ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Alma Mater Studiorum · Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Testing automatizzato Hardware-In-the-Loop:

APPLICAZIONE IN AMBITO MOTORSPORT

Relatore:
Chiar.mo Prof.
Mirko Viroli

Presentata da:
Andrea Serafini

IV^a Sessione
Anno Accademico 2023/2024

Grateful for all those
whose journeys overlapped mine;
You have made me, me.

Abstract

Il settore del motorsport richiede tecnologie avanzate per garantire affidabilità e sicurezza nei processi di sviluppo. Questa tesi affronta il tema del testing automatizzato Hardware-In-the-Loop applicato al mondo delle corse motociclistiche. L'obiettivo preposto è quello di progettare un sistema software per la validazione di software e componenti elettroniche attraverso test automatizzati.

Il progetto si colloca nel contesto di Ducati Corse, sfruttando tecnologie avanzate per migliorare l'affidabilità dei sistemi elettronici delle moto da loro sviluppate. Il sistema sviluppato integrerà strumenti come LabVIEW e TestStand, si baserà su linguaggi come Python e .NET, articolandosi in quattro moduli: un validatore di test, un generatore di codice, un esecutore multithread e un sistema di monitoraggio basato su OCR per il riconoscimento automatico degli allarmi.

Uno degli aspetti innovativi e più interessanti si è rivelato essere l'uso di algoritmi OCR per il rilevamento automatico degli allarmi visualizzati sui display di bordo. L'architettura software, configurabile e scalabile, permette agli ingegneri di adattare i test alle specifiche esigenze.

I risultati che questo sistema si pone di ottenere sono un miglioramento nella velocità e nella precisione dei test, riducendo i tempi di validazione e minimizzando l'intervento manuale e la conseguente possibilità di errore. L'automazione di questo processo rappresenterebbe quindi un passo in avanti nelle pratiche di validazione attualmente utilizzate per le ECU. Infine, vengono discusse possibili evoluzioni, come l'integrazione di tecniche di machine learning per l'analisi predittiva dei guasti e l'estensione del linguaggio di test per coprire più casistiche.

Questa tesi si propone quindi di fornire un sistema completo, innovativo e modulare, adattabile alle esigenze di un settore in continua evoluzione.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei sorgenti

Acronimi

API Application Programming Interface.

BMS Battery Management System.

BoB Breakout Box.

CLR Common Language Runtime.

DSL Domain-Specific Language.

DUT Device Under Test.

ECU Electronic Control Unit.

GUI Graphical User Interface.

HIL Hardware-In-the-Loop.

HW hardware.

I/O Input/Output.

MBD Model Base Design.

MIL Model-In-the-Loop.

NI National Instruments.

PIL Processor-In-the-Loop.

SIL Software-In-the-Loop.

SW software.

UTF Universal-Coded-Character-Set Transformation Format.

UUT Unit Under Test.

VBA Visual Basic for Applications.

VCU Vehicle Control Unit.

VI Virtual Instrument.

VIs Virtual Instruments.

W3C World Wide Web Consortium.

XML eXtensible Markup Language.

Introduzione

Il mondo del motorsport, nel particolare quello motociclistico, impone elevati standard di affidabilità e sicurezza per i sistemi elettronici di controllo del veicolo. La complessità crescente delle centraline elettroniche, dei sensori e dei dispositivi di controllo ha reso necessaria nel tempo l'adozione di metodologie di testing rigorose e altamente specializzate. Il testing Hardware-In-the-Loop delle suddette centraline rappresenta una delle soluzioni più efficaci per la validazione di questi sistemi elettronici, consentendo di simulare scenari realistici in un ambiente controllato prima dell'implementazione su veicolo con notevoli vantaggi. Tra questi vantaggi verranno in seguito illustrati:

- rapidità di implementazione
- elasticità ai cambiamenti
- costi contenuti
- determinismo nei test

Questa tesi ha origine da un progetto che si pone l'obiettivo di sviluppare un sistema automatizzato per il testing HIL, nato dalla necessità di ridurre il tempo richiesto per la definizione ed esecuzione dei test, oltre che da quella di creare un meccanismo che garantisca ripetibilità e storico dei test stessi. Il progetto si è sviluppato quindi procedendo con la realizzazione di un'infrastruttura software denominata *HIL Test Automator*, articolata in quattro moduli principali:

- **Validator**: verifica la correttezza dei test definiti dagli strateghi in linguaggio naturale nel formato Excel
- **Generator**: traduce le specifiche dei test in codice eseguibile per il sistema HIL
- **Executor**: esegue i test sul banco HIL, gestendo inoltre l'acquisizione dei risultati
- **Watcher**: monitora in tempo reale gli allarmi e lo stato del sistema attraverso tecniche di riconoscimento ottico dei caratteri e analisi dei LED

Il sistema sviluppato si basa su e si integra con tecnologie come LabVIEW, TestStand, Python e .NET, permettendo una gestione flessibile e scalabile del processo di testing. Per l'implementazione è stata inoltre proposta un'architettura modulare che

consentisse future estensioni e personalizzazioni in base alle necessità dei team di sviluppo e testing.

Nei capitoli seguenti questo elaborato di tesi si presenterà il lavoro svolto in azienda, senza trascurare i dettagli di ricerca e approfondimento. La struttura della tesi è organizzata come segue:

- Capitolo 1: **Contesto**
  In questo primo capitolo verrà fornita un'introduzione di base al contesto aziendale all'interno del quale si è sviluppato il progetto, introducendo il lettore ad alcune particolarità del mondo delle competizioni motociclistiche, fino ad arrivare alla necessità di sistemi di testing avanzati.

- Capitolo 2: **Background**
  Nel secondo capitolo si procederà invece ad approfondire maggiormente il lato tecnico delle conoscenze necessarie per comprendere a pieno il prgetto. Verrà quindi fornita al lettore una panoramica sulle tecnologie HIL e sugli strumenti software utilizzati per l'interazione con il sistema. Questa sezione corrisponde principalmente alle fasi di studio e ricerca di materiali e documentazione svolte durante il progetto.

- Capitolo 3: **HIL Test Automator**
  Nel terzo capitolo verrà descritta la parte attuativa del progetto, fornendo una descrizione più dettagliata del sistema sviluppato e delle sue componenti. Saranno analizzate le scelte tecnologiche effettuate sulla base di necessità e vantaggi, cercando di non tralasciare le alternative che non sono state portate avanti fino al termine dello sviluppo per permettere in caso di ulteriori sviluppi futuri di analizzarle nuovamente e verificarne la validità.

- Capitolo 4: **Integrazione e distribuzione**
  I due argomenti trattati infine nel quarto capitolo fanno riferimento alle attività di integrazione e distribuzione che hanno caratterizzato la parte finale del progetto. I quattro moduli citati in precedenza sono stati sviluppati in maniera autonoma al fine di mantenere una struttura appunto più modulare possibile, che permettesse di manutenere, estendere e specializzare il sistema in maniera più facile. Per questo motivo si è resa necessaria la corretta definizione di come questi moduli avrebbero dovuto interfacciarsi e comunicare l'uno con l'altro, andando così a definire la loro integrazione. Verrà poi analizzato il processo seguito per la distribuzione del sistema all'interno del flusso di lavoro aziendale, rendendolo fruibile per gli utenti.

- Capitolo 5: **Simulazione di utilizzo**
  In quest'ultimo capitolo verrà mostrata infine una simulazione di utilizzo dell'intero sistema, permettendo al lettore di avere un contatto maggiore con la realtà del sistema sviluppato, mostrandone le funzionalità specifiche di ogni modulo ma proponendo l'idea del processo completo che si era prefissato il progetto.

Without data, you're just another person with an opinion.

W. Edwards Deming

Capitolo 1

Contesto

1.1 Azienda ospitante

Il progetto al centro di questo elaborato di tesi nasce all'interno di Ducati. La Ducati Motor Holding S.p.A.[Duca] è oggi la più nota azienda motociclistica italiana, fondata nel 1926 per volontà dell'ingegnere Antonio Cavalieri Ducati, era tuttavia denominata inizialmente *Società Scientifica Radio Brevetti Ducati* ed era appunto specializzata nella ricerca e produzione di tecnologie per le comunicazioni radio. Fu grazie alle capacità di Adriano, figlio di Antonio, e i suoi due fratelli Bruno e Marcello[Man], che l'azienda riuscì a imporsi sul mercato delle trasmissioni radiofoniche sfruttando i suoi brevetti nel mondo industriale. Seguendo il suo percorso di crescita la Ducati comincia ad ampliare sia il suo stabilimento che la sua offerta in quanto a prodotti commercializzati, finché anche lei come tante altre aziende italiane si vide costretta durante la seconda guerra mondiale a convertire la sua produzione da civile a militare.

Fu solo in seguito alla richiesta da parte dell'Istituto per la Ricostruzione Industriale nell'immediato dopoguerra che venne fondato il reparto motociclistico come branca dell'azienda, con l'obiettivo di produrre su licenza il *Cucciolo*, un motore monocilindrico di 48 cc con cambio a due velocità da applicare come propulsore ausiliario a una normale bicicletta.[Wika]

Fu proprio da questo piccolo antenato unito alla passione per la velocità che caratterizza la Motor Valley che si sviluppò poi quella che oggi è la famosa casa produttrice Ducati, culla di grandi innovazioni tecnologiche, numerosissimi modelli di moto diversi progettati e venduti in tutto il mondo, con numeri che sfiorano le 60.000 unità nell'anno 2023.

1.1.1 Ducati Corse

La storia di Ducati nelle competizioni su pista inizia negli anni 50 quando l'ingegnere Fabio Taglioni fece il suo ingresso nell'azienda. Da allora le rosse di Borgo Panigale hanno partecipato a svariate competizioni. Le presenze più importanti si dividono in due gruppi principali, quello delle derivate di serie e quello dei prototipi.

La Ducati partecipa infatti al campionato *Superbike* dal 1988, anno in cui si svolse la sua prima edizione, e che rimane tutt'oggi la categoria nella quale conta il maggior numero di successi. Il campionato mondiale *Superbike*, insieme a quelli di *Superstock* e *Supersport* fanno parte del primo gruppo di competizioni, nel quale vengono impiegate solo moto derivate di serie, ovvero basate su modelli venduti sul mercato.

Per quanto riguarda invece lo sviluppo di prototipi, dal 2003 il costruttore entra poi anche nel campionato mondiale *MotoGP*, la competizione più importante nel mondo delle due ruote su pista. Oggi il reparto corse della Ducati è composto da oltre 100 dipendenti, impegnati nello sviluppo delle moto da corsa più veloci del mondo. Quasi ogni fine settimana dell'anno alcune moto marchiate Ducati gareggiano in tutto il mondo in vari campionati, dalla *Superbike* alla *MotoGP*, dai campionati nazionali alle gare di durata, cercando di battere nuovi record e di superare il concetto di velocità. Dal 2023 è iniziata anche una nuova sfida per Ducati e Ducati Corse, infatti la casa di Borgo Panigale sarà l'unico costruttore del campionato mondiale *MotoE*, fornendo quasi 20 moto da corsa elettriche ai team clienti.

1.1.2 MotoE

La *MotoE World Cup* è un campionato motociclistico organizzato dalla FIM, ovvero la *Fédération Internationale de Motocyclisme*, rivolto esclusivamente alle moto elettriche. Dalla stagione inaugurale tenuta nel 2019 ogni anno quasi 20 piloti si sfidano nello stesso fine settimana di gara del calendario europeo del corrispettivo campionato con motori termici. Dalla collaborazione tra Ducati e Ducati Corse nasce così il prototipo *V21L* presentato al pubblico nel 2022 e successivamente utilizzato in questa competizione. [Ducb]

Durante lo sviluppo del progetto a tema di questo elaborato di tesi sono stati principalmente sfruttati gli strumenti hardware (HW) utilizzati nello sviluppo del modello V21L, nonché il suo stesso software (SW) per validare gli strumenti di test.

Figura 1.1: Prototipo Ducati MotoE

1.2 Testing

Risulta a questo punto importante anche sottolineare come nel mondo degli sport motoristici per ogni componente esista un numero inferiore di unità rispetto alla produzione industriale dello stesso, come i sistemi elettronici in questo campo tendano a essere più vicini a prototipi che a prodotti di serie, con l'obiettivo di raggiungere le massime prestazioni possibili, e come le rigide scadenze imposte dal calendario del campionato non consentano una tempistica flessibile nella progettazione e nello sviluppo del sistema.

Anche per questi motivi al giorno d'oggi l'industria dell'*automotive* fa largo uso del *V-model*, o modello di sviluppo a V, un modello di sviluppo software che estende quello a cascata.

Ciò che caratterizza questo approccio è che il ciclo di vita del SW viene suddiviso in due processi separati, a differenza di quello singolo e lineare del suo modello genitore, che possono e devono interagire l'uno con l'altro durante tutta la durata di un eventuale progetto. Come si può vedere in figura 1.2 il ramo di sinistra comprende le attività di definizione delle specifiche, progettazione e sviluppo, mentre il ramo di destra comprende i processi di integrazione, verifica e validazione.

Un'altra possibile distinzione all'interno di questo ciclo di vita può essere fatta considerando il SW in tre diverse fasi[Jha10]:

- **Unit test**: l'ambiente di test deve simulare i valori di input a livello della più piccola unità verificabile e controllarne l'output corrispondente. Questi test vengono eseguiti con la granularità più fine, dove la più piccola parte funzionale di un codice, ovvero una *unità*, viene sottoposta a dei test allo scopo di assicurarsi che funzioni secondo i requisiti del sistema.
- **Test di integrazione**: la seconda fase consiste invece nel verificare il funzionamento di più unità insieme all'interno del loro ambiente di lavoro, focalizzandosi sulla loro interazione. Risulta inoltre importante verificare che i flussi di dati tra i vari moduli soddisfino le specifiche definite a livello funzionale.
- **Test di sistema**: quest'ultima fase consiste nel testare l'intero sistema dopo avervi incluso le singole unità testate in precedenza. Gli aspetti importanti da tenere in considerazione sono l'intercomunicazione tra le singole componenti e il loro impatto complessivo sul comportamento del sistema.

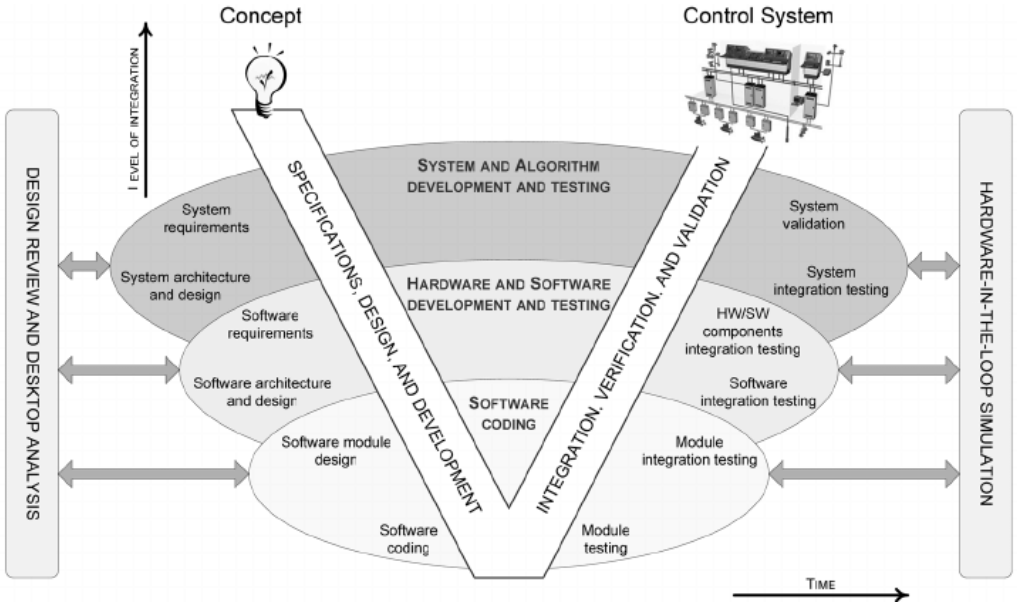


Figura 1.2: Modello di sviluppo a V [SE06]

Uno dei metodi principali utilizzati nel campo dell'informatica per verificare la correttezza di un'implementazione è la simulazione. Come si può osservare nella figura 1.3, nel modello a V il processo di simulazione può essere suddiviso in quattro diverse parti:

- Model-In-the-Loop (MIL)
- Software-In-the-Loop (SIL)
- Processor-In-the-Loop (PIL)
- Hardware-In-the-Loop (HIL)

Il primo elemento sottoposto alla simulazione è il modello, ovvero la descrizione più astratta del sistema. Successivamente si includerà nel processo della simulazione anche il software in esecuzione sul dispositivo specifico e i processori, o FPGA a seconda della specifica struttura del sistema. Infine, ove possibile, si espanderà il sistema aggiungendo l'hardware che si occuperà di eseguire il software utilizzando una combinazione di componenti reali e simulati al fine di svolgere funzioni *real-time*.

Quest'ultimo è forse fra tutti lo step più importante per la validazione finale del prodotto sviluppato. Ne deriva che in ambiti nei quali viene richiesto un livello di sicurezza più elevato, come quelli di *automotive* e *aerospace*, la simulazione HIL sia diventata parte fondamentale delle strategie di sviluppo e controllo dei test, aumentando la qualità dei test e riducendo tempi e costi di sviluppo. [Jos19]

Nello specifico caso del motorsport trattato in questa tesi si può considerare lo sviluppo di un HIL relativamente definitivo. Una volta convalidato correttamente l'hardware, è possibile che la stessa architettura venga riutilizzata per più di una stagione. I principali cambiamenti tra una Electronic Control Unit (ECU) sviluppata per la nuova stagione basandosi su quella precedentemente in uso, o ancora

su più breve termine tra due gare a distanza di una sola settimana, avvengono per la maggior parte nel software. E proprio per queste ultime modifiche apportate durante la stagione che influenzano spesso solo piccole parti dell'intero software, si evidenzia la necessità di verificare e testare le nuove versione attraverso una procedura specifica. Per ottenere questo risultato si effettueranno prima di tutto dei test di non regressione per evidenziare eventuali problemi causati dall'integrazione del codice modificato con l'intero sistema. Successivamente si eseguiranno una serie di *unit test*, attraverso un processo che ne prevede la pianificazione da parte dell'ingegnere di strategia, comunemente chiamato stratega, l'esecuzione e la verifica della correttezza dei risultati ottenuti confrontandoli con i requisiti attesi [IEE86].

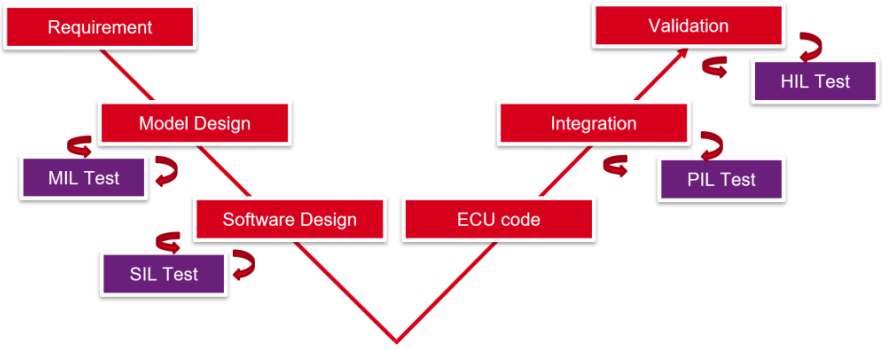


Figura 1.3: Modello di sviluppo a V [Hoa]

1.3 Hardware-In-the-Loop

Il test HIL si occupa di mettere in comunicazione un controller a un sistema di test che simula il funzionamento del prodotto finale in condizioni reali. Questo approccio consente di eseguire test e iterazioni di progettazione come se i dispositivi fossero in funzione sul campo con tutti i componenti. Diventa così possibile eseguire facilmente migliaia di scenari ipotetici per sfruttare al massimo il controller evitando i costi e i tempi associati ai corrispettivi test fisici.

Come detto in precedenza, questo tipo di test risulta utile anche grazie alla possibilità che offre di testare nuovi componenti, definiti Device Under Test (DUT), indipendentemente da quale sia il livello di progettazione e costruzione di un sistema, risultando particolarmente indicato per ambiti in cui i sistemi evolvono in maniera non lineare e mantenendo una grande volatilità.

Figura 1.4: Sistema sotto test [NIf]

Facendo riferimento alla figura 1.4 si può prendere in considerazione un'automobile e la ECU del suo motore. La ECU sarà responsabile ad esempio della conversione di valori letti dai sensori in azioni, come la regolazione della quantità in ingresso nel motore in base alla pressione esercitata sull'acceleratore. Si può considerare che i test eseguiti grazie all'HIL chiudano il gap tra i test MIL, nei quali solo i modelli vengono validati, e il sistema vero e proprio.

L'HIL si andrà quindi a sostituire al motore dell'automobile, offrendo una simulazione che comprenda HW e SW e che interagisca attraverso segnali Input/Output (I/O) come se il motore fosse presente. Nella figura 1.6 è mostrato il ciclo chiuso all'interno del quale si trovano l'HIL e il DUT, dove entrambi genereranno ed elaboreranno vari tipi di segnali, analogici, digitali o comunicazioni su bus seriali. Dato che gli aggiornamenti possono essere fatti via SW sarà quindi possibile incorporare rapidamente modifiche sia alla ECU che alla simulazione, ampliando lo spazio dei possibili test eseguibili, raggiungendo il grado di copertura necessario rimuovendo il rischio di danneggiare sistemi fisici complessi e costosi.

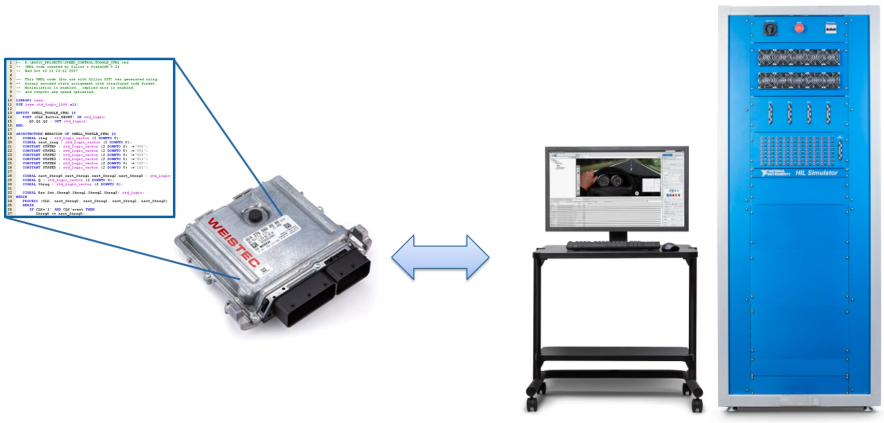


Figura 1.5: Motore sostituito dall'HIL [NIf]

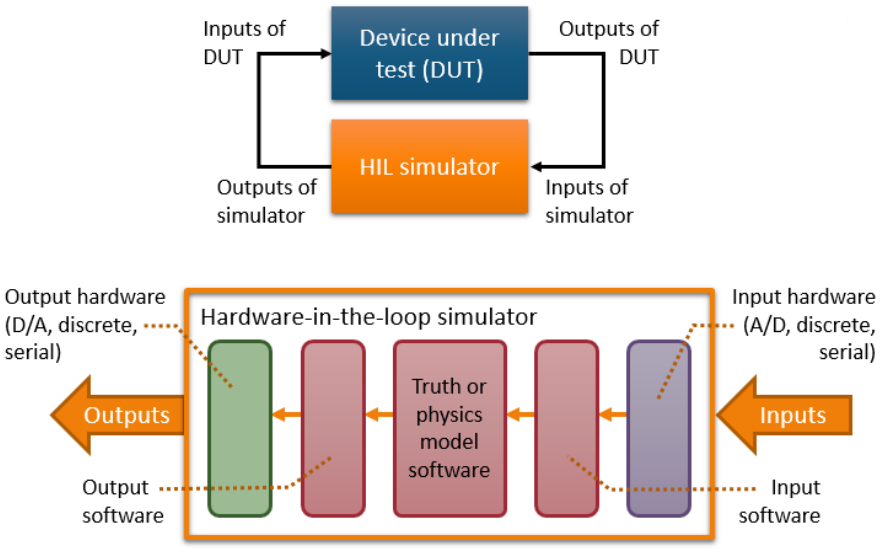


Figura 1.6: Hardware all'interno del loop [Vis20]

Per ottenere dei risultati che abbiano un valore importante da questi test la qualità del SW che gestisce la simulazione è di fondamentale importanza. Il SW deve essere affiancato da un HW che non solo tenga conto delle specifiche di sistema come il tipo di connettori utilizzati, ma che permetta anche la generazione di fallimenti e tutti gli scenari tipici del mondo reale.

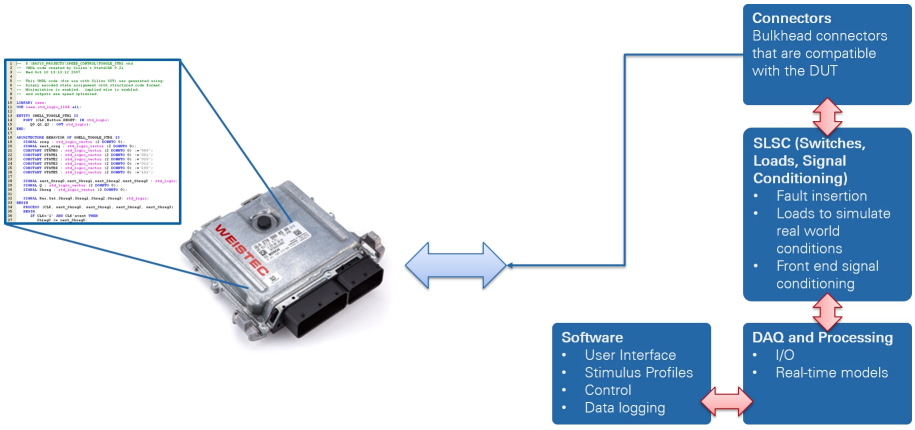


Figura 1.7: Requisiti per la simulazione [NIf]

Storicamente i *banchi di prova* richiedevano un intervento manuale, lasciando così spazio all'errore umano. Tutti i diversi tipi di fallimento, come ad esempio la disconnessione di un cavo o la perdita di alimentazione, necessitavano l'intervento di un operatore solitamente attraverso degli interruttori. La continua necessità di un tecnico presente al test per verificarne il corretto svolgimento e il tempo necessario per validare il corretto comportamento del banco in condizioni nominali rendevano molto inefficienti questo tipo di test.

Nella figura 1.8 è riportato uno schema generale che illustra il setup di un banco di prova HIL per applicazioni nel settore automobilistico. Le componenti mostrate sono:

- **Hardware-In-the-Loop**: un'architettura HW capace di emulare un sistema reale e di comunicare con componenti fisiche attraverso moduli di I/O.
- **Plant model**: solitamente un set di modelli diversi, progettati per simulare diverse parti di un sistema e caricate direttamente sull'HIL
- **PC**: ha il compito di interagire con l'HIL con l'obiettivo di progettare l'ambiente di test e successivamente permettere all'utente di visualizzare un'interfaccia grafica con la quale operare nei vari casi di utilizzo
- **Electronic Control Unit**: solitamente corrisponde con il *dispositivo sotto test* DUT
- **Automotive components**: componenti fisiche tipiche del sistema, come ad esempio attuatori, pulsantiere o schermi

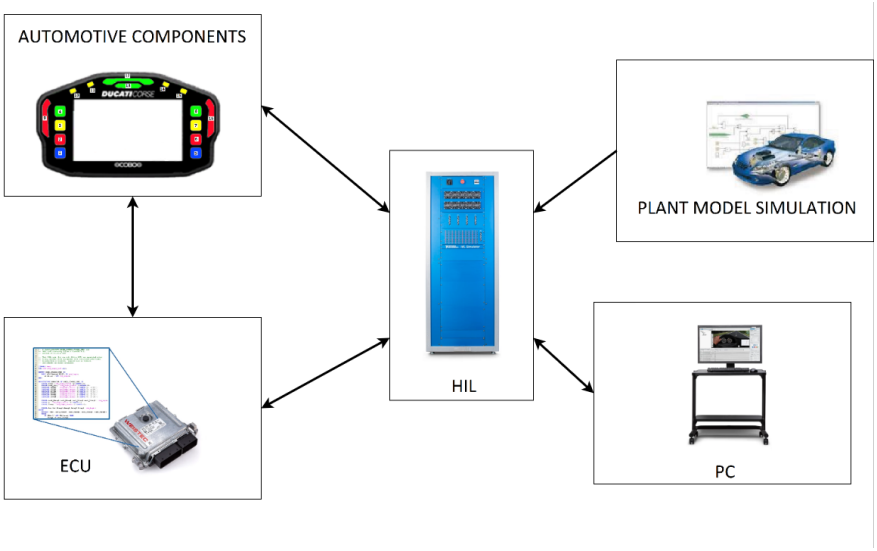


Figura 1.8: Composizione del sistema HIL

Software and racing have one thing in common: the moment you think you've got it perfect, something breaks.

Anonymous developer

Capitolo 2

Background

In questo capitolo verrà descritto l'ambiente all'interno del quale si è sviluppato questo progetto di tesi, analizzando lo stato attuale dei processi e i software utilizzati.

Figura 2.1: Esempio di file di test

L'approccio utilizzato per la generazione dei test ha come punto di partenza un file Excel contenente le specifiche descritte dallo stratega del quale è possibile vedere un estratto nella figura 2.1. Nella parte alta sono riportate informazioni utilizzate per tenere traccia delle versioni di HW e SW utilizzate al momento della stesura del file, mentre nella parte bassa vengono scritti i test veri e propri. Nella prima colonna viene riportato un numero identificativo della *run*, ovvero della singola esecuzione della simulazione, che in genere corrisponde all'uscita dalla *pit lane*, alcuni giri di pista completi e infine l'ultimo giro di rientro ai box, ma che può anche essere ampliata alla lunghezza di gara in caso fosse necessario per un determinato test. Nella parte centrale della tabella sono riportate altre informazioni relative alle impostazioni iniziali da utilizzare per le prove, mentre nell'ultima colonna a destra sono definite le effettive specifiche di test.

2.1 Hardware

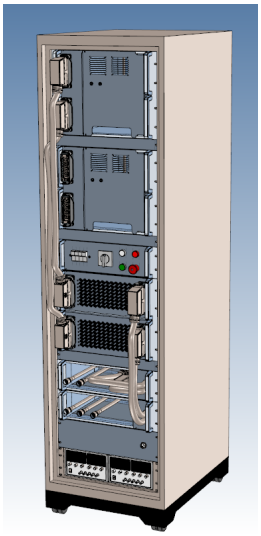


Figura 2.2: Struttura HIL Ducati

Durante lo svolgimento di questo progetto l'ente ospitante ha reso possibile acquisire importante esperienza mettendo a disposizione componenti HW e SW altamente specializzati sviluppati da altre aziende leader nel settore, come ad esempio l'HIL sviluppato da Alma Automotive. Quest'ultima, nata nel 2002 come spin-off dell'Università di Bologna, rappresenta l'unione sinergica fra le conoscenze acquisite nelle attività di ricerca accademica e gli anni di esperienza nello sviluppo di soluzioni sul campo. L'azienda si è poi evoluta per offrire prodotti *chiavi in mano* e servizi di consulenza tecnica supportati da soluzioni HW e SW personalizzate. [Autb]

La struttura definitiva, visibile nella figura 2.2, assemblata utilizzando principalmente componenti prodotti da National Instruments (NI), è composta da cinque sezioni:

- **Pannello di alimentazione**: un pannello utilizzato per accendere e spegnere il sistema, provvisto di interruttori di sicurezza per un'arresto d'emergenza
- **Scompartimenti ECU/DUT**: due scompartimenti predisposti per accogliere i DUT, in particolare due ECU per le quali sono già stati predisposti tutti i necessari connettori
- **Pannello delle Breakout Box (BoB)**: integrato all'interno di uno scompartimento contenente l'elettronica responsabile dell'acquisizione e generazione dei segnali, i quali si possono attivare e disattivare utilizzando dei pin posti fronte al pannello BoB. In breve questo pannello rappresenta il cuore delle connessioni elettriche dell'HIL, permettendo agli utenti di leggere i valori dei segnali e gestirne eventuali disconnessioni
- **Scompartimenti di carico**: contengono tutti i sensori e attuatori fisici che non vengono simulati dal sistema ma che sono necessari per il corretto funzionamento della ECU. Sono presenti due scompartimenti di questo tipo, carat-

terizzati solo dall'elettronica che l'utente sceglie di posizionarci all'interno, in maniera da poter parallelamente avere due DUT

- **Alimentatori**: due alimentatori programmabili che si occupano di fornire corrente elettrica a tutto il sistema

La presenza di componenti fisiche all'interno dell'HIL che nasce con l'obiettivo di effettuare test simulando il più possibile la realtà risulta utile agli sviluppatori innanzitutto per poter testare il componente stesso e il suo algoritmo di controllo prima di essere assemblato, oltre che poter generare dei fallimenti del sistema anche dal punto di vista fisico.

2.2 Software National Instruments

Come accennato in precedenza, l'HIL sviluppato per Ducati si basa su componenti National Instruments[NIc] e per questo motivo anche l'insieme degli strumenti utilizzati per configurarlo e utilizzarlo sono prodotti dalla stessa azienda, leader nella produzione appunto di strumenti HW e SW per la misura e l'automazione industriale.

2.2.1 VeriStand

Il framework utilizzato è quindi quello di VeriStand, uno strumento sviluppato appositamente da NI per creare ed eseguire test, simulare un intero sistema e interagire con il DUT. Grazie inoltre alla capacità di importare modelli creati con altri tool per simulare il pilota, la moto e la pista, è stato possibile sviluppare tutti i modelli utilizzando MatLab.

Attraverso le interfacce fornite da VeriStand sarà inoltre possibile interagire in tempo reale con i segnali di I/O del modello, potendo così simulare tutte le tipologie di situazioni come la perdita della comunicazione via CAN, o del BMS, ovvero il *Battery Management System*, oltre a qualsiasi possibile fallimento.

L'interfaccia grafica sviluppata per interagire con il modello della MotoE, riportata in figura 2.3, si suddivide in varie sezioni a seconda delle funzioni:

- **TestBench Control** (grigio): controllo del banco di prova, gestisce l'avvio e lo spegnimento della ECU e monitora lo stato del veicolo sulla base della sua macchina a stati interna
- **Driver Inputs** (arancione): input del pilota, simula la pressione del pilota sulla pulsantiera posta sul manubrio, rappresentata graficamente. Durante le simulazioni potrebbe essere necessario premerli per inserire il *pit limiter*, il *launch control* o cambiare le mappe del motore selezionate
- **Battery Model** (verde): modello della batteria, contiene le informazione necessarie per monitorare lo stato del pacco batteria della moto, come lo stato di carica, indicato con SoC, lo stato dei contatti, il voltaggio e la temperatura

- **BMS Model** (blu): modello del Battery Management System (BMS), permette di interagire con il sistema di controllo della batteria, generando dei fallimenti nello stesso o simulando semplicemente la fase di ricarica della batteria
- **Torque Manager** (marrone): gestore della coppia, imposta i valori di inizializzazione di coppia e frizione
- **Lap Set Up** (rosa): impostazione del giro, permette di interagire con il modello che rappresenta la vera simulazione dei giri in pista, riportandola al punto di partenza, avviandola e inizializzando la comunicazione via CAN
- **Inverter Model** (viola): modello dell'inverter, permette di controllare la velocità della moto e del motore, oltre alla sua corrente e alla macchina a stati dell'inverter stesso
- **DCDC Model** (giallo): modello del DCDC, fornisce le informazioni relative al convertitore e all'alimentazione dal lato del basso voltaggio, permettendo di generare fallimenti che possono causare problemi direttamente alla ECU

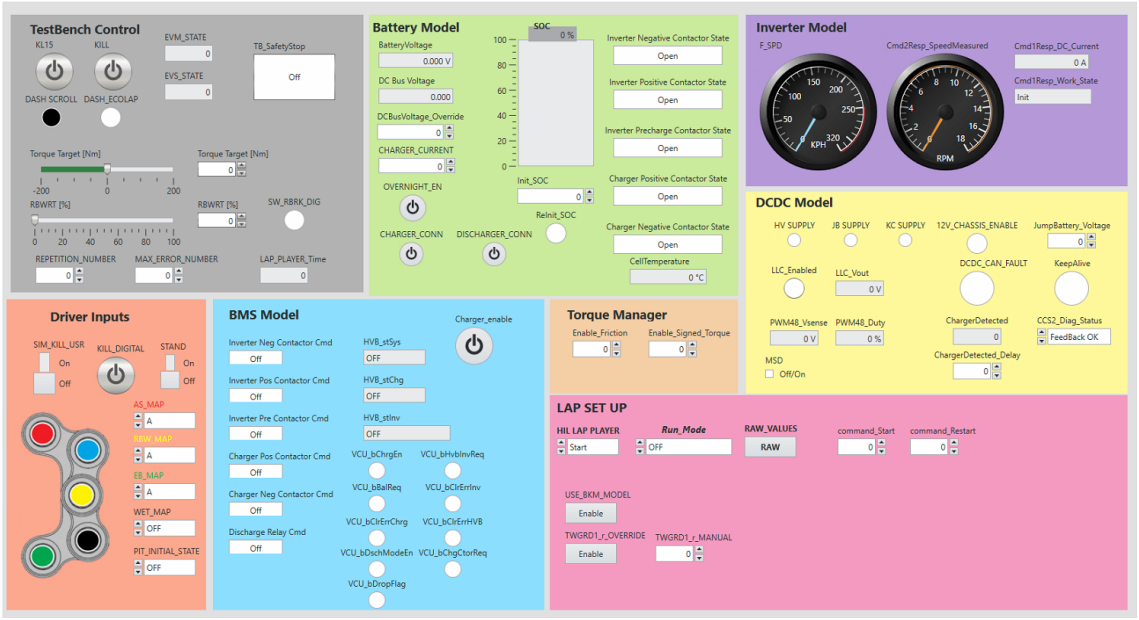


Figura 2.3: Interfaccia VeriStand per la MotoE

2.2.2 Stimulus profile editor

Questo tool appartenente alla stessa *suite* di VeriStand permette la generazione degli *stimulus profile* e delle *real-time sequences*. La separazione di queste due componenti consente una grande riusabilità mantenendo però una chiara correlazione con i file di definizione di sistema di VeriStand per i singoli test. Oltre a stimulus profiles lineari e consequenziali è possibile crearne anche di complessi includendo processi decisionali, sotto-sequenze e una varietà di costrutti di programmazione. Questi elementi si combinano per offrire un ambiente flessibile in grado di eseguire test in

tempo reale. Qui di seguito verranno approfonditi questi concetti e alcune delle funzionalità generalmente usate per creare gli stimulus profile. [NIa]

Real-time sequences

Le sequenze real-time sono i costrutto fondante degli stimulus profile e vengono eseguite dal *NI VeriStand Real-Time Engine*. Queste sequenze possono includere cicli `While` e `For`, variabili e dichiarazioni condizionali, oltre a concetti più complessi come l'introduzione come già accennato di sotto-sequenze e di multitasking all'interno della singola sequenza. Un estratto delle primitive disponibili può essere osservato nella figura 2.4

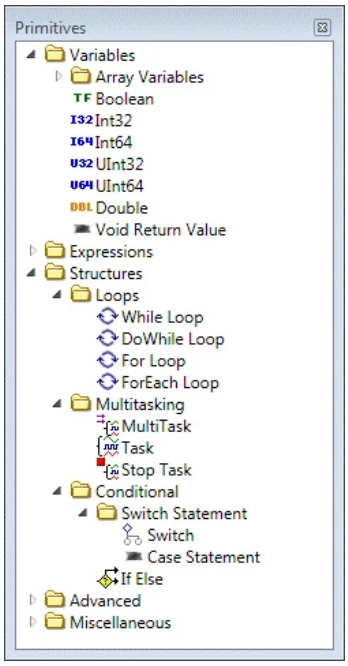


Figura 2.4: Schema dei costrutti di programmazione disponibili [NIa]

Oltre a questi costrutti, una sequenza può anche essere utilizzata per generare alcune tra le forme d'onda comunemente utilizzate, come le onde sinusoidali o quadre, come visibile nella figura 2.5.

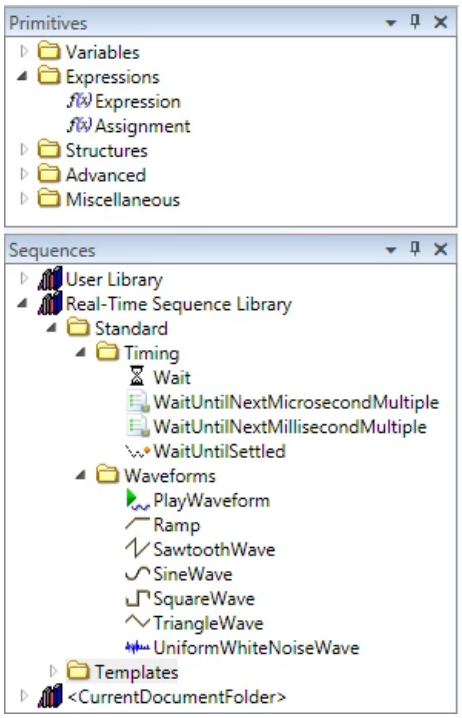


Figura 2.5: Sequenze ed espressioni matematiche disponibili [NIa]

Una sequenza di test si compone di tre sezioni che possono essere ritrovate anche nella figura 2.6:

- Setup
- Main
- Cleanup

La prima viene utilizzata per stabilire le condizioni iniziali di una sequenza, come ad esempio l'assegnamento di valori alle variabili o la misurazione preliminare di alcune grandezze di sistema. La sezione centrale è quella nella quale la maggior parte delle informazioni verrà processata e dove saranno condotti i test effettivi. Al termine di questa sezione l'ultima si occuperà di ripristinare uno stato conosciuto del sistema preparandolo per futuri test.

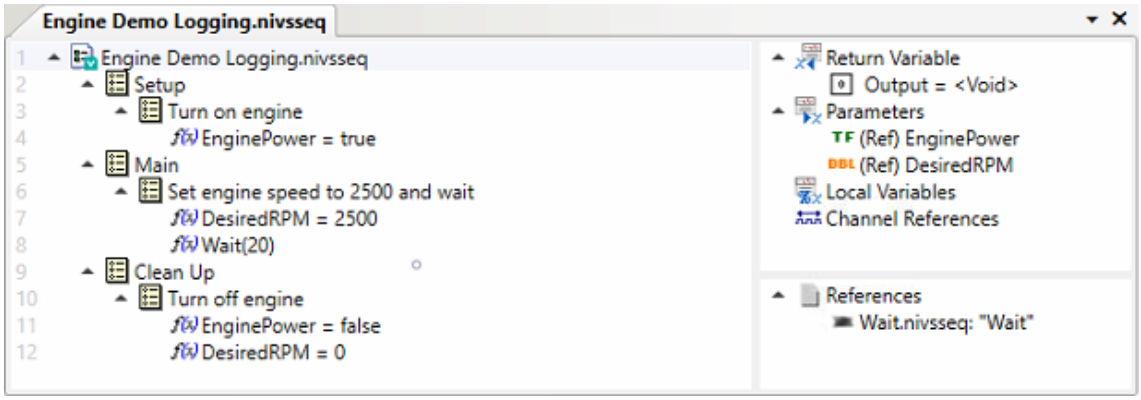


Figura 2.6: Esempio di sequenza real-time

Stimulus Profile

Lo stimulus profile agisce da esecutore per le sequenze appena descritte, ma può anche agire direttamente sui progetti VeriStand aprendoli e chiudendoli, effettuare logging dei dati e un'analisi *pass/fail* sul risultato delle sequenze. Analogamente alle sequenze anch'esso è composto da *Setup*, *Main* e *Cleanup*, tuttavia le fasi iniziali e finali vengono solitamente utilizzate per la gestione dei log e dei progetti VeriStand, permettendo al'utente di passare da un tipo di test all'altro senza bisogno di intervenire manualmente sul sistema.

La sezione principale è invece utilizzata per richiamare le sequenze, offrendo la possibilità di richiamarne un insieme in maniera consequenziale se bisogno di effettuare i passaggi manualmente. Questa divisione tra stimulus profile e sequenze permette di creare librerie di sequenze di test che possono essere richiamate in diverse occasioni in base alle necessità. Nella figura 2.8 si può osservare l'interfaccia grafica dell'editor all'interno della quale è presente uno stimulus profile, riportato ingrandito anche nella figura 2.7, che include l'avvio del logging nella fase di Setup, le chiamate a una serie di sequenze e poi l'arresto del log nella fase di Cleanup.

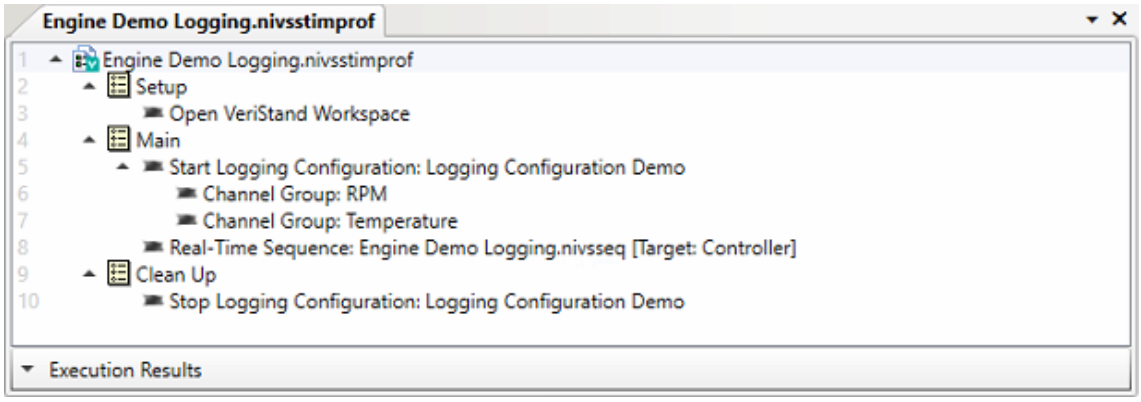


Figura 2.7: Demo di stimulus profile

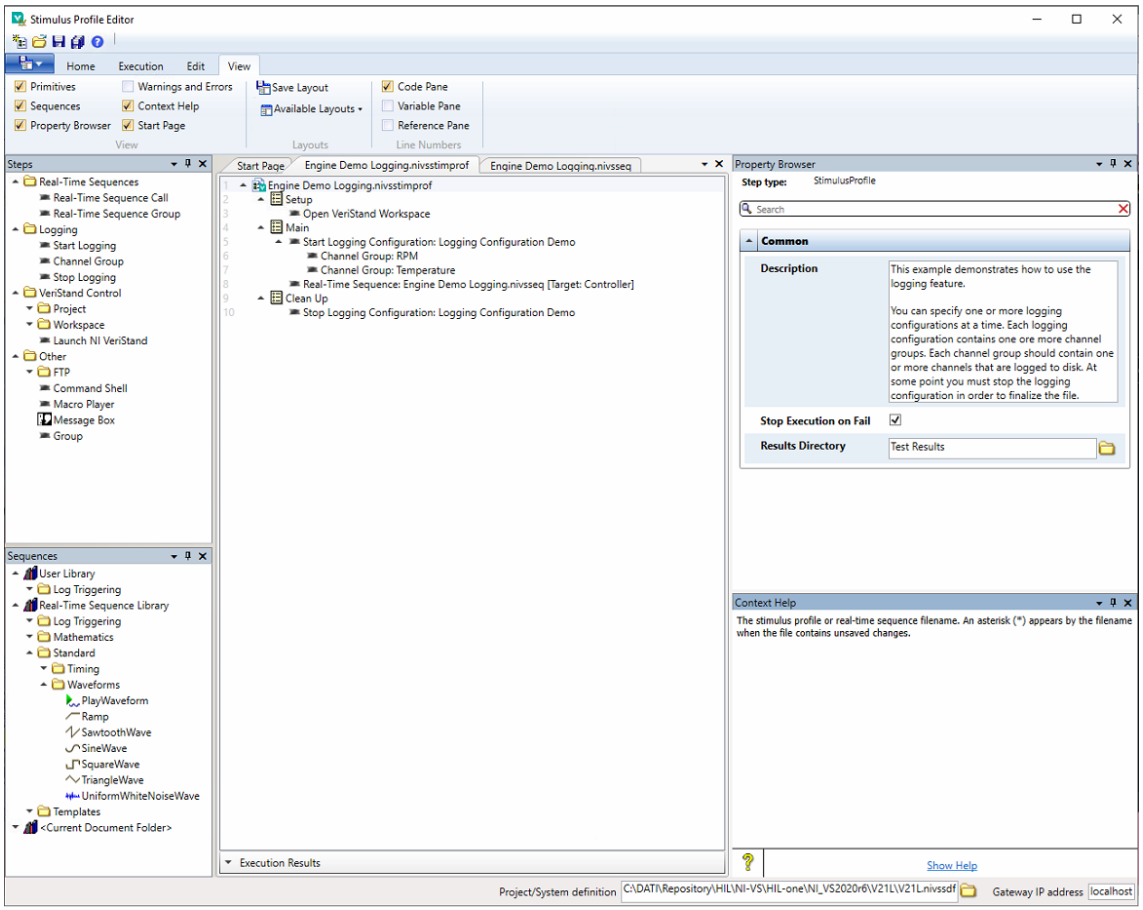


Figura 2.8: Esempio di interfaccia di stimulus profile [NIa]

2.3 Strumenti di test

Esistono diverse metodologie per interagire con la simulazione e, come descritto in precedenza, la modalità di utilizzo attuale prevede lo sviluppo dei test attraverso lo Stimulus Profile Editor. Questo tool produce due tipi di file diversi per lo stimulus profile e per le sequenze, rispettivamente `.nivsstimprof` e `.nivsseq`, i quali se analizzati risultano essere scritti in eXtensible Markup Language (XML), offrendo così la possibilità di modificare manualmente i file per variare i test da svolgere. Questo approccio può risultare però complesso in quanto non si avrebbe controllo sulla correttezza dei file prodotti. Ponendo l'obiettivo di comprendere i processi utilizzati da VeriStand sono state svolte ricerche sugli strumenti utilizzabili in alternativa a Stimulus Profile Editor e sulle Application Programming Interface (API) messe a disposizione da NI. Prima di esaminare questi strumenti alternativi è necessario analizzare il funzionamento di VeriStand.

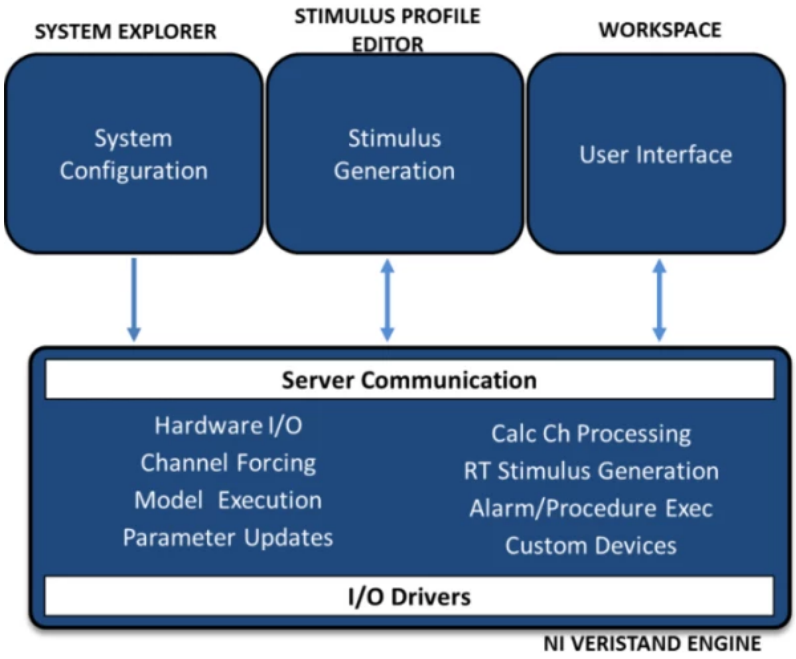


Figura 2.9: Funzionamento di VeriStand [Nid]

Nella figura 2.9 è riportato il modo in cui vengono create le applicazioni real-time l'utilizzo di varie sorgenti. La finestra del *system explorer* viene utilizzata per configurare l'*engine* in esecuzione su di un target come ad esempio un sistema real-time PXI, mentre la finestra *workspace* permette successivamente di interagire con l'applicazione in tempo reale attraverso un'interfaccia utente che include una serie di strumenti anche per monitorarne lo stato.

Nella figura 2.10 si può osservare infine uno schema che riporta in blu le componenti configurate utilizzando l'ambiente di VeriStand, mentre in bianco sono riportati gli strumenti che possono essere utilizzati per creare componenti aggiuntivi i quali a loro volta potranno essere aggiunti come componenti nativi funzionando perfettamente con l'applicazione. Si andranno ora ad analizzare più nel dettaglio questi strumenti.

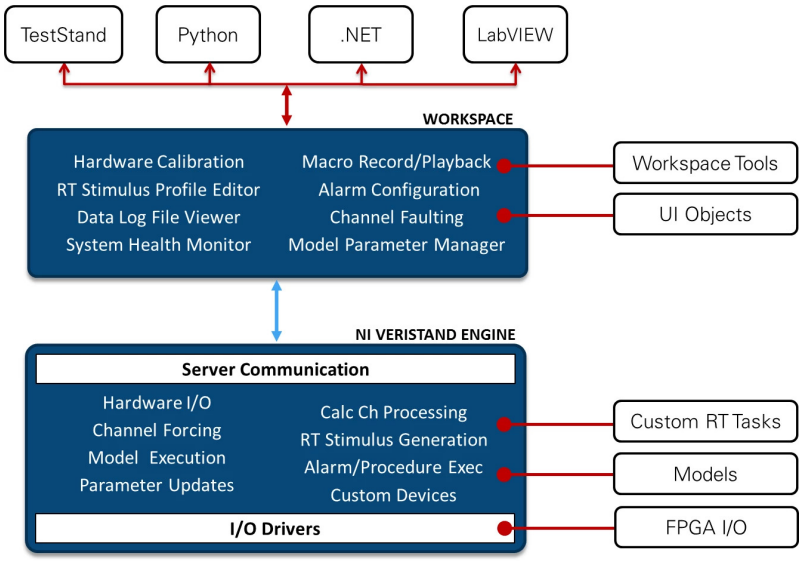


Figura 2.10: Interoperabilità di VeriStand [Nid]

2.3.1 TestStand

Il tool TestStand è stato sviluppato da NI per ampliare le funzionalità relative alla generazione di sequenze e stimulus profile. Nasce come strumento per gestire i sistemi di test, facilitandone lo sviluppo, l'esecuzione e il debug, fornendo all'utente la completa visuale sui processi e i risultati. Anche in questo caso analogamente allo Stimulus Profile Editor si potranno creare delle sequenze utilizzando un linguaggio di alto livello utilizzando delle funzioni apposite, inserite all'interno di una struttura suddivisa in tre parti come visto in precedenza. Scegliere di utilizzare TestStand offrirebbe la possibilità di:

- personalizzare i test in base alle esigenze,
- automatizzare l'estrazione e il salvataggio dei risultati,
- migliorare l'efficienza parallelizzando i test,
- replicare efficientemente i test,
- trovare errori nei test attraverso gli strumenti di debug integrati,
- personalizzare l'interfaccia in base alle necessità.

Nonostante tutti questi vantaggi la soluzione non è stata adottata per questo progetto poiché, come già definito in attività precedenti, oltre a essere una soluzione particolarmente costosa non apporterebbe nessun miglioramento al processo di generazione delle sequenze.

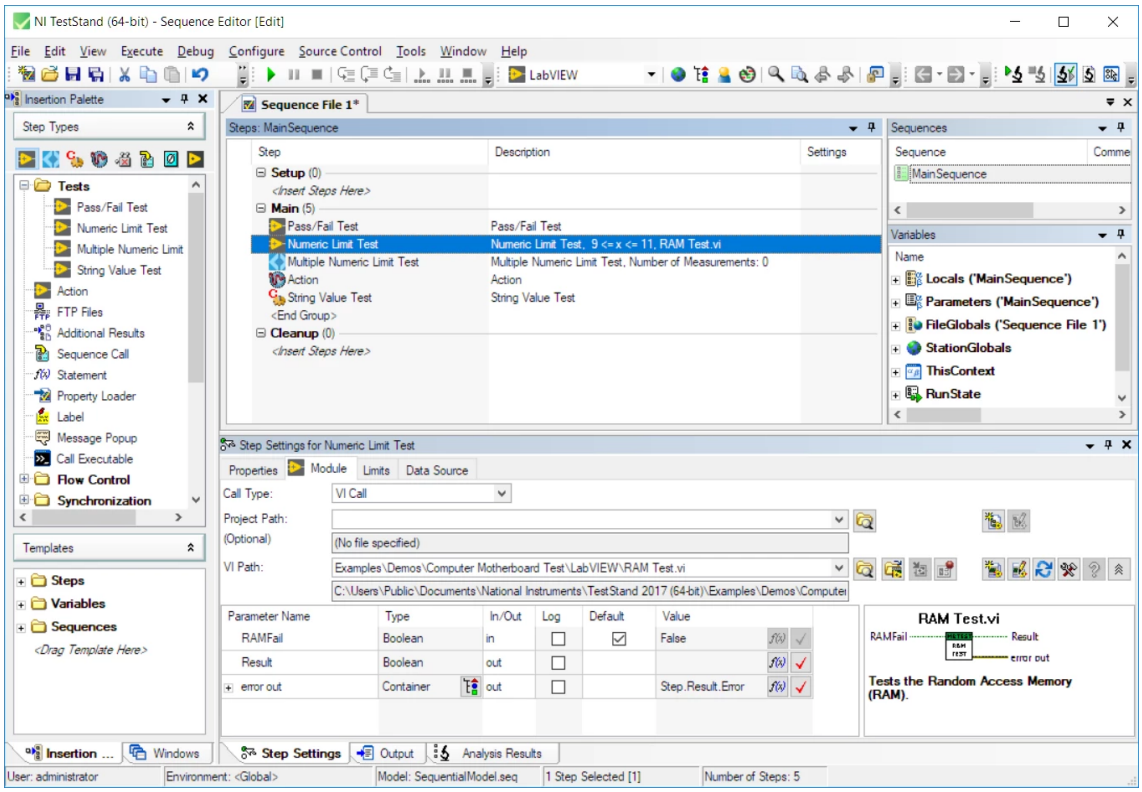


Figura 2.11: Interfaccia di TestStand [NIb]

2.3.2 LabVIEW

LabVIEW, abbreviazione di *Laboratory Virtual Instrumentation Engineering Workbench* è l'ambiente di sviluppo integrato per il linguaggio di programmazione visuale di NI. Tale linguaggio grafico, chiamato *Linguaggio G*, permette di utilizzare icone e linee al posto di codice testuale per interagire con il sistema. Al posto delle classiche istruzioni programmando utilizzando LabVIEW si utilizzerà uno paradigma legato al flusso dei dati nel quale le informazioni fluiscono attraverso i nodi del diagramma rappresentato. Questi diagrammi determineranno l'ordine di esecuzione delle funzioni che andranno a interagire con i Virtual Instruments (VIs), rappresentazioni che imitano gli strumenti fisici.

Anche questa alternativo offre svariati vantaggi, tra i quali:

- riduzione del tempo necessario per impostare il sistema, avendo accesso ai driver di migliaia di dispositivi, programmi d'esempio e documentazione per connettere virtualmente ogni strumento,
- possibilità di riutilizzare librerie provenienti da altri linguaggi come ad esempio C, C++, .NET, Python e MATLAB,
- semplicità nello sviluppo di interfacce utente professionali per la visualizzazione dei risultati,
- accesso a una ampia gamma di corsi on-line e non, per migliorare le proprie conoscenze tecniche del tool per ogni livello di capacità.

Anche in questo caso il tool non è stato portato avanti come proposta dato che la sua complessità potrebbe risultare non intuitiva per l'utente, andando così a complicare il processo che si vuole andare a snellire, oltre a essere anche più costoso di quello descritto in precedenza.

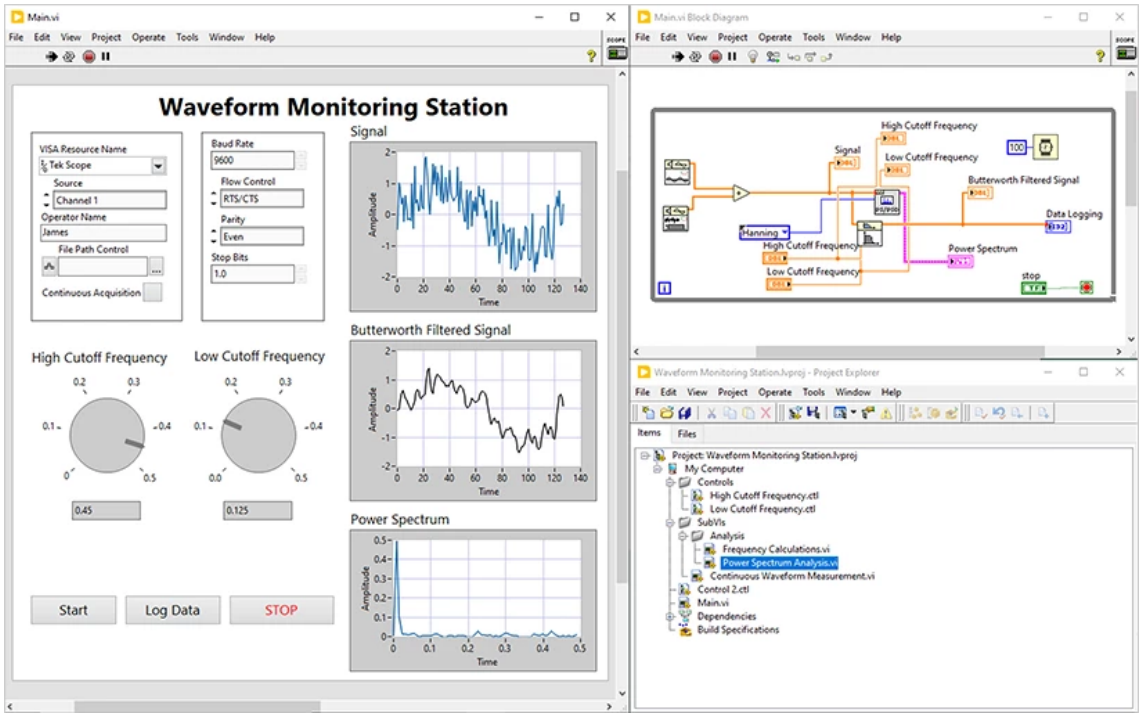


Figura 2.12: Interfaccia di LabVIEW [NIg]

2.3.3 .NET

.NET è un framework multi piattaforma open source gratuito per la creazione di app moderne e servizi cloud avanzati. [Mic] Offre la possibilità di utilizzare diversi linguaggi, editor e librerie nati per lo sviluppo in vari campi, come quello dell'IoT, del Web e delle applicazioni mobile desktop. I tre linguaggi supportati per lo sviluppo in .NET sono:

- **C#**: linguaggio semplice e moderno, orientato agli oggetti e con una sintassi molto simile al C,
- **F#**: linguaggio open source con una sintassi molto semplice e che richiede poco codice per creare un software, molto efficace per applicazioni di tipo mission-critical,
- **Visual Basic**: altro linguaggio orientato agli oggetti, sviluppato da Microsoft stessa, permette di velocizzare l'implementazione di applicazioni *type-safe* in .NET.

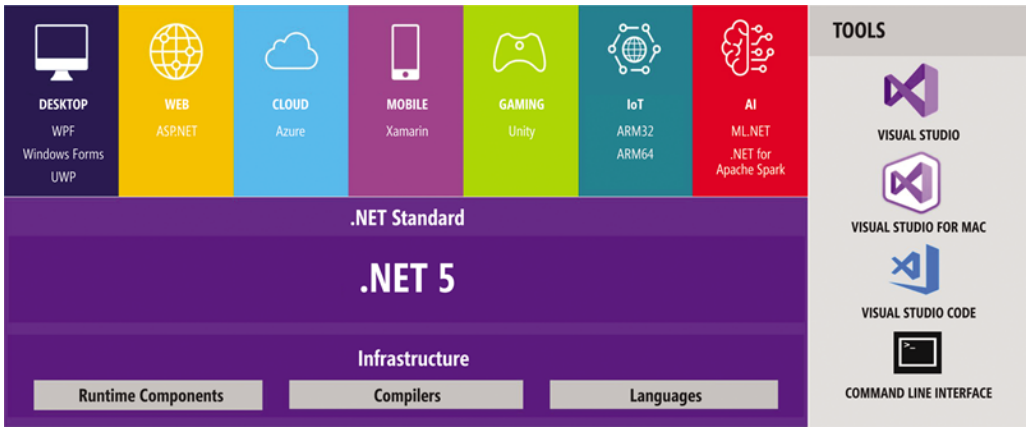


Figura 2.13: Caratteristiche di .NET

Grazie al fatto di essere multi piattaforma, o più comunemente *cross-platform*, questo framework permette di sviluppare facilmente applicazioni indipendentemente dal sistema operativo sul quale verranno eseguite. Una delle caratteristiche però più interessanti è l'esistenza della CLI, acronimo che indica la *Common Language Infrastructure*, una specifica immaginabile come una sorta di modello per la creazione di linguaggi compatibili con .NET che ha permesso la nascita di altri linguaggi come ClojureCLR, Eiffel, IronPython, PowerBuilder e molti altri. Nonostante in precedenza l'approccio al progetto attraverso questo framework fosse stato scartato la decisione è stata in parte rivista durante questo lavoro di tesi per raggiungere tutti i requisiti. Questo è stato possibile anche grazie alle API compatibili con .NET messe a disposizione da NI che permettono di accedere a svariate funzioni.

2.3.4 Python

Python è un linguaggio di programmazione di alto livello, adatto a vari utilizzi tra i quali si possono elencare le applicazioni distribuite, lo scripting e la computazione numerica. Questo linguaggio è caratterizzato da una tipizzazione dinamica e supporta più paradigmi di programmazione come quello procedurale, quello orientato agli oggetti e quello funzionale. Viene spesso definito un linguaggio *con le batterie incluse* grazie alla quantità e alla qualità delle librerie alle quali offre accesso.

In una fase di analisi svolta precedentemente questo linguaggio era stato scelto come miglior punto di partenza per questo progetto, basandosi principalmente sulla sua accessibilità a livello di costi, sulla grande disponibilità di tutorial e librerie disponibili on-line, e sulla presenza anche in questo caso di API messe a disposizione direttamente da NI per l'interoperabilità tra Python e VeriStand.

Verranno ora brevemente illustrati i punti principali sorti dall'analisi delle API [NIe], riportando innanzitutto come riferimento in 1 un esempio fornito direttamente da NI.

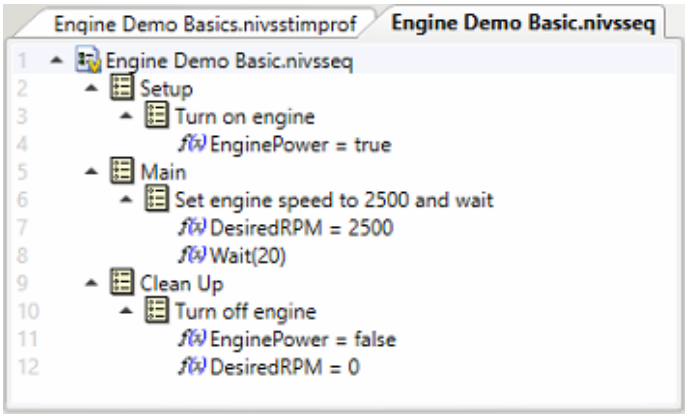


Figura 2.14: Esempio *Engine Demo Basic* in VeriStand

Nella figura 2.14 può essere osservata direttamente dall'interfaccia grafica di VeriStand la sequenza *Engine Demo Basic*, uno degli esempi forniti insieme al programma per illustrare le possibili applicazioni e funzionalità. Proprio partendo da questa dimostrazione è stato poi generato il seguente corrispettivo in Python.

Dopo gli `import` relativi alle API si può osservare la prima particolarità, ovvero il decorator `@nivs_rt_sequence`, che dovrà essere preposto a ogni sequenza real-time presente. Sarà inoltre necessario utilizzando il tag `@NivsParam` indicare per ogni parametro il tipo del dato, il su valore di default e se viene passato per valore o per riferimento.

La sequenza, di più facile interpretazione nella sua versione VeriStand, prevede che:

- si accenda il motore,
- si imposti il valore di RPM desiderate al valore preso in ingresso,
- si attendano 20 secondi,
- si spenga il motore,
- si ripristini l'originale valore di RPM.

La funzione `ChannelReference` può essere richiamata per accedere ai singoli canali per poi estrarne il valore attraverso la proprietà `.value`, necessaria anche per accedere ai valori dei parametri stessi. Si può notare come in questo caso la struttura *setup-main-cleanup* tipica della sequenza venga collassata all'interno della stessa funzione, mantenendone però la consequenzialità.

Anche la successiva funzione `run_engine_demo` viene eseguita come sequenza grazie all'apposito decorator. In questa funzione vengono impostati i parametri e viene richiamato l'effettivo test. La chiamata risulta analoga a quella che si farebbe per una normale funzione Python, con la differenza che se i parametri vengono passati *per riferimento* allora dovranno avere un tipo definito.

Le due funzioni successive hanno obiettivi analoghi ma comportamenti molto diversi. La prima, `run_non_deterministic`, esegue la sequenza in maniera non deterministica, eseguendola sull'host sfruttando le ClientAPI che vengono installate

insieme a VeriStand, comunicando con il gateway per leggere e scrivere i valori dei canali. Nel caso si utilizzi un IDE Python sarà anche possibile effettuare il debug della funzione come se fosse normale codice Python. La seconda funzione, `run_deterministic`, invece compila ed esegue la sequenza in maniera deterministica all'interno del motore di VeriStand, per cui non sarà possibile effettuare il debug della funzione. La funzione `run_py_as_rtseq` accetta come parametro il nome della funzione Python che si vuole eseguire come sequenza real-time. Nel caso alla funzione non venga preposto il decorator necessario verrà generato l'errore `:any:‘niveristand.errors.VeristandError‘`

```
1  from niveristand import nivs_rt_sequence, NivsParam,
   ↪  realtimesequencetools
2  from niveristand.clientapi import BooleanValue, ChannelReference,
   ↪  DoubleValue
3  from niveristand.library import wait
4
5  @nivs_rt_sequence
6  @NivsParam("engine_power", BooleanValue(0), NivsParam.BY_REF)
7  @NivsParam("desired_rpm", DoubleValue(0), NivsParam.BY_REF)
8  def engine_demo_basic(engine_power, desired_rpm):
9
10     engine_power_chan = ChannelReference("Aliases/EnginePower")
11     desired_rpm_chan = ChannelReference("Aliases/DesiredRPM")
12     engine_power_chan.value = engine_power.value
13     desired_rpm_chan.value = desired_rpm.value
14     wait(DoubleValue(20))
15     engine_power_chan.value = False
16     desired_rpm_chan.value = 0
17
18 @nivs_rt_sequence
19 def run_engine_demo():
20     engine_demo_basic(BooleanValue(True), DoubleValue(2500))
21
22 def run_non_deterministic():
23     run_engine_demo()
24
25 def run_deterministic():
26     realtimesequencetools.run_py_as_rtseq(run_engine_demo)
27
28 if __name__ == "__main__":
29     run_non_deterministic()
30     print("Finished non-deterministic")
31     run_deterministic()
32     print("Finished deterministic")
```

Listing 1: Esempio *Engine Demo Basic* in Python

In racing, there are no shortcuts. Only precision, speed, and relentless iteration.

Anonymous

Capitolo 3

HIL Test Automator

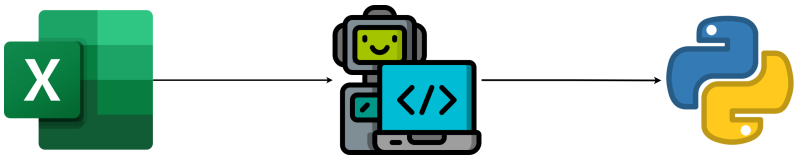
3.1 Analisi dei requisiti

In questa sezione verranno analizzati i requisiti fissati per questo progetto di tesi, partendo da un'introduzione all'attuale metodologia utilizzata per l'esecuzione dei test, e introducendo quelli che sono i vincoli e gli obiettivi.

Al momento la procedura di test prevede come nodo centrale che uno degli sviluppatori effettui manualmente i test richiesti dallo stratega attraverso un foglio Excel.

In situazioni con deadline molto ravvicinate e spesso soggette a variazioni causate da fattori esterni, come nel caso dello sviluppo di SW nei periodi precedenti alle gare o durante il fine settimana stesso della competizione, la quantità di tempo che un programmatore può dedicare ad attività critiche diventa determinante. Per questo motivo l'azienda ospitante ha espresso la volontà di implementare uno strumento automatizzato che permetta di minimizzare la necessità di interazione con lo sviluppatore permettendo di parallelizzare le attività di test ad altri progetti, mettendo in mano anche a personale meno tecnico un meccanismo semplificato per fornire dei risultati agli strateghi.

Oltre a ciò con l'avanzare del tempo e all'aumentare della complessità del sistema SW e HW è risultata chiara la necessità di poter cominciare a creare uno standard relativo alle attività di testing, che ne permetta un ripetibilità sistematica. Così facendo si potrebbe ad esempio impostare una sorta di libreria di test organizzati in batterie da poter essere eseguiti rapidamente e senza sforzo per effettuare i test di non regressione necessari per poter garantire la qualità di ogni release.



In precedenza questo progetto era già stato avviato, attraversando una prima fase di ricerca e selezione di alcune caratteristiche del sistema che si aspirava a creare. Dopo aver quindi analizzato le documentazioni raccolte e i materiali prodotti è stata effettuata una definizione più dettagliata dei requisiti necessari all'azienda.

Come già anticipato il concetto centrale del progetto risultava essere l'automazione del processo di test. Questo processo si articola su cinque punti:

- lo stratega compila un foglio Excel descrivendo alcuni test che ritiene necessari
- il tester riceve il foglio di test e imposta l'ambiente di lavoro attraverso VeriStand e lo Stimulus Profile Editor
- per ogni test, corrispondete a una *run*, attraverso l'editor viene creata un sequenza all'interno della quale vengono inseriti manualmente tutti i canali e i parametri necessari
- il tester avvia attraverso l'editor ogni sequenza e ne verifica l'andamento attraverso l'interfaccia di VeriStand. Nel caso in cui il test per essere considerato superato con successo debba generare un allarme che viene mostrato al pilota attraverso lo schermo posizionato sul cruscotto, anche chiamato GRD, il tester dovrà rimanere presente a verificare visivamente la corretta esecuzione
- al termine di tutte run il tester dovrà raccogliere i risultati ottenuti e comunicarli nuovamente allo stratega

I primi due punti analizzati come chiavi del progetto sono stati la traduzione in sequenze e la verifica visiva del sistema. Per quanto riguarda la traduzione, in base all'analisi precedente i punti di forza di automazione incentrata su Python risultavano evidenti, per cui il primo requisito definito è stato quello di sviluppare un tool che permettesse di generare automaticamente del codice eseguibile come sequenze real-time partendo dalle specifiche di test scritte in linguaggio non tecnico. Per quanto riguarda la verifica visiva è stato invece richiesta una fase di analisi e ricerca mirata a trovare la soluzione che più si adattasse alle necessità, con l'obiettivo di creare un sistema che prendesse in input il segnale di una webcam orientata verso un GRD collegato direttamente all'HIL e che fosse capace di interpretare i segnali trasmessi. Il cruscotto realizzato dal reparto corse è caratterizzato da uno schermo centrale attorniato da una serie di led, sul quale vengono mostrate le informazioni principali necessarie al pilota, come le mappe, la marcia, i giri del motore e gli allarmi, questi ultimi definiti dalla stringa mostrata e da colore e frequenza con la quale vengono illuminati i led a contorno dello schermo. Per questo il secondo requisito è stato quello di sviluppare un tool che fosse in grado di analizzare questo tipo di informazioni in input e interpretare informazioni e allarmi, tenendone traccia.

A contorno di questi requisiti fondamentali sono stati poi collegati i punti mancanti al processo, introducendo la richiesta di dare al sistema anche la capacità di eseguire in autonomia batterie di test senza necessariamente dover intervenire tra uno e l'altro, e di definire una reportistica che concludesse il ciclo dell'attività di test.

Il progetto

Nelle sezioni successive di questo capitolo verrà descritta l'effettiva implementazione dell'elaborato in oggetto a questa tesi, trattando oltre alla versione finale dello stesso anche le strade alternative che non sono state percorse e le motivazioni dietro a queste scelte.

Il sistema finale si articola su quattro componenti,

- **validator**
- **generator**
- **executor**
- **watcher**

ognuno dei quali svolge un compito fondamentale nell'automazione del processo. Per questo progetto infatti il punto focale sul quale ci si è principalmente concentrati sulla creazione di un sistema che seguisse l'intero percorso dei test, dalla loro generazione fino alla verifica dei risultati.

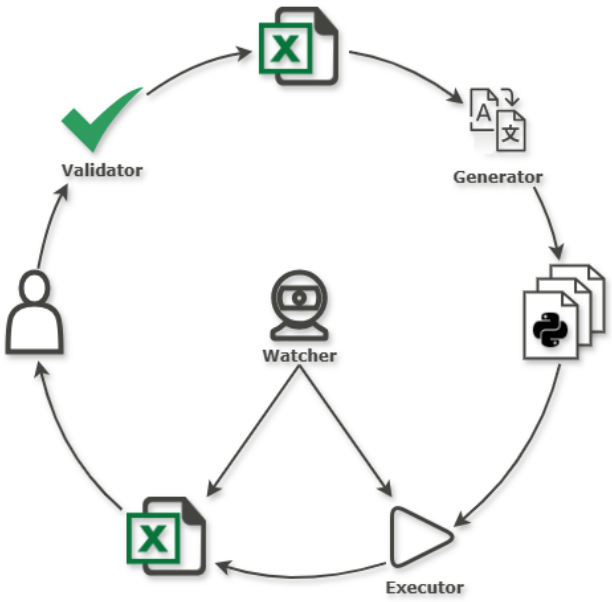


Figura 3.1: Struttura HIL Automator

Il processo sviluppato e visibile nella figura 3.1 prevede come punto di partenza l'utilizzo da parte dell'utente del primo tool per creare il foglio Excel che conterrà le specifiche per i test da svolgere. Questo file verrà poi aperto e interpretato dal secondo tool che si occuperà di scomporlo e tradurlo in codice eseguibile, creando una sequenza Python per ogni riga di test presente nel file. Questi file appena creati, tutti o solo un sottoinsieme, potranno essere selezionati attraverso il terzo tool ed eseguiti direttamente sull'HIL, generando dei file `.csv` di report. Il quarto e ultimo tool si occuperà invece di osservare il GRD inviando al tool incaricato di eseguire i test ogni qualvolta venga rilevata la presenza di un allarme sul cruscotto, generando infine anch'esso un report contenente tutte le informazioni raccolte. Questi tool verranno ora analizzati nel dettaglio dei loro funzionamenti.

3.2 Test validator

Il primo punto tratta la generazione delle specifiche dei test.

Il sistema attualmente in uso prevede la definizione delle specifiche di test attraverso un foglio Excel che aderisca a uno specifico template. I test in questo foglio vengono quindi descritti dallo stratega utilizzando il linguaggio naturale, senza uno standard ben definito. Una proposta precedentemente formulata prevedeva di standardizzare una delle colonne del foglio di test imponendo una struttura che si avvicinasse il più possibile a quella utilizzata nelle chiamate di funzione nel codice, con lo scopo di facilitare poi lo step di generazione che verrà descritto in seguito.

Run	Prove da svolgere	Note
1	Fail(IMD_Fail, 80, 20, 1)	Prove FAULT: simulare FAULT IMD
2	Fail(ActuatedTorque, 80, 20, 10)	Prove FAULT: simulare errore di coerenza di coppia
3	Fail(TWGRD1_r, 80, 5, 6000); Fail(TWGRD1_r, 95, 376, 6000); Fail(TWGRD2_r, 95, 376, 6000);	Prove FAULT: verifica funzionamento FAIL TWGRD

Tabella 3.1: Precedente proposta formato test

In seguito a un confronto con l'utente finale di questo strumento, sono stati aggiunti alcuni vincoli allo sviluppo. I punti risultanti da questa analisi sono i seguenti, agevolare la migrazione dal sistema attuale a quello in fase di sviluppo ed evitare che l'utente si trovi in difficoltà, minimizzare l'*overhead* dato dall'introduzione di un nuovo strumento.

Nella tabella 3.1 è riportato un esempio della proposta precedentemente descritta, che però non rispetta i vincoli appena visti. In particolare il problema che si evidenzia è che non si può dare per assodato che tutti gli strateghi utilizzatori di questo SW siano pratici di informatica e di conseguenza il passaggio repentino dal linguaggio naturale a uno di programmazione potrebbe rendere difficile la transizione. Questo approccio faceva inoltre affidamento sulla correttezza della definizione delle specifiche, senza offrire allo stratega la possibilità di validare il foglio appena prodotto.

Per risolvere entrambi i problemi è stata presa in considerazione l'idea di sviluppare un Domain-Specific Language (DSL) per la definizione dei test.

Domain-Specific Language

Con DSL si indica un linguaggio di programmazione specifico per un determinato dominio di applicazione, contrariamente a un *General-Purpose Language* che invece permette di spaziare tra più domini. Un DSL può essere di varia natura e comprende

quindi sia linguaggi visuali che testuali. La linea che separa i DSL e linguaggi di scripting non è ben definita, ma in generale si può vedere come i primi siano privi di funzioni di basso livello per l'accesso al *filesystem*, capacità di comunicazione tra processi, e altre caratteristiche tipiche dei linguaggi come quelli di scripting. Molti DSL non vengono compilati in codice eseguibile o *bytecode*, ma in diversi formati a seconda della necessità.

Lo sviluppo quindi di un DSL permetterebbe all'utente di verificare autonomamente la correttezza del test appena definito, nonché di utilizzare un linguaggio più prossimo a quello utilizzato fin'ora. L'obiettivo sarà quindi quello di definire i test non come mostrato in precedenza, ma come riportato nella tabella 3.2

Run	Prove da svolgere
1	Fail sul canale IMD_Fail a partire da 20 s per 1 ms con valore 80
2	Fail sul canale ActuatedTorque a partire da 20 s per 10 ms con valore 80
3	Fail sul canale TWGRD1_r a partire da 5 s per 6000 ms con valore 80; Fail sul canale TWGRD1_r a partire da 376 s per 6000 ms con valore 95; Fail sul canale TWGRD2_r a partire da 376 s per 6000 ms con valore 95;

Tabella 3.2: Proposta formato test con DSL

In questo modo si riuscirebbe a evitare che gli strateghi stravolgano il loro modo di descrivere i test, ma si renderebbe necessario sviluppare un'applicazione con interfaccia grafica propria che permetta di effettuare controlli su eventuali errori nella scrittura. Tenendo fede al vincolo di non apportare modifiche sostanziali al processo attualmente in uso sono state ricercate strade alternative. La scelta che infine è risultata essere quella più calzante ai requisiti è stata quella di sviluppare un *Office Add-in*.

3.2.1 Excel Add-in

L'utilizzo degli *Add-ins* di Office permette di migliorare l'esperienza d'uso di Word, Excel o un qualsiasi altro programma del pacchetto Office. Il termine Add-ins fa riferimento a componenti aggiuntive di varia natura, che abilitano diverse funzionalità.

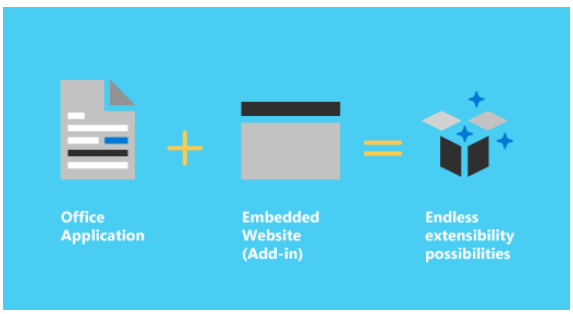


Figura 3.2: Office Add-in [o36b]

Questo tipo di Add-ins di Office è di tipo web, ovvero l'applicazione, come ad esempio Excel, legge il file manifest e collega i pulsanti del *ribbon* del componente e i comandi personalizzati del menu nell'interfaccia utente. Quando necessario, carica il codice JavaScript e HTML dell'Add-in, il quale viene eseguito in un browser o una Webview all'interno di una sandbox.

Seguendo le linee guida fornite da Microsoft è stata creata la prima versione dell'Add-in, aderente al template fornito. Il componente sfrutterà quindi il *taskpane*, in italiano anche chiamato riquadro delle attività, ovvero una sezione solitamente posizionata a destra nelle schermate delle applicazioni Office, per mostrare all'utente le informazioni utili e i controlli messi a sua disposizione, mentre utilizzerà delle funzioni implementate in JavaScript per la parte di logica.

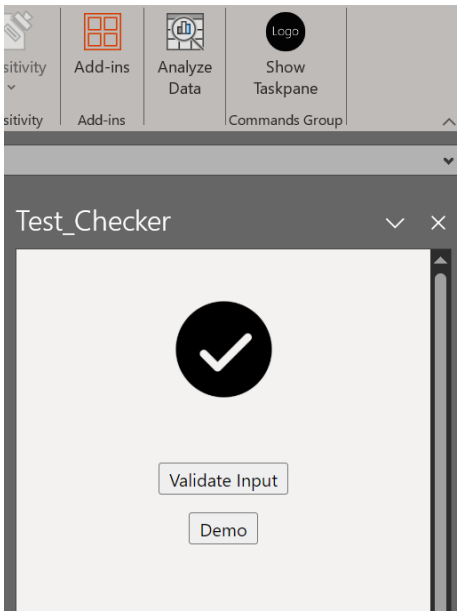


Figura 3.3: Excel Add-in task pane

Andando a modificare il codice all'interno del file `manifest.xml` e `taskpane.html` è stata creata un'interfaccia utente elementare che permettesse di effettuare le azioni principali, generare un template per la scrittura di un nuovo test e validarne poi la specifiche. Il risultato può essere osservato nella figura 3.3.

Per quanto riguarda invece la logica implementata è stato deciso di utilizzare una verifica della correttezza basata su *tokens* ed *espressioni regolari*, indicate anche come *regex*, come si può vedere qui di seguito in un estratto di questo sistema.

```
input = "Fail sul canale IMD_Fail"

const token = /Fail sul canale /i;
const regex = /[A-Za-z0-9_]+/i;

const pattern = new RegExp(token + regex, "i");
pattern.test(input)
```

In questo esempio è mostrata l'espressione `input` inserita dall'utente per simulare un fallimento del canale specificato. Le due variabili rappresentano quindi le due componenti dell'espressione, il `token`, ovvero la componente fissa che identifica il significato della variabile seguente, e la `regex` necessaria per riconoscere gli identificatori dei canali. Le due variabili sono poi concatenate in un oggetto di tipo `RegExp` per poter così richiamare il metodo `pattern.test` e verificare che l'input inserito corrisponda

Questo controllo viene effettuato per ogni componente variabile delle specifiche del test, come il momento di inizio e il valore da impostare. Basandosi poi sulla presenza e la correttezza di stringhe corrispondenti a questi pattern verrà controllata la correttezza nella loro correlazione. Esisteranno infatti pattern da inserire obbligatoriamente tra le specifiche come quelli appena descritti, e altri invece opzionali come quello utilizzato per definire la durata dell'errore, che può anche persistere naturalmente fino al termine della simulazione.

All'interno del comportamento sono poi stati inserite due diverse tipologie di controllo per migliorare l'usabilità dell'Add-in. Nell'estratto 2 è riportata la funzione impostata come *callback* alla pressione del bottone `Validate Input` visibile nella figura 3.3. Questa funzione si occupa di eseguire la validazione di tutte le celle all'interno di un range definito sulla base del template utilizzato per la definizione dei test, permettendo così di convalidare rapidamente anche un file preesistente.

```
async function runValidation() {
  await Excel.run(async (context) => {
    const sheet = context.workbook.worksheets.getActiveWorksheet();
    const range = sheet.getRange(testRange);
    range.load("values");
    clearResult();

    await context.sync();

    // Iterazione di tutte le celle del range
    range.values.forEach((row, rowIndex) => {
      const input = row[0]; // Estrazione del valore nella cella
      const cell = range.getCell(rowIndex, 0);
      let address = testCol + (rowIndex + testFirstRow);
      validateCell(input, cell, address);
    });
    await context.sync();
  });
}
```

Listing 2: Funzione di validazione Excel

La seconda tipologia di controllo invece si basa sulla capacità dell'Add-in di reagire a eventi generati in questo caso da Excel, come ad esempio l'avvenuta modifica di una cella. Nel codice mostrato in 3 è riportata proprio la registrazione dell'handler `handleCellChange` che dovrà verificare la validità del test inserito nella cella. Que-

sta registrazione avviene dinamicamente al caricamento dell'applicazione, richiamato dalla funzione `Office.onReady`.

```
function setupEventHandlers() {
  Excel.run(async (context) => {
    const sheet = context.workbook.worksheets.getActiveWorksheet();
    sheet.onChanged.add(handleCellChange);
    await context.sync();
  });
}

async function handleCellChange(event) {
  await Excel.run(async (context) => {
    const sheet = context.workbook.worksheets.getActiveWorksheet();
    const range = sheet.getRange(event.address);
    range.load("values, address");

    await context.sync();

    const address = range.address.split("!")[1];
    // Verifica di appartenenza della cella al range predefinito
    if (address.startsWith(testCol) && address.split(testCol)[1] >=
    ↪  testFirstRow) {
      const input = range.values[0][0];
      clearResult();
      validateCell(input, range, address);
      await context.sync();
    }
  });
}
```

Listing 3: Funzione di validazione automatica cella

Dopo aver eseguito il comando `npm start` all'interno della cartella contenente il progetto l'Add-in verrà avviato diventando così operativo. Quello che si può osservare nella figura 3.4 è il risultato della pressione sul pulsante `Demo` il quale come anticipato andrà a generare il template per la definizione del test. Per rendere più immediata la comprensione del risultato della validazione è stato scelto di impostare anche il colore dello sfondo delle celle in maniera che ne rispecchiasse l'esito. In questo caso si può notare come lo sfondo rosso della cella corrisponda infatti a due messaggi di errore mostrati attraverso l'html nel riquadro delle attività, i quali indicano una errata conformazione dell'input in corrispondenza dei pattern relativi al momento di inizio del fallimento e il suo valore.

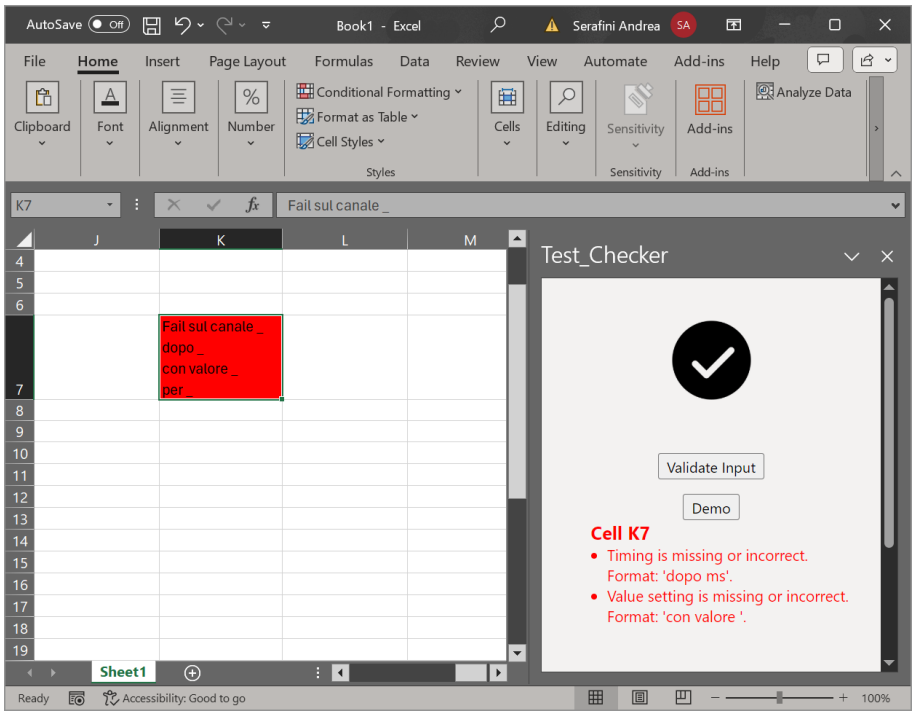


Figura 3.4: Excel Add-in test errato

Nella figura 3.5 si può invece vedere un esempio di test correttamente descritto, con l'interfaccia grafica che ne comunica appunto la validità all'utente, impostando uno sfondo verde alla cella. Infine l'ultimo caso è riportato nella figura 3.6, nella quale si può vedere un foglio Excel nel quale oltre a un test corretto l'utente ha inserito anche un test che ha generato un *warning*, indicato da una cella con lo sfondo giallo. Questo richiamo all'attenzione dell'utente serve a indicare che nonostante i requisiti necessari affinché il test risulti valido siano tutti soddisfatti, è stata rilevata una porzione di testo che non rientra negli standard definiti, come la durata `fino alla fine` inserita in questo esempio. La decisione di aggiungere questa casistica tra le possibilità deriva dall'eventuale necessità da parte dello stratega di eseguire un test diverso e non ancora supportato dal sistema, oppure di aggiungere delle note ai test descritti, per cui potrebbe aggiungere volontariamente del testo che non potrà poi essere processato dal tool di generazione del codice, ma che non precluda l'avanzamento dei test correttamente formattati.

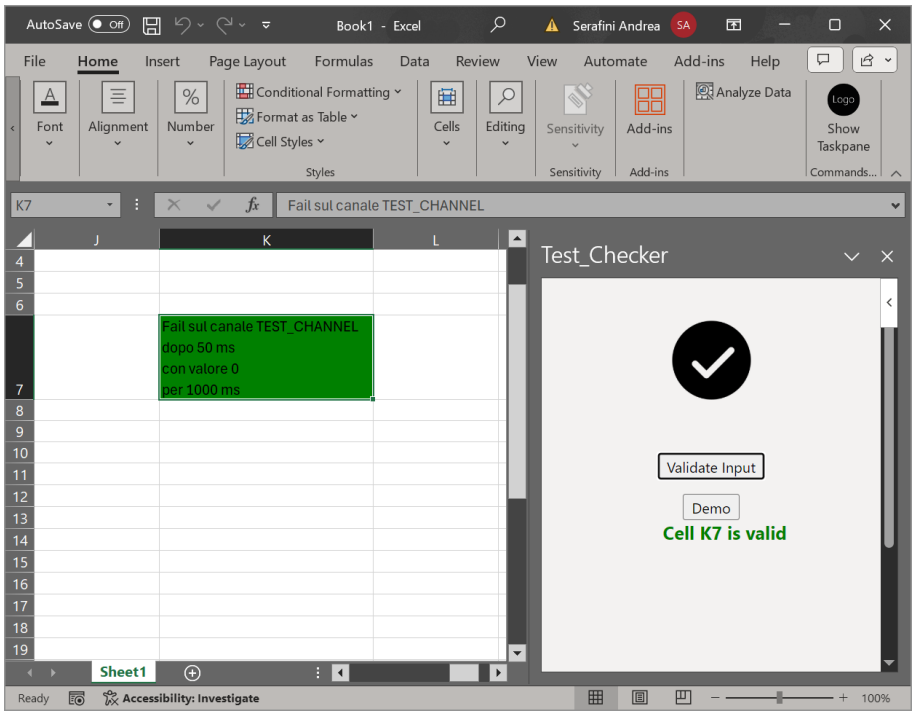


Figura 3.5: Excel Add-in test corretto

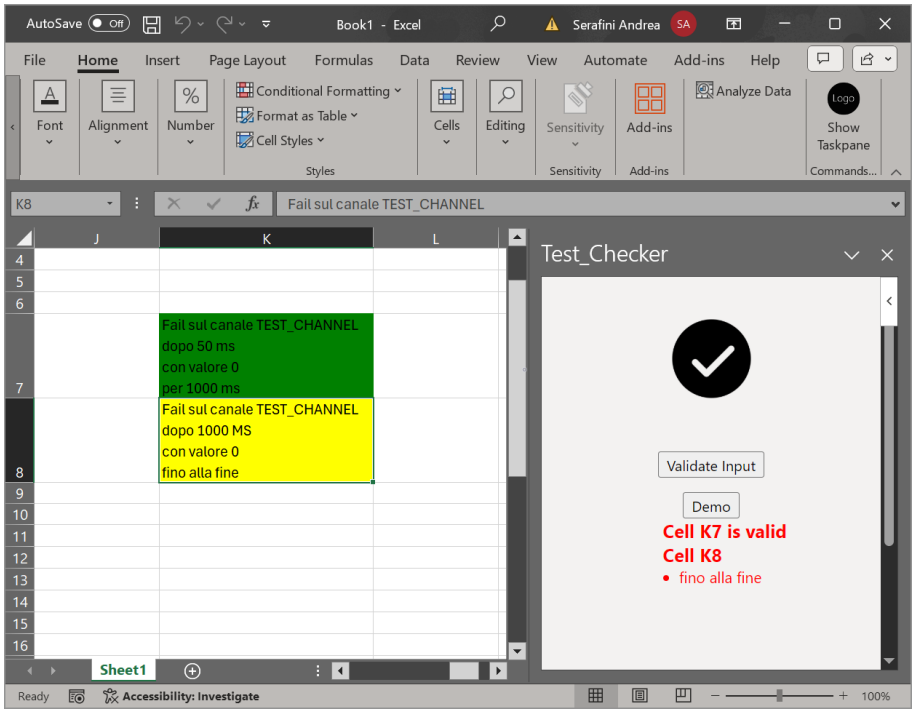


Figura 3.6: Excel Add-in warning

Una volta raggiunto uno stato soddisfacente di sviluppo di questo componente si è passati alla fase di distribuzione del SW agli utenti finali per effettuare una prima fase di test di usabilità, ed è proprio in questa fase che si sono evidenziate alcune problematiche. Esistono varie modalità di rilascio e pubblicazione messe a disposizione

da Microsoft [o36a], ma non tutte sono facilmente attuabili nell'ambito di sviluppo di questo progetto. Durante le fasi descritte fino a questo punto il SW era stato testato sfruttando la metodologia definita come *sideloading*, la quale permette durante il processo di sviluppo di eseguire l'Add-in in locale, eseguendolo grazie a Node.js all'indirizzo `https://localhost:3000/`. La metodologia successiva proposta è quella del *Network share*, che prevede di pubblicare l'Add-in su un server accessibile diverso da `localhost`, tuttavia anche questo metodo viene ancora indicato solo per le fasi sviluppo. Le alternative proposte per la fase di rilascio ufficiale prevedono l'utilizzo del catalogo *SharePoint* o di *Microsoft Exchange server*, entrambe scelte non ottimali a causa delle complicazioni risultanti e a fronte della quantità molto limitata di utilizzatori di questo strumento all'interno dell'azienda. Per questo motivo grazie al supporto e ai consigli di un collega più esperto è stato deciso di cambiare il paradigma sul quale si basava l'Add-in, passando da un'implementazione basata su JavaScript, Node.js e un'interfaccia definita in linguaggio HTML, a una basata su Visual Basic for Applications (VBA), un'implementazione di Visual Basic inserita all'interno di alcune applicazioni Microsoft come ad esempio la suite di Office. Nata ormai più di trent'anni fa, nonostante lo stretto legame con Visual Basic VBA non permette di eseguire applicazioni stand-alone, ma può essere utilizzato per controllare quasi ogni aspetto dell'applicazione ospite, manipolandone i menu e le barre degli strumenti. I principali oggetti di questo linguaggio sono le *subroutine* e le *funzioni*. Le prime, anche dette procedure o macro, permettono di eseguire automaticamente una serie di azioni all'interno dell'ambiente nel quale viene lanciata l'applicazione, mentre le funzioni richiedono almeno una variabile indipendente con il relativo valore numerico o testuale in ingresso. [Wikb]

Si è quindi proceduto alla conversione dalla struttura precedente a quella basata su VBA. Il nuovo formato dell'Add-in si compone di due soli file, uno con estensione `.xlam` e uno con estensione `.xlsm`, la prima indica il file di un Add-in con abilitata la possibilità di eseguire delle macro, descritte in precedenza, e permette di aggiungere nuove funzionalità a Excel, mentre la seconda indica una cartella di lavoro, con eventuali fogli di lavoro compresi all'interno, anch'essa con la possibilità di eseguire delle macro.

XLAM

Aprendo file contenente l'Add-in sviluppato si visualizzerà la schermata riportata in figura 3.7 nella quale è possibile vedere il *ribbon*, in italiano indicato con il nome di barra multifunzione, personalizzato con i pulsanti necessari. Dato che il formato `.xlam` nella sua effettiva funzione altro non è che un formato di compressione utilizzato per compattare tutte le informazioni necessarie a Excel con una minor occupazione di memoria, per modificarne alcune parti, come ad esempio le componenti grafiche dei ribbon, è necessario estrarne il contenuto. Per poter effettuare quindi la modifica e l'aggiunta del codice XML riportato in 4 relativo a questo progetto è stato utilizzato lo strumento Office RibbonX Editor [And].

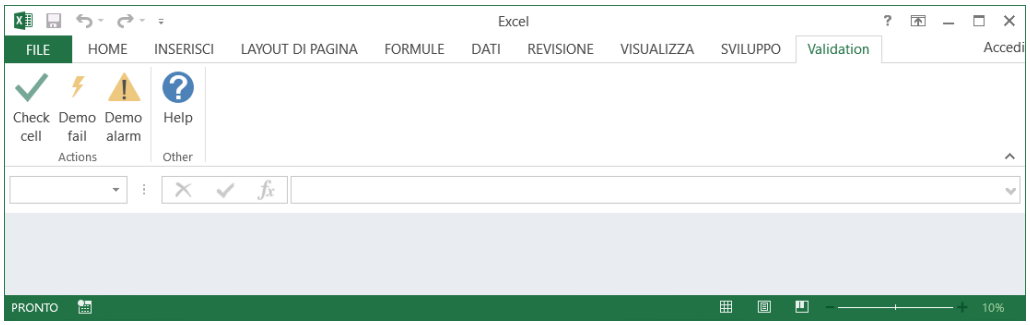


Figura 3.7: Barra multifunzione personalizzata

```
1  <customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
2      <ribbon startFromScratch="false">
3          <tabs>
4          <tab id="customTab" label="Validation">
5              <group id="customGroup" label="Actions">
6                  <button id="customButton1" label="Check cell"
                       ↪  imageMso="AcceptProposal" size="large"
                       ↪  onAction="CheckCallback"/>
7                  <button id="customButton2" label="Demo fail"
                       ↪  imageMso="AutoFormatChange" size="large"
                       ↪  onAction="DemoFailCallback"/>
8                  <button id="customButton3" label="Demo alarm"
                       ↪  imageMso="MacroSecurity" size="large"
                       ↪  onAction="DemoAlarmCallback"/>
9              </group>
10             <group id="customGroup2" label="Other">
11                 <button id="customButton4" label="Help" imageMso="Help"
                       ↪  size="large" onAction="HelpCallback"/>
12             </group>
13         </tab>
14         </tabs>
15     </ribbon>
16 </customUI>
```

Listing 4: Barra multifunzione personalizzata

Una volta avviato Excel è possibile aprire una finestra secondaria, mostrata nella figura 3.8, dalla quale è possibile scrivere codice all'interno degli oggetti di Excel, ovvero i fogli e le cartelle di lavoro, oppure creare nuovi moduli indipendenti. Nel riquadro a destra è possibile vedere il modulo `Callbacks` contenente le subroutine necessarie per collegare i pulsanti impostati nel listato 4 con i relativi comportamenti. Infine nell'estratto riportato in 5 come esempio di traduzione è mostrata la funzione utilizzata per validare una stringa con un'espressione regolare.

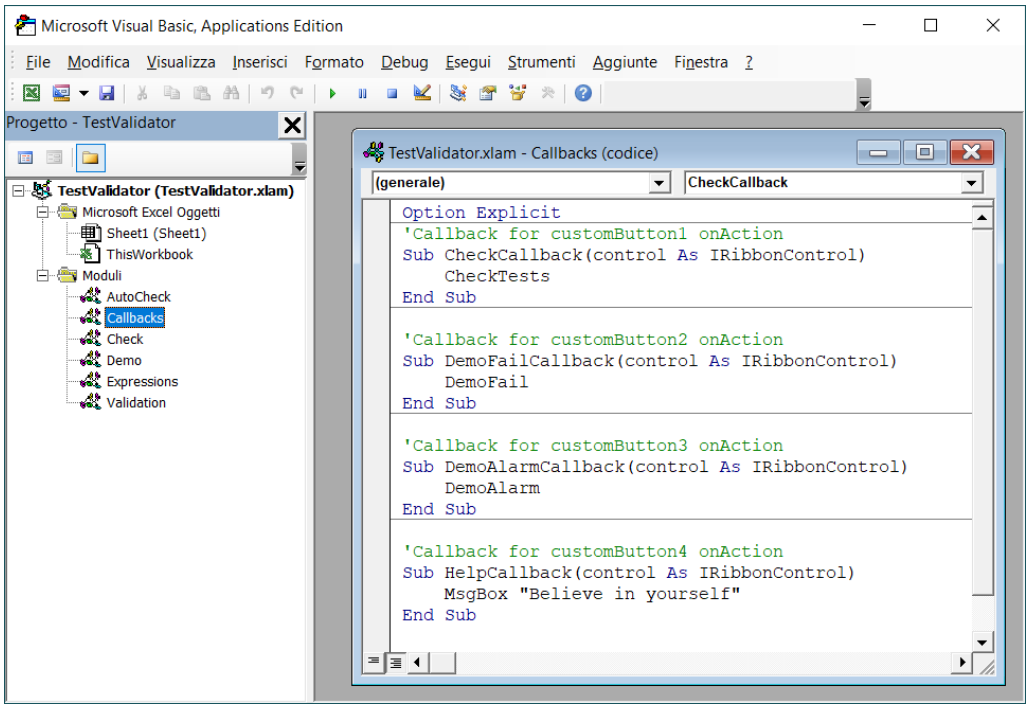


Figura 3.8: Interfaccia per VBA di Excel

```vba
' Validate a string against a pattern
Public Function ValidateWithRegex(inputString As String, pattern As
↪  String) As Boolean
    Dim regex As Object
    Set regex = CreateObject("VBScript.RegExp")

    With regex
        .pattern = pattern
        .IgnoreCase = True
        .Global = False
    End With

    ValidateWithRegex = regex.Test(inputString)
End Function
```

Listing 5: Funzione di verifica regex in VBA

XLSM

Il secondo file sviluppato come già detto è quello relativo alla cartella di lavoro, o *workbook*, da utilizzare per la definizione dei test. In questo workbook oltre ad aver abilitato l'utilizzo delle macro sono state aggiunte delle subroutine in VBA delle quali viene riportato un estratto in 6. Le funzioni `Workbook_Open` e `Workbook_BeforeClose` sono state implementate con lo scopo di abilitare e conseguentemente disabilitare l'Add-in solo nel momento in cui venga aperto un file di test basato sul template in oggetto. In questo modo oltre a evitare confusione all'utente che potrebbe già avere svariati ribbon specifici per altri scopi e quindi abilitati su altri

tipi di fogli di lavoro, si prevengono anche errori causati dal fatto che diversi file non avrebbero la formattazione attesa dal tool. L'ultima funzione invece è necessaria per far si che questa versione dell'Add-in si comporti in maniera analoga a quella precedente, ovvero reagendo alla modifica delle celle richiamando la funzione di validazione definita dal file XLAM

In questo modo allo stratega basterà creare un nuovo file Excel copiando il template fornito e manterrà in ogni nuovo workbook le stesse funzionalità di validazione senza estendere a tipologie di file diverse.

```
1  Option Explicit
2  Private WithEvents App As Application
3
4  Private Sub Workbook_Open()
5      ' Abilita il componente aggiuntivo all'apertura del file
6      EnableAddIn "TestValidator", True
7      Set App = Application
8      ' Restore application settings
9      ' Application.ScreenUpdating = True
10     ' Application.Calculation = xlCalculationAutomatic
11 End Sub
12
13 Private Sub Workbook_BeforeClose(Cancel As Boolean)
14     ' Disabilita il componente aggiuntivo alla chiusura del file
15     EnableAddIn "TestValidator", False
16     Set App = Nothing
17 End Sub
18
19 Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Target As
   ↪  Range)
20     On Error Resume Next
21     Application.Run "CheckOnChange", Sh, Target
22     On Error GoTo 0
23 End Sub
```

Listing 6: Subroutine per il caricamento dell'AddIn e la sua reattività

3.3 Code generator

In questa sezione verrà descritto l'applicativo sviluppato per la generazione dei test che si occuperà quindi di tradurre in sequenze eseguibili le specifiche dei test definite nel foglio Excel di cui si è parlato in precedenza.

In figura 3.9 viene mostrata l'interfaccia grafica presentata all'utente da questo tool. Nella parte alta si possono osservare i tre campi richiesti in ingresso per poter eseguire la traduzione, la sequenza base della quale si parlerà meglio in seguito, un database e un file di test.

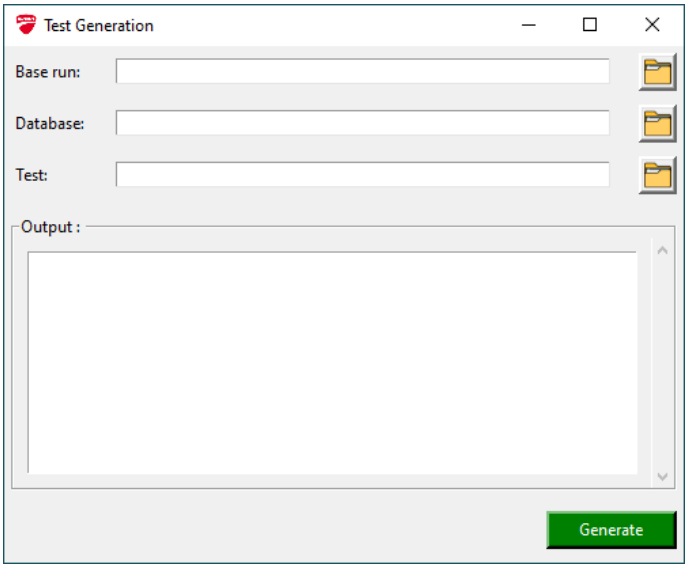


Figura 3.9: Interfaccia utente per la generazione dei test

Per quanto riguarda il file identificato dalla dicitura database si fa riferimento a un file Excel contenente le associazioni tra l'identificatore di un canale e il suo corrispettivo percorso all'interno dell'HIL. Un estratto di questo file riportato nella tabella 3.3 permette appunto di vedere la correlazione tra i valori dei segnali che vengono inseriti dallo stratega nelle specifiche dei test e il riferimento che verrà inserito all'interno del codice generato per inizializzare un oggetto di tipo `ChannelReference`.

Signals	Channel Reference
TWGRD1_r	Targets/Controller/Simulation Models/Models/LAP_-MANAGER/Inports/TWGRD1_r
TWGRD2_r	Targets/Controller/Simulation Models/Models/LAP_-MANAGER/Inports/TWGRD2_r
IMD_Fail	Targets/Controller/Simulation Models/Models/HIL_-BMS_SIM/Inports/I/HVB_Diag_ImdLoR_Fault

Tabella 3.3: Esempio tabella riferimenti dei canali

3.3.1 Sequenze base

Per la definizione delle strutture di base dalle quali originare le sequenze di test è stato necessario partire innanzitutto dall'analisi delle sequenze utilizzate finora sull'applicativo Stimulus Profile Editor. Per gli esempi verrà tenuto in considerazione l'approccio utilizzato per i test MotoE.

Nella figura 3.10 si può vedere come le sequenze solitamente eseguite in successione sono due, `Automatic_Mapping` e `RunBase`, dove la prima permette di impostare in maniera automatica le mappe da utilizzare come punto di partenza sia per mantenere una ripetibilità dei test sia per evitare che un residuo di quelli precedenti ne influenzi il risultato. La seconda sequenza, riportata in maggior dettaglio nella figura 3.11, ha invece il compito di eseguire la simulazione vera e propria, con l'aggiunta del test definito di volta in volta.

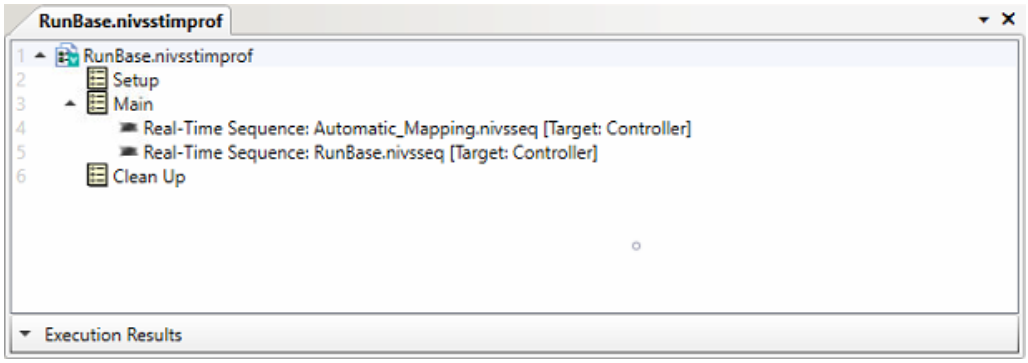


Figura 3.10: Stimulus profile

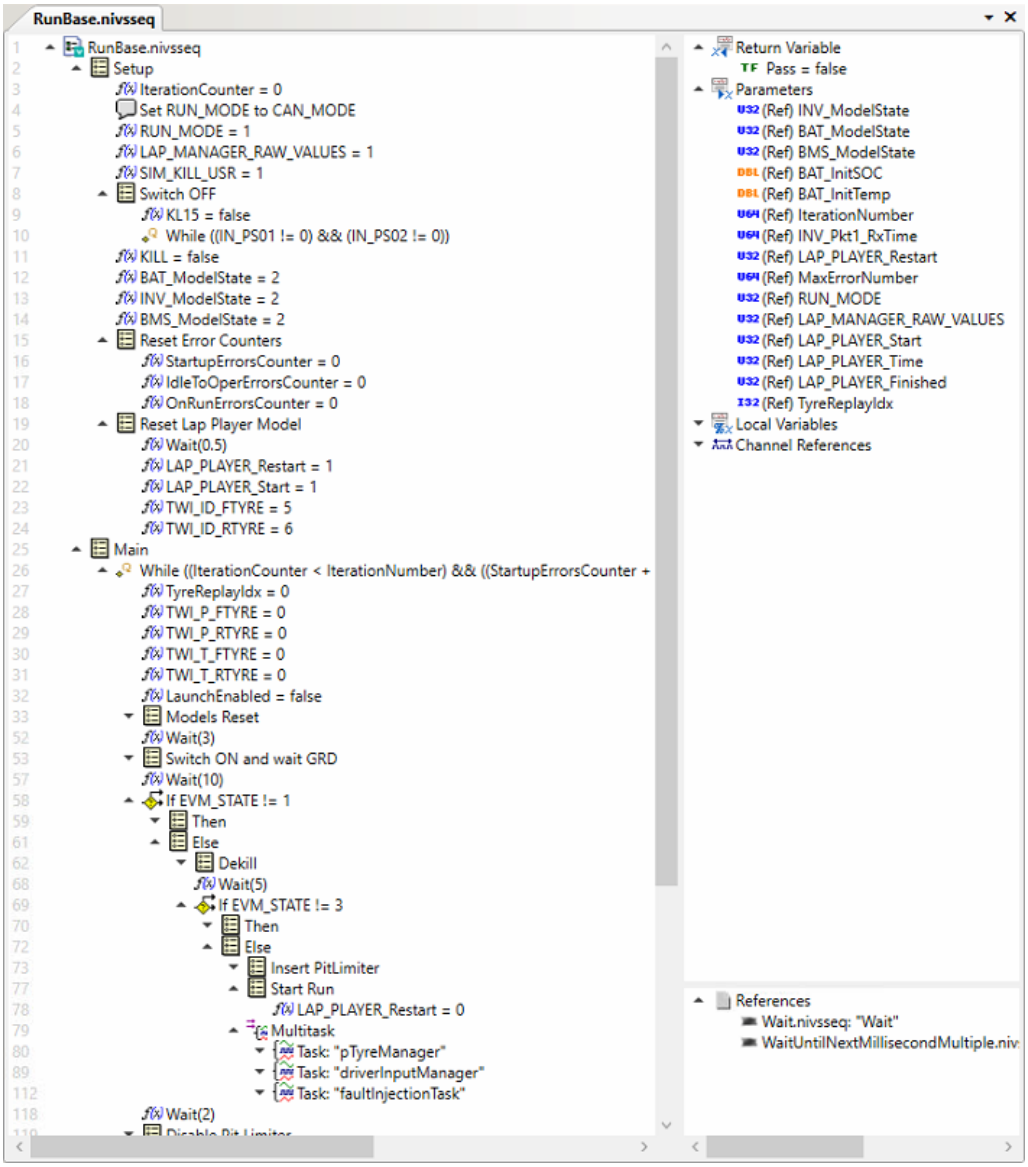


Figura 3.11: Sequenza Real time

Una volta compreso il comportamento della sequenza `RunBase`, sia visualizzandola nel suo formato grafico che analizzandola nel formato XML per verificarne le effettive implementazioni, è stata creata la corrispondente versione in Python della quale è riportato un estratto minimale in 7. Nell'estratto è anche possibile vedere come successivamente per una maggior comodità in fase di esecuzione sia stata aggiunta anche la chiamata alla sequenza `Automatic_Mapping` anch'essa tradotta. Per rendere poi ancora più completo sarà sufficiente applicare le stessa logica di traduzione per le altre sequenze di base, siano esse differenti nel numero di giri di pista simulati o nella moto alla quale fanno riferimento.

```
1  from niveristand import ...
2
3  @nivs_rt_sequence
4  def main_sequence( ... ):
```

```
5       #Setup
6       Automation_Stop_Flag.value = 0
7       #Main
8       while ( ... ):
9       with multitask() as mt:
10        @task(mt)
11        def pTyreManager():
12          while ( ... ):
13          stop_task(pTyreManager)
14          nivs_yield()
15      #Cleanup
16      LAP_PLAYER_Restart.value = 1
17
18      return res.value
19
20  @nivs_rt_sequence
21  def run_sequence(alarm1, alarm2):
22      #CHANNEL REFERENCES
23
24      ret.value = main_sequence( ... )
25      return ret.value
26
27  @nivs_rt_sequence
28  def mapping_sequence( ... ):
29      return res.value
30
31  @nivs_rt_sequence
32  def run_mapping_sequence():
33
34      res = BooleanValue(False)
35      res.value = mapping_sequence( ... )
36      return res.value
37
38  def run_deterministic():
39      map_result = run_py_as_rtseq(run_mapping_sequence)
40      if map_result is True:
41          run_result = run_py_as_rtseq(run_sequence)
42          return ("Test Passed!" if run_result else "Test Failed
            ↪  (expected)!")
43
44  if __name__ == '__main__':
45      print(run_deterministic())
```

Listing 7: Sequenza base MotoE

Dopo aver definito i file e le informazioni necessari in ingresso al tool per poter generare le sequenze di test, si è proceduto a implementare il tool stesso. Basato su uno script Python e con una semplice interfaccia utente implementata attraverso la libreria Tkinter, alla pressione del pulsante `Generate` verificherà la presenza di tutti gli input necessari prima di passare al passo successivo. A verifica avvenuta verrà quindi aperto il file dei test e, valutandone nuovamente la correttezza, verranno estratte le specifiche dei test da eseguire. Dopo aver creato una cartella, con il nome corrispondente a quello del file dei test per favorire una possibile ricerca in

un secondo momento, verranno generate ciclicamente tante sequenze quante sono le *run*, termine alternativo per indicare le singole righe del file che corrisponde quindi a un'esecuzione completa della simulazione. Per ogni run gli step eseguiti saranno i seguenti:

- **Estrazione riferimenti dei canali**: dopo aver utilizzato le regex per estrarre dalla specifica gli identificativi dei canali necessari, essi verranno ricercati all'interno del file database riportato nella tabella 3.3 per generare delle stringhe con il formato `CHANNEL_ID = ChannelReference('path/from/DB')`
- **Generazione fail**: per ogni blocco logico del test, ad esempio un singolo fallimento su un canale con valore e tempistiche, verranno estratte sempre utilizzando le regex le variabili definite dallo stratega per poi essere inserite in un blocco di codice come quello riportato in 8, generato attraverso funzioni sviluppate appositamente per ritornare un array di stringhe correttamente formattate
- **Inserimento linee di codice**: creando poi una copia della sequenza base scelta verrà generato un file di destinazione, il cui nome dipenderà dal codice dato alla run attualmente analizzata, all'interno della cartella precedentemente creata. Ricercando ora all'interno del codice specifici placeholder come `#Generated channel refences` o `#Generated tasks` il tool andrà a inserire le linee di codice appena generate nelle apposite sezioni all'interno della sequenza

```
1  #Start#
2  @task(mt)
3  def TaskTest1():
4      while Automation_Stop_Flag.value == 0 and LAP_PLAYER_Finished.value
       ↪  < 1 and (EVM_STATE.value == 3):
5          if ((LAP_PLAYER_Time.value > 100) and (LAP_PLAYER_Time.value <
           ↪  110)):
6              fault(Alarm_received, 3)
7          else:
8              clearfault(Alarm_received)
9          wait_until_next_ms_multiple(5)
10         nivs_yield()
11     clearfault(Alarm_received)
12     stop_task(TaskTest1)
13 #Generated tasks
```

Listing 8: Task generato automaticamente per un test

3.4 Test executor

L'obiettivo principale di questo tool era quello di automatizzare il processo di esecuzione dei test per cui, attraverso l'interfaccia utente mostrata nella figura 3.12, l'utente potrà aggiungere un numero di sequenze a piacere tra quelle generate in precedenza prima di avviarne l'esecuzione. I pulsanti `Start` e `Stop` rimarranno disabilitati fino all'aggiunta di almeno una sequenza il primo, mentre fino all'avvio dell'esecuzione il secondo. La finestra `Output` presente nella parte bassa dell'interfaccia si popolerà dei risultati ottenuti dalle esecuzioni dei test a seguito della loro terminazione.

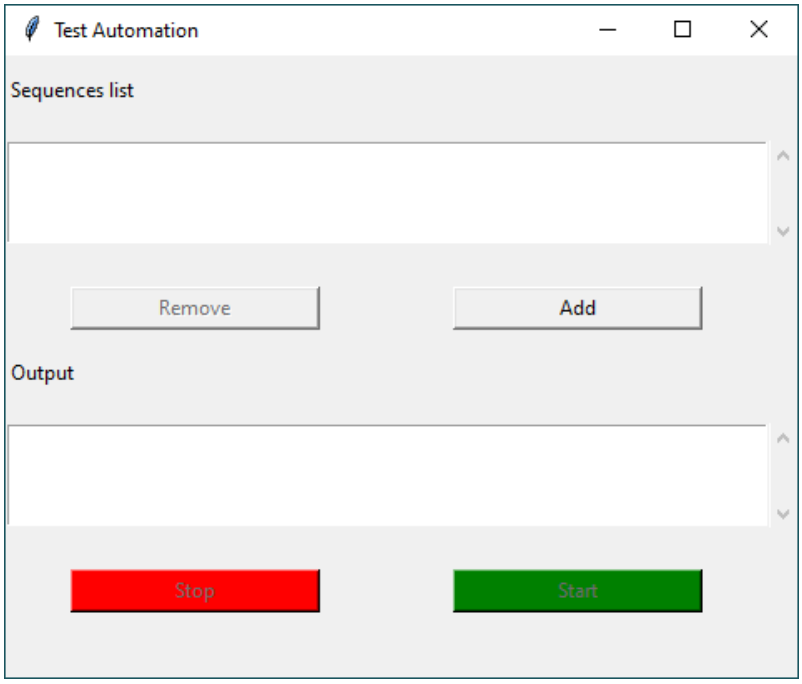


Figura 3.12: Interfaccia utente per l'esecuzione dei test

3.4.1 Multithreading

A causa della natura delle API sviluppate da NI l'esecuzione di una sequenza risulta essere un task bloccante per lo script Python che la avviasse. Per soddisfare il requisito di responsività del tool necessaria all'utente per poter interrompere una campagna di test anticipatamente, premendo sul pulsandte `Stop`, il quale avvierà sempre parallelamente la sequenza riportata in 9, è stato utilizzato il modulo `threading` di Python, il quale costituisce un'interfaccia di alto livello per la gestione dei processi.

All'interno dello script che si occupa di gestire l'interfaccia grafica è stato quindi inserito un `worker thread` che verrà avviato in seguito alla pressione del pulsante `Start`. A questo processo verranno passate in ingresso due variabili, una coda di percorsi che puntano alle sequenze da eseguire e una di tipo `Event`, proveniente dallo stesso modulo, che permette di segnalare al processo con un evento che è stata richiesta l'interruzione da parte dell'utente e che quindi non dovrà essere valutata la parte restante della coda di sequenze.

Dal modulo appena descritto è stato importato anche il tipo di dato `Lock` che, con un nome esplicativo della sua funzione, permette al processo responsabile dell'esecuzione delle sequenze di scrivere l'output da loro generato all'interno della finestra `Output` del processo padre.

Alla pressione del pulsante

```
from niveristand import nivs_rt_sequence, run_py_as_rtseq,
↪ realtimesequencetools
from niveristand.clientapi import ChannelReference

@nivs_rt_sequence
def run_sequence():

    Automation_Stop_Flag = ChannelReference ('Targets/Controller/User
    ↪ Channels/Automation_Stop_Flag')
    Automation_Stop_Flag.value = 1

if __name__ == '__main__':
    print("Stop test")
    print(run_py_as_rtseq(run_sequence))
```

Listing 9: Semplice sequenza real-time in Python

3.4.2 Risultati dei test

Logger

La chiamata alla funzione `run_py_as_rtseq` permette di avere come valore di ritorno solamente un booleano che perciò anche se impostato da una condizione in sé significativa, come ad esempio che la simulazione si è svolta nella sua interezza e non interrotta prima, risulta comunque essere poco indicativo del vero svolgimento del test. Per questo motivo è stato ricercato un metodo che permettesse di avere un log più dettagliato dei valori assunti durante la simulazione da determinati canali che fungessero da cartine al tornasole per la comprensione del reale stato del sistema. All'interno delle API fornite da NI per Python è presente al metodo deprecato `NIVeriStand.Workspace2.StartDataLogging` che di conseguenza non può essere utilizzato per questo scopo. Approfondendo la ricerca è stato poi preso spunto da un'implementazione reperita in un repository [che] per arrivare a quella che è la versione attuale del servizio di logging.

Per risolvere questo problema è stata quindi utilizzata la libreria *Python.NET*, anche indicata come `pythonnet` [pyt]. Questo pacchetto permette agli sviluppatori che utilizzano Python di integrare facilmente con il Common Language Runtime (CLR) di .NET, per cui ha reso possibile utilizzare le API pensate per .NET all'interno dello script Python sviluppato.

```
import clr
clr.AddReference("NationalInstruments.VeriStand.ClientAPI")
from NationalInstruments.VeriStand.ClientAPI import Logging
```

```
4
5  CSVLogFile_Ref = TextLogFile(location,
   ↪  FileConflictOperation.CreateUnique, Delimiter.Comma, 3)
6  CSVLogFile_Ref.AddChannel("alarm","Targets/Controller/User
   ↪  Channels/Alarm_received", False)
7
8  #Start Data Logging Session
9  Data_Logging_Manager_Ref.StartDataLoggingSession("LoggingSession",
   ↪  Logging_Specification_Ref)
10 #Stop Data Logging Session
11 return Data_Logging_Manager_Ref.StopDataLoggingSession("LoggingSession")
```

Listing 10: Utilizzo API .NET in Python

Utilizzando le funzioni riportate nell'estratto 10 è risultato quindi possibile selezionare un insieme di canali dei quali tenere traccia, impostando alcuni parametri, come ad esempio la frequenza di campionamento, memorizzandone i valori in un file `.csv`. La generazione di questo file permetterà, attualmente all'utente ma in futuro a un sistema automatizzato, di generare una reportistica dettagliata sull'andamento dei test, verificando non solo il comportamento nominale del sistema, ma anche che i fallimenti siano stati generati correttamente e quale sia stata la risposta della ECU all'evento, potendolo anche confrontare con una situazione già attesa a priori.

3.5 GRD watcher

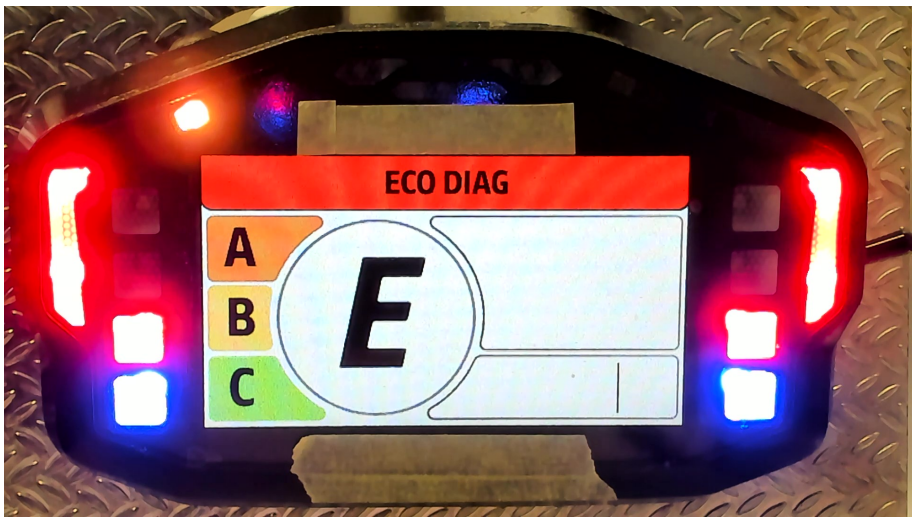


Figura 3.13: GRD, cruscotto sviluppato per le Ducati da competizione

Come anticipato in questo capitolo verrà illustrato l'ultimo tool del sistema, incaricato di mantenere un controllo visivo sul GRD. Per raggiungere questo obiettivo è stata utilizzata la Logitech Brio mostrata in figura 3.14, una webcam commerciale facilmente reperibile ma di buona qualità la quale oltre a supportare una risoluzione 4K permette di personalizzare lo zoom e la frequenza alla quale vengono inviate le informazioni. Per ottenere i risultati migliori è stata utilizzata in modalità FullHD con una risoluzione di 1920x1080 e una frequenza di 60 fps.

Figura 3.14: Webcam utilizzata per il progetto

Anche questo tool è stato sviluppato come script Python, strutturando il lavoro su diversi processi paralleli per aumentare la velocità alla quale vengono processate le informazioni per ogni frame. Le tipologie di thread che vengono utilizzate durante l'esecuzione sono:

- **Rate Counter**: si occupa di mantenere un contatore delle iterazioni e calcola il rate al quale vengono eseguite le operazioni

- **Video Stream**: utilizzando la libreria OpenCV raccoglie i frame dalla sorgente video e li mette a disposizione degli altri thread
- **OCR**: si occupa di riconoscere la componente testuale degli allarmi
- **Led Analyzer**: identifica i led accesi e ne traccia il comportamento
- **Logger**: si occupa di gestire il salvataggio delle informazioni raccolte e della comunicazione con gli altri tool

3.5.1 OCR allarmi

I sistemi di OCR, acronimo che indica la *Optical Character Recognition*, sono programmi dedicati al rilevamento e riconoscimento di caratteri contenuti in un'immagine e al loro trasferimento in forma digitale. In particolare per questo progetto è stato utilizzato il modulo EasyOCR, caratterizzato dalla sua flessibilità e facilità di utilizzo.

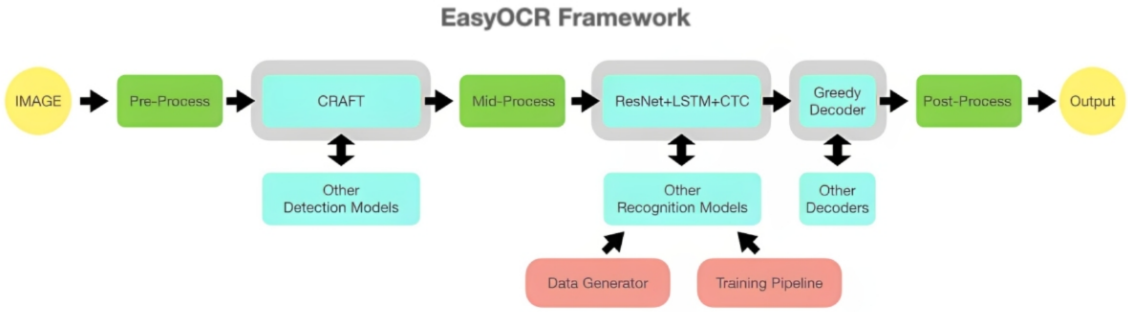


Figura 3.15: Framework EasyOCR[Jai]

Prima di essere analizzati i frame subiscono una fase di *preprocessing*. In una prima fase si era tentato utilizzando una soglia adattiva determinata utilizzando il parametro `ADAPTIVE_THRESH_GAUSSIAN_C` di OpenCV che applica una soglia calcolata come somma pesata secondo una funzione gaussiana sottraendo poi la costante `C`, ma i risultati non erano soddisfacenti. La soluzione adottata infine è stata una semplice conversione `COLOR_BGR2GRAY`, la quale converte i colori del frame dalla codifica BGR a una codifica in scala di grigi prima di sottoporre il nuovo frame al lettore di caratteri.

Dopo una prima implementazione di questa sezione del codice si è notato grazie al processo focalizzato sulla misurazione della frequenza di elaborazione che il riconoscimento dei caratteri risultava essere di gran lunga il più lento causando spesso perdite di frame sufficienti per non percepire un allarme apparso per un tempo limitato.

Per risolvere questa problematica sono state attuate alcune misure di mitigazione andando ad agire innanzitutto sull'immagine passata in ingresso al sistema di OCR. Per limitare la quantità di calcoli necessari per il preprocessing e l'estrazione

dei caratteri di ogni frame è stato necessario ridurre la dimensione delle immagini valutate. Grazie alla natura statica del simulatore è stato possibile impostare il sistema composto da webcam e GRD in maniera che l'inquadratura fosse fissa, in maniera da poter estrarre dei riferimenti da una singola immagine per definire le ROI, ovvero le *Region Of Interest*, aree delimitate all'interno delle quali si trovano le informazioni ricercate. Data questa necessità è stata presa in considerazione la possibilità, successivamente posposta alla fase di progetto come possibile sviluppo futuro, di progettare un supporto stabile che vincolasse le due componenti creando una struttura che permettesse in futuro di ripristinare un'inquadratura specifica per un tipo di HW sottoposto a test. Per questa prima fase di sviluppo è stato invece scelto di sviluppare un piccolo script di supporto che permettesse all'utente di generare attraverso alcune semplici interfacce grafiche contenente i parametri necessari ogni qualvolta venga variata l'inquadratura. Questo strumento verrà citato anche nelle sezioni successive, in quanto utilizzato per l'estrazione di altri parametri oltre a quelli necessari al processo di OCR.

Una volta definite quindi le cinque aree da sottoporre all'analisi, delle quali è possibile vedere un esempio nella figura 3.16, il tool per ogni frame verificherà la presenza di testo all'interno di ognuna di esse. Per ridurre ulteriormente il carico di lavoro al *reader* è stato impostato innanzitutto un sottoinsieme di caratteri attesi, basato sulla conoscenza delle possibili stringhe mostrate, dopodiché, in seguito a test effettuati per verificare sia rapidità che correttezza dei risultati, è stata impostata la ricerca per singole parole piuttosto che per paragrafi. Ciò significa che in questa fase il sistema, prendendo ad esempio la figura sopracitata e in particolare la prima ROI relativa all'allarme, non restituirà un valore letto unico corrispondente alla stringa `"ECO DIAG"`, bensì i due valori separati `"ECO"` e `"DIAG"` che poi vengono successivamente concatenati risparmiando tempo di esecuzione. Il risultato finale è riportato nella figura 3.17.

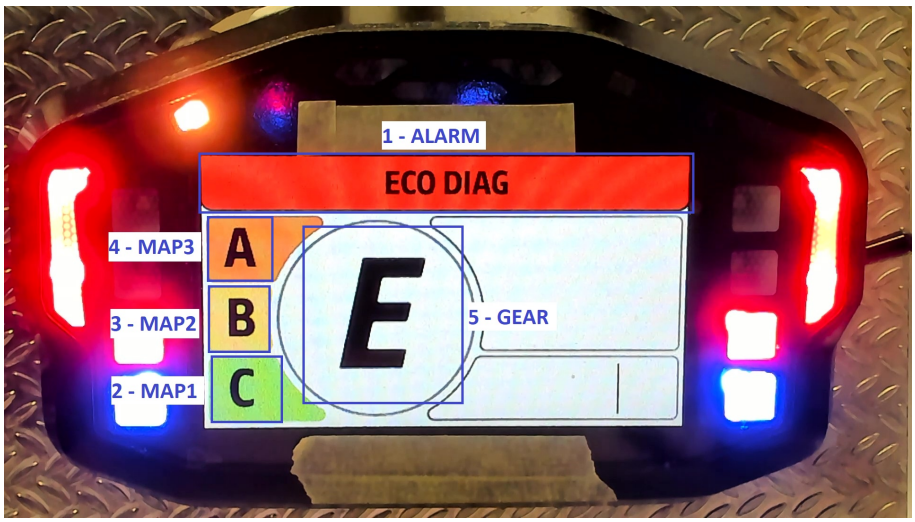


Figura 3.16: Esempio di regioni del frame contenenti testo

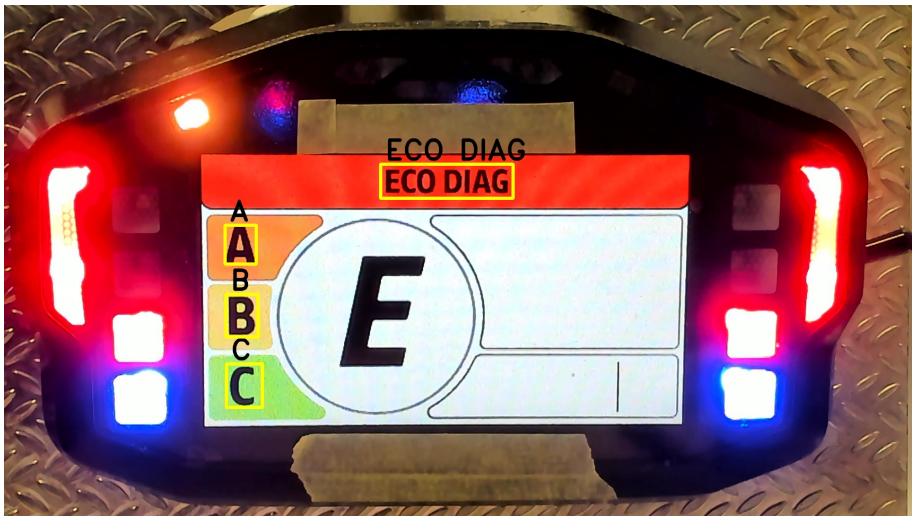


Figura 3.17: Esempio risultati ottenuti con EasyOCR

3.5.2 Riconoscimento led

Per quanto riguarda il riconoscimento dei led è stata seguita una logica simile a quella descritta in precedenza, strutturando anche in questo caso il thread in maniera che prenda in ingresso delle aree del frame originale per poi effettuare il preprocessing dell'immagine prima di estrarre le informazioni relative ai led. Come prima cosa sono state quindi definite le ROI mostrate nella figura 3.18, derivate anch'esse dai parametri impostati dall'utente analogamente a quelle per il riconoscimento dei caratteri, dopodiché vengono effettuate una serie di trasformazioni sull'immagine:

- conversione da BGR a scala di grigi
- applicazione di una sfocatura gaussiana
- applicazione di una soglia binaria su valori prossimi al bianco
- varie iterazioni di erosione
- varie iterazioni di dilatazione

Una volta eseguite queste operazioni le zone bianche verranno etichettate in base alla prossimità tra i pixel per definire le diverse regioni connesse, filtrate poi in base alla loro dimensione per evitare errori. La maschera risultante da queste operazioni è riportata nella figura 3.19. Questa maschera verrà poi utilizzata per identificare i contorni delle zone bianche come mostrato invece nella figura 3.20.

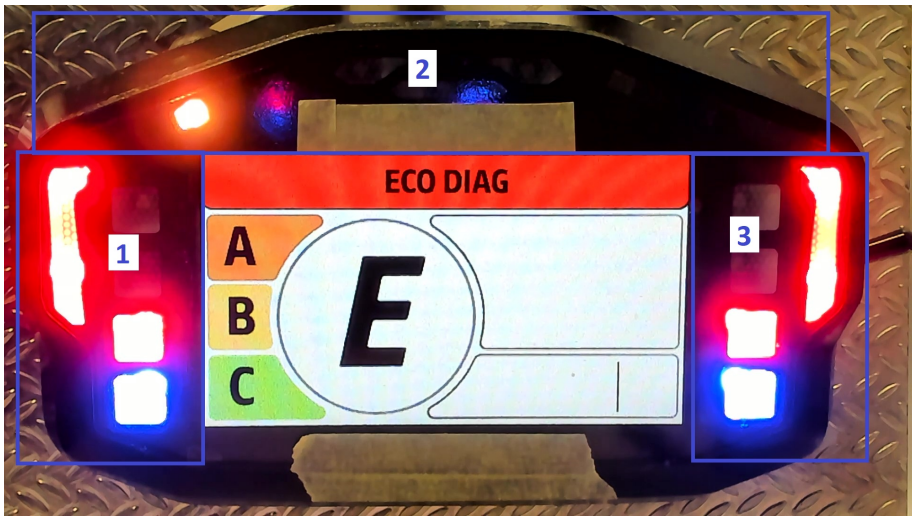


Figura 3.18: Esempio di regioni del frame contenenti i led

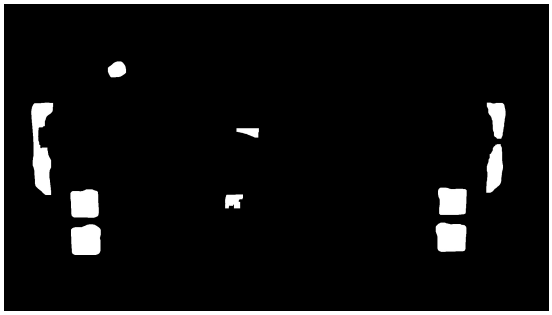


Figura 3.19: Maschera dei led

Figura 3.20: Contorni rilevati dei led

Una volta identificate le zone interessate sarà necessario collegarle al led corrispondente, identificato da un numero visibile nella figura 3.21. Per farlo verranno utilizzati gli ultimi parametri richiesti all'utente in fase di impostazione del sistema, ovvero il punto centrale di ogni led, visibili come punti rossi nella figura 3.22. Calcolando poi la distanza tra il centroide della zona bianca sarà sufficiente determinare l'indice del led corrispondente in base alla distanza tra i due punti.

Figura 3.21: Centroidi ordinati dei led

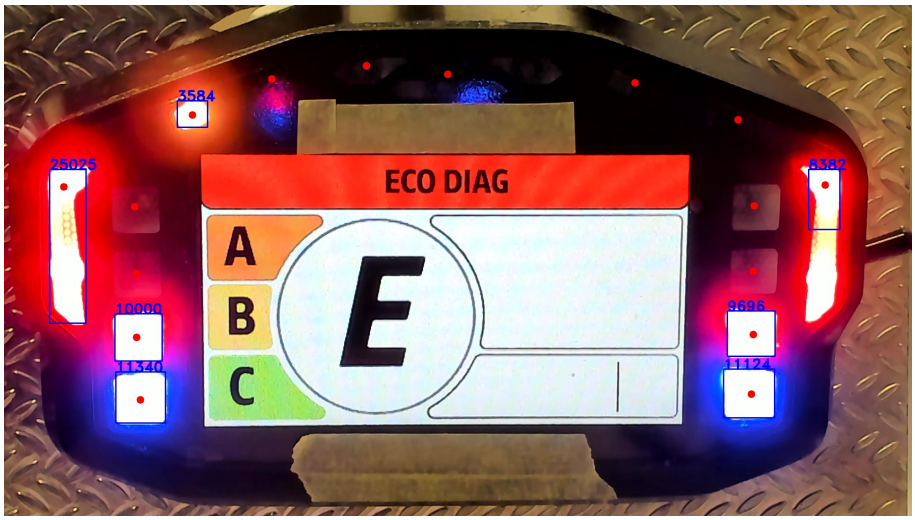


Figura 3.22: Risultato analisi dei led

Per soddisfare poi il requisito per cui il tool dovesse essere in grado di riconoscere il comportamento dei led nel tempo, identificandone quindi uno tra i quattro possibili

- **spento** (-)
- **acceso** (On)
- **lampeggiante** (f)
- **lampeggiante veloce** (Ff)

è stato necessario aggiungere una logica che mantenesse in memoria lo stato *acceso/-spento* istantaneo registrato dal tool accumulando così uno storico che permettesse di identificarne anche il comportamento. Nella figura 3.23 è possibile vedere una rappresentazione grafica dello stato dei led secondo i dati raccolti dal tool durante un test. Nella figura si può osservare come nella parte iniziale a sinistra del

grafico ci siano sei led accesi dei quali quattro cominceranno una fase di lampeggiamento veloce poco la metà del tempo totale, mentre il secondo led partendo a contare dalla parte alta dell'immagine è caratterizzato da due fasi di lampeggiamento normale. Basandosi sull'analisi di questi e di altri dati raccolti sono state definite poi logiche per l'identificazione dei comportamenti basate sugli intervalli di tempo corrispondenti ai *duty cycle* noti a priori.

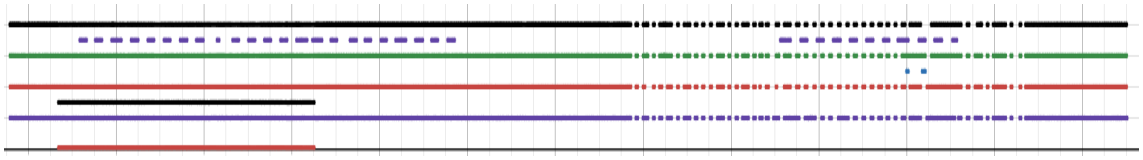


Figura 3.23: Rappresentazione dello stato dei led nel tempo

3.5.3 Logger

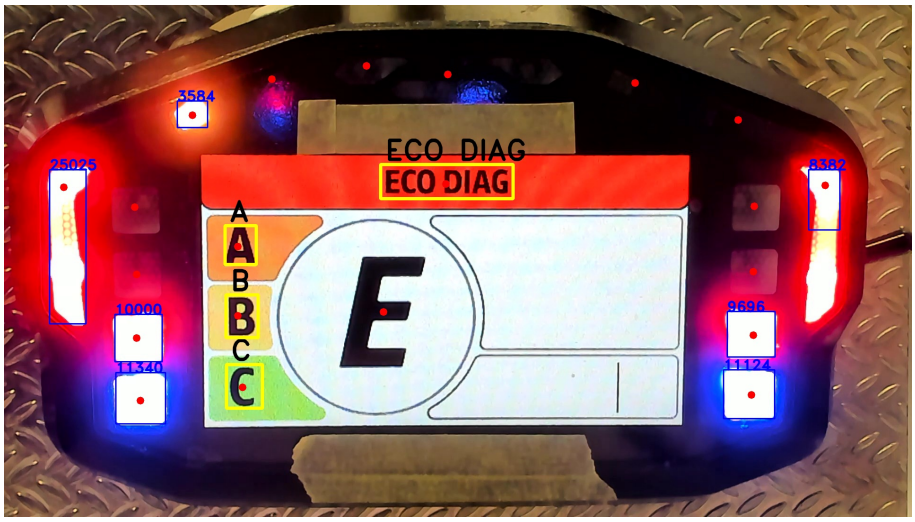


Figura 3.24: Output finale dei processi di riconoscimento

Dopo aver raccolto tutte le informazioni dai vari thread è stato poi sufficiente aggiungere un processo che si occupasse di stamparle all'interno di un file `.csv`. Per non perdere informazioni in arrivo era necessario mantenere una frequenza di esecuzione sufficientemente elevata, tuttavia ciò causava un aumento delle operazioni di scrittura immotivato nel caso in cui non ci fossero informazioni nuove. Per ovviare a questa problematica si è proceduto a ottimizzare il processo di logging, implementando una verifica che confrontasse l'ultima informazione ricevuta con l'ultima riportata sul file e procedesse di conseguenza alla scrittura solo nel caso in cui fosse rilevata una differenza tra le due. Nella figura 3.25 è riportato un esempio di file prodotto dal tool, nel quale è possibile osservare nell'ordine le seguenti informazioni:

- Valore del tempo al momento della scrittura
- Testo letto dall'OCR in corrispondenza dell'allarme

- Valori di mappe e marcia impostata sempre derivanti dall'OCR
- Valore reale dell'allarme
- Stringa corrispondente ai comportamenti dei led

TIMESTAMP	ALARM	MAP1	MAP2	MAP3	GEAR	REAL ALARM	LED
4.938							On,On,-,-,On,On,-,-,On,F,-,-,-,-,-,On
5.249	ECO DIAG	A	B	C	E	['ECO DIAG']	On,On,-,-,On,On,-,-,On,F,-,-,-,-,-,On
9.677	ECO DIAG	A	B	C	E	['ECO DIAG']	-,On,-,-,-,On,-,-,On,F,-,-,-,-,-,On
13.292	ECO DIAG	B	A	C	E	['ECO DIAG']	-,On,-,-,-,On,-,-,On,F,-,-,-,-,-,On
13.396	ECO DIAG	B	A	C	E	['ECO DIAG']	-,On,-,-,-,On,-,-,On,-,-,-,-,-,-,On
16.417	POWER CYCLE BA	B	A	C	E	['POWER CYCLE - BA']	-,On,-,-,-,On,-,-,On,-,-,-,-,-,-,On
16.718	POWER CYCLE BA	B	A	C	E	['POWER CYCLE - BA']	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff
19.936	POWER CYCLE BA	A	B	C	E	['POWER CYCLE - BA']	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff
20.546	POWER CYCLE BA	A	B	C	E	['POWER CYCLE - BA']	-,Ff,-,-,-,Ff,-,-,Ff,F,-,-,-,-,-,Ff
24.163							-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff
24.363	POWER CYCLE BA	B	A	C	E	['POWER CYCLE - BA']	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff
25.372	ECO DIAG	B	A	C	E	['ECO DIAG']	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff
25.774	ECO DIAG	B	A	C	E	['ECO DIAG']	-,On,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff
25.977	ECO DIAG	B	A	C	E	['ECO DIAG']	-,On,-,-,-,On,-,-,On,-,-,-,-,-,-,On

Figura 3.25: Esempio di log derivante dall'osservazione

	A	B	C	D	E	F	G	H	I
1	Type	Name	Enable	Message	Leds	Timeout	Debounce	Background	Foreground
2	alarm114	INSULATION_FAIL	TRUE	INSULATION FAIL	Ff,Ff,-,Ff,Ff,Ff,Ff,Ff,-,-,-,-,-,-,-,-	0	0	#FFFF0000	#FF000000
3	alarm113	BATTERY_OVERHEAT	TRUE	BATTERY OVERHEAT	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff	0	0	#FFFF0000	#FF000000
4	alarm111	PWT_STOP	TRUE	BIKE STOP	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff	0	0	#FFFF0000	#FF000000
5	alarm110	PWT_STOP_MSHL	TRUE	BIKE STOP	On,-,-,-,On,-,-,-,-,-,-,-,-,-,-,-	0	0	#FF0000FF	#FFFFFFFF
6	alarm109	PWT_ENGINE_STOP	TRUE	MOTOR STOP	-,Ff,-,-,-,Ff,-,-,Ff,-,-,-,-,-,-,Ff	0	0	#FFFF0000	#FF000000
7	alarm108	PWT_ENGINE_STOP_MSHL	TRUE	MOTOR STOP	On,-,-,-,On,-,-,-,-,-,-,-,-,-,-,-	0	0	#FF0000FF	#FFFFFFFF
8	alarm105	RBW_FAILURE	TRUE	RBW FAILURE	-,F,-,-,-,F,-,-,F,-,-,-,-,-,-,F	0	0	#FFFF0000	#FF000000
9	alarm104	RBW_FAILURE_MSHL	TRUE	RBW FAILURE	-,-,-,-,-,-,On,-,-,-,-,-,-,-,-,-	0	0	#FFFFFF00	#FF000000
10	alarm101	AS_FAILURE	TRUE	TC KO-PUSH YELLOW	-,-,-,-,-,-,-,-,Ff,-,-,-,-,-,-,Ff	0	0	#FFFFFF00	#FF000000
11	alarm100	AS_FAILURE_PRAC	TRUE	TC KO	-,-,-,-,-,-,-,-,Ff,-,-,-,-,-,-,Ff	0	0	#FFFFFF00	#FF000000
12	alarm93	AS_OFF	TRUE	TC OFF	-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-	0	0	#FFFFFF00	#FF000000
13	alarm92	AS_OK	TRUE	TC ACTIVE	-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-	30	0	#00ff00	#000000
14	alarm86	LC_ON	TRUE	LAUNCH MODE	-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-	0	0	#FF0000FF	#FFFFFFFF
15	alarm48	PIT_ON	TRUE	PIT LIMIT	-,-,-,-,-,-,-,-,-,-,Ff,Ff,-,-,-	0	0	#FFC0C0C0	#FF000000
16	alarm34	FORK_DIAG	TRUE	FORK DIAG	-,-,-,-,-,-,On,-,-,-,-,-,-,-,-,-	0	0	#FFFFFF00	#000000
17	alarm29	CHARGING	TRUE	CHARGING	-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-	0	0	#FF0000FF	#FFFFFFFF
18	alarm28	DISCHARGING	TRUE	DISCHARGING	-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-	0	0	#FF0000FF	#FFFFFFFF
19	alarm14	TURN_OFF	TRUE	SWITCHED OFF	On,-,-,-,On,-,-,-,-,-,-,-,-,-,-,-	0	0	#FF0000FF	#FFFFFFFF
20									

Figura 3.26: Tabella degli allarmi

Un'ultima verifica che viene effettuata prima della scrittura di una riga di log è la ricerca della corrispondenza dell'allarme. Nonostante il processo di estrazione delle stringhe dal video si sia rivelato molto preciso, non si può considerare esente da piccoli errori sul singolo carattere, per questo motivo utilizzando la libreria `difflib` le stringhe di allarme ricevute dall'OCR vengono confrontate con le corrispondenti stringhe attese. Nella figura 3.26 è riportato un estratto di una *tabella degli allarmi*, un file che permette di tenere traccia di tutti i possibili allarmi impostati sul sistema. Il processo utilizzando la funzione `get_close_matches` calcolerà il *match* migliore tra l'allarme ricevuto e l'insieme delle stringhe attese ottenute tramite l'estrazione dei valori contenuti nella colonna `Message`, corrispondenza che poi verrà scritta nel log in corrispondenza della colonna `Real alarm`.

Inside every large language is a small language struggling to get out.

T. Hoare

Capitolo 4

Integrazione e distribuzione

4.1 Integrazione

L'interazione principale tra i tool appena descritti si limita alla condivisione di risorse comuni. Data la necessità da parte degli utenti di utilizzare un sistema che si basi sugli stessi parametri per tutti, è stata impostata una struttura fissa di cartelle posizionate all'interno della rete aziendale in maniera che in seguito a una modifica effettuata su una risorsa non ci siano problemi di aggiornamento. Questo permette di avere un'unica cartella di progetto all'interno della quale si trovano tutte le cartelle contenenti gli input necessari e le quelle all'interno delle quali posizionare l'output dei singoli tool. Così facendo è stato possibile sviluppare i tool in maniera che utilizzassero solo percorsi relativi, avendo ad esempio una semplice corrispondenza tra la cartella `Generated code` all'interno della quale il *generator* inserirà i file generati e la stessa utilizzata dall'*executor* per selezionare le sequenze da eseguire.

4.1.1 Reazione agli allarmi

Una forma di integrazione più stretta è invece quella che è stata creata tra l'*executor* e il *watcher*. Come anticipato uno dei requisiti era che il sistema fosse reattivo agli allarmi e per ottenere questo risultato era necessario che le due componenti sopracitate comunicassero tra di loro.

Per raggiungere questo obiettivo è stata utilizzata la libreria `pyzmq` che contiene i necessari collegamenti per utilizzare la libreria ZeroMQ[auta] all'interno di codice Python. Questa libreria open source si propone come una soluzione universale per lo scambio di messaggi, offrendo funzioni per la comunicazione asincrona pensate per applicazioni distribuite o concorrenti. L'implementazione permette la creazione di una socket all'avvio dei tool, creando così un canale di comunicazione. Ogni qualvolta il tool incaricato dovesse percepire la presenza di un allarme sul cruscotto invierà un messaggio contenente il vero testo dell'allarme ottenuto come visto in precedenza. Il tool che sta eseguendo le sequenze di test riceverà questa informazione e si preoccuperà di accedere alla tabella degli allarmi dalla quale estrarrà il codice numerico identificativo dell'allarme stesso e lo utilizzerà come parametro in ingresso

per eseguire parallelamente la sequenza riportata in 13. Questa sequenza imposterà il canale scelto al valore in input, che verrà poi letto da eventuali sequenze di test scritte per reagire a questo stimolo con ad esempio la pressione simulata di un pulsante da parte del pilota.

```
self.context = zmq.Context()
self.socket = self.context.socket(zmq.REQ)
self.socket.connect("tcp://localhost:5555")
self.socket.send_string(REALALARM)
```

Listing 11: Codice per l'invio dell'allarme

```
context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")
message = socket.recv(flags=zmq.DONTWAIT)
```

Listing 12: Codice per la ricezione dell'allarme

```
from niveristand import nivs_rt_sequence, run_py_as_rtseq,
↪  realtimesequencetools
from niveristand.clientapi import ChannelReference

@nivs_rt_sequence
def run_sequence(val):

    Alarm_received = ChannelReference ('Targets/Controller/User
    ↪  Channels/Alarm_received')
    # Insert alarm
    Alarm_received.value = val


if __name__ == '__main__':
    val = DoubleValue(get_alarm_number(sys.argv[1]))
    run_py_as_rtseq(run_sequence,{"val":val})
    print("Sent alarm " + sys.argv[1])

```

Listing 13: Sequenza real-time per l'invio di un allarme

4.2 Distribuzione

Per poter passare alla fase di distribuzione del tool era poi necessario trovare una metodologia migliore rispetto all'esecuzione di un file Python per permettere agli utenti di utilizzare il sistema. Per raggiungere questo obiettivo sono state seguite due strade diverse, a seconda della tipologia di tool, descritte nelle seguenti sezioni.

4.2.1 Generazione eseguibili

I tre componenti sviluppati in Python sono stati distribuiti sfruttando PyInstaller[Cor], famoso tool utilizzato dagli sviluppatori per creare degli eseguibili a partire da degli script Python. La caratteristica principale di questo strumento è la sua capacità di analizzare gli script per trovare tutti i moduli e le librerie necessarie per la sua corretta esecuzione, per poi procedere a creare una copia di ognuno di essi, aggiungerci una copia anche dell'interprete Python attivo, e poi confezionare tutto in una sola cartella o addirittura un solo file.

Nel listato 14 è riportato uno script scritto per automatizzare e semplificare l'attività di distribuzione del software. Per ognuno dei tre tool la procedura sarà analoga, dopo essersi spostati all'interno della specifica cartella contenente i sorgenti verrà lanciato il comando necessario per generare l'eseguibile, che verrà successivamente copiato dalla cartella `dist` sia nella cartella di sviluppo che in quella di distribuzione.

Si può poi notare dai comandi relativi al confezionamento dei file `.exe` che ogni tool ha richiesto uno specifico trattamento. All'inizio di ogni istruzione si trova ovviamente `pyinstaller`, seguito dai due flag `-w` e `-F` che rispettivamente indicano al programma di creare un eseguibile che non faccia uso di una console per le operazioni di I/O e di generare solamente un file `.exe` contenente tutte le informazioni necessarie al tool piuttosto che creare una cartella dedicata.

Le particolarità maggiori si evidenziano per quanto riguarda i primi due eseguibili. Nel primo caso viene utilizzata `-add-data=res:res`, un'opzione che permette di aggiungere ulteriori file di dati, o cartelle come in questo caso, all'applicazione. Gli argomenti dovranno avere il formato "source:dest_dir", dove *source* indica il percorso al file, o cartella, da copiare, mentre *dest_dir* indica il percorso alla cartella di destinazione relativamente all'eseguibile. Nel secondo caso viene invece utilizzata l'opzione `-hidden-import`, utilizzata per importare all'interno dell'eseguibile una libreria non visibile all'interno del codice, come nel caso di `clr` per il tool di esecuzione dei test.

```
1  cd 2 - Code generation
2  pyinstaller --add-data=res:res -w -F CodeGeneration.py
3  copy dist\CodeGeneration.exe ..\
4  cd ..
5  copy CodeGeneration.exe "..\HIL Automator"
6
7  cd 3 - Execution
8  pyinstaller -w -F --hidden-import=clr TestAutomation.py
9  copy dist\TestAutomation.exe ..\
10 cd ..
11 copy TestAutomation.exe "..\HIL Automator"
12
13 cd 4 - GRD recognition\src
14 pyinstaller -w -F GrdRecognition.py
15 copy dist\GrdRecognition.exe ..\..\
16 cd ..\..
17 copy GrdRecognition.exe "..\HIL Automator"
```

Listing 14: Codice per automatizzare la generazione degli eseguibili

4.2.2 Generazione installer

Per quanto riguarda infine il tool per la validazione dei fogli Excel è stato necessario trovare un metodo alternativo per facilitare l'installazione. Su consiglio del tutor aziendale per questo progetto sono stati analizzati alcuni esempi sviluppati direttamente da lui per strumenti analoghi.

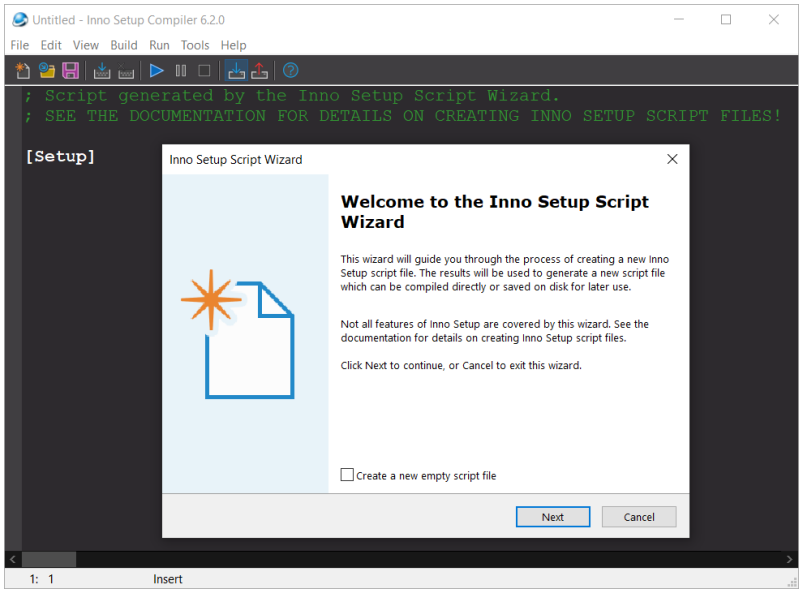


Figura 4.1: Wizard di Inno Setup

Il programma utilizzato è Inno Setup, un *software libero* scritto in linguaggio Delphi che permette a partire da uno script di generare un programma di installazione. Nonostante sia stato creato nel 1997 ancora oggi rivaleggia e spesso sorpassa varie alternative commerciali. Come è possibile vedere nella figura 4.1 Inno Setup utilizza un semplice e intuitivo *wizard* che permette attraverso dei campi da compilare di generare una prima versione dello script in formato `.iss` necessario per la generazione dell'installer.

Nell'estratto dello script utilizzato per la creazione dell'installer riportato in figura 4.2 è possibile vedere alcuni dei punti chiave per la personalizzazione dell'installazione. In corrispondenza della variabile `DefaultDirName` viene infatti inserito il percorso utilizzato internamente all'azienda per l'installazione di nuovi tool, mentre nelle due variabili subito seguenti vengono impostate la cartella all'interno della quale Inno Setup dovrà salvare il file e il nome del file stesso, per cui relativamente alla posizione dello script nel filesystem l'installer si troverà in `InnoOutput\TestValidator_installer.exe`. Nelle ultime due righe infine sono riportate le sorgenti dalle quali importare i file da includere nell'instraller, aggiungendo poi che il primo avrà destinazione corrispondente a quella dell'app, ovvero come indicato da `DefaultDirName`, mentre il file `.xlam` dovrà essere posizionato nella cartella `Microsoft\AddIns` all'interno del percorso `userappdata`, una costante che

indica il percorso `C:\Users\[your username]\AppData\Roaming`, scelta obbligata da Microsoft per poter eseguire l'Add-in.

Nella figura 4.3 si può infine vedere avviato il wizard di installazione generato grazie a Inno Setup nella sua prima schermata all'interno della quale richiede all'utente di selezionare la cartella d'installazione che può essere modificata rispetto alla precedente `DefaultDirName`.

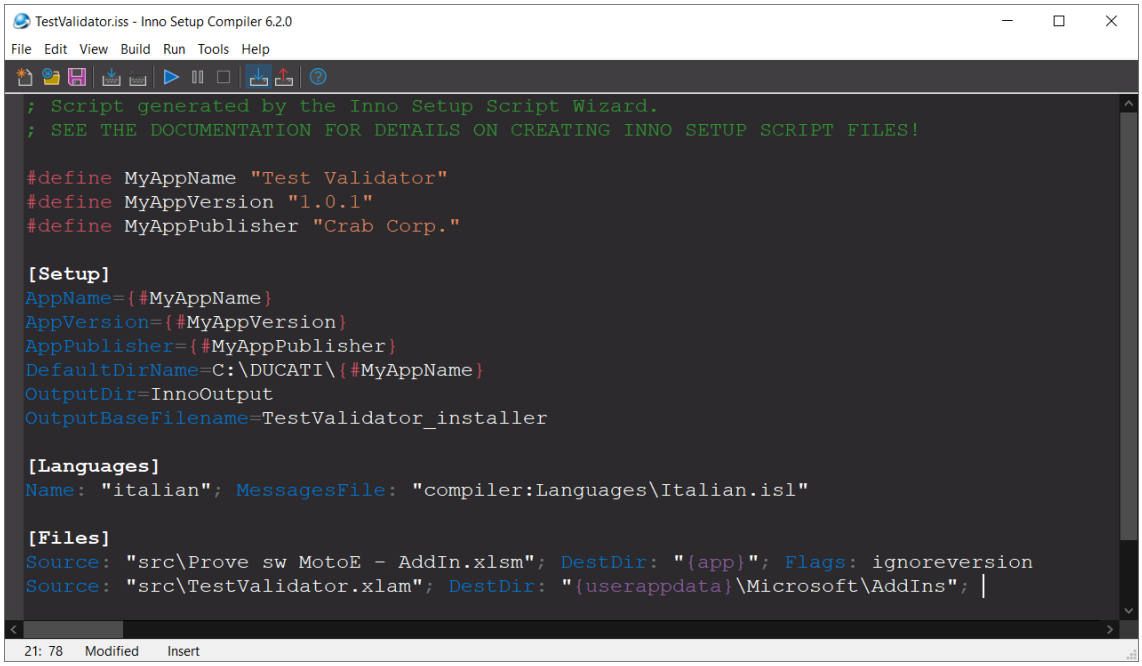


Figura 4.2: Script per la generazione dell'installer

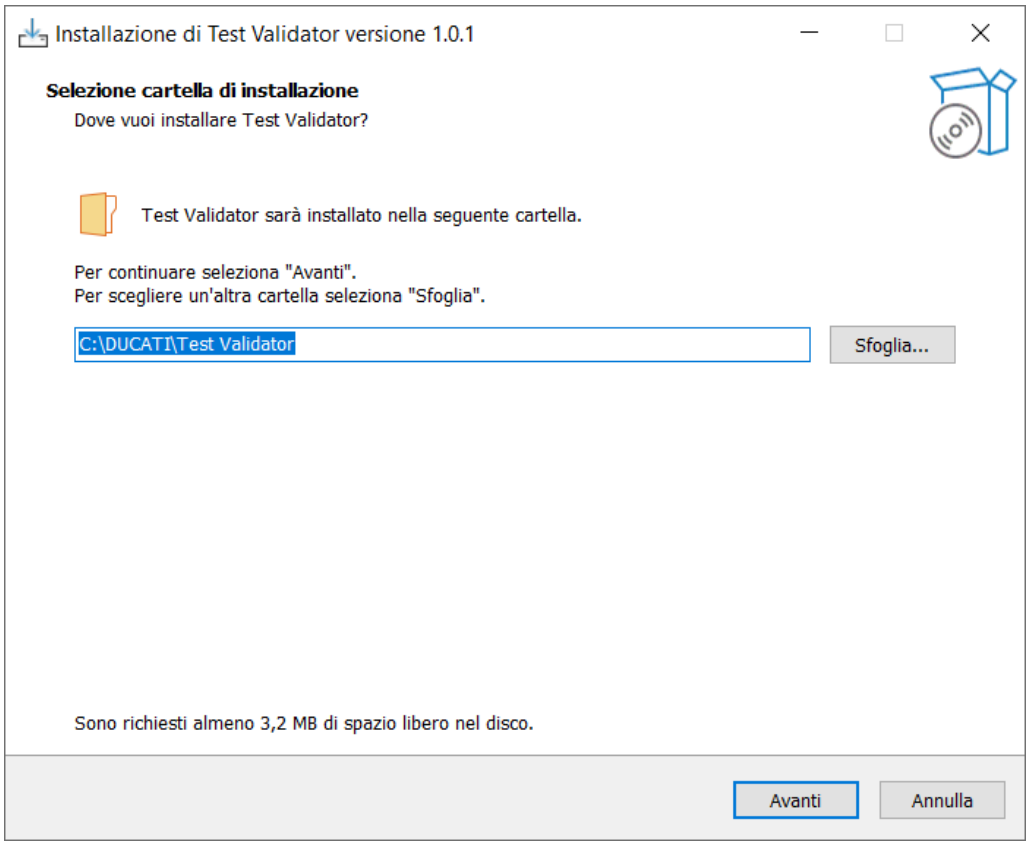


Figura 4.3: Wizard di installazione

The check engine light is just a suggestion.

Every race team ever

Capitolo 5

Simulazione di utilizzo

In questo capitolo verranno riportati alcuni screenshot relativi a una simulazione di utilizzo dell'intero sistema sviluppato.

Il primo strumento utilizzato è quello per la validazione dei fogli di test, per cui si è simulato il ruolo dello stratega. Come prima cosa all'utente verrà fornito l'installer di cui si è parlato nel capitolo 4, all'avvio del quale verrà mostrata l'interfaccia riportata nella figura 5.1.

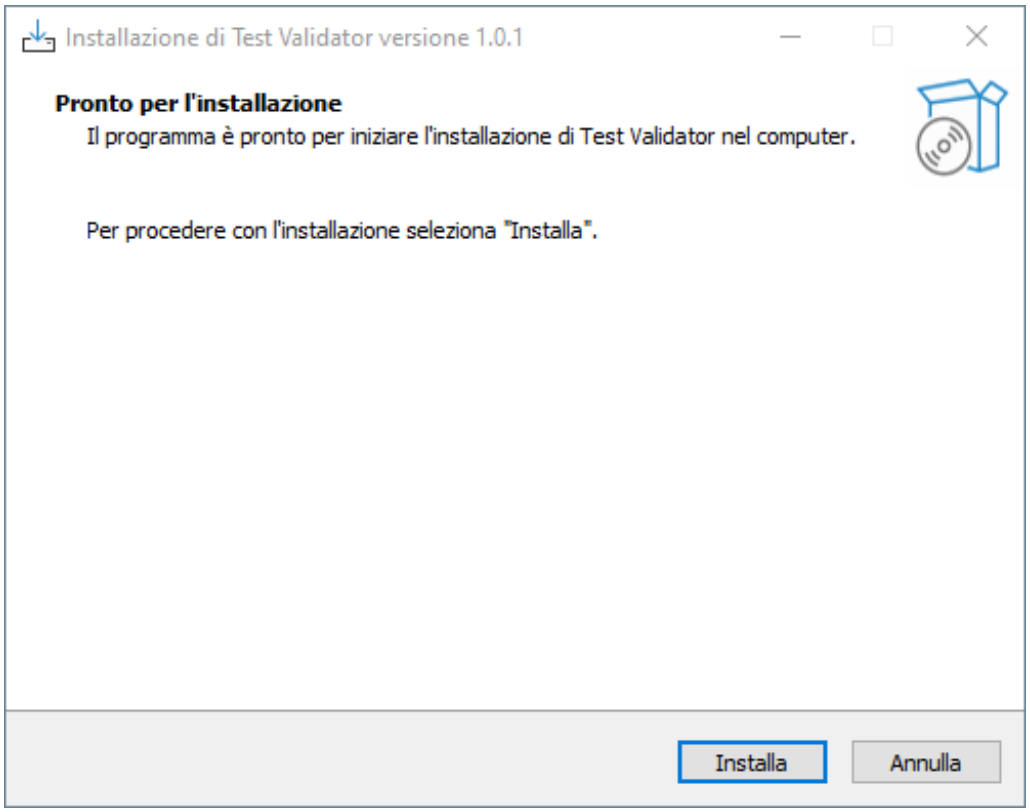


Figura 5.1: Interfaccia installer Test Validator

Una volta seguita la procedura guidata l'utente potrà verificare la corretta installazione dell'Add-in accedendo al percorso di destinazione come riportato nella figura 5.2, e controllare che il file sia presente.

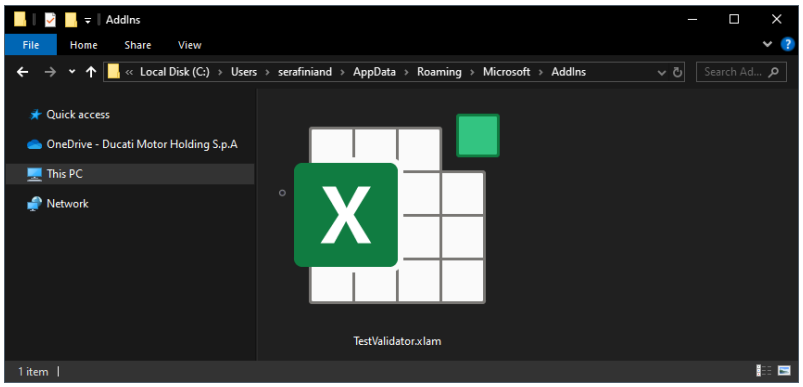


Figura 5.2: File Add-in al path di destinazione

Una volta localizzato invece il file template da utilizzare lo stratega potrà procedere direttamente al suo lavoro, ovvero la definizione dei test. Nella simulazione effettuata per verificare l'intera usabilità sono stati scritti anche test contenenti errori, come quello mostrato nella figura 5.3. In questa figura è possibile riconoscere il chiaro segnale di errore dato dallo sfondo rosso della cella contenente l'errore, oltre alla *dialog* centrale che evidenzia anche l'effettiva parte mancante nel test, ovvero l'indicazione `con valore <value>`.

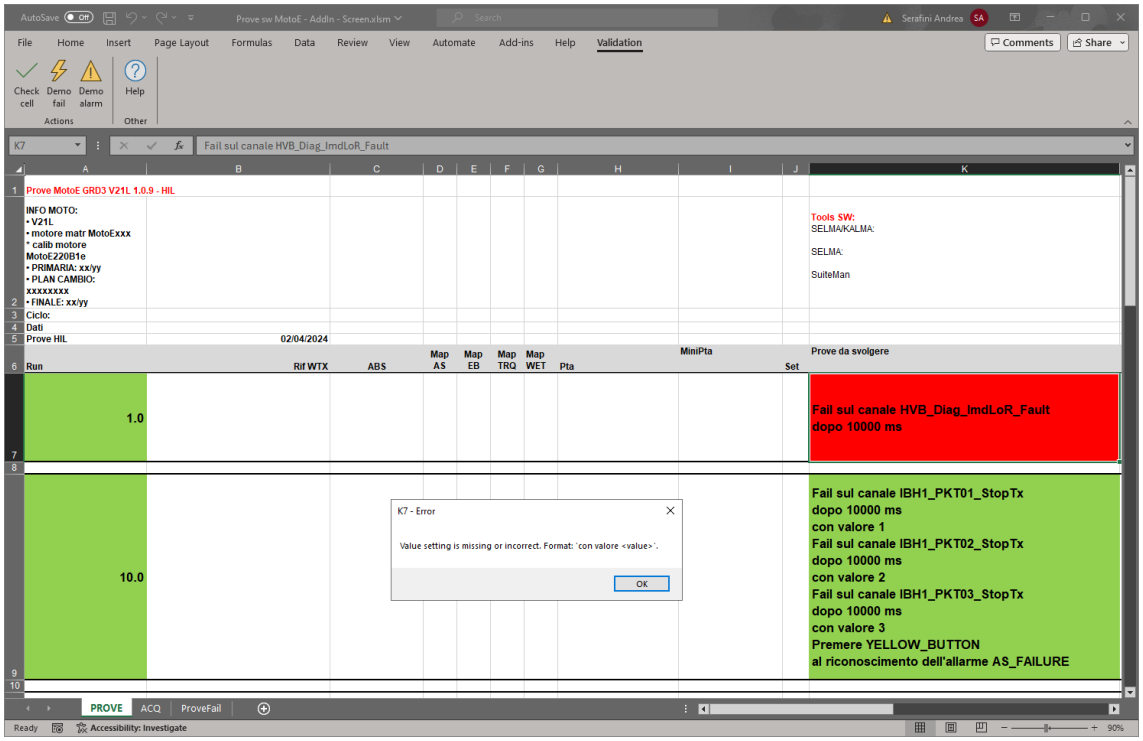


Figura 5.3: Foglio di test con errore

Nella figura 5.4 invece il test relativo alla run 1.0 risulta corretto, presentando un `Fail` con valore 1 sul canale desiderato. La run 10.0 è stata invece strutturata più complessa, inserendo più test, a partire da diversi `Fail` su canali diversi, fino alla reazione simulata del pilota a un segnale di allarme. Questo file rappresenta la versione finale ottenuta dallo stratega e passata allo step successivo come input.

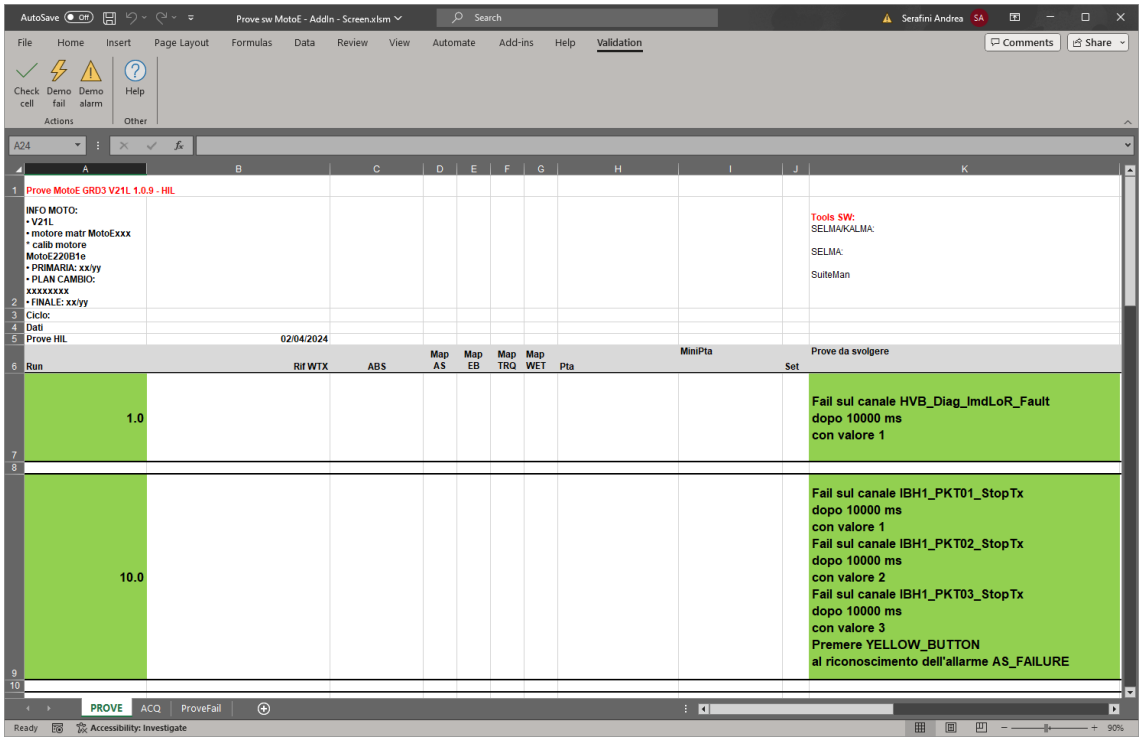


Figura 5.4: Foglio di test corretto

Nella figura 5.5 analizzando l'interfaccia proposta all'utente per generare il codice è possibile vedere le impostazioni utilizzate per questa simulazione. Dopo aver selezionato lo scheletro base per la sequenza da utilizzare, in questo caso il semplice `Loop_RunE.py`, il database contenente i riferimenti ai canali utilizzati, riportato nella figura 5.6 e creato appositamente semplificato per l'esecuzione di questa dimostrazione ma con valori utilizzati nelle situazioni reali, e il file dei test appena validato, l'utente potrà procedere alla pressione del pulsante `Generate`. Nella parte inferiore dell'interfaccia si può poi vedere l'output che indica l'avvenuta generazione del codice.

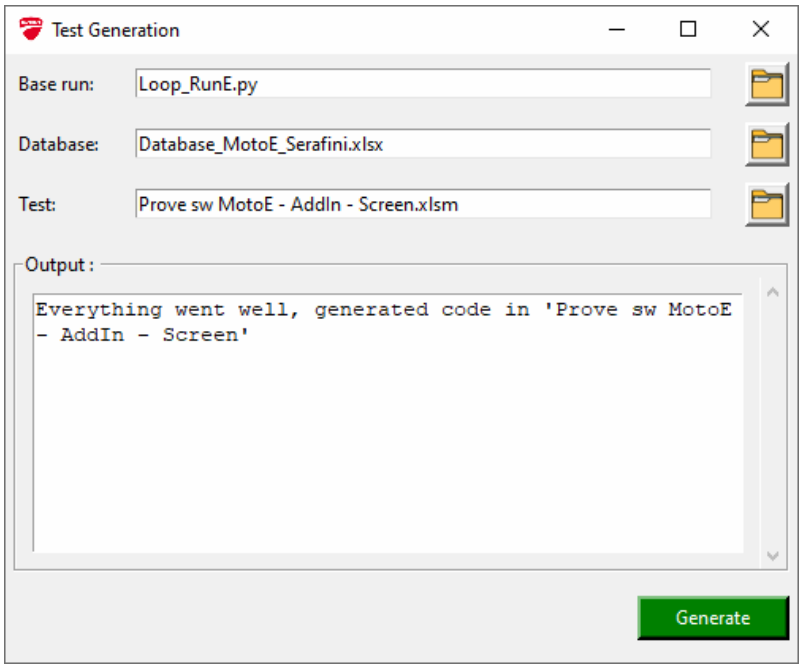
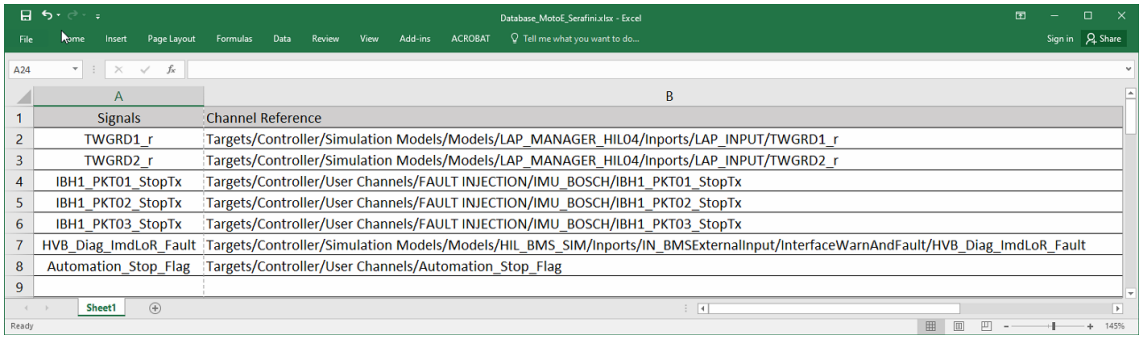


Figura 5.5: Interfaccia generatore di codice



	A	B
1	Signals	Channel Reference
2	TWGRD1_r	Targets/Controller/Simulation Models/Models/LAP_MANAGER_HIL04/Inports/LAP_INPUT/TWGRD1_r
3	TWGRD2_r	Targets/Controller/Simulation Models/Models/LAP_MANAGER_HIL04/Inports/LAP_INPUT/TWGRD2_r
4	IBH1_PKT01_StopTx	Targets/Controller/User Channels/FAULT INJECTION/IMU_BOSCH/IBH1_PKT01_StopTx
5	IBH1_PKT02_StopTx	Targets/Controller/User Channels/FAULT INJECTION/IMU_BOSCH/IBH1_PKT02_StopTx
6	IBH1_PKT03_StopTx	Targets/Controller/User Channels/FAULT INJECTION/IMU_BOSCH/IBH1_PKT03_StopTx
7	HVB_Diag_ImdLoR_Fault	Targets/Controller/Simulation Models/Models/HIL_BMS_SIM/Inports/IN_BMSExternalInput/InterfaceWarnAndFault/HVB_Diag_ImdLoR_Fault
8	Automation_Stop_Flag	Targets/Controller/User Channels/Automation_Stop_Flag
9		

Figura 5.6: Database dimostrativo di riferimenti a canali

Un passo che poi può essere eseguito o meno a discrezione dell'utente in base alle specifiche necessità è quello di verificare le sequenze prodotte. Nella figura 5.7 è riportata l'interfaccia appena mostrata accanto ai file generati. Questi ultimi sono stati correttamente nominati come le corrispettive run e posizionati all'interno di una cartella riportante il nome del file di test utilizzato. Nella figura 5.8 è poi riportato un estratto del codice appena generato, in particolare la seconda run definita dallo stratega, nel quale è possibile vedere i cinque task nei quali è stato scomposto il test in fase di generazione. La leggibilità del codice prodotto permette eventualmente all'utente di intervenire manualmente modificando alcuni parametri di test, oppure direttamente aggiungendone di nuovi e rimuovendone di superflui creando nuove run derivate.

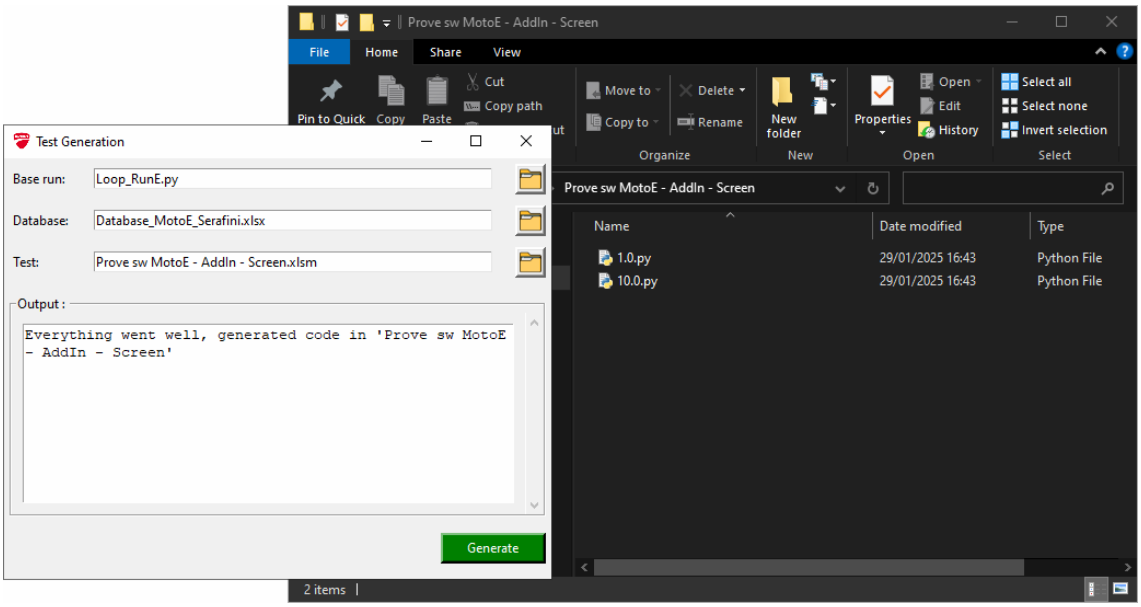


Figura 5.7: Sequenze generate

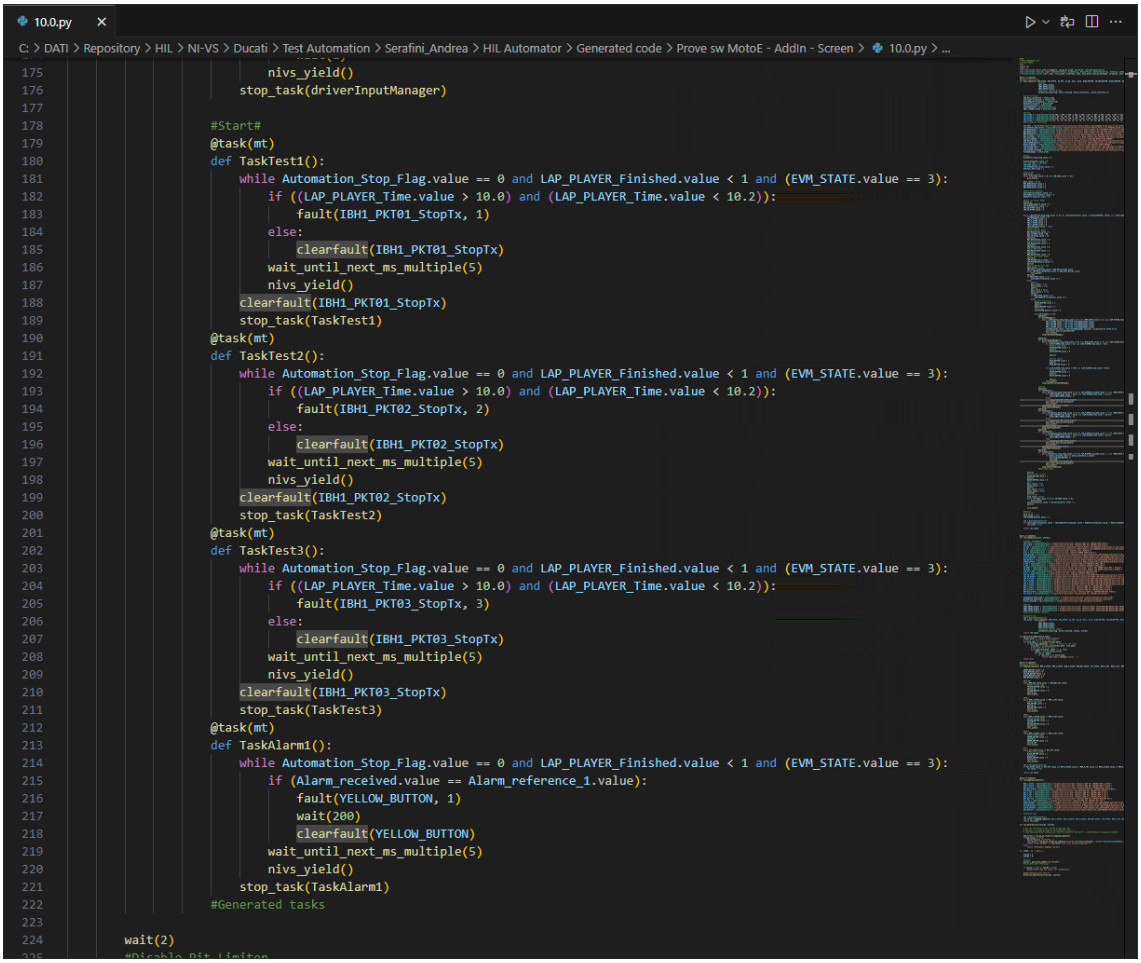


Figura 5.8: Sequenza run 10.0

Terminata la generazione è poi possibile passare allo step successivo, l'esecuzione dei test. Essendo una simulazione completa di utilizzo è stato ipotizzato che fosse il primo utilizzo del sistema, rendendo necessario effettuare la calibrazione del watcher come illustrato nei capitoli precedenti. Nella figura 5.9 è riportato quindi solo un particolare di tutto il processo di estrazione dei parametri utili al riconoscimento del cruscotto che è stato eseguito in questa simulazione.

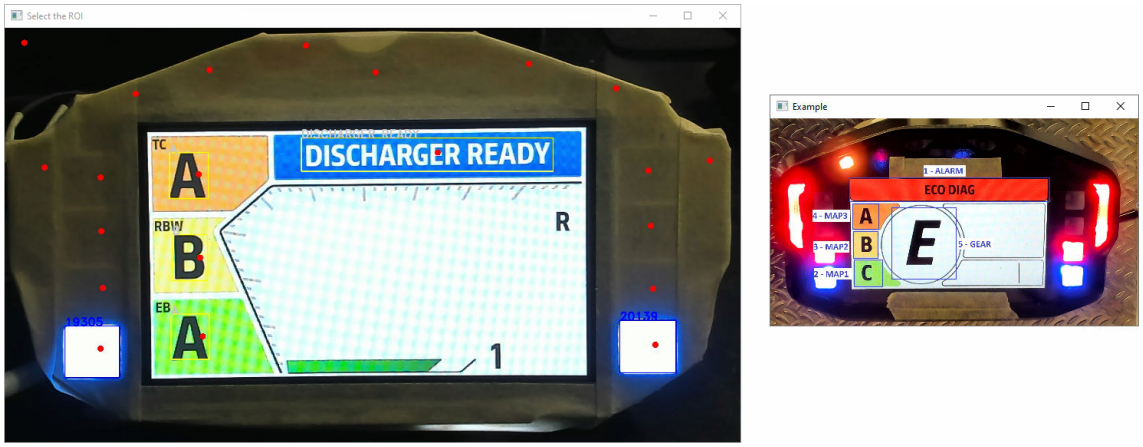


Figura 5.9: Estrazione ROI testuali

Prima di avviare l'esecuzione effettiva dei test l'utente potrà selezionare attraverso l'interfaccia mostrata nella figura 5.10 quale webcam collegata al sistema dovrà essere utilizzata come dispositivo di input per il watcher.

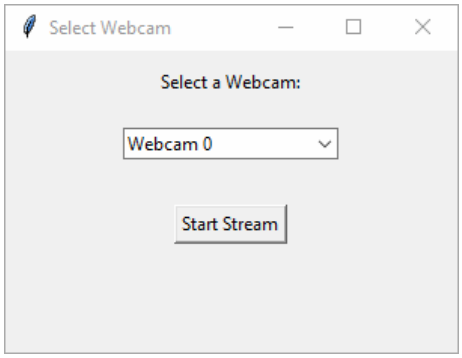


Figura 5.10: Interfaccia selezione webcam

Una volta avviato il componente del sistema finalizzato al riconoscimento degli allarmi, l'utente procederà all'apertura dell'interfaccia dell'esecutore, aggiungendo tutti i test che si vogliono eseguire. Nella figura 5.11 si può vedere appunto la run 10.0 aggiunta e pronta per essere avviata.

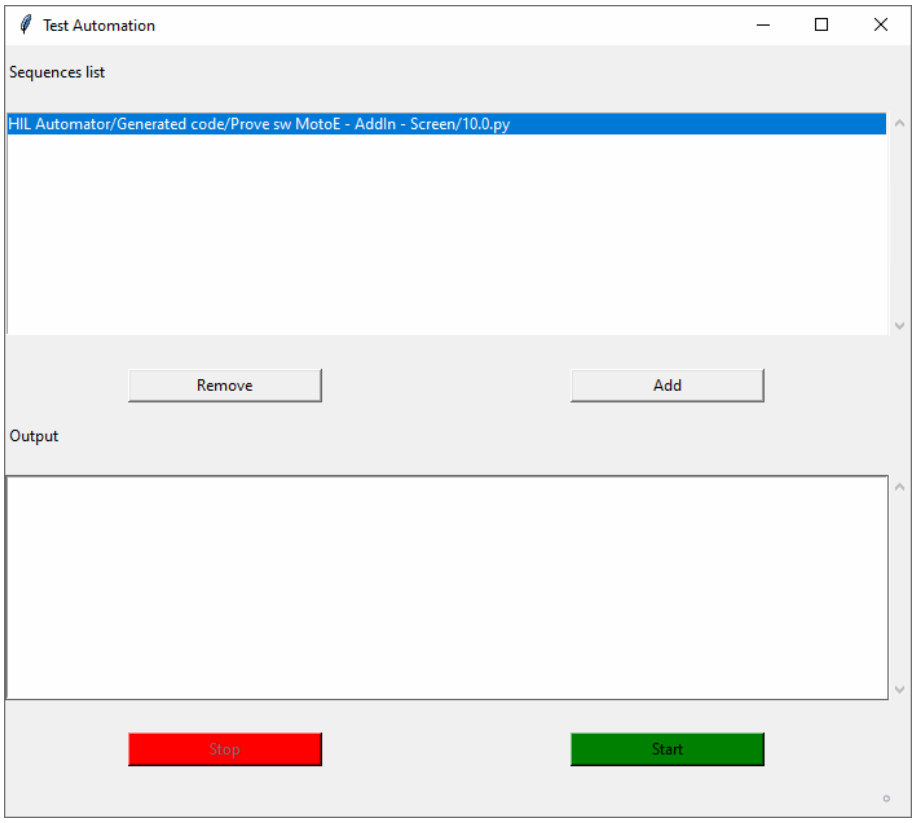


Figura 5.11: Interfaccia reattiva per l'esecuzione

Nel caso di campagne di test più lunghe e strutturate l'utente potrebbe poi voler sovrapporre le finestre appena aperte all'interfaccia personalizzata sviluppata per VeriStand, permettendo come si può vedere nella figura 5.12 di monitorare l'andamento del sistema da più punti di vista.

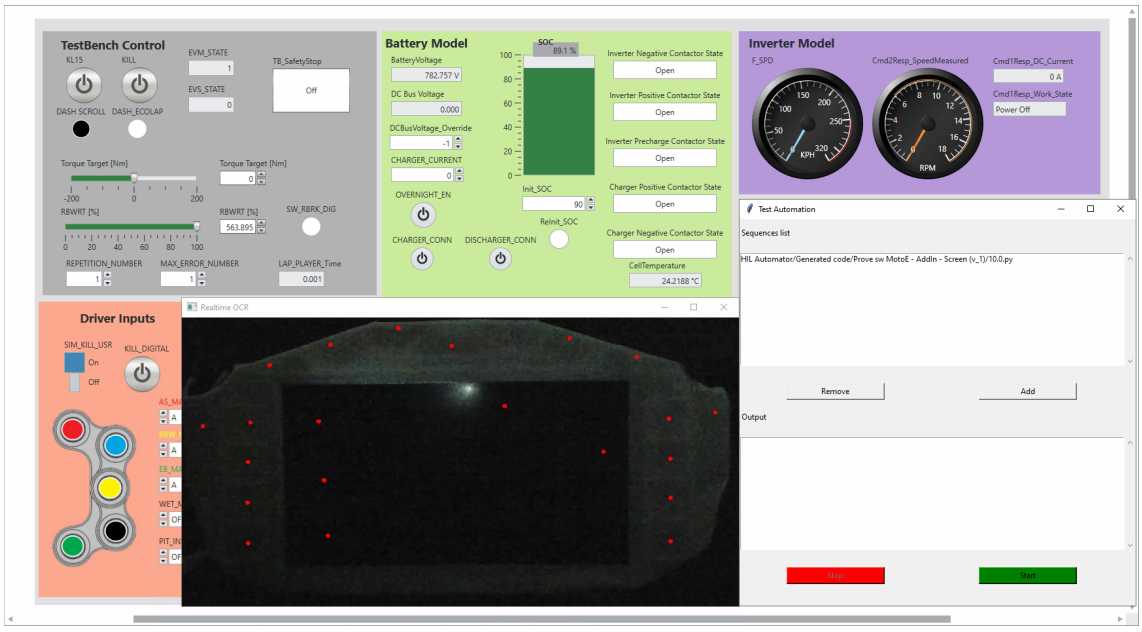


Figura 5.12: Interfaccia utilizzata durante i test

Durante l'esecuzione il watcher mostrerà in tempo reale le informazioni rilevate e significative, come ad esempio i LED che risultano accesi e i testi riconosciuti per gli allarmi e le mappe selezionate, come mostrato nella figura 5.13.

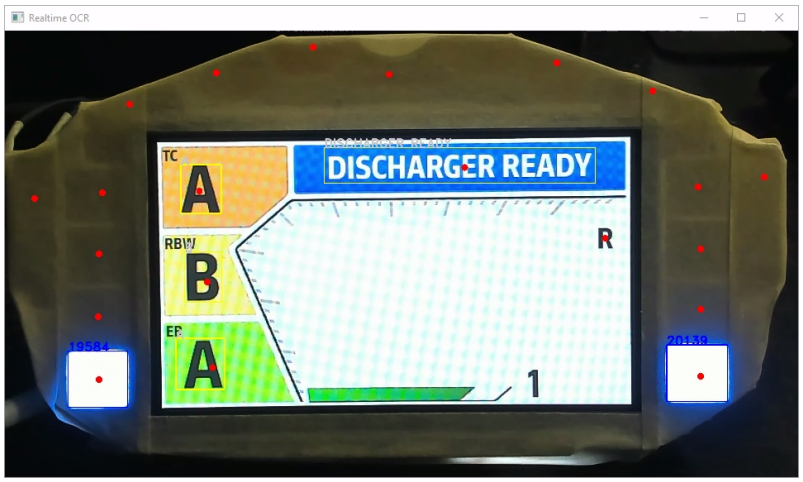


Figura 5.13: Riconoscimento allarmi e LED

Al termine dell'esecuzione l'interfaccia riportata nella figura 5.14 mostrerà all'utente il risultato ottenuto nella sezione di output. In questo caso esemplificativo è stata eseguita una sola run, per cui l'output risulta limitato, mentre nel caso di esecuzioni in blocco verrebbero stampate le informazioni di tutti i test in corrispondenza della conclusione di ognuno di essi. Qui in particolare è riportato l'esempio di un'esecuzione interrotta manualmente, come si può capire dal messaggio `Stop flag setted correctly`, seguita poi dalle indicazioni di esecuzione terminata correttamente e di dove sia stato salvato il log.

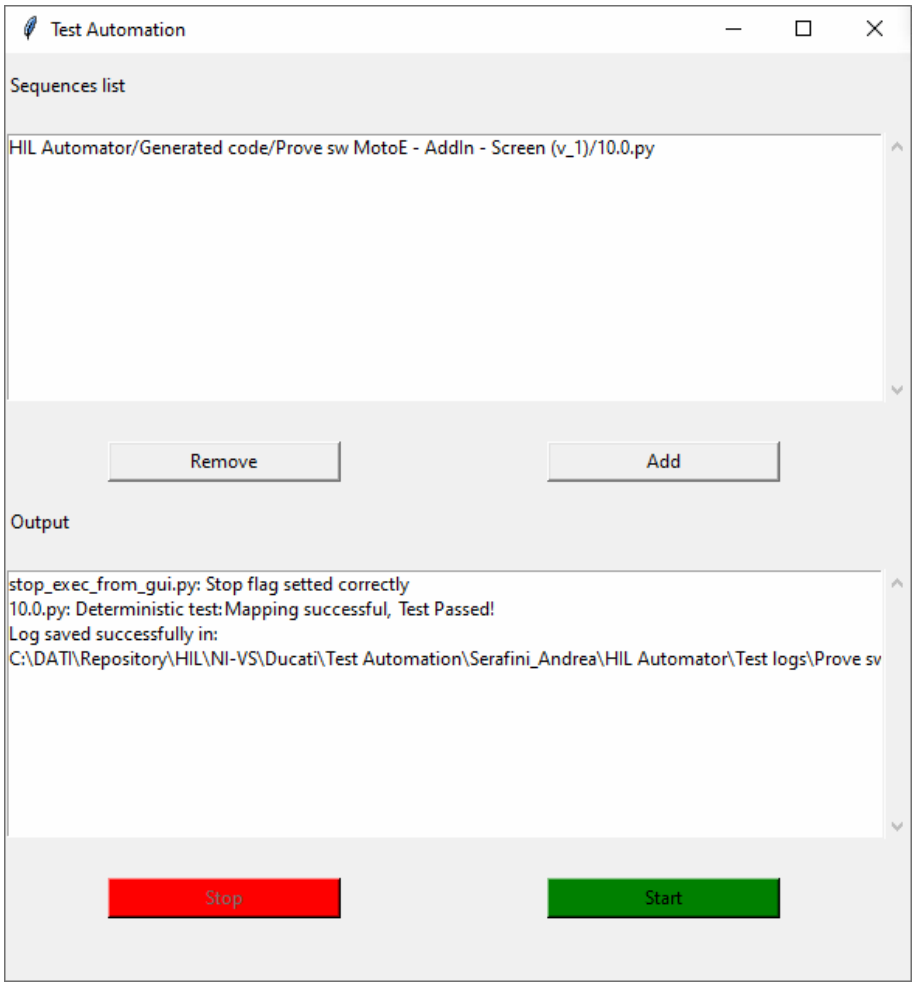


Figura 5.14: Interfaccia al termine dell'esecuzione

Terminata l'esecuzione dei test si potrà poi procedere alla fase di analisi dei risultati. Nella figura 5.15 è riportato il file `csv` prodotto come log contenente i valori dei canali letti direttamente dall'HIL. Si può notare come, nelle linee successivamente evidenziate, siano riportati i valori che indicano il corretto svolgimento di almeno uno dei task del test eseguito, il quale aveva come obiettivo quello di simulare la pressione del pulsante giallo da parte del pilota, corrispondente ai valori `1` presenti nella colonna `Yellow button`, successiva alla lettura del valore corrispondente all'allarme atteso, in questo caso `101`, nel canale riportato nella colonna `Alarm`.

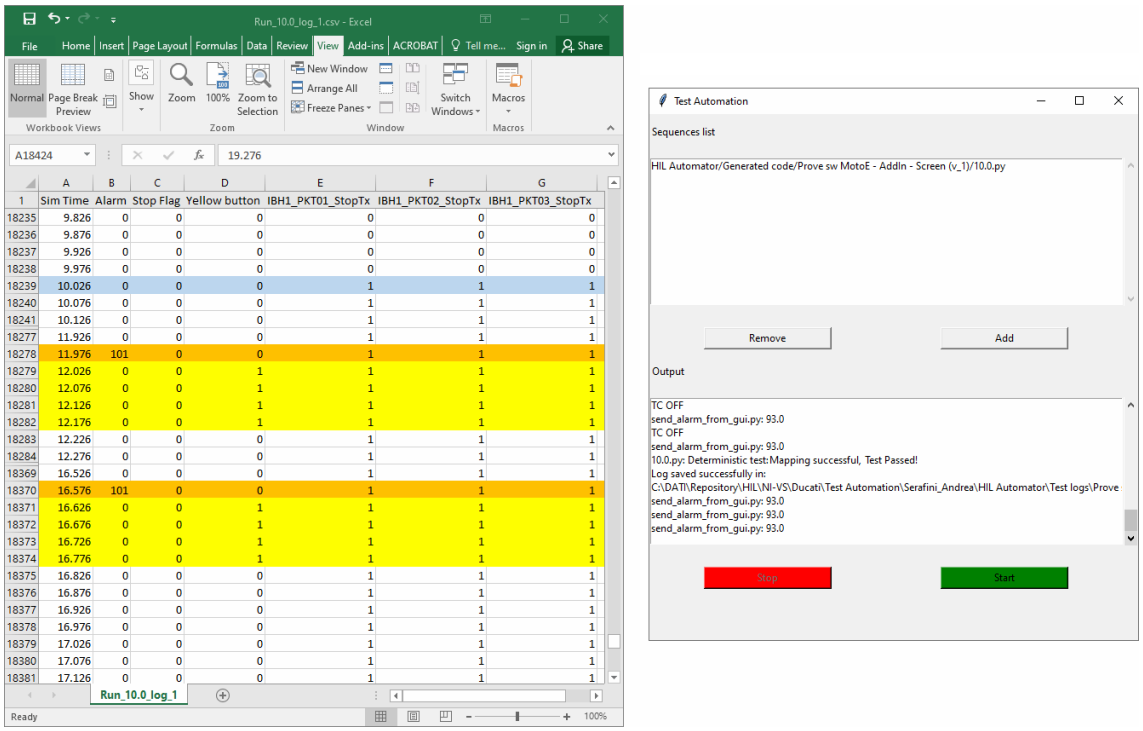


Figura 5.15: Correlazione log e interfaccia

Infine nella figura 5.16 è riportato il confronto tra il log ottenuto dai canali dell'HIL e il log ottenuto invece dal watcher. Qui si può notare la correlazione tra il riconoscimento da parte del sistema di OCR della stringa `TC KO-PUSH YELLOW`, rappresentativa dell'allarme `AS_FAILURE` atteso dalla sequenza per avviare la reazione da parte del sistema.

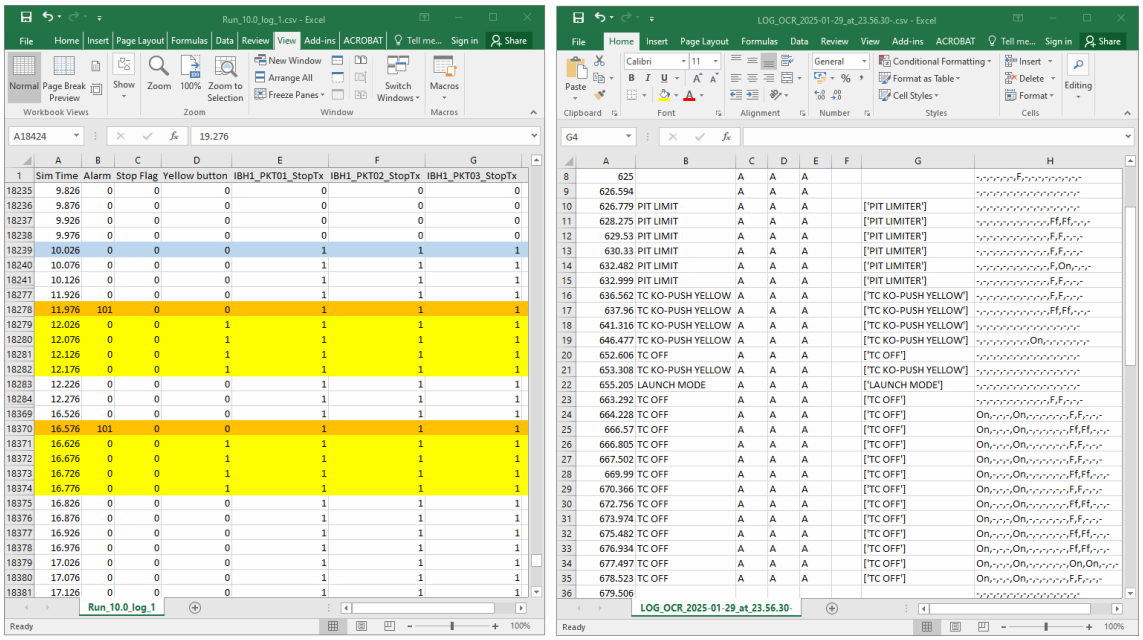


Figura 5.16: Confronto log esecutore e watcher

There is considerable overlap between the intelligence of the smartest bears and the dumbest tourists.

Yosemite Park Ranger

Capitolo 6

Conclusioni e sviluppi futuri

Il progetto presentato in questo elaborato di tesi ha dimostrato in primo luogo la possibilità di implementare un sistema automatizzato per supportare il processo di testing attraverso l'HIL e in secondo luogo la sua efficacia, fornendo una soluzione scalabile, affidabile e adattabile alle esigenze dello sviluppo software nel mondo delle competizioni moderne. Grazie all'automazione delle fasi di generazione delle specifiche, esecuzione dei test e analisi dei risultati il tool *Test Automator* si è rivelato un valido strumento per ridurre i tempi di debugging e migliorare l'affidabilità dei test stessi.

Tra i principali vantaggi emersi dall'implementazione del sistema si possono trovare:

- riduzione degli errori manuali nella definizione e nell'esecuzione dei test, potenzialmente causati da *copia e incolla* tra varie specifiche e dalla carenza di regole ben definite per la stesura dei test,
- maggiore ripetibilità e affidabilità dei risultati ottenuti, permettendo di istituire con il tempo una sorta di database contenente un gruppo di test standard da ripetere quando necessario effettuare ad esempio verifiche di non regressione, riducendone i tempi e alzandone la precisione,
- integrazione con strumenti software avanzati per l'analisi e la validazione dei dati.

Durante le fasi di ricerca e analisi è stato inoltre possibile approfondire strumenti e metodologie di test altamente specializzate, interfacciandosi con componenti hardware specifiche del mondo motociclistico. L'ambiente di sviluppo creato da National Instruments si è rivelato molto interessante da utilizzare in quanto permette di confrontarsi con progetti di complessità variabile, risultando al contempo facile da comprendere a un livello di studio e potente in applicazioni pratiche.

Il progetto in sé aveva l'obiettivo di dimostrare la fattibilità e l'utilità di questo sistema, rimanendo all'interno delle tempistiche prestabilite, per cui vi sono inevitabilmente alcuni aspetti che potrebbero essere migliorati e sviluppati ulteriormente. Tra questi aspetti i possibili sviluppi futuri che sono risultati più interessanti sono:

- **Ottimizzazione del riconoscimento OCR**: migliorare la precisione del riconoscimento testuale per ridurre al minimo gli errori nel rilevamento degli allarmi, oltre che ridurne i tempi di esecuzione e la richiesta di risorse. Modificando il modello utilizzato si potrebbe raggiungere l'obiettivo, ma potrebbe risultare interessante anche la possibilità di utilizzare l'attuale sistema di riconoscimento non efficiente ma comunque preciso per etichettare grandi quantità di video e immagini con lo scopo di addestrare un modello direttamente all'interno dell'azienda in maniera da avere un maggior controllo sul sistema potendo agire a vari livelli di dettaglio.
- **Integrazione con Machine Learning**: sviluppare algoritmi di apprendimento automatico per identificare anomalie nei dati raccolti e prevedere guasti. Un'estensione del sistema che possa permettere di ampliare il loop di test, andando a ricollegare i dati prodotti dall'Executor e dal Watcher con una nuova generazione di test utili per l'approfondimento di un errore riconosciuto autonomamente.
- **Automazione completa del flusso di lavoro**: implementare un'integrazione più stretta con i sistemi di sviluppo software per un testing ancora più efficace, creando un processo che permetta di avere uno storico delle sue versioni, dei test effettuati e dei loro risultati.
- **Estensione del linguaggio per i test**: ampliare il numero di tipologie di test supportate dal validatore e dal generatore di sequenze arrivando a supportare un insieme di test più dettagliati anche per quanto riguarda i possibili parametri utilizzati. Una possibilità ancora più ambiziosa potrebbe essere quella di implementare l'utilizzo di un modello LLM per la definizione di test e la conseguente generazione del relativo codice.

In conclusione, questa tesi rappresenta un punto di partenza per il miglioramento delle pratiche di testing, offrendo un sistema innovativo e flessibile, in grado di evolversi con le esigenze future dell'azienda.

Riferimenti bibliografici

[And] Fernando Andreu. fernandreu/office-ribbonx-editor: An overhauled fork of the original Custom UI Editor for Microsoft Office, built with WPF — github.com. https://github.com/fernandreu/office-ribbonx-editor. [Accessed 01-01-2025].

[auta] The ZeroMQ authors. ZeroMQ — zeromq.org. https://zeromq.org/. [Accessed 01-01-2025].

[Autb] Alma Automotive. Soluzioni per Test e Sviluppo Prodotto - Alma Automotive — alma-automotive.it. https://alma-automotive.it/it/. [Accessed 01-01-2025].

[che] chericksgit. chericksgit/VeriStandAutomation: Automate VeriStand in Python — github.com. https://github.com/chericksgit/VeriStandAutomation. [Accessed 01-01-2025].

[Cor] David Cortesi. PyInstaller Manual; PyInstaller 6.11.1 documentation — pyinstaller.org. https://pyinstaller.org/en/stable/. [Accessed 01-01-2025].

[Duca] Ducati. La storia di ducati. https://www.ducati.com/it/it/heritage. [Accessed 01-01-2025].

[Ducb] Ducati. Motoe il prototipo. https://www.ducati.com/it/it/azienda/innovation/moto-e/prototipo. [Accessed 01-01-2025].

[Hoa] Nhan Nguyen Hoang. Understanding the testing environments in automotive development: Mil, sil, pil, and hil. https://blog.nashtechglobal.com/understanding-the-testing-environments-in-automotive-development-mil-sil-pil-and-hil/. [Accessed 01-01-2025].

[IEE86] IEEE. Ieee standard for software unit testing. *ANSI/IEEE Std 1008-1987*, pages 1–28, 1986.

[Jai] JaidedAI. GitHub - JaidedAI/EasyOCR: Ready-to-use OCR with 80+ supported languages and all popular writing scripts including Latin, Chinese, Arabic, Devanagari, Cyrillic and etc. — github.com. https://github.com/JaidedAI/EasyOCR. [Accessed 01-01-2025].

[Jha10] Ashutosh Kumar Jha. Development of test automation framework for testing avionics systems. In *29th Digital Avionics Systems Conference*, pages 6.E.5–1–6.E.5–11, 2010.

[Jos19] Adit Joshi. *Automotive Applications of Hardware-in-the-loop (HIL) Simulation*. SAE International, 2019.

[Man] Marco Mansell. https://www.ducatimotor.it/lastoria.htm. [Accessed 01-01-2025].

[Mic] Microsoft. .NET | Costruire. Test. Distribuisci. — dotnet.microsoft.com. https://dotnet.microsoft.com/it-it/. [Accessed 01-01-2025].

[NIa] NI. Creating Real-Time Stimulus Profiles in NI VeriStand — ni.com. https://www.ni.com/en/shop/data-acquisition-and-control/application-software-for-data-acquisition-and-control-category/what-is-veristand/creating-real-time-stimulus-profiles-in-ni-veristand.html. [Accessed 01-01-2025].

[NIb] NI. Guide to Effective Test Software Development with TestStand — ni.com. https://www.ni.com/en/support/documentation/supplemental/06/guide-to-effective-test-software-development-with-teststand.html. [Accessed 01-01-2025].

[NIc] NI. Test and Measurement Systems, a part of Emerson — ni.com. https://www.ni.com/en.html. [Accessed 01-01-2025].

[Nid] Ni. Using NI VeriStand with Other Software Environments to Create Real-Time Test Applications — ni.com. https://www.ni.com/en/shop/data-acquisition-and-control/application-software-for-data-acquisition-and-control-category/what-is-veristand/using-ni-veristand-with-other-software-environments-to-create-re.html. [Accessed 01-01-2025].

[NIe] NI. VeriStand Python Documentation, niveristand-python 3.2.2 documentation — niveristand-python.readthedocs.io. https://niveristand-python.readthedocs.io/en/latest/. [Accessed 01-01-2025].

[NIf] NI. What is hardware-in-the-loop (hil)? — ni.com. https://www.ni.com/en/solutions/transportation/hardware-in-the-loop/what-is-hardware-in-the-loop-.html. [Accessed 01-01-2025].

[NIg] NI. What is NI LabVIEW? Graphical Programming for Test & Measurement — ni.com. https://www.ni.com/en/shop/labview.html. [Accessed 01-01-2025].

[o36a] o365devx. Deploy and publish Office Add-ins - Office Add-ins — learn.microsoft.com. https://learn.microsoft.com/en-us/office/dev/add-ins/publish/publish. [Accessed 01-01-2025].

[o36b] o365devx. Office Add-ins platform overview - Office Add-ins — learn.microsoft.com. https://learn.microsoft.com/en-us/office/dev/add-ins/overview/office-add-ins. [Accessed 01-01-2025].

[pyt] pythonnet. Python.NET — pythonnet.github.io. https://pythonnet.github.io/. [Accessed 01-01-2025].

[SE06] Roger Skjetne and Olav Egeland. Hardware-in-the-loop testing of marine control system. *Modeling, Identification and Control*, 27:239–258, 10 2006.

[Vis20] S Vishnu. Simulation of hybrid boost fly back converter: A virtual hardware-in-the-loop simulation approach. *Perspectives in Communication, Embedded-systems and Signal-processing-PiCES*, 4(5):112–116, 2020.

[Wika] Wikipedia. Ducati cucciolo - wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/Ducati_Cucciolo. [Accessed 01-01-2025].

[Wikb] Wikipedia. Visual Basic for Applications - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Visual_Basic_for_Applications. [Accessed 01-01-2025].

Ringraziamenti

Desidero ora ringraziare il professor Mirko Viroli e l'azienda Ducati, in particolare il mio referente Andrea *Denny* De Vito, per avermi innanzitutto dato la possibilità di cimentarmi in questo progetto, che si è rivelato essere per me una sfida e al tempo stesso una fonte di grandi soddisfazioni, ma soprattutto per avermi accompagnato nella conclusione del mio percorso universitario. Un ringraziamento poi va a Carlo Edoardo Malaspina che insieme a Denny mi ha aiutato e indirizzato durante questi mesi, e per aver entrambi dimostrato grande disponibilità e tanta pazienza nei miei confronti. Un grazie infine a tutto il team Ducati Corse che mi ha accolto a braccia aperte e mi ha reso partecipe della squadra che fin da ragazzino ho ammirato in televisione a ogni gara.

I miei ringraziamenti vanno poi a tutte le persone che, durante i primi passi di questo mio percorso di studi, durante le ultime tirate verso il traguardo o durante tutto il cammino in questi anni, mi hanno supportato e aiutato in mille modi diversi. Grazie a tutti quelli che mi hanno dedicato il loro tempo, che mi hanno aiutato a superare momenti difficili e che si sono goduti con me i momenti felici.

Grazie a chi ha avuto la pazienza di sopportarmi, a chi ha creduto in me quando avevo smesso di farlo anche io, a chi ha voluto essere partecipe di questa parte della mia vita nel bene e nel male, grazie per avermi sempre aiutato a rialzarmi.

Sono felice di portarmi a casa il proverbiale pezzo di carta, ma ancora di più sono grato di aver avuto l'occasione di conoscere tutti voi che state leggendo queste parole, che avete contribuito a rendermi la persona che sono oggi arricchendomi di cose che non si possono ottenere dai libri.

Grazie mille a tutti e *buon proseguimento di vita.*