Relazione

Progetto A.A. 2022/2023
ADAS made trivial: rappresentazione ispirata alle interazioni in un sistema di guida autonoma

Diciotti Matteo	Manucci Agostino	Montes Anacona Àlvaro
7072181	7084379	7117731

15 luglio 2023

Titolo Progetto A.A. 2022/2023 – ADAS made trivial: rappresentazione ispirata alle interazioni in un sistema di guida autonoma

Autori Lista degli autori ordinata per numero di matricola

Matricola	Cognome	Nome	e-mail
7072181	Diciotti	Matteo	matteo.diciotti@stud.unifi.it
7084379	Mannucci	Agostino	agostino.mannucci@stud.unifi.it
7117731	Montes Anacona	Álvaro	alvaro.montes@stud.unifi.it

Obiettivo Obiettivo del progetto è costruire un'architettura, estremamente stilizzata e rivisitata, per sistemi ADAS, descrivendo possibili interazioni e alcuni comportamenti tra componenti in scenari specifici.

Introduzione

Introduzione al progetto Un ***A**dvanced **D**river **A**ssistance **S**ystem* è un sistema composto da varie componenti che cooperano affinché un veicolo o un mezzo possano assistere un conducente nella guida e, in alcuni contesti specifici e limitati, sostituirsi al guidatore stesso.
Il progetto realizzato cerca di riprodurre il sistema simulativo di un ADAS presentato nella richiesta[1] dell'elaborato nella quale sono definite varie componenti suddivise in quattro gruppi: *interfaccia*, *attuatori*, *sensori* e *controllo*.
La simulazione è resa effettiva dall'inserimento di stringhe rappresentanti le azioni che dovrebbero essere eseguite dalle varie componenti di un ADAS in dei file di log specifici per ogni componente ed in un terminale adibito alla visualizzazione dei comandi inviati da parte dell'unità di controllo. In particolare le componenti implementate sono nove, sette obbligatorie e due facoltative[2].

[1] Per visionare la richiesta dell'elaborato riferirsi al documento Allegato_1.pdf compreso nella cartella del progetto
[2] Per conoscere quali elementi facoltativi sono stati implementati visionare la tabella a pagina 4

I sensori, attraverso l'acquisizione di dati dal file *urandomARTIFICIALE.binary* o dal dispositivo */dev/urandom*, utilizzato come sorgente binaria casuale, simulano l'acquisizione di dati da parte di sensori reali e li inviano all'unica componente di controllo, la componente *central-ECU*. Gli attuatori, invece, ricevono messaggi dall'unità di controllo ed eseguono scritture negli appositi file di log, simulando così l'azione di un attuatore reale. Infine, la componente con cui si interfaccia l'esecutore del programma è la Human-Machine Interface che simula l'interfaccia del guidatore attendendo da parte dallo stesso utente alcuni specifici comandi(in questo caso scritti su di un emulatore di terminale) e mostrando le conseguenze di questi ed alcuni risultati: nel contesto della simulazione saranno mostrati su di un secondo emulatore di terminale tutti i comandi che l'unita di controllo ha inviato alle varie componenti del sistema.

Impostazione del lavoro Al fine di realizzare un sistema che simulasse un ADAS sono state eseguite alcune fasi per il completamento del programma: l'analisi delle richieste, la strutturazione teorica dell'architettura del sistema, l'implementazione pratica, la verifica e revisione dell'elaborato.
Ad una prima fase di studio delle richieste congiunta tra gli autori del progetto è immediatamente succeduta la fase di strutturazione dello stesso, anch'essa svolta unitamente tra i relatori, giungendo alla teorizzazione dell'architettura mostrata in figura 1. Sono stati quindi divisi i lavori per l'implementazione del programma tra i componenti del gruppo e uniti gli elaborati in un unico componimento comprensivo di makefile e file binari per l'esecuzione in modalità *ARTIFICIALE* del programma. Sono state quindi eseguite ciclicamente le due fasi di verifica e di modifica: nella prima fase sono stati svolti test per verificare il corretto funzionamento dell'elaborato in entrambe le modalità e nella seconda fase sono state apportate le modifiche necessarie affinché il programma producesse i risultati attesi. Infine e' stata prodotta la presente relazione.

Specifiche

Caratteristiche HW e SW Per l'implementazione sono stati utilizzati tre hardware differenti con tre sistemi operativi differenti:

- **ArchLinux**, architettura x86_64, kernel Linux 6.1.34-1-LTS
- **Manjaro 23.0.0**, architettura x86_64, kernel Linux 5.15.114-2-MANJARO
- **Ubuntu 20.04**, architettura x86_64, kernel Linux 5.15.0-75-generic

Il progetto è stato ideato avendo come obbiettivo un programma portatile[3] che fosse preimpostato per eseguire su di un sistema **Ubuntu Linux 22.04**.

Istruzioni compilazione ed esecuzione Nel presente paragrafo si mostrano i passaggi necessari affinché si possa eseguire il programma sul proprio dispositivo:

Estrazione e compilazione

1. Estrarre il file .zip in una cartella a piacimento. (Esempi comando: `tar -x $PROJECT_NAME$; 7z x $PROJECT_NAME$`);
2. Spostarsi nella cartella creatasi con l'estrazione. (Esempio comando: `cd ./$PROJECT_NAME_FOLDER$`);
3. Eseguire i comandi di compilazione e installazione del programma: `make; make install`

Terminata questa procedura il programma dovrebbe risultare pronto per l'esecuzione.

[3]NOTA SULLA PORTABILITÀ: è stato preso come riferimento per la portabilità del sistema lo standard **_POSIX_C_SOURCE 200809L**, il quale garantisce un'ottimo grado si portabilita su sistemi *UNIX* e *UNIX-like*. È altresì necessario notare però che l'utilizzo di alcune funzioni e system-callS, come signal(2) la quale è sconsigliata dai manuali *POSIX* per questioni di portabilità, e l'utilizzo della sorgente /dev/urandom potrebbero limitare lo spettro di piattaforme su cui eseguire il programma ai "soli" sistemi *UNIX-like*. Dato il contesto accademico, è stato comunque preferito l'utilizzo delle funzioni suddette per non discostarsi eccessivamente degli argomenti del corso.

Esecuzione

1. Assicurarsi di essere con un terminale nella directory sorgente del programma, la stessa contenente le cartelle bin, src, include etc. e il file eseguibile *ADAS-simulator*.

2. Avviare l'inizializzazione e l'esecuzione del programma tramite la chiamata di `./ADAS-simulator $MODALITÀ$`, dove `$MODALITÀ$` rappresenta la modalità `ARTIFICIALE` o `NORMALE` del programma[4]. Oltre al secondo argomento (il quale risulta obbligatorio, pena la mancata esecuzione del programma e la restituzione di un errore di sintassi), il comando supporta anche l'opzione `--term` nella quale si può specificare l'emulatore di terminale che si desidera utilizzare come terminale di output. L'emulatore impostato di default è *gnome-teminal*. Per utilizzarne uno differente inserire il comando di shell che apre un nuovo terminale della tipologia scelta (Esempi: `./ADAS-simulator ARTIFICIALE --term konsole` oppure `./ADAS-simulator ARTIFICIALE --term xfce4-terminal`)

3. Eseguito il comando dovrebbe aprirsi automaticamente un nuovo terminale rappresentante il terminale di output. Non eseguire alcun comando sul terminale di output e posizionarsi con il cursore sul terminale di input, ovvero il terminale nel quale è stato eseguito il comando di avvio del programma.

4. Il programma è inizializzato: si possono ora eseguire i comandi INIZIO, PARCHEGGIO per iniziare l'esecuzione del programma e successivamente ARRESTO per simulare un arresto del veicolo[5].

Elementi facoltativi La tabella a pagina 4 mostra gli elementi facoltativi implementati seguiti da una breve descrizione dell'implementazione.

[4]Per la comprensione delle differenze delle modalita di esecuzione riferirsi al paragrafo 4 della richiesta *Allegato_01.pdf*
[5]Per la comprensione dei comandi inseribili dal terminale rifarsi al paragrafo 2, sotto-paragrafo *Human-Machine Interface*, della richiesta *Allegato_01.pdf*

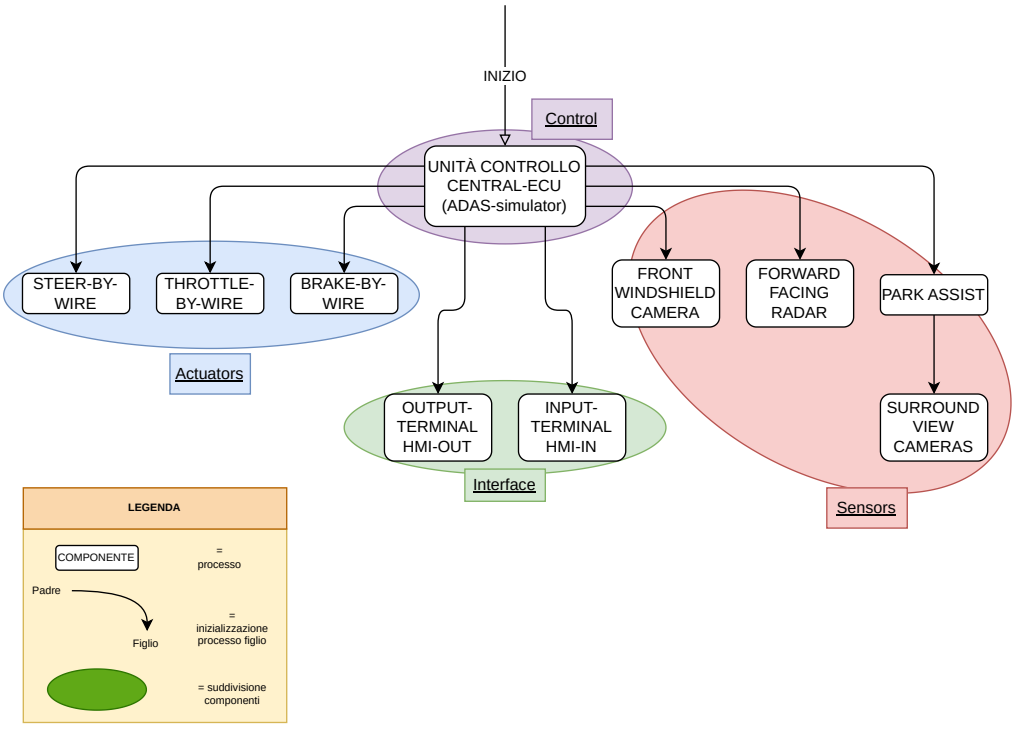

Figura 1: In figura è mostrata la gerarchia implementata nel programma. La *central-ECU* rappresenta il processo antenato di tutte gli altri processi, *park assist* risulta genitore dei *surround view cameras*.

#	Elemento facoltativo	Realizzato (SI/NO)	Descrizione dell'implementazione con indicazione del metodo/i principale/i
1	Ad ogni accelerazione, c'è una probabilità di 10^{-5} che l'acceleratore fallisca. In tal caso, il componente *throttle control* invia un segnale alla *central-ECU* per evidenziare tale evento, e la central-ECU avvia la procedura di *ARRESTO*	SI	Metodo: *throttle-control* $\rightarrow$ *throttle_failed()*. Il metodo, tramite la funzione `rand()` della `libc` ottiene un numero aleatorio il cui modulo per 10000 simula una probabilità del 1 su 10^5 se eguagliato a 0[a]
2	Componente *forward facing radar*	SI	Sorgente *bytes-sensors.c*. Il processo esegue un ciclo infinito di letture, invii e scritture su file di log.
3	Quando si attiva l'interazione con park assist, la *central-ECU* sospende (o rimuove) tutti i sensori e attuatori, tranne *park assist* e *surround view cameras*	SI	Nella *central-ECU* vengono segnalati con un `SIGKILL` tutti i processi attuatori e sensori esistenti prima di procedere all'inizializzazione del parcheggio.
4	Il componente *park assist* non è generato all'avvio del sistema, ma creato dalla *central-ECU* al bisogno	SI	La *central-ECU* esegue la `fork` per la creazione di *park-assist* in *park_assist_init()*, il quale inizializza il processo prima di entrare nel ciclo di parcheggio.
5	Se il componente *surround view cameras* è implementato, *park assist* trasmette a *central-ECU* anche i byte ricevuti da *surround view cameras*	SI	Nel ciclo principale di *park-assit* si esegue una `read` da *cameras.pipe* (non bloccante) e una `write` dei dati ricevuti sulla socket *assist.sock*
6	Componente *surround view cameras*	SI	Sorgente *bytes-sensors.c*. Vedi facoltativo 2 - *forward facing radar*
7	Il comando di *PARCHEGGIO* potrebbe arrivare mentre i vari attuatori stanno eseguendo ulteriori comandi (accelerare o sterzare). I vari attuatori interrompono le loro azioni, per avviare le procedure di parcheggio	SI	Nella *central-ECU*, nel ciclo principale, dopo la lettura da hmi-input, *kill(-processes_groups.actuators_group, SIGKILL)*. Si esegue la segnalazione di chiusura immediata dei processi *steer* e *throttle* (brake rimane per il ciclo di frenata).
8	Se la *central-ECU* riceve il segnale di fallimento accelerazione da *throttle control*, imposta la velocità a 0 e invia all'output della *HMI* un messaggio di totale terminazione dell'esecuzione	SI	Nella *central-ECU*, una volta ricevuto il segnale di *fallimento accelerazione*, gestito tramite *ECU_signal_handler* si esegue la procedura di arresto, stampa la stringa di terminazione e si conclude l'esecuzione dei processi.

[a] La probabilità non risulta esattamente 10^{-5} dato l'intervallo di valori producibili da `rand()`, ma la differenza risulta trascurabile ai fini del progetto.

Descrizione architettura sistema

Nella seguente sezione viene presentata l'architettura del progetto e le scelte implementative prese, cercando di descrivere i motivi che hanno portato alle singole decisioni prese.

Implementazione componenti Segue una tabella nella quale si mostrano quali sorgenti implementano le componenti

Componente	Sorgente
Human-Machine Interface	hmi-input.c hmi-output.c
steer-by-wire	steer-by-wire.c
throttle control	throttle-control.c
brake-by-wire	brake-by-wire.c
front windshield camera	windshield-camera.c
forward facing radar	bytes-sensors.c
park assist	park-assist.c
surround view cameras	bytes-sensors.c
central-ECU	central-ECU.c

Gerarchia del programma La gerarchia del progetto è stata ottenuta dalla richiesta cercando di massimizzare la semplicità. Questa rappresenta lo scheletro del progetto e definisce quindi il flusso di lavoro dello stesso. In particolare si può notare in figura 1 che esistono due soli processi che inizializzano dei figli, ovvero l'unita di controllo centrale *central-ECU*, che rappresenta anche il programma genitore di tutto il sistema, e *park-assist*, che inizializza il suo unico figlio *surround-view cameras*.
È stato scelto di rendere l'unità di controllo la componente genitore di tutto il sistema dato che è in collegamento con la quasi totalità dei processi agenti, cosicché il sistema fosse inizializzato e gestito nel flusso di questa. La soluzione sembrava essere coerente con la strutturazione dei sistemi ADAS reali.
Nella *central-ECU* vengono quindi inizializzati le pipe, vengono eseguite varie `fork` per la creazione e la connessione in lettura/scrittura ai pipe degli attuatori, delle due componenti relative alla *Human-Machine Interface* ed infine i due sensori *front windshield camera* e *forward facing radar*. L'unico componente figlio della *central-ECU* non immediatamente inizializzato rimane *park-assist*[6] il quale verrà inizializzato (comprensivo della socket che utilizzerà per comunicare con la *central-ECU*) e messo in esecuzione dalla stessa unità di controllo quando questa riceverà un comando di *PARCHEGGIO* dall'interfaccia oppure dal sensore *windshield camera*.
Con le stesse motivazioni è stato deciso di rendere la *central-ECU* "server" nella connessione socket tra questa e *park assist*. In questo modo sarà sempre la central-ECU a gestire il flusso di lavoro e a dettare i tempi del sistema. All'avvio della procedura di parcheggio, successivamente al ciclo di rallentamento, la *central-ECU* avvierà la componente *park-assist*, la quale si connetterà alla socket e inizializzerà *surround-view cameras*. Terminata l'inizializzazione, l'unita di controllo invierà un messaggio ("INIZIO") tramite la socket a *park assist*, la quale leggerà per 30 iterazioni (intervallate da `sleep(1)`) i dati dalla sorgente binaria e i dati ricevuti dalla componente figlia tramite la pipe di comunicazione e li invierà alla *central-ECU* che li scandirà alla ricerca di particolari pattern[7]. Se non venissero evidenziate congruenze allora la central-ECU informerà tramite un messaggio ("CONTINUA") *park assist* perché possa

[6] La decisione di non inizializzare il componente immediatamente deriva dalla richiesta, ovvero dall'elemento facoltativo numero 4. Vedi la tabella di pagina 4.

[7] Per conoscere i pattern binari (espressi in codifica esadecimale) prendere visione del paragrafo 2, sotto-paragrafo *Componente central-ECU* della richiesta *Allegato_01.pdf*

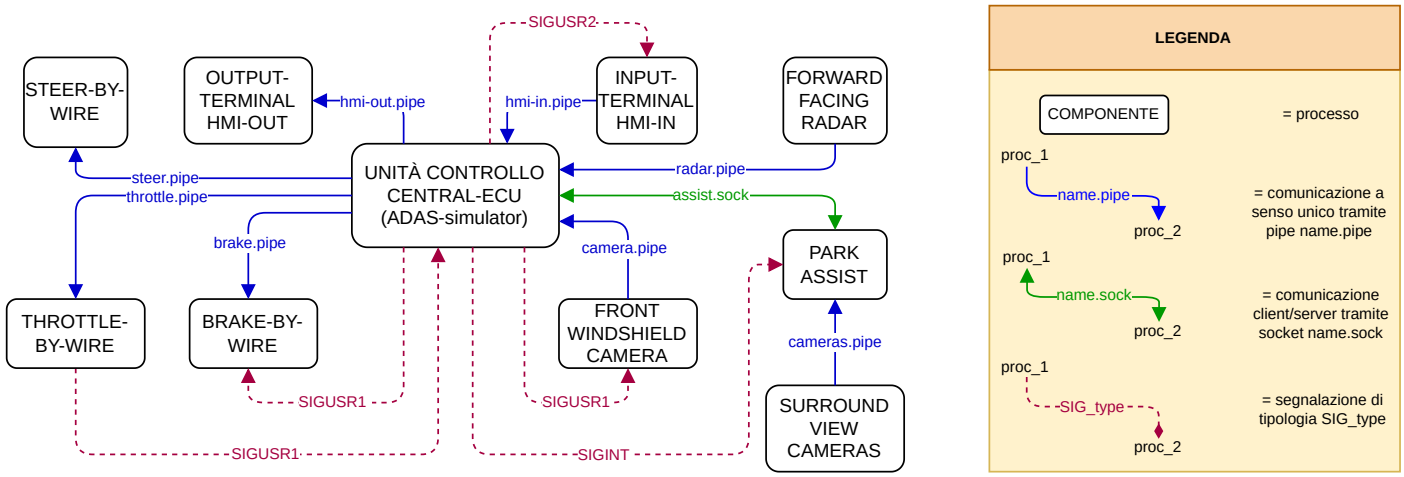

Figura 2: In figura sono rappresentati i percorsi comunicativi inseriti nel sistema: sono presenti 8 pipe, 1 socket e 12 di segnalazioni, 5 rappresentate e 7 assenti. Sono infatti stati esclusi i percorsi delle segnalazioni `SIGKILL` per alleggerire il grafico.

proseguire nell'iterazione successiva. Se così non fosse allora verrà posto sulla socket un messaggio di riavvio ("RIAVVIO"), affinché il parcheggio inizi nuovamente il ciclo delle 30 iterazioni. Concluse le 30 iterazioni senza "insuccessi" allora l'esecuzione del parcheggio risulterà conclusa e il processo terminerà.

IPC nel sistema In figura 2, a pagina 6, si mostra una schematizzazione molto stilizzata della rete comunicativa del sistema. La struttura di comunicazione maggiormente sfruttata all'interno del sistema è la **pipe** la quale implementa 8 canali di comunicazione su 9 (segnali esclusi). Il motivo della scelta delle **pipe** a discapito di altri metodi risiede nel fatto che le comunicazioni sono sostanzialmente unidirezionali (sensori $\rightarrow$ unità di controllo, unità di controllo $\rightarrow$ attuatori, con le due evidenti eccezioni di *park assist* e di *surround-view cameras*). La scelta ha permesso di implementare un sistema relativamente semplice, con un'unica **socket**, struttura più complessa da implementare.
Il protocollo di gestione dei canali risulta unico per tutte le tipologie di canali e per tutti i processi: il processo padre, l'unico processo che comunica con i propri processi figli, produce ed inizializza correttamente il file `.pipe` o `.sock` (rappresentante socket `UNIX`) nella directory *tmp* (dopo aver avuto l'accortezza di eliminare creazioni pendenti da vecchie esecuzioni tramite un `unlink`), il processo figlio si connette con l'adeguata procedura, differente tra pipe e socket, al canale di comunicazione nella fase inizializzazione del processo. Se la connessione non dovesse riuscire (se la `openat` o la `connect` dovessero generare un errore) il figlio terminerebbe con codice di errore `EXIT_FAILURE` e stamperebbe il codice tramite `perror(3)` nel file di log adibito a *stderr*: `./log/errors.log`
Per l'implementazione del parcheggio il canale di comunicazione è stato strutturato sotto forma di una socket per semplificare la comunicazione tra i due processi e perché la gestione del contenuto della socket fosse meno soggetto a problemi di concorrenza. Ogni volta che la *central-ECU* riceve dalla socket uno dei pattern non ammissibili, che simulano una situazione di parcheggio non accettabile, il protocollo prevede che invii un messaggio di riavvio a *park assist*. Il protocollo prevede che ad ogni ciclo *park assist* legga dalla pipe non bloccante *cameras.pipe* i byte da inviare alla *central-ECU* e che li inoltri. Se, eseguito l'invio, riceve dalla *central-ECU* un messaggio che indica la necessità di riavvio, allora non fa altro che resettare il contatore di iterazioni e riprendere il ciclo.

Gestione dei log I log rappresentano, nel sistema implementato, la simulazione delle componenti reali, ovvero mostrano le azioni eseguite dalle varie componenti.
Per ogni componente, esclusa la *Human-Machine Interface*, è presente un file di log specifico nel quale vengono scritte le azioni eseguite. Gli inserimenti nel file di log seguono le richieste specificate nella richiesta del progetto *Allegato_1.pdf*. Perché le shell di input e di output risultassero pulite da ogni

eventuale segnalazione diretta allo *stderr* è stato adottato il file `./log/errors.log` come soluzione per inserirvi tutti i messaggi dirette a questo.

Funzioni condivise Durante lo sviluppo del sistema sono risultate utili la definizione di alcune funzioni di utilità generale per il sistema e di alcune macros, anch'esse sfruttate in più contesti. Tutto ciò è stato inserito all'interno del sorgente `service-functions.c`, il cui header risulta quindi `service-functions.h`.

Esempi di esecuzione

Nella seguente sezione sono mostrati 2 esempi di funzionamento: il primo mostra un funzionamento in modalità ARTIFICIALE con comando dato da input "INIZIO" e alcuni "ARRESTO", nel secondo si esegue in modalità NORMALE direttamente il parcheggio con un terminale selezionato. All'interno dello `.zip` è stata inserita una cartella *Esempi di funzionamento* nella quale sono stati inseriti i file di log risultanti dalle esecuzioni presentate.

Esempio: ARTIFICIALE standard Il presente esempio è teso a mostrare un'esecuzione tipica del programma. Il programma è stato eseguito tramite il comando `./ADAS-simulator ARTIFICIALE`. Non appena il programma si è inizializzato è stato inserito il comando "INIZIO". Il programma si è avviato correttamente iniziando ad inserire sul terminale di output i comandi impartiti da parte della *central-ECU* alle rispettive componenti. Sono stati inseriti, durante l'esecuzione, quattro comandi di "ARRESTO". Il programma ha eseguito correttamente la procedura di arresto, bloccando istantaneamente l'auto e ripartendo la corsa istantaneamente. La procedura di parcheggio, ultimo comando da parte della *windshield camera*, è terminata dopo 30 iterazioni (30 secondi).

Esempio: NORMALE terminale personalizzato Il presente esempio è teso a mostrare un'esecuzione del programma relativamente atipica e potenzialmente problematica, un'esecuzione nella quale l'utente inserisce inizialmente una stringa non accettabile, poi il comando "ARRESTO", sebbene il mezzo non sia ancora in movimento ed infine il comando "PARCHEGGIO". Il programma è stato eseguito tramite il comando `./ADAS-simulator NORMALE -term xfce4-terminal`. Non appena il programma si è inizializzato è stato inserito come comando "42", che è risultato non accettabile, poi "ARRESTO", anch'esso non accettabile, ed infine "parcheggio" che ha dato il via alla procedura di parcheggio, terminata con successo dopo un ciclo di iterazioni. È stata infatti implementata una funzione per il confronto tra stringhe case-insensitive per i comandi dalla hmi-input.

Indice