Capitolo 4

Componenti Software

Il programmi di controllo delle applicazioni di *visual servoing* sviluppate in questa tesi sono stati scritti nel linguaggio C++ e realizzati all'interno dell'ambiente di sviluppo Visual C 6.0.

Questi codici sono stati studiati per essere eseguiti sotto il sistema operativo *Windows NT*. In questo tipo di applicazioni dove il fattore velocità è molto importante, bisogna sfruttare al massimo le potenzialità dell'hardware che costituisce l'intero sistema. L'utilizzo di macchine basate su sistemi operativi *non Real Time*, come ad esempio NT, richiede una buona conoscenza della struttura del sistema operativo su cui si opera se non si vuole incorrere in un drastico calo delle prestazioni.

4.1 Windows NT

Windows NT è un sistema operativo (S.O) multitasking e timesharing di tipo *"preemptive"* a 32 bit concepito per i moderni microprocessori e compatibile con diverse architetture, la sua struttura e detta a *micro kernel* ovvero il codice eseguito in modalità kernel è stato ridotto al minimo e tutte le funzionalità più complesse sono state implementate come processi eseguiti in modalità utente. Il *kernel* è orientato agli oggetti: un *tipo di oggetto* in NT è un tipo di dato definito dal sistema che ha un insieme di attributi (valori dei dati) e possiede un insieme di metodi (cioè funzioni o operazioni). Un oggetto è semplicemente un'istanza di un specifico tipo di oggetto.

In NT tutte le risorse sia fisiche che logiche (ad esempio memoria di massa e file) sono rappresentate sotto forma di oggetti. Questo comporta un'implemen-

tazione più elaborata e codici in generale più voluminosi perché per l'utilizzo di questi oggetti è necessario definire sia la struttura dati interna che rappresenta la risorsa sia una interfaccia con l'esterno che determina le modalità (*metodi*) con cui è possibile manipolare l'oggetto.

Tutto questo però aumenta la stabilità del sistema perché ogni oggetto sarà utilizzato solo nelle maniere consentite dai suoi *metodi* e la sua struttura dati interna che rappresenta lo stato della risorsa, rimmarrà nascosta e quindi protetta da utilizzi non consentiti (*data hidden*).

Thread

Come molti altri moderni sistemi operativi, NT associa la nozione di processo al codice eseguibile. Ogni processo non è niente altro che una struttura dati che schematizza il *contesto*[1] di esecuzione del codice. In realtà l'unità minima di esecuzione è rappresentata dal *"thread"*, ogni processo è formato da uno o più thread che condividono lo stesso spazio degli indirizzamenti definito dal processo che li contiene. Lo *scheduling* dei thread è caratterizzato da una struttura a 31 livelli di priorità dinamica ed ogni livello è gestito a *Round Robin* cioè, scaduto il quanto di tempo del thread correntemente in esecuzione, il sistema operativo riprende il controllo (come anticipato NT è un S.O di tipo preemptive) e dealloca il thread congelandone il contesto di esecuzione[2], dopo di che verrà caricato il prossimo thread della lista dei thread pronti ad eseguire.

Mentre quindi più processi operano in un sistema ad *ambiente locale* (ovvero ogni processo ha un proprio spazio degli indirizzamenti, un proprio stack e proprie variabili) i thread all'interno di ogni singolo processo eseguono in un *ambiente globale* cioè condividono tutti le stessa area dati e codice (determinate dal processo che li contiene).

Questo tipo di struttura se da un lato aumenta la velocità dello scheduling dall'altro introduce problemi di *competizione* e *cooperazione* per le risorse che, se non risolti, portano a svariati tipi di *deadlock*.

Windows NT rende disponibili alcuni strumenti sia per gestire lo scheduling

[1] Vedere in [1].

[2] Vedi *"Context switching"* in [1], questa operazione è molto più veloce quando applicata a thread dello stesso processo perché le informazioni da aggiornare sono pochissime dato che molte strutture dati sono in comune e quindi non necessitano di essere aggiornate.

dei thread (cambio priorità e deallocazione thread controllata tramite la funzione *Sleep*) sia per evitare problemi di deadlock (sezioni critiche e semafori).

4.1.1 Organizzazione dei thread

L'architettura software dei sistemi di asservimento visuale sviluppati in questa tesi è costituita da un unico processo, diviso in tre thread:

1. thread di esecuzione del *codice di controllo* (vedere figura 4.4): è il thread principale e si incarica oltre che dell'inizializzazione del sistema e della messa in esecuzione degli altri thread anche della sincronizzazione di tutte le parti.

2. thread di *lettura* (vedere figura 4.4): si incarica della lettura dei caratteri che arrivano dalla seriale (ovvero dal controllore UNIVAL). Questo thread è quasi sempre deallocato a causa della grande lentezza del collegamento seriale.

3. thread di *esecuzione comandi*: organizza l'esecuzione dei comandi lanciati dal thread principale e si preoccupa quindi della trasmissione di questi (tramite il collegamento seriale) al controllore UNIVAL (vedere il paragrafo 4.5).

Altri thread vengono lanciati durante l'utilizzo della libreria Mil-Lite (vedere il paragrafo 4.2) per l'acquisizione dell'immagine dal frame grabber ma la loro gestione è completamente nascosta al programmatore.

4.1.2 API Application programming interface

API è il nome dato da Microsoft alle librerie di sistema, quell'insieme di funzioni che formano l'interfaccia tra il livello utente ed il codice eseguito a livello kernel. In questa tesi si sono utilizzate le seguenti funzioni API[3]:

1. ```
   HANDLE CreateThread(
       PSECURITY_ATTRIBUTES psa,
       DWORD cbStack,
       PTHREAD_START_ROUTINE pfnStartAddr,
   ```

[3]Per avere informazioni dettagliate sui parametri passati vedere [2].
   ```

```
PVOID pvParam,
DWORD fdwCreate,
PDWORD pdwThreadID);
```

crea un Thread eseguendo la routine puntata da `pfnStartAddr` passando come parametro la variabile puntata da `pvParam` ritorna la Handle che identifica il thread appena creato.

2. `Sleep(n);`

   interviene nello scheduling sospendendo il thread chiamante per n millisecondi[4], se $n = 0$ forza semplicemente la deallocazione del thread chiamante per rilasciare le risorse di calcolo ad altri thread.

3. Una *sezione critica* è una piccola porzione di codice che andando ad alterare risorse condivise da più thread (ad esempio variabili globali) necessita di essere acceduta in maniera mutuamente esclusiva per impedire effetti collaterali non predicibili sulla gestione della risorsa stessa[5].

   ```
   VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
   ```

   inizializza la struttura dati `pcs` di tipo `CRITICAL_SECTION`.

4. `VOID EnterCriticalSection(PCRITICAL_SECTION pcs);`

   definisce l'entrata del thread nella sezione critica `pcs` all'interno della quale si agirà sulla risorsa, se la risorsa è già occupata da altri thread, quello chiamante si sospende finchè la risorsa non è stata liberata cioè finchè nessun thread è dentro una sezione critica di nome `pcs`.

5. `VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);`

   il thread chiamante indica l'uscita da una sezione critica. Se ci sono thread sospesi in attesa della liberazione della sezione critica `pcs` ne viene liberato uno.

[4] Non essendo NT un S.O. Real Time il numero di millisecondi effettivamente trascorsi non è garantito.

[5] Vedere [1, 2].

4.2 La libreria MIL-lite

MIL-lite è un sottoinsieme della libreria MIL (Matrox Imaging Library) e conserva di questa tutte le funzionalità di acquisizione manipolazione e visualizzazione di immagini, sono assenti invece tutte le funzionalità inerenti la visione artificiale che in questa tesi verranno espletate dalla libreria *IMEL*.

La MIL gestisce ogni oggetto fisico, per esempio un frame grabber od un display, come se fosse un oggetto virtuale. Dichiarando con opportune funzioni tutti i dispositivi si definisce l'ambiente (virtuale) in cui si opera, per cui ogni processo dovrà definire prima di tutto la propria *applicazione MIL* (*Application*) la quale poi sarà in grado di gestire uno o più sistemi (*system*) contenenti a loro volta diversi dispositivi (virtuali) come *buffer dati*, *display* e *dagitizers* opportunamente dichiarati (vedere figura 4.1).

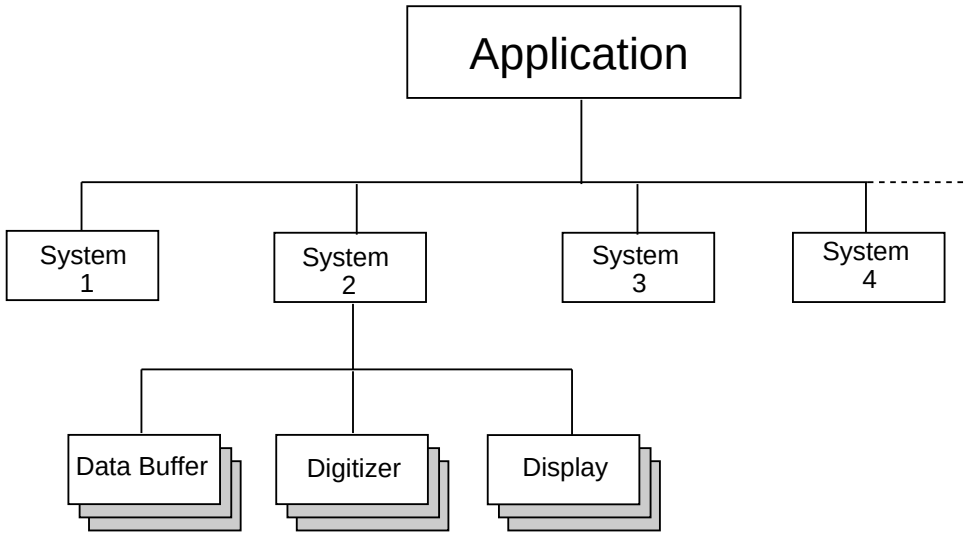

Figura 4.1: Struttura di una applicazione MIL

Di seguito si tratteranno le principali funzioni della libreria MIL-Lite utilizzate nella realizzazione di questa tesi[6] sottolineandone, più che la sintassi di utilizzo, la funzione logica.

1. `MIL_ID MappAlloc(InitFlag,ApplicationIdPtr);`

   crea un dispositivo virtuale *"applicazione"* associato al puntatore `ApplicationIdPtr` di tipo `MIL_ID`.

[6] Per una trattazione più esaustiva si rimanda ai manuali di riferimento [7, 8]

2. `MIL_ID MsysAlloc(SystemNamePtr,SystemNum,InitFlag, SystemId);`

   crea un dispositivo virtuale *"sistema"* che sarà identificato dal puntatore `SystemId`.

3. `MIL_ID MdigAlloc(SystemId,DigNum,DataFormat,InitFlag, DigIdPtr);`

   crea un dispositivo virtuale *digitalizzatore* identificato dal puntatore `DigIdPtr` ed appartenente al *sistema* `SystemId`. Il parametro `DataFormat` indica le modalità con cui vengono elaborate dal frame grabber le immagini provenienti dalla telecamera, queste configurazioni vengono descritte in un file con estensione DCF (digitizer configuration format) generato da un software dedicato.

4. `MIL_ID MdispAlloc(SystemId, DispNum, DispFormat, InitFlag, DisplayIdPtr);`

   crea un dispositivo virtuale di tipo *display* atto a visualizzare le immagini elaborate dal frame grabber, tale display è identificato dal puntatore `DisplayIdPtr`.

5. `MIL_ID MbufAlloc1d(SystemId, SizeX, Type, Attribute, BufIdPtr);`

   alloca nel sistema `SystemId` un buffer unidimensionale di lunghezza `SizeX` identificato dal puntatore `BufIdPtr`. In questa maniera verrà generata la mappa della **Look Up Table**, struttura che in questa tesi è stata utilizzata per generare una immagine a soglia (cioè in bianco e nero) da un'immagine a 256 sfumature di grigio[7].

6. `MIL_ID MbufAlloc2d(SystemId, SizeX, SizeY, Type, Attribute, BufIdPtr);`

   alloca nel sistema `SystemId` un buffer di dimensioni `SizeX` e `SizeY` identificato dal puntatore `BufIdPtr`.

   - '*Type*': specifica il tipo dei dati di cui è formato il buffer. In Tabella 4.1 sono riportati i valori possibili.

[7] Vedere [16] pag. 29 e manuali Mil-Lite [8, 7].

Tipi di dato	Descrizione
M_SIGNED	Array di interi con segno
M_UNSIGNED (Default)	Array di interi senza segno
M_FLOAT	Array di float

Tabella 4.1: Valori del parametro *Type*

- '*Attribute*': specifica gli attributi del buffer. In Tabella 4.2 sono riportati i valori di utilizzo piú frequente.

Attributi	Dati contenuti
M_IMAGE	Immagine
M_LUT	Look Up Table
M_ARRAY	Generico
Utilizzi	**Descrizione**
M_DISP	Consente la visualizzazione
M_GRAB	Immagine catturata
M_PROC	Consente la modifica

Tabella 4.2: Valori del parametro *Attribute*

7. `void MgenLutRamp(LutId, StartIndex, StartValue, EndIndex, EndValue);`

   funzione che genera una mappatura a rampa o a soglia della *Look Up Table* puntata da `LutId`.

8. `long MbufInquire(BufIdPtr, InquireType,UserVarPtr);`

   ritorna il valore del parametro `InquireType` del buffer `BufIdPtr`. Questa funzione può essere utilizzata per poter ottenere l'indirizzo in cui è mappato il buffer dell'immagine catturata dalla telecamera ponendo

   `InquireType=M_HOST_ADDRESS`

   in questo modo si può attuare un accesso diretto alla memoria aumentando la velocità dell'elaborazione dell'immagine, oppure con

   `InquireType=M_PITCH`

   si possono ricavare le dimensioni dell'immagine catturata.

9. `void MdigControl(DigId, ControlType, Value);`

   questa funzione permette di configurare le modalità di acquisizione della immagine da parte del *digitalizzatore* puntato da `DigId`, tra queste modalità si trovano ad esempio la dimensione dell'immagine acquisita (`M_GRAB_SCALE`), oppure, funzione molto importante, il campionamento asincrono delle immagini che giungono dal frame grabber

   `ControlType=M_GRAB_MODE` e `Value=M_ASYNCHRONOUS`

   caratteristica che permette di elaborare una immagine mentre la successiva viene acquisita[8], in questa maniera è possibile ottenere un considerevole incremento della velocità di elaborazione del sistema.

4.3 La libreria "IMEL"

Una volta utilizzata la libreria MIL-lite per acquisire l'immagine (ripresa dalla telecamera) attraverso il frame-grabber ed avendone creato una opportuna versione binarizzata attraverso una *look up table*[9], si passa all'utilizzo della libreria IMEL[10] (Elaborazione Immagine) che implementa alcuni degli algoritmi di visione artificiale utili ad un sistema di *visual servoing*. I principali compiti svolti da questa libreria sono:

- **Etichettamento degli oggetti (Labeling)**

  Conteggio del numero di oggetti presenti nell'immagine.

- **Estrazione delle caratteristiche**

  Per ogni oggetto vengono calcolate la posizione, l'area, gli estremi ed eventualmente l'orientamento.

- **Filtraggio degli oggetti**

  Eliminazione di oggetti che hanno una'area piccola.

L'utilizzo di questa libreria parte dalla definizione di un oggetto ad esempio chiamato *image* di tipo `Imel`

[8] Una immediata applicazione di questa funzionalità è la tecnica del *"Double Buffering"*, vedere [8].

[9] Vedi paragrafo 4.2.

[10] Per il codice vedere l'appendice E pag.141.

```
Imel image;
```

Passando, attraverso opportuni metodi, a questo oggetto l'indirizzo e la dimensione del buffer contenente l'immagine da analizzare , questo restituirà una lista di elementi `FIGURE`. Ogni elemento di questa lista rappresenta un figura rilevata nell'immagine passata, attraverso l'ispezione dei campi dei vari elementi della lista (ogni elemento `FIGURE` è infatti una struttura) è possibile leggerne le caratteristiche quali la posizione, area e orientamento (vedere la figura riassuntiva 4.3).

I metodi dell'oggetto *image* che permettono questo sono:

1. `int ImelInit(TIPO_PIXEL* image_ext,int parent_size_x, int parent_size_y);`

   inizializza l'oggetto di tipo *Imel* che dovrà elaborare una immagine puntata da `image_ext` di tipo `TIPO_PIXEL` (vedere il file header imel.h per la definizione di questo tipo) e di dimensioni `(parent_size_x,parent_size_y)`. Questa immagine verrà chiamata immagine *"Padre"*.

2.
```
int ImelLabeling(
    TIPO_PIXEL* image_ext,
    int size_x,     // Dimensione  x immagine child
    int size_y,     // Dimensione  y immagine child
    int ofx,        // Offset in x da cui parte l'image child
    int ofy,        // Offset in y da cui parte l'image child
    int defx,       // Definizione etichettamento in x
    int defy);      // Definizione etichettamento in y
```

   il significato dei parametri passati al metodo `ImelLabeling` è evidenziato in figura 4.2, dove `image_ext` rappresenta l'indirizzo in cui è allocata l'immagine *Padre* da analizzare. Ipotizzando che le dimensioni di questa immagine siano `(parent_size_x, parent_size_y)`, la funzione analizza solo una sotto immagine di dimensione `(size_x,size_y)` spostata di un offset `(ofx,ofy)` con un passo di campionamento `defx` in orizzontale ed `defy` in verticale. La possibilità di poter sottoporre all'analisi solo

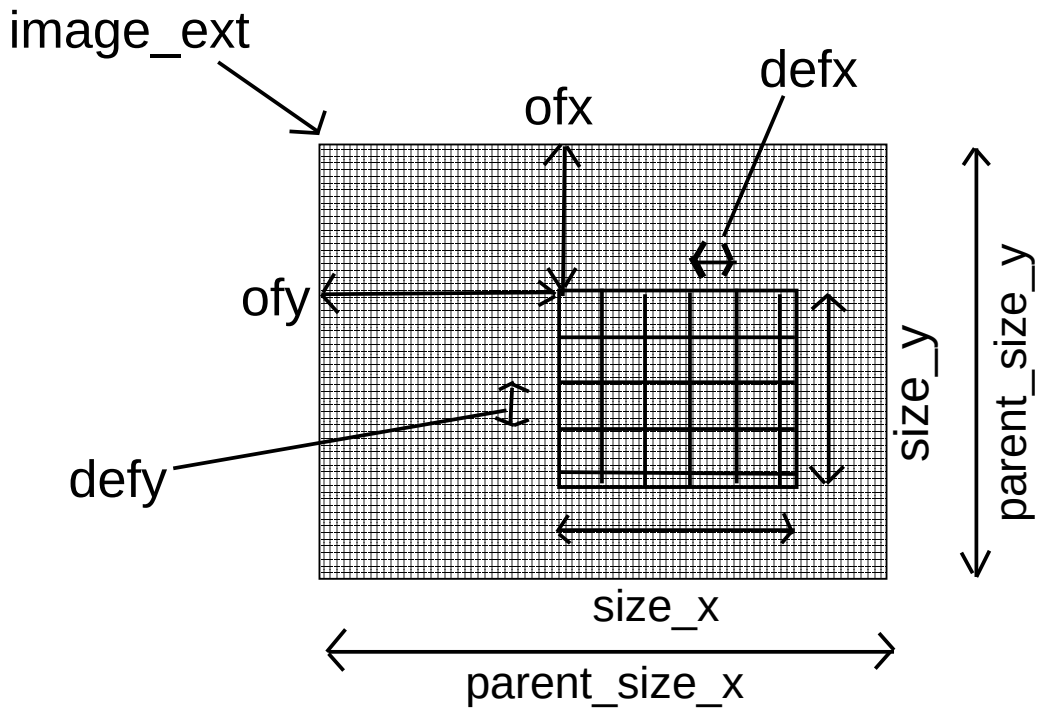

Figura 4.2: Rappresentazione dei parametri del metodo ImelLabeling.

una porzione dell'intera immagine passata permette di velocizzare notevolmente l'elaborazione se si conosce la zona in cui è posizionata la figura da analizzare[11]. Questo vantaggio è ulteriormente accentuato dalla possibilità di poter definire il passo di campionamento e quindi il numero di bit da analizzare[12]. Il metodo restituisce un intero che rappresenta il numero di figure rilevate nell'immagine analizzata. Per accedere alle strutture dati contenenti le caratteristiche degli oggetti rilevati bisogna utilizzare il metodo `ImelObjectInfo` (vedere più avanti).

[11] Tecniche largamente utilizzate nelle applicazioni di *visual servoing* quando l'elaborazione dell'immagine è molto pesante [21], infatti, all'aumentare della dimensione dell'immagine si ha un rapido decremento delle prestazioni del sistema di calcolo come indicato in [23].

[12] Notare che ponendo $ImelLabeling(image\_ext, parent\_size\_x, parent\_size\_y, 0, 0, 1, 1)$ si analizza l'intera immagine passata.

Dettagli implementativi

Di seguito viene presentato l'algoritmo di base che implementa l'etichettamento di una immagine binaria [12], procedimento già descritto in 3.5.3. Avendo una immagine *img* di dimensioni ($NUMLIN \times NUMCOL$) si può scrivere

```
/*  Inizializzazione : si associa ad ogni pixel non nullo
dell'immagine una etichetta distinta  (passo 1)*/

for ( i = 1; i < NUMLIN;i++)
    for ( j = 1; j < NUMCOL; j++)
        if ( image [ i ] [ j ] == 1 )
            label [ i ] [ j ] = newlabel ();
        else label [ i ] [ j ] = 0;

do {
    /*  Passo 2 "top-down" */
    change = FALSE;
    for ( i = 1; i < NUMLIN;i++)
        for ( j = 1; j < NUMCOL; j++)
            if ( label [ i ] [ j ] != 0 )
                {
                    m = min_topdown_neighborhood ( i , j ) ;
                    if  ( m != label [ i ] [ j ] )  change = TRUE ;
                    label [ i ] [ j ] = m;
                }
    /* Passo 3 "bottom-up" */
    for ( i = NUMLIN-1; i >= 0; i--)
        for ( j = NUMCOL-1; j >= 0; j--)
            if ( label [ i ] [ j ] != 0 )
                {
                    m = min_bottomup_neighborhood ( i , j ) ;
                    if  ( m != label [ i ] [ j ] )  change = TRUE ;
                    label [ i ] [ j ] = m;
```

```
                    }
while ( change = TRUE ) ;
```

Le due funzioni `min_topdown_neighborhood (i,j)` e `min_bottomup_neighborhood (i,j)` restituiscono la label minima dei pixel vicini di (i, j) già attraversati dal passo corrente[13], mentre `newlabel()` restituisce semplicemente un numero intero sempre diverso.

3. `FIGURE *ImelObjectInfo(int modo);`

   questo metodo ritorna un puntatore ad un struttura `FIGURE` contenente tutti i dati dell'oggetto corrispondente e facente parte di una lista creata dal metodo `ImelLabeling`. A quale oggetto si accede dipende dalle impostazioni del parametro *modo* come indicato nella tabella 4.3.

modo	**descrizione**
IMEL_MODE_NEXT	Ritorna l'oggetto corrente e passa al prossimo.
IMEL_MODE_START	Sposta l'oggetto corrente al primo elemento della lista.
IMEL_MODE_END	Sposta l'oggetto corrente all'ultimo elemento della lista
$n \geq 1$	ritorna n-esimo oggetto della lista degli oggetti trovati.

Tabella 4.3: Valori del parametro *modo*

4. `int ImelFilter(int thresold);`

   elimina dalla lista di elementi `FIGURE` tutti gli oggetti che hanno $area < thresold$. Questa funzione è molto utile per eliminare un eventuale rumore di fondo presente nell'immagine della telecamera, tale rumore infatti viene elaborato come un insieme piuttosto grande di oggetti di piccolissime dimensioni (ognuno avente un'area di pochi pixel). Infine il metodo restituisce il numero di oggetti rimasti dopo il filtraggio.

5. `int ImelOrientation(FIGURE*target);`

   calcola l'orientamento della figura puntata da `target`. È stato scelto di implementare questa propietà a parte e non di farla calcolare dal metodo `ImelLabeling` insieme alla posizione e all'area, perché, essendo una

[13] L'implementazione realizzata in questa tesi, chiamata *Imel*, pur partendo dalla traccia dell'algoritmo qui presentato, avendo però come obiettivo *una elaborazione grafica veloce*, è stata scritta in una forma più involuta ma efficiente.

caratteristica dell'immagine molto laboriosa da estrarre[14], porterebbe ad un drastico calo delle prestazioni se applicata a troppi oggetti. Dopo aver chiamato questo metodo è possibile leggere l'orientamento dell'oggetto sottoposto al calcolo semplicemente accedendo al campo `teta` (che è il nome dell'angolo di inclinazione) della struttura `target` di tipo `FIGURE`. Infine il metodo ritorna un intero indicante eventuali errori.

6. `bool ImelFree();`

   rilascia tutte le risorse allocate per l'oggetto IMEL, ritorna *true* se non vi sono stati errori nelle operazioni di deallocazione.

Tracking di un oggetto

È stato implementato nella libreria IMEL un ultimo metodo che rende possibile il riconoscimento di un target all'interno di una sequenza di immagini.

1. `int Tracking(int Kp,int Ka,int Kt);`

   Questo metodo determina all'interno della lista di oggetti estratti dalla imagine *image* un oggetto *target*. L'obiettivo è di agganciare tale oggetto e non perderne più traccia, per implementare questa operazione si è deciso di considerare come *"target da inseguire"* quell'oggetto che minimizza la seguente funzione costo:

$$C_i = K_p \cdot (P_i - P_{target})^2 + K_a \cdot (A_i - A_{target})^2 + K_t \cdot (O_i - O_{target})^2 \quad (4.1)$$

   Dove K_p, K_a, K_t sono opportuni pesi che determinano il fattore che ha più importanza (tra posizione, area ed orientamento) nel decidere quale sia l'oggetto *target*, mentre P_i, A_i, O_i sono rispettivamente posizione, area ed orientamento dell' i-esimo elemento nella lista degli N oggetti trovati nell'immagine *image*. L'elemento che ha costo minimo (Min C_i con $i = 1 \ldots N$) viene considerato come *target* e le sue proprietà vengono memorizzate in $P_{target}, A_{target}, O_{target}$ di modo che possa essere preso come punto di riferimento per determinare al prossimo fotogramma che giunge dalle telecamera il nuovo *target*.

[14] Vedere a questo proposito l'appendice A.

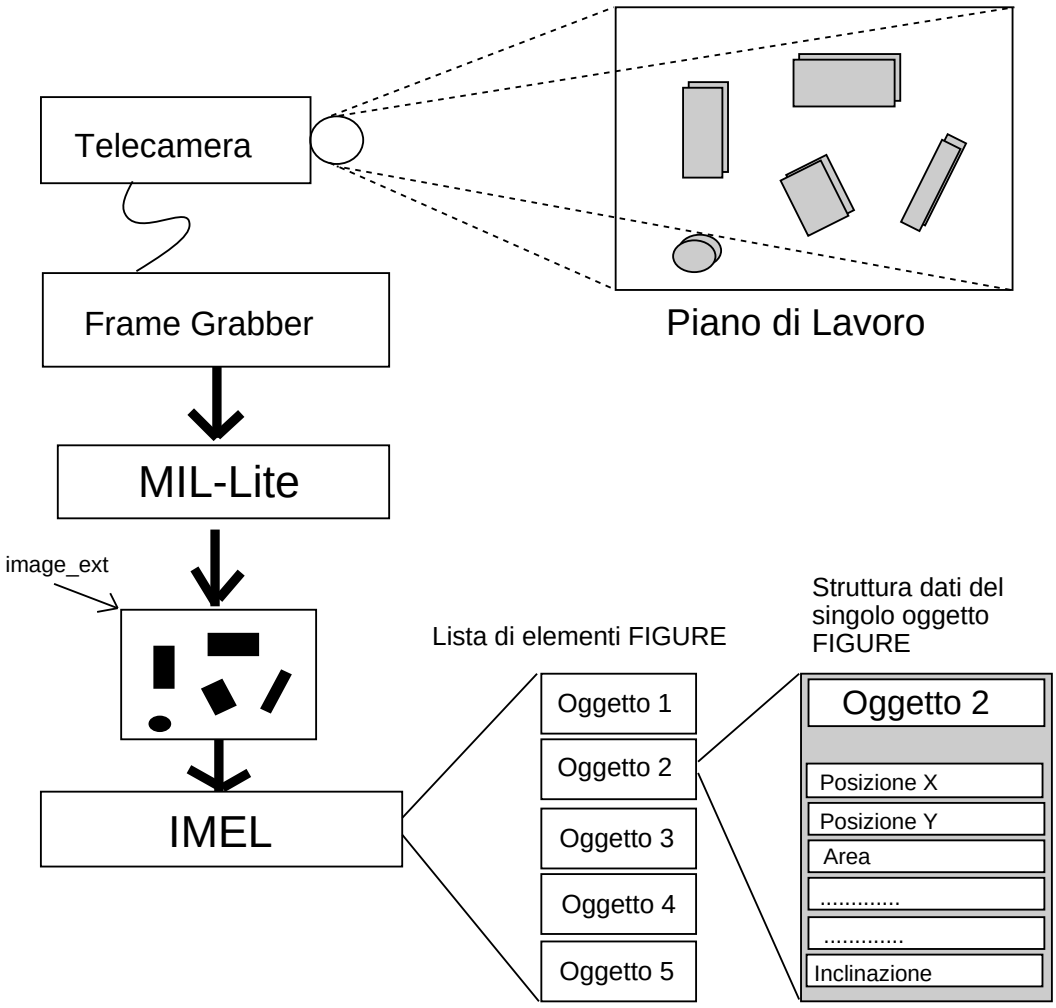

Figura 4.3: Schema riassuntivo dell'utilizzo della libreria IMEL.

In poche parole, quindi, ad un dato istante viene considerato come *target* quell'oggetto che ha *distanza* minima (Min C_i) dalla posizione che aveva il *target* all'istante precedente. In tale contesto la *distanza* è da considerarsi una misura in uno spazio a 4 dimensioni, due dimensioni per le coordinate spaziali P_i sullo schermo (x,y), una per l'area A_i ed una per l'orientamento O_i.

Il metodo infine restituisce un intero indicante la posizione (ordinale) dell'oggetto *target* nella lista degli oggetti rilevati.

4.4 L'oggetto "Matrice"

Si è deciso di realizzare un oggetto `Matrix`[15] (matrice) per aumentare l'espressività del linguaggio di programmazione. Ciò è stato possibile utilizzando al meglio gli strumenti resi disponibili dal C++ come l'*overload* degli operatori[16], in questa maniera si è potuta implementare un'algebra di base delle matrici, aumentando la compattezza delle espressioni (e quindi la leggibilità) a scapito di un leggero calo di prestazioni dovuto all'utilizzo di un tipo di dato più strutturato.

L'oggetto `Matrix` è gestibile attraverso diversi metodi ed operatori, definendo ad esempio i seguenti oggetti matrice

```
Matrix A(M,N);
Matrix B(M,N);
Matrix C(N,N);
```

ed un `float` z, si ha che:

1. A(x,y)

   Ritorna l'elemento di riga x e colonna y della matrice A, dove le $x \in [1..N]$ e $y \in [1..M]$.

2. A(x,y)=z

   Assegna alla riga x e colonna y della matrice A il valore dello scalare z.

3. A=B

   Assegna alla matrice A il valore della matrice B.

4. INV(A)

   Restituisce l'inversa di A.

5. TRASP(A)

   Restituisce la trasposta di A.

6. DET(A)

   Restituisce il determinante della matrice A.

[15] Per il codice vedere l'appendice F.4 pag.187.
[16] Vedere [3] Cap.8.

7. A+B

   Restituisce la somma delle matrice A e B.

8. A-B

   Restituisce la differenza delle matrici A e B.

9. A*C

   Restituisce il prodotto delle matrice A e C.

10. z*A

    Ritorna il prodotto della matrice A per lo scalare z.

Le precedenti espressioni sono componibili e seguono le normali regole di precedenza; gli errori devono essere gestiti dall'utente attraverso il controllo della variabile pubblica `MatrixError` dell'oggetto matrice.

4.5 L'oggetto "Robot"

Si è voluto implementare un oggetto chiamato *Robot*[17] che incapsulasse tutte le funzionalità di comunicazione con il controller UNIVAL II. Attraverso la definizione di pochi metodi è possibile quindi inviare stringhe di comando al PUMA 260 oppure aspettare il prompt o una particolare stringa di risposta dal controllore.

Definito allora un oggetto `Puma` di tipo *Robot*

```
ROBOT Puma;
```

sono stati implementati i seguenti metodi:

1. `ROBOT_RESULT Initialize();`

   inizializza tutte le strutture dati necessarie al funzionamento dell'oggetto *Robot* lanciando sia il thread di lettura che quello di scrittura, crea il buffer caratteri e la coda comandi schematizzati in figura 4.4 e configura la parta di comunicazione COM con il controller UNIVAL.

[17] Per il codice vedere l'appendice F.2 pag.170.

2. `ROBOT_RESULT SendCommand(char*cmd);`

   invia la stringa di comando puntata da `cmd` al controllore del Puma 260.

3. `ROBOT_RESULT Wait(char *cmd);`

   aspetta che il controllore invii la stringa puntata da `cmd`, se la stringa non viene ricevuta entro un certo tempo si genera un errore di attesa.

4. `ROBOT_RESULT Prompt(char *prompt);`

   aspetta che il controllore restituisca una stringa di prompt.[18].

5. `ROBOT_RESULT Sleep(char *cmd);`

   disabilita l'esecuzione dei comandi per un tempo in millisecondi pari al valore numerico della striga puntata da `cmd` (ad esempio "250" indica una attesa di 250 ms).

6. `ROBOT_RESULT Terminal();`

   genera una emulazione di un terminale con il controllore UNIVAL, così che sia possibile inviare direttamente da tastiera i comandi al robot. Per uscire dal terminale usare il metacomando *"quit"*.

7. `ROBOT_RESULT ShutDown();`

   rilascia tutte le strutture dati necessarie al corretto funzionamento dell'oggetto di tipo *Robot*. Da utilizzare quando l'oggetto *Robot* non è più da utilizzarsi.

8. `ROBOT_RESULT Asincronous();`

   abilita la modalità asincrona di esecuzione comandi (modalità di default) in cui tutti i comandi inviati vengono memorizzati in una coda ed eseguiti uno dopo l'altro, per cui è possibile lanciare una serie di comandi e portare avanti altri compiti senza preoccuparsi dell'esecuzione di ogni singolo comando. Tutto questo è fattibile tramite l'utilizzo di 2 Thread, il primo dedicato alla ricezione di stringhe dalla seriale, il secondo dedito alla effettiva esecuzione dei vari comandi associati ai metodi dell'oggetto.

[18] Vedere file Robot.h per le definizioni delle stringhe di *prompt*.

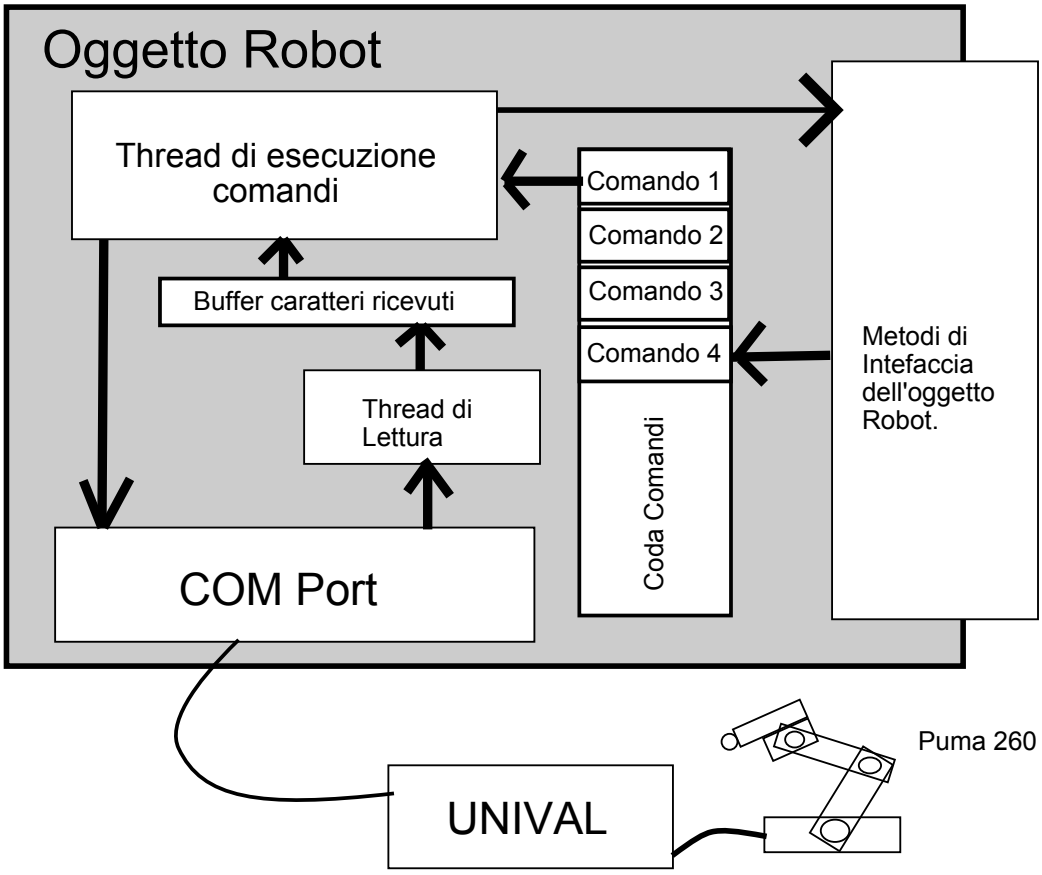

Figura 4.4: Scomposizione funzionale dell'oggetto *Robot*.

In figura 4.4 si vede un esempio di quattro comandi lanciati dal programma principale attraverso i metodi dell'oggetto e solo uno (comando 1) sta effettivamente per essere eseguito.

9. `ROBOT_RESULT Sincronous();`

   abilita la modalità sincrona di esecuzione dei comandi. In tale modalità il metodo richiamato restituisce un errore (ROMBOT_CMD_BLOCK) se nel frattempo è in corso di esecuzione un altro comando[19], se invece la coda comandi è vuota il comando verrà eseguito ed il metodo ritornerà il codice ROBOT_OK. Notare che in questa modalità non si può lanciare più di un comando alla volta, questo però permette di controllare esattamente quale operazione (comando) si stà effettivamente eseguendo.

[19] Un comando si intende in "esecuzione" quando il *thread di esecuzione comandi* lo sta eseguendo, vedere figure 4.4.

4.6 La libreria "Control"

Questa libreria[20] riunisce l'insieme di tutte quelle funzioni strettamente legate al controllo della dinamica del robot nell'applicazione *"eye in hand"*.

In riferimento ai capitoli 6 e 7 sono stati quindi implementati i regolatori PD e PID, un modello dinamico del sistema reale, il *predittore di Smith* ed il regolatore non lineare denominato in questa tesi come PIDNL[21] oltre ad una serie di filtri come quello di Butterworth ed filtro a variabili di stato.

A titolo di esempio si riportano di seguito alcune procedure:

1. `void PID_TIME(double time,Matrix* B,TIPO K,TIPO Ti,TIPO Td, TIPO N,Matrix* E,Matrix* Exit);`

   procedura che implementa in forma digitale il regolatore PID, l'ingresso è rappresentato dalla matrice `Matrix` *E che esprime il segnale errore, l'uscita è data dalla matrice *Exit. Entrambe queste matrici hanno una dimensione 1×2 (sono quindi dei vettori) perché operano sul piano visivo e quindi vengono espresse nelle coordinate x,y di tale piano.

   I parametri (K,Ti,Td,N) caratterizzano le proprietà dinamiche del regolatore PID. La matrice `*B` (2×2) rappresenta il cambiamento di base introdotto dalla telecamera come spiegato nel paragrafo 3.4.2, in fine `time` indica il tempo di chiamata della procedura.

2. `void PREDICTOR_PUMA(double time,Matrix* E,Matrix* Exit,int mode);`

   procedura che implementa al suo interno in forma digitale un modello (a stati) del sistema costituito dal robot Puma 260 e dalla seriale. Tale procedura è in grado di fungere sia da stimatore della posizione attuale del robot PUMA 260 sia da *predittore* della posizione futura (vedere il predittore di Smith paragrafo 6.5).

   L'ingresso è rappresentato dalla matrice `*E` (1×2) l'uscita invece dalla matrice `*Exit`. Se `mode=1` si aggiorna l'ingresso del modello e `*Exit` coincide con l'uscita stimata del sistema, se `mode=0` viene semplicemente restituita l'uscita stimata del sistema in `*Exit`, mentre se `mode=2` la matrice `*Exit` contiene la predizione del valore futuro del sistema.

[20] Per i codici vedere l'appendice F.3 pag. 174.

[21] Per una migliore comprensione dei prototipi delle procedure di questa libreria si consiglia prima la lettura dei capitoli sopraelencati.

4.7 Linguaggio VAL II del Robot Puma 260

I sistemi di programmazione dei robot si possono catalogare fondamentalmente entro tre ampie categorie: "sistemi di *guida* (guiding system) nei quali l'utilizzatore guida il robot nei suoi movimenti, sistemi di programmazione a *livello del robot* (robot-level-system) nei quali l'utilizzatore scrive dei programmi che specificano le operazioni di movimento e di rilevazione sensoriale che è necessario eseguire, e sistemi di programmazione al *livello dei compiti* (task-level-system) nei quali l'utilizzatore specifica le operazioni per mezzo degli effetti che si dedidera producano sugli oggetti"[22].

Il VAL II[23] è considerato a *livello di robot* ed è un vero e proprio linguaggio di programmazione che gestisce diversi tipi di dato che vanno dai numeri reali alle stringhe, ha diverse istruzioni di condizionamento di flusso come il *goto* e l'*if-then* oltre che un completo set di istruzioni dedicate alla movimentazione e al controllo della dinamica del braccio robotico. È possibile quindi scrivere programmi che vengono interpretati ed eseguiti dal controllore Unival. Le caratteristiche principali del linguaggio Val II possono essere cosí classificate:

1. impostazione dinamica dei parametri di accelerazione, decelerazione e velocità.

2. controllo del moto del robot secondo più sistemi di riferimento (Word e Tool).

3. generazione di moti circolari e curvi.

4. passibilià di definire subroutine e di gestirne il passaggio parametri.

5. possibilità di eseguire più programmi contemporaneamente in maniera concorrente (*robot control programs e process control programs*).

6. capacità di modificazione in tempo reale della traiettoria del robot.

L'ultima proprietà è indispensabile per la realizzazione del *controllo in retroazione visiva*, infatti, utilizzando un controllo in tempo reale della traiettoria (detto *Alter mode*), non è necessario interrompere il movimento del robot se questo si

[22] Vedere [17] pag.149.
[23] Vedere [4, 15, 16].

sta allontanando dall'obiettivo, basta correggerne la direzione con una opportuna istruzione. Se non si sta utilizzando un controllo in tempo reale, una volta impartito un movimento al braccio robotico, ogni altro tentativo di movimento viene sospeso fin tanto che il primo non è stato compiuto.

4.7.1 Programmi di controllo del robot(Robot Control Programs)

Un *programma di controllo del robot* è un programma VAL II che ha la possibilità di controllare direttamente il robot attraverso un insieme di istruzioni tra cui anche quelle per la movimentazione del braccio meccanico. Il numero di programmi che può essere memorizzato dipende solo dalla memoria disponibile nel controllore Unival ma può essere mandato in esecuzione (istruzione `execute`) uno ed un solo *programma di controllo del robot* alla volta.

4.7.2 Programmi di controllo dei processi(Process Control Programs)

I *programmi di controllo dei processi* sono particolari processi che possono essere lanciati dal *programma di controllo del robot* in maniera concorrente, possono dialogare con quest'ultimo[24] ed effettuare ogni tipo di operazione tranne che eseguire istruzioni di movimentazione del braccio (prerogativa questa del solo *programma di controllo del robot*). È possibile lanciare (istruzione `pcexecute`)fino ad un massimo di 4 processi di questo tipo ad ognuno dei quali è associata una *time slice* cioè un certo quantitativo del tempo di elaborazione dell'Unival.

4.7.3 Controllo in tempo reale (Alter Mode)

Come anticipato il controllo di traiettoria in real–time è una delle caratteristiche piú evolute del controllore Unival. Questa funzionalitá consente, anche se limitatamente a spostamenti di tipo cartesiano, di correggere una traiettoria di movimento.

Si accede a questa modalitá di funzionamento tramite l'istruzione '*ALTER*' che può essere lanciata solo da un *programma di controllo del robot*. Dal momen-

[24] Attenzione Val II esegue in un ambiente globale, per cui ogni variabile, se non dichiarato esplicitamente il contrario, è visibile da ogni processo che esegue.

to in cui viene impartita, la traiettoria corrente puó essere alterata in ciascuno dei suoi sei gradi di libertá tramite l'istruzione '*ALTOUT*'

Sintassi

```
ALTER (-1,<Mode>)
```

- '*Mode*' consente di specificare le caratteristiche assumibili dalla modalitá di modifica real–time delle traiettorie. Puó essere una variabile, un numero o un'espressione matematica ed ad ogni bit corrisponde una delle

Bit N^o	Significato
0	**OFF**: Cambiamenti non cumulativi **ON**: Cambiamenti cumulativi
1	**OFF**: Seleziona le coordinate TOOL **ON**: Seleziona le coordinate WORLD
2	non usato
3	non usato
4	deve essere necessariamente **ON**
5	non usato
6	non usato
7	non usato

Tabella 4.4: Bit mask del parametro *Mode*

possibili caratteristiche della modalitá. Si veda la tabella 4.4 per i dettagli sul significato dei singoli bit. Nell'applicazione ad anello chiuso sviluppata in questa tesi, si è scelta la modalità *cumulativa* in coordinate *Tool*. La modalità *cumulativa*, permette di inviare i soli scostamenti relativi rappresentati dall'errore ottenuto dalla differenza tra la posizione corrente e la posizione voluta. Mentre lavorare in coordinate *Tool* permette di rendere indipendente il sistema dalla posizione-inclinazione del piano di lavoro.

Sintassi

```
ALTOUT 0,{<x>},{<y>},{<z>},{<rx>},{<ry>},{<rz>}
```

- '*x*', '*y*', '*z*' rappresenta la quantitá, in millimetri, di cui deve essere variata la traiettoria nelle direzioni x, y e z rispettivamente[25].

- '*rx*', '*ry*', '*rz*' rappresenta la rotazione, in gradi, che deve essere impressa alla traiettoria lungo gli assi rx, ry, rz rispettivamente.

Questa istruzione deve essere inviata al controllore almeno ogni 32ms affinché la modalitá *ALTER* permanga, se questa condizione non viene rispettata, l'esecuzione del programma si arresta generando un errore.

In questa tesi è stato scritto un *programma di controllo dei processi* con l'unico scopo di inviare il comando `altout` e permettere quindi al programma principale di disinteressarsi del vincolo appena citato rendendo così più flessibile l'intera applicazione.

Per uscire dall'*Alter mode* si utilizza l'istruzione `noalter`.

4.7.4 Programmi *Pick*,*Goa* e *Goa1*

Senza scendere nei particolari tutti i programmi VAL II scritti in questa tesi[26] eseguono ciclicamente 3 passi fondamentali

1. Invio Prompt tramite seriale al PC.

2. Lettura dalla seriale della posizione (locazione) da far assumere al braccio robotico.

3. Movimentazione braccio.

4.7.5 Istruzioni utilizzate

Di seguito verranno presentate una serie di istruzioni Val II utilizzate in questo lavoro.

1. `PROMPT{<messaggio>},{<var>}`

   Unival invia alla seriale la stringa `<messaggio>` e aspetta in ritorno l'arrivo del valore da associare alla varibile `<var>`.

[25] Notare che aggiornando la posizione ogni 32ms, questo equivale ad impostare una velocità.
[26] Per i codici vedere l'appendice D pag.135.

2. `MOVE{<location>}`

   Muove l'organo terminale del robot dalla posizione corrente alla posizione indicata dalla locazione `<location>`. Il movimento avviene attraverso una curva interpolata, se invece si vuole percorrere una traiettoria rettilinea bisogna utilizzare l'istruzione `MOVES`.

3. `BREAK`

   Forza l'attesa della fine di un movimento (conclusione istruzione MOVE) prima di eseguire qualsiasi altra operazione.

4. `APPROS<location>,<distance>`

   Compie un movimento verso la locazione `<location>` fermandosi però ad una distanza `<distance>` (lungo l'asse z in coordinate *Tool*) da `<location>`.

5. `DEPARTS<distance>`

   Genera uno spostamento di ampiezza `<distance>` lungo l'asse z nel sistema di riferimento *Tool*.

4.8 Locazioni e Trasformazioni

Facendo riferimento al capitolo 3 si esporrà come il controllore Unival definisce la posizione dell'organo terminale del braccio e di come sia possibile definire operatori di rototraslazione applicabili a quest'ultimo[27].

4.8.1 Locazioni (Location)

Nel linguaggio Val II una *locazione* indica la posizione e l'orientamento dell'organo terminale del braccio rispetto al sistema di riferimento *"Word"*. Per definire posizione ed orientamento di un vettore nello spazio bastano 6 elementi, 3 per le coordinate cartesiane e 3 per le rotazioni intorno agli assi del sistema di riferimento.

[27] Questo tipo di operazione è l'equivalente della matrice T_{tot} utilizzata nella 3.20.

4.8.2 Trasformazioni (Transformation)

Nel paragrafo 3.1.4 è stato descritto come rappresentare attraverso le trasformazioni omogenee una rototraslazione di un vettore (formula 3.15 di seguito ripresentata)

$$T = \left(\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & O_x \\ r_{21} & r_{22} & r_{23} & O_y \\ r_{31} & r_{32} & r_{33} & O_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right) = \left(\begin{array}{c|c} R_{3\times3} & O_{3\times1} \\ \hline 0_{1\times3} & 1_{1\times1} \end{array}\right) \tag{4.2}$$

Rileggendo il paragrafo sopra citato, si può notare come gli elementi della matrice di rotazione $R_{3\times3}$ pur essendo 9 in realtà derivino da sole 3 variabili che rappresentano appunto le rotazioni intorno a ciascun asse del sistema di riferimento (nell'esempio di 3.1.4 chiamate rispettivamente R_x, R_y, R_z). Questo evidenzia come per definire in maniera univoca una rototraslazione di un vettore sia ridondante utilizzare una matrice omogenea formata da 16 elementi quando basterebbero solo 6 elementi, 3 per la posizione (elementi di $O_{3\times1}$) e 3 per le rotazioni (cioè gli angoli di rotazione).

Convenzioni sugli angoli di rotazione

Notare che per la proprietà 6 del paragrafo 3.1.2 si deduce che per definire in maniera univoca sia una *locazione* che una *trasformazione* bisogna definire la sequenza di rotazioni intorno agli assi del sistema di riferimento. A questo proposito nella letteratura robotica sono state definite diverse convenzioni per definire una rotazione intorno ad un asse generico[28]:

1. Angoli di Eulero
2. Angoli di ROLL,PITCH,YAW
3. Angoli O,A,T

Val II utilizza la 3 per definire una *trasformazione*, eseguendo in sequenza le rotazioni O, A e T[29] attraverso l'istruzione

```
TRANS({<X>},{<Y>},{<Z>},{<O>},{<A>},{<T>})
```

[28] Per una definizione esauriente delle convenzioni 1 e 2 vedere [20].

[29] Per la definizione dei tre angoli si rimanda al manuale di riferimento [4].

Per ultimo, una proprietà molto importante è la possibilità di comporre diverse *trasformazioni* in maniera del tutto analoga a quanto visto nel paragrafo 3.1.5.