.2).
Per ogni modulo viene memorizzata la struttura RT_SHELL:

```
typedef struct RT_THREAD {
    /* system params */
    thread_t threadID;
    lwpid_t lwpID;
    /* priority params */
```

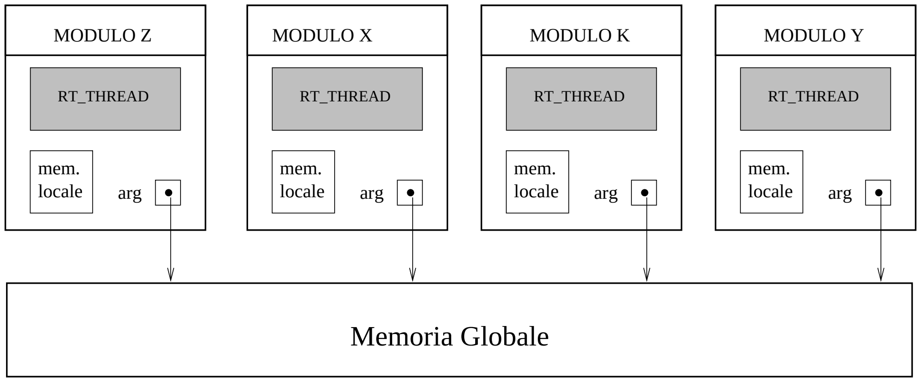


Figura 4.2: Rappresentazione delle strutture dati dei moduli di controllo

```
pcparms_t pcparms;
rtparms_t *rtparmsp;
/* timer params */
struct sigevent evp;
timer_t timerID;
clockid_t clock_id;
int flags;
struct itimerspec value;
struct itimerspec ovalue;
timespec_t rqtp;
timespec_t rmtp;
long cycles;
hrtime_t startR, endR, startV, endV, c;
hrtime_t tot_comp_time;
/* other params */
int state;
/* log onto a file */
unsigned char enabled_log;
char filename[15];
FILE *log;
/* user params */
mutex_t mutexRTParams;
timespec_t A;
timespec_t C;
timespec_t T;
timespec_t D;
void *arg;
void (*StartUpFnx) (struct RT_THREAD *, void *);
void (*CtrlFnx) (struct RT_THREAD *, void *);
void (*ReleaseFnx) (struct RT_THREAD *, void *);
```

```
    void (*TimingFaultFnx) (struct RT_THREAD *, void *);
} RT_THREAD;
```

Nella struttura dati vengono memorizzate le informazioni necessarie all'esecuzione del modulo. Alcune componenti vengono definite automaticamente dalla struttura di controllo e non devono essere utilizzate se non per gli scopi previsti dal progettista, altre vengono definite dall'utilizzatore dell'architettura nella fase di creazione dei moduli. Le prime dovrebbero essere tenute nascoste all'utente[1], mentre l'esistenza ed il significato delle seconde deve essere ben noto all'utente per poter compiere una adeguata scelta sui valori di inizializzazione.
In fase di creazione del modulo l'utente sceglie i seguenti parametri:

- parametri per la gestione del tempo (fig. 4.3);
  - A: intervallo che deve trascorrere prima dell'attivazione del modulo;
  - C: tempo necessario al processore per eseguire completamente il task senza interruzioni (tempo di calcolo);
  - T: periodo;
  - D: tempo entro cui il task deve essere completato (deadline);
- puntatori alle funzioni eseguite dal modulo:
  - *StartUpFnx*: azione eseguita nella fase di inizializzazione
  - *CtrlFnx*: azione eseguita ad ogni ciclo per un modulo periodico o ogni volta che un evento esterno lo richiede per un modulo non periodico
  - *ReleaseFnx*: azione eseguita nella fase di terminazione
  - *TimingFaultFnx*: azione eseguita per il recupero di un errore temporale
- parametri per la descrizione degli andamenti temporali

[1]Il linguaggio C, utilizzato per lo sviluppo dell'architettura, non essendo un linguaggio ad oggetti non offre, purtroppo la possibilità di creare elementi di una classe privati. Pertanto anche variabili che non dovrebbero essere visibili all'utilizzatore possono essere modificate.

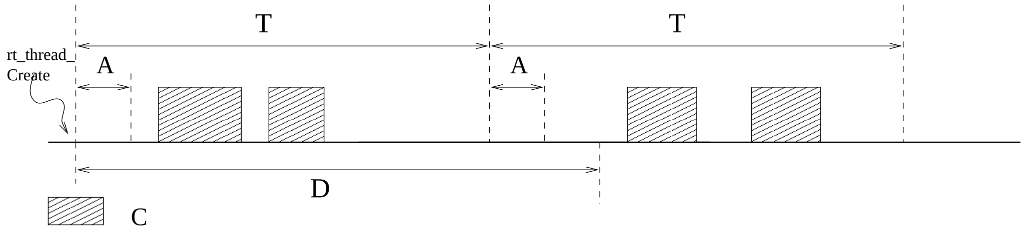


Figura 4.3: Parametri passati in fase di creazione della thread

   - *enabled_log*: flag di scelta per la generazione del file dei tempi di esecuzione
   - *filename*: file contenente i tempi di esecuzione

- *arg*: puntatore alla struttura dati condivisa da tutti i moduli.

4.4.3 Fasi di vita del modulo

Inizializzazione dell'architettura Per rendere attiva l'architettura sono necessarie alcune fasi di inizializzazione prima di creare i moduli che andranno a controllare il sistema.
Oltre alle definizioni che si rendono necessarie per il sistema hardware che si sta utilizzando (ad es., attivazione di hardware condiviso da piú moduli) è necessario compiere la chiamata alle funzioni *rt_shell_init()* e *bufInit()* (par. 4.5.2). La funzione rt_shell_init abilita l'uso dei micro-state con la chiamata alla funzione *ms_set(1)* (par. 3.5.1) ed installa un gestore di segnali per i timer.

Creazione del modulo I moduli necessari all'applicazione vengono creati con la chiamata alla funzione:

```
thread_t rt_thread_Create( timespec_t A,
                           timespec_t C,
                           timespec_t T,
                           timespec_t D,
                           unsigned char enabled_log,
                           void *arg,
                           void (*StartUpFnx) (struct RT_THREAD *, void *),
                           void (*CtrlFnx) (struct RT_THREAD *, void *),
                           void (*ReleaseFnx) (struct RT_THREAD *, void *),
                           void (*TimingFaultFnx) (struct RT_THREAD *, void *),
```

char *filename)

La funzione *rt_thread_Create()* alloca una nuova struttura *RT_THREAD* ed inizializza le variabili in essa contenute.
Quindi crea la thread di gestione del modulo con la chiamata:

```
if (thr_create(NULL, 0, (void *) rt_thread_shell, (void *) rt, THR_DETACHED |
THR_BOUND, &rt→threadID ) != 0) {
        fprintf(stderr,"Cannot create rt_thread_shell\n");
        return(-1);
    }
```

Le thread di controllo dei moduli sono create *THR_DETACHED*, perché in questo modo quando la thread termina, tutte le sue risorse e l'identificatore della thread possono essere riutilizzate e *THR_BOUND* perché in tal modo ogni thread è legata permanentemente ad un LWP. In questo modo le thread sono schedulate dal kernel e possiedono propri allarmi e temporizzatori.
Terminata l'operazione di creazione un puntatore alla struttura del modulo viene inserito nella coda dei moduli creati, contenuta nella struttura *sys_info*.

Esecuzione del modulo Ogni thread esegue la funzione *rt_thread_shell()*. Essa compie le seguenti azioni:

- se è stato richiesto crea il file *filename* che conterrà le informazioni temporali di esecuzione del modulo;
- crea un temporizzatore impostato sul tempo reale;
- sospende la thread con l'API *nanosleep()* per l'intervallo di tempo indicato dal parametro A;
- esegue la funzione *StartUpFnx*;
- esegue ciclicamente la funzione *rt_thread_fnx()* fino a quando la variabile *state*, contenuta nella struttura del modulo, viene posta a RELEASE;
- esegue la funzione *ReleaseFnx*;
- distrugge il temporizzatore;

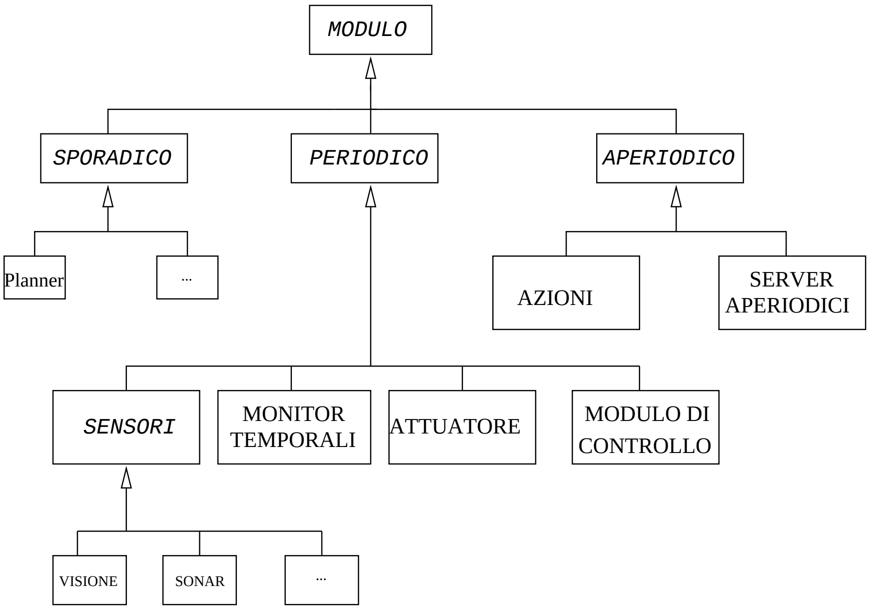


Figura 4.4: Classificazione dei moduli di controllo realizzabili nella libreria proposta

- elimina il puntatore alla struttura della thread dalla coda nella struttura *sys_info*;
- termina la thread.

La funzione *rt_thread_fnx* gestisce i comportamenti ciclici del modulo, eseguendo la funzione *CtrlFnx* ogni periodo T.

4.4.4 Classificazione dei moduli

In fig. 4.4 è rappresentata la libreria di moduli di controllo che l'architettura proposta mette a disposizione. Sfruttando i parametri per la gestione del tempo forniti dalla classe base *modulo* e le tecniche di schedulazione del sistema operativo Solaris 2.x è possibile specificare diverse categorie di moduli. Un modulo *sporadico* è un modulo che può essere eseguito quando il tempo di CPU non è richiesto da altri moduli con vincoli temporali. Per realizzarlo è sufficiente generarlo come aperiodico e lasciare l'LWP che lo esegue nella classe time-sharing.
Un modulo *periodico* viene creato indicando, al momento della sua creazione,

il suo periodo e la sua deadline.
Un modulo *aperiodico* viene creato inizializzando la variabile T, che definizce il suo periodo, ad un valore nullo. Poiché i moduli aperiodici vengono solitamente eseguiti in risposta ad un particolare evento essi avranno all'inizio della funzione di ciclo una sincronizzazione su una variabile condizione del tipo:

```
mutex_lock(&(arg→putMutex));
cond_wait(&(arg→putCond),&(arg→putMutex));
mutex_unlock(&(arg→putMutex));
```

Spetterà al modulo di controllo segnalare ad ogni modulo aperiodico quando la sua esecuzione è necessaria.
Come sottolineato nel par. 4.2, i moduli hanno criticità diverse nei confronti del tempo. Diventa quindi di fondamentale importanza riuscire a strutturare i moduli in una gerarchia di priorità di esecuzione e verificare almeno staticamente che i moduli appartenenti al livello servo rispettino sempre i propri limiti temporali.
Grazie agli strumenti offerti da Solaris 2.x è possibile utilizzare thread kernel in tempo reale con priorità diverse. Ricordando che la classe di scheduling *real-time* ha un intervallo di priorità che va da 0 a 59 si è deciso di assegnare ai moduli i seguenti valori:

```
#define UPPER_PRIO 51 /* the rcictrl task is executed by a maxprio-1 RT thread */


#define SIGHAND_PRIO 11 /* Signal Handler */
#define DRIVER_PRIO 10 /* I/O Driver RT_Thread */
#define SENSOR_PRIO 9 /* Sensor Data Analysis RT_thread */
#define MANAGER_PRIO 8 /* Control Module */
#define TASK_PRIO 7 /* Task RT_Thread */
```

A più alta priorità si colloca il modulo di controllo delle parti elettromeccaniche perché dovendo mandare ai servo regolatori i nuovi valori per il moto dei singoli assi del robot deve rispettare il proprio ciclo per non produrre "jitter" nei movimenti che possono danneggiare gli organi meccanici.
Per verificare il rispetto dei vincoli temporali da parte dei moduli "hard real-time" si possono sfruttare tecniche di scheduling statico che a partire dalla

conoscenza dei tempi di esecuzione e della priorità dei singoli moduli sono in grado di prevedere il comportamento temporale del sistema.
Quando si dispone di moduli periodici, aperiodici e sporadici si possono realizzare tutti i moduli necessari per le applicazioni di controllo di un sistema robotico evoluto.
A livello servo si utilizzeranno moduli periodici per realizzare la gestione dei sensori (par. 4.5) e moduli periodici per il controllo delle parti elettromeccaniche; a livello supervisore si realizzeranno moduli aperiodici per l'esecuzione delle azioni; a livello di pianificazione si utilizzeranno moduli aperiodici per la programmazione di nuovi compiti e moduli periodici per il controllo delle prestazioni del sistema.

4.5 Moduli sensoriali

Nei sistemi robotici evoluti si delega spesso la gestione dei molteplici dispositivi sensoriali a microprocessori esterni. In tal modo il sistema centrale non sarà gravato dalle computazioni sui dati grezzi ma riceverà direttamente le informazioni richieste per l'applicazione. Ogni sottosistema sensoriale è, però, dotato della propria parte di controllo e del proprio sincronismo. Pertanto tali sottinsiemi sono asincroni gli uni rispetto agli altri ed è indispensabile stabilire preventivamente una politica per il trasferimento delle informazioni.

4.5.1 Comunicazione tra sottosistemi

Un sottosistema sensoriale ed il sistema di controllo possono comunicare tra loro in due modi:

- *Sincrono*: un sottosistema sensoriale manda un messaggio o un dato al sistema di controllo e si pone in attesa di una risposta; quando il destinatario riceve il messaggio conferma il ricevimento con una comunicazione. La risposta può essere l'indicazione del successo od insuccesso o può contenere i dati eventualmente richiesti dal primo sistema.

- *Asincrono*: il sottosistema sensoriale ed il sistema di controllo operano indipendentemente, comunicando attraverso l'accesso ad una struttura dati condivisa.

La comunicazione sincrona è più semplice da progettare ma è preferibile utilizzare una comunicazione asincrona che isola il concetto di tempo reale all'interno di ogni sistema. Infatti, l'uso di comunicazioni asincrone, eliminando l'azione di attesa, consente di utilizzare moduli periodici invece di thread la cui esecuzione è controllata dall'arrivo di richieste da parte di altri componenti.
Quando i sottosistemi sensoriali comunicano con il sistema di controllo in modo asincrono è necessario che:

- il sistema che compie una comunicazione non si blocchi: solo in questo modo i tempi di esecuzione di ogni modulo saranno prevedibili
- i moduli di interfaccia per il sistema ricevente forniscano sempre l'informazione più recente, indipendentemente dal numero di comunicazioni avvenute da ogni altro modulo
- siano previsti meccanismi di sincronizzazione che evitino il danneggiamento delle informazioni

Per rispondere alle specifiche appena formulate, è stato realizzato un meccanismo di comunicazione asincrono che si basa su un vettore circolare (fig. 4.5). Il mittente deposita la propria informazione nell'elemento indicato dall'indice, mentre il ricevente preleva l'ultima informazione inserita nel vettore, che si trova nella posizione precedente a quella segnalata.
Si è supposto che ci sia un solo produttore per ogni tipo di sensore che periodicamente inserisce dei dati nella memoria condivisa costituita dal vettore circolare. Contemporaneamente uno o più riceventi o consumatori leggono i dati dallo stesso vettore.

Ad ogni pacchetto di informazioni ricevuto viene associata una indicazione temporale. Essa è costituita da una variabile indicante il tempo fisico, che permette di ricavare le frequenze di arrivi e la loro distribuzione e da un contatore, che consente di controllare quanti pacchetti di dati vengono perduti se il ricevente è più lento del mittente. Se l'indicazione temporale del dato ricevuto è la stessa del dato letto in precedenza il ricevitore è più veloce del mittente e deve riutilizzare i vecchi dati.

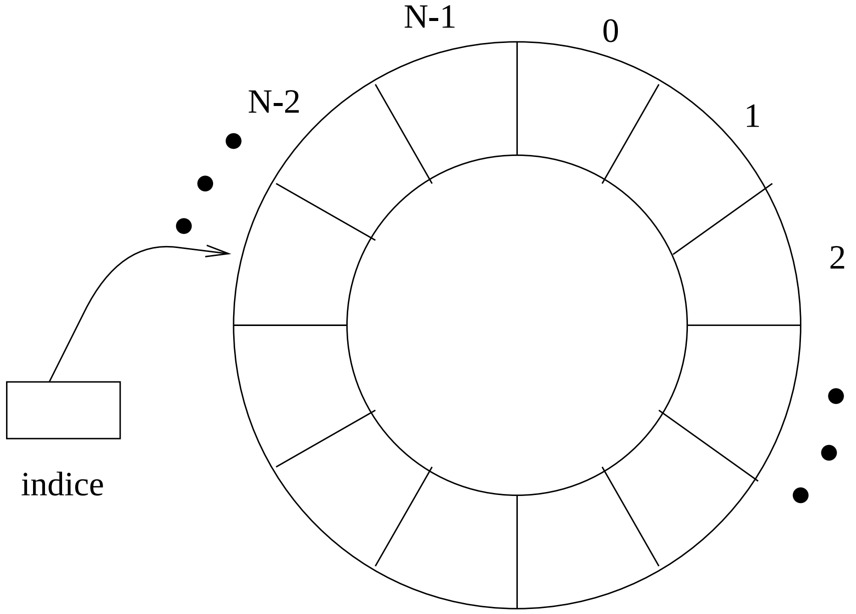


Figura 4.5: Vettore circolare

4.5.2 Realizzazione

Il codice per inserire nuovi sensori nell'architettura si trova nel modulo *buffer_shell*.

Il produttore delle informazioni sensoriali crea una struttura per la comunicazione asincrona attraverso l'istruzione *bufCreate*. La struttura creata, definita in buffer_shell.h, è la seguente:

```
typedef struct _image_type{
    void *buffer_pointer; /* Pointer to the buffer */
    hrtime_t time_stamp;
    int counter_stamp;
}image_type;

typedef struct _sensor{
    int id; /* Identifier of the sensor */
    int index; /* Index for the next image */
    image_type image[BUFFER_SIZE]; /* Buffer for BUFFER_SIZE images*/
}sensor;
```

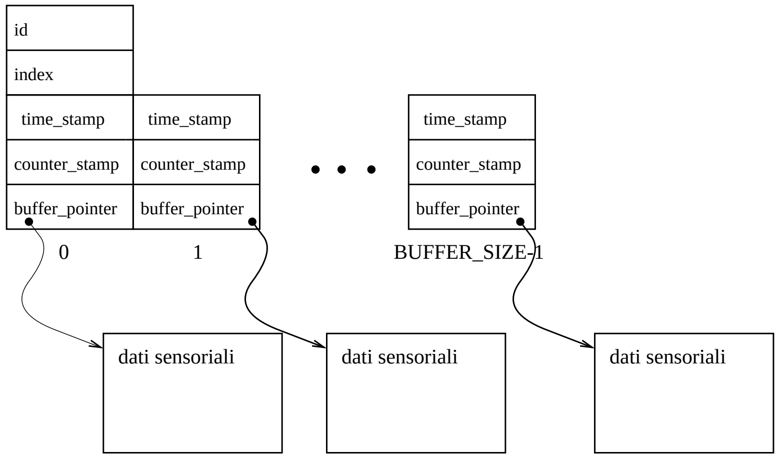


Figura 4.6: Struttura dati per la comunicazione sensoriale

La variabile *id* identifica univocamente il buffer di comunicazione, *index* indica l'elemento del vettore circolare dove dovrà essere inserito il prossimo flusso di informazioni ricevute dal produttore. Gli elementi del vettore sono strutture contenenti *time_stamp* che segnala il tempo fisico in cui è avvenuta la memorizzazione del dato, *counter_stamp* che contiene il numero di arrivo del blocco di dati ed un puntatore ad un'area di memoria che conterrà le informazioni ricevute dal produttore. Per garantire la genericità del codice è stato necessario ricevere, tra i parametri in ingresso, la dimensione dei dati ricevuti dal sensore e allocare per ogni elemento una zona di memoria sufficiente a contenere ogni flusso di informazioni. Diversamente sarebbe stato necessario indicare una struttura dati comune a tutti i sensori che, seppur più semplice da realizzare, avrebbe reso il codice statico.
Per introdurre la gestione di un nuovo sensore dopo aver creato la nuova struttura *_sensor* viene inserito un puntatore ad essa nella lista ai sensori condivisa da tutte le thread.
Il produttore può inserire un blocco di informazioni nel vettore con l'istruzione *bufWrite*. Essa aggiorna i valori temporali inserendo in *time_stamp* il tempo reale ricavato dalla chiamata di sistema *gethrtime()* ed incrementando *counter_stamp*. Successivamente, questa inserisce i dati nella struttura puntata da *buffer_pointer* utilizzando la funzione il cui puntatore è stato passato tra i parametri in ingresso.
Quando il sensore non è più usato, il produttore deve distruggere la struttura di comunicazione con *bufDestroy* per liberare la memoria impiegata.

Il consumatore delle informazioni utilizza tre funzioni *bufAttach*,*bufRead* e *bufDetach*. La funzione *bufAttach* cerca nella lista dei sensori quello che ha come identificatore l'intero *id* che le viene passato, quindi restituisce un puntatore alla struttura di comunicazione del sensore. Si è reso necessario inserire una variabile condizione per gestire questa operazione di "aggancio" al sensore. Il consumatore infatti deve attendere che il sensore sia stato inserito nella lista dal produttore e che almeno un dato sia stato ricevuto. L'uso di una variabile condizione, che conduce ad attesa bloccante del consumatore, non è in contrasto con i principi indicati nel paragrafo precedente perché avviene solo nella fase di inizializzazione della thread consumatore. Con l'istruzione *bufRead* il consumatore riceve un puntatore all'ultima struttura dati inserita dal produttore.
Infine l'istruzione *bufDetach* indica al sistema che il consumatore ha terminato di utilizzare la risorsa.

4.5.3 Utilizzo del sistema realizzato

Basandosi sulla struttura descritta nei paragrafi precedenti l'inserimento di nuovi sensori nell'architettura diventa estremamente semplice. Sono stati creati due file, *mysample_driver.h* e *mysample_driver.c* (Appendice A), che facilitano questa operazione celando allo sviluppatore la struttura necessaria per la condivisione dei dati tra le diverse thread.

4.5.4 Discussione

La struttura presenta alcuni aspetti che possono essere migliorati. Prima di tutto è necessario prevedere dei meccanismi di sincronizzazione per evitare una sovrapposizione tra la lettura e la scrittura dei dati. Infatti, poiché l'operazione *bufRead* restituisce un puntatore agli ultimi dati inseriti, se durante il loro utilizzo il produttore immette altri BUFFER_SIZE (dimensione del vettore circolare) blocchi di dati, va a sovrascrivere la zona di memoria che il consumatore sta utilizzando.
I sensori utilizzati nel compito di riferimento (cap. 4.2) sono piuttosto lenti, pertanto non è stato necessario introdurre alcuna sincronizzazione. Tuttavia, per incrementare la robustezza dell'architettura, non si può trascurare questo problema.

Per proteggere il consumatore è sufficiente associare un mutex ad ogni elemento del vettore circolare e fare in modo che, se il produttore trova la cella del vettore occupata da un consumatore, il dato venga inserito nella cella successiva.
Può essere mossa anche una critica sulla economicità della struttura. Infatti per realizzare un sistema di comunicazione sarebbe sufficiente un meccanismo di comunicazione a triplo buffer utilizzando solo 3 strutture dati([20]). Tuttavia, si è scelto di memorizzare più informazioni contemporaneamente per poter sviluppare applicazioni in cui è necessario tener traccia di una sequenza di dati sensoriali ricevuti dall'ambiente esterno.

4.6 Individuazione e recupero di errori temporali

Nell'architettura di governo robotico non tutti i moduli devono rispettare rigidamente i propri vincoli temporali. Spesso alcuni moduli possono operare in soft real-time, cioè essi, sporadicamente, possono non rispettare le proprie deadline, soprattutto se una azione appropriata viene intrapresa quando questo limite viene superato.
Nell'attuale fase di progettazione il supporto architetturale previsto nella libreria è in grado di rilevare quando il tempo di deadline viene raggiunto da un modulo che non ha ancora completato il suo ciclo. Le azioni da intraprendere in caso di fallimento vengono scelte dall'utente in base al tipo di modulo ed alle funzioni che esso realizza. La funzione di gestione del fallimento potrà pertanto bloccare l'esecuzione che ha superato la deadline, mandare dei messaggi ad altri moduli, richiedere la chiusura graduale del sistema, mantenere delle statistiche sulle frequenze di fallimento o, se il modulo gestisce un algoritmo "any-time", restituire il valore approssimato ottenuto.

4.6.1 Realizzazione

Il meccanismo di rilevazione del fallimento delle deadline è stato realizzato con l'uso di un timer (par. 3.5.2).
Nella fase di inizializzazione dell'architettura viene installato il gestore dei segnali provenienti dai timer con le istruzioni:

```
sys_info.initT = gethrtime();
```

```
/* init rt_thread_sighandler (timing fault) */
sigemptyset(&sys_info.sigv.sa_mask);
sys_info.sigv.sa_flags = SA_SIGINFO;
sys_info.sigv.sa_sigaction = rt_thread_sighandler;

if( sigaction(SIGUSR1, &sys_info.sigv, 0 ) == -1 ) {
    perror("rt_shell_Init fatal error on sigaction");
    exit(-1);
}
```

In questo modo ogni volta che il processo cattura un segnale SIGUSR1 verrà eseguita la funzione di gestione *rt_thread_sighandler*.
Ogni modulo crea il proprio timer all'inizio della sua esecuzione con le istruzioni:

```
/* timer create */
    me->evp.sigev_notify = SIGEV_SIGNAL;
    me->evp.sigev_signo = SIGUSR1;
    /* me->evp.sigev_value.sival_int = 1; not useful here anymore */
    me->evp.sigev_value.sival_ptr = (void *) me;

    me->clock_id=CLOCK_REALTIME; /* CLOCK_VIRTUAL,
CLOCK_PROF */

    if( timer_create( me->clock_id, &me->evp, &me->timerID)==-1 ) {
        fprintf(stderr,"timer_create fails\n");
        rt_shell_exit();
    }
```

Le istruzioni precedenti generano un timer basato sul tempo reale che quando scade spedisce il segnale SIGUSR1.
Terminata la fase di creazione il timer viene inizializzato, all'inizio di ogni ciclo di esecuzione del modulo, al valore del parametro D passato in fase di creazione:

```
  /* set timer to the current thread deadline */
    me->value.it_value.tv_sec = D.tv_sec;
    me->value.it_value.tv_nsec = D.tv_nsec;
    if(timer_settime(me->timerID, me->flags, &me->value, NULL)!=0) {
```

```
        fprintf(stderr,"timer_settime\n");
        rt_shell_exit();
    }
```

Se il timer scade prima del termine dell'esecuzione della *CtrlFnx* associata al modulo viene spedito un SIGUSR1, gestito dalla funzione *rt_thread_sighandler*. Terminato il ciclo di esecuzione il timer viene fermato ponendolo a 0:

```
  /* set timer to the current thread deadline */
   me->value.it_value.tv_sec = D.tv_sec;
   me->value.it_value.tv_nsec = D.tv_nsec;
   if(timer_settime(me->timerID, me->flags, &me->value, NULL)!=0) {
        fprintf(stderr,"timer_settime\n");
        rt_shell_exit();
    }
```

Quando un timer scade viene chiamata la funzione *rt_thread_sighandler*, gestore dei segnali, a cui viene passato, attraverso la variabile *evp→si_value.sival_ptr* il puntatore alla struttura del modulo che possiede il timer che ha generato il segnale. Essa controlla il puntatore a funzione *TimingFaultFnx*: se è stata segnalata una funzione per il recupero dell'errore il gestore la esegue, altrimenti segnala semplicemente l'identificatore della thread proprietaria del timer scaduto.

4.7 Gestione dei segnali

In Solaris 2.x ogni thread possiede la propria maschera per i segnali, e quindi è possibile determinare quali segnali ogni thread può ricevere ed associare un gestore dei segnali dipendente non solo dal segnale a cui è associato ma anche dalla thread che lo ha ricevuto.
Nelle applicazioni che utilizzano le thread è meglio non realizzare gestori di segnali specifici per ogni thread. Per semplificare la fase di programmazione ed avere comunque l'efficienza che si desidera è preferibile costruire una thread che gestisca i segnali che arrivano all'intero processo [21].
Sono possibili due tecniche per progettare la thread di gestione dei segnali. La prima tecnica prevede di mascherare l'arrivo dei segnali asincroni a tutte le thread tranne una, e quindi di fare in modo che questa thread esegua tutti

i gestori dei segnali. In questo caso è sufficiente creare la thread, e quindi bloccarla immediatamente, cercando di decrementare un semaforo. Anche se sospesa, la libreria delle thread la risveglierà per eseguire il gestore dei segnali.
Un altro metodo, che è quello adottato nell'architettura, utilizza la chiamata di sistema *sigwait()*. La thread che la invoca si sospende in attesa. Quando un segnale arriva, essa si risveglia e sulla base del suo valore, può decidere quindi quali azioni compiere.
Nella fase di creazione delle thread sono stati mascherati tutti i segnali, tranne SIGUSR1, necessario alle thread per gestire il proprio timer. Poiché le thread ereditano la maschera dei segnali da chi le ha create, tutte le thread della librearia possono vedere solo il segnale asincrono SIGUSR1.
L'unica thread con un comportamento diverso è la thread che per l'appunto gestisce i segnali. Questa thread, infatti, con la API *sigwait()*, si blocca in attesa di tutti i segnali definiti con il suo argomento. Quando arriva un segnale, *sigwait()* maschera ancora i segnali e restituisce l'identificatore del segnale ricevuto.

```
sigset_t set;
int sig;

sigfillset(&set); /* catch all signals */

while(1) {
    /* wait for a signal to arrive */
    switch (sig=sigwait(&set)) {
        /* here you would add whatever signal you needed to catch */
        case SIGINT:
            printf("Interrupted with signal %d\n",sig);
            printf("Metto exit a 1\n");
            (arg->exit)=1;

            break;
        case SIGTSTP:
            munlockall();
            printf("Exiting....\n");
            exit(0);
            break;
        default:{
            printf("GOT A SIGNAL = %d\n",sig);
            break;
        }
    }
}
```

Esempio di thread gestore dei segnali

4.8 Rilevazione automatica dei tempi

Negli algoritmi di scheduling si suppone di conoscere esattamente il tempo di esecuzioni dei singoli moduli, anche se la sua determinazione è complessa. Nella maggior parte dei casi, sono necessari dei metodi manuali per calcolare il tempo necessario per l'esecuzione di un ciclo. Questo modo di procedere, oltre ad essere lungo, deve essere svolto nuovamente ogni volta che dei cambiamenti vengono introdotti nel codice.
Solaris 2.x offre degli strumenti che consentono di compiere automaticamente le rilevazioni sui tempi di esecuzione delle singole thread e sul carico del processore.

Con l'uso di *gethrtime()* e *gethrvtime()* (par. 3.5.1) si possono calcolare i tempi di utilizzo virtuali e reali della CPU mentre con i timer è possibile realizzare un sistema per la rilevazione dei "timing fault". Sfruttando queste chiamate di sistema sono stati realizzati dei meccanismi che consentono di memorizzare per ogni modulo le seguenti informazioni:

- all'interno della struttura associata alla thread:

  cycles numero di cicli eseguiti (per un modulo periodico) o eventi trattati (per un modulo aperiodico)

  c tempo virtuale utilizzato nell'ultimo ciclo

  overrun numero di cicli che non hanno rispettato il periodo

  tim_fault numero di cicli che non hanno rispettato la deadline

- per ogni ciclo nel file *filename.log* associato ad ogni modulo:
  - tempo reale del ciclo
  - tempo virtuale del ciclo
  - numero di ciclo.

Le informazioni sui tempi di CPU utilizzati vengono memorizzati su file, compiendo un'operazione computazionalmente costosa. Per questo motivo è possibile rifiutare l'operazione di osservazione dei tempi inizializzando a 0, nella fase di creazione del modulo, il valore del flag *enabled_log* contenuto nelle strutture collegate alle thread.
Un'altra utilità per la rilevazione dei tempi è il modulo *time_monitor* contenuto in *system_threads*. Se viene creato come modulo periodico, ad ogni ciclo, stampa nel file *comp_time.dat* le seguenti informazioni:

- valore di *gethrtime()* nel momento presente
- tempo reale trascorso dall'inizializzazione dell'architettura
- tempo virtuale utilizzato da ogni thread.

4.9 Discussione

Si deve verificare, a conclusione di questo capitolo sulla descrizione della libreria progettata, se il sistema di controllo risponde alle specifiche formulate nel

par. 2.4.1 Il primo requisito di un'architettura è che il sottosistema in tempo reale non debba essere riprogrammato quando si modificano l'ambiente e lo scopo. Poiché la libreria che si trova tra sistema operativo e applicazione non è dipendente da ciò che si vuole realizzare il primo requisito è rispettato. Il progettista, inoltre, si deve preoccupare solo della specifica relativa all'azione che il modulo deve compiere, cioé della realizzazione delle funzioni, dipendenti dall'applicazione *StartUpFnx*, *CtrlFnx*, *ReleaseFnx* e *TimingFaultFnx*, e della modalità di esecuzione dello stesso scegliendo in modo opportuno i parametri temporali A,T,D e C. Come si può osservare le tecniche introdotte per assicurare la periodicità delle thread o rilevare il non rispetto delle deadline sono completamente trasparenti.
Il secondo requisito è di poter definire architetture in grado di gestire processi che condividano la risorsa CPU ed eventualmente siano operanti a diverse frequenze.
Solaris offre la possibilità di specificare classi real-time a diversa priorità. Sfruttando tale modalità di esecuzione e gli strumenti per la gestione ad alta risoluzione (3.5.2) è possibile consentire la condivisione della CPU mantenendo però una gerarchia nell'uso che privilegi i moduli appartenenti al livello servo, con caratteristiche di tempo più critiche rispetto ai moduli appartenenti al livello di pianificazione.
All'architettura é poi richiesto che la componente di intelligenza artificiale possa riprogrammare il sottosistema in tempo reale per evitare sovraccarichi delle risorse. Tale requisito è soddisfatto mediante la collezione di supporti per la gestione delle situazioni di sovraccarico e degli errori temporali.
L'ultima requisito è che il sottosistema di intelligenza artificiale conservi le caratteristiche di scelta "intelligente" dell'azione da svolgere e di flessibilità.
Nella fase di realizzazione corrente il sottosistema di intelligenza artificiale è costituito da un modulo periodico di controllo che, dopo aver verificato lo stato del mondo rappresentato dalla struttura dati condivisa decide quali azioni compiere. Poiché non è stata realizzata una componente di pianificazione sofisticata, la scelta dei moduli da attivare e le loro proprietà temporali sono decise a priori. Concludendo il sottosistema di intelligenza artificiale riesce a prendere decisioni "intelligenti" sull'azione da svolgere ed è in grado di modificare i parametri temporali dei moduli se c'è un sovraccarico del sistema ma non è in grado di modificare l'insieme dei moduli eseguiti dal livello di esecuzione e di prevedere se l'insieme dei moduli sarà eseguibile dal sistema robotico.

Capitolo 5

Realizzazione dell'architettura di governo per un compito di manipolazione

5.1 Descrizione del dominio

Un compito di collocazione di parti in scatole è assegnato ad un manipolatore robotico. Un piccolo robot mobile trasporta degli oggetti verso il manipolatore, fungendo da nastro trasportatore o AGV. Nessun vincolo viene posto sul numero dei pezzi posti sulla sommità del robot e sulla sua frequenza di arrivo.

Quando il robot mobile giunge vicino al manipolatore si ferma in attesa che esso prelevi i pezzi, ripartendo appena rileva di non essere più carico. Un limite di tempo viene posto all'attesa da parte del robot mobile: se il manipolatore non riesce a portare a termine lo scarico entro tale limite, il robot mobile ritorna al punto di carico.

Il manipolatore deposita temporaneamente gli oggetti in un'area nei pressi del punto di arrivo del robot, in attesa che venga individuato un percorso tra il deposito e la posizione che il pezzo dovrà occupare nella scatola.

La pianificazine del percorso è dinamica, cioè tiene conto delle variazioni nel punto iniziale, nel punto finale e nella configurazioe dell'ambiente. È noto che gli algoritmi di pianificazione dei percorsi di questo tipo sono di complessità esponenziale se completi, e comunque particolarmente onerosi [22]. In particolare la pianificazione di un percorso richiede tempi dell'ordine del

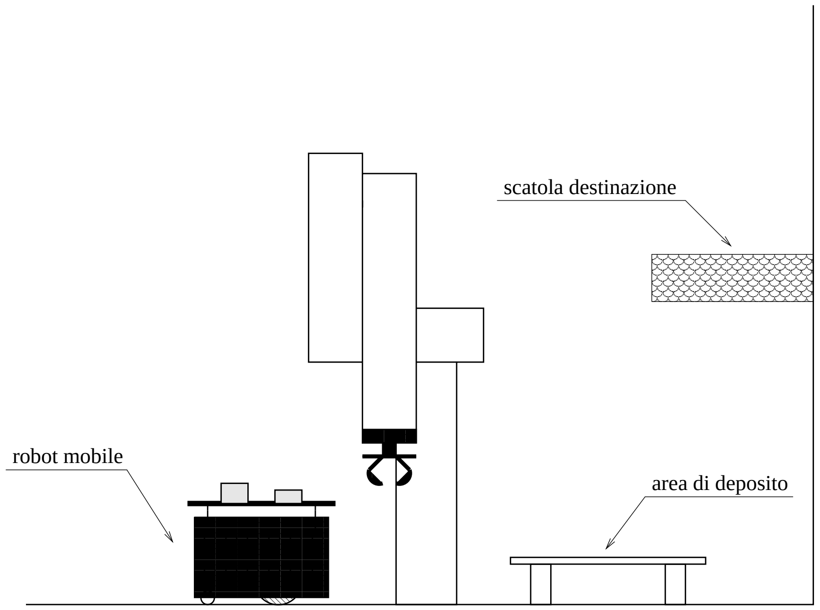


Figura 5.1: Rappresentazione del dominio

minuto nello scenario del task di riferimento con l'elaborazione di un pianificatore di tipo RPP [23] da parte della CPU microSPARC che assolve anche ai compiti di governo del robot.

Il problema descrive un ambiente di lavoro dinamico e presenta alcune sorgenti di incertezza:

- frequenza di arrivo del robot mobile;
- posizionamento degli oggetti sul robot mobile;
- posizionamento degli oggetti sull'area temporanea;
- posizione della scatola;
- tempo di calcolo del percorso;
- tempo di esecuzione del percorso.

La caratteristica di interesse di questo dominio non sta tanto nella complessità delle singole operazioni, quanto nella necessità di scegliere l'azione che in ogni momento risulta essere la più efficace per seguire gli eventi dinamici ed incerti dell'ambiente.

5.2 Un modello per il dominio

La pianificazione di azioni per un robot diventa più complessa all'aumentare della incertezza e dinamicità dell'ambiente in cui opera.
Per descrivere adeguatamente il comportamento del sistema è necessario progettare programmi di controllo flessibili, spesso composti da un insieme di moduli concorrenti in grado di comunicare tra di essi. La complessità del programma rende necessario l'uso di tecniche formali per la sua rappresentazione e per compiere, a partire da essa, una analisi del comportamento dinamico del sistema. Al momento non esiste ancora un metodo universalmente accettato per rappresentare i compiti dei sistemi robotici e ragionare su di essi.

Tra le tecniche proposte vi sono anche le Reti di Petri [24].
Esse consentono di rappresentare l'ordine di esecuzione di un insieme di operazioni, di esprimere l'uso di risorse condivise e di indicare la ripetizione di certe azioni.
La loro idoneità come risposta al problema di rappresentazione del compito sistemi robotici, unita alla possibilità di avere a disposizione pacchetti per l'analisi di sistemi dinamici modellati con Reti di Petri Stocastiche Generalizza (GSPN) [24] i presso il Dipartimento di Ingegneria dell'Informazione ha condotto al loro impiego in questa tesi.

Il modello GSPN del dominio del problema, rappresentato in fig. 5.2, consiste di 4 parti:

- una sottorete (a) che modella i movimenti compiuti sequenzialmente, da parte del manipolatore, per prelevare un oggetto che si trova sulla sommità del robot mobile e porlo nell'area di deposito temporaneo (tray);
- una sottorete (b) che modella i movimenti sequenziali compiuti dal manipolatore per portare un oggetto dall'area di deposito temporaneo alla posizione finale all'interno della scatola;

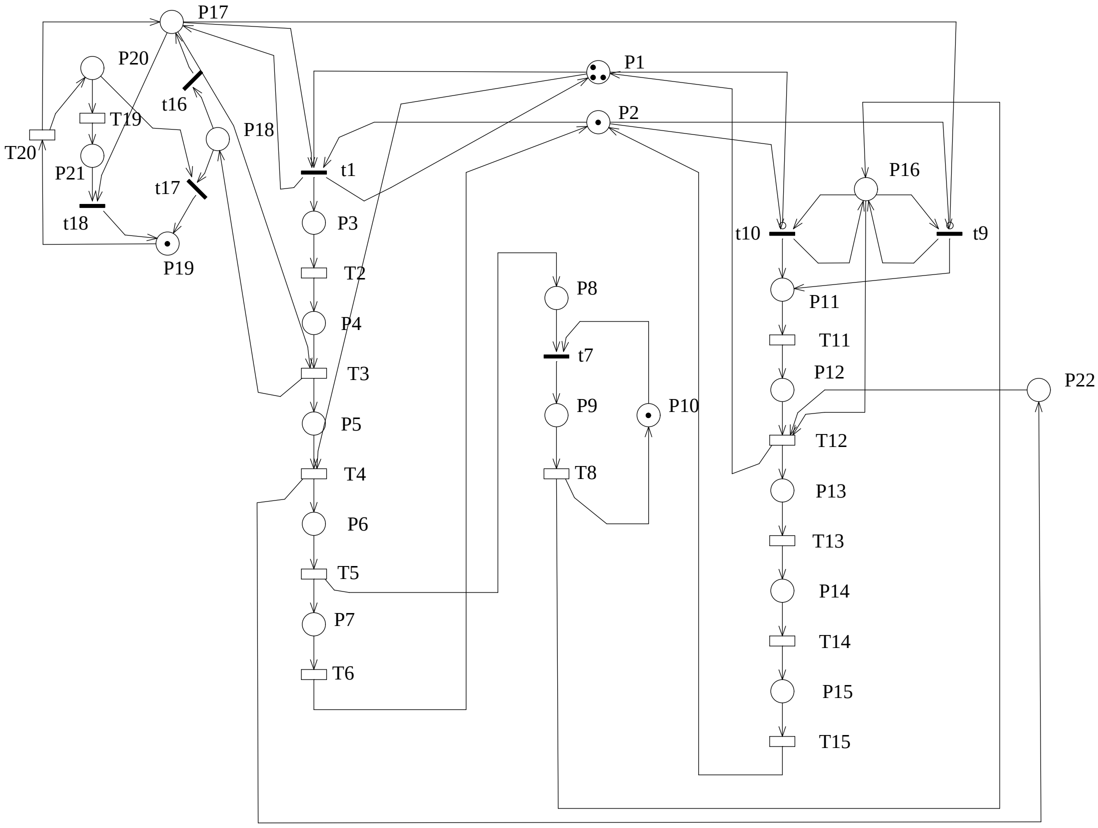


Figura 5.2: Rete di Petri proposta per il dominio del problema

- una sottorete (c) che modella l'arrivo del robot mobile;
- una sottorete (d) che rappresenta l'operazione di calcolo del percorso migliore tra tray e posizione nella scatola.

Il modello GSPN in fig. 5.2 approssima alcuni aspetti del compito reale mentre esprime più accuratamente alcuni elementi di particolare interesso dello scenario di riferimento.
Tra gli aspetti modellati sinteticamente evidenziamo l'arrivo del robot mobile, descritta come arrivo di oggetti separati e la descrizione statica del tempo di calcolo necessario per il pianificatore. Il calcolo del percorso, rappresentato dalla transizione T8, ha infatti, nel modello, durata media fissa definita dalla velocità di transizione, mentre l'applicazione è in grado di modificare

i parametri del sistema di controllo per variare dinamicamente il tempo di CPU utilizzato dal modulo di controllo sporadico che calcola il percorso.
Gli aspetti di particolare interesse espressi dal modello sono la descrizione accurata delle sequenze di azioni del manipolatore e della politica di scelta tra i processi che compongono il sistema.

Posti	*Significato*
P1	Numero di posti vuoti nel tray
P2	Risorsa manipolatore
P3	Movimento dalla posizione di parcheggio al robot mobile
P4	L'oggetto sul robot mobile viene prelevato
P5	Movimento dal robot mobile al tray
P6	L'oggetto viene depositato sul tray
P7	Il manipolatore torna in posizione di parcheggio
P8	Numero di oggetti richiedenti la risorsa CPU per il calcolo del percorso
P9	Computazione del percorso
P10	Risorsa CPU
P11	movimento dalla posizione di parcheggio al tray
P12	L'oggetto sul tray viene prelevato
P13	Movimento dal tray alla posizione finale nella scatola
P14	L'oggetto viene lasciato
P15	Movimento dalla scatola alla posizione di parcheggio
P16	Oggetti con il percorso tray - posizione finale calcolato
P17	Presenza oggetto sul robot mobile
P18	Decisione se ultimo oggetto
P19	Movimento del robot dalla zona scarico alla zona carico e ritorno
P20	Temporizzatore per il ritorno del robot mobile dopo il tempo massimo
P21	Oggetti sul tray

Tabella 5.1: Descrizione dei posti della rete 5.2

5.3 Il sistema robotico

La componente sperimentale di questo lavoro di tesi è stata compiuta utilizzando un sistema robotico disponibile presso il Laboratorio di Robotica del

Transizioni	*Significato*
t1	inizio dell'azione di prelevamento dell'oggetto dal robot mobile
T2	il manipolatore si trova sul robot mobile
T3	l'oggetto è stato prelevato dal robot mobile
T4	il manipolatore è sul tray
T5	l'oggetto è stato depositato sul tray
T6	il manipolatore viene rilasciato
t7	inizio della computazione del percorso tra tray e scatola
T8	il piano è stato computato
t9	Non ci sono più oggetti sul robot mobile
T10	Non c'è più spazio sul tray
T11	il manipolatore è sul tray
T12	il manipolatore ha prelevato un oggetto
T13	il manipolatore è sulla scatola
T14	l'oggetto è nella scatola
T15	il manipolatore è in posizione di parcheggio
t16	sul robot mobile c'è un altro oggetto
t17	sul robot mobile non ci sono più oggetti
t18	inizio del movimento del robot mobile verso la zona di carico
T19	il tempo di attesa è scaduto
T20	il robot mobile è in zona di scarico

Tabella 5.2: Descrizione delle transizioni della rete 5.2

Dipartimento di Ingegneria dell'Informazione di Parma. Il sistema robotico è costituito da:

- un robot mobile
- un manipolatore robotico
- una unità di elaborazione
- la libreria RCI/RCCL
- un sistema di visione
- una dotazione sensoriale di prossimità.

Transizioni	*Durata media*
T2	4
T3	4
T4	4
T5	3
T6	2
T8	45
T10	2
T11	4
T12	9
T13	1
T14	9
T15	2
T19	40
T20	240

Tabella 5.3: Durata media delle operazioni compiute nel dominio di esempio rispetto ad un tempo minimo

5.4 Il robot mobile

Modulus (fig. 5.3) è un robot mobile con struttura cilindrica, sulla cui sommità una base di appoggio permette di trasportare degli oggetti leggeri verso il manipolatore robotico. Il controllo principale è affidato a una scheda GCB11 che utilizza il microcontrollore ad 8 bit Motorola MC68HC11F1FN, programmabile in linguaggio C.
La dotazione sensoriale è costituita da 3 sensori ottici, posti perpendicolarmente al suolo sulla parte anteriore del robot e di 4 sensori infrarossi posti sotto la base di appoggio. Le informazioni ricavate dai tre sensori ottici consentono al Modulus di seguire una linea nera tracciata sul piano di lavoro e di individuare due linee perpendicolari alla precedente che indicano la zona di scarico del manipolatore. I quattro sensori infrarossi rivelano la presenza di oggetti sulla base di appoggio e sono stati inseriti per consentire al Modulus di ripartire quando il manipolatore ha prelevato tutti i pezzi. Una volte arrivato nella zona di scarico il robot mobile aspetta per un intervallo di qualche decina di secondi, trascorso il quale riparte anche se non tutti gli oggetti sono stati prelevati. Il robot mobile comunque riparte verso la zona

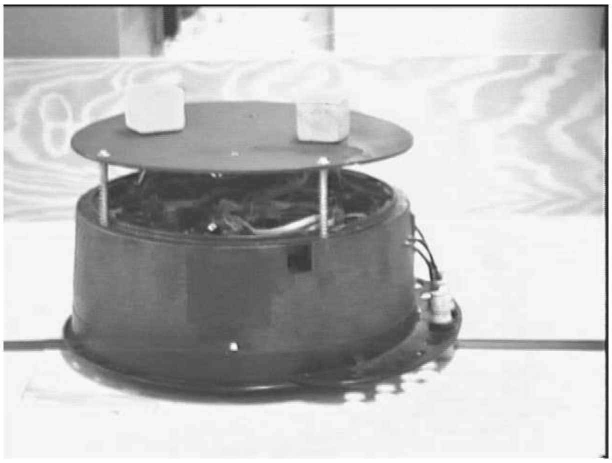


Figura 5.3: Il robot mobile Modulus

di carico una volta che il manipolatore ha prelevato l'ultimo pezzo.

5.5 Il manipolatore robotico e l'unità di elaborazione

5.5.1 Architettura hardware

L'attuale architettura è composta da un cestello VME (fig. 5.4) su cui sono inserite una workstation Unix con CPU microSparc e sistema operativo Solaris 2.4, una scheda XYCOM XVME 240 per l'I/O digitale a 32 bit e una scheda a 32 bit per l'I/O di un sistema video.
La porta parallela presente nel cestello VME è connessa con la porta parallela DEC DRV11 del controllore UNIMATION. Sul controllore Unimation è presente un software di comunicazione, chiamato *moper*, che viene caricato, durante la fase di inizializzazione, attraverso la porta seriale che collega il controllore Unimation con la macchina Sparc. Il *moper* spedisce i comandi di posizione, ricevuti dalla porta parallela, ai microprocessori che controllano i singoli giunti del manipolatore robotico.
Il manipolatore robotico è un braccio meccanico della serie PUMA560 (fig. 5.5), prodotto dall'Unimation Inc. È dotato di sei gradi di libertà e presenta una cinematica antropomorfa, particolarmente flessibile per compiti di manipolazione e di esplorazione dell'ambiente di lavoro. All'estremità del braccio

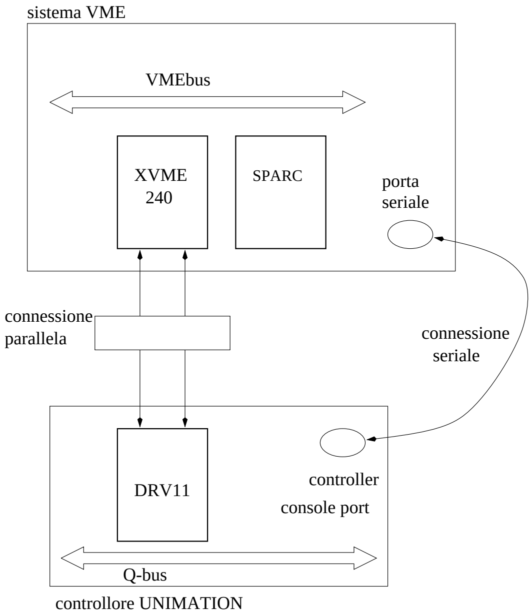


Figura 5.4: Comunicazione tra stazione UNIX e controllore Unimation

sono stati applicati un supporto per alcuni organi di sensorialità ed una pinza con una apertura fino a 70mm, attivata pneumaticamente.

5.5.2 Supporti alla programmazione

Sulla stazione Sparc è installato il pacchetto di librerie RCCL/RCI sviluppato presso il McGill Research Centre for Intelligent Machines.
L'architettura software del sistema RCCL/RCI è mostrata in figura 5.6. Essa è composta da un *livello di pianificazione* che permette il controllo ad alto livello del robot: attraverso le primitive, realizzate dalla libreria RCCL, è possibile specificare una azione con i suoi parametri ad alto livello. Sarà poi il livello di controllo a compiere la traduzione dei movimenti da coordinate cartesiane allo spazio dei giunti e ad assicurare l'opportuna interpolazione di traiettoria.
Il *livello di controllo* è composto da un processo real-time ad alta priorità che calcola l'interpolazione del percorso, compie una traduzione da coordinate cartesiane allo spazio dei giunti, linearizza il moto tra due segmenti di traiettoria contigui e produce un insieme di valori per i giunti ad una frequenza di 30 Hz. Questi valori vengono poi mandati ai servo regolatori che

Figura 5.5: Il robot della serie PUMA 560

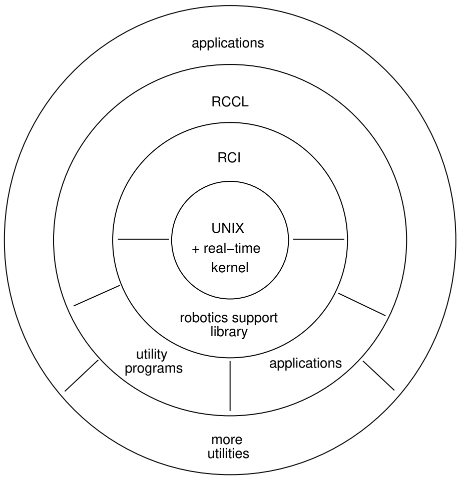


Figura 5.6: Componenti software del sistema RCCL/RCI

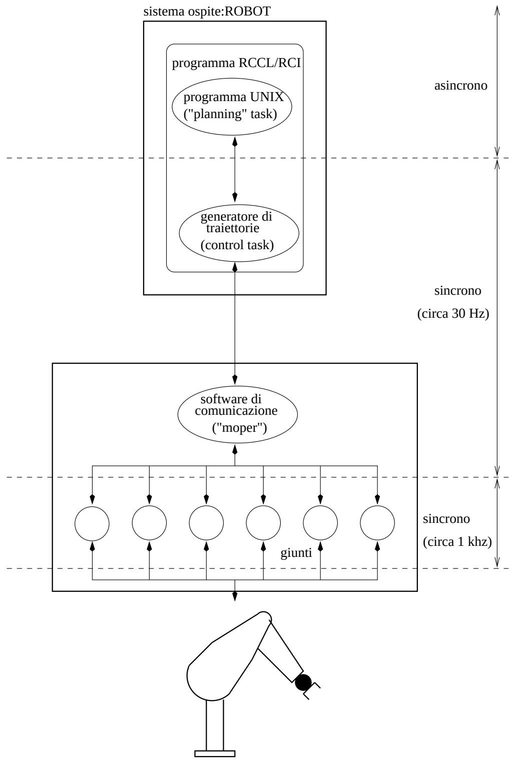


Figura 5.7: Un sistema RCCL/RCI

controllano direttamente i singoli assi di movimento del robot movimenti del robot. Queste funzioni vengono fornite dalla libreria RCI.

Quindi il processo di controllo è strutturato in tre livelli: un livello di pianificazione (*planning task*) asincrono, un livello di controllo (*control task*) sincrono e un livello di controllo dei giunti (fig. 5.7). Per realizzare i primi due livelli si utilizzano le librerie RCCL/RCI, mentre il controllo dei giunti viene lasciato al controllore Unimation.

5.6 Dotazione sensoriale

La dotazione sensoriale eterocettiva è il presupposto per l'autonomia di un sistema robotico in un ambiente non totalmente strutturato. L'uso di meccanismi per la percezione esterna permette infatti al robot di interagire con il suo ambiente in modo versatile. Un robot che può "percepire" e "vedere"

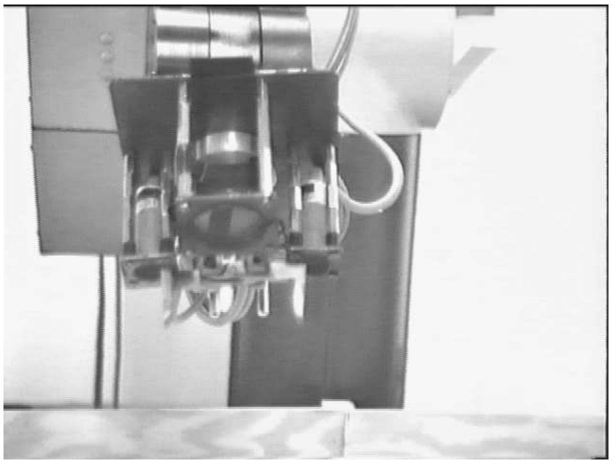


Figura 5.8: Organo di presa e sensori di prossimità

è piú semplice da istruire nell'effettuazione di compiti complessi mentre, allo stesso tempo, richiede meccanismi di controllo meno rigorosi delle macchine preprogrammate. Inoltre un sistema sensoriale istruibile può essere adattato ad una maggiore varietà di compiti e raggiungere così un grado di universalità che si traduce in costi di produzione e di mantenimento più bassi.
Il sistema robotico utilizzato è dotato di sensore ad ultrasuoni, di infrarossi e di un sistema di visione fisso.

5.6.1 Sistema ad ultrasuoni ed infrarossi

Vicino all'organo di presa sono montati un sistema ad ultrasuoni e due fotocellule in posizione parallela alla pinza (fig. 5.8).

Il *sistema ultrasonico* è un sensore Polaroid Ultrasonic Ranging System, costituito da un trasduttore elettrostatico, montato sulla piastra di sostegno dell'organo di presa e da una schedina di interfaccia (Polaroid Ultrasonic RangingBoard 6500), che elabora il segnale inviato e ricevuto dal trasduttore.
A seguito dell'elaborazione il sistema ultrasonico fornisce due segnali:

- *il tempo di volo*: tempo impiegato dal segnale emesso dal trasduttore per raggiungere una superficie riflettente e ritornare al medesimo

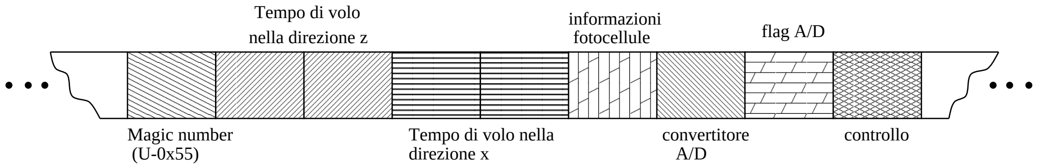


Figura 5.9: Struttura delle informazioni ricevute dal microcontrollore

trasduttore; consente di calcolare la distanza rilevata mediante una semplice conversione matematica;

- *ampiezza*: campione dell'inviluppo della prima parte della forma d'onda ricevuto dal trasduttore.

Il sensore rivela oggetti che si trovano a distanze comprese tra 10cm circa e 10m. L'ampiezza del fascio del sonar è di circa 20°, quindi la distanza misurata non necessariamente è in direzione normale al sensore stesso, il che implica notevoli limitazioni nella risoluzione. Questo problema, comune a tutti i sensori ultrasonici, motiva il loro uso principalmente per il controllo di rotta e per evitare gli ostacoli.

Le due *fotocellule* sono state inserite per coprire la zona di cecità del sensore ultrasonico. Infatti la loro portata, che dipende dal colore e dalla luminosità della superficie riflettente, copre le distanze inferiori ai 100mm, raggiungendo un valore massimo di 120 mm nel caso di oggetti bianchi.

Per minimizzare il carico computazionale del controllore principale, la gestione del sensore ultrasonico e delle fotocellule è affidata ad una scheda a microcontrollore dotata di CPU MC68HC11, collegata tramite porta seriale al sistema UNIX.
Quando la scheda a microprocessore esegue il programma *monitor.asm* manda sulla seriale, ogni 40msec, un messaggio di 9 byte contenente le informazioni significative estratte dai sensori di prossimità(fig. 5.9).

5.6.2 Il sistema di visione

Se la percezione della prossimità riveste un ruolo significativo nel perfezionamento delle prestazioni, la visione rappresenta la più potente delle doti percettive del robot. La complessità computazionale degli algoritmi usati per l'elaborazione delle immagini richiede che l'integrazione della percezione visiva avvenga mediante un sistema hardware dedicato.

Caratteristiche tecniche

Il sistema di visione utilizzato è un DataLogic DVS1000, programmabile per l'acquisizione ed elaborazione di immagini ad uso industriale. Esso presenta le seguenti caratteristiche:

- unità centrale di elaborazione, costituita da un microprocessore a 12.5 MHz Motorola MC68000, che coordina le attività di tutti i componenti;
- unità di elaborazione dei segnali con microprocessore Hitachi HD81831Y20 DSP, che lavora come co-processore e si occupa di eseguire più velocemente manipolazioni delle immagini;
- quattro banchi di memoria esterna:
  - due banchi di 64Kbytes, chiamati BANK0 e BANK1
  - un banco di 64K parole di 16bit ciascuna, chiamato BANK2
  - un banco di 64K parole di 6 bit ciascuna, chiamato FRAME GRABBER, in cui vengono memorizzate le immagine acquisate. Il DSP non può scrivere questo banco;
- risoluzione dell'immagine pari a 256x256 pixel a 64 toni di grigio;
- programmazione in C con l'uso di librerie che implementano alcuni algoritmi standard per l'elaborazione delle immagini.

Informazioni sensoriali

Le informazioni che vengono fornite dal sistema di visione devono consentire di individuare come sono disposti gli oggetti nello spazio del manipolatore robotico.

Prima del suo utilizzo in questa tesi il sistema di visione era già stato impiegato per realizzare l'inseguimento di un oggetto da parte del manipolatore robotico. Questo lavoro precedente ha agevolato l'individuazione delle funzioni della libreria utili per ricavare le informazioni richieste. Si è resa però necessaria una completa revisione del codice: lo si è suddiviso in più moduli per renderlo più generale; si sono individuati ed eventualmente eliminati alcuni errori delle librerie di sistema per renderlo più robusto; si è deciso di non visualizzare a video il risultato delle operazioni sulle immagini per aumentare la velocità. I moduli costruiti sono:

- *serial* funzioni per la comunicazione seriale tra il sistema di visione e la stazione Sparc
- *time* funzioni per la determinazione delle prestazioni temporali
- *utility* funzioni semplici per la manipolazione di immagini e di verifica di parametri
- *wind_par* funzioni che a partire da una porzione dell'immagine contenente un'oggetto ne determinano alcuni parametri (baricentro, angolo, colore)
- *funz_imm* funzioni complesse per la gestione delle immagini contenute in un banco di memoria.

Per consentire una maggiore leggibilità del codice, tutti i parametri del programma sono stati inseriti nel file *param.h.*
Dopo aver configurato il canale di comunicazione seriale ed aver ricavato il rapporto tra pixel dell'immagine e centimentri nell'immagine reale, il programma compie ciclicamente i seguenti passi:

- l'immagine viene acquisita, digitalizzata e memorizzata nel FRAME_GRABBER;
- la funzione *binary*, contenuta nel modulo *funz_imm* binarizza l'immagine nel FRAME_GRABBER. La binarizzazione esporta nel BANK0 una versione dell'immagine con solo 2 livelli di grigio (bianco o nero).
- a partire dall'*occupancy array* [1] si ricavano le catene di Freeman [25], che rappresentando il solo contorno di un oggetto permette una rappresentazione più conciso dell'informazione contenuta nell'immagine;
- la funzione *ind_window_box*, contenuta nel modulo funz_imm, seleziona, a partire dalle catene di Freeman, gli oggetti il cui rettangolo circoscritto ha dimensioni in pixel corrispondenti alle dimensioni in centimetri degli oggetti reali che si trovano sul Modulus;

[1] l'*occupancy array* è un vettore bidimensionale con dimensioni pari alla risoluzione dell'immagine i cui elementi avranno valore 1 se il corrispondente pixel è acceso, 0 altrimenti

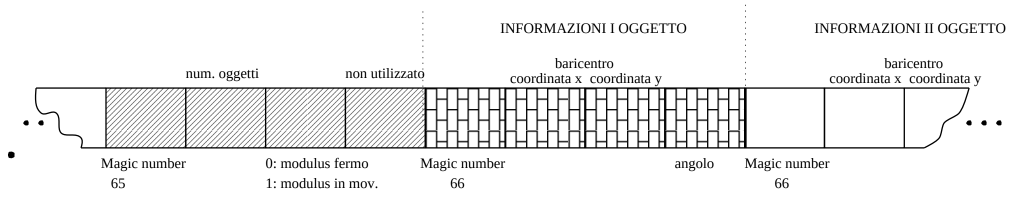


Figura 5.10: Struttura delle informazioni ricevute dal sistema di visione

- Si ricavano i baricentri e gli angoli di rotazione rispetto all'orizzontale dell'asse principale d'inerzia degli oggetti selezionati, sfruttando le proprietà dei momenti [25].

Il sistema di visione, dal confronto con i baricentri trovati in due cicli successivi di scansione dell'immagine, riesce a stabilire se il Modulus è fermo o in movimento e se qualche oggetto è stato prelevato, decidendo quindi, se l'ambiente si è modificato, di inviare informazioni al sistema centrale.
Le informazioni che vengono comunicate, attraverso una linea seriale, all'unità Sparc sono:

- informazioni sullo *stato del Modulus*;
- *numero di oggetti* presenti nell'immagine;
- per ogni oggetto, le *coordinate del baricentro* e l'*angolo di rotazione.*

La struttura delle informazioni trasmesse é mostrata in fig. 5.10.
Queste informazioni sono significative solo se si è compiuta una calibrazione della telecamera, cioè, se si sono individuati i parametri che consentono di convertire le coordinate in due dimensioni delle immagini nelle coordinate tridimensionali del sistema di riferimento del manipolatore robotico. Questa operazione particolarmente delicata, è descritta in una delle appendici.
Durante la fase di verifica del programma sono stati identificati alcuni errori del sistema. Alcuni pongono delle restrizioni al programmatore obbligandolo ad utilizzare alcune variabili solo come globali o ad inserire stampe a video inutili se non per evitare "Bus Error" incomprensibili.
Oltre a questi errori, sono presenti errori anche nelle librerie che rendono inutilizzabili alcune funzionalità che il sistema dovrebbe offrire.
Le funzioni di libreria *bet_init()*, *bet_start()* e *bet_trace()*, che usate congiuntamente consentono di determinare la catena di Freeman, entrano facilmente

in un ciclo infinito bloccando il sistema. Si è parzialmente recuperato questo errore ponendo il limite MAXCATENE (pari a 30) al numero di catene determinabili. In questo modo si rende operativo il sistema anche quando l'immagine è complessa e quindi più facilmente il sistema si porterebbe in una situazione di errore.
Nella libreria fornita con il sistema di visione è presente la funzione *reg_moment()* che dovrebbe calcolare i momenti da cui determinare gli invarianti che consentono una classificazione dell'oggetto. Diverse prove, compiute variando i parametri passati alla funzione e calcolando gli invarianti, hanno portato però alla conclusione che la funzione non produce i risultati promessi, visto che per lo stesso oggetto, ruotato in posizioni diverse, si ottengono invarianti dai valori estremamente diversi. Questo errore non ha consentito di lasciare al sistema di visione il carico della elaborazione necessaria per il riconoscimento degli oggetti.

Prestazioni

Dopo aver verificato la correttezza delle informazioni ricavate dal programma descritto nel paragrafo precedente si è proceduto ad un calcolo delle prestazioni del sistema DataLogic DVS1000 per avere una misura dei tempi di ciclo della percezione visiva. Una prima serie di esperimenti è stata compiuta ponendo un numero variabile di oggetti su sfondo nero (tab. 5.4). Come era prevedibile all'aumentare del numero di oggetti si ha un aumento del tempo minimo e medio necessari per rilevare le informazioni, a causa di una maggiore complessità dell'immagine. I tempi massimi non sono significativi perché inficiati dall'errore presentato precedentemente che porta il sistema a determinare le catene di Freeman di 30 oggetti anche se questi non sono presenti.
In una successiva serie di esperimenti (tab. 5.5) compiuti sostituendo allo sfondo nero uno sfondo non ingegnerizzato, ossia il Modulus sulla linea nera che indica il percorso, i tempi aumentano notevolmente poiché il maggior numero di oggetti nell'immagine fa aumentare sensibilmente i tempi necessari ai complessi algoritmi di elaborazione dell'immagine. Alla luce di questi risultati si è preferito avere una informazione meno frequente piuttosto che dover porre vincoli all'ambiente di lavoro.
Come informazione aggiuntiva, si è poi cercato di verificare se l'orientazione dell'oggetto o la sua posizione all'interno dell'immagine potessero influenzare i tempi di calcolo delle informazioni. I risultati, ottenuti facendo variare i parametri di angolo di rotazione e di posizione di un oggetto e riportati nelle tabelle 5.7 e 5.6, non evidenziano alcuna dipendenza tra i tempi e queste due

num. oggetti	*medio*	*minimo*	*massimo*
0	0.297	0.282	0.707
1	0.384	0.383	0.394
2	0.424	0.405	0.429
3	0.444	0.441	0.450
4	0.464	0.461	0.484
5	0.505	0.501	0.511

Tabella 5.4: Tempi (in sec.) per l'individuazione dei baricentri di uno o più oggetti posti su sfondo nero (50 prove)

num. oggetti	*medio*	*minimo*	*massimo*
0	0.773	0.744	1.198
1	0.805	0.785	0.827
2	0.831	0.813	0.876
3	0.842	0.820	0.883
4	0.878	0.816	0.927
5	0.916	0.897	0.938

Tabella 5.5: Tempi (in sec.) per l'individuazione dei baricentri di uno o più oggetti posti sul Modulus posto sul piano di lavoro non nero (50 prove)

variabili.

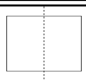
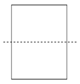
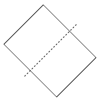
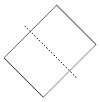
angolo	*medio*	*minimo*	*massimo*
	0.385	0.383	0.403
	0.385	0.384	0.401
	0.401	0.385	0.406
	0.385	0.384	0.402

Tabella 5.6: Tempi, in secondi, necessari per individuare un baricentro di un oggetto posto su sfondo nero con diverse orientazioni(120 prove)

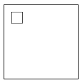
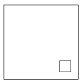
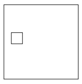
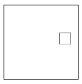
posizione nell'immagine	*medio*	*minimo*	*massimo*
	0.385	0.383	0.399
	0.404	0.396	0.405
	0.384	0.384	0.402
	0.385	0.384	0.400

Tabella 5.7: Tempi, in secondi, necessari per individuare un baricentro di un oggetto posto su sfondo nero in diverse posizioni nell'immagine ricevuta (120 prove)

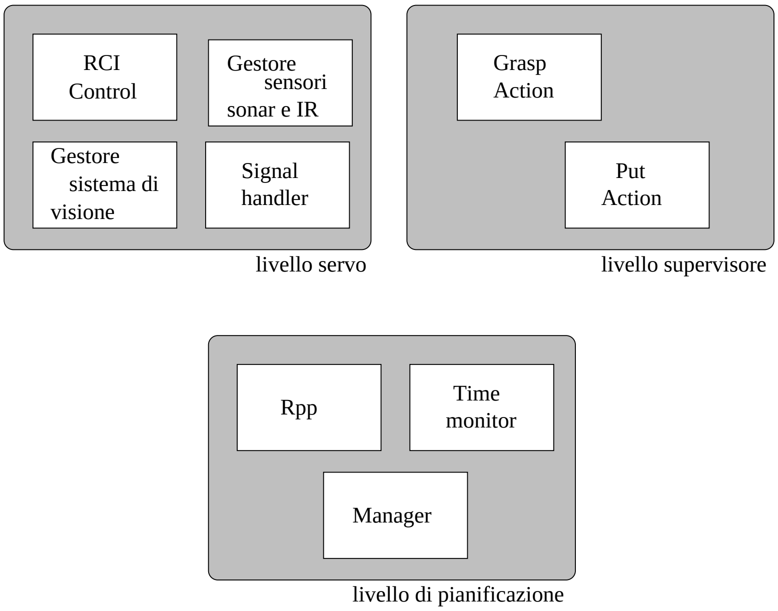


Figura 5.11: Moduli di controllo sviluppati per la realizzazione dell'applicazione proposta

5.7 Moduli di controllo

Sfruttando la libreria descritta nel cap. 4 e gli strumenti appena descritti si è realizzato il compito proposto nel par. 5.1.
In fig. 5.11 sono mostrati, classificati per livello, i moduli di controllo che compongono l'applicazione. Poiché non è stata ancora completata la realizzazione dell'architettura proposta, la suddivisione dei moduli nei tre livelli del sistema di controllo del manipolatore robotico non rispecchia pienamente l'appartenenza ai tre livelli dell'architettura proposta.
In particolare a livello supervisore si trovano i moduli che compiono il ciclo di azioni per la cattura dell'oggetto dalla sommità del robot mobile ed il ciclo di azioni per lo spostamento dell'oggetto in una scatola destinazione.
Attualmente la sequenza di azioni è determinata staticamente dal progettista ma, se l'architettura fosse completa, la sequenza di azioni verrebbe determinata dinamicamente dal livello di pianificazione.
Un'altra conseguenza del non avere sviluppato un'algoritmo di pianificazione completo è la presenza nel sistema di controllo di un modulo, il manager, che compie funzioni a due livelli. Da una parte modifica i valori dei parametri

temporali compiendo un'azione del livello di supervisione, dall'altra verifica le condizioni di risveglio per i moduli aperiodici compiendo un'azione di pianificazione.
Infine, l'ultima semplificazione è stata l'introduzione del modulo *time_monitor* nel livello di pianificazione poiché nell'applicazione è stato utilizzato soprattutto per ottenere informazioni sui comportamenti. Se il livello di pianificazione dell'architettura fosse completamente sviluppato il modulo *time_monitor* si collocherebbe al livello di controllo per trasmettere informazioni di feedback al livello di pianificazione.

5.7.1 Livello servo

A livello servo si collocano i moduli per il controllo delle parti elettromeccaniche e per la lettura dei dati dai sensori.
Il generatore di traiettorie per il manipolatore Puma viene creato autonomamente dalla libreria RCI/RCCL (par. 5.5.2) associando la sua esecuzione ad una thread a priorità massima meno uno che viene creata durante la fase di inizializzazione del *livello di controllo* (par. 5.5.2). Poiché essa possiede la priorità maggiore fra tutte le thread è garantito che la sua esecuzione avrà precedenza su tutte le trame di controllo e che non verrà mai revocata. La thread non è tuttavia creata a priorità massima per consentire, in caso di guasto, il recupero del sistema attraverso una thread di priorità massima.
A livello servo si trovano anche i moduli di controllo per la lettura dei dati sensoriali.
L'architettura hardware utilizzata presenta due sistemi sensoriali eterocettivi: uno di prossimità costituito da ultrasuoni ed infrarossi (par. 5.6.1) ed uno di visione (par. 5.6.2). Per ognuno di questi è stato realizzato un modulo di controllo per la gestione dei sensori, cioé un modulo periodico ad alta priorità la cui esecuzione può essere revocata solo dalla thread del generatore di traiettorie.

5.7.2 Livello supervisore

A livello supervisore si collocano le azioni di alto livello. Nel dominio studiato sono necessari due cicli di azioni che vengono gestite da due thread. Il primo ciclo di azioni sposta un'oggetto dalla sommità del robot mobile

Modulus alla zona di deposito temporaneo. Il secondo ciclo di azioni consente il posizionamento dell'oggetto in una collocazione specifica di una scatola destinazione seguendo il percorso calcolato da un algoritmo di pianificazione. In fig. 5.12 è mostrato l'evolversi degli stati in cui si trova il manipolatore durante i due cicli di azioni. I due insiemi di stati condividono lo stato WAIT_OK che descrive la situazione del sistema quando è in attesa di verificare le condizioni per uno dei due cicli di azioni. Il passaggio da un ciclo di azioni all'altro deve essere compiuto esclusivamente quando il sistema si trova in questo stato. Infatti, come sottolineato nel par. 2.4.3, non ci può essere una transizione da uno stato condiviso ad uno non condiviso perché occorre garantire che il sistema, quando riceve un nuovo programma, si trovi in uno stato in grado di gestirlo.

5.7.3 Livello di pianificazione

A livello di pianificazione vi sono le azioni di controllo delle prestazioni del sistema, le simulazioni di compiti e la programmazione di nuovi compiti che il sistema dovrà eseguire.
A questo livello l'applicazione realizzata presenta tre moduli di controllo: *time_monitor*, *manager* e *compute_rpp*.
Il codice di *time_monitor* è eseguito da un modulo di controllo con priorità utente. Il modulo produce periodicamente informazioni sui comportamenti temporali dei moduli di controllo dell'applicazione.
Il secondo modulo di controllo di questo livello è un modulo periodico che esegue il codice contenuto in *manager.c*. Esso si colloca alla priorità più alta tra i moduli del livello di pianificazione perché deve essere reattivo agli eventi dell'ambiente che vengono rappresentati nella struttura condivisa da tutte le thread. Ciclicamente questo modulo controlla se sono verificate le condizioni per il risveglio dei moduli aperiodici bloccati su una variabile condizione (*grasp*, *put* e *compute_rpp*). Se questo accade risveglia la thread corrispondente. A questo modulo spetta anche il diritto di modificare i parametri delle altre thread quando si accorge che il sistema é sovraccarico o quando vuole concedere piú tempo ai moduli sporadici.
L'ultimo modulo di controllo che compone questo livello è *compute_rpp*. Questo modulo esegue un algoritmo di pianificazione [22][26], che determina un percorso senza collisioni all'interno di uno spazio di lavoro stazionario. Esso si basa su un approccio Random Path Planning, che sfrutta una ricerca di tipo

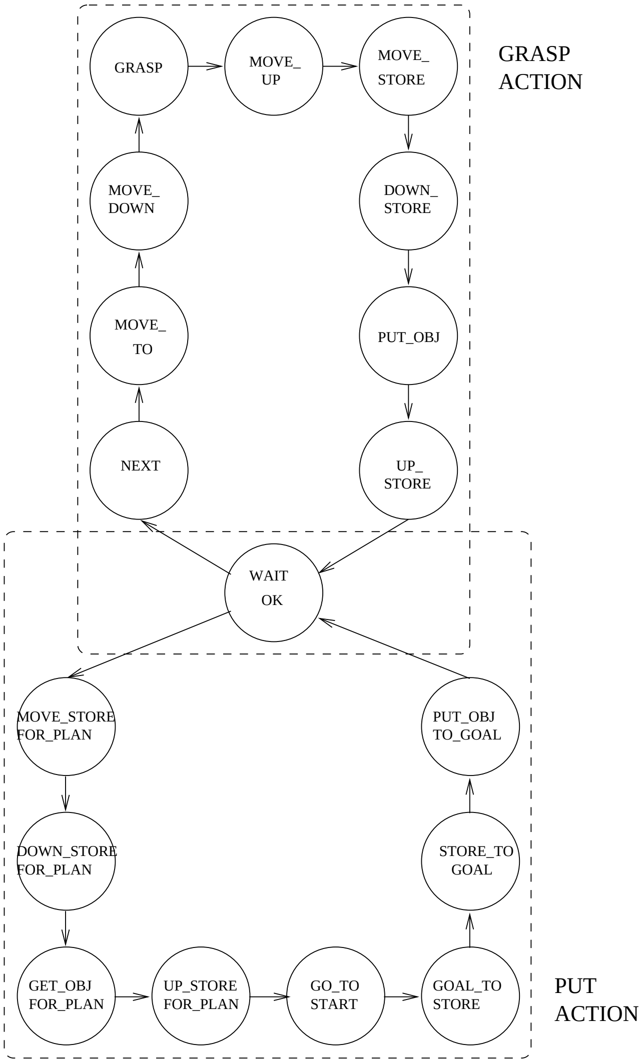


Figura 5.12: Sovrapposizione di uno stato per consentire il passaggio da un ciclo di azioni all'altro

casuale nello spazio delle configurazioni libere del robot. Poiché gli algoritmi di pianificazione dei percorsi di questo tipo sono di complessitá esponenziale se completi, e comunque particolarmente onerosi, questo modulo presenta tempi di computazione elevati, dell'ordine del minuto, ed elevata varianza. Per queste sue caratteristiche é stato integrato nell'architettura di controllo come modulo sporadico. Quando viene risvegliato dal manager, esso viene schedulato solo allorché il processore è libero, ossia quando non ci sono moduli periodici a più alto livello attivi. Questo tipo di modulo soffre quindi la possibilità di *starvation* quando la CPU è sovraccaricata da moduli periodici.

Prima di mostrare un esempio di interazione tra i moduli di controllo che compongono l'architettura di governo è necessario descrivere brevemente le strutture dati che sono state realizzate per l'applicazione. Nella zona di memoria globale sono contenuti:

- dati richiesti dalla libreria
  - identificatori delle thread che eseguono il codice dei moduli di controllo
  - variabili condizione per i moduli aperiodici
  - la variabile *exit*, che viene posta ad 1 per indicare alle thread la chiusura dell'applicazione
  - puntatore alla lista dei sensori
- dati richiesti dall'applicazione
  - variabili di inizializzazione per i sensori
  - puntatore alla struttura che descrive il manipolatore robotico
  - puntatore alla struttura *store* (fig. 5.13

Per comprendere le chiamate di funzioni utilizzate nel prossimo paragrafo è necessario descrivere brevemente la struttura *store*:

```
typedef struct _place_inf {
  int object_in; /* 1: posto occupato, 0: posto libero */
  int plan; /* 1: rpp completato, 0: rpp non completato */
  coord storePos;
  coord goal; /* goal della pianificazione */
```

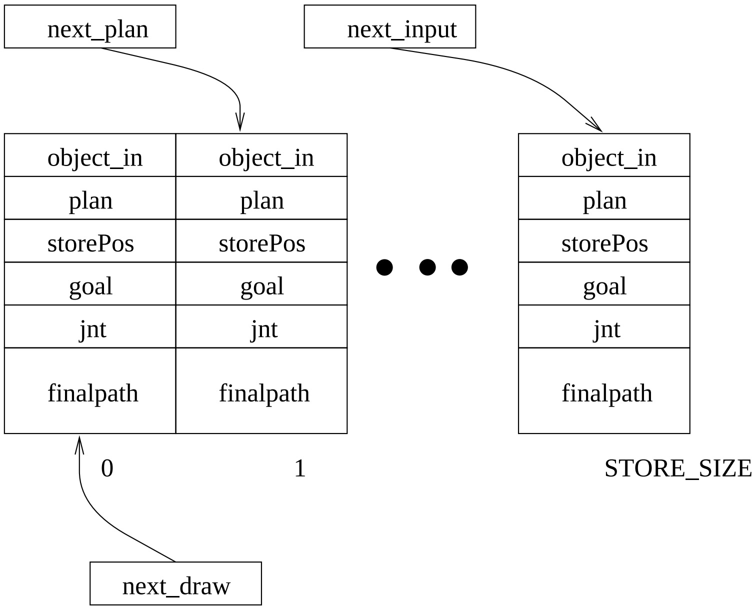


Figura 5.13: La struttura dati *store*

jntStruct jnt; /∗ *valori dei giunti nella configurazione iniziale*∗/
path finalpath; /∗ *path calcolato*∗/
} place_inf;

typedef struct _store_inf{
int next_input; /∗ *posto dove inserire il prossimo oggetto* ∗/
int next_plan; /∗ *prossimo da pianificare* ∗/
int next_draw; /∗ *posto dove andare a prelevare (se e' pronto) il prossimo oggetto* ∗/
place_inf place[STORE_SIZE];
} store_inf;

Per ogni posto disponibile nell'area di deposito temporanea sono presenti due flag: *object_in* per indicare se il posto è occupato già da un oggetto e *plan* per indicare se il percorso per l'oggetto presente è già stato pianificato.
Le variabili *storePos*, *goal*, *jnt* contengono le informazioni di ingresso per il pianificatore, mentre in *finalpath* viene inserito il percorso ottenuto.
Le variabili *next_input*, *next_plan* e *next_draw* indicano rispettivamente la posizione dove inserire le informazioni del prossimo oggetto prelevato dal robot mobile, la posizione dove ricavare le informazioni di ingresso per la prossima

pianificazione e la posizione dove si trovano le informazioni sul prossimo oggetto da prelevare dall'area temporanea.

5.7.4 Libreria per l'applicazione

Per accedere a *store* sono state definite delle procedure che, nascondendo la struttura dati, permettono di modificarla senza per questo dover aggiornare il codice dei moduli di controllo.

- *initStore()*: inizializzazione struttura dati
- *storedObj()*: aggiorna i dati quando il manipolatore ha depositato un oggetto nell'area temporanea
- *drewObj()*: aggiorna i dati quando il manipolatore preleva un oggetto dall'area temporanea per portarlo nella posizione finale
- *computedRppObj()*: aggiorna le informazioni quando un percorso è stato pianificato
- *findStorePos()*: restituisce un puntatore alle coordinate della posizione finale
- *findGoal()*: restituisce un puntatore alla struttura contenente le coordinate della posizione finale per il pianificatore
- *findJntPos()*: restituisce un puntatore alla struttura contenente il valore di partenza dei giunti per il pianificatore
- *findFinalpath()*: restituisce un puntatore alla struttura che dovrà contenere il piano calcolato dal pianificatore
- *findNextRpp()*: restituisce un puntatore al piano che il manipolatore dovrà eseguire per portare l'oggetto nella posizione finale
- *findDrawPos()*: restituisce un puntatore alla posizione iniziale dell'oggetto nell'area di partenza
- *memStartPos()*: memorizza il valore dei giunti della posizione iniziale che verranno poi passati al pianificatore.

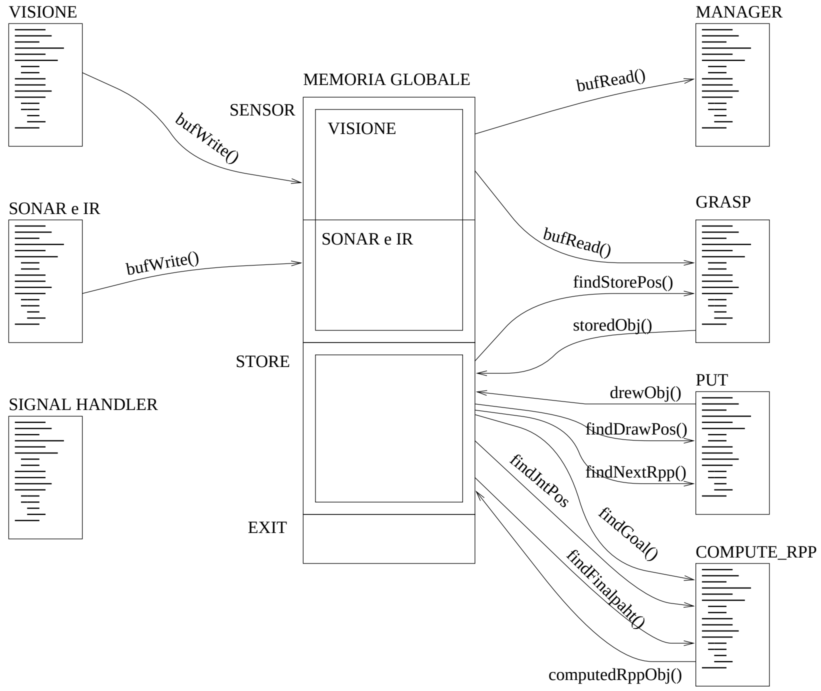


Figura 5.14: Funzioni utilizzate per l'accesso alla memoria globale

5.8 Relazioni tra i moduli di controllo

I moduli di controllo che realizzano l'applicazione si scambiano informazioni utilizzando una memoria globale. Attraverso le funzioni associate al modulo di gestione dei sensori il sistema di visione e i sensori di prossimità mantengono aggiornato il modello simbolico dell'ambiente. Dalla lettura di queste informazioni il manager potrà decidere quali azioni compiere.
I moduli di azione *grasp* e *put* ed il modulo di pianificazione aggiornano le informazioni sugli oggetti su cui si sta operando, modificando il contenuto della struttura *store*.
In fig. 5.15 è rappresentato un esempio di applicazione che l'architettura di governo è in grado di gestire.
La thread iniziale crea tutti i moduli di controllo necessari, quindi si sospende (1).
All'arrivo di un oggetto il modulo di controllo *manager* legge le informazioni ricavate dal modulo di visione quindi, poiché si verificano le condizioni per l'esecuzione del ciclo di azioni di *grasp* risveglia la thread collegata (2). Quando l'oggetto prelevato viene depositato nell'area di deposito tempora-

neo, il *manager* risveglia la thread che esegue il modulo del pianificatore (3). L'arrivo di nuovi oggetti porta all'esecuzione di nuove azioni di grasp solo se la zona di deposito temporanea non è già completa (4). In tal caso, se un percorso è già stato calcolato verrà eseguito il ciclo di azioni di *put* per liberare un posto (5).

5.9 Validazione sperimentale

In ultima istanza, una serie di esperimenti relativi all'esecuzione del compito descritto, ha consentito di validare l'architettura proposta. Particolare attenzione è stata data all'analisi dei tempi di esecuzione dei moduli di controllo periodici (tempi di elaborazione e rispetto della deadline imposta).
In questa fase sperimentale si è deciso di impiegare l'algoritmo *Rate Monotonic* [27]. Inizialmente i periodi dei moduli di controllo sono stati fissati tentando di massimizzare il valore del fattore di utilizzo, poi si è verificato il comportamento del sistema in codizioni operative di sovraccarico sempre piu' rilassate. Tutte queste prove sono state effettuate modificando in modo proporzionale i valori dei periodi imposti, quindi i rapporti di priorita' sono rimasti inalterati.
Per ognuno di questi esperimenti è stato calcolato il tempo di esecuzione del modulo sporadico *compute_rpp*, relativo al calcolo dell'algoritmo di pianificazione della traiettoria. In condizioni critiche, quando il sistema è marcatamente reattivo perché ai moduli sensoriali è riservato gran parte del tempo di CPU, il tempo necessario per il completamento di *compute_rpp* si innalza. Negli esperimenti finora compiuti purtroppo non si è riusciti a rilevare questo incremento del tempo. Poiché i tempi di varianza nei tempi di calcolo del percorso sono di qualche decina di secondi questo ha mascherato l'incremento del tempo dovuto alle condizioni di sovraccarico della CPU. Solo attraverso un'analisi statistica sará possibile verificare quanto la variazione dei parametri dei moduli periodici incide sulle prestazione dei moduli sporadici.

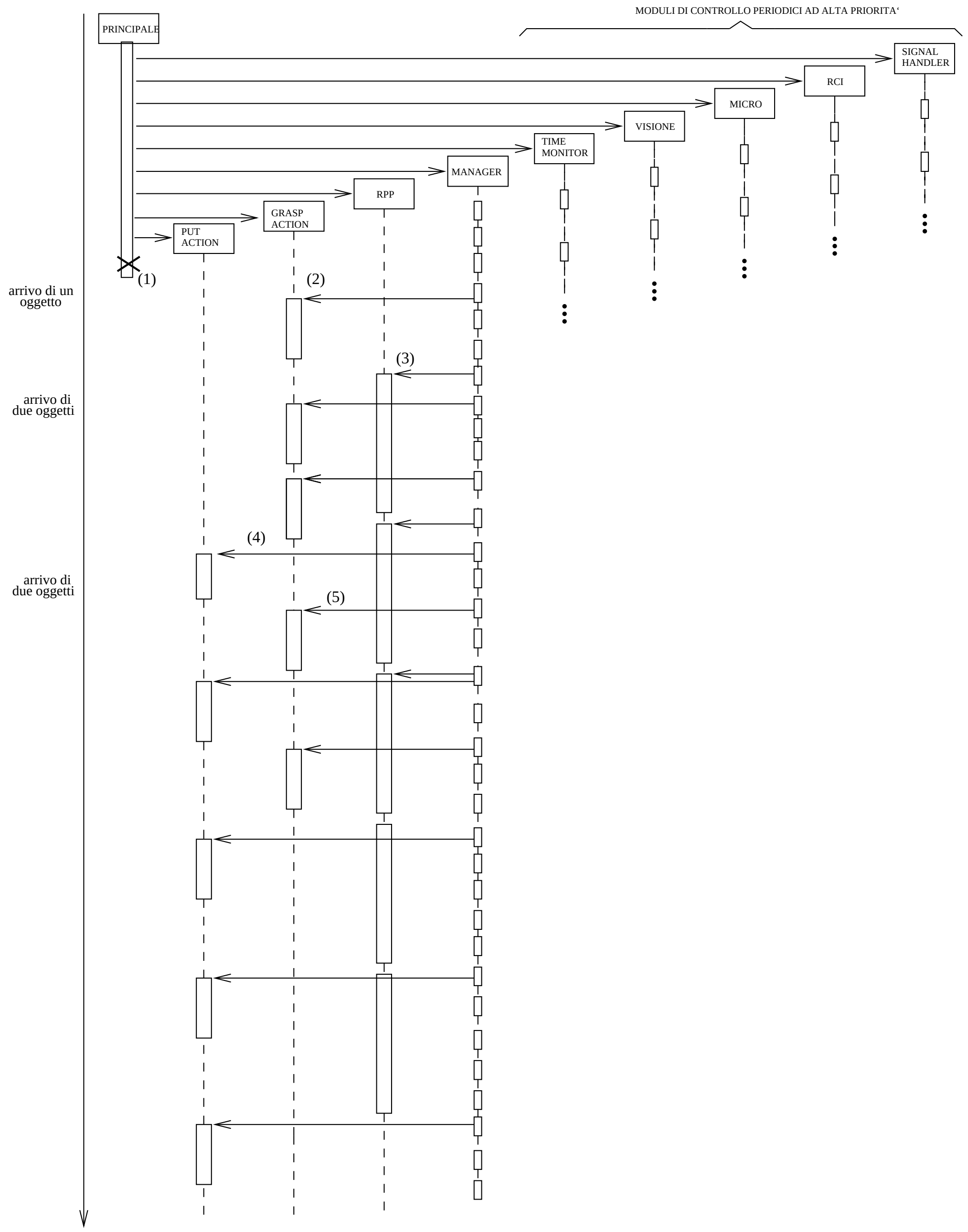


Figura 5.15: Relazioni che intercorrono tra i moduli di controllo in una situazione di esempio

Capitolo 6

Conclusioni

Questa tesi ha proposto un'architettura a tre livelli per il governo di un manipolatore robotico evoluto. L'architettura consente di realizzare una cooperazione efficace tra un sottosistema deliberativo ed un sottosistema reattivo. A partire dall'architettura è stato progettato e realizzato un sistema di controllo per un manipolatore robotico multisensoriale operante in un ambiente parzialmente non strutturato. Il sistema di controllo è dotato di proprietà di riconfigurabilità e grazie al livello di astrazione offerto, è in grado di esprimere tutte le azioni necessarie per il governo di un manipolatore, sollevando il progettista dai problemi di interfacciamento tra i moduli e di gestione del tempo.

Sfruttando le thread, le loro proprietà e gli strumenti di misura del tempo ad alta risoluzione offerti da Solaris 2.x, è stata costruita una libreria di moduli software che possono essere utilizzati per costruire i tre livelli della gerarchia del controllo. Tale libreria fornisce:

- processi periodici ed in grado di rispondere a vincoli temporali severi per realizzare il controllo delle parti elettromeccaniche, la lettura dei sensori e l'analisi dei dati del livello servo;
- processi aperiodici da utilizzarsi per le azioni di alto livello del livello supervisore;
- processi sporadici o aperiodici per il controllo delle prestazioni del sistema, per la simulazione e per la produzione di nuovi compiti da parte del livello di pianificazione.

L'integrazione di oggetti provenienti da questa libreria e la loro esecuzione in concorrenza consentono di costruire un'applicazione in grado di rispettare vincoli temporali.

Il sistema di controllo progettato è stato utilizzato per realizzare un'applicazione con sorgenti di incertezza, presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma.

Numerose sono le possibili estensioni per incrementare le funzionalità dell'architettura nel governo di un sistema robotico evoluto, in particolare:

- L'architettura attualmente si avvale di un livello di pianificazione molto semplificato. La progettazione e la realizzazione del livello di pianificazione potrebbe avvalersi del pacchetto software UM-PRS [28] (che realizza l'approccio PRS).

- Allo stato attuale le verifiche di schedulabilità sono eseguite staticamente sotto il controllo del progettista. È opportuno prevedere lo studio di un algoritmo di analisi di schedulabilità che sia in grado di determinare la possibilità di esecuzione da parte della CPU dei moduli di controllo proposti dal pianificatore, basandosi sulle informazioni di utilizzo di risorse e di tempo.

- Tutta la libreria è stata realizzata utilizzando i tipi di dati astratti offerti dal C. Un passo in avanti logico consiste nel realizzare il progetto con linguaggi orientati agli oggetti, come il C++ che dà il vantaggio dell'ereditarietà nella creazione di nuovi moduli di controllo che si basano su quelli definiti nella libreria.

- Lo scopo finale per la progettazione e la realizzazione di questi sistemi di controllo è di offrire la possibilità di realizzare a minor costo ed in tempi rapidi applicazioni per ambienti dinamici. Anche con il sistema di controllo migliore, possono però verificarsi degli errori. Quando questo accade, il sistema dovrebbe riuscire a diagnosticare la causa dell'errore per consentire all'utente azioni di recupero. Purtroppo, il debugging efficiente di un sistema in tempo reale rappresenta ancora un tema di ricerca per ora privo di soluzioni soddisfacenti.

- Nella libreria è presente un meccanismo per la cattura ed il recupero degli errori temporali. La scelta della funzione da richiamare in caso di errore dipende attualmente dalla sola thread che non rispetta la deadline. Un passo avanti potrebbe essere quello di rendere la scelta della funzione di recupero dipendente anche dallo stato dell'ambiente.

Appendice A

File di esempio per la riconfigurazione del sistema

A.1 mysample_driver.h

```
/* sample of automatically generated code for quickly developing
 * new sensor modules
 */

#ifndef _MYSAMPLE_DRIVER_H
#define _MYSAMPLE_DRIVER_H

/* Identifier of this sensor */
#define MYSAMPLE 2

/* Define here the information that the producer
 * write to the buffer each time that it calls
 * bufWrite
 */
typedef struct _mysample {
    /*** my sample information goes here ***/
}mysample;

void mysampleDr_startup(RT_THREAD *me, void *data);
void mysampleDr_ctrl(RT_THREAD *me, void *data);
void mysampleDr_release(RT_THREAD *me, void *data);

#endif
```

A.2 mysample_driver.c

```
/**** This is a sample for the sensor module
 * please replace mysample with sensor_name
 * and put your sensord-dependent code
 * where show by the XXX
 */

/* From the standard C library */
#include <string.h>
#include <sys/filio.h>
#include <stdio.h>
#include <sys/termios.h>
#include <thread.h>
#include <limits.h>
#include <fcntl.h>

/* For the use of priocntl */
#include <sys/lwp.h>
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

#include "list.h"
#include "rt_shell.h"
#include "sysfnx.h"

/* From the program library */
#include "defs.h"
#include "param_task.h"
#include "system_threads.h"
#include "buffer_shell.h"
#include "sysfnx.h"

#include "mysample_driver.h"

#define SIZE_MYSAMPLE_IMAGE sizeof(mysample)

/* Global variables: please use them only in this
 * unit.
 */

sensor *mysample_buffer_pointer;
```

```
/* Function for the use of buffer */
int mysample_data_write(mysample *image_pointer)
{
    /* sensor-dependent 'read' code from sensor
     * and
     * buffer-dependend 'write' code to buffer
     * goes here
     */

    /* Example:
     * image_pointer->fieldA = varA;
     * where fieldA is one of the element of the
     * structure that have to be initializated
     * at the value of varA read from the sensor
     */
}

/* XXX - driver_dependent function code goes here */

void mysampleDr_startup(RT_THREAD *me, void *data )
{
    thread_arg_type *arg;
    arg = data;

    /* XXX - thread_dependent startup code goes here */


    mysam-
ple_buffer_pointer=bufCreate(arg->sensors,MYSAMPLE,SIZE_MYSAMPLE_IMAGE);

}

void mysampleDr_ctrl(RT_THREAD *me, void *data)
{
    thread_arg_type *arg;
    arg = data;

    /* XXX - thread-dependent 'cycle' code goes here */

    bufWrite(mysample_buffer_pointer,mysample_data_write);

    if (arg->exit)
        me->state = RELEASE;
```

```
}
void mysampleDr_release(RT_THREAD *me, void *data )
{
    thread_arg_type *arg;
    arg = data;

    /* XXX - thread-dependent 'release' code goes here */

    bufDestroy(mysample_buffer_pointer,arg->sensors);
    printf("MYSAMPLE: exiting...\n");
    thr_exit((void *)arg->mysample);
}
```

Appendice B

Calibrazione della telecamera fissa

B.1 Introduzione

La calibrazione di una telecamera nel contesto della visione 3D è il processo che porta alla determinazione delle caratteristiche geometriche ed ottiche interne alla telecamera (*parametri intrinseci*) e/o della posizione 3D e dell'orientazione del sistema di coordinate della telecamera rispetto ad un sistema di coordinate di riferimento (*parametri estrinseci*).
vspace3mm
Gli scopi della calibrazione sono:

- *Ricavare delle informazioni 3D dalle coordinate in 2D delle immagini.* Le informazioni che solitamente si desidera ricavare sono la posizione degli oggetti, le loro caratteristiche e la posizione ed orientazione di una camera mobile rispetto ad un sistema di coordinate di riferimento.

- *Ricavare le coordinate delle immagini in 2D dalle informazioni 3D.* Questo e' utile quando si hanno applicazioni di visione che devono controllare il verificarsi di comandi impartiti ai sistemi robotici.

B.2 Trasformazioni di base

Le matrici di trasformazione che descritte brevemente in questo paragrafo consentono di porre in relazione le coordinate di un punto espresse nei diversi

sistemi di coordinate utilizzati [1].
Nel seguito si indicherà un generico punto, di coordinate (X, Y, Z), con la matrice $V = [X, Y, Z]^T$ e la sua trasformazione, di coordinate $(X*, Y*, Z*)$, con il vettore $V^* = [X^*, Y^*, Z^*]^T$.

B.2.1 Coordinate omogenee

Poiché una matrice 3x3 non fornisce nessuna indicazione per la trasformazione di scala, viene introdotta una quarta coordinata nel vettore posizione $V = [X, Y, Z]^T$, che diventa quindi $V_h = [wX, wY, wZ, w]^T$. Il vettore V_h è espresso in *coordinate omogenee.*
La rappresentazione in coordinate omogenee è utile per sviluppare trasformazioni matriciali che comprendano rotazione, traslazione, trasformazione di scala e trasformazioni prospettiche.
Il vettore fisico N dimensionale associato al vettore N+1 dimensionale, rappresentato in coordinate omogenee, si ottiene dividendo le coordinate omogenee per la coordinata (N+1)-esima. Quindi le componenti di V si ottengono dividendo le prime tre componenti di V_h per w, la quarta componente.

B.2.2 Matrice di traslazione

La traslazione di un punto di coordinate XYZ in una nuova posizione utilizzando gli spostamenti X_0, Y_0, Z_0 puo' essere espressa in forma matriciale, in coordinate omogenee, scrivendo:

$$\begin{bmatrix} X^* \\ Y^* \\ Z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

dove:

$$T = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

é la matrice di traslazione.

[1] Per una trattazione completa vedi [29] dalla pagina 35 alla pagina 58

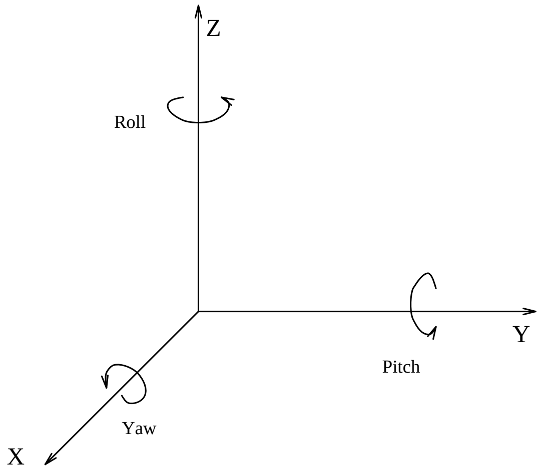


Figura B.1: Roll, pitch e yaw

B.2.3 Matrice di trasformazione di scala

La trasformazione di scala secondo i fattori S_x,S_y,S_z, lungo gli assi X,Y,Z è data dalla matrice di trasformazione:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

B.2.4 Matrice di rotazione

Una matrice di rotazione 4x4 trasforma le sue coordinate espresse in un sistema di coordinate rotazionale OUVW nelle coordinate espresse in un sistema di coordinate di riferimento OXYZ. La rotazione di un punto intorno all'asse coordinato Z di un angolo θ (roll) è espresso dalla matrice di rotazione:

$$R_\theta = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La rotazione di un punto intorno all'asse coordinato Y di un angolo β (pitch) è espresso dalla matrice:

$$R_\beta = \begin{bmatrix} cos\beta & 0 & sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\beta & 0 & cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Infine, la rotazione di un punto intorno al'asse coordinato X di una angolo α (yaw) è ottenuta utilizzando la matrice di trasformazione:

$$R_\alpha = \begin{bmatrix} 1 & 0 & 0 & \\ 0 & cos\alpha & -sin\alpha & 0 \\ 0 & sin\alpha & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

B.3 Trasformazione di prospettiva

Una trasformazione di prospettiva proietta punti 3D su un piano, consentendo di individuare quale sara' l'immagine di un mondo tridimensionale[2]. Facendo riferimento alla figura B.2 ed ai simboli in essa utilizzati e supponendo che tutti i punti abbiano $Z > 0$, si cercherà di trovare una trasformazione che, a partire da un punto di coordinate (X_0, Y_0, Z_0), individui quali sono le sue coordinate (x_0, y_0) nel sistema di riferimento della telecamera. Applicando le proprietà dei triangoli simili otteniamo:

$$\frac{x_0}{l} = -\frac{X_0}{Z_0 - l} = \frac{X_0}{l - Z_0}$$

$$\frac{y_0}{l} = -\frac{Y_0}{Z_0 - l} = \frac{Y_0}{l - Z_0}$$

. Quindi le coordinate sul piano immagine del punto 3D proiettato sono:

$$x_0 = \frac{lX_0}{l - Z_0}$$

$$y_0 = \frac{lY_0}{l - Z_0}$$

Anche in questo caso si può costruire una matrice che compie una trasformazione, in questo caso di prospettiva, quando e' applicata al vettore

[2]Per una trattazione completa vedi [29] alle pagine 394-401

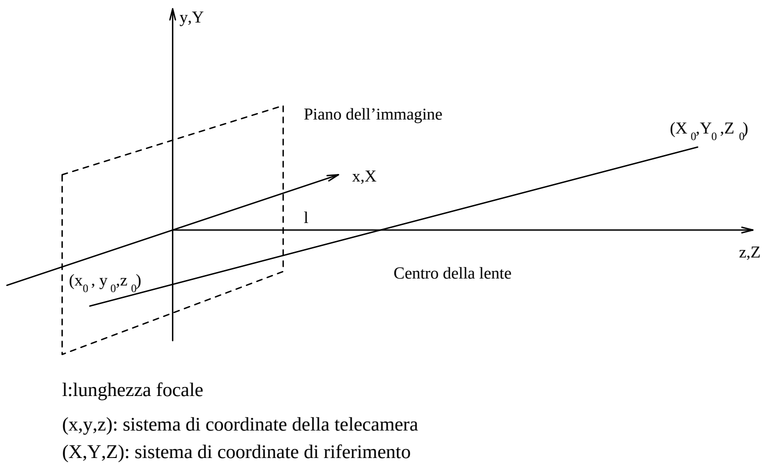


Figura B.2: Modello di base del processo di costruzione dell'immagine. Il sistema di coordinate della telecamera (x,y,z) è allineato al sistema di coordinate di riferimento (X,Y,Z)

$V_h = [wX_0, wY_0, wZ_0, w]^T$.
Denotando con C_h le coordinate omogenee del punto nell'immagine e con P la *matrice di trasformazione della prospettiva*, potremo scrivere:

$$C_h = PV_h = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{l} & 1 \end{bmatrix} \begin{bmatrix} wX_0 \\ wY_0 \\ wZ_0 \\ w \end{bmatrix} = \begin{bmatrix} wX_0 \\ wY_0 \\ wZ_0 \\ \frac{-wZ_0}{l} + w \end{bmatrix}$$

A questo punto risulta alquanto semplice individuare le coordinate cartesiane del punto che indicheremo con C:

$$C = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} \frac{lX_0}{l-Z_0} \\ \frac{lY_0}{l-Z_0} \\ \frac{lZ_0}{l-Z_0} \end{bmatrix}$$

Le prime due componenti del vettore C sono le coordinate (x_0, y_0) del punto sul piano immagine mentre la terza componente non è di alcun interesse.
In modo analogo e' possibile ricavare le formule che legano le coordinate 3D alle coordinate 2D del punto sul piano immagine.
Con passaggi simili ai precedenti si ottengono le equazioni:

$$X_0 = \frac{lx_0}{l+z}$$

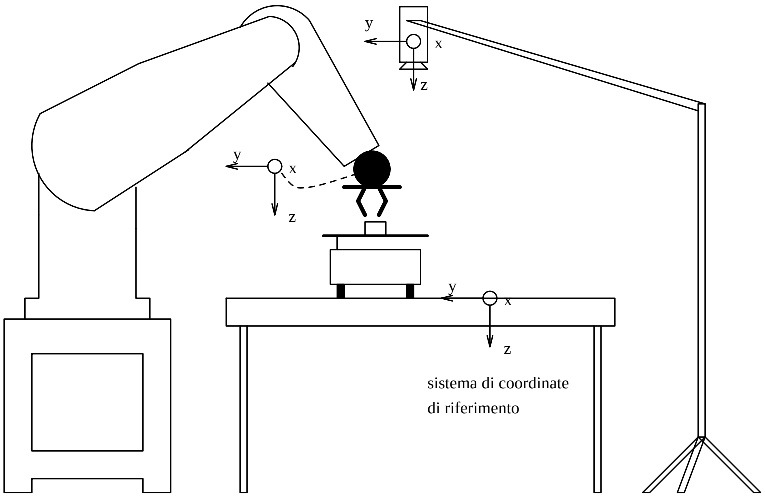


Figura B.3: La stazione di lavoro

$$Y_0 = \frac{ly_0}{l+z}$$

$$Z_0 = \frac{lz_0}{l+z}$$

Eliminando la variabile indipendente z_0 si ottengono le seguenti equazioni:

$$X_0 = \frac{x_0}{l}(l - Z_0)$$

$$Y_0 = \frac{y_0}{l}(l - Z_0).$$

Ovvia e‘ la conclusione che per ricostruire un punto 3D dalla sua immagine con la trasformazione prospettica inversa si richiede la conoscenza di almeno una delle coordinate del punto nel sistema di riferimento.

B.4 Stazione di lavoro

La nostra stazione di lavoro e‘ dotata di una telecamera monocromatica TC 500 [30] e di un robot a 6 giunti Unimate PUMA560.
La telecamera vede il robot e quindi può individuare l'origine del sistema di coordinate ad esso associato. Essa vede anche gli oggetti manipolati dal robot che si trovano sul tavolo.
Descriveremo ora le trasformazioni necessarie per portarsi dalle coordinate

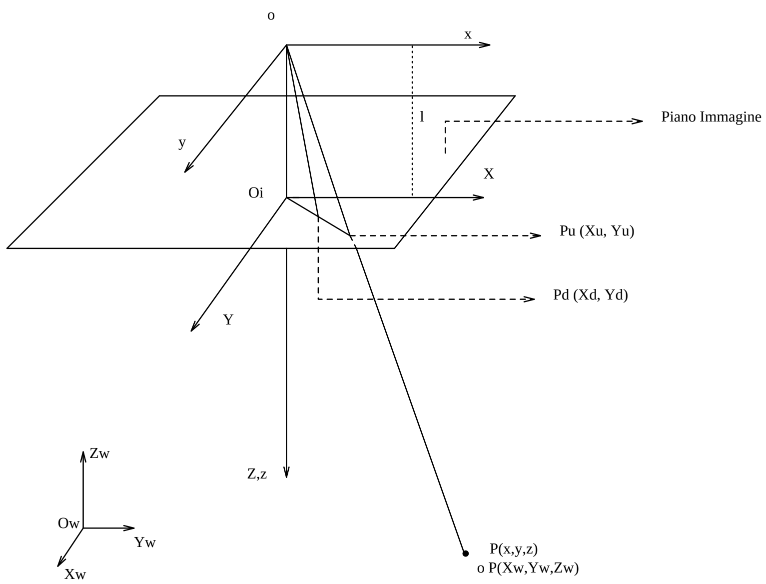


Figura B.4: Geometria della telecamera con la trasformazione di prospettiva e la distorsione radiale delle lenti. Indichiamo con (X_w, Y_w, Z_w) le coordinate del punto nel sistema di coordinate world frame, con (X, Y, Z) le coordinate del punto nel sistema di riferimento della telecamera, sistema che è centrato nel punto 0, con (X_u, Y_u) le coordinate del punto nel piano immagine passante per O_i se il modello della telecamera è senza distorsione, con (X_d, Y_d) le reali coordinate del punto sul piano immagine che differiscono da (X_u, Y_u) per la distorsione delle lenti

del punto nel piano immagine della telecamera alle coordinate del sistema di riferimento fondamentale del PUMA560.
Dobbiamo costruire due matrici di trasformazione: la prima che converta le coordinate del punto nel piano immagine in coordinate del sistema di coordinate associato al tavolo, che chiameremo *world frame*, la seconda che converta le coordinate del *world frame* nelle coordinate del sistema solidale al PUMA.

B.4.1 Il modello della telecamera

Per quanto riguarda la costruzione della prima trasformazione riprendiamo brevemente l'approccio di Roger Y. Tsai [31] e facciamo riferimento alla figura B.4.
Oltre alle coordinate indicate in figura è importante ricordare un'altra cop-

pia di coordinate (X_f, Y_f) per l'immagine discreta memorizzata nella frame memory espresse in numero di pixel (nel nostro specifico caso uno dei BANK della memoria esterna del DSP [32].
Descriviamo ora i passi da seguire:

- *dalle coordinate* (X_w, Y_w, Z_w) *alle coordinate della telecamera* (X, Y, Z)
La trasformazione è un semplice rototraslazione. I parametri da calibrare in questo passo sono i parametri estrinseci espressi dalle componenti della matrice R e T. Una volta trovate queste si applicano le operazioni tra matrici indicate nel primo paragrafo

- *dalle coordinate* (X, Y, Z) *alle coordinate* (X_u, Y_u) *ideali*
Utilizzando le proprietà dei triangoli simili nel caso di una telecamera ideale otteniamo:
$$X_u = l\frac{x}{z}$$
$$Y_u = l\frac{y}{z}$$
dove l è l'effettiva distanza focale e rappresenta il parametro da individuare in questo passo.

- *dalle coordinate* (X_u, Y_u) *alle coordinate distorte a causa della distorsione radiale* (X_d, Y_d)
La distorsione radiale e' rappresentabile con due coefficienti (D_x, D_y) che sommati alle coordinate ideali mi danno le coordinate distorte:
$$X_d + D_x = X_u$$
$$Y_d + D_y = Y_u$$
I coefficienti D dipendono a loro volta dai coefficienti di distorsione κ_i secondo le seguenti equazioni:
$$D_x = X_d(\kappa_1 r^2 + \kappa_2 r^4 + ...)$$
$$D_y = Y_d(\kappa_1 r^2 + \kappa_2 r^4 + ...)$$
$$r = \sqrt{X_d^2 + Y_d^2}$$
Ci sono due tipi di distorsione: radiale e tangenziale. Per ogni distorsione sarebbe necessaria una serie infinita. È comunque sufficiente considerare solo la distorsione radiale e di questa un solo termine per ottenere ottimi risultati. Questo termine rappresenta il parametro da individuare in questo passo della calibrazione

- *dalle coordinate reali* (X_d, Y_d) *alle coordinate dell'immagine* (X_f, Y_f)
  La trasformazione da utilizzare è la seguente:

$$X_f = s_x d_x'^{-1} X_d + C_x$$

$$Y_f = d_y'^{-1} Y_d + C_y$$

$$d_x' = d_x \frac{N_{cx}}{N_{fx}}$$

dove:

(C_x, C_y) numero della riga e della colonna del centro dell'immagine nella frame memory

d_x distanza da centro a centro di due elementi sensori adiacenti nella direzione X

d_y distanza da centro a centro di due elementi sensori adiacenti nella direzione Y

N_{cx} numero di elementi sensori nella direzione X

N_{fx} numero di pixel in una linea come campionato dal computer

I precedenti valori se non vengono esplicitamente forniti dal costruttore possono essere ricavati sperimentalmente. Il parametro che deve invece essere obbligatoriamente calibrato in questa fase è il fattore di scala s_x utilizzato per eliminare una certa incertezza sui valori di d_x e d_y dovuta alla scansione dell'immagine e ad errori sui tempi di acquisizione[31].

Combinando tutte queste trasformazioni è possibile arrivare a due equazioni che relazionano le coordinate (X_f, Y_f) alle coordinate 3D dell'oggetto nel sistema di coordinate della telecamera e, con una semplice rototraslazione, alle coordinate dell'oggetto nel world frame.
vspace3mm
Un modo molto più semplice per calibrare la telecamera è utilizzare tecniche che ricercano un insieme di parametri (es. coefficienti di una matrice di trasformazione) con delle equazioni lineari, ignorando la dipendenza tra i parametri.
Per utilizzare questi metodi è necessario conoscere un insieme di punti immagine le cui coordinate di riferimento siano note.
Supponiamo di indicare con $C_h = [c_{h1}, c_{h2}, c_{h3}, c_{h4}]^T$ le coordinate omogenee

di un punto nel piano immagine della telecamera e con $V_h = [X, Y, Z, 1]^T$ le coordinate dello stesso punto nel sistema di riferimento.
Possiamo quindi scrivere:

$$\begin{bmatrix} c_{h1} \\ c_{h2} \\ c_{h3} \\ c_{h4} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Poichè per le proprietà delle coordinate omogenee le coordinate della telecamera in forma cartesiana sono date da:

$$x = \frac{c_{h1}}{c_{h4}}$$

$$y = \frac{c_{h2}}{c_{h4}}$$

sviluppando i calcoli è possibile ottenere le due seguenti equazioni:

$$a_{11}X + a_{12}Y + a_{13}Z - a_{41}xX - a_{42}xY - a_{43}xZ - a_{44}x + a_{14} = 0$$

$$a_{21}X + a_{22}Y + a_{23}Z - a_{41}yX - a_{42}yY - a_{43}yZ - a_{44}y + a_{24} = 0$$

che presentano dodici coefficienti incogniti.
Per calibrare la telecamera é quindi sufficiente individuare un numero di punti maggiore di 6 di coordinate note (X_i, Y_i, Z_i), ricavarne le corrispondenti coordinate nel piano immagine della telecamera (x_i, y_i) e usare questi valori nelle equazioni precedenti per determinare i parametri incogniti.

B.4.2 Il modello del Puma

La seconda trasformazione consiste in una semplice rototraslazione con cambiamento di scala.
Nella figura B.5 viene messo in evidenza un *sistema di coordinate di base per il manipolatore.* Tutti i successivi spostamenti del Puma utilizzano come base questo sisema di riferimento.
Generalmente per semplicità si introduce una transizione[3] p:

```
p=allocTransRot(NULL,UNDEF,P_X,P_Y,P_Z,xunit,180.0);
```

con:

[3] Per i termini e la semantica RCCL vedi [33]

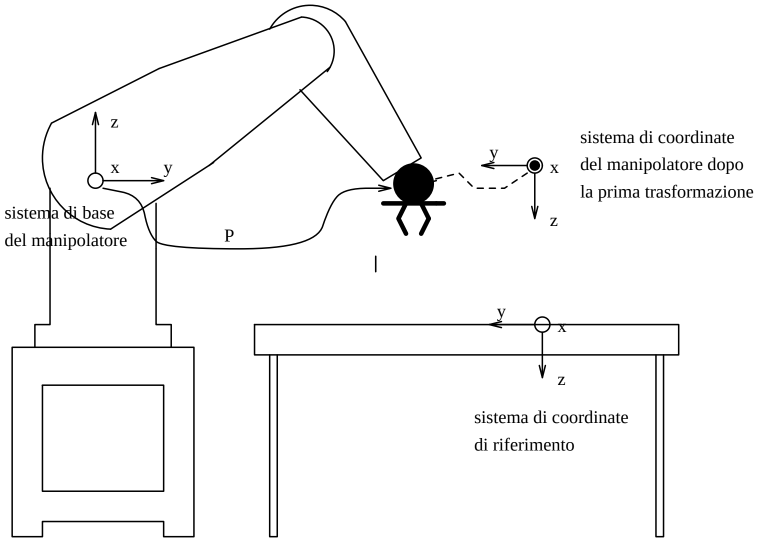


Figura B.5: Il sistema di riferimento del Puma

```
#define P_X     149.09
#define P_Y     597.187
#define P_Z     -15.2664
```

Questa transizione comporta una traslazione del sistema di riferimento fino al centro del tool quindi una rotazione intorno all'asse x di 180^0.
Dall'applicazione della transizione precedente si ottiene il secondo sistema di coordinate che risulta più intuitivo da utilizzare per gli spostamenti del Puma.
Una volta individuato questo secondo sistema di riferimento è necessario riuscire a determinare i parametri (rappresentati dagli elementi di una matrice traslazione T e di una matrice rotazione R e da un fattore di cambiamento di scala) che consentano di passare dalle coordinate del world frame alle coordinate del sistema di coordinate del tool del Puma.

B.5 L'attuale tecnica di calibrazione

La possibilità di avere un sistema di coordinate di riferimento world frame solidale con il tavolo è utile perchè in questo modo si spezza il processo di calibrazione in due parti: calibrazione del Puma rispetto al world frame e

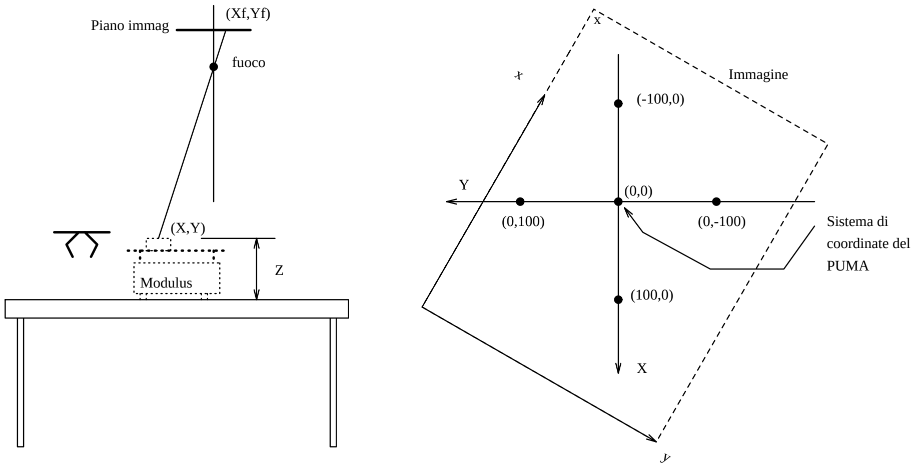


Figura B.6: Procedimento di calibrazione implementato

calibrazione della telecamera rispetto al world frame.
Questo è utile perchè, quando si sposta la telecamera, si deve procedere alla sola ricalibrazione del suo sistema di coordinate rispetto a quello del tavolo. Questa maggiore praticità richiede però uno sforzo implementativo superiore. Poichè la calibrazione della telecamera riveste nella mia tesi un carattere secondario, quello che è stato implementato è un metodo molto più semplice e con risultati peggiori, ma soddisfacenti per le esigenze del task.
Nella calibrazione vengono considerati solo il sistema di coordinate dell'immagine che si forma nella frame memory (indicheremo le coordinate con (X_f, Y_f) ed il sistema di coordinate solidale al tool (indicheremo queste coordinate con (X, Y, Z)).
Sappiamo dal paragrafo sulla trasformazione di prospettiva che per ricostruire un punto 3D dalla sua immagine con la trasformazione prospettica inversa si richiede la conoscenza di almeno una delle coordinate del punto nel sistema di riferimento.
Poichè gli oggetti sono sempre alla medesima distanza dal piano del tavolo si ignorerà il problema della determinazione di Z, considerata già nota, e si cercherà di ricostruire dalle coordinate (X_f, Y_f) le sole coordinate X e Y . Per individuare la corrispondenza tra i due sistemi di coordinate si fa muovere il Puma in cinque punti di coordinate (X, Y) conosciute (individuati da • nella figura B.6) e si trovano le corrispondenti coordinate (X_f, Y_f). Con qualche semplice operazione è ora possibile ottenere i parametri di traslazione, rota-

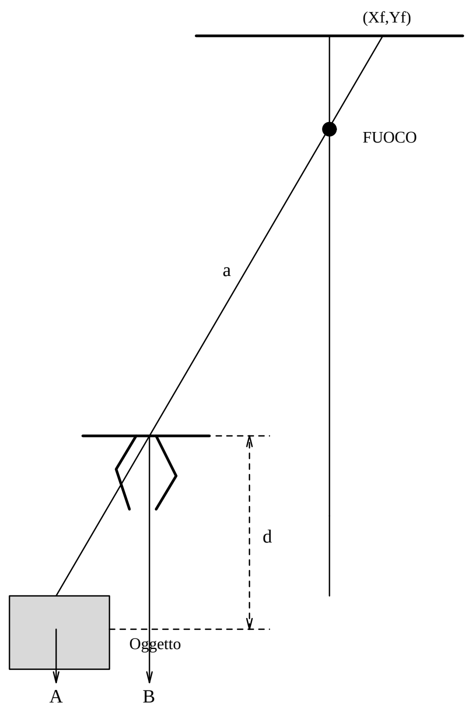


Figura B.7: Al punto (X_f, Y_f) corrispondono tutti i punti che appartengono alla linea a. Se calibro facendo muovere il tool ad una distanza d rispetto all'altezza dell'oggetto, un oggetto con baricentro A farà muovere il puma sul punto B con conseguente fallimento della presa

zione e cambiamento di scala che consentono di trasformare le (X_f, Y_f) in (X, Y).
Si deve prestare attenzione all'altezza del tool rispetto al tavolo: poichè non si è calibrata l'altezza Z, anche pochi centimetri di differenza rispetto al valore predefinito darebbero un errore di qualche cm nella presa dell'oggetto (vedi fig. B.7).

Per individuare dove si trova il tool del Puma, quando esso viene portato nei cinque punti di coordinate note, é stato applicato un marker sul tool di 2 cm di diametro (figura B.8) a una distanza di 5cm dal centro del gripper. L'individuazione del marker tra gli oggetti presenti in un immagine richiede come operazione preliminare la determinazione dell'altezza della telecamera. Per questa operazione si è costruito un marker rappresentante un cerchio di 20 cm (fig. B.8) quindi si è proceduto ad una determinazione sperimentale a diverse altezze della dimensione in pixel dell'immagine del marker.

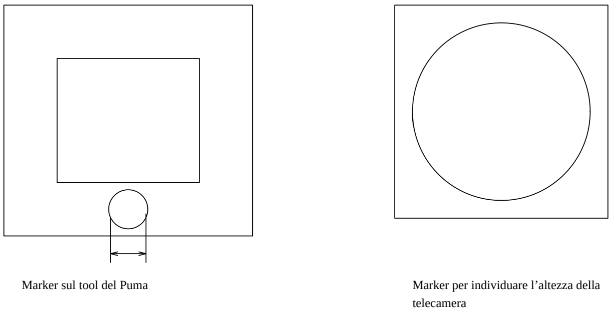


Figura B.8: Forma e posizione dei marker utilizzati

Con i dati ottenuti si è proceduto al calcolo della regressione cubica (fig. B.9) poichè la relazione tra pixel ed altezza non è lineare) ottenendo la seguente equazione che mi restituisce l'altezza della telecamera (y) in funzione delle dimensioni in pixel del marker (x):

$$y = 356.14611 + -4.48450327x + 0.0239406429x^2 + (-4.59540627e - 05)x^3$$

Con lo stesso procedimento è possibile individuare una equazione cubica che esprima a quanti pixel corrisponde un cm (y) in funzione dell'altezza della telecamera (x):

$$y = 28.7736831 + -0.584003673x + 0.00502810843x^2 + (-1.55743592e - 05)x^3$$

Dalla precedente equazione risulta semplice individuare quale tra gli oggetti inquadrati dalla telecamera ha le dimensioni in pixel corrispondenti a 2 cm.

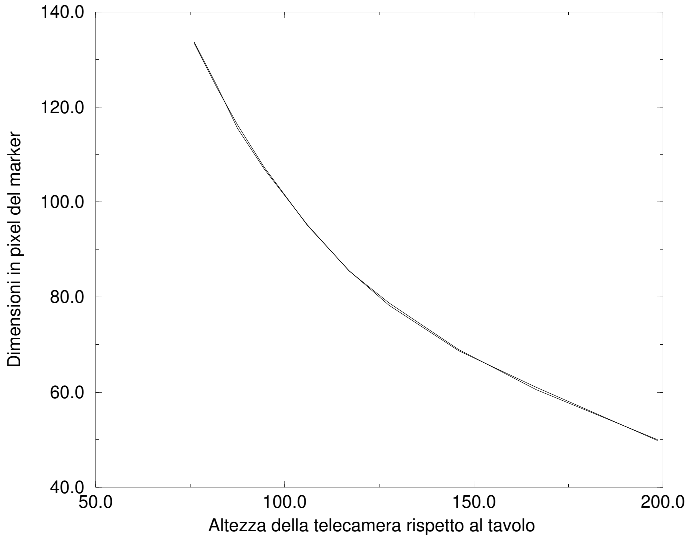


Figura B.9: Interpolante dell'altezza della telecamera

Bibliografia

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

[2] Chapman D. Agre P.E. Pengi: an implementation of a theory of activity. *Proceedings of the 6th National Conference on Artificial Intelligence, San Matteo, CA*, 1987.

[3] C.S.G.Lee K.S. Fu, R.C. Gonzalez. *ROBOTICS: Control, Sensing, Vision, and Intelligence*. Industrial Engineering Series. McGRAW HILL INTERNATIONAL EDITIONS, 1987.

[4] S. Fleury M. Herrb R. Chatila. Design of a modular architecture for autonomous robot. *IEEE Int. Conf. on R.A., San Diego*, May 1994.

[5] K. G. Shin D. J. Musliner, E. H. Durfee. CIRCA: a cooperative intelligent real time control architecture. *IEEE Transactions on systems, man, and cybernetics*, 23(6), 1993.

[6] John A. Stankovic. "Misconceptions about Real-Time computing: a serious problem for next-generation systems". *IEEE Computer Magazine*, Vol. 21(No. 10), 1988.

[7] D. J. Musliner J. A. Hendler E. H. Durfee J. K. Strosnider C. J. Paul. The challenges of Real-Time AI. *IEEE Computer*, January 1995.

[8] J.K. Strosnider and C.J.Paul. A structured view of real time problem solving. *AI Magazine*, Summer 1994.

[9] F.F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: a high level supervision and control language for autonomous mobile robots.

Proceedings of the 1996 IEEE international conference on robotics and automation, Minneapolis, Minnesota, april 1996.

[10] R. A. Brooks. A robust level control system for a mobile robot. *IEEE Journal of robotics and automation*, 7(1), 1986.

[11] N. J. Nilsson. Shakey the Robot. *SRI A. I. Center Techinal Note 323*, April 1984.

[12] P.S. Rosenbloom J. E. Laird, A. Newell. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33, 1987.

[13] A. Agrawala J. Hendler. Mission critical planning: AI on the MARUTI real-time operating system. *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, November 1990.

[14] J. Connell. SSS: A hybrid architecture applied to robot navigation. *Proc. IEEE int'l Conf. on Robotics and Automation*, 2719-2724 1992.

[15] E. Gatt. Integrating Reaction and Planning in a Heterogeneous Asychornous Architecture for Mobile Robot Navigation. *Proc. of AAAI Symposium on Integrated Intelligent Architectures, ACM Sigart Bulletin*, August 1991.

[16] D. M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transaction on Robotics and Automation*, June 1993.

[17] D. M. Lyons A. J. Hendriks. Using perception to plan incremental adaptations. *Proc. IEEE int'l Conf. on Robotics and Automation*, 855-860 1994.

[18] M. Schoppers. A software architecture for hard real-time execution of automatically synthesized plans or control laws. *Conf. on Intelligent Robotics in field, factory, service, and space*, March 1994.

[19] S.i R. Kleiman S. Barton D. Shah D. Stein M. Weeks M. L. Powell. SunOS multi-thread architecture. *Usenix*, Winter 1991.

[20] D.B. Steward. Real time software design and analysis of reconfigurable multi sensor based systems. Technical report, Carnagie Mellon University, 1994.

[21] B. Lewis D.J. Berg. *A guide to multithreaded programming: threads primer*. Sunsoft Press, 1996.

[22] J. Barraquand J. C. Latombe. Robot motion planning: a distributed representation approach. *The Int. J. of Robotics Research*, December 1991.

[23] T.Y.Li, J.Barraquand, and J.C.Latombe. "3D Randomized Path Planner". Technical report, Robotics Laboratory, Department of Computer Science, Stanford University, 1989.

[24] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, 1994.

[25] A. Bigi. Integrazione del sistema di visione DVS1000 nell'architettura di governo di un robot manipolatore. Technical report, Università degli Studi di Parma, 1995.

[26] A. Alessandretti. Algoritmi paralleli per la pianificazione dei movimenti di sistemi robotici. *Università degli Studi di Parma*, Luglio 1997.

[27] James W. Layland C.L. Liu. Scheduling algorithms for multiprograming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[28] E. H. Durfee M. J. Huber J. Lee P. Kenny. *UM-PRS V3.0 Programmer and User Guide.* Department of Electrical Engineering and Computer Science Univesity of Michigan.

[29] K. S. Fu R. C. Gonzalez C.S. G. Lee. *ROBOTICA.* Mc Graw-Hill, 1989.

[30] Datalogic. *TC 500 Operator's Manual.*

[31] R. Y. Tsai. A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Camera and Lenses. *IEEE Journal of Robotics and Automation*, 3(4), 1987.

[32] Datalogic. *SDP-01 DEVELOPMENT SYSTEM Programmer's Manual.*

[33] J. Lloyd and V. Hayward. *Multi-RCCL User's Guide.* Computer Vision and Robotics Laboratory McGill Research Centre for Intelligent Machines McGill University.