UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Ingegneria Informatica, delle Comunicazioni ed Elettronica

Elaborato finale

Prolog planner

Task planner logico per la manipolazioni di blocchi tramite un UR5

Supervisore
Prof. Luigi Palopoli

Laureando
Davide De Martini

Anno accademico 2022/2023

Ringraziamenti

A tutti

Indice

Sommario

Sommario è un breve riassunto del lavoro svolto dove si descrive l'obiettivo, l'oggetto della tesi, le metodologie e le tecniche usate, i dati elaborati e la spiegazione delle conclusioni alle quali siete arrivati.

Il sommario dell'elaborato consiste al massimo di 3 pagine e deve contenere le seguenti informazioni:

- contesto e motivazioni
- breve riassunto del problema affrontato
- tecniche utilizzate e/o sviluppate
- risultati raggiunti, sottolineando il contributo personale del laureando/a

1 Introduzione

In questo primo capitolo viene introdotto il caso di studio, partendo dalla definizione del problema, definendo poi gli obiettivi desiderati e concludendo con una spiegazione della struttura della tesi.

Per la mia tesi triennale mi sono voluto concentrare sul campo dell'intelligenza artificiale e la robotica. Il caso di studio identificato dal prof. Luigi Palopoli è l'utilizzo di Prolog, un linguaggio logico e dichiarativo, per l'implementazione di un task planner ad alto livello. Questo deve essere in grado di coordinare un manipolatore robotico, nel nostro caso un UR5 della Universal Robots, per l'operazione di manipolazione di blocchi. Il planner tramite delle primitive come *ruota* e *muovi* deve essere in grado di pilotare il braccio robotico per la creazione di un pilastro. Esso è stato concepito ad alto livello per permettere la compatibilità con più tipologie di robot e la flessibilità delle applicazioni. Nel caso studiato verrà utilizzato per la creazione, appunto, di un pilastro, ma questa astrazione ad "alto livello" permette di essere usato per più compiti: ad esempio per la creazione di una costruzione come una casa o qualsiasi struttura che si può creare con dei blocchi di Lego. Queste primitive verrando inviate al nodo controllore del movimento che, tramite algoritmi di cinematica e interpolazione dei punti, troverà le configurazioni dei giunti adatte al movimento richiesto.

Successivamente si è voluto implementare il tutto in un ambiente simulato, così da avere un riscontro anche visivo del lavoro svolto e capire le possibili applicazioni in un contesto reale. La simulazione è stata svolta con ROS e Gazebo, due strumenti open source ampiamente utilizzati nel mondo della robotica. Il primo ci permette di comunicare al robot tramite una serie di topic e servizi, mentre il secondo ci permette di simulare l'ambiente in cui il robot si trova.

La scelta di questo caso di studio è nata principalmente dal mio interesse verso la robotica e le applicazioni dell'intelligenza artificiale in vari contesti. Il caso studiato è stato un approccio nuovo che non avevo mai considerato e fin da subito mi è interessato. L'utilizzo di costrutti logici per rappresentare la conoscienza in un robot è un approccio molto interessante che può essere applicato in molti altri contesti. Un altro motivo che mi ha portato a scegliere questo caso di studio è la sua possibilità di espansione. Due possibili miglioramenti potrebbero essere la temporizzazione delle azioni e la successiva ottimizzazione del tempo di esecuzione (*makespan*) finale e l'utilizzo del machine learning per istanziare i fatti iniziali. Sapere che questo lavoro possa essere portato avanti e migliorato è sicuramente un altro motivo principale della scelta di questo caso di studio. Mi sono quindi rivolto al prof. Palopoli e lui mi ha proposto di sviluppare questo progetto. La scelta di utilizzare Prolog è nata dal fatto che è un ottimo strumento per modellare la conoscenza in un agente. Esso permette di

creare delle "basi di conoscenza", contenenti sia fatti che regole, su cui successivamente fare inferenza. L'esecuzione di un programma prolog è comparabile alla dimostrazione di un teorema mediante la regola di inferenza della soluzione. Tutte queste premesse, e l'apmpio utilizzo di prolog nel mondo dell'intelligenza artificiale, ci hanno incuriosito e portato allo sviluppo del progetto.

1.1 Definizione del problema

Il caso di studio presentato non è nuovo nel mondo dell'intelligenza artificiale. L'utilizzo di linguaggi logici per applicazioni di planning è, ed era, uno degli approcci preferiti dalla comunità per risolvere problemi in questo dominio. È difficile trovare delle applicazioni nella vita reale che utilizzano solamente Prolog, questo il più delle volte viene usato insieme ad altri strumenti o linguaggi di programmazione tipo Java.

Inizialmente la sfida è stata di prendere domestichezza con il linguaggio di programmazione. Essendo un paradigma che non avevo mai utilizzato e, abituato ai linguaggi imperativi come Python, Java, ho provato a usare un approcio simile in Prolog, fallendo. Questo perchè, al contrario di come siamo abituati, Prolog fa un uso massiccio della ricorsione e il concetto di variabile è differente da come siamo abituati. Presa domestichezza la sfida diventò quella di definire un modo efficace per rappresentare l'ambiente studiato. Dovevo riuscire a trovare un astrazione della realtà tale che, utilizzando una stringa, riuscisse a rappresentare tutte le informazioni necessarie sulle proprietà attuali del blocco di lego. Successivamente questa si è trasformata nel identificare le azioni base che il nostro planner doveva inviare al robot. Tutto ciò doveva essere svolto mantenendo consistenza tra il mondo modellato e la base di conoscenza, quindi evitare che un blocco che si è spostato nel mondo reale rimanga nella sua posizione precedente nella base di conoscenza.

1.2 Obiettivi desiderati

In sede di sviluppo del progetto abbiamo voluto specializzare il planner nella costruzione di pilastri di altezza scelta dall'utente. Questi sono composti dai *megablocks*, dei blocchi di lego di dimensione maggiorata, per facilitare la presa al nostro manipolatore. Nella base di conoscenza questi saranno rapressentati come dei fatti *ground*, ciò significa che il nostro interprete Prolog li assumerà per veri a priori. Il predicato che svolgerà la creazione del pilastro dovrà restituire tutte le possibili soluzioni che soddisfano il nostro problema. Successivamente una serie di azioni saranno calcolate per la costruzione del pilastro e queste verranno inviate al robot. Esso è un UR5 del produttore Universal Robots, un manipolatore a 6 gradi di libertà. Alla sua estremità è presente un gripper a due dita parallele della Soft Robotics che permette la presa dei blocchi. Tramite il piano generato, il robot dovrà costruire il pilastro richiesto con i blocchi messi a disposizione. Riceverà quindi una serie di azioni ad alto livello e convertirà queste in movimenti veri e propri del braccio tramite il nodo di motion planning.

1.3 Struttura della tesi

Questa tesi inizierà presentando lo stato dell'arte delle applicazioni in Prolog, sia l'uso come puro linguaggio logico che nei contesti di intelligenza artificiale e robotica. Vedremo l'evoluzione del linguaggio negli anni e la sua applicazione in ambiti sia di ricerca che di industria. Successivamente andrò a spiegare cosa è ROS e gli strumenti che ho utilizzato per simulare il robot. In questa sezione descriverò l'architettura e i nodi creati per il progetto, soffermandomi anche su concetti di cinematica e di motion planning. In seguito ci sarà un capitolo dedicato più dettagliatamente al caso di studio in cui il programma Prolog sarà spiegato più dettagliatamente e infine un capitolo dedicato alle conclusioni e ai lavori futuri.

Iniziamo ora introducendo appunto lo stato dell'arte di prolog e le pubblicazioni notabili, andando

a dare un approfondimento anche al mondo dell'industria.

2 Stato dell'arte

In questo capitolo verranno presentati i principali lavori che compongono lo stato dell'arte di Prolog. In particolare verrà inizialmente introdotto il linguaggio, poi sarà presentata la storia di Prolog, partendo dalla creazione fino a come lo conosciamo ora. Infine verrà introdotto la sua applicazione nel mondo portando i lavori notabili, sviluppando il focus nel ambito dell'automazione e della robotica. Per la stesura di questo capitolo ho utilizzato principalmente il libro *The art of Prolog* [4] e il lavoro di Körner [2].

2.1 Prolog

In questa sezione verrà introdotto il linguaggio, dandogli una definizione e una descrizione della sua attuale implementazione.

Ad oggi Prolog è considerato il linguaggio di programmazione logica più utilizzato e più importante. La sua evoluzione, però, non è stata lineare. Negli anni ha subito diverse modifiche, la cosa curiosa è che, a differenza di altri linguaggi, queste modifiche venivano integrate nel progetto principale invece che creare dei linguaggi paralleli. In ogni caso i pilastri del linguaggio sono rimasti invariati, qui di seguito elenco i principi fondamentali che ogni implementazione di Prolog deve rispettare:

1. Il linguaggio deve essere basato sulla logica dei predicati di primo ordine utilizzando le clausole di Horn. Sia per la creazione delle basi di conoscenza che per le interrogazioni delle basi di conoscenza (query).
2. Avere l'abilità di manipolare predicati e clausole come termini, così che i meta-predicati possano essere scritti come normali predicati.
3. Risoluzione SLD [3] basata al principio di risoluzione di Robinson.
4. Unificazione di termini arbitrari che includono variabili.
5. Esplorazione depth-first dell'albero di ricerca.

La spiegazione di questi requisiti minimi è ben descritta dall'articolo di Körner [2]. I punti fondamentali per un implementazione di Prolog sono quindi: l'uso delle clausole di Horn, la risoluzione SLD e l'unificazione di termini arbitrari. Ci sono altre importati funzioni che un implementazione dovrebbe supportare:

6. Negazione come fallimento e altri aspetti logici come la disgiunzione e l'implicazione.
7. Possibilità di alterare il contesto dell'esecuzione durante la risoluzione.
8. Possibilità di utilizzare la programmazione con *constraints*.

Di sicuro ci sono molte altre funzioni che devono essere presenti in una implementazione di Prolog, ho voluto elencare soltanto le principali.

La *Regola del pollice* per capire se un risolutore logico può essere considerato un implementazione di Prolog è la seguente. Il test richiede che la ben nota regola *append/3* possa essere scritta esattamente come segue

```
append([], X, X).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

e questa può essere interrogata in differenti modi: fare l'append tra due liste `append([1,2],[c,d], R)`, oppure decostruire una lista come `append(A,B,[1,2])` o infine istanziarla con argomenti arbitrari come `append([X|T],[c],[Z,Z,Z])`.

Descriviamo ora più in dettaglio la struttura di Prolog e la sua sintassi.

2.1.1 Struttura e sintassi

Iniziamo definendo la tipologia di dati supportata da Prolog. Il dato in Prolog viene chiamato termine, esso può essere un atomo, un numero, una variabile oppure un termine composto.

- Atomo: è un termine che non è né un numero né una variabile. Gli atomi sono usati per rappresentare nomi di oggetti, relazioni o costanti.
- Numero: i numeri in Prolog possono essere sia interi che reali.
- Variabili: sono usate per rappresentare oggetti o valori sconosciuti. Le variabili in Prolog iniziano con una lettera maiuscola o con il carattere di sottolineatura.
- Termine composto: esso è composto da un atomo chiamato *funtore* e un numero di argomenti. Il numero di argomenti è chiamato *arità* del termine composto. Casi speciali di termini composti sono le liste e le stringhe.

L'utilizzo di questi dati ci permette di definire le *clausole di Horn* [1] che sono alla base del linguaggio. La sintassi del prolog è basata appunto sulla logica dei predicati di primo ordine, limitata però alle clausole di Horn. Queste sono delle disguinzioni di letterali in cui al massimo uno dei letterali è positivo. Un esempio è il seguente:

$$\neg umano(X) \vee mortale(X) \tag{2.1}$$

Questo significa che:

$$\forall X(\neg umano(X) \vee mortale(X)) \tag{2.2}$$

Utilizzando l'equivalenza logica:

$$\neg X \vee Y \equiv X \Rightarrow Y \tag{2.3}$$

Quindi:

$$\forall X(umano(X) \Rightarrow mortale(X)) \tag{2.4}$$

In questo esempio possiamo vedere come è costituita una clausola di Horn. Se la premessa (*umano*) è vera allora anche la conseguenza (*mortale*) è vera.

Le clausole possono essere senza testa:

```
umano(luca).
padre(livio, lorenzo).
```

Oppure con testa:

```
mortale(lorenzo) :- umano(lorenzo).
```

Come detto prima, la logica in Prolog è espressa in termini di relazioni tra fatti e regole, la verifica di queste relazioni prende il nome di *query* o *interrogazione*. Un fatto in Prolog può essere visto come una clausola con il corpo vuoto, ad esempio:

```
umano(luca).
```

In questo esempio vediamo come si può esprimere logicamente che luca sia un umano. Un fatto può essere visto anche come una regola che a priori è sempre vera:

```
umano(luca) :- true.
```

Una regola, invece, è una clausola completa di testa e corpo, la testa è vera solamente se anche il corpo è vero. Un esempio di regola è quello visto prima:

```
mortale(X) :- umano(X).
```

Il corpo di una regola è un insieme di predicati, questi possono essere congiunti o disgiunti. L'operatore di congiunzione è la virgola (`,`) mentre quello di disgiunzione è il punto e virgola (`;`). Prolog viene fornito di predicati predefiniti, spesso utilizzati per la manipolazione di liste, aritmetica, input/output. Esempi di questi sono *append*, *is*, *write*, *read*, ecc.

L'insieme di questi 'costrutti' compone l'interità della sintassi base in prolog. Ci sono anche altri operatori, uno dei più importanti è il seguente: Il predicato \+ /1 definisce la *negazione come fallimento*. Ciò permette a Prolog di essere un sistema di ragionamento non monolitico.

```
legale(X) :- \+ illegale(X).
```

Per definire degli algoritmi iterativi come quelli che siamo abituati a vedere in linguaggi imperativi, Prolog utilizza la ricorsione. Io stesso ne ho fatto largo uso per la mia base di conoscenza, un esempio è il seguente:

```
list_length([], 0).

list_length([_|T], N) :-
    list_length(T, N1),
    N is N1 + 1.
```

In questo esempio il predicato è utilizzato per calcolare la lunghezza di una lista, vediamo come ci sia il caso base (lista vuota) e il caso ricorsivo. In ogni caso al momento l'ho usato soltato come esempio, nel capitolo 3 spiegherò meglio la struttura del programma e come ho risolto i problemi incontrati in Prolog.

2.1.2 Esecuzione

L'esecuzione di un programma Prolog è basata sulla ricerca di un insieme di predicati che soddisfano una data query. Il metodo di risoluzione utilizzato è la risoluzione SLD [3] (Selection, Linearization, and Driving). Questo processo di inferenza consiste in più fasi:

1. Selezionare una clausola goal, essa rappresenta quello che si desidera dimostrare.

2. Unificare la clausola goal con quella del programma, quindi trovare un modo di rendere uguali i termini nelle due clausole.

3. Risolvere la clausola unificata, il programma userà quindi le sostituzioni fatte nella fase precedente per risolvere la clausola. Questo genererà nuove clausole che verranno aggiunte all'insieme di clausole da risolvere.

4. Ripetere i passaggi 2 e 3 fino a quando non si raggiunge un punto di terminazione. La strategia di scelta delle clausole è la selezione lineare più a sinistra.

5. Il processo termina quando viene raggiunto uno di questi casi:

    - Una clausola vuota viene generata dimostrando quindi che la query è vera.
    - Non è possibile selezionare ulteriori clausole per unificare e risolvere.
    - Viene raggiunto il limite di profondità o di tempo (prestabilito dall'utente).

Per esplorare tutte le possibili soluzioni di un problema, Prolog utilizza il *backtraking*. Se durante l'esecuzione di una regola o l'unificazione di un fatto si raggiunge un punto in cui non è più possibile trovare delle soluzioni, Prolog torna indietro (backtrack) per cercare altre possibilità. Questo processo quindi permette di riprendere l'esplorazione delle alternative non ancora considerate. Durante il processo di backtracking Prolog annulla tutte le assegnazioni fatte precedentemente e cerca altre alternative. Le variabili unificate vengono quindi scollegate per permettere la ricerca di altre soluzioni. Un esempio di questo strumento:

```
padre(giovanni, maria).
padre(giovanni, giuseppe).
madre(maria, francesca).

genitore(X, Y) :- padre(X, Y).
genitore(X, Y) :- madre(X, Y).
```

Se noi dovessimo eseguire la query *genitore(giovanni, X)* Prolog troverebbe due soluzioni: $X = maria$ e $X = giuseppe$. Facendo così Prolog ha esplorato tutte le alternative possibili dell'albero di ricerca.

2.1.3 Debugging

Prolog fornisce un insieme di strumenti per il debugging, uno di questi è il *trace*. Questo strumento permette di vedere l'esecuzione del programma passo passo. In questo modo è possibile vedere come Prolog risolve le query e quali clausole vengono selezionate. Nella console di SWI-Prolog è possibile attivare il trace con il comando *trace.* e disattivarlo con *notrace.*. Avviandolo ad ogni chiamata di un predicato verrà mostrato il suo nome e i suoi argomenti. Inoltre verrà mostrato il risultato della sua risoluzione. Questo renderà il debugging di un programma più semplice, e permetterà di capire meglio il funzionamento di Prolog.

2.2 Evoluzione di prolog

La sezione seguente si concentra sull'evoluzione del linguaggio Prolog nel corso degli anni. Sin dalla sua ideazione negli anni 70', Prolog ha subito notevoli sviluppi, adattamenti e miglioramenti per adeguarsi alle esigenze deglla comunità.

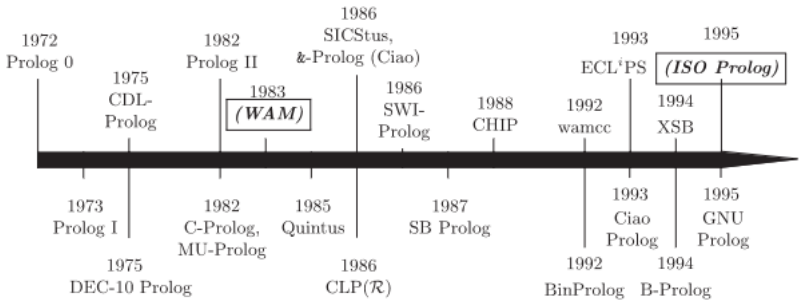

Figura 2.1: Evoluzione di Prolog [2]

Di seguito vedremo una panoramica delle principali tappe che hanno contrassegnato l'evoluzione di Prolog (vedi figura 2.1). Esploreremo le innovazioni chiave e i contributi accademici che hanno, negli anni, arricchito il linguaggio. Vedremo poi le varie varianti di Prolog che sono state sviluppate nel corso degli anni, e come esse abbiano influenzato il linguaggio principale.

Prolog fa parte principalmente di tre rami della ricerca: intelligenza artificiale, risoluzione automatica dei teoremi e processamento del linguaggio.

Il campo dell'intelligenza artificiale naque all'incirca nel 1956 e diede velocemente vita al linguaggio di programmazione funzionale LISP. Da li in poi l'interesse aumentò e diede vita ad altri linguaggi più o meno ad alto livello.

La soluzione automatica di teoremi è un campo della matematica che si occupa di trovare dimostrazioni automatiche di teoremi matematici. La risoluzione può essere utilizzata per ottenere una procedura semi-decisionale per la logica predicativa ed è al centro della maggior parte delle procedure di inferenza nella programmazione logica.

A seguito di questi progressi, un visionario precoce nello sviluppo del campo della programmazione logica è stato Cordall Green, che già alla fine degli anni 60 immaginava come estendere la risoluzione per costruire automaticamente la soluzione di un problema. In particolare voleva applicare questo strumento per la risoluzione di domande basate sulla logica di primordine. Questo rappresentò il primo zenit della programmazione logica impiegata in applicazioni di AI.

Un altro progetto notabile è quello di Ted Elcock che sviluppò Absys, un linguaggio di programmazione dichiarativo che anticipò alcune delle funzioni di Prolog come la negazione come fallimento, operatori di aggregazione e il backtracking.

Nel frattempo Alain Colmerauer stava provando a sviluppare un sistema di conversazione uomo-macchina, ciò lo portò a sviluppare Q-systems, un sistema impegato per le traduzioni da Inglese a Francese dei rapporti meteorologici Canadesi. Il suo obiettivo di modificare Q-systems, per ottenere un sistema domanda e risposta, lo portò a sviluppare un linguaggio di programmazione basato sulla logica predicativa. Questo linguaggio fu chiamato Prolog, e fu sviluppato nel 1972. Il suo creatore, Alain Colmerauer, diede quindi vita a Prolog nel 1972. Il risultato fu non solo la prima applicazione del linguaggio naturale (NL) di quello che oggi conosciamo come Prolog, ma la base di Prolog stesso: un sistema di risoluzione lineare ristretto alle clausole di Horn che poteva risolvere problemi in maniera non deterministica.

L'articolo di Körner [2] fornisce una panoramica completa dell'evoluzione di Prolog, e di come questo linguaggio sia stato influenzato nel corso degli anni. Io mi concentrerò soltato su alcuni aspetti chiave che hanno portato Prolog a diventare il linguaggio che conosciamo oggi. Se siete interessati a una panoramica più completa seguendo una linea temporale come quella in figura 2.1 vi consiglio caldamente di leggere l'articolo.

2.2.1 Tappe fondamentali dell'evoluzione di Prolog

Nella sua prima decade di vita Prolog subì molte variazioni, la sua prima versione fu Prolog 0, seguita da Prolog 1. In queste versioni il linguaggio prevedeva già qualche predicato built in che poi fu incluso nello standard ISO. Queste versioni includevano già il predicato `dif/2`, questo predicato fu importante perchè fece prendere piede al concetto di unificazione. Successivamente seguì l'aggiunta della negazione come fallimento, questa fu una delle aggiunte più importanti, in quanto permise di risolvere problemi che non erano risolvibili con la sola unificazione.

Nel 1974 David H.D. Warren sviluppò Warplan, un sistema di pianificazione automatica basato su Prolog. Lo sviluppo di questo progetto, che poi divento la sua tesi di dottorato, lo portò a voler ottimizzare l'interprete Prolog, siccome ritenuto lento rispetto ad altri linguaggi di più alto livello come LISP. Sviluppò quindi un nuovo interprete, chiamato WAM (Warren Abstract Machine).

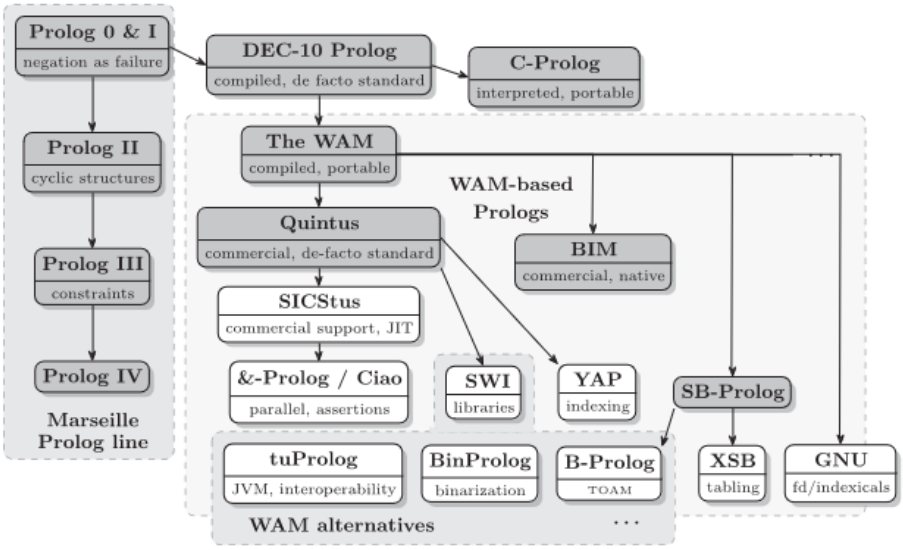

Figura 2.2: Quandro generale delle versioni di Prolog [2]

WAM era, ed è tutt'ora, lo standard per i compilatori Prolog. Due delle implemenatazioni più di successo sono *GNU Prolog* e quella utilizzata per il mio progetto *SWI-Prolog*. Entrambe queste implementazioni sono open source e disponibili online.

GNU Prolog è un compilatore Prolog gratuito e open source, è stato sviluppato da Daniel Diaz. È stato sviluppato per essere un compilatore Prolog standard, e quindi non include estensioni proprietarie. La sua implementazione utilizza WAM come base per produrre codice in C che verrà poi

compilato da GCC. C fu poi rimpiazzato da un linguaggio miniassembly specializzato perchè ritenuto più efficiente.

SWI-Prolog naque dalla mancanza di Quintus (un implementazione di Prolog) di chiamate ricorsive tra Prolog e C. La potenza di questa implementazione è la community che la supporta, infatti è una delle implementazioni più utilizzate al mondo sia per la ricerca che non. Oltre che la forte community SWI-Prolog, al contrario di GNU Prolog, è interpretato, e non compilato. Questo permette di avere un ambiente di sviluppo molto più flessibile e adatto alla ricerca.

2.3 Lavori notabili

3 Descrizione del caso di studio

4 Descrizione dell'architettura ROS

5 Conclusione

Bibliografia

[1] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[2] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V Hermenegildo, Jose F Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, et al. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, 22(6):776–858, 2022.

[3] Robert A Kowalski. Predicate logic as a programming language. *Proceedings of IFIP*, pages 569–574, 1974.

[4] Paul J. Krause. The art of prolog—second edition by leon sterling and ehud shapiro, mit press, cambridge, ma1994, pp. 509, £19.95 (paperback), £44.94 (hardback), isbn 0-262-19338-8. *The Knowledge Engineering Review*, 10(4):411–411, 1995.