ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE - DIN

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA MECCANICA (5724)

TESI DI LAUREA

in

Industrial robotics

SVILUPPO E VALIDAZIONE DI UN'ARCHITETTURA DI CONTROLLO DISTRIBUITA PER PROCESSI WAAM ROBOTIZZATI: DAL FILE-BASED ALLO STREAMING REAL-TIME

CANDIDATO:
Jacopo Zepponi

RELATORE:
Prof. Ing. Claudio Melchiorri

CORRELATORE:
Luca Mazzuferi

Anno accademico 2025/2026

Sessione III

ai miei nonni, genitori, Christian e tutti i miei amici

This work is licensed under the Creative Common
Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).
To view a copy of this license, visit
`https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode`

Abstract

La Wire Arc Additive Manufacturing (WAAM) rappresenta una tecnologia promettente per la produzione additiva metallica, ma la sua industrializzazione rimane ostacolata dalla mancanza di ecosistemi software integrati. I sistemi WAAM robotizzati attuali si basano su un approccio "file-based" in cui migliaia di righe di codice KRL vengono generate offline e caricate staticamente sul controllore, limitando flessibilità, tracciabilità e possibilità di adattamento in tempo reale.

Questa tesi presenta lo sviluppo e la validazione di una nuova architettura di controllo software per processi WAAM robotizzati, basata su tre pilastri: (1) riorganizzazione del codice robot in logica indicizzata con riduzione del 78% della dimensione dei file, (2) modello dati unificato in formato JSON per layer planari e sezioni a spirale, (3) architettura distribuita di streaming PC–PLC–KUKA con triplo buffer per l'alimentazione in tempo reale del processo.

L'architettura proposta abilita procedure di recovery automatico, elimina la necessità di rigenerazione dei file, migliora tracciabilità e scalabilità, e pone le basi per controllo adattivo futuro. La validazione è stata condotta attraverso simulatori software dedicati che replicano l'intera catena di comunicazione, dimostrando correttezza e robustezza su dati reali del manufatto.

Indice

Abstract **5**

Introduzione **11**

1 Analisi della macchina presente in azienda **17**
1.1 Architettura generale del sistema . . . 17
1.2 Funzionamento e logica di interazione . . . 19
1.3 Ruolo della macchina nel contesto del lavoro di tesi . . . 19

2 Introduzione alla tecnologia WAAM **21**
2.1 Origine e definizione del WAAM . . . 21
2.2 Principio di funzionamento . . . 23
2.2.1 Pianificazione del processo . . . 23
2.2.2 Deposizione del materiale . . . 24
2.2.3 Controllo delle proprietà e post-processing . . . 25
2.3 Tecnologie di arco utilizzabili nel processo WAAM . . . 25
2.3.1 Gas Tungsten Arc Welding (GTAW o TIG) . . . 25
2.3.2 Gas Metal Arc Welding (GMAW o MIG) . . . 25
2.3.3 Cold Metal Transfer (CMT) . . . 26
2.3.4 Plasma Arc Welding (PAW) . . . 27
2.3.5 Sintesi . . . 27
2.4 Vantaggi, limiti e applicazioni della tecnologia WAAM . . . 27
2.4.1 Vantaggi del WAAM . . . 27
2.4.2 Limiti e criticità . . . 28
2.4.3 Applicazioni industriali del WAAM . . . 29

3 Stato dell'arte: slicing e controllo robotico nel WAAM **31**
3.1 Slicing e pianificazione della traiettoria . . . 31
3.2 Monitoraggio e controllo robotico nel WAAM . . . 33

4 Analisi del processo esistente e sviluppo della nuova logica KRL **35**
4.1 Descrizione del workflow esistente: slicing e generazione del codice KRL . 37
4.2 Analisi delle criticità del codice KRL originale . . . 39
4.3 Sviluppo di script Python per la semplificazione e riorganizzazione del codice KRL . . . 41
4.3.1 Script per la riorganizzazione del file .dat (kuka_dat_to_array_v_-updated.py) . . . 41
4.3.2 Script per la semplificazione del file .src (create_simplified_src_-v3.py) . . . 42

4.4 Vantaggi della nuova logica KRL 44
4.5 Verifica di equivalenza geometrica dei waypoint 45
4.5.1 Metodologia di verifica 45
4.5.2 Risultato 46
4.5.3 Implicazioni 46
4.6 Progettazione delle procedure di recovery 48
4.6.1 Motivazione e requisiti funzionali 48
4.6.2 Architettura generale del sistema di recovery 49
4.6.3 Procedura per calcolo di un punto intermedio 51
4.6.4 Sintesi del flusso di esecuzione e della logica di recovery 52

5 Gestione del processo di stampa a spirale e parsing complessivo 57
5.1 Ruolo delle sezioni a spirale e limiti del pattern per layer planari 57
5.2 Analisi del codice KRL della sezione 4 e definizione del nuovo pattern . . . 60
5.3 Parsing complessivo del progetto e integrazione dei diversi tipi di layer . . 62

6 Progettazione dell'architettura di streaming PC-PLC-KUKA: dal concetto alla simulazione 67
6.1 Dall'array KRL al file JSON: i vantaggi di una struttura dati gerarchica . . . 67
6.1.1 Organizzazione strutturale e leggibilità 68
6.1.2 Interoperabilità e flessibilità 68
6.2 La nuova architettura di streaming (PC-PLC-KUKA) 69
6.2.1 Task 0: Il PC (Sender) 69
6.2.2 Task 1: Il PLC (Streamer) 69
6.2.3 Task 2 & 3: KUKA (SPS.SUB - Assembler & Buffer Logic) 70
6.2.4 Task 4: KUKA (MAIN.SRC - Robot Executor) 70
6.3 Vantaggi della nuova architettura 70
6.4 Progettazione della transizione verso l'implementazione hardware 71
6.4.1 Confronto tra attuale e nuova architettura 71

7 Simulazione software della comunicazione PC-PLC 75
7.1 Obiettivi e ruolo della simulazione PcPlcSimulation 75
7.2 Struttura logica del progetto 77
7.2.1 Modello di dominio dei pacchetti di processo 77
7.2.2 Dal JSON WAAM alla timeline di ProcessPacket 77
7.2.3 Rispetto dei vincoli ADS: PacketChunker 78
7.2.4 Report di verifica dei pacchetti 78
7.2.5 Serializzazione PLC-friendly: StProcessPacket e PacketSerializer . . 79
7.3 Simulazione PC–PLC e comportamento temporale 80
7.3.1 Memoria PLC simulata e interfaccia IPlcConnection 80
7.3.2 PacketSendService (PC Sender) 80
7.3.3 VirtualPlcRuntime (PLC virtuale) 81
7.4 Interfaccia grafica WPF per il monitoraggio del processo 81
7.4.1 Layout della finestra 81
7.5 Sintesi e considerazioni sulla simulazione 83

8 Simulazione software della catena PLC–KUKA su EtherCAT **85**
8.1 Obiettivi simulazione PC-PLC-KUKA 85
8.1.1 Motivazioni e contesto applicativo 86
8.1.2 Domande di ricerca e obiettivi tecnici 87
8.1.3 Criteri di validazione e ruolo della GUI 88
8.2 Richiamo dell'architettura da simulare 88
8.2.1 Ruolo della simulazione rispetto all'architettura reale 90
8.3 Modellazione comunicazione PLC–KUKA su EtherCAT 91
8.3.1 Struttura dell'immagine di processo EtherCAT 91
8.3.2 Flusso logico dei dati e concetto di StreamItem 92
8.3.3 Modello del task PLC EtherCAT: Task1PlcEtherCatStreamer 93
8.3.4 Modello del task KUKA SPS: Task2KukaAssembler e BUFF_1 94
8.3.5 Integrazione nel simulatore PLC–KUKA 95
8.4 Presentazione dei risultati tramite log e interfaccia grafica 97
8.4.1 Caso di test 1 – Bead lineare standard (layer 1, bead 6) 98
8.4.2 Caso di test 2 – Sequenza a spirale (layer 15, sequence 1) 100
8.4.3 Discussione critica dei risultati 103
8.5 Valutazione del ruolo del PLC nella catena PC–PLC–KUKA 104

Conclusioni **107**

Appendici **109**

A Immagini **111**

B Codice **123**

Introduzione

Contesto e motivazioni

Il Wire Arc Additive Manufacturing (WAAM) rappresenta una delle tecnologie più promettenti nell'ambito della produzione additiva metallica, combinando elevati tassi di deposizione, flessibilità geometrica e costi contenuti rispetto alle tecnologie a letto di polvere. Tuttavia, la sua industrializzazione rimane ostacolata da una serie di criticità legate alla complessità del processo: la gestione termica, la qualità superficiale, la sincronizzazione tra movimenti robotici e stati di processo, e soprattutto l'assenza di un ecosistema software standardizzato e integrato per la pianificazione, l'esecuzione e il controllo del processo.

L'impresa Loccioni, presso cui è stato condotto il presente lavoro di ricerca, dispone di una cella WAAM robotizzata basata su robot KUKA KR 50, tavola rotante come asse esterno, torcia PAW (Plasma Arc Welding) e sistema di rolling per la compattazione del materiale depositato. Questa configurazione, sebbene tecnologicamente avanzata, presenta una gestione del processo ancora fortemente vincolata a un approccio "file-based" tradizionale: i programmi di deposizione sono generati offline da strumenti CAD/CAM (Grasshopper/Rhinoceros), convertiti in migliaia di righe di codice KRL (KUKA Robot Language) e caricati staticamente nel controllore del robot prima dell'esecuzione.

Questa modalità operativa, pur funzionale per produzioni semplici e ripetitive, evidenzia limiti significativi quando applicata alla realtà di un processo WAAM industriale caratterizzato da:

- **Componenti geometricamente complessi:** il manufatto reale presenta sia sezioni planari con deposizione a cordoni paralleli, sia porzioni a spirale, difficili da gestire con una logica di programmazione uniforme tra i due tipi;
- **Necessità di interruzioni e recovery:** le operazioni di manutenzione della torcia, sostituzione del filo, ispezioni intermedie e gestione di anomalie di processo richiedono procedure di interruzione e ripresa automatizzate, difficilmente realizzabili con codice monolitico;
- **Rigidità del flusso di lavoro:** qualsiasi modifica al percorso di deposizione, anche minima, richiede la rigenerazione completa dei file KRL e il ricaricamento sul controllore, con conseguente allungamento dei tempi di setup e aumento del rischio di errore umano;
- **Scarsa tracciabilità:** la correlazione tra il modello CAD originale, i parametri di slicing, il codice robot generato e il manufatto fisico finale è frammentata su

più file e strumenti software, rendendo complessa la diagnosi post-processo e l'ottimizzazione iterativa dei parametri;

- **Impossibilità di adattamento in tempo reale:** l'architettura file-based esclude per principio la possibilità di integrare feedback da sensori di processo (ad esempio, profilometri laser per la misurazione del cordone depositato) e di correggere dinamicamente la traiettoria o i parametri di saldatura durante l'esecuzione.

Questi limiti non sono peculiari dell'impianto Loccioni, ma rappresentano criticità ricorrenti in molti sistemi WAAM robotizzati descritti in letteratura, dove la separazione tra pianificazione offline e esecuzione online, unita alla mancanza di formati dati standardizzati, costituisce un ostacolo alla scalabilità e alla diffusione industriale della tecnologia.

Obiettivi della tesi

Il presente lavoro si propone di affrontare queste criticità attraverso la progettazione, lo sviluppo e la validazione di una nuova architettura di controllo software per processi WAAM robotizzati, basata su tre pilastri fondamentali:

- **Riorganizzazione del codice robot:** trasformazione della rappresentazione lineare e monolitica dei programmi KRL in una logica strutturata basata su array, cicli e indici logici (bead, segmento), che riduca drasticamente la dimensione del codice, ne migliori la leggibilità e abiliti procedure di recovery automatico;

- **Modello dati unificato in formato JSON:** definizione di una rappresentazione gerarchica e auto-descrittiva dell'intero processo di deposizione, capace di integrare in un unico file sia layer planari sia sezioni a spirale, con metadati completi di contesto (layer, bead/sequence, tipo di pacchetto, punti, variabili di processo);

- **Architettura distribuita di streaming PC–PLC–KUKA:** superamento del paradigma file-based attraverso un sistema in cui il PC funge da sorgente centralizzata dei dati, il PLC Beckhoff gestisce il buffering e lo streaming su bus EtherCAT, e il controllore KUKA riceve ed esegue i pacchetti di processo in tempo reale, con logica di triplo buffer per prevenire interruzioni del flusso di deposizione.

L'architettura proposta è stata interamente sviluppata e validata in ambiente software, attraverso la costruzione di simulatori dedicati per le tratte PC→PLC e PLC→KUKA, che hanno permesso di verificare la correttezza della logica di comunicazione, la coerenza dei formati dati e il comportamento temporale del sistema su dati reali del manufatto, senza necessità di accesso continuativo all'impianto fisico.

Vantaggi dell'architettura proposta

L'adozione dell'architettura di streaming PC–PLC–KUKA comporta una serie di vantaggi significativi rispetto all'approccio tradizionale, articolabili su più livelli:

Vantaggi operativi immediati

- **Riduzione del codice KRL:** la trasformazione da rappresentazione lineare a logica indicizzata riduce la dimensione complessiva dei file .src e .dat di circa il 78%, semplificando debug, manutenzione e comprensione del programma;

- **Procedure di recovery automatizzate:** l'accesso ai waypoint tramite indici logici (bead, segmento) permette di implementare rientri deterministici dopo interruzioni, senza necessità di modifica manuale del codice. Il sistema è in grado di calcolare automaticamente traiettorie di avvicinamento sicure e di riprendere la deposizione dal punto esatto di interruzione;

- **Eliminazione della rigenerazione dei file:** modifiche ai parametri di processo, correzioni di traiettoria o aggiornamenti della strategia di deposizione richiedono solo l'aggiornamento di parte del file JSON sul PC, senza necessità di rigenerare, ricompilare e ricaricare programmi KRL sul robot.

Vantaggi di flessibilità e scalabilità

- **Gestione unificata di geometrie eterogenee:** il modello dati JSON e la logica di parsing sviluppata permettono di trattare in modo omogeneo layer planari, sezioni a spirale e potenzialmente altre strategie di deposizione future, all'interno della stessa pipeline di esecuzione;

- **Modularità del sistema:** la separazione netta tra pianificazione (PC), buffering/streaming (PLC) e esecuzione (KUKA) permette di evolvere ciascun modulo in modo indipendente. Ad esempio, è possibile sostituire l'algoritmo di slicing, sostituire il robot con un altra tipologia o integrare nuovi sensori agendo solo sul livello PC, senza toccare la logica di streaming PC-PLC–ROBOT;

- **Riusabilità del codice:** la stessa infrastruttura di comunicazione PC–PLC–KUKA può essere riutilizzata per processi WAAM diversi (materiali differenti, strategie di deposizione alternative) semplicemente modificando il contenuto del file JSON di progetto. Inoltre, cambiando il tool presente sul robot, si potrebbe andare a realizzare lavorazioni completamente differenti dal processo WAAM.

Vantaggi di tracciabilità e qualità

- **Correlazione diretta codice-manufatto:** ogni pacchetto di processo è arricchito con metadati completi (layer, bead/sequence, tipo, indici di chunk) che permettono di correlare univocamente ogni tratto di materiale depositato con i parametri di processo utilizzati e con la posizione nel modello CAD originale;

- **Logging strutturato:** l'architettura distribuita facilita la registrazione dettagliata di tutti gli eventi di processo (invio pacchetti, riempimento buffer, esecuzione movimenti, segnali di errore), generando dati utilizzabili per analisi post-processo, ottimizzazione dei parametri e validazione della qualità;

- **Simulazione end-to-end:** la possibilità di simulare l'intera catena PC–PLC–KUKA in ambiente software, con gli stessi formati dati e la stessa logica del sistema rea-

le, costituisce uno strumento potente per validazione, formazione degli operatori e test di nuove funzionalità senza rischi per l'impianto fisico.

Vantaggi strategici e di integrazione

- **Abilitazione di controllo adattivo:** sebbene non implementato nella presente tesi, l'architettura pone le basi per integrare feedback da sensori di processo (ad esempio, profilometri laser, termocamere, sensori di corrente) e modificare dinamicamente i pacchetti successivi in funzione delle misure effettuate, realizzando un vero controllo a ciclo chiuso del processo WAAM;

- **Interoperabilità multi-vendor:** l'uso di standard aperti (JSON per i dati, ADS/EtherCAT per la comunicazione) riduce il lock-in tecnologico verso specifici fornitori di software di slicing o di controllori robotici, facilitando future evoluzioni dell'impianto;

- **Formazione e manutenzione:** la maggiore chiarezza del flusso dati e la modularità del sistema riducono la curva di apprendimento per nuovi tecnici e semplificano le attività di manutenzione ordinaria e straordinaria;

- **Ricerca e sviluppo:** l'architettura proposta, validata tramite simulazione software, costituisce una piattaforma abilitante per la ricerca futura su algoritmi di ottimizzazione del percorso, strategie di deposizione multi-materiale, integrazione di tecniche di machine learning per la previsione di difetti, e altre tematiche di frontiera nel campo del WAAM.

Struttura della tesi

Il lavoro è organizzato in otto capitoli che seguono un percorso logico dal sistema esistente alla nuova architettura proposta, dalla modellazione concettuale alla validazione tramite simulazione software:

- **Capitolo 1** descrive l'architettura meccatronica della cella WAAM presente in azienda, evidenziando i vincoli hardware e le criticità del processo che guidano le scelte progettuali successive.

- **Capitolo 2** introduce il quadro tecnologico del WAAM, dalla classificazione nell'Additive Manufacturing alle tecnologie di arco, fornendo le basi concettuali per comprendere i requisiti del sistema di controllo.

- **Capitolo 3** analizza lo stato dell'arte su slicing, toolpath planning e controllo robotico nel WAAM, identificando il gap nella ricerca relativo all'assenza di ecosistemi software integrati.

- **Capitolo 4** presenta il primo intervento sul sistema: l'analisi del workflow esistente basato su file KRL, lo sviluppo di script Python per la riorganizzazione del codice in logica indicizzata, e la progettazione delle procedure di recovery automatico.

- **Capitolo 5** affronta la gestione delle sezioni a spirale, definendo un pattern di processo generalizzabile e sviluppando un parser dedicato che, integrato con quello dei layer planari, produce un modello dati unificato in formato JSON.
- **Capitolo 6** introduce l'architettura distribuita di streaming PC–PLC–KUKA, descrivendo il ruolo dei tre attori, la logica di triplo buffer e i vantaggi rispetto all'approccio file-based.
- **Capitolo 7** presenta la simulazione software della comunicazione PC→PLC via ADS, validando la generazione dei pacchetti, la serializzazione in formato PLC-friendly e l'handshake tra i due sistemi.
- **Capitolo 8** completa la validazione con la simulazione end-to-end della catena PLC→KUKA su EtherCAT, dimostrando la correttezza del protocollo di streaming, dell'assemblaggio dei pacchetti e del riempimento dei buffer su dati reali del manufatto.

Capitolo 1

Analisi della macchina presente in azienda

Il lavoro di ricerca è stato condotto su una cella WAAM robotizzata installata presso l'impresa Loccioni, assunta come caso studio sperimentale per lo sviluppo e la validazione della logica di controllo software descritta nella tesi. Questo capitolo inquadra l'architettura meccatronica dell'impianto e ne esplicita le implicazioni sul processo, evidenziando perché nel WAAM robotizzato la qualità dell'esecuzione dipende in modo critico non solo dalla traiettoria geometrica, ma dalla coerenza temporale tra movimenti e stati di processo (rotazione, arco, rolling e pause). Il capitolo è articolato in tre parti: la Sezione 1.1 descrive i principali elementi della cella (robot, asse esterno E1, torcia PAW e sistema di rolling); la Sezione 1.2 chiarisce la logica di interazione tra tali sottosistemi e la sincronizzazione richiesta durante la deposizione; infine, la Sezione 1.3 collega questi vincoli alla finalità del lavoro, ovvero la semplificazione e generalizzazione del codice KRL, l'abilitazione di procedure di rientro (recovery), la correlazione tra codice e manufatto e possibili correzioni online. Le caratteristiche qui introdotte costituiscono i vincoli di progetto che guidano, nei capitoli successivi, la riorganizzazione del programma robot e la definizione di una gestione dati più robusta lungo l'intera catena di esecuzione.

1.1 Architettura generale del sistema

La cella WAAM è composta dai seguenti elementi principali rappresentati nella figura 1.1:

1. Manufatto in deposizione – posizionato sulla tavola rotante e costruito strato su strato.

2. Robot antropomorfo KUKA KR 50 R2500-2 – con funzione di movimentazione della torcia di saldatura.

3. Tavola rotante KUKA KP1-H-2 (500 kg) – utilizzata come asse esterno E1, con rotazione continua e sincronizzata al robot per mantenere la corretta traiettoria di deposizione.

4. Martello di rolling – situato in posizione diametralmente opposta alla torcia, che agisce in continuo per la compattazione del cordone appena depositato.

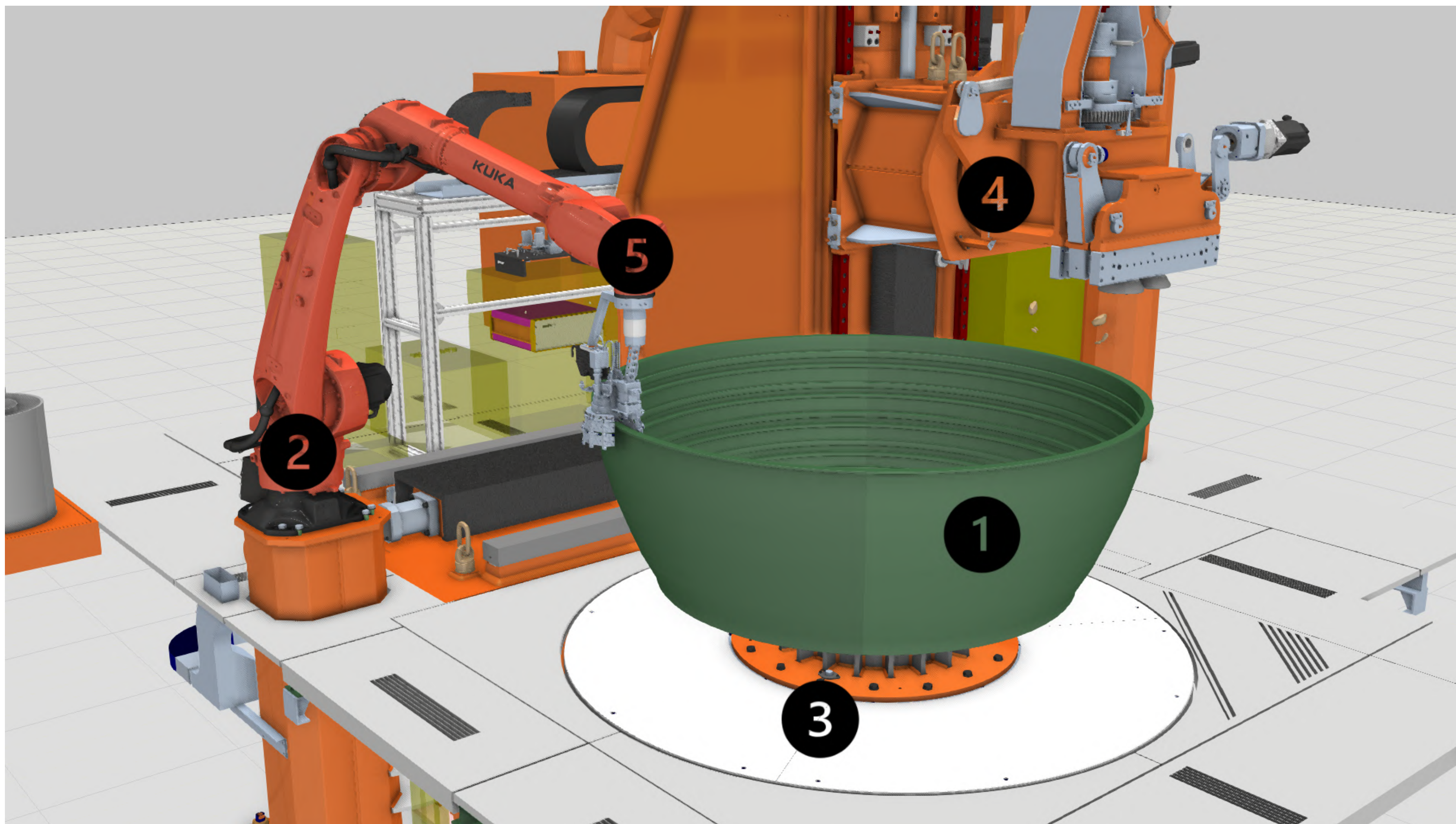

Figura 1.1: Configurazione attuale della cella WAAM presente in azienda.

5. Torcia di saldatura al plasma (PAW) – montata sull'end-effector del robot, con gas di copertura in Argon.

1.2 Funzionamento e logica di interazione

A differenza di molte configurazioni WAAM in cui è il robot a muoversi attorno al pezzo, in questa cella il braccio KUKA rimane sostanzialmente stazionario, mentre è la tavola rotante a far ruotare il componente portando progressivamente i punti di lavoro davanti alla torcia. In questo modo, la deposizione avviene in modo tangenziale e continuo, riducendo le necessità di ricalcolo delle traiettorie robotiche e semplificando il controllo della distanza torcia-pezzo. La sincronizzazione tra il movimento della tavola e la gestione del plasma è affidata al controllore KUKA (asse E1), che garantisce la coerenza cinetica tra la velocità di rotazione e il tempo di attivazione dell'arco.

Durante il processo, il martello di rolling opera in contrapposizione alla torcia, esercitando una pressione controllata sul cordone appena depositato per migliorarne la densità e la finitura superficiale. Il rolling è continuo, sincronizzato con la rotazione del pezzo e con le pause di deposizione, così da evitare eccessivo accumulo termico o deformazioni.

1.3 Ruolo della macchina nel contesto del lavoro di tesi

La macchina descritta rappresenta il caso di studio sperimentale su cui è stata sviluppata e validata la nuova logica di controllo software per il processo WAAM. L'obiettivo del lavoro non è stato la modifica fisica della cella, bensì la semplificazione e generalizzazione del codice KRL utilizzato per la deposizione, in modo da:

- migliorare la navigabilità e leggibilità del programma,
- permettere rientri automatizzati nel processo dopo interruzioni o manutenzioni,
- eliminare la rigenerazione del codice KRL per ogni modifica dei parametri del processo.

La comprensione dell'architettura meccatronica del sistema è dunque servita come base per ridefinire l'interfaccia software-hardware, migliorando la comunicazione tra i diversi moduli del processo: slicing, generazione del percorso, codice robotico e validazione post-processo.

Capitolo 2

Introduzione alla tecnologia WAAM

Dopo aver descritto nel Capitolo 1 la cella robotizzata e i vincoli impiantistici del caso studio, questo capitolo introduce il quadro tecnologico necessario per leggere il resto dell'elaborato, definendo in modo univoco termini e concetti legati al Wire Arc Additive Manufacturing (WAAM). Il WAAM è un processo di Additive Manufacturing riconducibile alla famiglia Directed Energy Deposition (DED), in cui un arco elettrico fonde un filo metallico depositato strato su strato: tale approccio risulta particolarmente interessante in ambito industriale per produttività, costo del materiale e scalabilità dei componenti, ma presenta criticità che ne rendono complessa l'industrializzazione (gestione termica, qualità superficiale e requisiti di controllo).
Il capitolo è organizzato come segue: la Sezione 2.1 inquadra origine e definizione del WAAM nel panorama dell'Additive Manufacturing; la Sezione 2.2 descrive il principio di funzionamento lungo le principali fasi operative, dalla pianificazione del processo (CAD/slicing/tool-path) alla deposizione e al post-processing; la Sezione 2.3 sintetizza le tecnologie di arco impiegabili e il loro impatto su stabilità e produttività; infine, la Sezione 2.4 riassume vantaggi, limiti e applicazioni industriali. La sintesi qui presentata consente di leggere criticamente lo stato dell'arte su slicing e controllo robotico discusso nel capitolo successivo.

2.1 Origine e definizione del WAAM

Negli ultimi anni, l'*Additive Manufacturing* (AM) si è affermato come una delle innovazioni più significative nel campo della produzione industriale, proponendo un paradigma alternativo rispetto ai metodi sottrattivi tradizionali. L'AM è un processo di fabbricazione in cui il materiale viene unito progressivamente, strato dopo strato, fino a ottenere un componente tridimensionale. Come riportato da Srivastava et al. [1], "AM can be defined as the method of joining raw materials to obtain parts against the conventional manufacturing method of material subtraction from the total mass."

Secondo la classificazione proposta dalla American Society for Testing and Materials (ASTM) [2], i processi di Additive Manufacturing si suddividono in sette categorie principali: *vat photopolymerization*, *material jetting*, *binder jetting*, *material extrusion*, *powder bed fusion* (PBF), *sheet lamination* e *directed energy deposition* (DED). All'interno di tale classificazione, le tecnologie per la fabbricazione di componenti metallici rientrano principalmente nelle categorie PBF e DED, che rappresentano i due approcci più consolidati in ambito industriale [3].

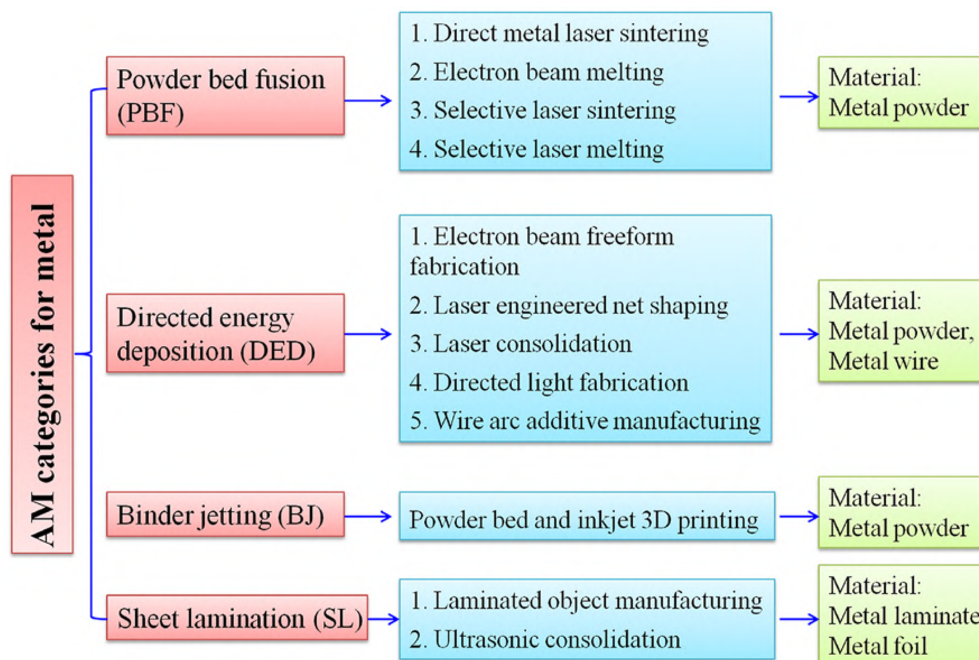

Figura 2.1: Categorie dell'Additive Manufacturing

La Figura 2.1, tratta da [3], illustra le principali categorie di AM applicate ai materiali metallici. La categoria *Powder Bed Fusion* (PBF) comprende tecniche come la *Selective Laser Sintering* (SLS) e l'*Electron Beam Melting* (EBM), che impiegano polveri metalliche fuse selettivamente tramite laser o fascio di elettroni.

La categoria *Directed Energy Deposition* (DED), invece, include processi quali *Electron Beam Freeform Fabrication* (EBF3), *Laser Engineered Net Shaping* (LENS) e il *Wire Arc Additive Manufacturing* (WAAM). Questi processi utilizzano una sorgente energetica direzionale (laser, fascio di elettroni o arco elettrico) per fondere un materiale di apporto, in forma di filo o polvere metallica, depositandolo strato su strato per generare la geometria desiderata.

Nel caso specifico del WAAM, la sorgente energetica è un arco elettrico di saldatura, mentre il materiale di apporto è un filo metallico continuo alimentato da un sistema di *wire feeding*. Il processo è assimilabile a una saldatura automatizzata multistrato, in cui la torcia segue percorsi programmati che riproducono la geometria del modello CAD. Sebbene il primo brevetto relativo a questa tecnologia risalga al 1925, solo negli ultimi due decenni il WAAM ha conosciuto un significativo sviluppo grazie all'evoluzione della robotica industriale e dei sistemi di controllo numerico [1].

L'interesse verso il WAAM è cresciuto parallelamente alla diffusione delle tecnologie DED, che nel 2020 rappresentavano circa il 16% del mercato globale dell'Additive Manufacturing [4]. Questo incremento è dovuto ai vantaggi del WAAM: elevati tassi di deposizione (fino a 10 kg/h), basso costo del materiale di apporto, uso di fili metallici standard e possibilità di realizzare componenti di grandi dimensioni con poche limitazioni geometriche. Tali caratteristiche rendono la tecnologia adatta a settori come quello aerospaziale, navale, energetico e *oil & gas*.

Dal punto di vista termico, il WAAM differisce dalla saldatura convenzionale. Nelle giunzioni tradizionali la conduzione termica è tridimensionale, mentre nel WAAM è prevalentemente direzionale verso la base del componente: "In joint welding, the thermal conductivity takes place in more dimensions than in WAAM processes, where the heat has to be deduced, in most cases, in one direction: to bottom of the part" [4]. Questa caratteristica influisce sul profilo termico e sulla microstruttura del materiale depositato, richiedendo un controllo accurato dei parametri di processo.

Un ulteriore elemento distintivo del WAAM riguarda la scelta dell'unità esecutiva. Secondo Li et al. [3], "Besides the heat source, another essential component of WAAM hardware system is the executing unit. There are generally two options: industrial robots or Computer Numerical Control (CNC) machines." I sistemi CNC offrono maggiore precisione, ma comportano costi di investimento elevati, mentre le soluzioni robotiche risultano più flessibili ed economiche, con precisioni tipiche comprese tra 0.5 e 2 mm: "The cost of CNC machine is much higher than the robot. [...] The robotic-based WAAM has better economy than the CNC-based WAAM" [3].

Quando il sistema di alimentazione del filo è separato dalla torcia di saldatura, la torcia viene ruotata per mantenere la consistenza del cordone; in tali casi, l'impiego di un braccio robotico è preferibile per la maggiore libertà di movimento: "When a separate wire feed system is used in WAAM, [...] the robot is more preferred due to its flexibility" [3].

2.2 Principio di funzionamento

Il processo di *Wire Arc Additive Manufacturing* (WAAM) comprende un insieme di fasi integrate che vanno dalla modellazione digitale del componente alla deposizione strato-su-strato, fino ai trattamenti successivi di finitura e miglioramento delle proprietà meccaniche e microstrutturali [1].

2.2.1 Pianificazione del processo

Il flusso operativo inizia con la modellazione tridimensionale del componente mediante software CAD. Il modello viene esportato in formato `.STL`, standard per le tecnologie additive, e successivamente elaborato tramite software di *slicing*. Quest'ultimo suddivide la geometria 3D in strati bidimensionali di spessore definito e genera il percorso di deposizione ottimale (*tool-path planning*), stabilendo la sequenza dei cordoni, le direzioni di avanzamento e i parametri geometrici principali.

Una corretta pianificazione è essenziale per garantire qualità dimensionale, ridurre l'accumulo di errori e limitare le deformazioni. Al termine, il sistema genera un file di istruzioni numeriche (CNC o KRL) contenente i comandi di deposizione — traiettorie, velocità, potenza dell'arco e alimentazione del filo — da eseguire dal robot o dalla macchina CNC.

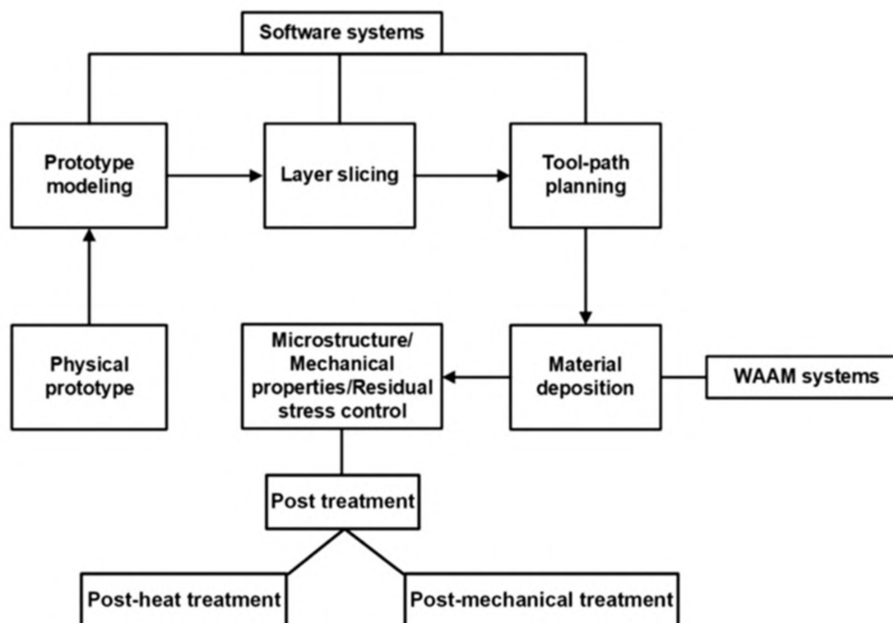

Figura 2.2: Fasi del processo WAAM

2.2.2 Deposizione del materiale

La fase di deposizione costituisce il cuore del processo WAAM. In un sistema tipico, la deposizione è eseguita da un braccio robotico dotato di torcia di saldatura e sistema di alimentazione del filo (*wire feeder*). L'arco elettrico fonde il filo metallico che viene depositato sul substrato sotto forma di cordone. Ogni nuovo strato viene costruito sopra il precedente, seguendo le traiettorie definite dal file di controllo, fino alla realizzazione completa del componente tridimensionale.

Un aspetto critico è la gestione termica interpass, poiché l'accumulo di calore può compromettere la stabilità dimensionale e generare tensioni residue. Per ridurre tali effetti, i software di *slicing* avanzati includono funzioni per:

- regolare dinamicamente la larghezza e la distanza tra i cordoni;
- prevenire sovrapposizioni eccessive tra bead successivi;
- ottimizzare la sequenza di riempimento per ridurre i tempi di spostamento;
- inserire pause di raffreddamento controllate tra i passaggi.

Inoltre, come evidenziato da Srivastava et al. [1], i moderni software di pianificazione consentono la configurazione di GPIO (*General-Purpose Input/Output*) per attivare o disattivare automaticamente il generatore di saldatura e impostare parametri di processo in base ai dati del modello.

Nei sistemi più evoluti, l'unità esecutiva può essere integrata con una tavola rotante (*rotary table*) per aumentare la libertà di movimento e con un sistema di alimentazione del filo separato, che consente di mantenere costante la geometria del cordone anche su superfici complesse.

2.2.3 Controllo delle proprietà e post-processing

Al termine della deposizione, il pezzo grezzo presenta una superficie rugosa e una microstruttura anisotropa, dovuta alla solidificazione direzionale. Per questo motivo sono necessari trattamenti successivi volti a migliorare la qualità e le prestazioni del componente.

Come mostrato in Figura 2.2, il *post-processing* comprende:

- **Trattamenti termici**, finalizzati a ridurre le tensioni residue e affinare la microstruttura;
- **Trattamenti meccanici**, come fresatura o rettifica, per raggiungere le tolleranze dimensionali e la finitura superficiale richieste.

Durante questa fase vengono analizzate la microstruttura, la diluizione tra i cordoni e le proprietà meccaniche (resistenza, durezza, tenacità), per verificare la conformità del componente agli standard di qualità. I dati ottenuti vengono spesso utilizzati per ottimizzare i parametri di deposizione nei cicli successivi, realizzando un processo di miglioramento continuo che integra modellazione CAD, pianificazione, deposizione e controllo qualità.

2.3 Tecnologie di arco utilizzabili nel processo WAAM

Il processo di *Wire Arc Additive Manufacturing* (WAAM) può impiegare diverse tipologie di sorgenti ad arco, ciascuna caratterizzata da specifiche modalità di generazione dell'arco elettrico, trasferimento del materiale e controllo termico.

Come illustrato in Figura 2.3, le tecnologie di saldatura impiegate nell'Additive Manufacturing si suddividono principalmente in quattro categorie: basate su fascio di elettroni, basate su laser, a stato solido e basate su arco elettrico. È quest'ultima classe, la *Arc-Based Welding*, a costituire la base del WAAM, comprendendo varianti come la *Gas Metal Arc Welding* (GMAW), la *Gas Tungsten Arc Welding* (GTAW), la *Cold Metal Transfer* (CMT) e la *Plasma Arc Welding* (PAW) [3].

2.3.1 Gas Tungsten Arc Welding (GTAW o TIG)

Il processo *Gas Tungsten Arc Welding* (GTAW), noto anche come *Tungsten Inert Gas* (TIG), impiega un elettrodo non consumabile in tungsteno per generare l'arco tra elettrodo e pezzo, con protezione mediante gas inerte (argon o elio). Questo metodo è apprezzato per l'elevata qualità metallurgica e la precisione, rendendolo adatto a materiali difficili da saldare e applicazioni di alta precisione. Tuttavia, presenta bassi tassi di deposizione (inferiori a 1.5 kg/h) e maggiore complessità nel controllo del percorso di deposizione rispetto ad altri processi, limitandone l'impiego a produzioni di piccola scala.

2.3.2 Gas Metal Arc Welding (GMAW o MIG)

Il processo *Gas Metal Arc Welding* (GMAW), o *Metal Inert Gas* (MIG), è tra i più diffusi nella tecnologia WAAM. L'arco si forma tra un elettrodo consumabile e il pezzo, mentre

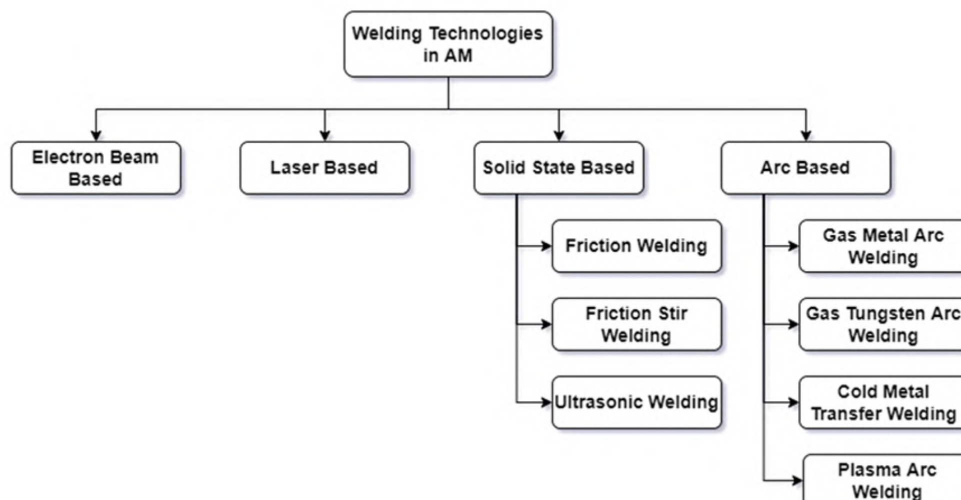

Figura 2.3: Principali tecnologie di saldatura utilizzate nell'Additive Manufacturing

un gas protettivo (inerte o attivo) previene l'ossidazione del bagno di fusione.

Come riportato da [1], "Tool path generation and programming are easier in GMAW than GTAW and PAW as both have an external wire feeding system. Two to three times higher deposition rate is provided by GMAW which is the highest among all."

Il GMAW garantisce quindi efficienza elevata, velocità di deposizione 2–3 volte superiori rispetto ai processi TIG o PAW e una maggiore semplicità di controllo. Secondo [3], "GMAW is increasingly utilized as an additive manufacturing technique since it overcomes most of the limitations of conventional AM methods. This is affordable, has high deposition rates, can work for many metals and can be used for fabricating medium to large sized parts." Grazie a tali caratteristiche, il GMAW rappresenta la tecnologia di riferimento per sistemi WAAM robotizzati e per componenti metallici di medie e grandi dimensioni.

2.3.3 Cold Metal Transfer (CMT)

La tecnologia *Cold Metal Transfer* (CMT) è una variante evoluta del GMAW, sviluppata da Fronius per ridurre l'apporto termico e migliorare la stabilità dell'arco. Il principio si basa su un controllo digitale sincronizzato tra l'arco e l'alimentazione del filo: il filo avanza e si ritrae ciclicamente, interrompendo il cortocircuito e riducendo il calore trasferito al pezzo.

Come descritto in [4], "The 'cold metal transfer' (CMT), developed by the company Fronius, is the most used process for WAAM based on gas metal arc welding (GMAW) in the literature." Il CMT consente un apporto termico ridotto del 40–60% rispetto al GMAW tradizionale e una deposizione stabile anche su materiali sensibili alle deformazioni, come leghe di alluminio e titanio. È quindi la soluzione preferita per applicazioni che richiedono alta precisione geometrica, basso stress termico e qualità metallurgica elevata.

2.3.4 Plasma Arc Welding (PAW)

Il processo *Plasma Arc Welding* (PAW) deriva dal TIG e utilizza un getto di plasma concentrato come sorgente di energia. Come riportato da [3], "PAW is similar to GTAW, but the PAW arc is more focused than a GTAW arc. It improves arc stability, increases heat transfer efficiency, and promotes welding speeds." Grazie alla maggiore densità energetica, il PAW offre elevata stabilità dell'arco e minima deformazione del pezzo, sebbene presenti una maggiore complessità e costi superiori rispetto a GMAW o CMT. È impiegato principalmente in applicazioni ad alta precisione, come quelle aeronautiche.

2.3.5 Sintesi

La scelta della tecnologia di arco nel WAAM dipende dal bilanciamento tra velocità di deposizione, qualità metallurgica e stabilità del processo:

- Il **GMAW** è la soluzione più diffusa, grazie alla produttività e versatilità;
- Il **CMT** è preferito per applicazioni ad alta precisione e basso apporto termico;
- I processi **TIG** e **PAW** trovano impiego in ambiti dove è richiesta qualità metallurgica superiore.

La comprensione delle diverse tecniche di saldatura e del loro impatto sul processo di deposizione costituisce la base per l'ottimizzazione del WAAM e per lo sviluppo di strategie di controllo più efficienti.

2.4 Vantaggi, limiti e applicazioni della tecnologia WAAM

La tecnologia Wire Arc Additive Manufacturing (WAAM) presenta una serie di vantaggi significativi rispetto ad altre tecniche di Additive Manufacturing metallico, che la rendono particolarmente attrattiva sia per la produzione di componenti di grande scala sia per la riparazione di parti esistenti. Come evidenziato nel confronto riportato nella figura 2.4, il WAAM si distingue per elevata velocità di costruzione, ampia flessibilità di piattaforma, grande volume di lavoro disponibile e notevole efficienza nell'utilizzo del materiale, a fronte di una minore accuratezza dimensionale e di una qualità superficiale inferiore rispetto ai processi a letto di polvere (Powder Bed Fusion).

2.4.1 Vantaggi del WAAM

Il principale punto di forza del WAAM risiede nella sua elevata produttività. Il tasso di deposizione raggiunge valori compresi tra 15 e 130 g/min, equivalenti, al massimo, a 8 kg/h, risultando decisamente superiore rispetto ai processi di Additive Manufacturing basati su polveri [1].

Questa caratteristica, unita all'uso di filo metallico a basso costo, rende il WAAM una tecnologia altamente economica, poiché:

- il costo del filo è circa un terzo o la metà rispetto al costo della polvere metallica di pari composizione [3];

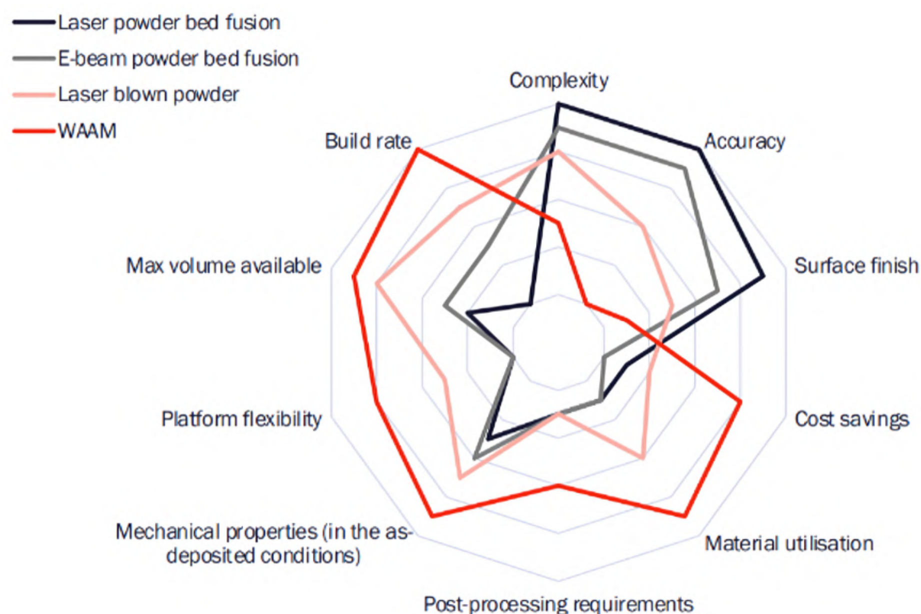

Figura 2.4: Confronto tra le principali tecnologie di Additive Manufacturing per metalli

- non è necessario un ambiente chiuso o sistemi di filtrazione per la manipolazione della polvere, riducendo ulteriormente i costi di sicurezza e gestione.

Il WAAM consente quindi di produrre componenti su scala molto ampia (oltre 2–3 m di lunghezza), ma risulta al contempo adatto anche per produzioni di piccola serie o riparazioni grazie al basso investimento iniziale richiesto. Un ulteriore vantaggio risiede nella superiorità metallurgica e meccanica dei pezzi prodotti: "WAAM products are superior in terms of mechanical and metallurgical properties when compared to conventional AM processes" [1]

Ciò è dovuto alla maggiore densità del materiale depositato, all'assenza di porosità derivante dall'uso di polveri e ai più bassi gradienti di raffreddamento, che favoriscono una microstruttura più omogenea [4].

Infine, l'elevata flessibilità della piattaforma robotica rende la tecnologia facilmente riconfigurabile: è possibile adattare la traiettoria della torcia e l'orientamento del filo per realizzare geometrie complesse e superfici organiche, difficilmente ottenibili con le tecniche tradizionali.

2.4.2 Limiti e criticità

Nonostante i numerosi vantaggi, il WAAM presenta ancora alcune limitazioni intrinseche che ne condizionano la diffusione su larga scala. La principale criticità riguarda la qualità superficiale: la rugosità elevata e la geometria del cordone richiedono lavorazioni di finitura post-processo (fresatura, tornitura o rettifica) per ottenere le tolleranze dimensionali desiderate. "These advantages come at the cost of surface finish which requires machining/post processing. Post processing can further enhance the quality of WAAM artefacts" [1]

Un ulteriore limite è rappresentato dalla precisione geometrica inferiore rispetto ai processi Powder Bed Fusion (PBF), che restano preferibili per la produzione di componenti ad alta complessità e piccoli dettagli.

Altre problematiche riguardano la gestione termica, a causa dell'elevato calore introdotto dall'arco, e la necessità di controllo attivo per prevenire deformazioni o tensioni residue. Tuttavia, la ricerca recente sta affrontando questi limiti tramite strategie di controllo adattivo, sensori di processo e modelli termici predittivi.

2.4.3 Applicazioni industriali del WAAM

Grazie alla sua versatilità e convenienza economica, il WAAM è oggi impiegato in numerosi settori industriali:

- Aerospaziale: per la produzione di staffe strutturali, supporti e parti di grandi dimensioni in leghe di titanio o Inconel, con una riduzione fino al 60% del materiale di scarto rispetto alle lavorazioni sottrattive.
- Navale e offshore: per la costruzione e riparazione di eliche, scafi, supporti e componenti di piping in acciaio inossidabile.
- Energetico: per la produzione di pale di turbina e componenti di centrali idroelettriche.
- Automotive e racing, dove viene utilizzato per prototipi funzionali e strutture personalizzate.

Capitolo 3

Stato dell'arte: slicing e controllo robotico nel WAAM

A valle dell'inquadramento tecnologico del WAAM presentato nel Capitolo 2 e della descrizione dell'impianto reale nel Capitolo 1, questo capitolo colloca il lavoro di tesi nel panorama della ricerca e delle soluzioni oggi disponibili, evidenziando un limite ricorrente: la mancanza di un ecosistema software realmente integrato e standardizzato per la pianificazione e l'esecuzione del processo WAAM. In particolare, la generazione delle traiettorie non può essere ricondotta a un semplice slicing "geometrico" tipico della stampa polimerica, poiché deve incorporare vincoli termici, scelte di strategia deposizionale e gestione degli eventi di processo (start/stop, transizioni, configurazioni locali), motivo per cui in molti casi si ricorre a strumenti parametrici e personalizzati (ad es. ambienti CAD/visual programming come Grasshopper). Parallelamente, il controllo robotico sta evolvendo da esecuzione offline di percorsi statici verso architetture di monitoraggio e controllo più adattive, in cui sensori e robot devono scambiarsi informazioni in modo affidabile e, potenzialmente, in tempo reale.
Il capitolo è articolato in due parti: la Sezione 3.1 analizza lo stato dell'arte su slicing e toolpath planning per WAAM, con attenzione alle strategie deposizionali e alle esigenze di personalizzazione; la Sezione 3.2 sintetizza i principali approcci di monitoraggio e controllo (sensoristica, controllo a ciclo chiuso e integrazione middleware) e le implicazioni architetturali sullo scambio dati tra moduli. Nel complesso, l'analisi mette in evidenza la necessità di soluzioni più flessibili e interconnesse rispetto ai tradizionali workflow basati su file statici, fornendo i presupposti concettuali per le scelte di sviluppo software e riorganizzazione del controllo presentate nei capitoli successivi.

3.1 Slicing e pianificazione della traiettoria

La generazione del percorso utensile (toolpath) per il WAAM parte, come per molti processi di Additive Manufacturing, da un modello CAD tridimensionale del componente, generalmente esportato in formato .STL. Tuttavia, la conversione di questo modello in istruzioni eseguibili da un sistema robotico WAAM presenta complessità specifiche non riscontrabili nei processi FDM o nei tradizionali software CAM.

La fase di slicing non si limita a dividere il modello in strati, ma deve considerare le peculiarità del processo di saldatura ad arco, tra cui l'interazione termica tra cordoni adiacenti, la gestione degli overhang e la minimizzazione dei difetti di fusione. Come

sottolineato da Sebok et al. [5], "existing toolpath generation options typically lack the appropriate features to account for all complexities of the WAAM process", evidenziando la necessità di strumenti di slicing dedicati, capaci di gestire la variabilità fisica del processo e l'adattamento locale dei parametri di deposizione.

Le strategie di slicing, infatti, possono essere viste come una forma di controllo a priori del processo, in particolare per quanto riguarda la gestione termica, che viene poi affinata dalle strategie di controllo in tempo reale discusse successivamente. Il metodo più diffuso resta lo slicing planare, in cui il modello 3D viene intersecato da piani paralleli per generare contorni bidimensionali successivi. A seconda della strategia di riempimento scelta, raster, zig-zag, contour-parallel (offset), concentrici o spiraliformi, si possono ottenere distribuzioni termiche e morfologiche differenti. Approcci continui, come la spirale o il percorso concentricamente connesso, riducono il numero di inneschi e spegnimenti dell'arco, migliorando la stabilità termica e riducendo difetti localizzati [5].

Nei software più evoluti, come quelli sviluppati in ambiente Rhino/Grasshopper, la generazione del toolpath è completamente parametrica: consente di adattare dinamicamente la densità dei cordoni, l'ordine di deposizione e le condizioni di start/stop in funzione della geometria locale. Peter et al. [6] sottolineano infatti come "programming manufacturing processes with Rhino 7 and Grasshopper provides the advantages of customization for process-dependent considerations", e come, grazie a plugin quali KUKA|prc, slicing, simulazione e generazione del codice KRL possano essere integrati in un unico ambiente.

I software di slicing specifici o customizzati spesso distinguono tra diversi tipi di cordoni per riempire la sezione del layer, come insets (perimetrali), skeletons (per riempire gap) e infills (riempimento interno), la cui combinazione e spaziatura ottimale è essenziale per la densità del layer, data la geometria tipicamente parabolica del cordone WAAM.

Molti ricercatori hanno sviluppato soluzioni software custom per affrontare le peculiarità del WAAM [5]:

- Gestione degli Overhang: La gravità limita gli angoli di overhang. Soluzioni includono l'adattamento dell'angolo della torcia e lo spostamento laterale del toolpath (path shifting) , permettendo overhang significativi. Strategie di slicing non planare o multi-asse mantengono l'orientamento ottimale del bagno fuso.
- Gestione degli Angoli Acuti: Per evitare vuoti, funzionalità di "corner sharpening" modificano localmente il toolpath estendendo i segmenti di cordone.
- Punti di Start/Stop: L'irregolarità geometrica dovuta ai transienti di innesco/spegnimento dell'arco richiede una configurazione attenta della posizione dei punti di start/stop (es. costanti, rotanti, random, in aree specifiche). Tecniche come il "tip wipe" (estensione del percorso) migliorano la chiusura dei loop.
- Controllo della Lunghezza dei Cordoni: Soglie minime e massime per la lunghezza dei cordoni e per il riempimento dei gap ottimizzano la deposizione e gestiscono la manutenzione della torcia.

- Gestione Termica tramite Path Planning: Strategie come l'"island optimization" (ottimizzazione dell'ordine di deposizione di geometrie disconnesse) o percorsi continui come la spirale o l'"inset connesso" aiutano a bilanciare la dissipazione termica e ridurre deformazioni.

- Slicing Multi-Materiale: Lo slicing deve gestire layer height differenti per materiali diversi, ordinando correttamente la sequenza.

- Smoothing del Percorso: Algoritmi come Douglas-Peucker riducono il numero di punti nel G-Code, utile per grandi componenti e per ridurre le accelerazioni.

Xia et al. [7] evidenziano come "although major progress has been made in process development and path slicing, a comprehensive process monitoring and control system is yet to be developed", indicando che la gestione termica e la compensazione geometrica rimangono i principali fronti di ricerca aperti.

La mancanza di software commerciali WAAM completi spinge all'uso di piattaforme parametriche (Rhino/Grasshopper) o ambienti di programmazione (MATLAB, Python) per algoritmi custom. Plugin come KUKA|prc o soluzioni open-source facilitano l'integrazione con robot e simulazione. Questo è riportato anche da Peter et al. [6], "existing slicing and path planning software for WAAM provide slicing capabilities but restrict fine adjustment of path plans and are costly", confermando che la ricerca di un ecosistema software realmente integrato per il WAAM è ancora in corso.

3.2 Monitoraggio e controllo robotico nel WAAM

La variabilità intrinseca dei processi di saldatura ad arco, unita ai complessi fenomeni termici e metallurgici coinvolti, rende essenziale l'adozione di sistemi di monitoraggio e controllo a ciclo chiuso per garantire la stabilità del processo e la qualità dei manufatti. Il controllo robotico nel WAAM si basa sulla capacità di osservare e correggere in tempo reale parametri chiave come la temperatura del bagno fuso, la larghezza e l'altezza del cordone, e la stabilità dell'arco. Li et al. [8] descrivono come nel WAAM siano impiegate diverse tecniche di monitoraggio, ottiche, acustiche, termiche ed elettriche, finalizzate al controllo in tempo reale del processo e alla prevenzione dei difetti, evidenziando il ruolo cruciale della sensoristica nella qualità del processo.

Diversi sistemi sensoriali sono stati investigati [8]:

- Sensori Visivi (CCD, CMOS, Termocamere, Laser Scanner): Utilizzati per monitorare la geometria del cordone (altezza, larghezza) , la dinamica del bagno fuso , la posizione del filo, la lunghezza dell'arco e i difetti superficiali. Termocamere IR monitorano la distribuzione termica e la temperatura interpass. Laser scanner 3D misurano il profilo del layer.

- Sensori Spettrali: Analizzano lo spettro dell'arco per informazioni su composizione del plasma, stabilità e contaminanti.

- Sensori Acustici: Correlano le emissioni acustiche a stabilità dell'arco, trasferimento metallico e difetti interni (cricche, porosità).

- Sensori Elettrici: Monitorano corrente e tensione d'arco per la stabilità del processo.
- Sistemi Multi-Sensore e Fusione Dati: L'integrazione di sensori diversi è considerata essenziale per un monitoraggio robusto. L'architettura ECS (Equivalent Contact Surface) mira a quantificare la dissipazione termica aggregando dati geometrici e di processo.

Dal punto di vista del controllo, gli algoritmi proposti in letteratura spaziano da approcci PID e fuzzy logic fino a metodi predittivi basati su modello (MPC) e tecniche di apprendimento iterativo o reti neurali [7]. Questi sistemi mirano principalmente a regolare l'altezza del layer e la potenza dell'arco, parametri che determinano direttamente l'accuratezza geometrica e le proprietà metallurgiche del pezzo. He et al. [9] hanno recentemente presentato un sistema multi-robot denominato Scan-n-Print, in cui un robot secondario esegue la scansione ottica del layer appena depositato, mentre il principale aggiorna in tempo reale la traiettoria del successivo, migliorando significativamente la fedeltà geometrica del componente.

Dal punto di vista architetturale, la tendenza è verso sistemi integrati e interoperabili, nei quali robot, sensori e controllori comunicano attraverso middleware aperti (come Robot Raconteur o OPC UA) che consentono lo scambio dati in tempo reale [10]. Questa integrazione permette di implementare controlli adattivi distribuiti, in cui la correzione dei parametri di processo avviene direttamente a bordo del controllore robotico, in funzione delle informazioni provenienti dai sensori. Come riportato da He et al. [10], il sistema sviluppato integra due robot industriali, una suite sensoriale e il controllore di saldatura mediante un framework middleware (Robot Raconteur), che consente la comunicazione e il coordinamento in tempo reale tra tutti i moduli del processo., evidenziando il ruolo centrale della comunicazione e del coordinamento tra moduli.

In conclusione, lo stato dell'arte nel controllo robotico del WAAM mostra una chiara convergenza verso approcci adattivi, intelligenti e multi-sensore, in cui il robot non è più un semplice esecutore di traiettorie, ma un nodo attivo di un sistema cognitivo capace di percepire, valutare e correggere in autonomia le deviazioni del processo. Nonostante i progressi significativi raggiunti, la realizzazione di sistemi completamente automatizzati, robusti e riproducibili rimane un obiettivo ancora aperto, che costituisce la base di partenza per le attività di sviluppo software presentate nei capitoli successivi di questa tesi.

Capitolo 4

Analisi del processo esistente e sviluppo della nuova logica KRL

Alla luce delle criticità evidenziate nel Capitolo 3 rispetto ai workflow WAAM basati su file statici, questo capitolo presenta il primo intervento concreto sul sistema: l'analisi del processo esistente di generazione ed esecuzione dei programmi KRL e lo sviluppo di una nuova logica software finalizzata ad aumentarne leggibilità, manutenibilità, robustezza e flessibilità operativa. Nel workflow iniziale, i file di controllo prodotti dallo slicing risultano composti da migliaia di righe di codice monolitico, rendendo poco pratiche sia le modifiche mirate sia la gestione di interruzioni e ripartenze senza interventi manuali sul programma.
La soluzione proposta consiste nello sviluppo di script Python che trasformano la rappresentazione lineare in una logica indicizzata basata su array e cicli, separando in modo più netto dati e logica di esecuzione. Tale ristrutturazione riduce drasticamente la dimensione complessiva dei file (circa 78%), ma soprattutto abilita un accesso ai waypoint per indici logici (bead/segmento) che avvicina il codice al business, andando a descrivere meglio ed in maniera più comprensibile il manufatto. Questo rende possibile un dialogo semplificato con il cliente, per la gestione dei suoi requisiti sul pezzo. Inoltre l'accesso ai waypoint per indici logici è un prerequisito per implementare procedure di recovery automatico: il robot può così riprendere la deposizione da un punto determinato dopo un'interruzione, calcolando traiettorie di rientro sicure senza modifiche manuali del codice.
Il capitolo è organizzato come segue: la Sezione 4.1 descrive il workflow esistente (slicing e generazione KRL); la Sezione 4.2 ne discute le criticità; la Sezione 4.3 dettaglia gli script sviluppati per la riorganizzazione dei file .dat e .src; la Sezione 4.4 illustra i vantaggi della nuova logica; la Sezione 4.5 verifica l'equivalenza geometrica dei waypoint tra versione originale e versione "a vettori"; mentre la Sezione 4.6 presenta la progettazione della logica di recovery. Nel complesso, i risultati qui ottenuti costituiscono la base tecnica per le evoluzioni architetturali discusse nei capitoli successivi, in particolare per il passaggio da un sistema basato su file statici a un'architettura di streaming PC–PLC–KUKA.

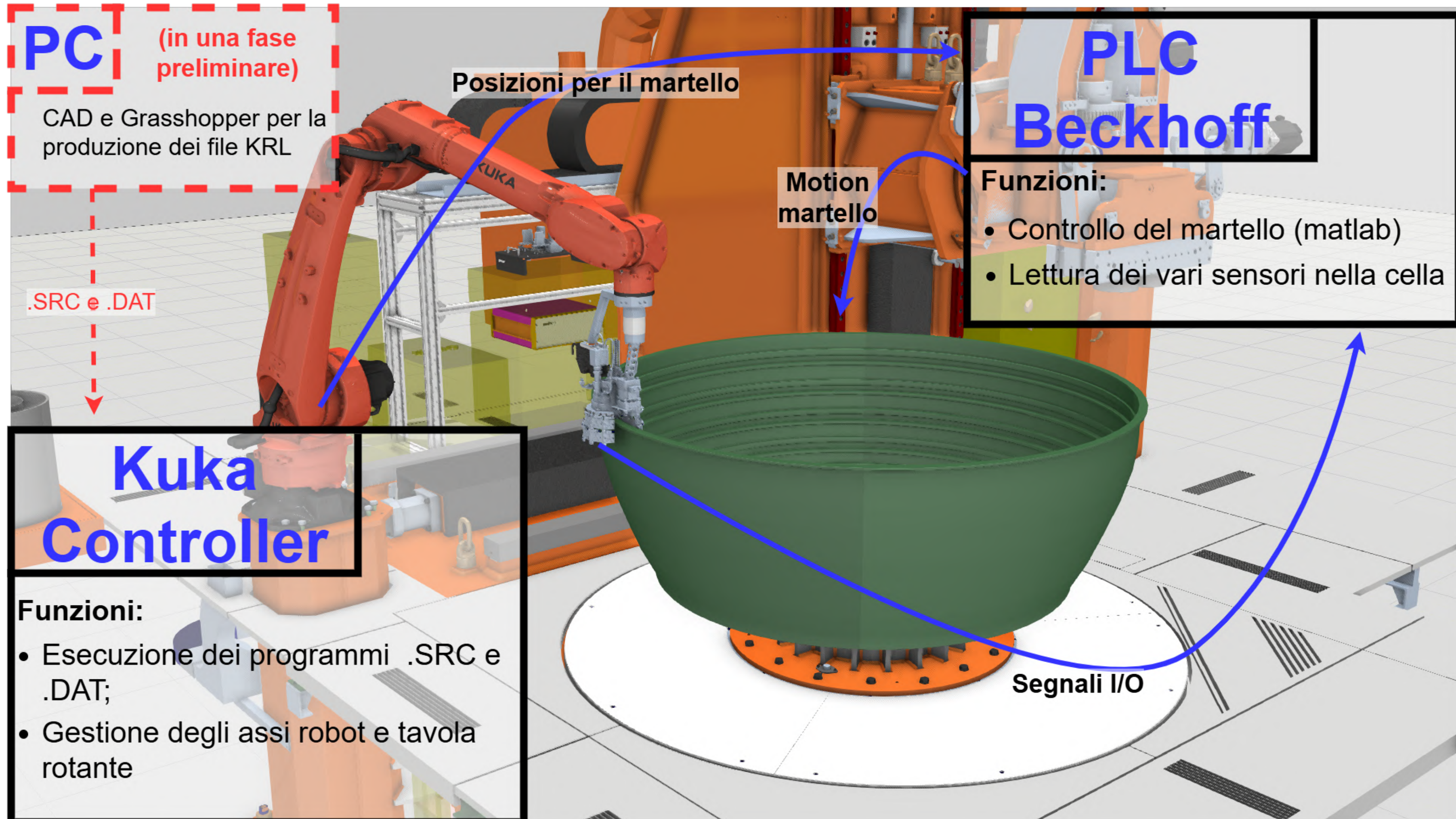

Figura 4.1: Schema a blocchi della catena di controllo WAAM attuale, basata su file KRL statici.

4.1 Descrizione del workflow esistente: slicing e generazione del codice KRL

Il processo per passare dal modello CAD del manufatto alla realizzazione del componente in uso attualmente si articola in diverse fasi, riassunte graficamente nella Figura 4.1.Nel seguente capitolo si analizza il passaggio dal modello CAD del manufatto al programma eseguibile dal robot KUKA, processo prevalentemente gestito all'interno dell'ambiente di progettazione parametrica Rhinoceros con il suo plugin Grasshopper.

- **Modellazione e slicing in Grasshopper**: Il modello CAD 3D del componente viene importato in Grasshopper, si può vedere la sezione e un suo ingrandimento dell'area di slicing nelle figure A.1 e A.2 . Qui, uno script dedicato, sviluppato in collaborazione con WAAM3D, visibile nell'immagine A.3 , esegue lo slicing del manufatto in layer orizzontali. L'utente configura parametri specifici per diverse "aree" (da qui in avanti chiamate sezioni) della sezione trasversale, in base alle caratteristiche richieste dal cliente (es. numero di bead, sovrapposizione, altezza). Lo script genera quindi i punti di deposizione per ciascun layer.

- **Generazione file intermedi (`.cpi`)**: Un output intermedio significativo del processo di slicing in Grasshopper è costituito da un file con estensione `.cpi`. Ogni riga di questi file, come si può notare nell'estratto in B.1, descrive un singolo punto della traiettoria, arricchito da parametri essenziali per la produzione (es. coordinate, orientamento torcia, parametri di saldatura specifici per quel punto). Vi è un esempio di una riga del file `.cpi` nella tabella 4.1 e nella tabella 4.2. Come si può notare dalle seguenti tabelle, le variabili dalla prima alla dodicesima sono state identificate e correlate al loro significato; mentre per le successive variabili, non è stato possibile trovare un significato per ogni singola variabile ma sicuramente contengono variabili di servizio come tipo di materiale, gas, layer e sezione correnti ecc.

Tabella 4.1: Formato riga `.cpi`: campi 1–12 (fissi)

Idx	Campo	Unità	Descrizione	Esempio
1	X locale	mm	Posizione TCP asse X nel sistema locale	468.04
2	Y locale	mm	Posizione TCP asse Y nel sistema locale	189.10
3	Z locale	mm	Posizione TCP asse Z nel sistema locale	0.20
4	A	deg	Orientamento—angolo A (roll)	180.00
5	B	deg	Orientamento—angolo B (pitch)	0.00
6	C	deg	Orientamento—angolo C (yaw)	-68.00
7	Angolo torcia	deg	Angolo torcia (torch angle)	180.00
8	E1	deg	Asse esterno E1 (in gradi)	338.00
9	Arc event	—	Evento arco: 0 = a regime, 1 = accensione, 2 = spegnimento	0
10	Corrente	A	Corrente di saldatura	230
11	Wire Feed Rate	m/min	Velocità filo	2.20
12	Travel speed	mm/s	Velocità di avanzamento	4.98

Tabella 4.2: Formato riga `.cpi`: campi 13–fine (variabili di servizio)

Idx	Campo (tipo)	Descrizione	Esempio
13	Service #1	Variabile di servizio	151
14	Service #2	Variabile di servizio	0
15	Service #3	Variabile di servizio	1
16	Service #4	Variabile di servizio	1
17	Service #5	Variabile di servizio	0
18	Service #6	Variabile di servizio	2
19	Service #7	Variabile di servizio	8
20	Service #8	Variabile di servizio	0
21	Service #9	Variabile di servizio	2
22	Service #10	Variabile di servizio	-360.00
23	Service #11	Variabile di servizio	1
24	Service #12	Variabile di servizio	0
25	Service #13	Variabile di servizio	1.40

Le coordinate X,Y e Z locali vengono calcolate dalle coordinate globali che descrivono il punto in cui la torcia si posiziona durante un'intera rotazione della tavola rotante. Di seguito vengono mostrati i calcoli necessari per il passaggio dal sistema globale a quello rotante:

$$\mathbf{P}_{\text{global}} = R_z(\phi)\,\mathbf{P}_{\text{local}} \quad \Longleftrightarrow \quad \begin{bmatrix} 504.80 \\ 0 \\ 0.20 \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 468.04 \\ 189.10 \\ 0.20 \end{bmatrix}$$

Sia $\{G\}$ il riferimento *globale* fissato alla cella e $\{L\}$ il riferimento *locale* solidale alla tavola rotante. Il vettore $\mathbf{P}_{\text{global}} \in \mathbb{R}^3$ rappresenta le coordinate del TCP in $\{G\}$, mentre $\mathbf{P}_{\text{local}} \in \mathbb{R}^3$ sono le coordinate dello stesso punto espresse in $\{L\}$. Se l'origine di $\{L\}$ coincide con quella di $\{G\}$ (nessuna traslazione), la trasformazione è una pura rotazione attorno all'asse Z di ampiezza ϕ:

$$\boxed{\mathbf{P}_{\text{global}} = R_z(\phi)\,\mathbf{P}_{\text{local}}} \qquad R_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

La rotazione $R_z(\phi)$ agisce sul piano XY lasciando invariata la quota Z (del TCP), come si vede dall'elemento $(3,3)$ pari a 1.

Nel caso specifico, facendo riferimento alla variabile $E1$ nella tabella 4.1, l'angolo della tavola è $\phi = 338°$. Sostituendo tale valore nella matrice $R_z(\phi)$ e moltiplicando per il vettore locale del punto si ottengono le coordinate globali $\mathbf{P}_{\text{global}}$ riportate nei file, confermando la coerenza tra rappresentazione locale (solidale alla tavola) e globale (fissa alla cella).

- **Generazione codice KRL** (`.src` e `.dat`): Successivamente, un'ulteriore elaborazione all'interno dello stesso script Grasshopper (strutturato in parte come com-

ponente con logica proprietaria con input/output predefiniti difficile da investigare) converte le informazioni dei file `.cpi` in due file distinti per ogni layer, destinati al controllore KUKA:

- Un file `.src`: Contenente la logica di movimento KRL, le chiamate alle funzioni di saldatura e i comandi ausiliari.
- Un file `.dat`: Contenente la dichiarazione e l'inizializzazione delle strutture dati KRL (principalmente E6POS o FRAME) che definiscono le coordinate cartesiane e l'orientamento (XYZABC) di ogni punto della traiettoria per quel layer, come si può notare nell'estratto nell'Appendice B.2. Nelle tabelle 4.3 e 4.4 viene presentato il confronto con la riga precedete del file `.cpi`.

  La prima tabella mostra che le posizioni coincidono e che gli angoli sono equivalenti tenendo conto della convenzione: in questi dati, A del `.cpi` corrisponde a C del `.dat`, mentre C del `.cpi` corrisponde ad A del `.dat` mod 360°.

  La seconda tabella verifica la coerenza dei parametri di processo: corrente, WFR e velocità combaciano (attenzione all'unità: m/s nel .dat), mentre gli altri canali e i parametri di weave sono dichiarati nel blocco WDAT. Con questa mappatura, quando nella riga `.cpi` leggi Arc event = 1, il punto omologo nel `.dat` userà un blocco Strike . . . ; se Arc event = 2, userà Crater . . . ; se 0, Weld . . . .

Tabella 4.3: Confronto campi geometrici: riga `.cpi` vs `E6POS` nel `.dat`

Campo	**.cpi**	**Unità**	**.dat (E6POS)**	**Note / mappatura**
X locale	468.04	mm	X = 468.04	coincidente
Y locale	189.10	mm	Y = 189.10	coincidente
Z locale	0.20	mm	Z = 0.20	coincidente
A	180.00	deg	C = 180	*Mapping:* `.cpi` A → `.dat` C
B	0.00	deg	B = 0	coincidente
C	-68.00	deg	A = -428	*Mapping:* `.cpi` C → `.dat` A; $-428° \equiv -68° \pmod{360°}$
Angolo torcia	180.00	deg	C = 180	in questa configurazione coincide con C del `.dat`
E1	338.00	deg	E1 = 338	coincidente

4.2 Analisi delle criticità del codice KRL originale

L'analisi approfondita dei file `.src` e `.dat` generati dal workflow esistente, come esemplificato dai file `DepRoll0001.src` e `DepRoll0001.dat` (estratti riportati in Appendice B.3 e B.4) , ha evidenziato diverse criticità che limitano la flessibilità e la manutenibilità del sistema:

Tabella 4.4: Confronto parametri di processo: riga .cpi vs WDAT nel .dat

Parametro	.cpi	Unità	.dat (WDAT7)	Note / mappatura
Arc event	0/1/2	—	Weld / Strike / Crater	**Mappatura**: 0 → blocco Weld{...}, 1 → Strike{...}, 2 → Crater{...}
Corrente	230	A	230	coincidente (Channel2)
Wire feed rate	2.20	m/min	2.2	coincidente (Channel3)
Travel speed	4.98	mm/s	0.00498	stessa grandezza: 0.00498 m/s = 4.98 mm/s
Program/Job	151	—	151	ID set parametri (Channel1)
Weave length	—	mm	4.00	definito in WDAT
Weave amplitude	—	mm	2.00	definito in WDAT
Weave angle	—	deg	0.00	definito in WDAT
Altri canali	—	—	0.8 / 8.0 / 0.0...	canali aggiuntivi secondo profilo WAAM

- **Complessità e scarsa leggibilità**: I file .src originali sono estremamente lunghi e ripetitivi. Come si osserva in DepRoll0001.src, la logica di deposizione consiste in una lunga sequenza di comandi di movimento espliciti (principalmente LIN o CIRC), replicata per l'intera traiettoria di un layer. Manca l'uso di cicli o strutture di controllo avanzate, rendendo il codice difficile da leggere (spesso migliaia di righe per layer), da comprendere e, soprattutto, da modificare. Ad esempio, per depositare un singolo bead composto da 36 punti di passaggio, il file .src conteneva 18 comandi CIRC quasi identici.

- **Struttura dati inefficiente nei file .dat**: Ogni file .dat contiene la dichiarazione KRL esplicita (DECL) di ogni singola variabile di posa (E6POS XPn), dati di avanzamento (FDAT FDATn), dati locali (LDAT LDATn) e potenzialmente altri parametri per ogni punto del layer (vedere DepRoll0001.dat). Questo approccio, sebbene funzionale, risulta inefficiente per diverse ragioni:

  - Pesantezza: Genera un numero molto elevato di variabili globali o locali, potenzialmente appesantendo la memoria del controllore KUKA.
  - Difficoltà di accesso: Non permette un accesso programmatico efficiente a un punto specifico tramite indici logici (es. "punto 5 del bead 3"). L'accesso avviene solo tramite il nome univoco della variabile (es. XP532), rendendo complesso implementare logiche basate sulla posizione relativa nel percorso.

- **Difficoltà nell'implementazione di logiche avanzate**: La struttura rigida e "appiattita" del codice .src rendeva estremamente arduo inserire logiche più sofisticate, come quelle necessarie per le procedure di recovery. Identificare il punto esatto di interruzione e calcolare la traiettoria di rientro richiedeva complessi meccanismi di ricerca e manipolazione del codice KRL esistente o l'introduzione di logiche esterne complesse.

- **Limitazioni nell'uso di KUKA|prc**: Inizialmente si era valutata la possibilità di utilizzare il plugin KUKA|prc per Grasshopper per rigenerare il codice KRL in modo più strutturato. Tuttavia, questa strada è stata abbandonata a causa della difficoltà intrinseca nel gestire tramite KUKA|prc la grande quantità di parametri di processo specifici (saldatura PAW, velocità, inclinazione della torcia, parametri del cliente) associati a ciascun punto e alla logica complessa di attivazione/disattivazione della saldatura e delle funzioni ausiliarie.

4.3 Sviluppo di script Python per la semplificazione e riorganizzazione del codice KRL

Per superare le limitazioni del codice KRL originale, si è optato per lo sviluppo di due script Python dedicati (`kuka_dat_to_array_v_updated.py` e `create_simplified_src_v3.py`, riportati in estratti nell'Appendice B), che automatizzano la trasformazione dei file generati da Grasshopper in una versione KRL ottimizzata.

4.3.1 Script per la riorganizzazione del file .dat (kuka_dat_to_array_v_updated.py)

Questo script Python svolge un'analisi programmatica e una successiva ristrutturazione di un file `.dat` KRL. L'obiettivo primario è convertire una lunga sequenza di dichiarazioni di variabili individuali in un insieme di array KRL unidimensionali, ottimizzando la struttura dei dati per un'elaborazione ciclica nel programma `.src` associato.

Il processo di conversione si articola nelle seguenti fasi:

- **Parsing e acquisizione dei dati B.5**: Lo script analizza il file `.dat` di input (es. `DepRoll0001.dat`) utilizzando espressioni regolari (RegEx). Questa fase si concentra specificamente sull'identificazione e l'estrazione dei seguenti blocchi di dichiarazione:
  - `E6POS XPn`: Le posizioni cartesiane principali del robot.
  - `E6POS XPn_hover`: Pose ausiliarie, di approccio o retrazione.
  - `WDATn`: Blocchi di dati di saldatura (del tipo `stArcDat_T`).
  - `WPn`: Blocchi di parametri di saldatura (anch'essi `stArcDat_T`).
  - `R_POS RPn`: Dati posizione del martello.
  - Lo script cattura e preserva anche tutte le altre dichiarazioni `DECL` non corrispondenti a questi pattern, per garantire l'integrità del file.
- **Categorizzazione degli indici B.6**: Questa è la logica fondamentale dello script. Invece di creare un array bidimensionale, lo script implementa una categorizzazione basata su un ciclo di 41 punti. Gli indici numerici (la n in XPn) estratti vengono mappati in cinque categorie distinte:
  - init: Punti in posizione 1, 42, 83... (formula: $1+41k$)
  - main: Punti nelle posizioni 2-37, 43-78...

  - end: Punti in posizione 38, 79, 120... (formula: $38+41k$)
  - end1: Punti in posizione 40, 81, 122... (formula: $40+41k$)
  - end2: Punti in posizione 41, 82, 123... (formula: $41+41k$)

  Con $k \in \mathbb{N}$.
  La posizione 39 di ogni ciclo viene volutamente ignorata.

- **Definizione e popolamento degli Array KRL B.7**: Lo script genera un nuovo file `.dat` (es. `DepRoll0001_array.dat`) contenente le dichiarazioni per una serie di array unidimensionali. Vengono invece creati array distinti per ciascuna categoria e tipo di dato. Ad esempio:

  - `DECL E6POS XP_init[N]`
  - `DECL E6POS XP[M]` (per i punti main)
  - `DECL E6POS XP_end[N]`
  - `DECL E6POS XP_hover_init[N]`
  - `DECL stArcDat_T WDAT_init[N]`
  - `DECL stArcDat_T WDAT[M/2]`
  - ... e così via per `WP, XP_end1, XP_end2`, etc.
  - Lo script deriva inoltre la variabile bead_max conteggiando il numero totale di variabili RPn (relative position) trovate, e crea un array `DECL R_POS RP[bead_max]` corrispondente.

  Con $N = bead_max$ e $M = N * 36$.

- **Output finale**: Il risultato è un file `.dat` che sostituisce migliaia di variabili singole con un set gestibile di array. Ogni array raggruppa logicamente i punti omologhi di cicli diversi (tutti i punti di inizio, tutti i punti principali, ecc.), pronti per essere indirizzati da un programma `.src` tramite un indice di ciclo. Lo script genera anche file .json e .csv ausiliari contenenti i punti `XP` estratti, utili per scopi di verifica e analisi dei dati.

4.3.2 Script per la semplificazione del file .src (create_simplified_src_v3.py)

Questo script agisce come un generatore di codice programmatico, progettato per tradurre un file `.src` KRL esteso e lineare in una versione compatta e ciclica. Esso rappresenta la seconda metà del processo di refactoring, agendo sul programma in modo complementare a come lo script precedente ha agito sui dati.

L'obiettivo è sostituire una sequenza statica di migliaia di comandi di movimento individuali con una logica di programma dinamica, basata su loop, che opera sugli array generati nel file `_array.dat`.

Il processo è così articolato:

- **Estrazione dell'intestazione e dei parametri**: Lo script non analizza l'intera logica del file `.src` originale (es. `DepRoll0001.src`). Esegue invece un'estrazione mirata:
  - Conserva l'intestazione del programma (le prime 22 righe), che contiene la definizione `DEF...ENDDEF` e le dichiarazioni di variabili locali.
  - Ricerca e memorizza i valori di parametri specifici del file, come `Theorical_Layer_Height` e `Top_Load`, per preservare le configurazioni uniche di quel programma.
- **Definizione di un template KRL ciclico B.8**: Il nucleo dello script è un template. Questo template non viene generato, ma è codificato direttamente nello script Python nella variabile `fixed_content` e rappresenta la nuova logica di esecuzione generalizzata per un qualsiasi cordone di saldatura.
- **Implementazione della logica a loop**: Il template KRL sostituisce la programmazione lineare con due loop FOR annidati, che costituiscono il cuore della nuova logica:
  - Loop esterno (per Bead):
    `FOR bead=bead_actual TO bead_max STEP 1`
    itera attraverso i cordoni di saldatura.
  - Loop interno (per Segmenti):
    `FOR seg=seg_actual TO seg_per_bead STEP 1`
    itera attraverso i 18 segmenti di movimento che compongono un singolo cordone.
- **Indirizzamento dinamico degli array**: Questa è l'innovazione fondamentale. All'interno dei loop, i comandi di movimento (es. LIN, CIRC) non richiamano più variabili statiche (come `XP1, XP38, XP39`...). Al contrario, indirizzano dinamicamente gli elementi degli array creati dallo script precedente:
  - Le sezioni `INIZIO_BEAD` e `FINE_BEAD` utilizzano l'indice `[bead]` per accedere ai punti di inizio e fine (es. `XP_init[bead]`, `WDAT_end[bead]`, `XP_hover_end[bead]`).
  - Il loop dei segmenti calcola dinamicamente gli indici per l'array principale (es. mid_idx, end_idx) per accedere ai punti `XP[mid_idx]` e `XP[end_idx]`, eseguendo la traiettoria di saldatura.
- **Gestione del recovery**: Il template include una logica `IF recovery_mode THEN`... che permette di riavviare l'esecuzione non dall'inizio, ma da uno specifico cordone (b_{Selected}) e segmento (s_{Selected}), capacità essenziale nei processi di produzione lunghi.
- **Assemblaggio del file finale B.9**: Lo script conclude il processo scrivendo il nuovo file `.src` (es. `DepRoll0001_semplificato.src`). Questo file è composto da:
  - L'intestazione originale (prime 22 righe).
  - Il template KRL ciclico.

– I valori dei parametri (`Theorical_Layer_Height`, `Top_Load`) estratti all'inizio, che vengono iniettati nelle posizioni corrette all'interno del template.

In sintesi, questo script funge da motore di "templating": svuota il programma `.src` originale della sua logica di movimento ripetitiva, preservandone solo l'intestazione e i parametri unici, e la sostituisce con una logica a loop compatta, efficiente e parametrizzata per operare sulla nuova struttura dati ad array.

4.4 Vantaggi della nuova logica KRL

L'adozione della nuova logica KRL, generata automaticamente dagli script Python di refactoring, non rappresenta un semplice "accorciamento" del programma robot, ma un cambio di paradigma: si passa da una programmazione lineare e monolitica a una logica strutturata basata su array, cicli e indici logici (bead, segmento). Questa scelta nasce come risposta diretta ai limiti del workflow file-based tradizionale (rigidità alle modifiche, scarsa tracciabilità e difficoltà di gestione di geometrie complesse e recovery), evidenziati nell'introduzione.

Riduzione della complessità e miglioramento della manutenibilità

Il beneficio più immediato è la drastica riduzione della dimensione complessiva del codice e, di conseguenza, della sua complessità. Nel caso di studio, la somma dei file originali (DepRoll0001.src e DepRoll0001.dat) è pari a 5212 righe, mentre la versione rifattorizzata (DepRoll0001_semplificato.src e DepRoll0001_array.dat) scende a 1133 righe: una riduzione di 4079 righe, circa 78.26%. Questo risultato, ottenuto introducendo due cicli FOR annidati al posto di migliaia di comandi ripetuti, rende il programma più leggibile, più semplice da debuggare e meno rischioso da modificare.

Separazione tra dati e logica: modularità e aggiornamenti localizzati

La riorganizzazione in array nel file .dat (punti, hover, parametri, ecc.) consente una separazione più netta tra dati di processo e logica di esecuzione: la struttura del .src diventa un template stabile che indirizza dinamicamente gli elementi degli array (ad esempio XP_init[bead], XP[mid_idx], WDAT_end[bead]). Ne consegue che molte modifiche alla traiettoria e ai settaggi possono essere gestite aggiornando il dataset nel .dat, senza intervenire sulla logica principale nel .src, riducendo interventi manuali e possibilità di errore.

Questa modularità abilita anche una parametrizzazione più "reale" del processo: velocità, stati, e parametri tecnologici possono essere associati in modo coerente a bead e segmenti, invece di risultare dispersi in istruzioni sequenziali, rendendo più naturale applicare modifiche globali (intero cordone) o locali (singolo segmento) in modo controllato.

Riduzione dei tempi di setup e del rischio di errore umano

Nel workflow tradizionale, anche una modifica minima richiede rigenerazione dei file KRL e nuovo caricamento sul controllore, con conseguente aumento dei tempi di

setup e rischio di errori (tipicamente su selezione di punti, coerenza dei parametri e versioning). Rendendo la logica più compatta e "template-based", e spostando la variabilità principalmente sui dati indicizzati, si riduce la necessità di operazioni manuali ripetitive e si migliora l'affidabilità del ciclo "modifica–test–deploy".

Standardizzazione e riusabilità del programma

Lo script di refactoring opera come un motore di "templating": preserva intestazione e parametri specifici, ma sostituisce la logica ripetitiva con una logica ciclica generalizzata. Questo favorisce la standardizzazione interna (stesso schema di esecuzione per programmi diversi) e facilita sia manutenzione sia trasferibilità del know-how (un solo modello logico da comprendere e validare).

Tracciabilità e diagnosi più immediate a livello di esecuzione

Un ulteriore vantaggio, spesso sottovalutato, è la tracciabilità: ragionare esplicitamente in termini di CurrentBead e CurrentSegment rende immediato identificare "dove" ci si trova nel processo e quale porzione di traiettoria è stata eseguita. Questo contrasta la frammentazione tipica del flusso tradizionale (CAD/slicing/codice/manufatto distribuiti su più strumenti e file), semplificando diagnosi post-processo e ottimizzazione iterativa.

Abilitazione delle procedure di recovery e continuità del cordone

L'accesso indicizzato ai punti costituisce inoltre il prerequisito tecnico per implementare procedure di interruzione e ripresa automatizzate, requisito tipico dei processi WAAM industriali (manutenzione torcia, cambio filo, ispezioni, anomalie). La logica integra la gestione del segnale IN[STOP_REQUEST] e routine dedicate (RECOVERY_EXIT(), RECOVERY_ENTRY()) che consentono una chiusura ordinata, salvataggio dello stato (bead/segmento) e ripartenza deterministica, mantenendo continuità geometrica del cordone.

4.5 Verifica di equivalenza geometrica dei waypoint

Prima di continuare con l'evoluzione della tesi è necessario testare la corretta creazione della sequenza di waypoint nel nuovo file `.dat` "a vettori".

Per certificare l'equivalenza della traiettoria TCP tra la versione "classica" (DepRoll 0001.src + DepRoll0001.dat) e quella "a vettori" (DepRoll0001_semplificato.src + DepRoll0001_array.dat) è stata confrontata la terna posizione-orientamento-posture/assi esterni di ogni punto omologo tra i due file `.dat`.

4.5.1 Metodologia di verifica

La Tab. 4.5 esplicita, per alcuni bead rappresentativi ($k = 1,3,5$), la corrispondenza biunivoca tra la notazione "classica" (punti $XP1, XP38, \ldots$) e la notazione "a vettori" (punti indicizzati XP_init[k], XP_end[k], …) mediante le regole di indicizzazione riportate nella colonna *formula*. In particolare:

- i punti di *start/hover* e *end*, un punto per tipo per ogni bead, sono ottenuti, attraverso gli indici n, m, a, b, con le varie formule riportate nella tabella, garantendo l'allineamento con i punti della sequenza classica;
- i punti interni di *loop* sono generati dalla relazione $XP[(j-1)+36(k-1)] = XPi$ con $i = j+41(k-1)$ e $j \in [2,37]$, con k che rappresenta il bead corrente, preservando l'ordine di percorrenza all'interno del bead.

A partire da tali regole, è stato effettuato un confronto punto a punto fra i due file `.dat`. Per ogni waypoint omologo si sono verificate le equivalenze su:

1. posizione TCP,
2. orientamento,
3. posture/assi esterni.

Sono stati estratti campioni *start/hover*, *loop* (due j interni) ed *end* per ciascun bead così da coprire l'intero ciclo del bead.

4.5.2 Risultato

L'applicazione delle regole di Tab. 4.5 ha restituito, per tutti i campioni considerati, una perfetta sovrapposizione dei waypoint tra le due rappresentazioni: le differenze risultano nulle. In particolare, i punti *start/hover* e *end* coincidono per costruzione (indici n, m, a, b), mentre i punti interni di *loop* rispettano l'ordinamento, preservando la progressione geometrica del percorso.

4.5.3 Implicazioni

La verifica dimostra che la versione "a vettori" riproduce fedelmente la traiettoria TCP e lo stato degli assi esterni della versione classica, con il vantaggio di:

- ridurre la ridondanza dei `.dat` e semplificare la generazione automatica dei waypoint;
- abilitare controlli e trasformazioni per indici (filtri su k e j) senza operare su liste lunghe di simboli;
- rendere più agevole l'integrazione con pipeline PC/PLC per l'invio "a pacchetti" (blocchi *start–loop–end*).

Alla luce di ciò, nei capitoli successivi si assumerà la rappresentazione "a vettori" come standard di riferimento per la definizione dei percorsi e per le interfacce di scambio dati.

Tabella 4.5: Certificazione uguaglianze

bead_k	array_name	classic_name	formula
1	`XP_hover_init[1]`	`XP1_hover`	XP_hover_init$[k]$ = XPn_hover $n = 1 + 41(k-1)$
1	`XP_init[1]`	`XP1`	XP_init$[k]$ = XPn $n = 1 + 41(k-1)$
1	`XP_end[1]`	`XP38`	XP_end$[k]$ = XPm $m = 38 + 41(k-1)$
1	`XP_end1[1]`	`XP40`	XP_end1$[k]$ = XPa $a = 40 + 41(k-1)$
1	`XP_end2[1]`	`XP41`	XP_end2$[k]$ = XPb $b = 41 + 41(k-1)$
1	`XP[6]`	`XP7`	XP$[(j-1)+36(k-1)]$ = XPi $i = [j + 41(k-1)]$ e $j \in [2;37]$
1	`XP[25]`	`XP26`	XP$[(j-1)+36(k-1)]$ = XPi $i = [j + 41(k-1)]$ e $j \in [2;37]$
3	`XP_hover_init[3]`	`XP83_hover`	XP_hover_init$[k]$ = XPn_hover $n = 1 + 41(k-1)$
3	`XP_init[3]`	`XP83`	XP_init$[k]$ = XPn $n = 1 + 41(k-1)$
3	`XP_end[3]`	`XP120`	XP_end$[k]$ = XPm $m = 38 + 41(k-1)$
3	`XP_end1[3]`	`XP122`	XP_end1$[k]$ = XPa $a = 40 + 41(k-1)$
3	`XP_end2[3]`	`XP123`	XP_end2$[k]$ = XPb $b = 41 + 41(k-1)$
3	`XP[78]`	`XP89`	XP$[(j-1)+36(k-1)]$ = XPi $i = [j + 41(k-1)]$ e $j \in [2;37]$
3	`XP[97]`	`XP108`	XP$[(j-1)+36(k-1)]$ = XPi $i = [j + 41(k-1)]$ e $j \in [2;37]$
5	`XP_hover_init[5]`	`XP165_hover`	XP_hover_init$[k]$ = XPn_hover $n = 1 + 41(k-1)$
5	`XP_init[5]`	`XP165`	XP_init$[k]$ = XPn $n = 1 + 41(k-1)$

continua nella pagina successiva

Tabella 4.5 – continua dalla pagina precedente

bead_k	array_name	classic_name	formula
5	`XP_end[5]`	`XP202`	XP_end[k] = XPm $m = 38 + 41(k-1)$
5	`XP_end1[5]`	`XP204`	XP_end1[k] = XPa $a = 40 + 41(k-1)$
5	`XP_end2[5]`	`XP205`	XP_end2[k] = XPb $b = 41 + 41(k-1)$
5	`XP[150]`	`XP171`	XP[$(j-1)+36(k-1)$] = XPi $i = [j + 41(k-1)]$ e $j \in [2;37]$
5	`XP[169]`	`XP190`	XP[$(j-1)+36(k-1)$] = XPi $i = [j + 41(k-1)]$ e $j \in [2;37]$

4.6 Progettazione delle procedure di recovery

4.6.1 Motivazione e requisiti funzionali

Nei processi WAAM industriali, le interruzioni sono eventi frequenti dovuti a diverse cause: manutenzione programmata della torcia, sostituzione del filo, anomalie di processo (es. porosità rilevata), o interventi di emergenza. Nel workflow originale (Sezione 4.1), la ripresa del processo dopo un'interruzione richiedeva:

- Identificazione manuale del punto di interruzione,
- Modifica del codice KRL per impostare i punti di start,
- Ricompilazione e upload del programma,
- Posizionamento manuale del robot in prossimità del punto di ripresa

Questo processo richiederebbe molto tempo e presenta rischi di errore umano (es. selezione del bead errato, sovrapposizioni o gap nella deposizione). La nuova logica KRL ciclica (Sezione 4.3) abilita l'implementazione di procedure di recovery di uscita e rientro automatiche grazie all'accesso indicizzato ai punti tramite `[bead, segment]`. I requisiti funzionali identificati sono:

- REQ-1: Ripresa automatica da un punto arbitrario (b_{Selected}, s_{Selected}) senza modifica del codice
- REQ-2: Traiettoria di rientro e uscita sicura che eviti collisioni con il pezzo parzialmente depositato
- REQ-3: Gestione corretta dello stato macchina (arco spento durante l'avvicinamento, riaccensione al punto corretto)
- REQ-4: Tracciabilità dell'interruzione (logging del punto esatto)

4.6.2 Architettura generale del sistema di recovery

Il sistema di recovery si compone di due procedure complementari:

- **RECOVERY_ENTRY()**: Gestisce il rientro nel processo dopo un'interruzione, portando il robot dal punto di `Home` alla ripresa del processo con traiettoria sicura.
- **RECOVERY_EXIT()**: Gestisce l'uscita controllata dal processo, salvando lo stato corrente e portando il robot in posizione sicura.

Per quanto riguarda la procedura di **RECOVERY_EXIT()**, questa si compone dei seguenti step necessari per garantire sicurezza e un'ottimizzazione del successivo rientro nel processo e attivati tramite una flag, che richiama la funzione di uscita dal processo:

1. Spegnimento della torcia e stop del movimento;
2. Salvataggio della posizione di ultima saldatura : $XP_{\text{last_weld}}$;
3. Disattivazione delle forze di rolling (o possibile rotazione della tavola di una quantità ancora da definire di gradi per completare la rullatura dell'ultima parte di materiale depositato prima della disattivazione; procedimento da gestire con il team di esperti del processo lato cliente);
4. Sollevamento della torcia in direzione dell'asse di lavorazione del tool, per evitare collisioni in qualunque posizione di stampa;
5. Raggiunta una posizione di sicurezza, si effettua un movimento `PTP` verso la posizione di `Home` desiderata per la manutenzione.

La procedura di **RECOVERY_ENTRY()** necessita, quindi, della conoscenza dell'ultima posizione di saldatura e dei due indici sopra citati e salvati tramite la funzione di uscita. Il flowchart sintetico è il seguente:

1. Sollevamento dalla posizione di `Home` ad un piano di sicurezza;
2. Movimento in PTP fino ad una posizione rialzata rispetto al punto di fine del segmento precedente al segmento di uscita dal processo
   ($XP_{\text{hover_end}}[s_{\text{Selected}} - 1]$)
3. Discesa lineare fino al punto precedentemente citato($XP_{\text{end}}[s_{\text{Selected}} - 1]$), seguendo la direzione di lavorazione del tool, per evitare collisioni;
4. Calcolo del punto intermedio per passare dalla posizione attuale fino alla posizione $XP_{\text{last_weld}}$, con un movimento `CIRC` (che richiede appunto due punti in input). Per fare questo, serve capire dove si posiziona il punto di ultima saldatura rispetto al punto intermedio del segmento $s_{\texttt{Selected}}$. Grazie alla schematizzazione nella figura 4.2, si può analizzare le due configurazioni differenti:

   - Caso $XP_{\text{last_weld}}.E1 < XP_{\text{mid}}[s_{\text{Selected}}].E1$: In questo caso il punto $XP_{\text{last_weld}}$ si trova prima del punto intermedio del segmento da completare. Per effettuare un movimento `CIRC` dal punto XP_{end} al punto di ultima saldatura, è necessario trovare un punto intermedio tra i due da utilizzare nel movimento. Questo punto viene identificato come XP_{circ} e la trattazione per il calcolo di quest'ultimo verrà affrontata in seguito nella sezione 4.6.3.

- Caso $XP_{\text{last_weld}}.E1 > XP_{\text{mid}}[s_{\text{Selected}}].E1$: In quest'altro caso invece, si può utilizzare il punto XP_{mid} come punto intermedio per il movimento CIRC, senza il bisogno di calcolare alcun punto aggiuntivo.

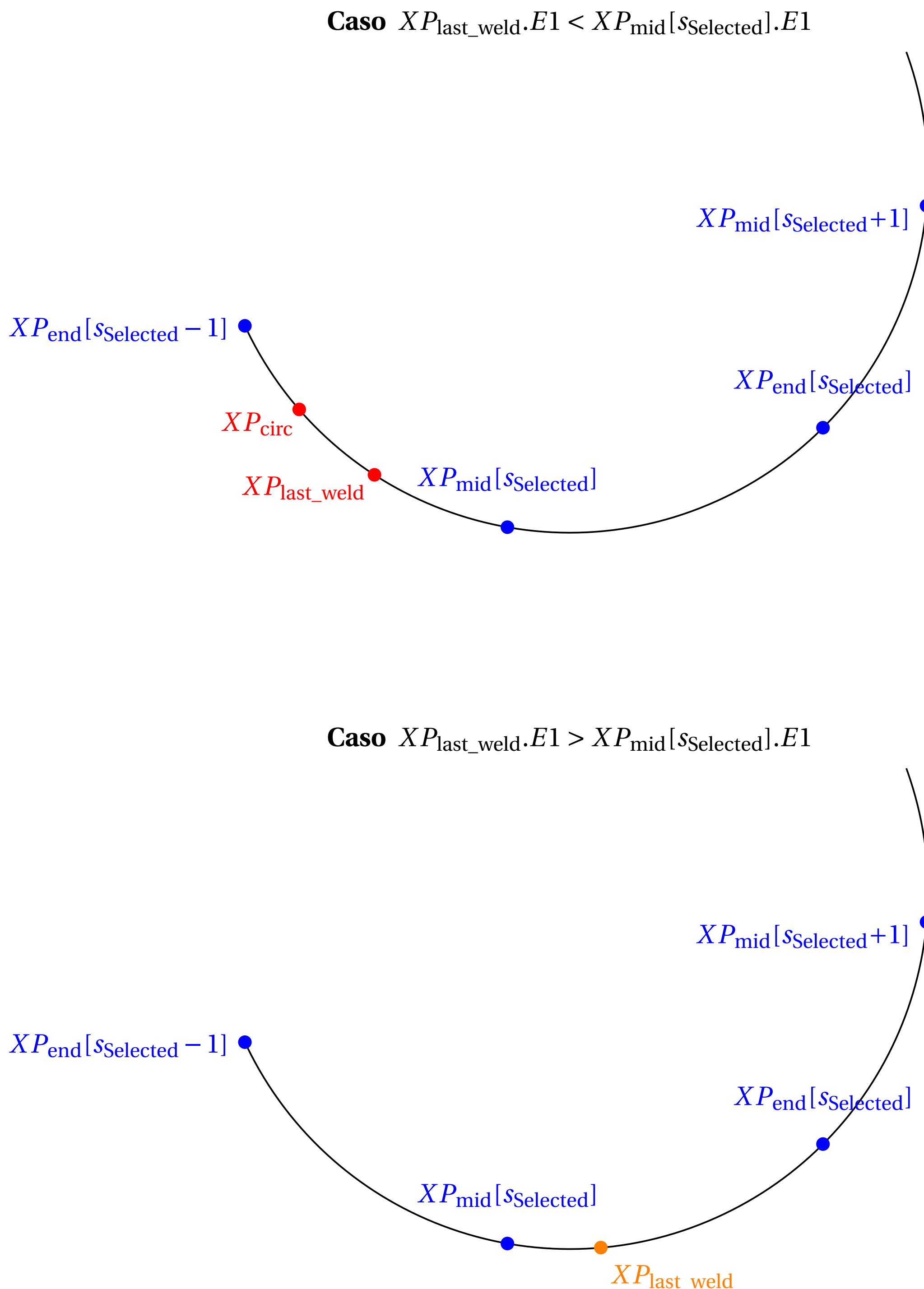

Figura 4.2: Schema delle possibili posizioni di procedura di $XP_{\text{last_weld}}[s_{\text{Selected}}]$.

5. Accendere la torcia in $XP_{\text{last_weld}}$ (discussione futura da affrontare con il cliente: forse è necessario accendere la torcia in una posizione precedente per formare un cordone ottimale);

6. Movimento `CIRC` dalla posizione $XP_{\text{last_weld}}$ alla posizione $XP_{\text{end}}[s_{\text{Selected}}]$. Anche in questo caso, in base alla posizione di $XP_{\text{last_weld}}$ vengono effettuati due step differenti:
   - Caso $XP_{\text{last_weld}}.E1 < XP_{\text{mid}}[s_{\text{Selected}}].E1$: In questo caso il punto $XP_{\text{last_weld}}$ si trova prima del punto intermedio del segmento da completare. Per effettuare un movimento `CIRC` dal punto $XP_{\text{last_weld}}$ al punto finale del segmento, posso utilizzare il punto $XP_{\text{mid}}[s_{\text{Selected}}]$ come punto intermedio del movimento. Non si necessità del calcolo di punti aggiuntivi.
   - Caso $XP_{\text{last_weld}}.E1 > XP_{\text{mid}}[s_{\text{Selected}}].E1$: In quest'altro caso invece, è necessario calcolare un punto intermedio tra l'inizio e la fine del movimento `CIRC` dato che non si dispone di punti prestabiliti. Per il calcolo si può utilizzare la stessa procedura, utilizzata per il calcolo di XP_{circ} e mostrata in seguito.
7. Ritorno al processo standard per la stampa manufatto.

4.6.3 Procedura per calcolo di un punto intermedio

Nei casi in cui sia necessario eseguire un movimento `CIRC` tra due punti giacenti sulla circonferenza (indicati come XP_1 e XP_2) ma non si disponga di un punto intermedio predefinito, è necessario calcolarne uno seguendo la procedura descritta di seguito.

Logica di calcolo

La procedura si articola nei seguenti passaggi:

1. **Calcolo dell'angolo intermedio**: Si determina l'angolo della tavola rotante corrispondente al punto medio dell'arco compreso tra XP_1 e XP_2:
$$E1_{\text{middle}} = \frac{XP_2.E1 - XP_1.E1}{2} \tag{4.1}$$
2. **Conversione in coordinate locali**: Poiché le coordinate globali rimangono costanti (come calcolate nelle sezion 4.1), è possibile ottenere le coordinate locali del punto intermedio conoscendo l'angolo della tavola rotante. Questo si ottiene applicando la formula inversa della moltiplicazione tra matrici di trasformazione utilizzata in precedenza per il passaggio da coordinate locali a globali.
$$\mathbf{P}_{\text{local}} = R_z^{-1}(\phi)\mathbf{P}_{\text{global}} \quad \Longleftrightarrow \quad \begin{bmatrix} XP_{\text{middle}}.x \\ XP_{\text{middle}}.y \\ XP_{\text{middle}}.z \end{bmatrix} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} XP_{\text{global}}.x \\ XP_{\text{global}}.y \\ XP_{\text{global}}.z \end{bmatrix}$$
3. **Assegnazione dei parametri tecnologici**: I parametri tecnologici del punto intermedio XP_{middle} vengono copiati dal punto di partenza XP_1:
$$\begin{aligned} XP_{\text{middle}}.B &= XP_1.B, \\ XP_{\text{middle}}.C &= XP_1.C, \\ XP_{\text{middle}}.S &= XP_1.S, \\ XP_{\text{middle}}.T &= XP_1.T, \\ &\dots \end{aligned} \tag{4.2}$$

4. **Calcolo dell'angolo del robot**: L'angolo A del robot per il punto intermedio viene calcolato secondo la seguente logica:

$$XP_{\text{middle}}.A = -(E1_{\text{middle}} + 90) - 360 \cdot n \tag{4.3}$$

dove n è pari a 1 se:

$$XP_{\text{middle}}.A > -710^\circ \tag{4.4}$$

altrimenti n è pari a 0.

Questa procedura viene applicata nei precedentemente citati casi durante la procedura di RECOVERY_ENTRY(). La procedura garantisce continuità cinematica del movimento circolare e coerenza con i vincoli cinematici del robot, assicurando che la traiettoria di rientro sia fluida e priva di discontinuità.

4.6.4 Sintesi del flusso di esecuzione e della logica di recovery

La figura 4.3 riassume in un unico schema il comportamento complessivo del programma principale di deposizione e della relativa logica di recovery. Il flusso si apre con lo stato `START`, in cui il controllo verifica se il sistema si trovi in una normale condizione di avvio oppure in una condizione di ripartenza, attraverso il flag `recovery_mode`. In assenza di interruzioni pregresse (`recovery_mode = FALSE`) l'esecuzione procede direttamente verso il ciclo di deposizione; in caso contrario viene richiamata la routine `RECOVERY_ENTRY()`.

La funzione `RECOVERY_ENTRY()` ha il compito di riallineare in modo sicuro lo stato del robot con lo stato del processo. In primo luogo viene calcolato il nuovo punto di rientro (*calcolo target*) sulla base delle informazioni salvate al momento dell'interruzione (bead corrente, segmento corrente, eventuali flag di avanzamento). Successivamente il TCP viene portato in una posizione di avvicinamento sicura tramite movimenti `PTP` e solo in un secondo momento viene eseguito un movimento `LIN` di rientro sul cordone, così da minimizzare i rischi di collisione. Una volta sul cordone, inzia il movimento `CIRC` per garantire la continuità geometrica della traiettoria depositata, come presentato nella sezione 4.6.2 .

Una volta completata l'eventuale fase di rientro, il controllo entra nel ciclo principale di deposizione, modellato come un doppio ciclo annidato. Il ciclo esterno `FOR bead = actual TO max` scorre tutti i cordoni dello strato, a partire dal bead corrente fino all'ultimo bead da realizzare. Per ciascun cordone è prevista una fase di *inizio bead*, che comprende il posizionamento in hover, la discesa sul punto di innesco e la preparazione dei flag di processo (accensione arco, sincronizzazione con PLC, ecc.).

All'interno di ogni bead, il ciclo interno `FOR seg = actual TO 18` governa la deposizione dei singoli segmenti che compongono il cordone. Ogni iterazione esegue un movimento `CIRC` tra la coppia di punti `XP[mid]`, `XP[end]`, realizzando così un tratto di traiettoria di deposizione. In questo modo la geometria complessiva del cordone viene scomposta in archi elementari, che il robot esegue in sequenza mantenendo continuità di velocità e orientamento.

Durante il normale funzionamento, il programma effettua un controllo sul segnale

digitale $IN[STOP_REQUEST] proveniente dal PLC o dal livello supervisore. In condizioni normali (STOP_REQUEST = FALSE) il flusso prosegue verso il segmento successivo; in caso di richiesta di stop (STOP_REQUEST = TRUE) viene richiamata la routine RECOVERY_EXIT(). Questa funzione implementa una chiusura ordinata del processo: l'arco viene spento, lo stato del processo viene salvato (indice del bead, indice del segmento, eventuali flag ausiliari) e il robot viene riportato in una posizione di sicurezza tramite un PTP verso Home. Solo dopo queste operazioni il programma esegue lo stato HALT, interrompendo l'esecuzione in modo controllato e rendendo possibile una successiva ripartenza tramite la già descritta RECOVERY_ENTRY().

Se non viene richiesta alcuna interruzione, al termine di ogni cordone viene eseguita la fase di *fine bead*, che comprende il *retract* del TCP e il reset dei flag interni utilizzati durante la deposizione. Il controllo verifica quindi se tutti i bead previsti siano stati completati (*All beads?*); in caso negativo il ciclo riparte dal bead successivo, mentre in caso positivo viene raggiunto lo stato END, che sancisce la corretta conclusione del job di stampa per questo layer.

In sintesi, lo schema di Figura 4.3 mette in evidenza come il programma principale e la logica di recovery non siano due moduli indipendenti, ma parti di un unico flusso di controllo. La presenza esplicita delle funzioni RECOVERY_ENTRY() e RECOVERY_EXIT(), insieme alla gestione strutturata dei cicli su bead e segmenti, consente di garantire da un lato la continuità geometrica del cordone WAAM e, dall'altro, la possibilità di interrompere e riprendere il processo in maniera deterministica e ripetibile.

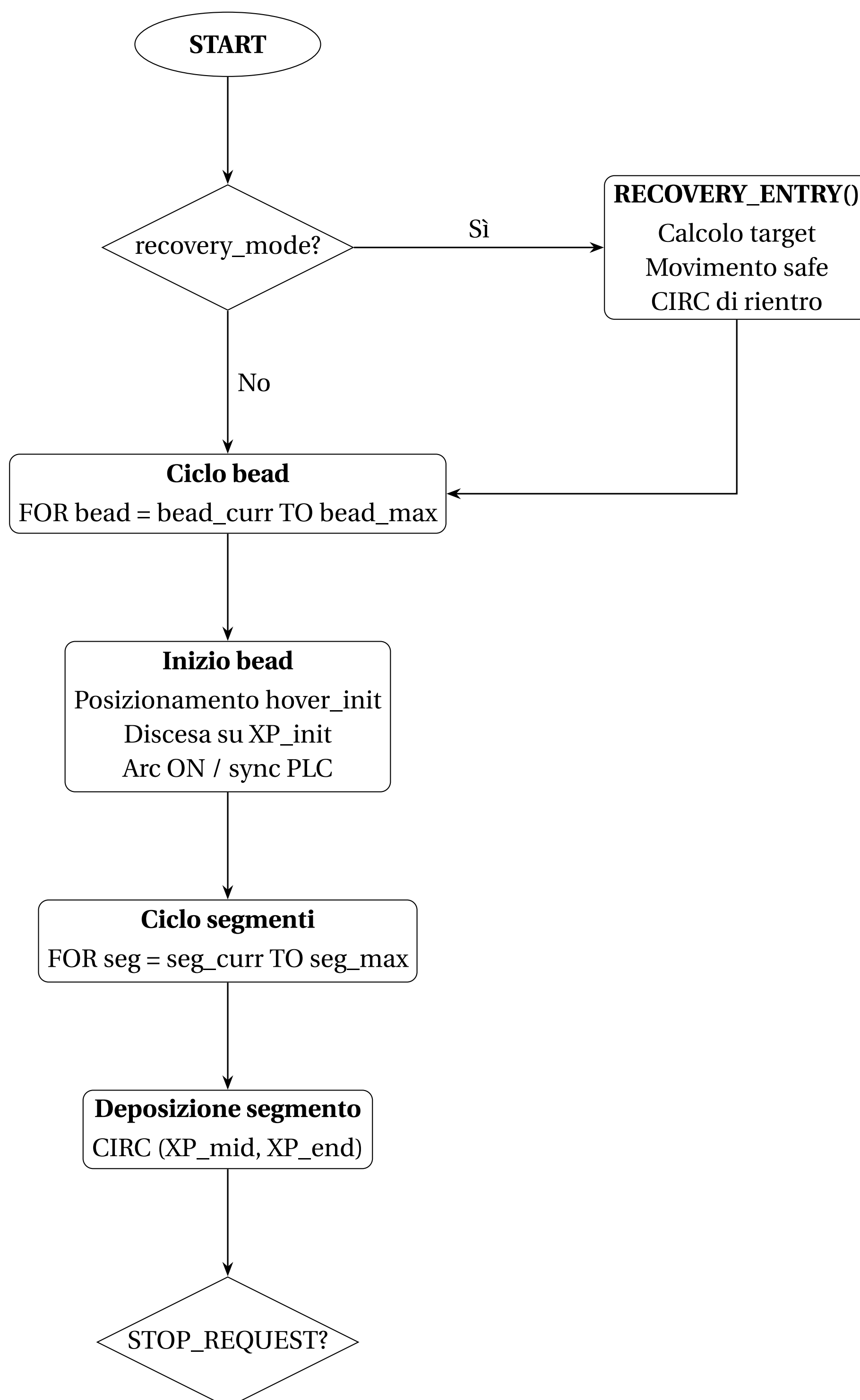

Figura 4.3: Flowchart del programma principale di deposizione e della logica di *recovery* (segue).

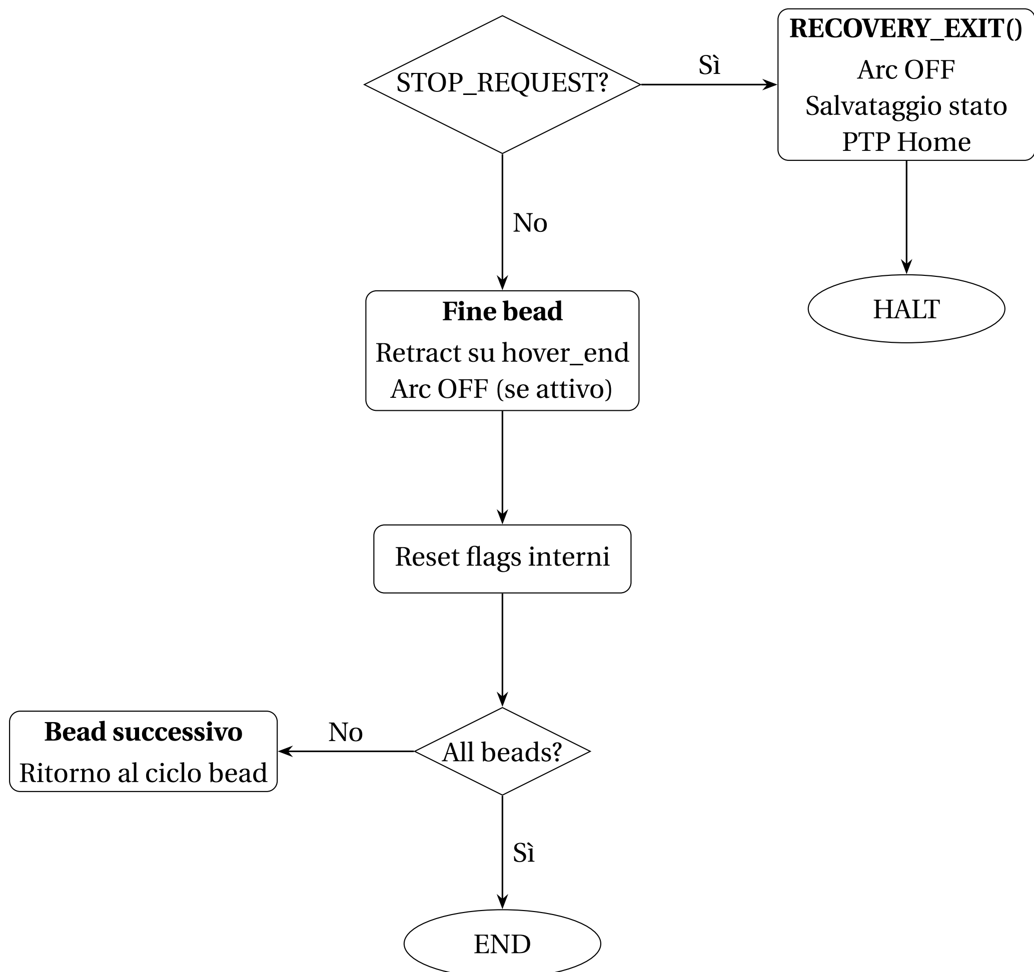

Figura 4.3: Flowchart del programma principale di deposizione e della logica di *recovery*.

Capitolo 5

Gestione del processo di stampa a spirale e parsing complessivo

La complessità geometrica del componente reale richiede di superare la sola logica di deposizione per strati planari sviluppata nel Capitolo 4. In particolare, alcune porzioni del manufatto necessitano di una deposizione a spirale continua, nella quale decadono le ipotesi di regolarità che nel capitolo precedente hanno permesso di descrivere il processo tramite layer e bead con un numero di punti e una struttura di comandi ripetitiva. In queste sezioni la traiettoria deriva dal moto combinato robot–tavola e si organizza in macro-tratti non riconducibili a cordoni paralleli; applicare direttamente il parsing "planare" porterebbe a indici inconsistenti e, soprattutto, alla perdita della correlazione tra traiettoria, stato dell'arco e rotazione dell'asse esterno.
Il capitolo affronta quindi la gestione delle sezioni elicoidali analizzando il codice KRL dedicato e definendo un pattern di processo generalizzabile. La Sezione 5.1 chiarisce il ruolo delle spirali e i limiti del modello per layer planari; la Sezione 5.2 studia in dettaglio la struttura del programma di riferimento e introduce una logica di parsing basata su macro-sequenze delimitate da posizionamenti R_PTP e classificate tramite il contesto tecnologico dell'arco; infine, la Sezione 5.3 integra i due parser (planare e spirale) in un'unica pipeline che produce un modello dati unificato in formato JSON, in cui l'intero componente viene rappresentato come "timeline di stampa" ordinata e pronta per la conversione in pacchetti Start/Loop/End. Questo output costituisce l'input diretto per l'architettura di controllo e comunicazione descritta nel capitolo successivo.

5.1 Ruolo delle sezioni a spirale e limiti del pattern per layer planari

Nel capitolo precedente è stato descritto il flusso di lavoro adottato per la gestione dei layer normali del componente da stampare: sezioni planari ottenute tramite slicing tradizionale, caratterizzate da cordoni paralleli e da una struttura molto regolare del codice KRL. Ogni layer viene scomposto in bead, ciascuno descritto da un numero fisso di punti di passaggio, e la logica di parsing fa leva proprio su questa regolarità: una sequenza ordinata di pose XP_n, raggruppate in blocchi ripetuti (hover iniziale, innesco, punti di saldatura, spegnimento, hover finale).

Questa impostazione funziona molto bene per le porzioni del pezzo in cui la depo-

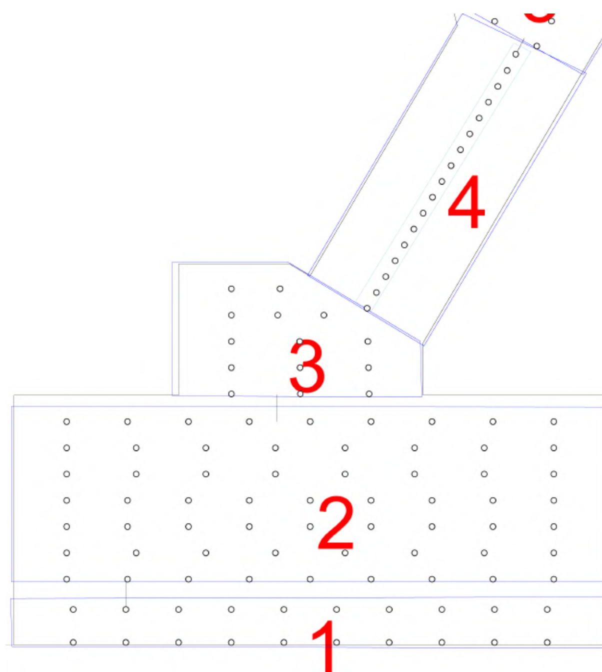

Figura 5.1: Sezioni di diversa tipologia

sizione può essere descritta come successione di strati "piatti" e tra loro indipendenti: l'orientamento della torcia rimane sostanzialmente costante, la tavola rotante non introduce una cinematica complessa. In figura 5.1 sono richiamate, ad esempio, le sezioni 1 e 2, dove i punti di deposizione risultano distribuiti su una griglia quasi regolare e organizzati in bead paralleli.

Nel componente reale, tuttavia, sono presenti anche porzioni geometricamente più complesse, in cui la saldatura non procede per "strati piani" sovrapposti, ma assume l'andamento di una traiettoria elicoidale che avvolge la superficie del pezzo. È il caso delle sezioni a spirale, generate per esempio nella zona inclinata nella figura 5.1. In queste regioni la testa di deposizione si muove lungo una direzione prevalente, mentre, in parallelo, viene comandata la rotazione dell'asse esterno, producendo un percorso complessivo a elica che non può essere ricondotto a un semplice riempimento planare.

Dopo i layer "massivi" 1 e 2, utilizzati per costruire il basamento, e il raccordo 3, si osserva la zona 4, in cui la densità dei punti e l'orientamento delle traiettorie iniziano a differenziarsi rispetto allo schema precedente. Questo cambiamento diventa ancora più evidente nello sviluppo mostrato in figura 5.2, dove di osserva il seguente sviluppo del manufatto con le sezioni 7-12: lungo le sezioni più sottili, i punti non sono più organizzati in cordoni paralleli, ma disposti secondo file oblique e leggermente incurvate, che seguono l'andamento dello sviluppo superficiale. Il risultato è un pattern di deposizione che si avvicina a una spirale continua, piuttosto che a una sovrapposizione di layer indipendenti.

Dal punto di vista del controllo e dell'analisi dei dati, questa differenza geometrica ha conseguenze significative. Nei layer normali:

- esiste un numero fisso di punti per ciascun bead,

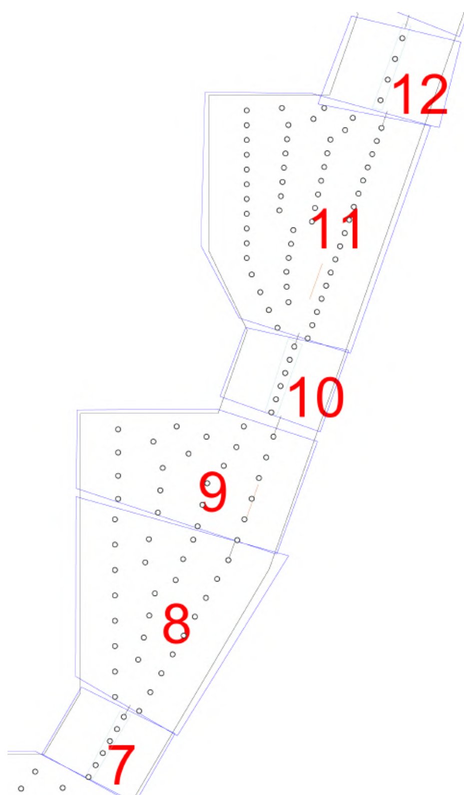

Figura 5.2: Sezioni nella zona inclinata

- i bead sono percorsi in modo sequenziale e ripetitivo,
- la struttura dei file `.dat` e `.src` ricalca questo schema (nomi delle variabili, ordine dei comandi, sequenze $XP_{\text{hover_init}} - XP - XP_{\text{end}}$, ecc.),
- è relativamente semplice associare a ogni punto un indice logico "(bead, posizione nel bead)" e costruire array ordinati per il successivo invio al PLC.

Nelle sezioni a spirale, al contrario, nessuna di queste ipotesi è garantita:

- il numero di punti tra un innesco e lo spegnimento dell'arco può variare in modo sensibile,
- la traiettoria non è suddivisa in bead paralleli ma in macro-tratti che seguono il moto combinato robot–tavola.

Se si cercasse di applicare direttamente a queste sezioni la logica di parsing sviluppata per i layer planari, si incorrerebbe in una serie di problemi: il numero dei punti per "bead" non sarebbe costante, gli indici non corrisponderebbero più a regioni omogenee della superficie, e soprattutto verrebbe persa la correlazione tra traiettoria, stato dell'arco e posizione della tavola rotante. In altre parole, il pattern "lineare" costruito nel capitolo 4 non è in grado di catturare la struttura fisica del processo a spirale: da un lato semplificherebbe eccessivamente la geometria (trattando una spirale come una successione di brevi segmenti rettilinei indipendenti), dall'altro non fornirebbe al PLC e al controllore KUKA le informazioni necessarie per ricostruire correttamente la logica della deposizione.

Per gestire il processo in modo coerente lungo l'intero componente, diventa quindi

necessario affiancare al pattern dei layer normali un pattern dedicato per le sezioni a spirale, capace di descrivere:

- come è organizzata la traiettoria in macro-sequenze legate alle posizioni del martello,
- quando l'arco viene acceso, mantenuto e spento all'interno di ciascuna sequenza,
- quali porzioni di movimento sono "a secco" (solo posizionamento, senza deposito) e quali, invece, contribuiscono effettivamente alla crescita del manufatto.

La sezione successiva analizzerà in dettaglio la struttura del codice KRL relativo a una sezione a spirale (estratto di DepRoll0001_spiral_sez_4 in B.10) per derivare da esso una logica di processo generalizzabile. Su questa base verrà poi costruito un nuovo algoritmo di parsing, specifico per le spirali, che potrà convivere con quello sviluppato per i layer normali all'interno di un'unica pipeline di preparazione dati per PLC e robot KUKA.

5.2 Analisi del codice KRL della sezione 4 e definizione del nuovo pattern

Per passare dalla descrizione puramente geometrica delle spirali alla loro gestione automatizzata è necessario esaminare in dettaglio il codice KRL generato per una sezione reale. A questo scopo è stato preso come riferimento il programma `DepRoll0001_spiral_sez_4.src`, rappresentativo della prima zona in cui la deposizione lascia la logica "a layer planari" e assume l'andamento elicoidale discusso nel paragrafo precedente.

Il programma non è suddiviso in loop su bead, ma in macro-blocchi delimitati da comandi di posizionamento del martello pressatore, che nel caso della sezione 4 sono:

$$\begin{gathered} R_PTP(RP0) \\ \dots \\ R_PTP(RP7) \\ \dots \\ R_PTP(RP14) \\ \dots \end{gathered}$$

Il primo passo per interpretare queste sezioni consiste proprio nell'individuare tutte le istruzioni $R_PTP(RPxx)$ e nel considerare ogni intervallo di righe compreso tra $R_PTP(RPk)$ e $R_PTP(RPk+1)$ come una macro–sequenza di lavorazione.

All'interno di ciascuna di queste macro–sequenze il pattern dei comandi è profondamente diverso rispetto ai layer normali. Invece di una serie di CIRC omogenei che percorrono un bead piano, si osservano:

- uno o più comandi CIRC iniziali, eseguiti con arco spento, utilizzati per posizionare il TCP lungo una traiettoria circolare prima dell'avvio della deposizione;

- una LIN immediatamente successiva a una chiamata `ArcMainNG(#ArcOnBeforeMoveStd, ...)`, che rappresenta l'innesco effettivo dell'arco sulla spirale;

- una sequenza di coppie CIRC (in numero variabile) eseguite con arco acceso, che costituiscono la traiettoria di saldatura vera e propria;

- una LIN associata a `ArcMainNG(#ArcOffBeforeMoveStd, ...)`, impiegata per spegnere l'arco e completare la macro–sequenza.

Questa struttura, ripetuta per tutte le coppie di posizionamenti R_PTP, non può essere ricondotta al modello bead–segmento sviluppato per i layer planari: il numero dei CIRC intermedi varia, non esiste un "bead_max" univoco, e soprattutto la presenza del posizionamento del martello come inizio/fine di sequenza rende più naturale ragionare in termini di sequenze tra due R_PTP piuttosto che in termini di layer orizzontali.

Questa logica viene implementata nel tool di parsing tramite una piccola macchina a stati che tiene traccia del contesto tecnologico determinato dalle chiamate alla macro ArcMainNG. Nel codice C# tale contesto è rappresentato dall'enumerazione:

Listing 5.1: Definizione dell'enum `ArcContext`

```
enum ArcContext
{
        None,
        TorchOn,     // ArcMainNG(#ArcOnBeforeMoveStd, ...)
        TorchOff,    // ArcMainNG(#ArcOffBeforeMoveStd, ...)
        ArcSwitch    // ArcMainNG(#ArcSwiBeforeMoveStd, ...)
};
```

Durante la scansione riga per riga del file, ogni volta che il parser incontra una chiamata `ArcMainNG(#ArcOnBeforeMoveStd, ...)` imposta lo stato corrente a `TorchOn`; in presenza di `ArcMainNG(#ArcOffBeforeMoveStd, ...)` passa a `TorchOff`; mentre `ArcMainNG(#ArcSwiBeforeMoveStd, ...)` porta lo stato a `ArcSwitch`. Lo stato viene poi "consumato" dal primo comando di movimento successivo (LIN o CIRC), che viene classificato in base al contesto.

La classificazione procede secondo regole semplici ma efficaci:

- un comando LIN che trova lo stato `TorchOn` viene interpretato come accensione torcia e memorizzato nel campo LinOn della sequenza;

- un LIN con stato `TorchOff` viene registrato come spegnimento torcia (LinOff);

- un CIRC con stato `ArcSwitch` viene considerato parte della traiettoria di saldatura e aggiunto alla lista WeldCircs;

- in assenza di contesto (stato None), il primo CIRC della sequenza viene interpretato come CIRC "a secco" (DryCirc), mentre gli eventuali CIRC successivi sono comunque associati alla saldatura, sulla base dell'osservazione empirica che nei programmi DepRoll i tratti a secco precedono sempre l'accensione, mentre i CIRC successivi all'ArcOn appartengono alla deposizione.

Queste regole, verificate manualmente sulla sezione 4, permettono di ricostruire in automatico lo stesso schema che era stato inizialmente disegnato a mano: per ogni intervallo$[RP_k; RP_{k+1}]$ il parser restituisce una descrizione compatta del tipo:

Listing 5.2: Riprendendo la sequenza riportata in B.10

```
R_PTP[7; 14] ->
CIRC (XP254, XP255) a "secco"
LIN XP256 accensione TORCIA
CIRC[(XP257, XP258); (XP507, XP508)]
LIN XP509 spegnimento TORCIA
```

Dal punto di vista implementativo, per ogni intervallo $[RP_k; RP_{k+1}]$ viene popolata una struttura dati `Sequence` che contiene:

- gli estremi `RStart` e `REnd` (le posizioni RP corrispondenti a inizio e fine della macro–sequenza);
- il CIRC iniziale DryCirc (se presente);
- il punto LinOn di accensione;
- a lista WeldCircs dei CIRC di saldatura;
- il punto LinOff di spegnimento.

Il parser svolge quindi il ruolo di traduttore tra il livello di programmazione robotica, caratterizzato da centinaia di istruzioni KRL, e un livello più alto in cui ogni sequenza è un oggetto compatto, pronto per essere serializzato e integrato nel modello dati globale del componente.

Nel capitolo successivo questa rappresentazione sarà fusa con quella dei layer normali all'interno della struttura JSON del progetto, consentendo di gestire in modo uniforme, all'interno della stessa pipeline PC–PLC–KUKA, sia le sezioni planari sia le sezioni a spirale.

5.3 Parsing complessivo del progetto e integrazione dei diversi tipi di layer

L'analisi delle sezioni a spirale e la definizione di un pattern dedicato avrebbero poco senso se rimanessero confinati a un esercizio locale sul singolo file KRL. L'obiettivo finale è, invece, disporre di una pipeline di parsing unica, capace di leggere l'intero set di programmi DepRoll (layer planari e spirali), ricondurli a una rappresentazione omogenea e metterli a disposizione del livello di controllo PC–PLC–KUKA in forma strutturata.

In questa prospettiva il lavoro svolto sui layer normali (capitolo 4) e su quelli a spirale (paragrafi 5.1-5.2) viene "fuso" in un'unica architettura di parsing, che opera sostanzialmente su tre piani:

1. riconoscimento e classificazione dei file di ingresso,

2. applicazione del parser appropriato a seconda del tipo di layer,
3. costruzione di un modello dati globale, ordinato secondo la sequenza reale di stampa.

Sul primo piano, il sistema parte da un insieme eterogeneo di file KRL generati dal flusso di programmazione (file `.dat` e `.src` organizzati per sezioni, parti). Attraverso regole basate sulla nomenclatura e sulla struttura interna dei programmi, ogni unità viene classificata come:

- **layer normale**, descritto dal file `DepRollXXXX_array.dat` in cui il contenuto principale è costituito da array di punti (XP_hover_init, XP, XP_end, ecc.) e da blocchi di saldatura regolari,
- **layer a spirale**, descritto da una coppia di file `_spiral_.src/.dat` in cui compaiono esplicitamente macro–movimenti R_PTP(RP...), seguiti da sequenze di LIN/CIRC con chiamate ArcMainNG che implementano la traiettoria elicoidale.

Questa fase di "smistamento" è cruciale: consente di decidere, per ciascuna sezione, quale logica di parsing attivare, evitando di forzare le spirali dentro uno schema pensato per i layer planari o viceversa.

Sul secondo piano agiscono i due parser specializzati, sviluppati in parallelo ma con un obiettivo comune.

Per i layer normali, il parser DAT utilizza la struttura regolare degli array per ricostruire, per ciascun bead, i pacchetti logici già introdotti nel capitolo precedente:

- un blocco di inizializzazione (hover iniziale, posizionamento di inizio cordone),
- un loop di saldatura definito da una sequenza di punti XP percorsi con arco attivo,
- un blocco di chiusura (spegnimento, hover finale).

Da questa informazione vengono generati oggetti ad alto livello che rappresentano, in maniera totalmente indipendente dal dettaglio KRL, il "pezzo di processo" da inviare allo strato di controllo: l'unità base è il bead di un layer planare.

Per i layer a spirale, invece, il parser applica la logica definita nella sezione 5.2. Il programma viene segmentato in sequenze comprese tra due posizionamenti R_PTP consecutivi, e all'interno di ciascuna sequenza la macchina a stati sull'arco (None, TorchOn, TorchOff, ArcSwitch) permette di individuare tutte le varie fasi delle `Sequence` precedentemente descritte.

Qui l'unità base non è più il bead, ma la sequenza RP–RP vista come blocco completo "posizionamento + deposizione". Anche in questo caso, il risultato del parsing non è un semplice elenco di comandi KRL, ma un oggetto ad alto livello che descrive il contenuto tecnologico della sequenza, pronto per essere serializzato in forma neutra.

Un aspetto chiave dell'architettura è che, nonostante la profonda differenza tra layer

planari e spirali, l'uscita dei due parser converge verso lo stesso tipo di struttura logica: in entrambi i casi, l'elemento di base è rappresentato da un trittico concettuale inizio–loop–fine (accensione – traiettoria di saldatura – spegnimento), associato a un certo insieme di punti geometrici e a un contesto di processo (velocità, parametri di arco, posizione della tavola). Questa scelta progettuale è ciò che rende possibile l'ultimo passo.

Sul terzo piano, infatti, entra in gioco il modello dati globale del progetto. Tutti gli oggetti prodotti dai due parser, bead di layer normali, sequenze di layer a spirale, vengono raccolti all'interno di una struttura unificata che rappresenta l'intero componente come lista ordinata di layer logici.

Ogni elemento di questa lista porta con sé:

- i riferimenti alla porzione di programma da cui deriva (nome file, sezione, parte),
- il tipo di layer (ad esempio NORMAL o SPIRAL),
- la posizione nella sequenza di fabbricazione (indice globale di layer),
- il contenuto tecnologico organizzato in sotto–blocchi Start / Loop / End, indipendentemente dal fatto che derivi da un bead planare o da una sequenza RP–RP.

Il risultato è una sorta di "timeline di stampa" che attraversa tutto il componente: il sistema non vede più un insieme sparso di file .dat e .src, ma una sequenza coerente di operazioni di deposizione, ciascuna descritta a un livello di astrazione che è compatibile con le necessità del PLC e del controllore KUKA. Dal punto di vista operativo, questo parsing complessivo produce un unico file di output (in formato JSON, con struttura rappresentata nella figura 5.3) in cui tutti i layer, normali e a spirale, sono allineati e pronti per essere convertiti in pacchetti di comunicazione. Lo stesso meccanismo di invio, la stessa logica di avanzamento e di recupero (recovery) possono così essere applicati in modo uniforme a tutte le porzioni del pezzo, senza che il PLC debba sapere se un certo layer proviene da un file DAT planare o da un file DAT a spirale: questa informazione resta confinata nel livello di parsing.

In sintesi, la creazione del parsing totale ha permesso di trasformare una collezione eterogenea di programmi KRL, costruiti con logiche diverse a seconda della geometria, in una rappresentazione unificata del processo WAAM, sulla quale è possibile innestare la logica di streaming, di supervisione e di recupero descritta nei capitoli successivi e precedenti.

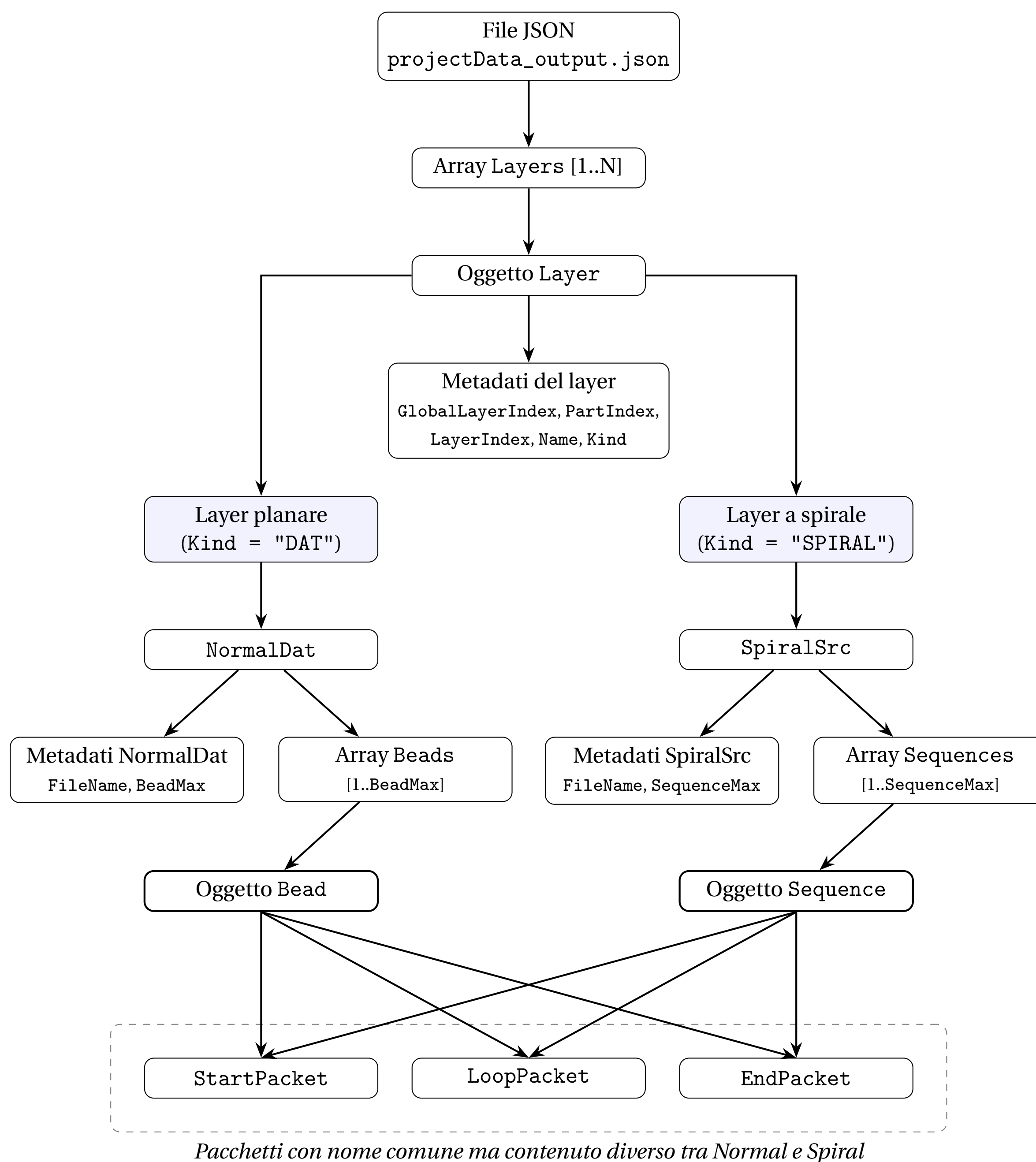

Figura 5.3: Struttura del JSON

Capitolo 6

Progettazione dell'architettura di streaming PC-PLC-KUKA: dal concetto alla simulazione

Dopo la riorganizzazione del codice KRL in strutture indicizzate e cicliche (Capitolo 4) e la definizione di un modello dati unificato in formato JSON per descrivere l'intero processo di deposizione, includendo sia layer planari che sezioni a spirale (Capitolo 5), emerge il limite ancora dominante dell'approccio "file-based": i dati di processo rimangono residenti nel controllore KUKA e ogni variazione richiede la rigenerazione e il caricamento di nuovi programmi. Tale vincolo penalizza scalabilità, interoperabilità e possibilità di introdurre correzioni o logiche avanzate senza interrompere il workflow.
Per superare questi limiti, il capitolo presenta una nuova architettura distribuita di streaming PC–PLC–KUKA, in cui il file JSON diventa sorgente centralizzata dei dati e l'esecuzione è alimentata in tempo reale tramite pacchetti di punti e parametri. Il sistema coinvolge tre attori con responsabilità distinte: PC (pianificazione e preparazione dei pacchetti), PLC (buffering e gestione I/O) e robot KUKA (ricezione ed esecuzione). La continuità del processo è garantita da una logica di triplo buffer sul robot, progettata per prevenire la data starvation e assicurare un flusso di esecuzione senza arresti.
Il capitolo è organizzato come segue: la Sezione 6.1 motiva il passaggio dal formato .dat a una struttura gerarchica JSON; la Sezione 6.2 descrive l'architettura di streaming e i task concorrenti (PC sender, PLC streamer, task nel SPS.SUB e main robot executor); la Sezione 6.3 sintetizza i vantaggi della soluzione proposta; infine, la Sezione 6.4 definisce la transizione verso l'implementazione hardware reale e il confronto tra layout attuale e layout proposto.Questo capitolo pone le basi concettuali per le simulazioni software presentate nei capitoli successivi, necessarie a validare l'architettura prima dell'implementazione su hardware reale.

6.1 Dall'array KRL al file JSON: i vantaggi di una struttura dati gerarchica

Sebbene l'uso degli array KRL (Cap. 4) abbia migliorato l'accesso ai dati, la struttura resta vincolata alla sintassi KRL ed è fondamentalmente piatta. I progetti multi-layer erano ancora frammentati in file `.dat` separati, rendendo l'accesso programmatico, ad

esempio, al "punto 5 del bead 3", inefficiente.

La soluzione è stata adottare JSON come formato per descrivere l'intero processo di deposizione. Questo passaggio introduce vantaggi fondamentali in termini di organizzazione, leggibilità e interoperabilità.

6.1.1 Organizzazione strutturale e leggibilità

A differenza della struttura piatta dei file `.dat`, dove le variabili sono dichiarate in una lunga lista (es. `XP_init[1]`, `XP_init[2]`, ecc.), il JSON impone una struttura gerarchica e auto-descrittiva.

- **Gerarchia chiara**: Un file JSON contiene un array di Layers, che a sua volta contiene un array di Beads. Ogni oggetto Bead contiene poi i pacchetti di dati specifici: StartPacket, LoopPacket e EndPacket (struttura del file JSON analizzata nella figura 5.3).

- **Raggruppamento logico**: Tutti i dati relativi a un singolo cordone di saldatura (es. Bead 1) sono contenuti in un unico oggetto. Questo raggruppa logicamente i punti di hover, i punti di percorso, i parametri di saldatura (WDAT) e le posizioni della tavola rotante (RP), evitando la dispersione su più file.

- **Auto-descrizione**: Chiavi JSON come "BeadIndex" o "StartPacket" rendono i dati immediatamente comprensibili, a differenza dei nomi delle variabili KRL che richiedono interpretazione.

6.1.2 Interoperabilità e flessibilità

Il vantaggio più significativo è lo svincolamento dalla piattaforma KUKA.

- **Standard universale**: JSON è uno standard ampiamente supportato, leggibile da qualsiasi linguaggio di programmazione moderno (C#, Python, JavaScript, ecc.). Al contrario, l'analisi della sintassi KRL dei file `.dat` richiede codice personalizzato o strumenti specifici.

- **Integrazione facilitata**: I dati in formato JSON possono essere facilmente utilizzati per alimentare database, API, interfacce HMI web o script di analisi Python, cosa impossibile con i file `.dat` nativi.

- **Gestione centralizzata**: L'intero progetto multi-layer risiede in un unico file JSON, semplificando la gestione, la modifica e il backup rispetto alla gestione di molteplici coppie di file `.src/.dat`.

- **Flessibilità**: Se il processo di saldatura cambia (es. 40 punti per cordone invece di 36), è sufficiente aggiornare il parser che genera il JSON. Qualsiasi altro software (come la simulazione C#) che legge il JSON continuerà a funzionare senza modifiche, poiché leggerà la nuova struttura dati.

6.2 La nuova architettura di streaming (PC-PLC-KUKA)

Avere un file JSON centralizzato risolve il problema dell'organizzazione dei dati, ma introduce una nuova sfida: come trasferire questa mole di dati (che può descrivere ore di processo) a un robot KUKA, che non è progettato per interpretare file JSON complessi in tempo reale.

La soluzione è stata implementare un'architettura di streaming asincrona che scompone il processo in tre componenti hardware/software distinti, ognuno con una responsabilità specifica. L'obiettivo è garantire che il robot non rimanga mai senza dati, utilizzando una logica a triplo buffer implementata sul KUKA.

Per validare questa architettura, prima di implementarla realmente, la costruzione di una simulazione (riportata in figura A.4) è sembrata la soluzione più efficiente.

I componenti dell'architettura sono:

- **PC (Sender)**: Un'applicazione (es. C#) che agisce come mittente.
- **PLC (Streamer)**: Un Soft-PLC (es. TwinCAT) che funge da intermediario e gestore dello streaming.
- **Robot KUKA (Receiver/Executor)**: Il robot che riceve i dati ed esegue il movimento.

Il flusso dati è suddiviso in "Task" concorrenti che simulano l'interazione tra i componenti.

6.2.1 Task 0: Il PC (Sender)

Il PC esegue la logica di più alto livello (esecuzione della simulazione in figura A.5):

- Carica l'intero file JSON complesso.
- Disaggrega il processo in una coda di "pacchetti" di processo (es. `L1 B1 Start`, `L1 B1 Loop`, `L1 B1 End`).
- Calcola i metadati necessari per ogni pacchetto, come PointCount e VarCount.
- Invia un intero pacchetto alla volta al PLC, non appena questo segnala di essere pronto.

6.2.2 Task 1: Il PLC (Streamer)

Il PLC agisce come un "buffer intelligente" e un "convertitore di formato" (esecuzione della simulazione in figura A.6):

- Attende la ricezione di un pacchetto completo dal PC.
- Una volta ricevuto, "smonta" il pacchetto.
- Invia i dati al KUKA un singolo dato alla volta (un punto E6Pos o una singola variabile double, come i valori di RotaryPosition o i parametri di saldatura).

6.2.3 Task 2 & 3: KUKA (SPS.SUB - Assembler & Buffer Logic)

Questi task non sono programmi robot principali, ma risiedono nel SPS.SUB (Submit Interpreter), un programma che gira in background sul KUKA (es. ogni 12ms) ed è progettato per la gestione veloce degli I/O. (esecuzione della simulazione in figura A.7)

- **Task 2 (Assembler)**: Il SPS.SUB attende il flag "Dato Valido" dal PLC (es. $IN[1001] == TRUE). Legge i singoli dati (punti o variabili) dagli input EtherCAT e li "riassembla" nel primo buffer del robot (es. `BUFF_1_PUNTI[]` e `BUFF_1_VARS[]`). Dopo ogni lettura, imposta un flag di "ricevuto" (es. `$OUT[1001]` = `TRUE`) per segnalare al PLC che è pronto per il dato successivo, implementando l'handshake hardware.
- **Task 3 (Buffer Logic)**: Sempre all'interno del SPS.SUB , questo task gestisce la logica del doppio buffer. Attende due condizioni: `BUFF_1` è pieno e `BUFF_2` è libero. Quando entrambe sono vere, esegue un "travaso" : copia i dati da `BUFF_1` a `BUFF_2`, svuota immediatamente `BUFF_1`, e segnala al Task 4 che un nuovo pacchetto è pronto (es. `g_bBuff2Pronto` = `TRUE`).

6.2.4 Task 4: KUKA (MAIN.SRC - Robot Executor)

Questo è il "cervello" del robot, il file `.src` principale che esegue il movimento (esecuzione della simulazione in figura A.8):

- Attende che il pacchetto sia pronto, utilizzando un semplice comando KRL : `WAIT FOR` (`g_bBuff2Pronto` == `TRUE`).
- Esegue un secondo travaso di sicurezza da `BUFF_2` al "buffer di esecuzione" `BUFF_3`.
- Svuota immediatamente `BUFF_2`, segnalando al Task 3 che può travasare il prossimo pacchetto.
- Esegue il movimento (es. `Start_Motion()`, `Loop_Motion()`) basandosi sui dati contenuti in `BUFF_3`.
- Finito il movimento, torna in attesa del pacchetto successivo.

6.3 Vantaggi della nuova architettura

Questa architettura di streaming, abilitata dal formato JSON, offre vantaggi cruciali rispetto a qualsiasi logica basata su file residenti sul robot:

- **Separazione delle competenze**: È il vantaggio principale. La logica di parsing complessa (JSON) è isolata sul PC. La logica di streaming I/O è isolata sul PLC. Il controllo del processo sul KUKA diventa "pulito" e semplice, concentrandosi solo sull'esecuzione dei movimenti.
- **Prevenzione della Data Starvation**: Il cuore del sistema è la logica a triplo buffer (`BUFF_1`/`BUFF_2`/`BUFF_3`). Questa permette al Task 2 (Assembler) di riempire il Pacchetto $N+1$ in `BUFF_1` contemporaneamente a quando il Task 4 (Main) sta

eseguendo il Pacchetto N. Questo garantisce che ci sia sempre un pacchetto pronto e che il robot non debba mai fermarsi ad attendere i dati, assicurando un processo di deposizione continuo.

- **Scalabilità e flessibilità**: L’intero processo WAAM è definito nel file JSON. Il KUKA è diventato un semplice "esecutore" che non conosce la logica complessiva del pezzo. Questo significa che processi estremamente lunghi o complessi possono essere eseguiti senza sovraccaricare la memoria o le capacità di elaborazione del controller KUKA. Inoltre, attraverso questa architettura, è possibile aggiungere o modificare i processi da effettuare con il robot KUKA, riuscendo ad implementare logiche di recovery o possibili correzioni di mancati riempimenti del bead, che in futuro potranno essere identificati da un algoritmo leggendo i dati provenienti dal profilometro montato sulla torcia.

- **Gestione dinamica del processo**: l’intero flusso può essere modificato sul PC senza necessità di rigenerare i file KRL.

- **Adattamento in tempo reale**: possibile integrare feedback e sensori per modificare il processo “on-the-fly”.

6.4 Progettazione della transizione verso l’implementazione hardware

Il passaggio dalla simulazione all’implementazione hardware richiede l’analisi dettagliata dei protocolli di comunicazione reali. Questa sezione descrive le scelte progettuali per la futura implementazione fisica, già validate logicamente tramite simulazione.

- **Protocollo PC<->PLC: ADS (Automation Device Specification)**: Nella simulazione, questo passaggio avviene tramite variabili condivise. Nella realtà, questa comunicazione è implementata tramite il protocollo ADS (Automation Device Specification) di Beckhoff. L’applicazione C# (client ADS) scrive l’intero pacchetto in una GVL (Global Variable List) sul PLC TwinCAT e imposta un flag booleano (es. g_bPcNewPacket) per segnalare la ricezione.

- **Protocollo PLC<->KUKA: EtherCAT**: Questo è il cambiamento più significativo. La comunicazione non avviene tramite semafori, ma tramite un Fieldbus, tipicamente EtherCAT. Il PLC TwinCAT è il Master EtherCAT, mentre il KUKA è configurato come Slave. Il PLC mappa i singoli dati direttamente su un’area di I/O (output EtherCAT). Il KUKA leggerà questi dati dai suoi input EtherCAT.

6.4.1 Confronto tra attuale e nuova architettura

Per rendere più chiaro l’impatto dello studio svolto non solo sui singoli moduli software ma sull’intero sistema WAAM, in Figura 6.1 è riportato un confronto diretto tra il layout originario della catena di controllo e quello proposto. Il layout attuale è centrato sui file KRL statici generati da Grasshopper e su uno scambio di soli segnali digitali tra

controllore KUKA e PLC; il layout proposto introduce invece un livello di pianificazione su PC, un file JSON gerarchico come unica sorgente dei dati di processo e un'architettura di streaming PC–PLC–KUKA basata su ADS ed EtherCAT, con bufferizzazione esplicita dei pacchetti. Questa riorganizzazione consente di ridurre drasticamente la dimensione del codice KRL, di separare in modo netto dati e logica di controllo e di abilitare funzioni avanzate come la simulazione end-to-end e le procedure di recovery automatico.

Tabella 6.1: Confronto tra architettura attuale e nuova architettura di streaming.

Caratteristica	**Layout attuale (Cap. 4)**	**Layout proposto (Cap. 6)**
Rappresentazione dei dati	Traiettorie codificate in molteplici file .SRC/.DAT, con migliaia di variabili XPn, WDATn, RPn	Unico file JSON gerarchico (layer, bead, pacchetti Start/Loop/End) + ProcessPacket / StProcessPacket
Architettura	Monolitica: Tutta la logica e i dati risiedono nel Controller KUKA	Distribuita: PC (Sender), PLC (Streamer), KUKA (Executor)
Trasferimento Dati	Caricamento manuale file statici (USB/Rete) pre-esecuzione	Streaming continuo in tempo reale (PC → PLC via ADS, PLC → KUKA via EtherCAT)
Manutenibilità del codice	File molto voluminosi, con poche strutture di controllo e logiche avanzate difficili da inserire.	Logica di alto livello in C#, pipeline modulare e riusabile.
Modificabilità	Ogni variazione del processo richiede rigenerare i file KRL completi	Le modifiche sono localizzate nel parser JSON e nel planner; gli eseguibili PLC/KUKA restano sostanzialmente invariati
Supporto al recovery	Identificazione del punto di ripartenza complessa, basata su nomi di variabile e posizione nel codice	Metadati strutturati (LayerIndex, BeadOrSequenceIndex, PacketKind) e logica a loop permettono rientri deterministici

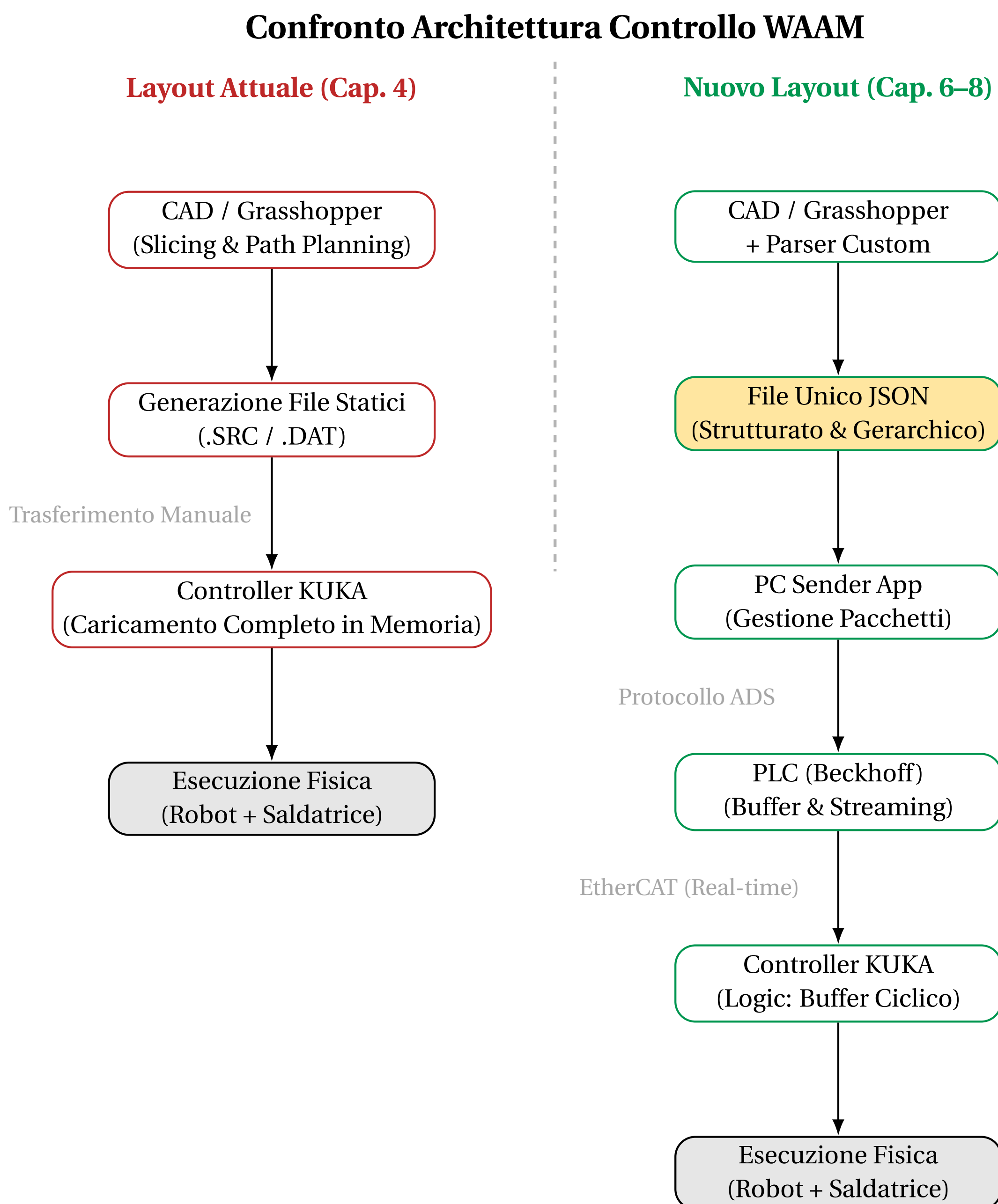

Figura 6.1: Confronto tra layout attuale e nuovo layout della catena di controllo WAAM.

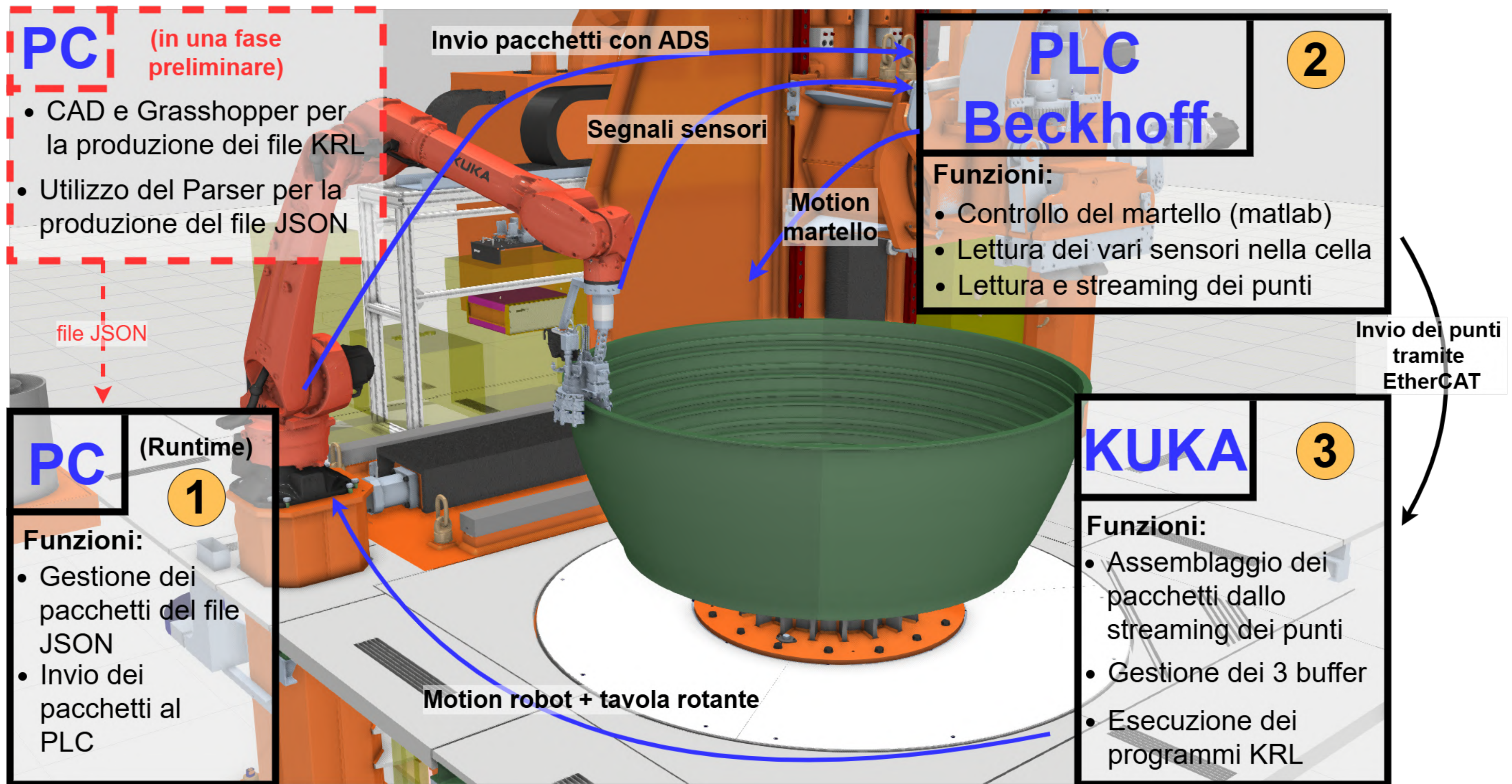

Figura 6.2: Schema a blocchi della catena di controllo WAAM nuova, basata su file JSON.

Capitolo 7

Simulazione software della comunicazione PC-PLC

Nel Capitolo 6 è stata introdotta l'architettura di streaming PC–PLC–KUKA, finalizzata a trasferire al controllore KUKA il contenuto del file JSON di progetto sotto forma di pacchetti di processo, superando il vincolo dei programmi KRL residenti sul robot e abilitando una gestione più flessibile di processi WAAM complessi. Prima di procedere all'implementazione fisica su PLC e robot, è tuttavia necessario disporre di un banco prova virtuale per verificare la correttezza della logica di invio, la coerenza del formato dei pacchetti rispetto ai vincoli di comunicazione e il comportamento temporale dell'handshake PC–PLC.
Questo capitolo presenta quindi lo sviluppo del simulatore software PcPlcSimulation, focalizzato sulla prima tratta della pipeline (PC→PLC). L'applicazione legge il file JSON di progetto e costruisce una timeline ordinata di pacchetti di processo, serializzandoli in una struttura compatibile con un PLC Beckhoff via ADS e gestendo automaticamente il chunking quando un pacchetto eccede i limiti dimensionali. La simulazione include inoltre un meccanismo di handshake basato su flag e buffer condivisi, osservabile tramite interfaccia grafica, che consente di validare l'assenza di conflitti e la coerenza dei dati trasmessi.
Il capitolo è organizzato come segue: la Sezione 7.1 chiarisce obiettivi e ruolo della simulazione; la Sezione 7.2 descrive la catena logica JSON→ProcessPacket→struttura PLC-friendly (inclusi chunking e report di verifica); la Sezione 7.3 analizza il comportamento temporale dell'handshake; la Sezione 7.4 introduce la GUI WPF di monitoraggio; infine, la Sezione 7.5 sintetizza i risultati e la trasferibilità della logica verso l'implementazione su PLC reale.

7.1 Obiettivi e ruolo della simulazione PcPlcSimulation

L'obiettivo principale di PcPlcSimulation è riprodurre in ambiente C# l'intero flusso dati che, in configurazione reale, collegherà il PC al PLC TwinCAT e da questo al robot KUKA. Nella figura 7.1 si delinea ciò che verrà poi approfondito nel seguente capitolo. La simulazione deve:

- leggere il file `projectData_output.json` generato dal parser dei file KRL (layer planari e a spirale);

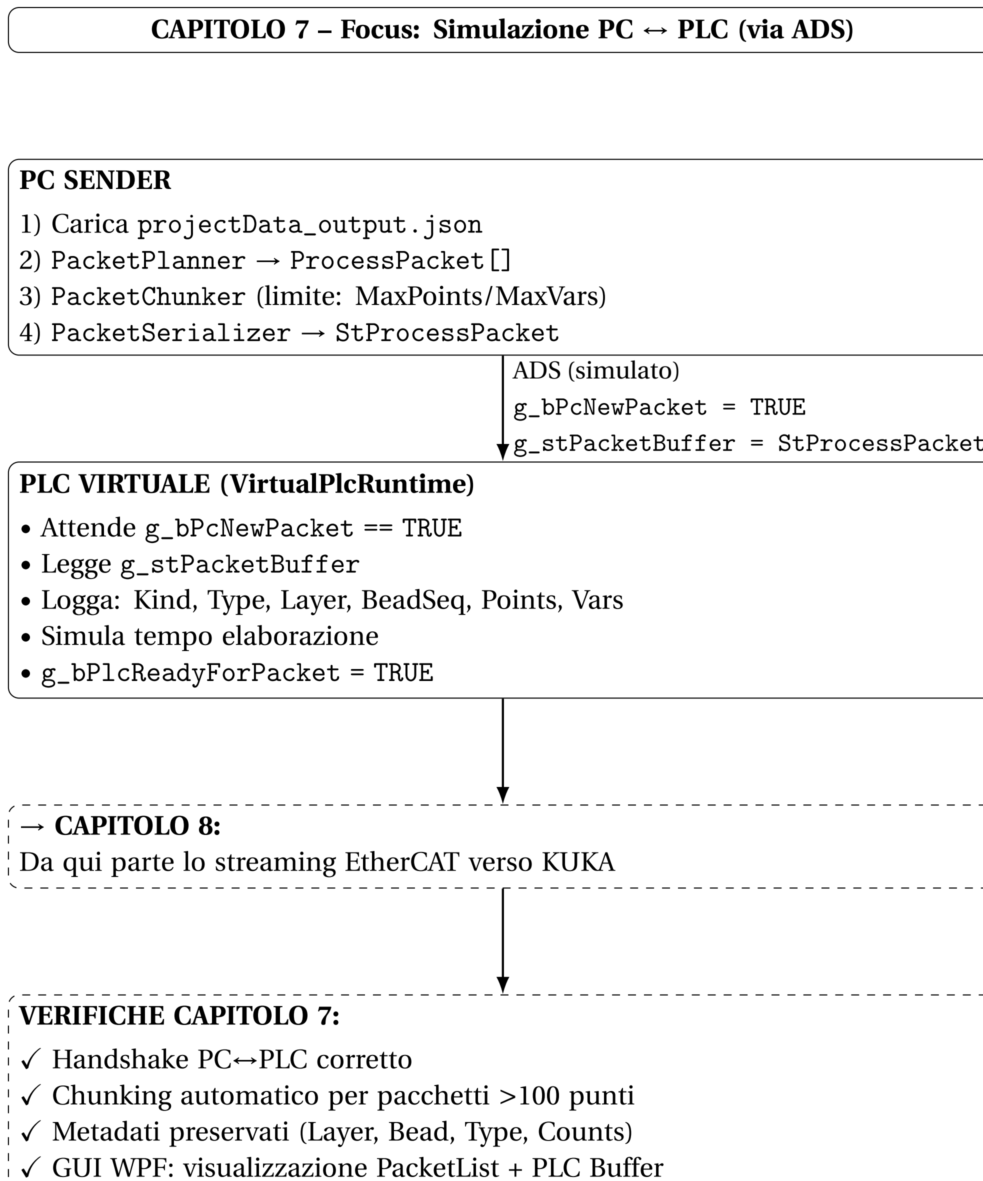

Figura 7.1: Simulazione PC↔PLC via ADS (Cap. 7): generazione e serializzazione del pacchetto su PC, ricezione e verifica su PLC virtuale, e collegamento allo streaming EtherCAT del Cap. 8.

- trasformare il contenuto del JSON in una timeline di pacchetti di processo che descrivono, in ordine temporale, tutte le operazioni di deposizione del progetto WAAM;
- serializzare ogni pacchetto in una struttura dati compatibile con un PLC reale (`struct ST_ProcessPacket`) rispettando i limiti di dimensione imposti dal protocollo ADS (numero massimo di punti e variabili per pacchetto);
- simulare lo scambio dati PC–PLC tramite variabili globali condivise che replicano i flag e il buffer che saranno effettivamente implementati lato TwinCAT;
- fornire un'interfaccia grafica che permetta di osservare la sequenza di invio, lo stato dei flag e il contenuto del buffer PLC, con la possibilità di rallentare o mettere in pausa la "stampa" per analizzarne il comportamento.

In questo modo PcPlcSimulation svolge il ruolo sia di strumento di validazione logica, consentendo di verificare, senza hardware, che il modello dati JSON, la logica di costruzione dei pacchetti e il protocollo di handshake siano coerenti e robusti, sia di palestra per l'implementazione reale dato che la stessa struttura di pacchetto e la stessa interfaccia PC→PLC utilizzate in simulazione sono pensate per essere riutilizzate praticamente senza modifiche nella futura applicazione con PLC reale. L'unica differenza sarà la sostituzione della connessione simulata con una connessione ADS reale.

7.2 Struttura logica del progetto

La parte core di PcPlcSimulation implementa la catena logica che va dal file JSON al buffer PLC. È organizzata in tre blocchi principali: modello di dominio, trasformazione JSON→pacchetti e serializzazione PLC-friendly.

7.2.1 Modello di dominio dei pacchetti di processo

A livello di dominio è stato definito un modello neutro che rappresenta in modo compatto ciò che serve per il processo WAAM, indipendentemente dalle limitazioni del PLC. L'elemento centrale è la classe `ProcessPacket`, che descrive una singola "unità di lavoro" da inviare alla pipeline:

- informazioni di contesto (indici di layer e di bead/sequence);
- tipo di pacchetto (enum PacketType: Start, Loop, End);
- famiglia di layer (PacketKind: DatLinear per i layer planari, Spiral per le sezioni a spirale);
- lista di punti E6POS (posizione e orientamento del TCP più asse esterno);
- lista di variabili numeriche (parametri di saldatura, posizioni della tavola rotante, eventuali canali aggiuntivi);
- metadati per la gestione di loop "spezzati" in più pacchetti (IsLoopChunk, ChunkIndex, TotalChunks).

Questo modello, completamente svincolato da ADS, rende semplice ispezionare e trasformare il processo WAAM con le normali strutture dati C# (liste, enumerable), demandando a uno stadio successivo l'adattamento ai vincoli del PLC.

7.2.2 Dal JSON WAAM alla timeline di ProcessPacket

Il primo stadio della pipeline core consiste nella traduzione del file JSON in una sequenza ordinata di `ProcessPacket`. Il modulo JsonModels descrive la struttura di `projectData_output.json`, in continuità con quanto esposto nel Capitolo 6 e nella figura 5.3: un oggetto `CombinedProjectData` contiene la lista di `CombinedLayer`, ciascuno caratterizzato dal campo Kind che distingue tra layer DAT (planari) e SPIRAL (sezioni a spirale).

A partire da questo modello, il componente `ProjectTimelineBuilder` attraversa tutti i layer in ordine di stampa (indice GlobalLayerIndex) e, per ciascuno, costruisce la successione di pacchetti logici:

- per i layer DAT recupera, per ogni bead, i tre pacchetti logici Start, Loop, End già individuati in fase di parsing;
- per i layer SPIRAL utilizza quanto definito nel capitolo sulla gestione delle spirali: per ogni sequenza genera pacchetti Start (eventuali archi a secco e parametri iniziali), Loop (sequenza di punti LIN/CIRC che descrivono la traiettoria elicoidale) ed End (movimento di uscita e parametri finali).

Il risultato è una timeline lineare di `ProcessPacket` che attraversa tutto il componente, nella quale layer planari e sezioni a spirale sono trattati in modo uniforme e già arricchiti con le informazioni di contesto necessarie al controllo.

7.2.3 Rispetto dei vincoli ADS: PacketChunker

Nella realtà, il pacchetto che il PC invierà al PLC non può essere arbitrariamente grande: il numero massimo di punti e variabili è limitato dalle dimensioni della struttura condivisa via ADS. Per rispettare questi limiti è stato introdotto il componente `PacketChunker`.

Quest'ultimo riceve in ingresso l'`IEnumerable<ProcessPacket>` prodotto dal ProjectTimelineBuilder e applica due vincoli:

- `MaxPointsPerPacket` (numero massimo di punti E6POS per pacchetto);
- `MaxVarsPerPacket` (numero massimo di variabili double per pacchetto).

Se un pacchetto eccede uno dei due limiti, viene automaticamente spezzato in più pacchetti "figli" che mantengono lo stesso Kind, PacketType e contesto di layer, ma contengono solo un sottoinsieme dei punti/variabili originali. Per permettere al PLC di riconoscere e ricomporre queste porzioni di loop, vengono impostati i metadati:

- `IsLoopChunk = true`;
- `ChunkIndex` (indice della porzione del loop);
- `TotalChunks` (numero totale di pacchetti in cui è stato diviso il loop).

In questo modo la timeline risultante è composta esclusivamente da pacchetti compatibili con la struttura dati del PLC, senza perdere informazione sul fatto che alcuni gruppi di pacchetti appartengono a un unico loop di deposizione.

7.2.4 Report di verifica dei pacchetti

La costruzione della timeline di ProcessPacket e il successivo chunking in pacchetti compatibili con il PLC introducono diversi passaggi automatici (filtri, suddivisioni, riordini) che, se non controllati, potrebbero generare discrepanze rispetto al contenuto originale del file JSON. Per questo motivo, al termine della fase di pianificazione dei pacchetti è stato introdotto un report di verifica offline, implementato nella classe

`PacketSummaryReport`.

L'idea è analoga a quella adottata nel Capitolo 4 per la verifica di equivalenza geometrica tra la rappresentazione "classica" e quella "a vettori": prima di procedere con la parte successiva del lavoro, si certifica che la trasformazione dei dati non abbia introdotto errori o perdite di informazione.

PacketSummaryReport opera direttamente sulla lista finale di ProcessPacket generata dal PacketPlanner e produce due tipi di output:

- un report testuale (e, opzionalmente, un file CSV) che elenca tutti i pacchetti con i principali metadati:
  - tipo di layer (Kind: DAT/SPIRAL),
  - tipo di pacchetto (PacketType: START/LOOP/END),
  - indici di layer e di bead/sequence,
  - numero di punti (PointCount) e di variabili (VarCount),
  - informazioni di chunk (IsLoopChunk, ChunkIndex, TotalChunks);
- un riepilogo aggregato che confronta, per ciascun layer, il numero di bead o di sequenze presenti nel JSON con il numero di pacchetti effettivamente generati, distinguendo fra pacchetti Start/Loop/End e, per i loop, fra pacchetti interi e pacchetti spezzati in chunk.

In questo modo il report funziona come un controllore offline sul planner: è possibile, ad esempio, verificare che per ogni bead lineare esistano esattamente tre pacchetti (Start, Loop, End), che il numero di sequenze spirale corrisponda al numero di intervalli $[RP_k; RP_{k+1}]$ individuati dal parser, oppure che il chunking non abbia prodotto pacchetti vuoti o con conteggi incoerenti rispetto ai limiti impostati.

Dal punto di vista operativo, il report viene generato in una fase separata rispetto alla simulazione PC–PLC vera e propria. In una tipica sessione di lavoro, il controllo deve essere effettuato prima di procedere all'esecuzione della simulazione/invio di dati.

7.2.5 Serializzazione PLC-friendly: StProcessPacket e PacketSerializer

Per simulare il PLC e, in prospettiva, comunicare con un controllore reale, è stata definita una struttura dati C# strettamente allineata a quella che verrà dichiarata in TwinCAT:

- `StE6Pos`, che contiene i campi numerici necessari a rappresentare un punto KUKA (coordinate cartesiane, orientamenti, asse esterno, posture);
- `StProcessPacket`, che riunisce:
  - i campi discreti (Kind, PacketType, Layer, PointCount, VarCount, IsLoopChunk, ChunkIndex, TotalChunks);

  - due array a dimensione fissa: `Points[]` (array di StE6Pos) e `Vars[]` (array di double).

Il ponte tra il modello neutro e la struttura PLC-friendly è il componente `PacketSerializer`, che per ogni ProcessPacket:

1. crea un nuovo `StProcessPacket`;
2. copia i metadati (tipo pacchetto, tipo layer, indici, informazioni di chunk);
3. copia fino a MaxPointsPerPacket punti nell'array Points[] e fino a MaxVarsPerPacket variabili in Vars[], lasciando a zero gli elementi non utilizzati;
4. calcola e imposta PointCount e VarCount.

Il risultato è un oggetto pronto per essere scritto 1:1 in una GVL del PLC via ADS, senza ulteriori conversioni.

7.3 Simulazione PC–PLC e comportamento temporale

Una volta definita la struttura del pacchetto, PcPlcSimulation implementa la simulazione dell'handshake PC–PLC descritto nel Capitolo 6, utilizzando tre variabili globali condivise:

- `g_bPlcReadyForPacket`: indica che il PLC è pronto a ricevere un nuovo pacchetto;
- `g_bPcNewPacket`: segnala che il PC ha scritto un nuovo pacchetto nel buffer;
- `g_stPacketBuffer`: contiene l'istanza corrente di StProcessPacket.

7.3.1 Memoria PLC simulata e interfaccia IPlcConnection

Il componente SimulatedPlcMemory implementa in C# una memoria condivisa che contiene i tre elementi sopra elencati, proteggendone l'accesso con un meccanismo di lock per evitare race condition tra thread concorrenti.

Su questa memoria si innesta la classe SimulatedPlcConnection, che implementa un'interfaccia astratta IPlcConnection con metodi generici come `ReadBool`, `WriteBool`, `ReadStruct`, `WriteStruct`. Dal punto di vista del sender, questa interfaccia è indistinguibile da una futura connessione ADS reale: cambierà solo l'implementazione, non la logica alta dell'applicazione.

7.3.2 PacketSendService (PC Sender)

Il servizio `PacketSendService` incapsula la logica di invio dei pacchetti verso il PLC (reale o simulato). Per ogni elemento della timeline:

1. attende che il PLC sia pronto, verificando che g_bPlcReadyForPacket == true e che g_bPcNewPacket == false;

2. utilizza PacketSerializer per trasformare il ProcessPacket corrente in StProcessPacket;
3. scrive l'intera struttura in g_stPacketBuffer tramite IPlcConnection;
4. imposta g_bPcNewPacket = true, segnalando al PLC la presenza di un nuovo pacchetto;
5. attende che il PLC consumi il pacchetto (cioè che g_bPcNewPacket torni false);
6. passa al pacchetto successivo, registrando a log ciascun passaggio.

Questa logica implementa un ping-pong deterministico che garantisce che il PC non sovrascriva mai un pacchetto non ancora letto dal PLC e che entrambe le parti rimangano sincronizzate senza l'uso di code complesse.

7.3.3 VirtualPlcRuntime (PLC virtuale)

Il comportamento del PLC è simulato dal componente `VirtualPlcRuntime`, un loop asincrono che replica la logica del Task 1 della pipeline reale (presentato nella sezione 6.2.2):

1. quando rileva che g_bPcNewPacket == true e g_bPlcReadyForPacket == true, copia localmente il contenuto di g_stPacketBuffer e mette g_bPlcReadyForPacket = false;
2. stampa a log le informazioni principali del pacchetto (tipo, layer, numero di punti e variabili, stato di chunking);
3. attende un tempo configurabile che mima il tempo di elaborazione che il PLC reale impiegherebbe per trasferire i dati verso il robot;
4. al termine, imposta g_bPcNewPacket = false e g_bPlcReadyForPacket = true, dichiarandosi pronto a ricevere il pacchetto successivo.

Il comportamento temporale risultante è molto vicino a quello che verrà implementato in ST su TwinCAT e rappresenta quindi un banco prova affidabile per la logica di streaming.

7.4 Interfaccia grafica WPF per il monitoraggio del processo

Per rendere osservabile il comportamento della pipeline, il core di PcPlcSimulation è stato integrato in una interfaccia grafica WPF. La GUI non aggiunge logica di processo, ma offre una visualizzazione strutturata dello stato interno della simulazione.

7.4.1 Layout della finestra

WAAM PC-PLC Simulation

Carica JSON | Avvia simulazione | Stop | Delay (ms): 1000 | Pacchetti: **278**

Pacchetti da inviare

	#	Kind	Type	Layer	Bead/Seq	IsChunk	ChunkIdx	TotChunks	Points	Vars
✓	1	DatLinear	Start	1	1	False	0	1	2	4
▶	**2**	**DatLinear**	**Loop**	**1**	**1**	**False**	**0**	**1**	**36**	**18**
	3	DatLinear	End	1	1	False	0	1	3	4
	4	DatLinear	Start	1	2	False	0	1	2	4
	5	DatLinear	Loop	1	2	False	0	1	36	18
	6	DatLinear	End	1	2	False	0	1	3	4
	7	DatLinear	Start	1	3	False	0	1	2	4
	8	DatLinear	Loop	1	3	False	0	1	36	18
	9	DatLinear	End	1	3	False	0	1	3	4
	10	DatLinear	Start	1	4	False	0	1	2	4
	11	DatLinear	Loop	1	4	False	0	1	36	18
	12	DatLinear	End	1	4	False	0	1	3	4
	13	DatLinear	Start	1	5	False	0	1	2	4
	14	DatLinear	Loop	1	5	False	0	1	36	18
	15	DatLinear	End	1	5	False	0	1	3	4
	16	DatLinear	Start	1	6	False	0	1	2	4
	17	DatLinear	Loop	1	6	False	0	1	36	18
	18	DatLinear	End	1	6	False	0	1	3	4
	19	DatLinear	Start	1	7	False	0	1	2	4

PLC virtuale

g_bPlcReadyForPacket: FALSE   g_bPcNewPacket: TRUE

Kind: 0   Type: 2   Layer: 1   Points: 36   Vars: 18

Points (E6POS)

#	X	Y	Z	A	B	C	E1	S	T
0	496.64	-87.57	0.2	-460	0	180	10	2	10
1	473.89	-172.48	0.2	-470	0	180	20	2	10
2	436.74	-252.15	0.2	-480	0	180	30	2	10
3	386.32	-324.16	0.2	-490	0	180	40	2	10
4	324.16	-386.32	0.2	-500	0	180	50	2	10
5	252.15	-436.74	0.2	-510	0	180	60	2	10

Vars

#	Value
0	1.45
1	1.45
2	1.45
3	1.45
4	1.45
5	1.45

Log

[PLC] Virtual PLC avviato.
[PC] Sto inviando pacchetto Kind=DatLinear, Type=Start, Layer=1, BeadSeqIndex=1
[PC] Inviato pacchetto Kind=DatLinear, Type=Start, Layer=1, BeadSeqIndex=1
[PLC] Ricevuto pacchetto Kind=0, Type=1, Layer=1, Points=2, Vars=4
[PC] Conferma consumo pacchetto Kind=DatLinear, Type=Start, Layer=1, BeadSeqIndex=1

[PC] Sto inviando pacchetto Kind=DatLinear, Type=Loop, Layer=1, BeadSeqIndex=1
[PC] Inviato pacchetto Kind=DatLinear, Type=Loop, Layer=1, BeadSeqIndex=1
[PLC] Ricevuto pacchetto Kind=0, Type=2, Layer=1, Points=36, Vars=18

Figura 7.2: Layout della finestra di simulazione

La finestra principale (in figura 7.2) è organizzata in:

1. Tabella di sinistra: una DataGrid che mostra la lista completa dei pacchetti da inviare (PacketListItem), con campi quali Kind, Type, indice di layer, bead/sequence, numero di punti e variabili, informazioni di chunk (IsLoopChunk, ChunkIndex, TotalChunks). Le righe sono colorate in base allo stato:
   - giallo e grassetto per il pacchetto in corso di invio;
   - verde per i pacchetti già inviati, con un'icona di stato che indica "in corso" o "completato".
2. Colonna di destra: pannello "PLC virtuale", che replica visivamente la memoria del PLC simulato:
   - indicatori grafici per i flag g_bPlcReadyForPacket e g_bPcNewPacket;
   - etichette che riportano i metadati del pacchetto attualmente presente in g_stPacketBuffer;
   - due DataGrid dedicate rispettivamente all'elenco dei punti E6POS e delle variabili numeriche del pacchetto corrente.
3. Log testuale: Un riquadro inferiore contiene una TextBox multiriga che mostra in tempo reale l'output di log. Per ottenere questo effetto è stato definito un writer personalizzato (UiLogTextWriter) e reindirizzato il flusso standard Console.Out verso il controllo grafico, così che tutti i messaggi generati da PC sender e PLC virtuale siano visibili durante la simulazione.

7.5 Sintesi e considerazioni sulla simulazione

PcPlcSimulation ha permesso di validare in modo sistematico la pipeline che collega il file JSON di progetto al controllore PLC, dimostrando che:

- la rappresentazione globale del processo WAAM in termini di ProcessPacket è sufficiente a descrivere sia i layer planari sia le sezioni a spirale;
- i limiti di dimensione imposti dal protocollo ADS possono essere rispettati tramite chunking automatico, senza introdurre complessità aggiuntiva nel PLC;
- l'handshake basato sui flag g_bPlcReadyForPacket e g_bPcNewPacket garantisce un flusso di pacchetti privo di conflitti e sovrascritture;
- la stessa logica utilizzata in simulazione può essere riutilizzata quasi integralmente in configurazione reale, sostituendo SimulatedPlcConnection con una connessione ADS TwinCAT e VirtualPlcRuntime con il codice ST che implementa il Task 1 dell'architettura di streaming.

Questi risultati potranno essere utilizzati in un implementazione reale futura per la realizzazione del sistema WAAM operativo.

Capitolo 8

Simulazione software della catena PLC–KUKA su EtherCAT

A valle della validazione della tratta PC→PLC presentata nel capitolo precedente, l'ultimo passo per consolidare l'architettura proposta consiste in una verifica completa end-to-end, estesa alla comunicazione PLC→KUKA su bus di campo EtherCAT (come visibile nella figura 8.1). L'obiettivo di questo capitolo è dimostrare che la catena PC–PLC–KUKA non è soltanto concettualmente corretta, ma può essere eseguita e verificata in modo coerente e conservativo: a partire dallo stesso file JSON di progetto, ogni pacchetto pianificato (metadati, punti e variabili di processo) deve risultare trasmesso, ricevuto e ricostruito nel buffer di esecuzione lato robot senza perdite, riordinamenti o ambiguità di contesto.
Per raggiungere tale scopo, il capitolo modella in ambiente software i componenti chiave dello streaming su EtherCAT: il task PLC che serializza i pacchetti in uno stream ciclico (tramite item di tipo Meta/Point/Variable e handshake data/ack) e la logica SPS lato KUKA che interpreta l'immagine di processo, assembla i pacchetti e riempie il buffer BUFF_1 fino alla condizione di completamento. La validazione viene condotta su dati reali del manufatto (porzioni planari e sezioni a spirale), combinando confronto numerico con i report di pianificazione, analisi dei log e osservazione tramite interfaccia grafica WPF, così da ottenere una verifica sia quantitativa sia qualitativa del comportamento della pipeline.
Il capitolo è organizzato come segue: la Sezione 8.1 definisce obiettivi e criteri di validazione; la Sezione 8.2 richiama l'architettura complessiva da simulare e il ruolo dei tre attori; la Sezione 8.3 dettaglia la modellazione del protocollo PLC–KUKA su EtherCAT (immagine di processo, StreamItem, task PLC e task KUKA di assemblaggio/riempimento BUFF_1); la Sezione 8.4 presenta e discute i risultati su casi rappresentativi; infine, la Sezione 8.5 valuta criticamente il ruolo del PLC nella catena, discutendo la possibilità di architetture alternative PC–KUKA dirette in prospettiva industriale. Nel complesso, la simulazione realizzata costituisce un "gemello digitale" del flusso dati e una specifica eseguibile dell'architettura proposta, riducendo rischio e tempi di messa in servizio sull'impianto reale.

8.1 Obiettivi simulazione PC-PLC-KUKA

L'obiettivo di questo capitolo è dimostrare che l'architettura proposta per la comunicazione PC–PLC–KUKA non è solo concettualmente corretta, ma può essere eseguita

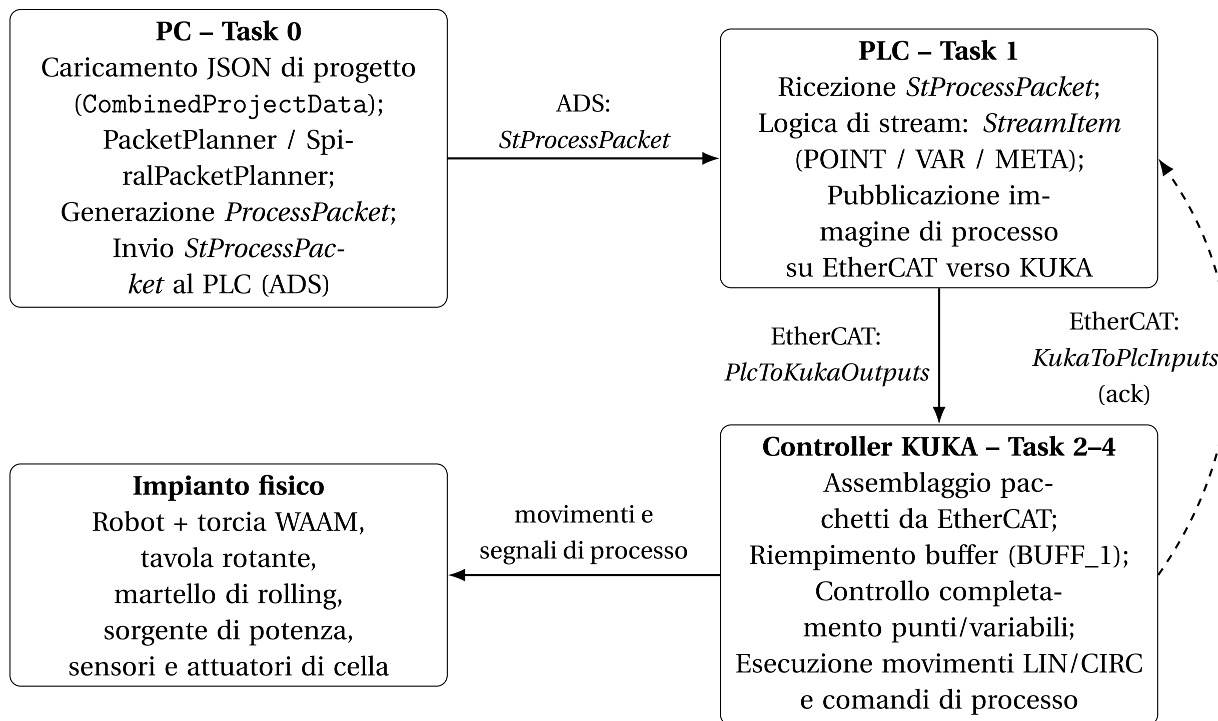

Figura 8.1: Architettura proposta di streaming PC–PLC–KUKA per il processo WAAM: dal JSON di progetto alla cella fisica tramite ADS e EtherCAT.

e verificata in modo completo in ambiente software, partendo dallo stesso file JSON utilizzato per la pianificazione dei percorsi WAAM. In altri termini, si vuole mostrare che l'intera catena dati, dalla descrizione geometrico-tecnologica del processo fino alla simulazione del riempimento del buffer BUFF_1 lato KUKA, è coerente, tracciabile e priva di perdite di informazione.

Questa simulazione end-to-end si inserisce in un contesto applicativo caratterizzato da un'elevata complessità: l'integrazione tra un sistema di pianificazione esterno (PC), uno o più PLC Beckhoff e un controllore robotico KUKA, tutti collegati tramite EtherCAT, comporta numerosi passaggi intermedi, conversioni di formato e vincoli hardware. Arrivare direttamente a testare questa architettura sull'impianto reale, senza un passaggio intermedio di validazione software, significherebbe esporsi a rischi elevati in termini di tempi di debug, possibili errori di comunicazione e potenziali collisioni o difetti di deposito durante le prove WAAM.

Dal punto di vista della tesi, la simulazione software completa contribuisce in modo diretto agli obiettivi generali: aumentare la robustezza del flusso dati WAAM e ridurre il numero di iterazioni di prova-errore necessarie in fase di messa in servizio. Disporre di una versione simulata della catena PC–PLC–KUKA permette di verificare che, prima ancora di accendere l'arco, il sistema sia in grado di veicolare correttamente tutti i dati di traiettoria e di processo richiesti.

8.1.1 Motivazioni e contesto applicativo

Una simulazione end-to-end è necessaria, innanzitutto, perché l'architettura PC–PLC–KUKA non è un semplice collegamento punto-punto, ma un sistema distribuito in cui più

attori cooperano in modo sincrono:

- il PC esegue il parsing dei file di partenza e genera un file JSON strutturato con tutti i layer, i bead e le sequenze a spirale;
- un primo livello logico traduce questa rappresentazione in ProcessPacket e successivamente in StProcessPacket, più adatti al mondo PLC;
- un ulteriore livello (task EtherCAT) serializza questi pacchetti in uno stream ciclico di dati (immagine di processo);
- infine il controllore KUKA legge lo stream, ricostruisce i punti e le variabili, e riempie i propri buffer interni (BUFF_1 e storici).

In ciascuna di queste trasformazioni esiste il rischio di introdurre errori difficili da individuare direttamente sul robot (punti mancanti, conteggi sbagliati, ordine non corretto dei pacchetti, incoerenza tra header e payload, ecc.). Una simulazione software completa consente di isolare questo livello di rischio: si può verificare che, dato un file JSON e una sequenza di pacchetti, il risultato lato KUKA simulato sia esattamente quello atteso, senza dover coinvolgere immediatamente l'hardware reale.

8.1.2 Domande di ricerca e obiettivi tecnici

Da questa motivazione derivano alcune domande chiave a cui il capitolo vuole rispondere:

- Coerenza della pipeline dati: a partire da un file JSON di progetto WAAM, l'intera catena JSON → ProcessPacket → StProcessPacket → StreamItem → PlcToKukaOutputs → BUFF_1 mantiene invariati il numero di punti, il numero di variabili e il contesto (layer, bead/sequence, chunking)?
- Correttezza del protocollo PC→PLC: la struttura StProcessPacket, con i suoi enumerativi compressi, gli indici di layer/bead/sequence e i conteggi PointCount/VarCount, viene generata e interpretata correttamente per tutte le tipologie di pacchetti (Start/Loop/End, layer DAT, layer a spirale, chunk di loop a spirale)?
- Affidabilità del protocollo PLC→KUKA su EtherCAT: l'immagine di processo progettata (strutture PlcToKukaOutputs e KukaToPlcInputs) garantisce:
  - il rispetto dei vincoli dimensionali tipici dei PDO EtherCAT,
  - un handshake chiaro data/ack,
  - una chiusura non ambigua dei pacchetti, per evitare la perdita o l'organizzazione errata di dati.

A queste domande si aggiunge un obiettivo trasversale: verificare che la modellazione software sia sufficientemente fedele da poter essere riutilizzata quasi direttamente nel contesto reale. In altre parole, si vuole dimostrare che il codice sviluppato per il simulatore non è un prototipo "usa e getta", ma una vera e propria specifica eseguibile dell'architettura di comunicazione PC–PLC–KUKA.

8.1.3 Criteri di validazione e ruolo della GUI

Per rispondere in modo convincente alle domande precedenti, il capitolo adotta tre criteri principali di validazione:

1. Confronto numerico con i report di pianificazione: i report generati dal PacketPlanner e dagli strumenti di parsing forniscono, per ogni layer e per ogni pacchetto, il numero atteso di punti e di variabili, insieme alle informazioni di contesto (indici di layer e bead, eventuale chunking, ecc.). Questi valori fungono da riferimento con cui confrontare ciò che il simulatore KUKA riceve effettivamente. Se, per ogni pacchetto, i contatori di punti e variabili in BUFF_1 coincidono con quelli dichiarati, si può affermare che la catena dati è conservativa.

2. Analisi dei log testuali: tutti i componenti della simulazione (PC Sender, PLC simulato, task EtherCAT, KUKA simulato) producono log di dettaglio: invio dei pacchetti, handshake PC–PLC, item EtherCAT (Meta, Point, Variable), log di riempimento “BUFF_1 FULL”, chiusure di pacchetto, eventuali warning in caso di mismatch. L’analisi di questi log permette di ricostruire ex post il comportamento della catena e di individuare eventuali anomalie nella sequenza di eventi.

3. Osservazione tramite interfaccia grafica: accanto ai log testuali, è stato sviluppato un simulatore grafico (progetto WPF) che rende osservabile il comportamento interno del sistema: per ogni pacchetto selezionato è possibile vedere:

   - la sequenza degli item effettivamente inviati dal PLC sull’immagine di processo;
   - lo stato corrente del BUFF_1 simulato lato KUKA (punti ricevuti, variabili ricevute, header descrittivo);
   - l’evoluzione dei conteggi attesi/ricevuti fino al log “BUFF_1 FULL”.

L’uso combinato di report, log e GUI consente una validazione sia quantitativa che qualitativa della soluzione proposta: non solo si controlla che i numeri tornino, ma si può anche vedere come il sistema li fa tornare, passo dopo passo.

8.2 Richiamo dell’architettura da simulare

Prima di entrare nel dettaglio della simulazione, è utile richiamare brevemente l’architettura PC–PLC–KUKA descritta nei capitoli precedenti, evidenziando quali componenti vengono modellati in questo capitolo e quali rimangono invece a livello concettuale o hardware. Riassunto grafico dell’intera architettura presentato nella figura 8.2

L’architettura proposta si basa su un file JSON centrale, come formato di scambio fra gli strumenti di pianificazione del percorso e la catena di controllo dell’impianto WAAM. A valle della generazione dei percorsi (DAT tradizionali e sezioni a spirale), un modulo di parsing e combinazione costruisce una struttura `CombinedProjectData` composta da un elenco ordinato di layer, ciascuno dei quali contiene i bead o le sequenze a spirale da depositare.

Su questa base si innestano tre livelli principali:

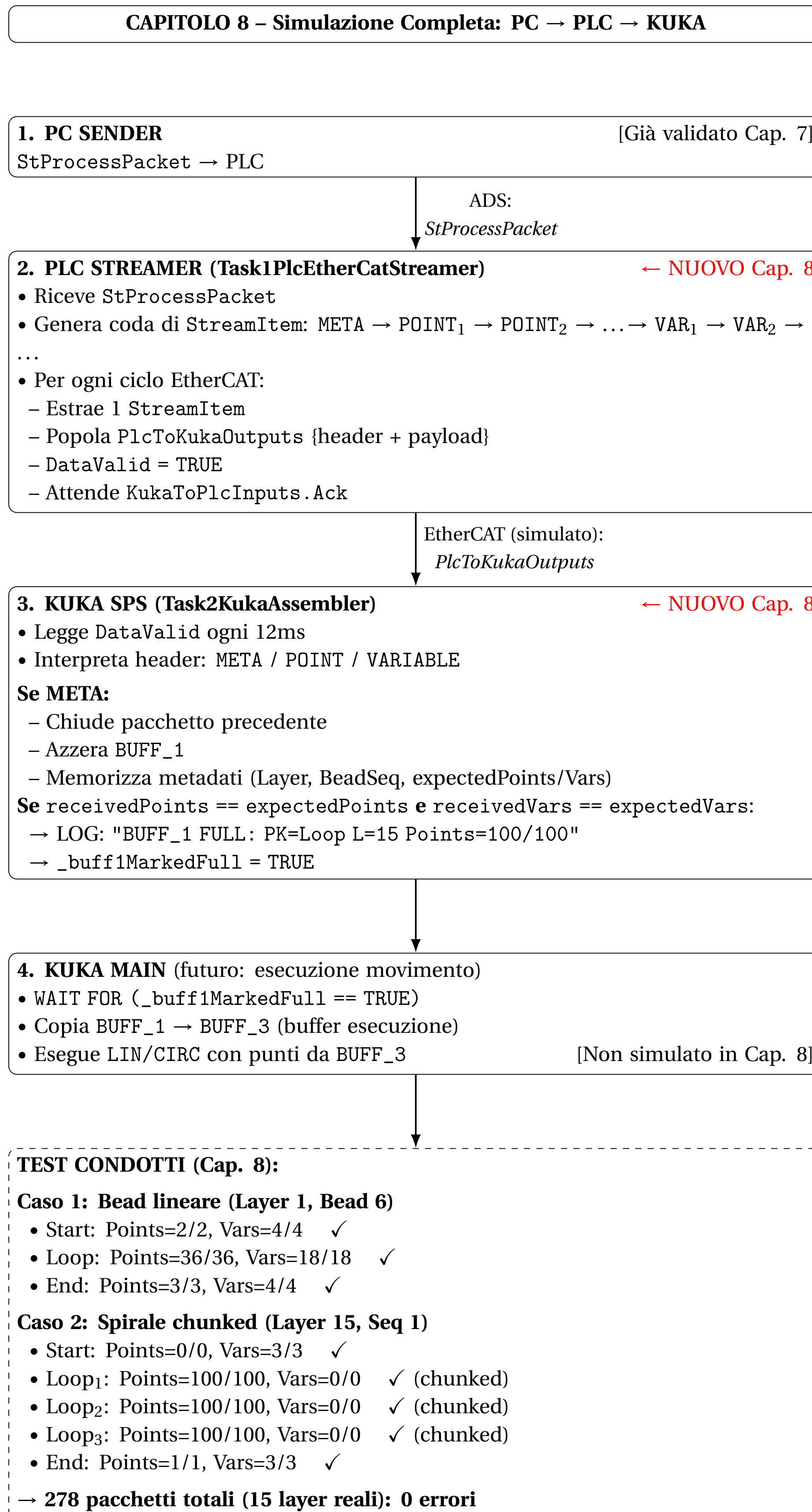

Figura 8.2: Flusso completo della simulazione PC–PLC–KUKA (Cap. 8): generazione pacchetti, streaming EtherCAT, assemblaggio su KUKA e casi di test.

- **PC WAAM planner e PC Sender**: Il PC non si limita a generare i percorsi robot, ma costruisce un modello logico di processo, sotto forma di ProcessPacket e successivamente di StProcessPacket, adatto a essere trasmesso a un PLC. In questa fase vengono codificati:
  - il tipo di pacchetto (Start/Loop/End),
  - il tipo di layer (planare, spirale, ecc.),
  - gli indici globali e locali (layer, bead, sequenza),
  - i punti di traiettoria e le variabili di processo associate.
- **PLC di processo e PLC di comunicazione**: Nello scenario reale, il PLC svolge due ruoli logici:
  - ricezione dei pacchetti logici dal PC tramite ADS/TC3 (qui modellato tramite `SimulatedPlcConnection` e `SimulatedPlcMemory`), memorizzazione in una sorta di "buffer di progetto" e messa a disposizione del task di comunicazione;
  - serializzazione dei pacchetti in uno stream ciclico di dati EtherCAT, attraverso un task dedicato che mappa le informazioni sui PDO del bus e gestisce il protocollo di handshake con il KUKA.
- **Controllore KUKA (KRC5)**: Sul lato robot, il controllore KUKA legge ad ogni ciclo l'immagine di processo EtherCAT e popola i propri buffer.

È esattamente questo ciò che il capitolo si propone di riprodurre in ambiente simulato.

8.2.1 Ruolo della simulazione rispetto all'architettura reale

La simulazione software implementata in questa tesi replica il comportamento dei tre ruoli principali – PC Sender, PLC, KUKA – sostituendo le componenti hardware e i protocolli di campo con moduli software equivalenti:

- la comunicazione PC–PLC viene modellata tramite una connessione simulata (`SimulatedPlcConnection`) e una memoria di processo (`SimulatedPlcMemory`) che espongono la stessa interfaccia logica che verrebbe usata con un PLC Beckhoff reale;
- la comunicazione PLC–KUKA su EtherCAT viene sostituita da un bus virtuale (EtherCatBus), su cui girano le stesse strutture dati (`PlcToKukaOutputs`, `KukaToPlcInputs`) previste per i PDO reali;
- il controllore KUKA viene modellato da una classe (`Task2KukaAssembler`) che implementa la logica SPS di lettura dell'immagine di processo e riempimento del BUFF_1, rispettando il protocollo di handshake e la gestione dei buffer prevista per l'impianto reale.

Questa scelta permette di lavorare con gli stessi formati e le stesse strutture dati che verranno utilizzati in campo, ma in un ambiente completamente controllato. In sintesi, la simulazione non è una semplificazione dell'architettura reale, ma un suo gemello logico focalizzato sul solo flusso dati. Nel seguito del capitolo verrà descritta nel dettaglio la modellazione della comunicazione PLC–KUKA su EtherCAT e saranno presentati i risultati ottenuti, combinando log e visualizzazione grafica.

8.3 Modellazione comunicazione PLC–KUKA su EtherCAT

Dopo aver definito l'architettura logica complessiva PC–PLC–KUKA, questo paragrafo entra nel dettaglio di come è stata modellata la comunicazione fra il PLC e il controllore KUKA su bus EtherCAT. L'obiettivo non è solo simulare uno scambio generico di dati, ma riprodurre fedelmente il modo in cui, nello scenario reale, un task PLC Beckhoff invierà i pacchetti di processo al KUKA tramite PDO EtherCAT, e un programma SPS sul KUKA ricostruirà i punti e le variabili di ogni pacchetto nel buffer BUFF_1.

La modellazione proposta si articola in tre elementi principali:

1. la definizione dell'immagine di processo EtherCAT (strutture `PlcToKukaOutputs` e `KukaToPlcInputs`) e del concetto di `StreamItem` come unità di dato trasmesso per ciclo;
2. l'implementazione del task PLC EtherCAT simulato (`Task1PlcEtherCatStreamer`), che traduce una coda di `StreamItem` nella sequenza di valori pubblicati sull'immagine di processo e gestisce l'handshake con il KUKA;
3. l'implementazione del task KUKA SPS simulato (`Task2KukaAssembler`), che legge l'immagine di processo, accumula i punti e le variabili in buffer locali e segnala il completamento del pacchetto quando tutti i dati attesi sono stati ricevuti.

Nei paragrafi successivi verranno descritti questi tre elementi e la loro integrazione all'interno del simulatore.

8.3.1 Struttura dell'immagine di processo EtherCAT

Nel sistema reale, la comunicazione PLC–KUKA avviene tramite PDO EtherCAT, ossia tramite due blocchi di memoria condivisi ciclicamente:

- un blocco di uscita PLC $\rightarrow$ KUKA (output del PLC, input del KUKA),
- un blocco di ingresso KUKA $\rightarrow$ PLC (output del KUKA, input del PLC).

Per poter simulare questo scambio, nel progetto è stata definita una coppia di strutture dati che rappresentano l'immagine di processo vista dai due sistemi:

1. `PlcToKukaOutputs` contiene tutto ciò che il PLC invia al KUKA ad ogni ciclo:
   - campi di handshake (ad esempio DataValid, che indica che il contenuto dell'output è valido per essere letto; bit di reset/ack);
   - un header di contesto del pacchetto in corso, che codifica:
     - il tipo di pacchetto (Start/Loop/End);
     - il tipo di layer (planare, spirale, ecc.);
     - gli indici di layer e di bead o sequenza;
     - indici di chunk (per gestire le spirali spezzate in più pacchetti Loop);
     - il numero atteso di punti e variabili per il pacchetto;
     - l'indice dell'elemento corrente all'interno del pacchetto (point index / variable index);

- un payload numerico che, a seconda del tipo di elemento trasmesso, contiene:
  - le coordinate del punto (X, Y, Z, orientazioni, asse esterno E1, ecc.);
  - oppure il valore di una variabile di processo (corrente, parametro tavola, ecc.);

2. `KukaToPlcInputs` contiene invece ciò che il KUKA rimanda al PLC:
   - bit di acknowledge (SpsAckData o equivalenti), che confermano la corretta lettura di un certo elemento;
   - eventuali campi diagnostici (errori, segnalazioni di mismatch, ecc.), che nella simulazione possono essere usati per testare scenari anomali.

Un vincolo importante è che la struttura `PlcToKukaOutputs` è stata dimensionata per rimanere compatibile con una finestra PDO realistica: anche sommando header e payload, la dimensione complessiva rimane sotto una soglia (tipicamente 256 byte) che rende plausibile il mapping su un modulo EtherCAT reale. Questo vincolo ha guidato alcune scelte di codifica, come l'uso di tipi interi compatti per gli enumerativi e l'attenzione al numero di campi numerici trasmessi per ogni elemento.

8.3.2 Flusso logico dei dati e concetto di StreamItem

Uno StreamItem rappresenta un'unità di dato che il PLC invia al KUKA in un singolo ciclo EtherCAT. Esistono tre tipologie fondamentali:

- **Meta**: è un elemento speciale che funge da testata del pacchetto. Quando il PLC invia uno StreamItem di tipo Meta, sta dicendo al KUKA:
  - che sta per iniziare la trasmissione di un nuovo pacchetto (Start/Loop/End);
  - quali sono i suoi metadati (tipo, layer, bead/sequence, eventuali indici di chunk);
  - quanti punti e quante variabili complessive ci si aspetta di trasmettere per questo pacchetto.
- **Point**: rappresenta un singolo punto di traiettoria all'interno del pacchetto corrente. Contiene le coordinate del punto e tutte le informazioni accessorie necessarie (orientazioni, asse esterno, eventuali flag di processo).
- **Variable**: rappresenta una singola variabile di processo associata al pacchetto corrente (ad esempio, una corrente di saldatura per quel tratto, un parametro tavola o un coefficiente di velocità).

A partire da ogni pacchetto logico destinato al PLC–KUKA (che deriva dai ProcessPacket già "compressi" in StProcessPacket nel livello PC–PLC), il sistema costruisce una lista ordinata di StreamItem. In generale, per un pacchetto tipo si ha una sequenza del tipo:

- 1 elemento Meta;
- N elementi Point (dove N è il numero di punti del pacchetto);

- M elementi Variable (dove M è il numero di variabili di processo del pacchetto).

Questa lista viene messa in coda a un componente di streaming (il task PLC simulato). Ogni ciclo EtherCAT, il PLC prende il prossimo StreamItem dalla coda, lo mappa sull'immagine di processo (`PlcToKukaOutputs`) e attende la conferma di ricezione da parte del KUKA.

8.3.3 Modello del task PLC EtherCAT: Task1PlcEtherCatStreamer

Il comportamento del task PLC di comunicazione è modellato dalla classe `Task1PlcEtherCatStreamer`. Questo componente riceve in ingresso la lista di StreamItem da trasmettere (una per ogni pacchetto di processo) e si occupa di popolare ad ogni ciclo la struttura `PlcToKukaOutputs`, implementando un semplice automa a stati finiti che riproduce il pattern data/ack tipico dei sistemi EtherCAT.

In sintesi, il suo funzionamento può essere descritto come segue:

- **Stato Idle**
  - se non ci sono StreamItem in coda, il task non fa nulla: mantiene DataValid a falso e l'output a valori neutri;
  - se è presente almeno uno StreamItem, lo estrae dalla coda e:
    - compila i campi dell'header in PlcToKukaOutputs in funzione del tipo (Meta, Point, Variable);
    - scrive il payload numerico appropriato (coordinate del punto o valore della variabile);
    - imposta DataValid = true per segnalare al KUKA che un nuovo elemento è pronto;
    - passa allo stato **WaitingAck**.
- **Stato WaitingAck**
  - il task rimane in questo stato finché il bit di acknowledge in KukaToPlcInputs non indica che il KUKA ha letto l'elemento;
  - una volta ricevuto l'ack:
    - resetta DataValid a falso e, se necessario, pulisce parte dell'output;
    - memorizza internamente il fatto che quel determinato elemento è stato trasmesso con successo (utile per logging e per statistiche);
    - torna allo stato **Idle** per prelevare il prossimo StreamItem.

Questo schema garantisce che, in ogni ciclo, al KUKA venga proposto al massimo un elemento (Meta, Point o Variable) e che non si proceda al successivo finché quello corrente non è stato esplicitamente riconosciuto. In questo modo si evita qualsiasi ambiguità su eventuali salti, elementi persi o sovrascritti: l'ordine degli StreamItem nella coda coincide esattamente con l'ordine degli elementi letti dal KUKA.

In un'implementazione reale, lo stesso pattern potrebbe essere riprodotto nel codice PLC (Structured Text) utilizzando:

- una struct o un blocco dati dedicato per l'output EtherCAT,
- una macchina a stati analoga (Idle, Send, WaitAck),
- un bit di ack letto dal PDO di ingresso proveniente dal KUKA.

8.3.4 Modello del task KUKA SPS: Task2KukaAssembler e BUFF_1

Sul lato KUKA, il comportamento dell'SPS di assemblaggio pacchetti è modellato dalla classe `Task2KukaAssembler`. Questo componente legge ad ogni ciclo la struttura PlcToKukaOutputs, interpreta i campi a seconda del tipo di elemento (Meta, Point, Variable) e aggiorna una serie di buffer e contatori interni.

Dal punto di vista logico, `Task2KukaAssembler` mantiene:

- buffer storici: con due elenchi globali di tutti i punti e variabili ricevuti.
- un buffer per pacchetto (BUFF_1), che contiene esclusivamente i punti e le variabili del pacchetto corrente con i metadati di contesto(tipo pacchetto, layer, bead/sequence, chunk, ecc.);
- una serie di contatori interni:
  - expectedPoints e expectedVars (valori attesi, derivati dall'elemento Meta);
  - receivedPoints e receivedVars (valori effettivamente ricevuti);
  - flag come _hasOpenPacket e _buff1MarkedFull, che indicano se c'è un pacchetto in corso e se è già stato segnalato il completamento.

La logica, rappresentata nel flowchart in figura 8.3, che viene eseguita ad ogni ciclo può essere riassunta così:

1. **Lettura di DataValid**
   - se `DataValid` è falso, il KUKA non fa nulla: non ci sono nuovi dati da processare;
   - se `DataValid` è vero, legge l'header per capire che tipo di elemento è stato trasmesso (Meta, Point, Variable).
2. **Gestione di un elemento Meta**
   - se era già aperto un pacchetto, lo chiude forzatamente controllando che i contatori attesi/ricevuti coincidano (eventualmente loggando un warning in caso di mismatch);
   - azzera il BUFF_1 e i contatori receivedPoints/receivedVars;
   - memorizza in BUFF_1 tutti i metadati del pacchetto (Kind, Type, Layer, Bead/Sequence, chunk, conteggi attesi, ecc.);
   - imposta expectedPoints e expectedVars con i valori forniti dal PLC;
   - imposta _hasOpenPacket = true, segnalando che un nuovo pacchetto è in corso.

3. **Gestione di un elemento Point**

   - se non c'è un pacchetto aperto, l'elemento viene ignorato o può generare un warning;
   - altrimenti:
     - aggiunge il punto sia al buffer storico (BuffPunti) sia al BUFF_1;
     - incrementa receivedPoints;
     - verifica se:

       $$\text{receivedPoints} == \text{expectedPoints} \wedge \text{receivedVars} == \text{expectedVars}$$

       Se sì, marca BUFF_1 come pieno (_buff1MarkedFull = true) e logga un messaggio del tipo: BUFF_1 FULL: PK=Start/Loop/End, Layer= ..., BeadSeq=..., Points=ricevuto/atteso, Vars=ricevuto/atteso.

4. **Gestione di un elemento Variable**

   - logica analoga a quella dei punti:
     - aggiunge la variabile al buffer storico (BuffVars) e al BUFF_1;
     - incrementa receivedVars;
     - verifica se, insieme ai punti, si è raggiunto il conteggio atteso per il pacchetto, eventualmente marcando BUFF_1 come pieno e loggando il completamento.

In questo modo, quindi, il BUFF_1 simulato riproduce il comportamento che si vorrebbe ottenere in KRL:

- ricevere esattamente i punti e le variabili di un pacchetto,
- sapere quando il pacchetto è completo e quindi pronto per essere consumato dalla parte di motion,
- poter verificare ex post, tramite log e contatori, che non ci siano stati né punti mancanti né duplicazioni.

8.3.5 Integrazione nel simulatore PLC–KUKA

Gli elementi descritti finora, immagine di processo, StreamItem, Task1PlcEtherCatStreamer, Task2KukaAssembler, vengono incapsulati all'interno di un motore di simulazione dedicato e di una interfaccia grafica WPF. In questo modo è possibile:

- alimentare il simulatore con i pacchetti generati a monte (derivanti dai ProcessPacket della pianificazione);
- eseguire ciclicamente il task PLC e il task KUKA, come avverrebbe in campo;
- registrare e visualizzare in tempo reale ciò che sta transitando sul bus (lista di StreamItem effettivamente inviati) e lo stato del BUFF_1 per ciascun pacchetto.

Il componente centrale è un engine che contiene:

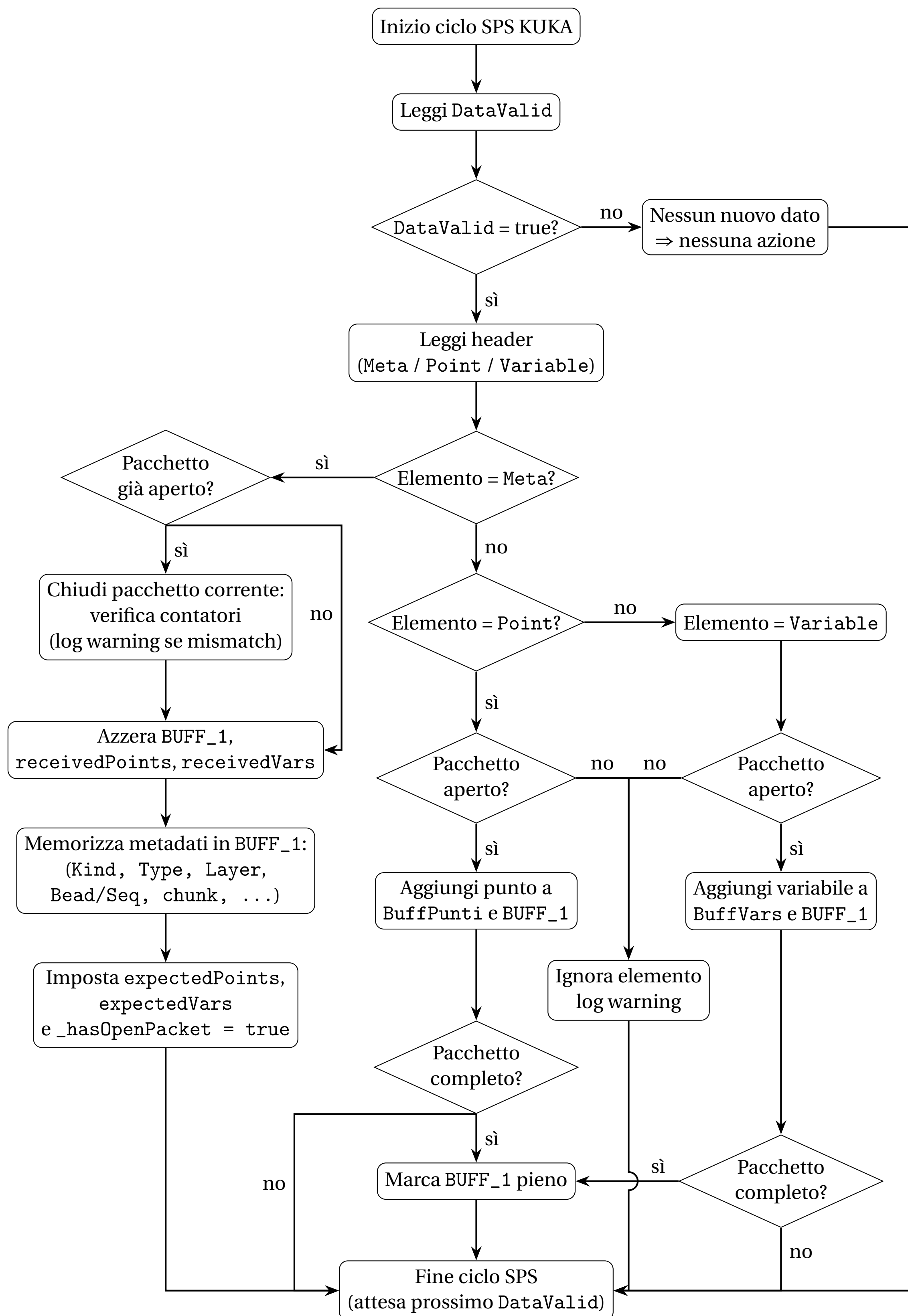

Figura 8.3: Flowchart del task KUKA SPS: Task2KukaAssembler e BUFF_1.

- un'istanza di EtherCatBus (struttura che incapsula i due blocchi PlcToKukaOutputs e KukaToPlcInputs);
- un'istanza del task PLC (Task1PlcEtherCatStreamer), a cui vengono forniti gli StreamItem derivati dai pacchetti da simulare;
- un'istanza del task KUKA (Task2KukaAssembler), che legge i valori dal bus e aggiorna i buffer.

Su questo motore si appoggia poi la GUI WPF, presentata nelle figure A.9 - A.14: la parte grafica non modifica la logica della simulazione, ma si limita a visualizzare, in forma di tabelle e testi descrittivi, i dati estratti dall'engine (lista pacchetti, item Ethernet realmente inviati, contenuto corrente del BUFF_1, log di completamento, ecc.).

In sintesi, la modellazione della comunicazione PLC–KUKA proposta in questo capitolo costituisce un modello software molto vicino a ciò che potrebbe essere implementato su PLC Beckhoff e KUKA KRC5: stessi formati, stesso protocollo di handshake, stessi concetti di buffer e di completamento pacchetto.

8.4 Presentazione dei risultati tramite log e interfaccia grafica

In questa sezione vengono presentati i risultati principali ottenuti con la simulazione software della catena PC–PLC–KUKA, utilizzando in modo congiunto:

- i log testuali prodotti dal simulatore, che descrivono in dettaglio la sequenza di pacchetti inviati, gli item EtherCAT (Meta, Point, Variable), i messaggi di completamento "BUFF_1 FULL" e gli eventuali warning;
- l'interfaccia grafica WPF sviluppata per il tratto PLC–KUKA, che offre una vista ad alto livello sull'avanzamento della simulazione: elenco dei pacchetti, contenuto del BUFF_1, header descrittivo del pacchetto corrente.

La simulazione non è stata eseguita su un caso sintetico, ma sul file JSON del pezzo reale, ricavato dal lavoro di slicing e parsing descritto nei capitoli precedenti. In particolare, sono stati simulati tutti i pacchetti che coprono i primi 15 layer reali del manufatto (che rappresentano la sezione in figura 5.1): layer planari a bead lineari e layer a spirale. Complessivamente il sistema simula 278 pacchetti, che vengono trasmessi dal PC al PLC e, successivamente, dal PLC al KUKA simulato tramite l'immagine di processo EtherCAT.

All'interno di questa simulazione completa sono stati selezionati due casi di test rappresentativi:

1. un bead lineare standard nel layer 1 (bead 6), modellato come tripletta Start–Loop–End;
2. una sequenza a spirale nel layer 15 (sequence 1), corrispondente ad una parte della sezione 4 nella figura 5.1, anch'essa modellata come tripletta Start–Loop–End, ma composta da più pacchetti loop per via della limitazione dimensionale del pacchetto, che ha richiesto la divisione dell'intero loop.

Per ciascun caso vengono analizzati estratti significativi di log, accompagnati dagli screenshot dell'interfaccia grafica relativi al pacchetto Start, al pacchetto Loop e al pacchetto End.

8.4.1 Caso di test 1 – Bead lineare standard (layer 1, bead 6)

Il primo caso di test considera un bead lineare planare appartenente al layer 1 del pezzo reale, in particolare il sesto bead. Il PacketPlanner scompone questo bead in tre pacchetti:

- un pacchetto Start, con due punti di approccio e quattro variabili di processo (correnti e parametri tavola) – Points=2, Vars=4;
- un pacchetto Loop, con 36 punti di traiettoria che descrivono il contorno circolare e 18 variabili di processo distribuite lungo il tratto di deposito – Points=36, Vars=18;
- un pacchetto End, con tre punti di allontanamento e quattro variabili per lo spegnimento controllato dell'arco – Points=3, Vars=4.

```
1  [PC] Sto inviando pacchetto Kind=DatLinear, Type=Start, Layer=1, BeadSeqIndex=6
2  [PC] Inviato pacchetto Kind=DatLinear, Type=Start, Layer=1, BeadSeqIndex=6
3  [PLC] Ricevuto pacchetto Kind=0, Type=1, Layer=1, BeadSeqIndex=6, Points=2,
       Vars=4
4  [PLC] Avvio simulazione EtherCAT verso KUKA per questo pacchetto...
5  [KUKA] META: L=1, Kind=Start, BeadSeq=6, PtsExpected=2, VarsExpected=4
6  [KUKA] POINT #1: L=1, PK=Start, BeadSeq=6, X=276.31 Y=-394.62 Z=200.2 S=2 T=10
7  [KUKA] POINT #2: L=1, PK=Start, BeadSeq=6, X=276.31 Y=-394.62 Z=0.2 S=2 T=10
8  [KUKA] VAR #3: L=1, PK=Start, BeadSeq=6, Val=0
9  [KUKA] VAR #4: L=1, PK=Start, BeadSeq=6, Val=481.74
10 [KUKA] VAR #5: L=1, PK=Start, BeadSeq=6, Val=295.2
11 [KUKA] VAR #6: L=1, PK=Start, BeadSeq=6, Val=0
12 [KUKA] BUFF_1 FULL: PK=Start L=1 BeadSeq=6 Points=2/2 Vars=4/4
13 [KUKA] BUFF_1 COMPLETE (end of packet, counts OK): PK=Start L=1 BeadSeq=6
       Points=2 Vars=4
14 [PLC] EtherCAT completato: Punti=2, Vars=4
15 [PC] Conferma consumo pacchetto Kind=DatLinear, Type=Start, Layer=1,
       BeadSeqIndex=6
16
```

Listing 8.1: Estratto del log di simulazione PLC–KUKA: Pacchetto Start per layer "normale"

Nel listato 8.1 (pacchetto Start) si riconoscono chiaramente i tre livelli della catena:

- il PC invia al PLC il pacchetto Kind=DatLinear, Type=Start, Layer=1, BeadSeqIndex=6, specificando di nuovo il numero di punti e variabili;
- il PLC conferma la ricezione del pacchetto e avvia la simulazione EtherCAT per questo pacchetto;
- il KUKA simulato:
  - riceve un elemento META con PtsExpected=2 e VarsExpected=4;

- riceve due POINT con le stesse coordinate di approccio, a quota Z=200,2 mm e Z=0,2 mm;
- riceve quattro VAR, corrispondenti a velocità di deposizione e posizione angolare della tavola;
- emette il log BUFF_1 FULL: PK=Start L=1 BeadSeq=6 Points=2/2 Vars=4/4, seguito da BUFF_1 COMPLETE, che conferma la chiusura corretta del pacchetto.

```
[PC] Sto inviando pacchetto Kind=DatLinear, Type=Loop, Layer=1, BeadSeqIndex=6
[PC] Inviato pacchetto Kind=DatLinear, Type=Loop, Layer=1, BeadSeqIndex=6
[PLC] Ricevuto pacchetto Kind=0, Type=2, Layer=1, BeadSeqIndex=6, Points=36,
    Vars=18
[PLC] Avvio simulazione EtherCAT verso KUKA per questo pacchetto...
[KUKA] META: L=1, Kind=Loop, BeadSeq=6, PtsExpected=36, VarsExpected=18
[KUKA] POINT #1: L=1, PK=Loop, BeadSeq=6, X=203.59 Y=-436.6 Z=0.2 S=2 T=10
[KUKA] POINT #2: L=1, PK=Loop, BeadSeq=6, X=124.68 Y=-465.32 Z=0.2 S=2 T=10
[... righe omesse ...]
[KUKA] POINT #35: L=1, PK=Loop, BeadSeq=6, X=340.64 Y=-340.64 Z=0.2 S=2 T=10
[KUKA] POINT #36: L=1, PK=Loop, BeadSeq=6, X=276.31 Y=-394.62 Z=0.2 S=2 T=10
[KUKA] VAR #37: L=1, PK=Loop, BeadSeq=6, Val=1.32
[KUKA] VAR #38: L=1, PK=Loop, BeadSeq=6, Val=1.32
[... righe omesse ...]
[KUKA] VAR #54: L=1, PK=Loop, BeadSeq=6, Val=1.32
[KUKA] BUFF_1 FULL: PK=Loop L=1 BeadSeq=6 Points=36/36 Vars=18/18
[KUKA] BUFF_1 COMPLETE (end of packet, counts OK): PK=Loop L=1 BeadSeq=6 Points
    =36 Vars=18
[PLC] EtherCAT completato: Punti=36, Vars=18
[PC] Conferma consumo pacchetto Kind=DatLinear, Type=Loop, Layer=1,
    BeadSeqIndex=6

```

Listing 8.2: Estratto del log di simulazione PLC–KUKA: Pacchetto Loop per layer "normale"

```
[PC] Sto inviando pacchetto Kind=DatLinear, Type=End, Layer=1, BeadSeqIndex=6
[PC] Inviato pacchetto Kind=DatLinear, Type=End, Layer=1, BeadSeqIndex=6
[PLC] Ricevuto pacchetto Kind=0, Type=3, Layer=1, BeadSeqIndex=6, Points=3,
    Vars=4
[PLC] Avvio simulazione EtherCAT verso KUKA per questo pacchetto...
[KUKA] META: L=1, Kind=End, BeadSeq=6, PtsExpected=3, VarsExpected=4
[KUKA] POINT #1: L=1, PK=End, BeadSeq=6, X=203.59 Y=-436.6 Z=0.2 S=18 T=3
[KUKA] POINT #2: L=1, PK=End, BeadSeq=6, X=195.94 Y=-440.09 Z=0.2 S=18 T=3
[KUKA] POINT #3: L=1, PK=End, BeadSeq=6, X=195.94 Y=-440.09 Z=200.2 S=2 T=10
[KUKA] VAR #4: L=1, PK=End, BeadSeq=6, Val=1.32
[KUKA] VAR #5: L=1, PK=End, BeadSeq=6, Val=481.74
[KUKA] VAR #6: L=1, PK=End, BeadSeq=6, Val=295.2
[KUKA] VAR #7: L=1, PK=End, BeadSeq=6, Val=0
[KUKA] BUFF_1 FULL: PK=End L=1 BeadSeq=6 Points=3/3 Vars=4/4
[KUKA] BUFF_1 COMPLETE (end of packet, counts OK): PK=End L=1 BeadSeq=6 Points
    =3 Vars=4
[PLC] EtherCAT completato: Punti=3, Vars=4
[PC] Conferma consumo pacchetto Kind=DatLinear, Type=End, Layer=1, BeadSeqIndex
    =6

```

Listing 8.3: Estratto del log di simulazione PLC–KUKA: Pacchetto End per layer "normale"

Log analoghi vengono prodotti per il pacchetto Loop (listato 8.2) e per il pacchetto End (listato 8.3). Nel caso del Loop, il KUKA simulato riceve:

- 36 punti distribuiti su un cerchio completo, con coordinate che ruotano di 360° intorno all'origine, mantenendo costante la quota Z=0,2 mm;
- 18 variabili di processo, tutte con lo stesso valore di riferimento, che vengono numerate da VAR #37 a VAR #54 in continuità con i punti.

Dal punto di vista grafico, le figure A.9, A.10, A.11 mostrano lo stato dell'interfaccia WPF rispettivamente per il pacchetto Start, Loop ed End di questo bead:

- nella colonna di sinistra è riportata la lista completa dei 278 pacchetti della simulazione PC–PLC–KUKA; le righe corrispondenti a Start, Loop ed End del bead 6 del layer 1 sono evidenziate quando il relativo pacchetto è in corso;
- nella tabella centrale vengono visualizzati gli StreamItem che il PLC ha effettivamente inviato verso il KUKA: si vede la sequenza Meta → Point → ... → Point → Variable → ..., con i campi Layer e BeadSeq coerenti con il contesto;
- nella tabella di destra è riportato il contenuto corrente del BUFF_1 simulato, suddiviso in:
  - tabella "Points", con l'elenco dei punti ricevuti;
  - tabella "Vars", con l'elenco delle variabili di processo ricevute;
  - header in alto del tipo PK=Loop Layer=1 BeadSeq=6 Points=36/36 Vars=18/18, che si aggiorna man mano che il pacchetto viene riempito.

Questo primo caso di test mostra che, per un bead lineare reale, la pipeline è in grado di:

- preservare i metadati: il layer, l'indice bead, il tipo pacchetto e i conteggi attesi coincidono tra report di pianificazione, log e GUI;
- conservare i dati: il numero di punti e variabili ricevuti dal KUKA simulato è esattamente pari a quello dichiarato in partenza;
- segnalare correttamente il completamento del pacchetto, tramite il messaggio "BUFF_1 FULL" e la chiusura "complete" lato KUKA e PLC.

8.4.2 Caso di test 2 – Sequenza a spirale (layer 15, sequence 1)

Il secondo caso di test riguarda una sequenza a spirale nel layer 15 del pezzo reale, associata al primo sequence index. In questo caso la geometria è più complessa e il numero totale di punti del tratto di deposito è tale da non poter essere inviato in un unico pacchetto al PLC. Il PacketPlanner applica quindi la logica di chunking del Loop, generando una tripletta Start–Loop–End in cui

- il pacchetto Start non contiene punti di traiettoria ma solo tre variabili di processo (Points=0, Vars=3), che definiscono la posizione del martello pressatore. La non presenza di punti di traiettoria è caratteristico dei pacchetto di start solo della prima sequence di ogni sezione a spirale. Il pacchetto di start della seconda sequence, contiene 2 punti con i quali si esegue un movimento CIRC senza deposizione di materiale.

- il tratto di deposito vero e proprio è diviso in tre pacchetti Loop consecutivi,i primi due contenenti 100 punti (il massimo impostato), mentre il terzo conterrà il numero di punti rimanenti; ogni Loop rappresenta un “pezzo” del loop completo della sequence;

- il pacchetto End contiene un punto di uscita e tre variabili (Points=1, Vars=3), utilizzate per spegnere l’arco e riportare il sistema in condizioni di sicurezza.

```
1  [PC] Sto inviando pacchetto Kind=Spiral, Type=Start, Layer=15, BeadSeqIndex=1
2  [PC] Inviato pacchetto Kind=Spiral, Type=Start, Layer=15, BeadSeqIndex=1
3  [PLC] Ricevuto pacchetto Kind=1, Type=1, Layer=15, BeadSeqIndex=1, Points=0,
       Vars=3
4  [PLC] Avvio simulazione EtherCAT verso KUKA per questo pacchetto...
5  [KUKA] META: L=15, Kind=Start, BeadSeq=1, PtsExpected=0, VarsExpected=3
6  [KUKA] VAR #1: L=15, PK=Start, BeadSeq=1, Val=490.59
7  [KUKA] VAR #2: L=15, PK=Start, BeadSeq=1, Val=320.62
8  [KUKA] VAR #3: L=15, PK=Start, BeadSeq=1, Val=30.54
9  [KUKA] BUFF_1 FULL: PK=Start L=15 BeadSeq=1 Points=0/0 Vars=3/3
10 [KUKA] BUFF_1 COMPLETE (end of packet, counts OK): PK=Start L=15 BeadSeq=1
       Points=0 Vars=3
11 [PLC] EtherCAT completato: Punti=0, Vars=3
12 [PC] Conferma consumo pacchetto Kind=Spiral, Type=Start, Layer=15, BeadSeqIndex
       =1
13
```

Listing 8.4: Estratto del log di simulazione PLC–KUKA: Pacchetto Start per layer a spirale

```
1  [PC] Sto inviando pacchetto Kind=Spiral, Type=Loop, Layer=15, BeadSeqIndex=1
2  [PC] Inviato pacchetto Kind=Spiral, Type=Loop, Layer=15, BeadSeqIndex=1
3  [PLC] Ricevuto pacchetto Kind=1, Type=2, Layer=15, BeadSeqIndex=1, Points=100,
       Vars=0
4  [PLC] Avvio simulazione EtherCAT verso KUKA per questo pacchetto...
5  [KUKA] META: L=15, Kind=Loop, BeadSeq=1, PtsExpected=100, VarsExpected=0
6  [KUKA] POINT #1: L=15, PK=Loop, BeadSeq=1, X=490.59 Y=0 Z=25.62 S=2 T=10
7  [KUKA] POINT #2: L=15, PK=Loop, BeadSeq=1, X=483.15 Y=-85.19 Z=25.65 S=2 T=10
8  [... righe omesse ...]
9  [KUKA] POINT #99: L=15, PK=Loop, BeadSeq=1, X=-85.53 Y=485.02 Z=28.9 S=2 T=10
10 [KUKA] POINT #100: L=15, PK=Loop, BeadSeq=1, X=0 Y=492.53 Z=28.94 S=2 T=10
11 [KUKA] BUFF_1 FULL: PK=Loop L=15 BeadSeq=1 Points=100/100 Vars=0/0
12 [KUKA] BUFF_1 COMPLETE (end of packet, counts OK): PK=Loop L=15 BeadSeq=1
       Points=100 Vars=0
13 [PLC] EtherCAT completato: Punti=100, Vars=0
14 [PC] Conferma consumo pacchetto Kind=Spiral, Type=Loop, Layer=15, BeadSeqIndex
       =1
15
```

Listing 8.5: Estratto del log di simulazione PLC–KUKA: Pacchetto Loop per layer a spirale

Nel listato 8.4 (Start spirale) si osserva che:

- il PC invia il pacchetto Kind=Spiral, Type=Start, Layer=15, BeadSeqIndex=1, con Points=0 e Vars=3;

- il PLC conferma la ricezione e avvia la simulazione EtherCAT;

- il KUKA simulato:
  - riceve un elemento Meta con PtsExpected=0 e VarsExpected=3;
  - riceve tre variabili (VAR #1, VAR #2, VAR #3) con valori che rappresentano i parametri caratteristici della spirale (raggio di partenza, velocità di rotazione, corrente nominale, ecc.);
  - logga BUFF_1 FULL con Points=0/0 e Vars=3/3, seguito da BUFF_1 COMPLETE.

Nel listato 8.5 (Loop spirale) è riportato l'estratto di log relativo al primo dei tre pacchetti Loop. Si tratta quindi di un Loop chunked, che rappresenta circa un terzo dei punti complessivi della spirale. Il comportamento che si osserva è il seguente:

- l'elemento Meta iniziale annuncia PtsExpected=100 e VarsExpected=0;
- il KUKA simulato riceve 100 POINT, ciascuno con coordinate X, Y, Z e orientazione che descrivono una spirale chiusa in pianta e crescente in quota; nel log si può vedere la progressione angolare dei punti, che percorrono ripetutamente il cerchio con raggio pressoché costante, mentre la quota Z aumenta gradualmente;
- il BUFF_1 raggiunge lo stato Points=100/100, Vars=0/0 e viene loggato BUFF_1 FULL;
- la chiusura del pacchetto viene confermata con BUFF_1 COMPLETE (end of packet, counts OK): Points=100 Vars=0, seguita dal log di completamento EtherCAT lato PLC.

Gli altri due pacchetti Loop chunked della stessa spirale mostrano un comportamento analogo: per ognuno di essi il KUKA simulato riceve esattamente i 100 punti annunciati nel Meta, il BUFF_1 viene riempito e marcato come completo, e il PLC chiude correttamente la trasmissione del chunk. La spirale completa è quindi realizzata come sequenza di tre pacchetti Loop, che il KUKA può consumare uno alla volta, mantenendo per ciascuno un BUFF_1 completo e consistente.

Dal punto di vista della GUI WPF, come visibile nella figura A.13, questo si riflette in:

- tre righe consecutive nella lista pacchetti, tutte etichettate come Kind=Spiral, Type=Loop, Layer=15, BeadSeq=1, che rappresentano i tre chunk del Loop;
- la possibilità, per l'utente, di selezionare ciascun Loop per vedere nel BUFF_1 i 100 punti relativi a quel singolo chunk e verificare che i conteggi Points=100/100, Vars=0/0 siano sempre soddisfatti.

```
1 [PC] Sto inviando pacchetto Kind=Spiral, Type=End, Layer=15, BeadSeqIndex=1
2 [PC] Inviato pacchetto Kind=Spiral, Type=End, Layer=15, BeadSeqIndex=1
3 [PLC] Ricevuto pacchetto Kind=1, Type=3, Layer=15, BeadSeqIndex=1, Points=1,
      Vars=3
4 [PLC] Avvio simulazione EtherCAT verso KUKA per questo pacchetto...
5 [KUKA] META: L=15, Kind=End, BeadSeq=1, PtsExpected=1, VarsExpected=3
6 [KUKA] POINT #1: L=15, PK=End, BeadSeq=1, X=495.56 Y=0 Z=34.06 S=2 T=10
7 [KUKA] VAR #2: L=15, PK=End, BeadSeq=1, Val=495.56
```

```
8  [KUKA] VAR #3: L=15, PK=End, BeadSeq=1, Val=329.06
9  [KUKA] VAR #4: L=15, PK=End, BeadSeq=1, Val=30.54
10 [KUKA] BUFF_1 FULL: PK=End L=15 BeadSeq=1 Points=1/1 Vars=3/3
11 [KUKA] BUFF_1 COMPLETE (end of packet, counts OK): PK=End L=15 BeadSeq=1 Points
       =1 Vars=3
12 [PLC] EtherCAT completato: Punti=1, Vars=3
13 [PC] Conferma consumo pacchetto Kind=Spiral, Type=End, Layer=15, BeadSeqIndex=1
14
```

Listing 8.6: Estratto del log di simulazione PLC–KUKA: Pacchetto End per layer a spirale

Infine, il listato 8.6 (End spirale) mostra un comportamento del tutto analogo al caso lineare:

- un Meta con PtsExpected=1, VarsExpected=3;
- un punto di spegnimento torcia;
- tre variabili di processo;
- i log “BUFF_1 FULL” e “BUFF_1 COMPLETE” con conteggi coerenti.

Anche in questo caso le figure A.12, A.13, A.14 (GUI WPF per Start, Loop, End della spirale) mostrano:

- nella lista pacchetti, la sequenza Start–3 Loop–End del layer 15, sequence 1;
- nella tabella centrale, la lunga sequenza di 100 StreamItem di tipo Point per il pacchetto Loop;
- nel BUFF_1, i 100 punti della spirale e i parametri di processo associati allo Start e all’End.

Questo secondo caso di test dimostra che la pipeline dati è in grado di gestire traiettorie complesse e altamente discretizzate (100 punti per un singolo pacchetto Loop) mantenendo:

- l’esatta corrispondenza fra conteggi attesi e conteggi ricevuti;
- la coerenza fra log e rappresentazione grafica;
- una gestione del completamento pacchetto identica a quella del caso lineare.

8.4.3 Discussione critica dei risultati

I risultati ottenuti sulla simulazione dei primi quindici layer reali del pezzo, comprendenti sia layer planari a bead lineari sia layer a spirale, mostrano in maniera abbastanza chiara che la catena: dati JSON $\rightarrow$ ProcessPacket $\rightarrow$ StProcessPacket $\rightarrow$ StreamItem $\rightarrow$ PlcToKukaOutputs $\rightarrow$ BUFF_1 si comporta in modo coerente e conservativo. Nei due casi di test analizzati nel dettaglio (bead lineare del layer 1, bead 6, e spirale del layer 15, sequence 1, con Loop spezzato in tre chunk) non è mai emerso uno scostamento tra il numero di punti e variabili dichiarato nei report di pianificazione, il contenuto effettivamente trasmesso in EtherCAT e ciò che il KUKA simulato ricostruisce all’interno del BUFF_1. Questo vale sia per pacchetti “piccoli” (Start/End con pochi punti e alcune variabili di processo) sia per pacchetti significativamente più pesanti (Loop lineari

con 36 punti e 18 variabili, Loop spiral chunked da 100 punti ciascuno).

Dal punto di vista del protocollo, la simulazione conferma la correttezza delle due interfacce logiche principali. Sul fronte PC→PLC, la struttura StProcessPacket si dimostra sufficiente a rappresentare in modo compatto ma completo tutti i metadati rilevanti (tipo pacchetto, layer, indice bead/sequence, conteggi di punti e variabili, eventuali indici di chunk), senza introdurre ambiguità sulla ricostruzione del contesto. Sul fronte PLC→KUKA, il modello basato su StreamItem (Meta, Point, Variable) e sulle strutture PlcToKukaOutputs / KukaToPlcInputs permette di implementare un flusso di dati a granularità fine, con un solo elemento trasmesso per ciclo e un handshake esplicito data/ack. Il comportamento osservato nei log, sequenze ordinate Meta→Point/Variable, messaggi “BUFF_1 FULL” e “BUFF_1 COMPLETE” emessi una sola volta per pacchetto, chiusura corretta anche in presenza di chunking, è esattamente quello che si desidera ottenere in vista dell’implementazione su PLC Beckhoff e KRC5 reali.

Naturalmente la simulazione presenta anche dei limiti. Il modello di comunicazione adottato è idealizzato: non vengono considerati ritardi di rete o errori di trasmissione che possono comparire su una rete EtherCAT reale. Inoltre, la simulazione si concentra esclusivamente sul flusso dati e sulla gestione dei buffer: la dinamica del robot, i limiti di velocità e accelerazione, l’interazione con il processo WAAM (formazione del cordone, stabilità dell’arco, ecc.) richiederanno una validazione separata. Infine, il mapping effettivo delle strutture PlcToKukaOutputs e KukaToPlcInputs sui PDO di un progetto TwinCAT e di un progetto KRC5 potrà introdurre ulteriori vincoli pratici che qui non sono stati simulati.

Nonostante questi limiti, la simulazione realizzata può essere considerata a tutti gli effetti una specifica eseguibile dell’architettura di comunicazione PC–PLC–KUKA. Le stesse strutture dati, lo stesso protocollo di handshake e la stessa logica di assemblaggio dei pacchetti utilizzate nel simulatore possono essere riutilizzate quasi direttamente nella fase di messa in servizio sull’impianto reale, riducendo in modo significativo il rischio di errore e il tempo necessario per il debug in campo. In altri termini, il lavoro svolto in questo capitolo non si esaurisce in un esercizio accademico, ma costituisce un passo concreto verso l’implementazione industriale della soluzione proposta.

8.5 Valutazione del ruolo del PLC nella catena PC–PLC–KUKA

In questo capitolo è stata presentata un’architettura di streaming in cui il PLC Beckhoff svolge il ruolo di nodo intermedio tra il PC di pianificazione e il controllore KUKA, fungendo da master EtherCAT verso il robot e da “immagine di processo” per la gestione dei buffer. Una domanda naturale, soprattutto in un’ottica di futura industrializzazione, è se tale passaggio intermedio sia realmente necessario oppure se sia possibile semplificare ulteriormente la catena di controllo eliminando il PLC e utilizzando direttamente il PC come master EtherCAT verso KUKA. Questa possibilità è resa teoricamente percorribile dal fatto che, grazie all’uso dei buffer nel controllore KUKA (gestione a pacchetti Start/Loop/End e logica di riempimento/consumo dei buffer), i requisiti di hard real-time sul canale di comunicazione risultano meno stringenti: il robot dispone in memoria di un certo orizzonte di punti e variabili di processo, ed è

quindi in grado di proseguire l'esecuzione anche in presenza di jitter moderato nel canale di streaming.

Dal punto di vista concettuale, l'eliminazione del PLC comporterebbe alcuni vantaggi evidenti. In primo luogo, l'architettura risulterebbe più semplice: un nodo in meno implica meno configurazioni, meno possibili punti di guasto, meno casi di errore da gestire (ad esempio, nessun disallineamento tra immagine di processo nel PLC e buffer effettivi nel KUKA). L'intera pipeline, dal file JSON generato a partire dal CAD/Grasshopper, fino alla traduzione in pacchetti EtherCAT, verrebbe concentrata in un unico ambiente software sul PC. Questo semplificherebbe anche le attività di manutenzione e evoluzione del sistema: le modifiche future alla struttura dei pacchetti, alle strategie di recovery o alle logiche di pianificazione potrebbero essere implementate agendo su un solo layer applicativo, senza la necessità di mantenere sincronizzati codice PC e PLC. In prospettiva di laboratorio o di prototipo, un'architettura PC–KUKA "diretta" potrebbe quindi risultare più flessibile e veloce da evolvere.

Tuttavia, questa semplificazione architetturale presenta una serie di controindicazioni non trascurabili quando si guarda a un impianto WAAM con requisiti industriali. Il primo elemento riguarda la safety: il PLC Beckhoff, nella soluzione proposta, non si limita a inoltrare pacchetti verso il KUKA, ma rappresenta il punto naturale in cui integrare logiche di sicurezza, gestione degli ingressi/uscite di campo, interblocco con altri dispositivi della cella, gestione degli arresti di emergenza e dei consensi all'avvio processo. Un PC general-purpose, privo di certificazioni e I/O industriali nativi, non è normalmente considerato un dispositivo idoneo per implementare logiche di sicurezza funzionale e richiederebbe, in ogni caso, hardware e software aggiuntivi per avvicinarsi a questo ruolo.

Un secondo aspetto critico riguarda il determinismo e i tempi ciclo. È vero che, grazie ai buffer nel controllore KUKA, il sistema non richiede necessariamente tempi di ciclo dell'ordine del millisecondo; tuttavia il PLC garantisce una base di temporizzazione deterministica e prevedibile per il rilascio dei pacchetti verso il robot e per la lettura degli acknowledge di ritorno. Un PC con sistema operativo general-purpose (ad esempio Windows) non è progettato per fornire la stessa stabilità nei tempi di scheduling dei task e nei tempi di risposta a eventi di campo, soprattutto in presenza di altri processi in esecuzione, aggiornamenti o carichi imprevisti. Per utilizzare il PC come master EtherCAT in modo affidabile sarebbe necessario ricorrere a soluzioni di controllo PC-based con estensioni real-time dedicate, con complessità non trascurabile e, di fatto, riportando il sistema verso una logica "tipo PLC" implementata su hardware PC.

Vi è poi una considerazione di robustezza e industrialità del dispositivo. Un PLC è progettato per funzionare in continuo, in ambienti industriali, con temperature e vibrazioni non ideali, con cicli di avvio/arresto ben controllati e con un comportamento prevedibile in caso di guasti elettrici o di comunicazione. Un PC standard da ufficio, pur essendo più flessibile sul piano software, non offre le stesse garanzie di robustezza, né in termini di hardware (alimentazioni, EMC, ecc.) né in termini di gestione degli stati di avvio, crash e recovery dopo power-off improvvisi. Delegare direttamente a questo dispositivo il ruolo di master EtherCAT significherebbe spostare il punto critico dell'impianto su un componente meno adatto alla continuità operativa tipica di un

processo WAAM industriale.

Infine, occorre considerare l'integrazione con il resto dell'impianto. Nella configurazione PC–PLC–KUKA il PLC può facilmente fungere da hub di coordinamento per altri sottosistemi: posizionatori aggiuntivi, sistemi di movimentazione pezzi, sensori di processo (corrente, tensione, temperatura), periferiche di sicurezza, HMI di linea. L'integrazione con questi elementi avviene tipicamente tramite protocolli industriali standard già supportati dal PLC (EtherCAT, Profinet, Modbus, ecc.), mentre in un'architettura PC–KUKA diretta sarebbe necessario implementare e manutenere driver e interfacce custom per ciascun sottosistema, con un aumento significativo della complessità complessiva.

In sintesi, la rimozione del PLC e l'adozione di un PC come master EtherCAT rappresentano una semplificazione interessante dal punto di vista della prototipazione e della flessibilità software, resa possibile dallo scarico dei vincoli di real-time sui buffer del controllore KUKA. Tuttavia, in un'ottica di impianto reale, i vantaggi di sicurezza, determinismo temporale, robustezza hardware e integrazione con il resto della cella offerti dal PLC giustificano il mantenimento di questo nodo intermedio nella catena di controllo. L'architettura proposta in questa tesi può quindi essere letta come un compromesso: il PC resta il luogo privilegiato per la pianificazione, la generazione del file JSON e la gestione di alto livello dei pacchetti, mentre il PLC assicura la mediazione industriale verso il KUKA e verso l'impianto, preservando al tempo stesso la possibilità, in scenari futuri specifici, di esplorare configurazioni PC-based più spinte.

Conclusioni

Il presente lavoro di tesi ha affrontato la sfida di evolvere l'ecosistema software di una cella robotizzata per il Wire Arc Additive Manufacturing (WAAM), con l'obiettivo di superare i limiti imposti dall'approccio tradizionale basato su file statici e monolitici. Partendo dall'analisi dell'impianto esistente presso l'azienda Loccioni e delle criticità intrinseche al processo WAAM, quali la gestione termica, la complessità delle traiettorie e la necessità di interventi di ripristino (recovery), la ricerca ha seguito un percorso evolutivo che si è articolato in due fasi principali: l'ottimizzazione del processo esistente e la progettazione di una nuova architettura di controllo orientata allo streaming dei dati.

Ottimizzazione del workflow KRL e logiche di Recovery

In una prima fase, l'attività si è concentrata sulla ristrutturazione del codice KRL. Attraverso lo sviluppo di algoritmi di parsing personalizzati (implementati in Python), è stato possibile trasformare la rappresentazione lineare e voluminosa dei programmi robot in una logica strutturata e indicizzata. Questo intervento ha prodotto risultati tangibili:

- Una drastica riduzione delle dimensioni dei file caricati nel controller;
- L'implementazione di procedure di recovery automatico, che permettono di interrompere e riprendere il processo in modo deterministico, risolvendo una delle problematiche in ambito industriale;
- L'unificazione della gestione di geometrie eterogenee (layer planari e sezioni a spirale) in un unico modello dati JSON, garantendo coerenza tra la pianificazione CAD e l'esecuzione fisica.

Architettura di Streaming PC-PLC-KUKA

Consapevoli che l'ottimizzazione del codice KRL non potesse comunque superare i vincoli di memoria e flessibilità del controller robotico, la seconda parte della tesi ha proposto un cambio di paradigma architetturale. È stato progettato un sistema distribuito basato su streaming, in cui il PC assume il ruolo di pianificatore dinamico, il PLC quello di buffer e gestore del real-time, e il robot quello di esecutore finale. Questa architettura sposta la complessità "fuori" dal robot, permettendo di gestire manufatti di dimensioni arbitrarie senza rigenerare interi programmi per ogni singola modifica.

Validazione tramite simulazione software

Non potendo procedere immediatamente all'implementazione hardware completa, l'architettura proposta è stata validata attraverso lo sviluppo di un ambiente di simulazione software dedicato (Capitoli 7 e 8). Le simulazioni hanno dimostrato che:

1. La pipeline di comunicazione PC→PLC è in grado di serializzare correttamente il file JSON in pacchetti, gestendo l'handshake e i limiti dimensionali del protocollo ADS.

2. La catena PLC→KUKA (simulata su logica EtherCAT) garantisce che il flusso di dati arrivi al buffer del robot senza perdita di informazioni, mantenendo la sincronizzazione necessaria per il processo di deposizione.

Sviluppi futuri

In conclusione, il lavoro svolto costituisce una piattaforma abilitante per l'evoluzione futura della cella WAAM. Aver validato la logica di streaming e recovery in ambiente simulato riduce significativamente i rischi della futura implementazione fisica. I prossimi passi naturali della ricerca riguarderanno:

- Il deployment del software di streaming sull'hardware reale (PLC Beckhoff e Robot KUKA).

- L'integrazione di sensori di processo (temperatura, visione) nel loop di controllo: grazie alla nuova architettura, i dati sensoriali potranno influenzare la generazione dei pacchetti in tempo reale, aprendo la strada a strategie di controllo adattivo (closed-loop control) oggi impossibili con l'approccio statico.

In definitiva, la tesi dimostra che per industrializzare il processo WAAM non è sufficiente controllare la torcia o il movimento, ma è necessario ripensare il flusso del dato, trasformando il robot da semplice esecutore di traiettorie pre-calcolate a nodo intelligente di un sistema di produzione digitale interconnesso.

Appendici

Appendice A

Immagini

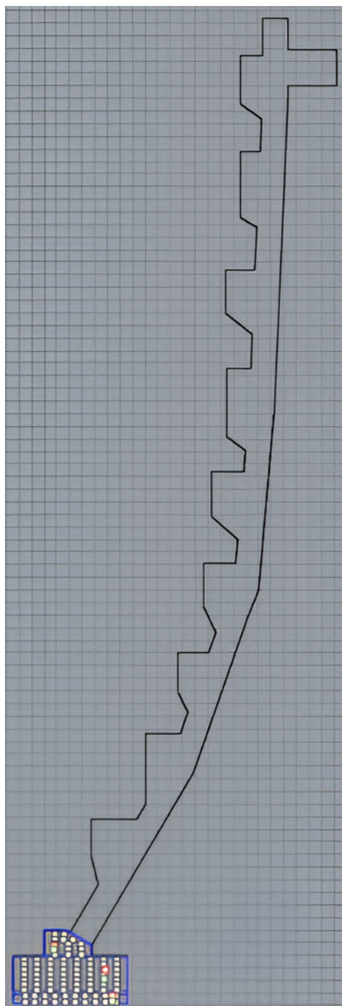

Figura A.1: Sezione del manufatto in Rhino

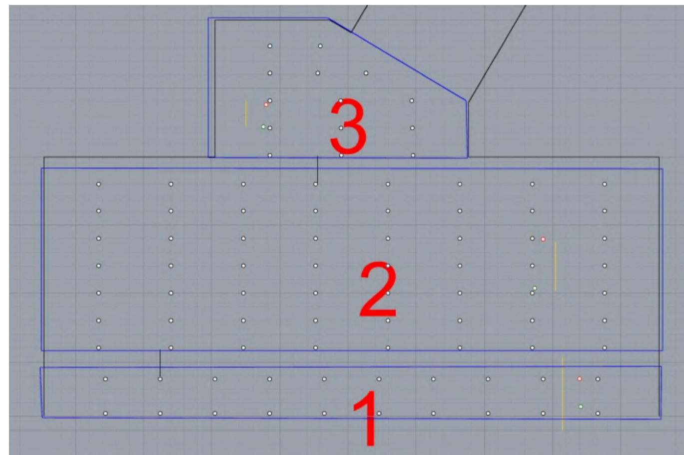

Figura A.2: Ingrandimento area in slicing

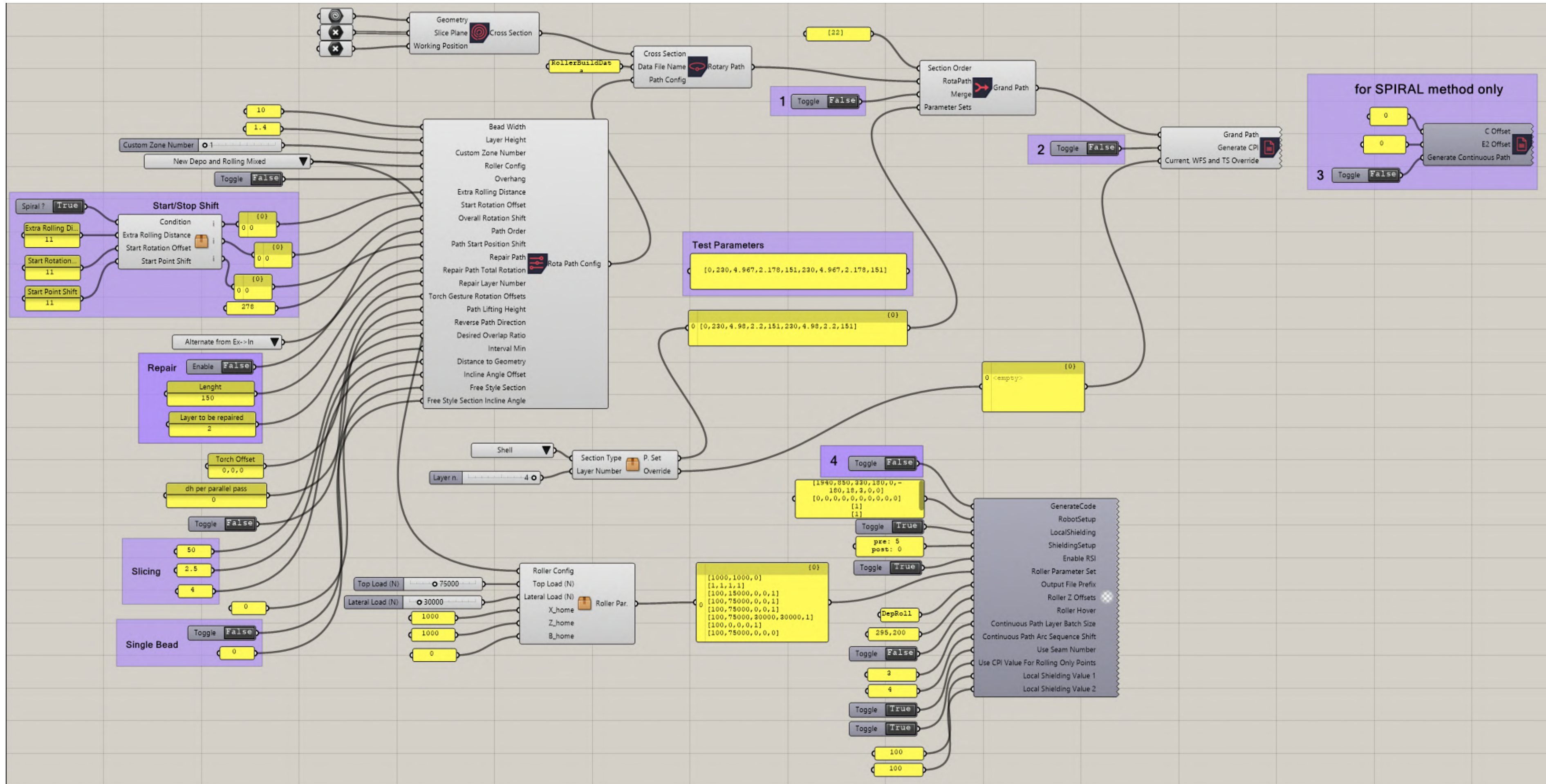

Figura A.3: Schema completo del programma Grasshopper

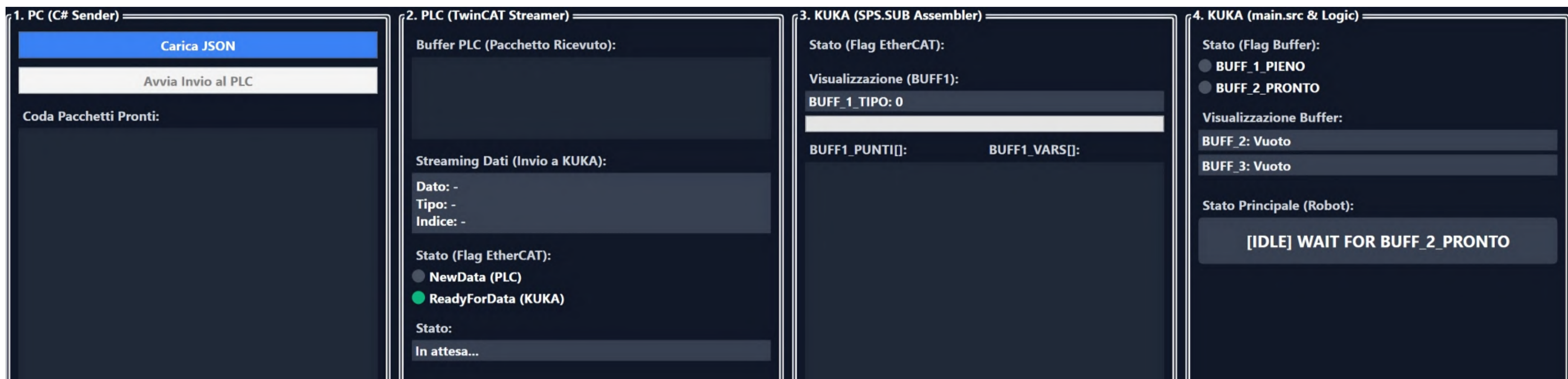

Figura A.4: Applicazione per la simulazione

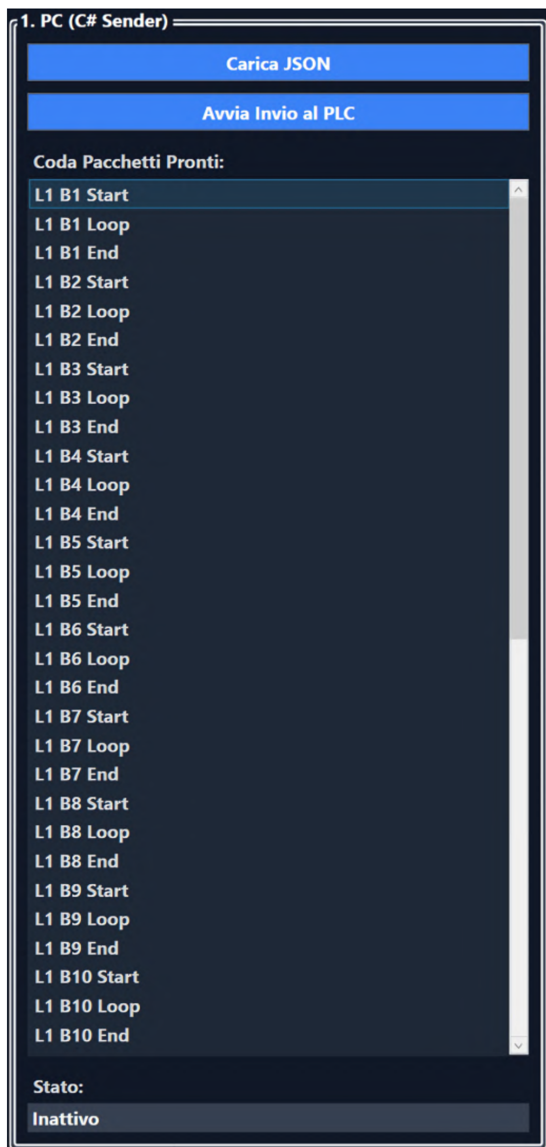

Figura A.5: Task 0 durante la simulazione

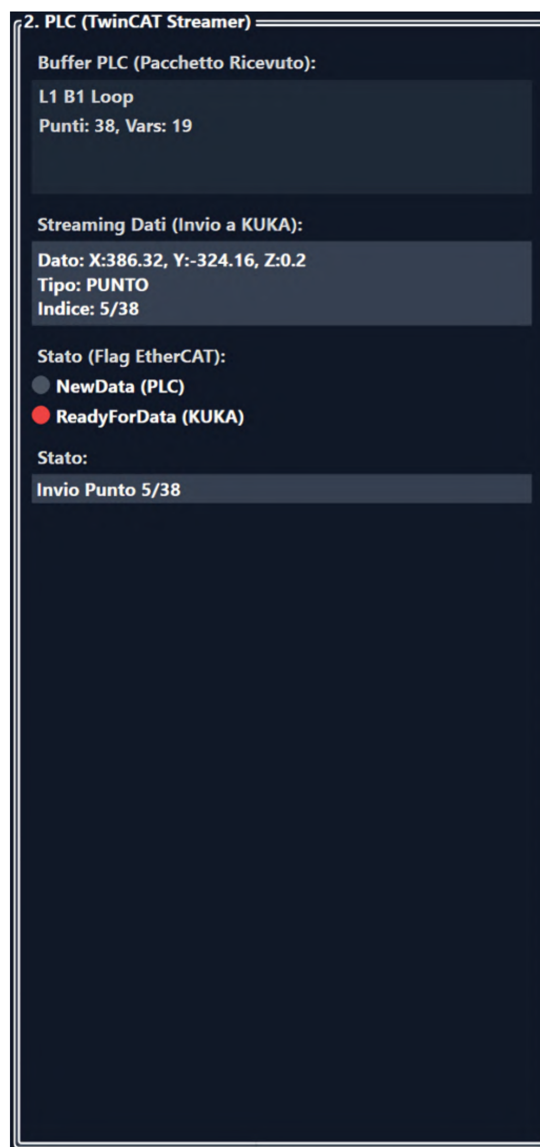

Figura A.6: Task 1 durante la simulazione

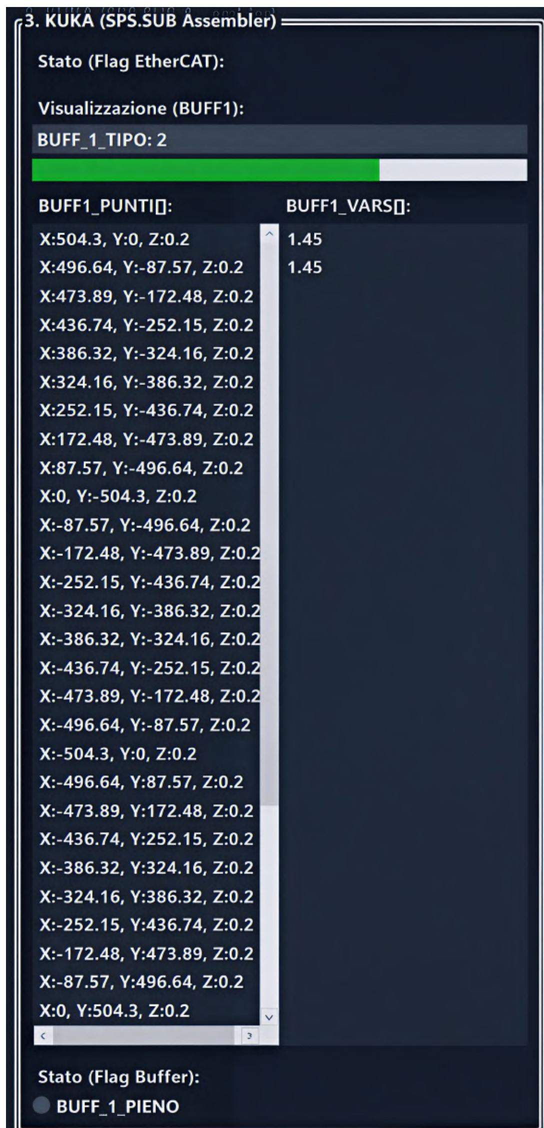

Figura A.7: Task 2 & 3 in simulazione

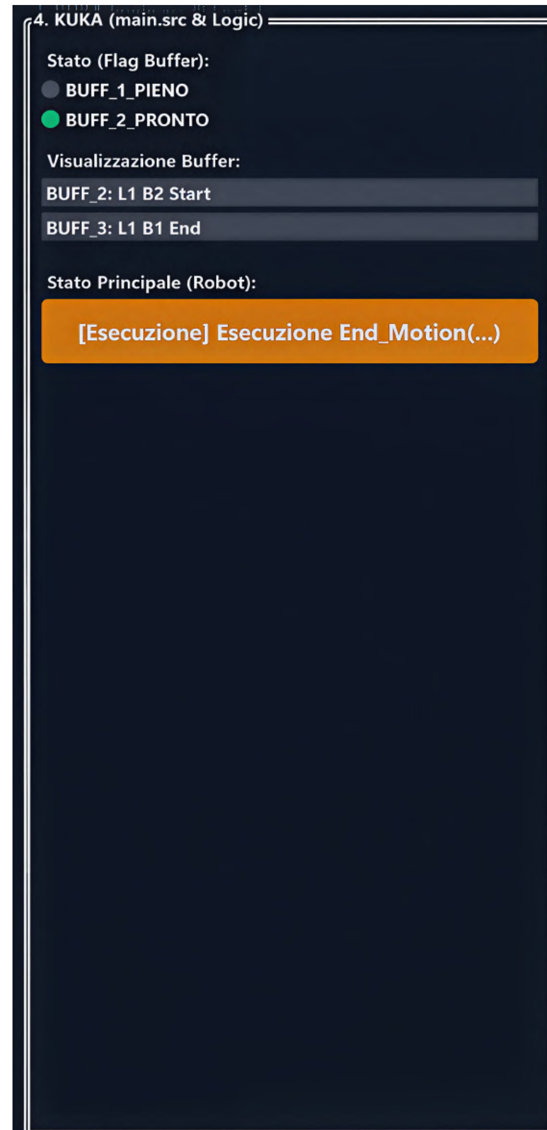

Figura A.8: Task 4 in simulazione

WAAM PLC-KUKA Simulation

Carica JSON progetto | Avvia simulazione | Stop | Delay (ms): 500

Simulando pacchetto 1/278 (Kind=DatLinear, Type=Start, Layer=1, BeadSeq=1)

Pacchetti PC → PLC → KUKA

#	Kind	Type	Layer	Bead/Seq	Pts	Vars
1	DatLinear	Start	1	1	2	4
2	DatLinear	Loop	1	1	36	18
3	DatLinear	End	1	1	3	4
4	DatLinear	Start	1	2	2	4
5	DatLinear	Loop	1	2	36	18
6	DatLinear	End	1	2	3	4
7	DatLinear	Start	1	3	2	4
8	DatLinear	Loop	1	3	36	18
9	DatLinear	End	1	3	3	4
10	DatLinear	Start	1	4	2	4
11	DatLinear	Loop	1	4	36	18
12	DatLinear	End	1	4	3	4
13	DatLinear	Start	1	5	2	4
14	DatLinear	Loop	1	5	36	18
15	DatLinear	End	1	5	3	4
16	DatLinear	Start	1	6	2	4
17	DatLinear	Loop	1	6	36	18
18	DatLinear	End	1	6	3	4
19	DatLinear	Start	1	7	2	4
20	DatLinear	Loop	1	7	36	18
21	DatLinear	End	1	7	3	4
22	DatLinear	Start	1	8	2	4
23	DatLinear	Loop	1	8	36	18
24	DatLinear	End	1	8	3	4
25	DatLinear	Start	1	9	2	4
26	DatLinear	Loop	1	9	36	18
27	DatLinear	End	1	9	3	4
28	DatLinear	Start	2	1	2	4
29	DatLinear	Loop	2	1	36	18
30	DatLinear	End	2	1	3	4
31	DatLinear	Start	2	2	2	4
32	DatLinear	Loop	2	2	36	18
33	DatLinear	End	2	2	3	4
34	DatLinear	Start	2	3	2	4
35	DatLinear	Loop	2	3	36	18
36	DatLinear	End	2	3	3	4
37	DatLinear	Start	2	4	2	4
38	DatLinear	Loop	2	4	36	18
39	DatLinear	End	2	4	3	4
40	DatLinear	Start	2	5	2	4
41	DatLinear	Loop	2	5	36	18
42	DatLinear	End	2	5	3	4
43	DatLinear	Start	2	6	2	4
44	DatLinear	Loop	2	6	36	18
45	DatLinear	End	2	6	3	4
46	DatLinear	Start	2	7	2	4

PLC virtuale – stream verso EtherCAT

#	Payload	Kind	Layer	BeadSeq	X	Y	Z	Var
0	Meta	Start	1	1	0.0	0.0	0.0	0.00
1	Point	Start	1	1	504.3	0.0	200.2	0.00
2	Point	Start	1	1	504.3	0.0	0.2	0.00
3	Variable	Start	1	1	0.0	0.0	0.0	0.00
4	Variable	Start	1	1	0.0	0.0	0.0	504.30
5	Variable	Start	1	1	0.0	0.0	0.0	295.20

KUKA – BUFF1 (pacchetto corrente)

PK=Start Layer=1 BeadSeq=1 Points=2/2 Vars=3/4

Points

#	X	Y	Z	A	B	C	E1	S
1	504.3	0.0	200.2	-90.0	0.0	180.0	0.0	2.0
2	504.3	0.0	0.2	-450.0	0.0	180.0	0.0	2.0

Vars

#	Value
3	0.00
4	504.30
5	295.20

Figura A.9: Interfaccia grafica per pacchetto di Start layer "normale"

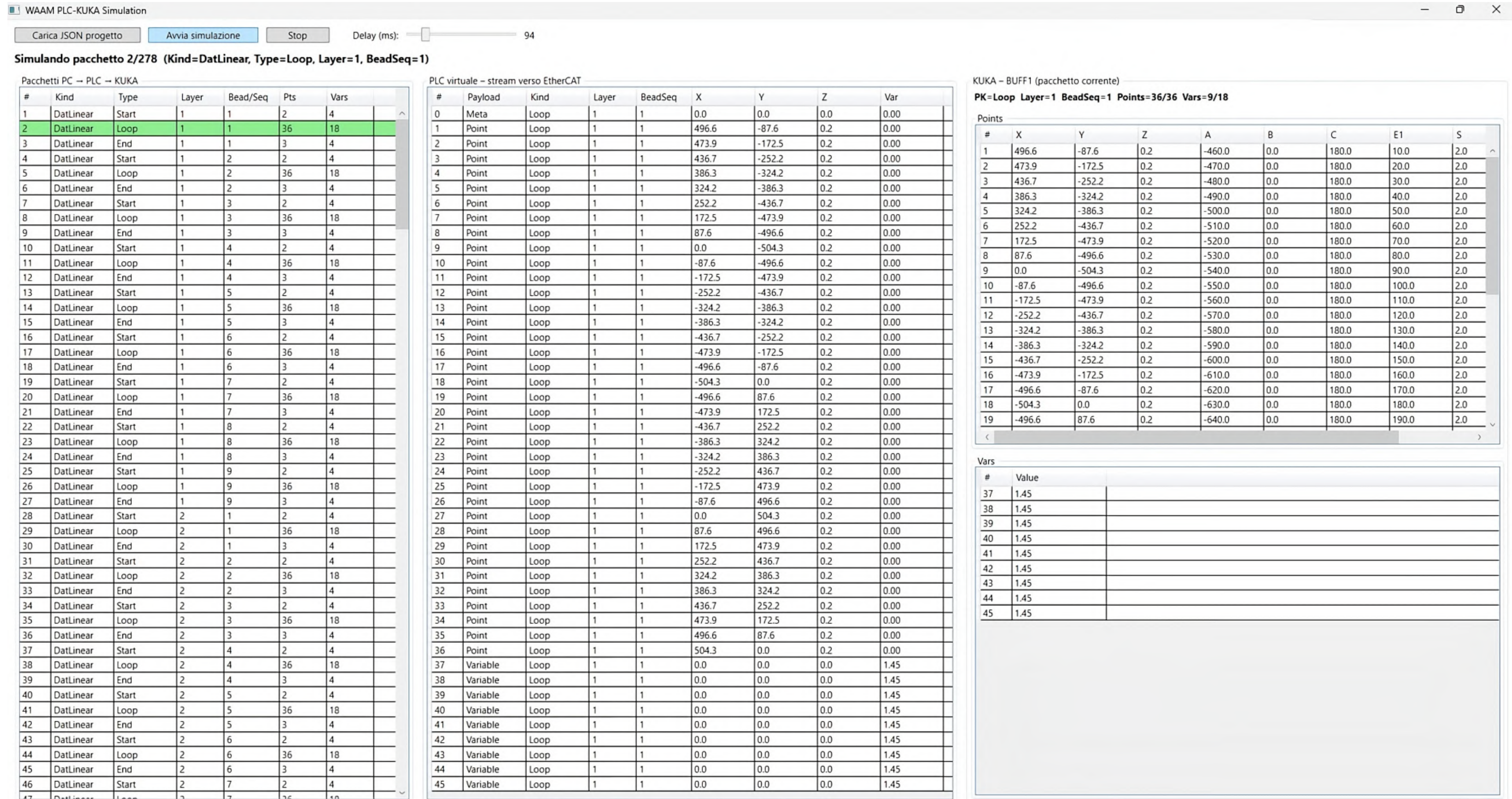

Figura A.10: Interfaccia grafica per pacchetto di Loop layer "normale"

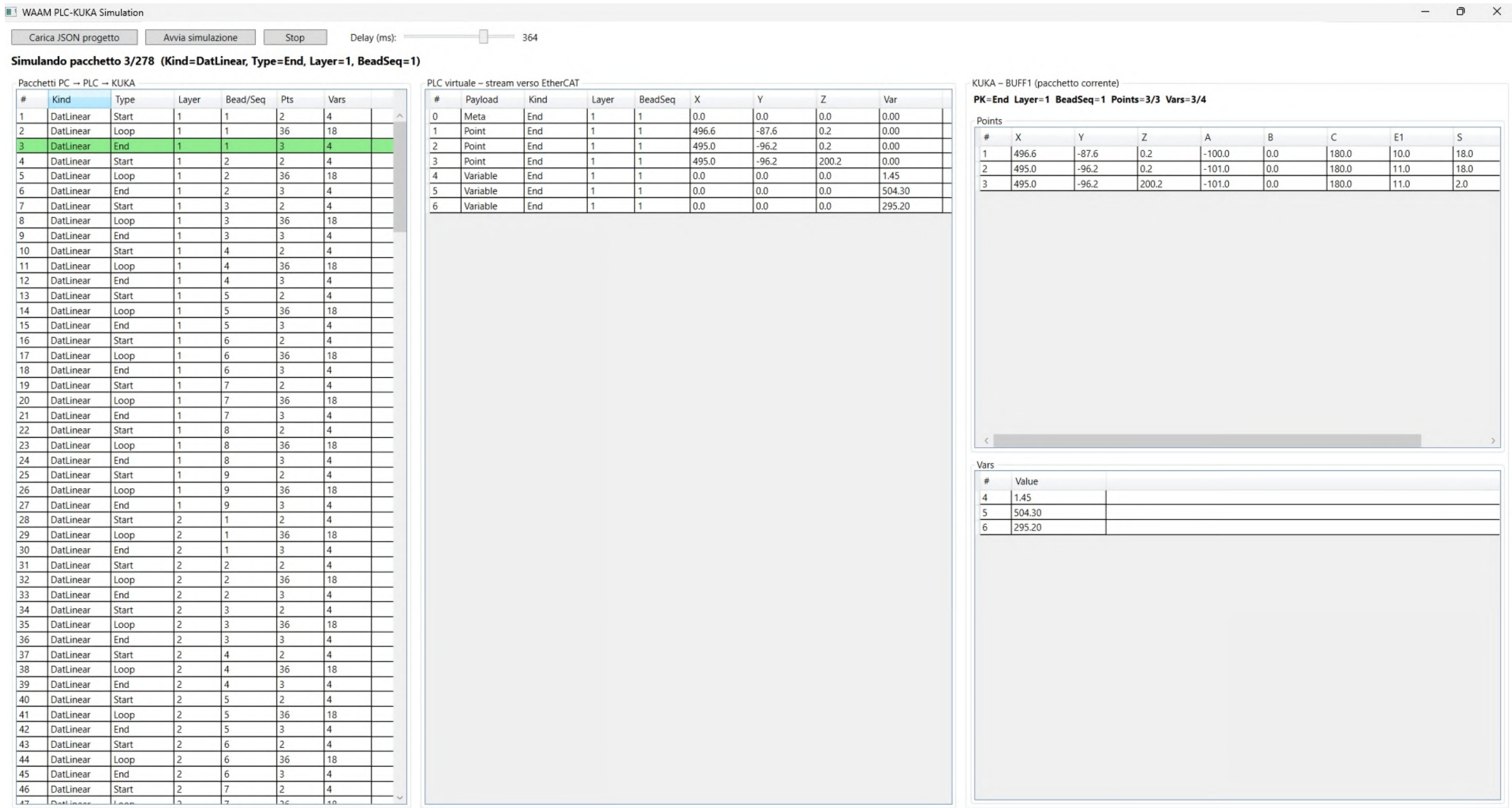

Figura A.11: Interfaccia grafica per pacchetto di End layer "normale"

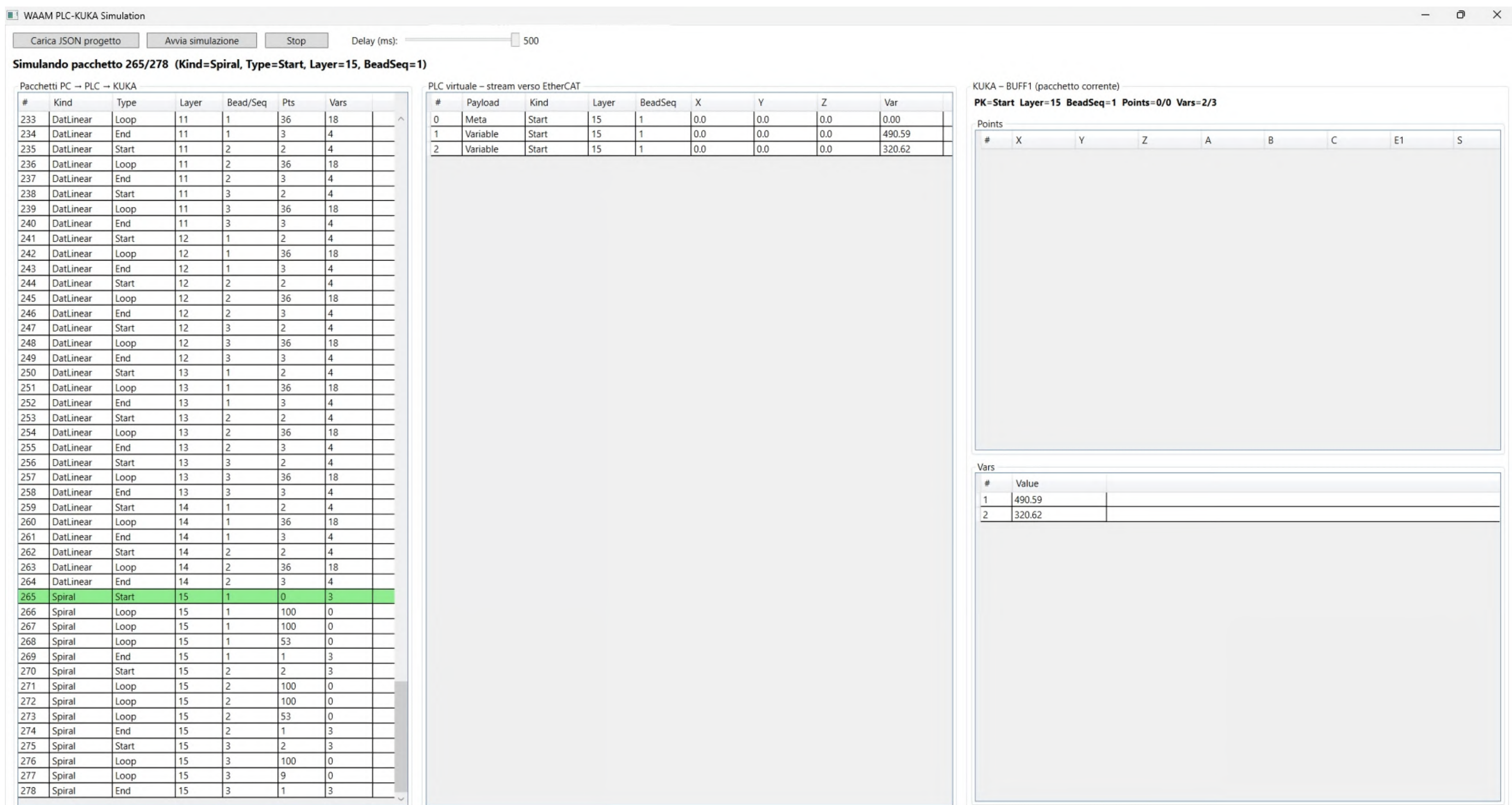

Figura A.12: Interfaccia grafica per pacchetto di Start di un layer a spirale

WAAM PLC-KUKA Simulation

Carica JSON progetto | Avvia simulazione | Stop | Delay (ms): 170

Simulando pacchetto 266/278 (Kind=Spiral, Type=Loop, Layer=15, BeadSeq=1)

Pacchetti PC → PLC → KUKA

#	Kind	Type	Layer	Bead/Seq	Pts	Vars
233	DatLinear	Loop	11	1	36	18
234	DatLinear	End	11	1	3	4
235	DatLinear	Start	11	2	2	4
236	DatLinear	Loop	11	2	36	18
237	DatLinear	End	11	2	3	4
238	DatLinear	Start	11	3	2	4
239	DatLinear	Loop	11	3	36	18
240	DatLinear	End	11	3	3	4
241	DatLinear	Start	12	1	2	4
242	DatLinear	Loop	12	1	36	18
243	DatLinear	End	12	1	3	4
244	DatLinear	Start	12	2	2	4
245	DatLinear	Loop	12	2	36	18
246	DatLinear	End	12	2	3	4
247	DatLinear	Start	12	3	2	4
248	DatLinear	Loop	12	3	36	18
249	DatLinear	End	12	3	3	4
250	DatLinear	Start	13	1	2	4
251	DatLinear	Loop	13	1	36	18
252	DatLinear	End	13	1	3	4
253	DatLinear	Start	13	2	2	4
254	DatLinear	Loop	13	2	36	18
255	DatLinear	End	13	2	3	4
256	DatLinear	Start	13	3	2	4
257	DatLinear	Loop	13	3	36	18
258	DatLinear	End	13	3	3	4
259	DatLinear	Start	14	1	2	4
260	DatLinear	Loop	14	1	36	18
261	DatLinear	End	14	1	3	4
262	DatLinear	Start	14	2	2	4
263	DatLinear	Loop	14	2	36	18
264	DatLinear	End	14	2	3	4
265	Spiral	Start	15	1	0	3
266	Spiral	Loop	15	1	100	0
267	Spiral	Loop	15	1	100	0
268	Spiral	Loop	15	1	53	0
269	Spiral	End	15	1	1	3
270	Spiral	Start	15	2	2	3
271	Spiral	Loop	15	2	100	0
272	Spiral	Loop	15	2	100	0
273	Spiral	Loop	15	2	53	0
274	Spiral	End	15	2	1	3
275	Spiral	Start	15	3	2	3
276	Spiral	Loop	15	3	100	0
277	Spiral	Loop	15	3	9	0
278	Spiral	End	15	3	1	3

PLC virtuale – stream verso EtherCAT

#	Payload	Kind	Layer	BeadSeq	X	Y	Z	Var
0	Meta	Loop	15	1	0.0	0.0	0.0	0.00
1	Point	Loop	15	1	490.6	0.0	25.6	0.00
2	Point	Loop	15	1	483.1	-85.2	25.6	0.00
3	Point	Loop	15	1	461.0	-167.8	25.7	0.00
4	Point	Loop	15	1	424.9	-245.3	25.7	0.00
5	Point	Loop	15	1	375.9	-315.4	25.8	0.00
6	Point	Loop	15	1	315.4	-375.9	25.8	0.00
7	Point	Loop	15	1	245.3	-425.0	25.8	0.00
8	Point	Loop	15	1	167.8	-461.1	25.9	0.00
9	Point	Loop	15	1	85.2	-483.3	25.9	0.00
10	Point	Loop	15	1	0.0	-490.8	25.9	0.00
11	Point	Loop	15	1	-85.2	-483.3	25.9	0.00
12	Point	Loop	15	1	-167.9	-461.2	26.0	0.00
13	Point	Loop	15	1	-245.4	-425.1	26.0	0.00
14	Point	Loop	15	1	-315.5	-376.0	26.1	0.00
15	Point	Loop	15	1	-376.0	-315.5	26.1	0.00
16	Point	Loop	15	1	-425.1	-245.4	26.1	0.00
17	Point	Loop	15	1	-461.3	-167.9	26.1	0.00
18	Point	Loop	15	1	-483.4	-85.2	26.2	0.00
19	Point	Loop	15	1	-490.9	0.0	26.2	0.00
20	Point	Loop	15	1	-483.5	85.2	26.2	0.00
21	Point	Loop	15	1	-461.4	167.9	26.3	0.00
22	Point	Loop	15	1	-425.2	245.5	26.3	0.00
23	Point	Loop	15	1	-376.1	315.6	26.4	0.00
24	Point	Loop	15	1	-315.6	376.1	26.4	0.00
25	Point	Loop	15	1	-245.5	425.3	26.4	0.00
26	Point	Loop	15	1	-167.9	461.4	26.4	0.00
27	Point	Loop	15	1	-85.3	483.6	26.5	0.00
28	Point	Loop	15	1	0.0	491.1	26.5	0.00
29	Point	Loop	15	1	85.3	483.7	26.6	0.00
30	Point	Loop	15	1	168.0	461.5	26.6	0.00
31	Point	Loop	15	1	245.6	425.4	26.6	0.00
32	Point	Loop	15	1	315.7	376.3	26.6	0.00
33	Point	Loop	15	1	376.3	315.7	26.7	0.00
34	Point	Loop	15	1	425.4	245.6	26.7	0.00
35	Point	Loop	15	1	461.6	168.0	26.8	0.00
36	Point	Loop	15	1	483.8	85.3	26.8	0.00
37	Point	Loop	15	1	491.3	0.0	26.8	0.00
38	Point	Loop	15	1	483.9	-85.3	26.9	0.00
39	Point	Loop	15	1	461.7	-168.0	26.9	0.00
40	Point	Loop	15	1	425.5	-245.7	26.9	0.00
41	Point	Loop	15	1	376.4	-315.9	26.9	0.00
42	Point	Loop	15	1	315.9	-376.4	27.0	0.00
43	Point	Loop	15	1	245.7	-425.6	27.0	0.00
44	Point	Loop	15	1	168.1	-461.8	27.1	0.00
45	Point	Loop	15	1	85.3	-484.0	27.1	0.00

KUKA – BUFF1 (pacchetto corrente)

PK=Loop Layer=15 BeadSeq=1 Points=63/100 Vars=0/0

Points

#	X	Y	Z	A	B	C	E1	S
1	490.6	0.0	25.6	-450.0	0.0	149.5	0.0	2.0
2	483.1	-85.2	25.6	-460.0	0.0	149.5	10.0	2.0
3	461.0	-167.8	25.7	-470.0	0.0	149.5	20.0	2.0
4	424.9	-245.3	25.7	-480.0	0.0	149.5	30.0	2.0
5	375.9	-315.4	25.8	-490.0	0.0	149.5	40.0	2.0
6	315.4	-375.9	25.8	-500.0	0.0	149.5	50.0	2.0
7	245.3	-425.0	25.8	-510.0	0.0	149.5	60.0	2.0
8	167.8	-461.1	25.9	-520.0	0.0	149.5	70.0	2.0
9	85.2	-483.3	25.9	-530.0	0.0	149.5	80.0	2.0
10	0.0	-490.8	25.9	-540.0	0.0	149.5	90.0	2.0
11	-85.2	-483.3	25.9	-550.0	0.0	149.5	100.0	2.0
12	-167.9	-461.2	26.0	-560.0	0.0	149.5	110.0	2.0
13	-245.4	-425.1	26.0	-570.0	0.0	149.5	120.0	2.0
14	-315.5	-376.0	26.1	-580.0	0.0	149.5	130.0	2.0
15	-376.0	-315.5	26.1	-590.0	0.0	149.5	140.0	2.0
16	-425.1	-245.4	26.1	-600.0	0.0	149.5	150.0	2.0
17	-461.3	-167.9	26.1	-610.0	0.0	149.5	160.0	2.0
18	-483.4	-85.2	26.2	-620.0	0.0	149.5	170.0	2.0
19	-490.9	0.0	26.2	-630.0	0.0	149.5	180.0	2.0

Vars

#	Value

Figura A.13: Interfaccia grafica per pacchetto di Loop di un layer a spirale

WAAM PLC-KUKA Simulation

Carica JSON progetto | Avvia simulazione | Stop | Delay (ms): 500

Simulando pacchetto 269/278 (Kind=Spiral, Type=End, Layer=15, BeadSeq=1)

Pacchetti PC → PLC → KUKA

#	Kind	Type	Layer	Bead/Seq	Pts	Vars
233	DatLinear	Loop	11	1	36	18
234	DatLinear	End	11	1	3	4
235	DatLinear	Start	11	2	2	4
236	DatLinear	Loop	11	2	36	18
237	DatLinear	End	11	2	3	4
238	DatLinear	Start	11	3	2	4
239	DatLinear	Loop	11	3	36	18
240	DatLinear	End	11	3	3	4
241	DatLinear	Start	12	1	2	4
242	DatLinear	Loop	12	1	36	18
243	DatLinear	End	12	1	3	4
244	DatLinear	Start	12	2	2	4
245	DatLinear	Loop	12	2	36	18
246	DatLinear	End	12	2	3	4
247	DatLinear	Start	12	3	2	4
248	DatLinear	Loop	12	3	36	18
249	DatLinear	End	12	3	3	4
250	DatLinear	Start	13	1	2	4
251	DatLinear	Loop	13	1	36	18
252	DatLinear	End	13	1	3	4
253	DatLinear	Start	13	2	2	4
254	DatLinear	Loop	13	2	36	18
255	DatLinear	End	13	2	3	4
256	DatLinear	Start	13	3	2	4
257	DatLinear	Loop	13	3	36	18
258	DatLinear	End	13	3	3	4
259	DatLinear	Start	14	1	2	4
260	DatLinear	Loop	14	1	36	18
261	DatLinear	End	14	1	3	4
262	DatLinear	Start	14	2	2	4
263	DatLinear	Loop	14	2	36	18
264	DatLinear	End	14	2	3	4
265	Spiral	Start	15	1	0	3
266	Spiral	Loop	15	1	100	0
267	Spiral	Loop	15	1	100	0
268	Spiral	Loop	15	1	53	0
269	Spiral	End	15	1	1	3
270	Spiral	Start	15	2	2	3
271	Spiral	Loop	15	2	100	0
272	Spiral	Loop	15	2	100	0
273	Spiral	Loop	15	2	53	0
274	Spiral	End	15	2	1	3
275	Spiral	Start	15	3	2	3
276	Spiral	Loop	15	3	100	0
277	Spiral	Loop	15	3	9	0
278	Spiral	End	15	3	1	3

PLC virtuale – stream verso EtherCAT

#	Payload	Kind	Layer	BeadSeq	X	Y	Z	Var
0	Meta	End	15	1	0.0	0.0	0.0	0.00
1	Point	End	15	1	495.6	0.0	34.1	0.00
2	Variable	End	15	1	0.0	0.0	0.0	495.56

KUKA – BUFF1 (pacchetto corrente)

PK=End Layer=15 BeadSeq=1 Points=1/1 Vars=1/3

Points

#	X	Y	Z	A	B	C	E1	S
1	495.6	0.0	34.1	-450.0	0.0	149.5	0.0	2.0

Vars

#	Value
2	495.56

Figura A.14: Interfaccia grafica per pacchetto di End di un layer a spirale

Appendice B

Codice

```
70.25 499.89 0.20 180.00 0.00 -8.00 180.00 278.00 1 230 2.20 4.98 151 0 1
    1 0 2 8 1 2 -360.00 1 0 1.40
155.99 480.09 0.20 180.00 0.00 -18.00 180.00 288.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
236.99 445.71 0.20 180.00 0.00 -28.00 180.00 298.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
310.79 397.79 0.20 180.00 0.00 -38.00 180.00 308.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
375.14 337.78 0.20 180.00 0.00 -48.00 180.00 318.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
428.09 267.50 0.20 180.00 0.00 -58.00 180.00 328.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
468.04 189.10 0.20 180.00 0.00 -68.00 180.00 338.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
493.77 104.95 0.20 180.00 0.00 -78.00 180.00 348.00 0 230 2.20 4.98 151 0
    1 1 0 2 8 0 2 -360.00 1 0 1.40
```

Listing B.1: Estratto da 0001.cpi: record waypoints/processi (campi 1–12 + variabili di servizio).

```
DECL E6POS XP1_hover = {X 70.25,Y 499.89,Z 200.2, A -8, B 0, C 180,
    S 2,T 10,E1 278,E2 0.0,E3 0,E4 0,E5 0.0,E6 0.0}

DECL E6POS XP1 = {X 70.25,Y 499.89,Z 0.2, A -368, B 0, C 180, S 2,T
    10,E1 278,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL stArcDat_T WDAT1={Strike {JobModeId[] "WAAM Standard",
   ParamSetId[] "WAAMPlanner",StartTime 0.5,PreFlowTime 5,Channel1
   151,Channel2 200,Channel3 0,Channel4 0.8,Channel5 8,Channel6
   0.0,Channel7 0.0,Channel8 0.0,PurgeTime 0.0},Weld {JobModeId[] "
   WAAM Standard",ParamSetId[] "WAAMPlanner",Velocity 0.00498,
   Channel1 151,Channel2 230,Channel3 2.2,Channel4 0.8,Channel5
   8.0,Channel6 0.0,Channel7 0.0,Channel8 0.0},Weave {Pattern #None
   ,Length 4.00000,Amplitude 2.00000,Angle 0.0,LeftSideDelay 0.0,
   RightSideDelay 0.0},Advanced {IgnitionErrorStrategy 1,
   WeldErrorStrategy 1,SlopeOption #None,SlopeTime 0.0,
   SlopeDistance 0.0,OnTheFlyActiveOn FALSE,OnTheFlyActiveOff FALSE
   ,OnTheFlyDistanceOn 0.0,OnTheFlyDistanceOff 0.0}}
DECL stArcDat_T WP1={Strike {SeamName[] " ", PartName[] " ",
   SeamNumber 0,PartNumber 0},Advanced {BitCodedRobotMark 0}}
```

```
DECL E6POS XP2 = {X 155.99,Y 480.09,Z 0.2, A -378, B 0, C 180, S 2,
   T 10,E1 288,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL E6POS XP3 = {X 236.99,Y 445.71,Z 0.2, A -388, B 0, C 180, S 2,
   T 10,E1 298,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL stArcDat_T WDAT3={Weld {JobModeId[] "WAAM Standard",ParamSetId
   [] "WAAMPlanner",Velocity 0.00498,Channel1 151,Channel2 230,
   Channel3 2.2,Channel4 0.8,Channel5 8.0,Channel6 0.0,Channel7
   0.0,Channel8 0.0},Weave {Pattern #None,Length 4.00000,Amplitude
   2.00000,Angle 0.0,LeftSideDelay 0.0,RightSideDelay 0.0},Advanced
    {IgnitionErrorStrategy 1,WeldErrorStrategy 1,SlopeOption #None,
   SlopeTime 0.0,SlopeDistance 0.0,OnTheFlyActiveOn FALSE,
   OnTheFlyActiveOff FALSE,OnTheFlyDistanceOn 0.0,
   OnTheFlyDistanceOff 0.0}}
DECL stArcDat_T WP3={Strike {SeamName[] " ", PartName[] " ",
   SeamNumber 0,PartNumber 0},Advanced {BitCodedRobotMark 0}}

DECL E6POS XP4 = {X 310.79,Y 397.79,Z 0.2, A -398, B 0, C 180, S 2,
   T 10,E1 308,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL E6POS XP5 = {X 375.14,Y 337.78,Z 0.2, A -408, B 0, C 180, S 2,
   T 10,E1 318,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL stArcDat_T WDAT5={Weld {JobModeId[] "WAAM Standard",ParamSetId
   [] "WAAMPlanner",Velocity 0.00498,Channel1 151,Channel2 230,
   Channel3 2.2,Channel4 0.8,Channel5 8.0,Channel6 0.0,Channel7
   0.0,Channel8 0.0},Weave {Pattern #None,Length 4.00000,Amplitude
   2.00000,Angle 0.0,LeftSideDelay 0.0,RightSideDelay 0.0},Advanced
    {IgnitionErrorStrategy 1,WeldErrorStrategy 1,SlopeOption #None,
   SlopeTime 0.0,SlopeDistance 0.0,OnTheFlyActiveOn FALSE,
   OnTheFlyActiveOff FALSE,OnTheFlyDistanceOn 0.0,
   OnTheFlyDistanceOff 0.0}}
DECL stArcDat_T WP5={Strike {SeamName[] " ", PartName[] " ",
   SeamNumber 0,PartNumber 0},Advanced {BitCodedRobotMark 0}}

DECL E6POS XP6 = {X 428.09,Y 267.5,Z 0.2, A -418, B 0, C 180, S 2,T
    10,E1 328,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL E6POS XP7 = {X 468.04,Y 189.1,Z 0.2, A -428, B 0, C 180, S 2,T
    10,E1 338,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL stArcDat_T WDAT7={Weld {JobModeId[] "WAAM Standard",ParamSetId
   [] "WAAMPlanner",Velocity 0.00498,Channel1 151,Channel2 230,
   Channel3 2.2,Channel4 0.8,Channel5 8.0,Channel6 0.0,Channel7
   0.0,Channel8 0.0},Weave {Pattern #None,Length 4.00000,Amplitude
   2.00000,Angle 0.0,LeftSideDelay 0.0,RightSideDelay 0.0},Advanced
    {IgnitionErrorStrategy 1,WeldErrorStrategy 1,SlopeOption #None,
   SlopeTime 0.0,SlopeDistance 0.0,OnTheFlyActiveOn FALSE,
   OnTheFlyActiveOff FALSE,OnTheFlyDistanceOn 0.0,
   OnTheFlyDistanceOff 0.0}}
DECL stArcDat_T WP7={Strike {SeamName[] " ", PartName[] " ",
   SeamNumber 0,PartNumber 0},Advanced {BitCodedRobotMark 0}}

DECL E6POS XP8 = {X 493.77,Y 104.95,Z 0.2, A -438, B 0, C 180, S 2,
   T 10,E1 348,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
```

Listing B.2: Estratto da `DepRoll0001.dat`: punti E6POS e parametri WDAT corrispondenti alle righe CPI.

```
$BWDSTART = FALSE
LDAT_ACT = LCPDAT
LDAT_ACT.APO_DIST = wArcApoDistance
```

```
FDAT_ACT = FweldA
BAS(#CP_PARAMS, gArcBasVelDefinition)
SET_CD_PARAMS (0)
$CIRC_TYPE=#BASE
TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO=-1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#ArcSwiMoveStd,
   WDAT3, WP3) PRIO = -1
ArcMainNG(#ArcSwiBeforeMoveStd, WDAT3, WP3)
CIRC XP2, XP3 C_Dis C_Dis
ArcMainNG(#ArcSwiAfterMoveStd, WDAT3, WP3)

$BWDSTART = FALSE
LDAT_ACT = LCPDAT
LDAT_ACT.APO_DIST = wArcApoDistance
FDAT_ACT = FweldA
BAS(#CP_PARAMS, gArcBasVelDefinition)
SET_CD_PARAMS (0)
$CIRC_TYPE=#BASE
TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO=-1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#ArcSwiMoveStd,
   WDAT5, WP5) PRIO = -1
ArcMainNG(#ArcSwiBeforeMoveStd, WDAT5, WP5)
CIRC XP4, XP5 C_Dis C_Dis
ArcMainNG(#ArcSwiAfterMoveStd, WDAT5, WP5)
```

Listing B.3: Estratto da `DepRoll0001.src` che evidenzia la struttura ripetitiva. Ogni blocco di movimento `CIRC` (o `LIN`) è replicato esplicitamente, dimostrando l'assenza di cicli e la scarsa leggibilità.

```
DECL E6POS XP2 = {X 496.64,Y -87.57,Z 0.2, A -460, B 0, C 180, S 2,
   T 10,E1 10,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL E6POS XP3 = {X 473.89,Y -172.48,Z 0.2, A -470, B 0, C 180, S
   2,T 10,E1 20,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL stArcDat_T WDAT3={Weld {JobModeId[] "WAAM Standard",ParamSetId
   [] "WAAMPlanner",Velocity 0.00332,Channel1 151,Channel2 230,
   Channel3 1.45,Channel4 0.8,Channel5 8.0,Channel6 0.0,Channel7
   0.0,Channel8 0.0},Weave {Pattern #None,Length 4.00000,Amplitude
   2.00000,Angle 0.0,LeftSideDelay 0.0,RightSideDelay 0.0},Advanced
    {IgnitionErrorStrategy 1,WeldErrorStrategy 1,SlopeOption #None,
   SlopeTime 0.0,SlopeDistance 0.0,OnTheFlyActiveOn FALSE,
   OnTheFlyActiveOff FALSE,OnTheFlyDistanceOn 0.0,
   OnTheFlyDistanceOff 0.0}}
DECL stArcDat_T WP3={Strike {SeamName[] " ", PartName[] " ",
   SeamNumber 0,PartNumber 0},Advanced {BitCodedRobotMark 0}}

DECL E6POS XP4 = {X 436.74,Y -252.15,Z 0.2, A -480, B 0, C 180, S
   2,T 10,E1 30,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL E6POS XP5 = {X 386.32,Y -324.16,Z 0.2, A -490, B 0, C 180, S
   2,T 10,E1 40,E2 0,E3 0,E4 0,E5 0.0,E6 0.0}
DECL stArcDat_T WDAT5={Weld {JobModeId[] "WAAM Standard",ParamSetId
   [] "WAAMPlanner",Velocity 0.00332,Channel1 151,Channel2 230,
   Channel3 1.45,Channel4 0.8,Channel5 8.0,Channel6 0.0,Channel7
   0.0,Channel8 0.0},Weave {Pattern #None,Length 4.00000,Amplitude
   2.00000,Angle 0.0,LeftSideDelay 0.0,RightSideDelay 0.0},Advanced
    {IgnitionErrorStrategy 1,WeldErrorStrategy 1,SlopeOption #None,
   SlopeTime 0.0,SlopeDistance 0.0,OnTheFlyActiveOn FALSE,
   OnTheFlyActiveOff FALSE,OnTheFlyDistanceOn 0.0,
   OnTheFlyDistanceOff 0.0}}
DECL stArcDat_T WP5={Strike {SeamName[] " ", PartName[] " ",
   SeamNumber 0,PartNumber 0},Advanced {BitCodedRobotMark 0}}
```

Listing B.4: Estratto da `DepRoll0001.dat` che illustra la struttura dati "piatta". Ogni punto (`DECL E6POS XPn`), dato di saldatura (`DECL stArcDat_T WDATn`), e parametro (`WPn`) è dichiarato come una variabile globale univoca, rendendo la gestione dei dati inefficiente e complessa.

```
# Espressioni regolari per il parsing dei blocchi KRL
decl_re = re.compile(r'^\s*DECL\s+E6POS\s+XP(\d+)\s*=\s*(\{.*)
   $', re.IGNORECASE)
wdat_re = re.compile(r'^\s*DECL\s+\w+\s+WDAT(\d+)\s*=\s*(\{.*)
   $', re.IGNORECASE)
wp_re = re.compile(r'^\s*DECL\s+\w+\s+WP(\d+)\s*=\s*(\{.*)$',
   re.IGNORECASE)
rp_re = re.compile(r'^\s*DECL\s+R_POS\s+(RP\d+)\s*=\s*(\{.*)$'
   , re.IGNORECASE)
xp_hover_re = re.compile(r'^\s*DECL\s+E6POS\s+(XP\d+_hover)\s
   *=\s*(\{.*)$', re.IGNORECASE)
```

Listing B.5: Definizione delle espressioni regolari per il parsing dei blocchi .dat.

```
def categorize_indices(indices):
    """
    Categorizza gli indici secondo il pattern:
    - init: 1, 42, 83, 124... (1 + 41n)
```

```
- main: 2-37, 43-78, 84-119... (36 elementi per ciclo)
- end: 38, 79, 120... (38 + 41n)
- end1: 40, 81, 122... (40 + 41n)
- end2: 41, 82, 123... (41 + 41n)
"""
init_indices = []
main_indices = []
end_indices = []
end1_indices = []
end2_indices = []

for idx in sorted(indices):
    pos_in_cycle = ((idx - 1) % 41) + 1

    if pos_in_cycle == 1:
        init_indices.append(idx)
    elif 2 <= pos_in_cycle <= 37:
        main_indices.append(idx)
    elif pos_in_cycle == 38:
        end_indices.append(idx)
    elif pos_in_cycle == 40:
        end1_indices.append(idx)
    elif pos_in_cycle == 41:
        end2_indices.append(idx)
    # pos 39 non viene usato in nessun array

return init_indices, main_indices, end_indices, end1_
   indices, end2_indices
```

Listing B.6: Logica di categorizzazione degli indici dei punti.

```
# ... (Omesse altre generazioni di array) ...

# XP (main)
if xp_main:
    body_lines.append(f"DECL E6POS XP[{len(xp_main)}]")
    body_lines.append("")
    for arr_idx, orig_idx in enumerate(xp_main, start=1):
        body_lines.append(f"XP[{arr_idx}] = {points[orig_
           idx]}")
    body_lines.append("")

# ... (Omesse altre generazioni di array) ...
```

Listing B.7: Estratto dalla funzione make_array_dat: generazione dell'array XP principale.

```
IF recovery_mode THEN
   bead_actual = b_Selected
ELSE
   bead_actual = 1
ENDIF

;INIZIO DEL LOOP PER OGNI BEAD;
FOR bead=bead_actual TO bead_max STEP 1


    ;FOLD INIZIO_BEAD
    Theorical_Layer_Height = 2.5
    Top_Load = 75000

    IF start_bead_Done == False THEN
        $BWDSTART = FALSE
        PDAT_ACT = PPDAT
        FDAT_ACT = FHome
        BAS(#PTP_PARAMS, 10.0)
        SET_CD_PARAMS(0)
        XhomeA.E1 = $POS_ACT_MES.E1
        XhomeA.E2 = $POS_ACT_MES.E2
        ;PTP XhomeA
        PTP $POS_ACT

        Ehome = $POS_ACT_MES
        $BWDSTART = FALSE
        PDAT_ACT = PPDAT
        FDAT_ACT = FHome
        BAS(#PTP_PARAMS, 60.0)
        SET_CD_PARAMS(0)
        Ehome.E1 = XP_hover_init[bead].E1
        Ehome.E2 = XP_hover_init[bead].E2
        PTP Ehome

        $BWDSTART = FALSE
        LDAT_ACT = LCPDAT
        FDAT_ACT = FweldA
        BAS(#CP_PARAMS, 0.2)
        SET_CD_PARAMS(0)
        LIN XP_hover_init[bead]

        R_PTP(RP[bead])

        R_START(FT_Full)

        IF (FT_Full.FT == 0) THEN
            WHILE TOP_FORCE_APPLIED == FALSE
                WAIT SEC 0.1
            ENDWHILE
        ENDIF

        R_START(FL_Full)

        IF (FT_Full.FL1 == 0) THEN
            WHILE LATERAL_FORCE_APPLIED == FALSE
```

```
                WAIT SEC 0.1
            ENDWHILE
        ENDIF

        $BWDSTART = FALSE
        LDAT_ACT = LCPDAT
        FDAT_ACT = FweldA
        BAS(#CP_PARAMS, 0.2)
        SET_CD_PARAMS(0)
        LIN XP_init[bead]
        ; start the local shielding gas and purge for ... sec
        $OUT[50] = TRUE
        $OUT[51] = TRUE
        WAIT SEC 5
        TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
           =-1
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
        TRIGGER WHEN DISTANCE = 1 DELAY = ArcGetDelay(#
           PreDefinition, WDAT_init[bead]) DO ArcMainNG(#
           PreDefinition, WDAT_init[bead], WP_init[bead]) PRIO = -1
        TRIGGER WHEN PATH = ArcGetPath(#OnTheFlyArcOn, WDAT_init[
           bead]) DELAY = ArcGetDelay(#GasPreflow, WDAT_init[bead])
            DO ArcMainNG(#GasPreflow, WDAT_init[bead], WP_init[bead
           ]) PRIO = -1
        TRIGGER WHEN PATH = ArcGetPath(#OnTheFlyArcOn, WDAT_init[
           bead]) DELAY = 0 DO ArcMainNG(#ArcOnMoveStd, WDAT_init[
           bead], WP_init[bead]) PRIO = -1
        ArcMainNG(#ArcOnBeforeMoveStd, WDAT_init[bead], WP_init[
           bead])
        LIN XP_init[bead]
        ArcMainNG(#ArcOnAfterMoveStd, WDAT_init[bead], WP_init[bead
           ])

        start_bead_Done = True  ;====== flag iniziale impostata su
           True

    ENDIF

    ;ENDFOLD INIZIO_BEAD

   IF recovery_mode THEN
      RECOVERY_ENTRY()
      seg_actual = s_Selected
   ELSE
      seg_actual = 1
   ENDIF

   ;INIZIO DEL LOOP PER OGNI SEGMENTO (18 SEGMENTI A BEAD);
   ;FOLD FOR_LOOP_SEGMENTI
   FOR seg=seg_actual TO seg_per_bead STEP 1

      mid_idx = (2*seg + (36*(bead-1)) - 1)
      end_idx = mid_idx + 1

      $BWDSTART = FALSE
      LDAT_ACT = LCPDAT
      LDAT_ACT.APO_DIST = wArcApoDistance
```

```
        FDAT_ACT = FweldA
        BAS(#CP_PARAMS, gArcBasVelDefinition)
        SET_CD_PARAMS (0)
        $CIRC_TYPE=#BASE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO SegDone[seg]=TRUE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO WAAM_RSI_CHECK() PRIO
            =-1
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#
            ArcSwiMoveStd, WDAT[end_idx], WP[end_idx]) PRIO = -1
        ArcMainNG(#ArcSwiBeforeMoveStd, WDAT[end_idx], WP[end_idx])
        CIRC XP[mid_idx], XP[end_idx] C_Dis C_Dis
        ArcMainNG(#ArcSwiAfterMoveStd, WDAT[end_idx], WP[end_idx])

    ENDFOR
    ;ENDFOLD FOR_LOOP_SEGMENTI

    ;FINE BEAD;
    ;FOLD FINE_BEAD
    $BWDSTART = FALSE
    LDAT_ACT = LCPDAT
    FDAT_ACT = FweldA
    BAS(#CP_PARAMS, gArcBasVelDefinition)
    SET_CD_PARAMS(0)
    TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO=-1
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
    TRIGGER WHEN PATH = ArcGetPath(#ArcOffBefore, WDAT_end[bead])
        DELAY = 0 DO ArcMainNG(#ArcOffBeforeOffStd, WDAT_end[bead],
        WP_end[bead]) PRIO = -1
    TRIGGER WHEN PATH = ArcGetPath(#OnTheFlyArcOff, WDAT_end[bead])
        DELAY = 0 DO ArcMainNG(#ArcOffMoveStd, WDAT_end[bead], WP_end
        [bead]) PRIO = -1
    ArcMainNG(#ArcOffBeforeMoveStd, WDAT_end[bead], WP_end[bead])
    LIN XP_end[bead]
    ArcMainNG(#ArcOffAfterMoveStd, WDAT_end[bead], WP_end[bead])

    Local_Shielding_Value_1 = 100
    Local_Shielding_Value_2 = 100
    WAIT SEC 0
    $OUT[50] = FALSE
    $OUT[51] = FALSE

    $BWDSTART = FALSE
    LDAT_ACT = LCPDAT
    LDAT_ACT.APO_DIST = wArcApoDistance
    FDAT_ACT = FweldA
    BAS(#CP_PARAMS, 0.00332)
    SET_CD_PARAMS (0)
    TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO=-1
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
    LIN XP_end1[bead] C_Dis C_Dis

    $BWDSTART = FALSE
    LDAT_ACT = LCPDAT
    LDAT_ACT.APO_DIST = wArcApoDistance
```

```
    FDAT_ACT = FweldA
    BAS(#CP_PARAMS, 0.00332)
    SET_CD_PARAMS (0)
    TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO=-1
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 1
    TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO BeadDone[bead]=TRUE
    LIN XP_end2[bead] C_Dis C_Dis

    oIsScanning = FALSE
    WAIT SEC 2

    $BWDSTART = FALSE
    LDAT_ACT = LCPDAT
    FDAT_ACT = FweldA
    BAS(#CP_PARAMS, 0.2)
    SET_CD_PARAMS(0)
    LIN XP_hover_end[bead]

    R_STOP(FL_End)

    IF (FT_Full.FL1 == 0) THEN
       WHILE LATERAL_FORCE_APPLIED == TRUE
          WAIT SEC 0.1
       ENDWHILE
    ENDIF

    R_STOP(FT_End)

    IF (FT_Full.FT == 0) THEN
       WHILE TOP_FORCE_APPLIED == TRUE
          WAIT SEC 0.1
       ENDWHILE
    ENDIF


    R_PTP(RP[bead])
    start_bead_Done = False
    ;ENDFOLD FINE_BEAD

  ENDFOR

oLayerCompleted = TRUE
WAAM_RSI_OFF()
WAAM_RSI_DELETE()
END
```

Listing B.8: Template KRL per il file .src semplificato (variabile fixed_content).

```python
# Crea il nuovo file
with open(src_out, 'w', encoding='utf-8') as f:
    # Scrivi le prime 22 righe di DepRoll0001.src
    for i in range(22):
        f.write(deproll_lines[i])

    # Sostituisci i valori nelle righe fisse
    for line in fixed_content.split('\n'):
```

```
if 'Theorical_Layer_Height' in line and '=' in
   line and theorical_layer_height:
    f.write('    ' + theorical_layer_height + '\n'
       )
elif 'Top_Load' in line and '=' in line and top_
   load:
    f.write('    ' + top_load + '\n')
else:
    f.write(line + '\n')
```

Listing B.9: Logica di assemblaggio del file .src finale.

```
R_STOP(FL_End)

IF (FT_Full.FL1 == 0) THEN
WHILE LATERAL_FORCE_APPLIED == TRUE
WAIT SEC 0.1
ENDWHILE
ENDIF

R_STOP(FT_End)

IF (FT_Full.FT == 0) THEN
WHILE TOP_FORCE_APPLIED == TRUE
WAIT SEC 0.1
ENDWHILE
ENDIF


R_PTP(RP7)

R_START(FL_Full)

WHILE LATERAL_FORCE_APPLIED == FALSE
WAIT SEC 0.1
ENDWHILE

$BWDSTART = FALSE
LDAT_ACT = LCPDAT
LDAT_ACT.APO_DIST = wArcApoDistance
FDAT_ACT = FweldA
BAS(#CP_PARAMS, 0.00498)
SET_CD_PARAMS (0)
TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
   =-1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 8
CIRC XP254, XP255 C_Dis C_Dis

$BWDSTART = FALSE
LDAT_ACT = LCPDAT
FDAT_ACT = FweldA
BAS(#CP_PARAMS, 0.2)
SET_CD_PARAMS(0)
LIN XP256
TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
   =-1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 8
TRIGGER WHEN DISTANCE = 1 DELAY = ArcGetDelay(#
   PreDefinition, WDAT256) DO ArcMainNG(#PreDefinition,
   WDAT256, WP256) PRIO = -1
TRIGGER WHEN PATH = ArcGetPath(#OnTheFlyArcOn, WDAT256)
   DELAY = ArcGetDelay(#GasPreflow, WDAT256) DO ArcMainNG(#
   GasPreflow, WDAT256, WP256) PRIO = -1
TRIGGER WHEN PATH = ArcGetPath(#OnTheFlyArcOn, WDAT256)
   DELAY = 0 DO ArcMainNG(#ArcOnMoveStd, WDAT256, WP256)
   PRIO = -1
ArcMainNG(#ArcOnBeforeMoveStd, WDAT256, WP256)
```

```
        LIN XP256
        ArcMainNG(#ArcOnAfterMoveStd, WDAT256, WP256)

        $BWDSTART = FALSE
        LDAT_ACT = LCPDAT
        LDAT_ACT.APO_DIST = wArcApoDistance
        FDAT_ACT = FweldA
        BAS(#CP_PARAMS, gArcBasVelDefinition)
        SET_CD_PARAMS (0)
        $CIRC_TYPE=#BASE
        TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
            =-1
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 8
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#
            ArcSwiMoveStd, WDAT258, WP258) PRIO = -1
        ArcMainNG(#ArcSwiBeforeMoveStd, WDAT258, WP258)
        CIRC XP257, XP258 C_Dis C_Dis
        ArcMainNG(#ArcSwiAfterMoveStd, WDAT258, WP258)

        $BWDSTART = FALSE
        LDAT_ACT = LCPDAT
        LDAT_ACT.APO_DIST = wArcApoDistance
        FDAT_ACT = FweldA
        BAS(#CP_PARAMS, gArcBasVelDefinition)
        SET_CD_PARAMS (0)
        $CIRC_TYPE=#BASE
        TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
            =-1
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 8
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#
            ArcSwiMoveStd, WDAT260, WP260) PRIO = -1
        ArcMainNG(#ArcSwiBeforeMoveStd, WDAT260, WP260)
        CIRC XP259, XP260 C_Dis C_Dis
        ArcMainNG(#ArcSwiAfterMoveStd, WDAT260, WP260)

        ...

        $BWDSTART = FALSE
        LDAT_ACT = LCPDAT
        LDAT_ACT.APO_DIST = wArcApoDistance
        FDAT_ACT = FweldA
        BAS(#CP_PARAMS, gArcBasVelDefinition)
        SET_CD_PARAMS (0)
        $CIRC_TYPE=#BASE
        TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
            =-1
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 14
        TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#
            ArcSwiMoveStd, WDAT506, WP506) PRIO = -1
        ArcMainNG(#ArcSwiBeforeMoveStd, WDAT506, WP506)
        CIRC XP505, XP506 C_Dis C_Dis
        ArcMainNG(#ArcSwiAfterMoveStd, WDAT506, WP506)

        $BWDSTART = FALSE
        LDAT_ACT = LCPDAT
```

```
LDAT_ACT.APO_DIST = wArcApoDistance
FDAT_ACT = FweldA
BAS(#CP_PARAMS, gArcBasVelDefinition)
SET_CD_PARAMS (0)
$CIRC_TYPE=#BASE
TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
    =-1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = FALSE
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 15
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO ArcMainNG(#
    ArcSwiMoveStd, WDAT508, WP508) PRIO = -1
ArcMainNG(#ArcSwiBeforeMoveStd, WDAT508, WP508)
CIRC XP507, XP508 C_Dis C_Dis
ArcMainNG(#ArcSwiAfterMoveStd, WDAT508, WP508)

$BWDSTART = FALSE
LDAT_ACT = LCPDAT
FDAT_ACT = FweldA
BAS(#CP_PARAMS, gArcBasVelDefinition)
SET_CD_PARAMS(0)
TRIGGER WHEN DISTANCE = 1 DELAY =0 DO WAAM_RSI_CHECK() PRIO
    =-1
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oIsScanning = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = 0 DO oLayerNum = 15
TRIGGER WHEN PATH = ArcGetPath(#ArcOffBefore, WDAT509)
    DELAY = 0 DO ArcMainNG(#ArcOffBeforeOffStd, WDAT509,
    WP509) PRIO = -1
TRIGGER WHEN PATH = ArcGetPath(#OnTheFlyArcOff, WDAT509)
    DELAY = 0 DO ArcMainNG(#ArcOffMoveStd, WDAT509, WP509)
    PRIO = -1
ArcMainNG(#ArcOffBeforeMoveStd, WDAT509, WP509)
LIN XP509
ArcMainNG(#ArcOffAfterMoveStd, WDAT509, WP509)

R_STOP(FL_End)

IF (FT_Full.FL1 == 0) THEN
WHILE LATERAL_FORCE_APPLIED == TRUE
WAIT SEC 0.1
ENDWHILE
ENDIF

R_STOP(FT_End)

IF (FT_Full.FT == 0) THEN
WHILE TOP_FORCE_APPLIED == TRUE
WAIT SEC 0.1
ENDWHILE
ENDIF


R_PTP(RP14)

R_START(FL_Full)

WHILE LATERAL_FORCE_APPLIED == FALSE
WAIT SEC 0.1
ENDWHILE
```

Listing B.10: Estratto da `DepRoll0001_sez_4.src` che evidenzia La struttura della macro-sequenza tra due R_PTP

put your greetings here..

Elenco delle figure

1.1 Configurazione attuale della cella WAAM presente in azienda. 18

2.1 Categorie dell'Additive Manufacturing 22
2.2 Fasi del processo WAAM 24
2.3 Principali tecnologie di saldatura utilizzate nell'Additive Manufacturing . 26
2.4 Confronto tra le principali tecnologie di Additive Manufacturing per metalli 28

4.1 Schema a blocchi della catena di controllo WAAM attuale, basata su file KRL statici. 36
4.2 Schema delle possibili posizioni di procedura di $XP_{\text{last_weld}}[s_{\text{Selected}}]$. 50
4.3 Flowchart del programma principale di deposizione e della logica di *recovery* (segue). 54
4.3 Flowchart del programma principale di deposizione e della logica di *recovery*. 55

5.1 Sezioni di diversa tipologia 58
5.2 Sezioni nella zona inclinata 59
5.3 Struttura del JSON 65

6.1 Confronto tra layout attuale e nuovo layout della catena di controllo WAAM. 73
6.2 Schema a blocchi della catena di controllo WAAM nuova, basata su file JSON. 74

7.1 Simulazione PC↔PLC via ADS (Cap. 7): generazione e serializzazione del pacchetto su PC, ricezione e verifica su PLC virtuale, e collegamento allo streaming EtherCAT del Cap. 8. 76
7.2 Layout della finestra di simulazione 82

8.1 Architettura proposta di streaming PC–PLC–KUKA per il processo WAAM: dal JSON di progetto alla cella fisica tramite ADS e EtherCAT. 86
8.2 Flusso completo della simulazione PC–PLC–KUKA (Cap. 8): generazione pacchetti, streaming EtherCAT, assemblaggio su KUKA e casi di test. 89
8.3 Flowchart del task KUKA SPS: Task2KukaAssembler e BUFF_1. 96

A.1 Sezione del manufatto in Rhino 111
A.2 Ingrandimento area in slicing 111
A.3 Schema completo del programma Grasshopper 112
A.4 Applicazione per la simulazione 113
A.5 Task 0 durante la simulazione 114
A.6 Task 1 durante la simulazione 114

A.7 Task 2 & 3 in simulazione 115
A.8 Task 4 in simulazione 115
A.9 Interfaccia grafica per pacchetto di Start layer "normale" 116
A.10 Interfaccia grafica per pacchetto di Loop layer "normale" 117
A.11 Interfaccia grafica per pacchetto di End layer "normale" 118
A.12 Interfaccia grafica per pacchetto di Start di un layer a spirale 119
A.13 Interfaccia grafica per pacchetto di Loop di un layer a spirale 120
A.14 Interfaccia grafica per pacchetto di End di un layer a spirale 121

Bibliografia

[1] M. Srivastava, S. Rathee, A. Tiwari, and M. Dongre, "Wire arc additive manufacturing of metals: A review on processes, materials and their behaviour," *Materials Chemistry and Physics*, vol. 294, p. 126988, 2023.

[2] "Standard Terminology for Additive Manufacturing Technologies, (Withdrawn 2015) — doi.org." `https://doi.org/10.1520/F2792-12A`. [Accessed 07-10-2025].

[3] Y. Li, C. Su, and J. Zhu, "Comprehensive review of wire arc additive manufacturing: Hardware system, physical process, monitoring, property characterization, application and future prospects," *Results in Engineering*, vol. 13, p. 100330, 2022.

[4] K. Treutler and V. Wesling, "The current state of research of wire arc additive manufacturing (waam): A review," *Applied Sciences*, vol. 11, no. 18, 2021.

[5] M. Sebok, C. Lai, C. Masuo, A. Walters, W. Carter, N. Lambert, L. Meyer, J. Officer, A. Roschli, J. Vaughan, and A. Nycz, "Slicing solutions for wire arc additive manufacturing," *Journal of Manufacturing and Materials Processing*, vol. 9, no. 4, 2025.

[6] H. R. Peter, K. Sargent, J. Penney, and T. Schmitz, "Path programming in rhino 7 for wire arc additive manufacturing," *Manufacturing Letters*, vol. 44, pp. 973–979, 2025. 53rd SME North American Manufacturing Research Conference (NAMRC 53).

[7] C. Xia, Z. Pan, J. Polden, H. Li, Y. Xu, S. Chen, and Y. Zhang, "A review on wire arc additive manufacturing: Monitoring, control and a framework of automated system," *Journal of Manufacturing Systems*, 1 2020.

[8] Y. Li, C. Su, and J. Zhu, "Comprehensive review of wire arc additive manufacturing: Hardware system, physical process, monitoring, property characterization, application and future prospects," *Results in Engineering*, vol. 13, p. 100330, 2022.

[9] C.-L. Lu, H. He, J. Ren, J. Dhar, G. Saunders, A. Julius, J. Samuel, and J. T. Wen, "Multi-robot scan-n-print for wire arc additive manufacturing," *ASME Letters in Translational Robotics*, vol. 1, p. 011003, 02 2025.

[10] H. He, C.-L. Lu, J. Ren, J. Dhar, G. Saunders, J. Wason, J. Samuel, A. Julius, and J. T. Wen, "Open-source software architecture for multi-robot wire arc additive manufacturing (waam)," *Applications in Engineering Science*, vol. 22, p. 100204, 2025.