UNIVERSITÀ POLITECNICA DELLE MARCHE

Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Model Based Design nel settore automotive: implementazione della logica di controllo di un on board charger e qualificazione del software con gli standard ISO26262 e AUTOSAR

Model Based Design in the automotive field: modelling and verifying the supervisory control logic of an on-board charger according to the ISO26262 and AUTOSAR standards

Relatore

Prof. GIANLUCA IPPOLITI

Dott. PAOLO BIZZARRI

Candidato

LUCA MEZZANOTTI

Ottobre 2021

Sommario

L'obiettivo di questo elaborato è duplice: in primo luogo si tratteranno i concetti teorici alla base della metodologia del Model Based Design in ambito automotive, successivamente si applicheranno quest'ultimi ad un problema reale.
Nello specifico il primo capitolo, dopo un confronto tra la metodologia di sviluppo software tradizionale e il Model Based Design, illustra i vari step che compongono lo schema di progettazione a V. In questa fase vengono discussi sia gli aspetti teorici sia una caratterizzazione di questi tramite i tool proposti da MathWorks. All'interno del secondo capitolo viene preso in considerazione un caso di studio reale. L'obiettivo è quello di progettare un supervisore logico per un carica batterie contenuto all'interno dei veicoli elettrici. In questo capitolo, verrà percorso step-by-step lo schema a V, fino alla fase di testing di tipo Processor-In-the-Loop. Gli ultimi due capitoli trattano i concetti base delle norme adottate nel settore automotive. Più nel dettaglio, il capitolo tre introduce la norma ISO 26262 e come questa può essere adottata nel processo di sviluppo software basato su metodologia Model Based Design. Il capitolo quattro fornisce le conoscenze di base dell'architettura AUTOSAR ed illustra come, tramite la metodologia trattata nell'elaborato, sia più facile progettare componenti compatibili con lo standard automobilistico AUTOSAR.

La parte finale del seguente elaborato è composta da tre appendici. La prima illustra il funzionamento alla base dei convertitori di tensione presenti all'interno degli on-board charger. La seconda descrive il modello matematico delle batterie al litio ed i processi di ricarica a cui esse sono sottoposte. Infine, la terza appendice riporta la porzione di codice C che è stato scritto per la gestione della comunicazione seriale, implementata per la fase di test di tipo Processor-In-the-Loop.

Ringraziamenti

Al termine di questo lavoro di tesi vorrei ringraziare il mio relatore Prof. Gianluca Ippoliti, sempre presente, puntuale e disponibile.

Un sentito grazie al Dott. Paolo Bizzarri, correlatore di tesi, per il supporto costante, le dritte indispensabili e la sua complicità nella realizzazione di ogni capitolo della mia tesi.

Grazie a tutti i Technology Leader di Teoresi S.p.A, che mi hanno accolto a braccia aperte fin da subito.

A Mamma e Babbo, al loro costante sostegno ed ai loro insegnamenti senza i quali oggi non sarei ciò che sono. Senza di voi, tutto questo non sarebbe stato possibile.

Grazie Sara, mia sorella, per avermi supportato o meglio sopportato, in questi anni nonostante la distanza.

Grazie Nonni, per aver gettato le basi di questa fantastica famiglia.

Un grazie a tutti gli amici incontrati durante questo fantastico percorso, grazie ai miei storici coinquilini Denis, Giaco, Fede per avermi fatto divertire tutte le sere in quel di Ancona.

Grazie a tutti i miei amici storici di Sant'Angelo, per avermi fatto trovare aperitivi caldi e birre fredde tutte le volte che tornavo a casa.

Un grazie immenso ad Angela, per aver condiviso con me questo percorso, non solo all'università.

"La semplicità è la massima raffinatezza"
Regis McKenna

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

MBD
Model Based Design

OBC
On Board Charger

BMS
Battery Management System

ROI
Return On Investment

SDLC
Systems development life cycle

SIL
Software In the Loop

PIL
Model In the Loop

HIL
Hardware In the Loop

UTP
Unit Test Plan

UAT
User Acceptance Test

UML

Unified Modeling Language

SysML

System Modeling Language

FSM

Finite State Machine

Capitolo 1

Model Based Design

1.1 Introduzione alla metodologia

Al giorno d'oggi, ridurre i tempi di produzione e i costi dell'intero processo è di fondamentale importanza se si vogliono mettere sul mercato prodotti innovativi. Lavorare in modo efficiente è indispensabile per ottenere successo all'interno di un mercato globalizzato, soprattutto per le industrie ad alta tecnologia come quella automobilistica, aerospaziale e delle comunicazioni, dove i controlli elettronici sono una parte vitale e preponderante di ogni nuovo prodotto. Sempre più spesso, all'interno dei prodotti in questione vi è una parte consistente di codice inserito in schede elettroniche con lo scopo di controllare e supervisionare i processi elaborativi. Vista la crescita esponenziale che la complessità dei sistemi sta affrontando in questi anni, le fasi di progettazione e sviluppo basate sul modello del sistema stanno prendendo sempre più piede a dispetto delle classiche metodologie di sviluppo software.
L'immagine 1.1 mostra come le linee di codice sono cresciute negli ultimi 30 anni e come la domanda di software è aumentata. Una delle prime cose da notare è che, intorno agli anni '80, c'erano circa mille righe di codice su un veicolo generico: questo significa che il software era contenuto e c'erano poche e semplici operazioni da eseguire. Per esempio, il costo di questo software e delle parti di elettronica era inferiore al 9% del costo totale del veicolo. Un'altra cosa che spicca, sono le decine di migliaia di righe di codice che vengono inserite intorno agli anni '90: questo spiega come la domanda stava crescendo così rapidamente solo pochi anni dopo l'introduzione dei componenti elettronici in sistemi puramente meccanici come le automobili. In quegli anni, il costo del software e delle parti elettroniche è aumentato fino al 20% del costo totale del veicolo. Un ultimo punto da notare è che, nel 2010, le righe di codice sono già nell'ordine di alcune decine di milioni, un certo ordine di grandezza rispetto a venti anni prima. Qui si evince come il solo software

componga circa il 30% del costo del veicolo alimentato con un motore termico e il 65% del costo del veicolo alimentato con motore elettrico/ibrido. Nell'ultimo caso, più della metà dei costi sono dovuti alla scrittura di codice, da qui la necessità di impiegare metodologie più evolute per la progettazione e realizzazione delle decine di centraline che sono presenti all'interno del mezzo.

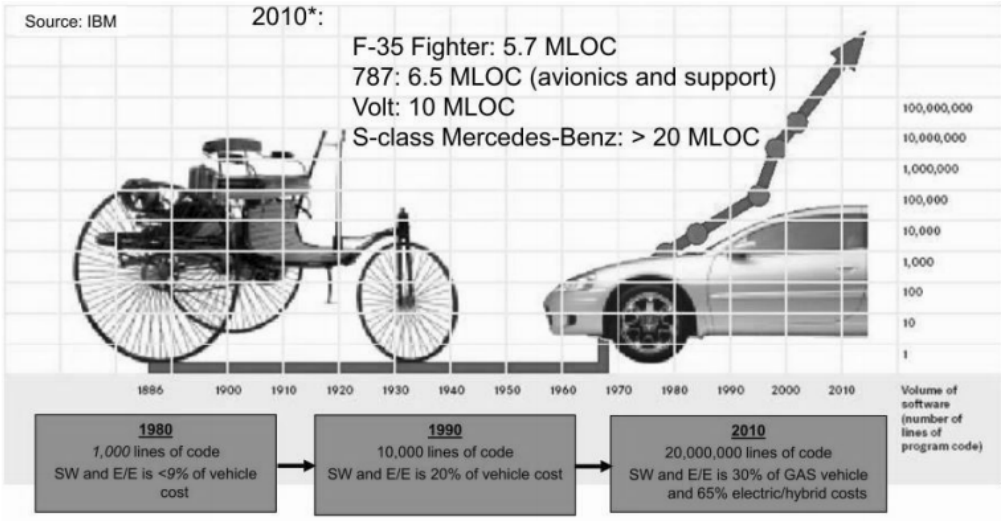

Figura 1.1: Andamento delle linee di codice nel settore automobilistico

Il grafico 1.1 arriva al 2010, le ipotesi di uno studio di IBM prevedeva che una macchina di classe premium come una Mercedes-Benz S-class potesse contenere una quantità di codice di oltre un gigabyte pari a circa 20 milioni di righe di codice [1].

Va da se che per un numero così elevato di istruzioni la complessità del sistema cresca in modo esponenziale. Le ragioni di un aumento così consistente di software stanno nella richiesta di nuove funzionalità sempre più complesse, ad esempio applicazioni di guida autonoma o sistemi di aiuto alla guida, ADAS. Un'altra motivazione può essere trovata nell'impiego di hardware sempre più prestante e sempre più a basso costo presente oggi nel mercato.

Inoltre, l'elettronica nelle auto consente ridurre i consumi, aumentare sicurezza, prestazioni e comfort. L'ambiente automobilistico moderno necessita di software riutilizzabile in modo da poter essere facilmente integrato in più autovetture della stessa tipologia o della stessa casa automobilistica. Questo aspetto implica l'utilizzo della metodologia Model Based Design (MBD) in quanto questa consente al progettista di creare algoritmi partendo da linguaggi grafici di modellazione deterministici (diagrammi a blocchi o macchine a stati finiti), seguendo un preciso flusso di lavoro che genera alla fine un codice di qualità riutilizzabile in altri progetti simili.

1.2 Confrontro tra MBD e metodologia tradizionale

Nei processi di progettazione tradizionali, le informazioni di progettazione, dette anche requisiti, vengono comunicati e gestiti direttamente sulla documentazione fornita dal cliente. Spesso questa documentazione è difficile da comprendere. Il codice viene creato manualmente direttamente interpretando i requisiti espressi in linguaggio naturale. Questo processo porta a fasi di sviluppo più dilatate nel tempo e ad un innalzamento delle possibilità d'errore dovuti alla mal interpretazione del documento stesso. Inoltre, modifiche future ai requisiti potrebbero rendere inadatto tutto il codice sviluppato in precedenza, rendendo l'integrazione di queste un processo complesso e oneroso.

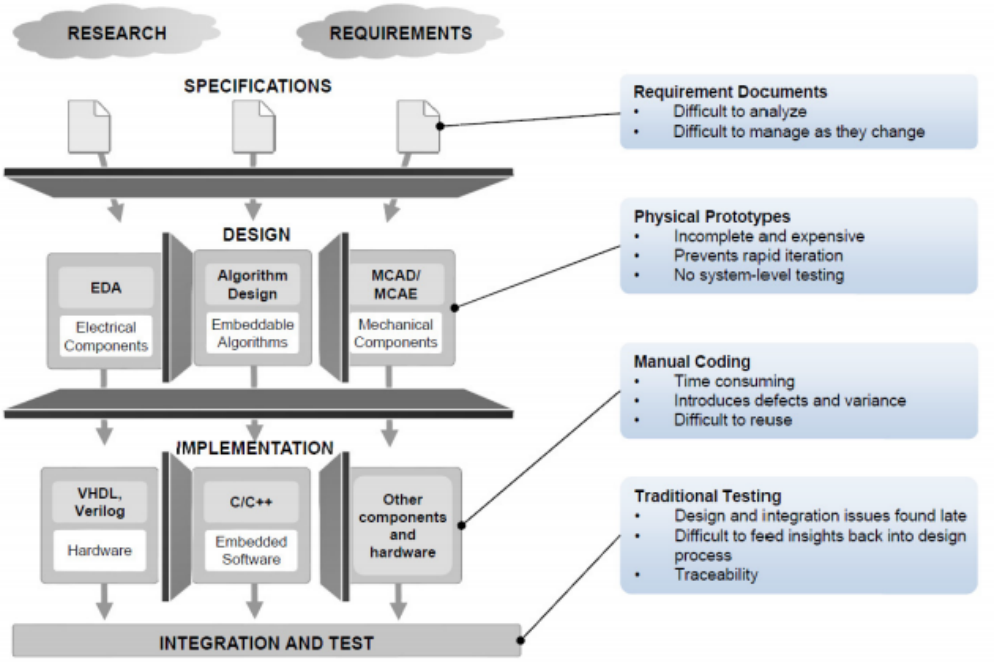

Figura 1.2: Metodologia tradizionale

In un processo di sviluppo tradizionale, i requisiti, la progettazione, l'implementazione e le attività di test vengono eseguiti in sequenza in ambienti di strumenti diversi, con molti passaggi manuali, in figura 1.2 viene rappresentato il work flow. Scomponendo il processo di sviluppo secondo la metodologia tradizionale possiamo evidenziare le seguenti macrofasi:

Specification è la prima fase del processo in cui si considerano tutti i requisiti che il prodotto finale deve rispettare. Questi generalmente vengono gestiti

tramite degli strumenti di gestione della documentazione come IBM DOORS. Si incontrano problemi nel momento in cui si hanno degli aggiornamenti dei requisiti in quanto manca un mapping tra i singoli requisiti e i processi elaborativi che vanno a fornire la soluzione ad essi.

Design è la fase in cui vengono definiti i requisiti specifici per le singole componenti: elettronica, algoritmi, meccanica ecc. Con questo approccio, non è possibile eseguire una rapida prototipazione dei prodotti sviluppati rischiando di propagare errori nelle fasi successive.

Implementation è la fase principale di questa metodologia ovvero la scrittura manuale del codice basandosi direttamente sui requisiti e sulla fase di design eseguita a monte. Questa fase è la più onerosa sia in ottica temporale, dunque la più costosa, sia in ottica intellettuale, ovvero scrivere del codice basandosi solo su informazioni testuali è molto complesso e si rischia di ottenere un comportamento diverso da quello richiesto.

Integration and test è l'ultima fase di questo processo lineare, in cui si convoglia tutto il lavoro fatto nella fase precedente e si operano dei test per verificare se il comportamento effettivo è uguale a quello atteso. Si riscontrano delle problematiche nel momento in cui questa condizione non è soddisfatta. Questo poichè è difficile, partendo dal codice, risalire ai requisiti non soddisfatti.

Al contrario della metodologia tradizionale vista in figura 1.2, la tecnica di Model Based Design, in figura 1.3, è di tipo iterativo: si nota una fase di test e verifica dei risultati che si sviluppa in modo trasversale rispetto a tutti gli altri processi. Questo meccanismo consente di testare i componenti sviluppati, sia in fase di design sia in fase implementativa, prima di procedere con le fasi successive evitando dunque di propagare eventuali errori.

Un'altra caratteristica che contraddistingue il MBD è l'ambiente di sviluppo che viene utilizzato. Come si vede in figura 1.3 la macro fase di design racchiude al suo interno altre fasi come la definizione dei modelli fisici alla base del problema e l'implementazione degli algoritmi di controllo, entrambi questi processi condividono tra di loro l'ambiente di sviluppo. Questa caratteristica permette di legare ciò che il progettista realizza rispetto a ciò che il cliente desidera. Scopo di questo meccanismo è la facile risoluzione di eventuali problemi che possono verificarsi in caso di modifica dei requisiti iniziali.

1.3 Vantaggi del Model Based Design

Le organizzazioni che adottano la progettazione basata su modelli realizzano risparmi che vanno dal 20 al 60%, se confrontati ai metodi tradizionali, [1], [2].

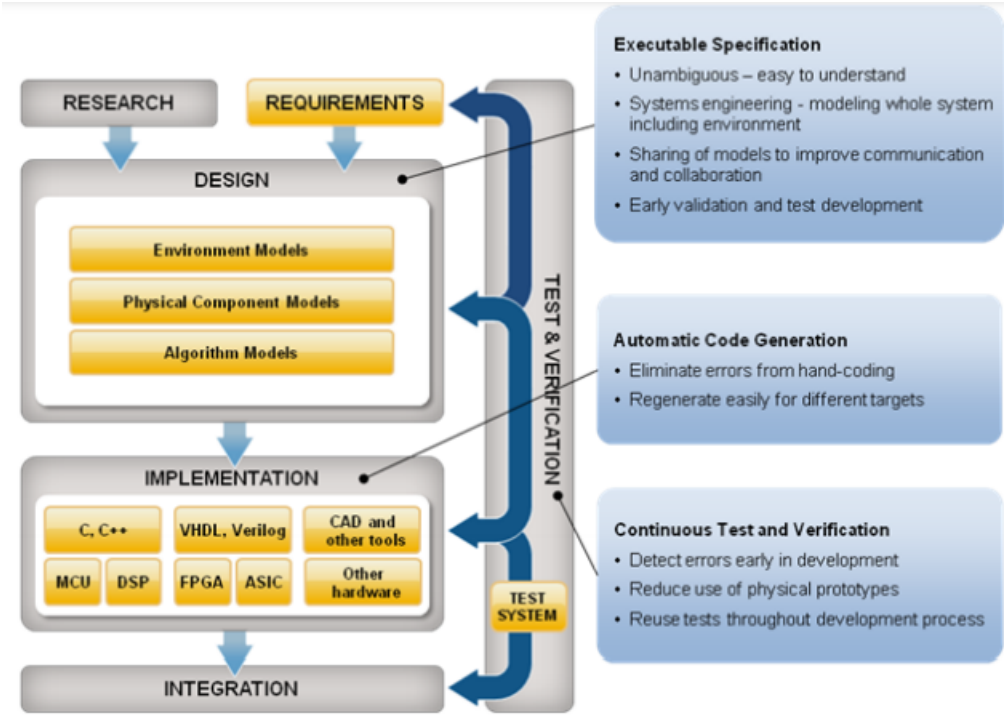

Figura 1.3: Model Based Design

La maggior parte di questi risparmi deriva da una migliore analisi dei requisiti, combinati con test e verifiche precoci e continue durante tutte le fasi. Poiché i requisiti e i progetti vengono simulati utilizzando i modelli, i difetti vengono scoperti molto prima nel processo di sviluppo. Questo implica che la risoluzione dell'errore è molto più economica rispetto ad una eventuale propagazione nelle fasi successive, figura 1.4.

In figura 1.5, è stata riportata una simulazione di costi e valutazione dell'indice ROI (Return on Investment) in un progetto automotive, [2], sia considerando la metodologia tradizionale, sia il Model Based Design. Le voci più interessanti, riportate nella prima colonna sono: tempo di sviluppo, numero di progettisti e costo totale del progetto. In particolare si nota come, nel caso in cui si adotti il MBD, tutti e tre gli indicatori scelti assumono valori inferiori rispetto alla metodologia di sviluppo tradizionale. Ovvero, a fronte di minor risorse umane, minor costi di progetto e minor tempo di sviluppo si riesce ad ottenere un prodotto migliore e più mantenibile. La percentuale riportata nell'ultima riga, 38.8% è la percentuale di risparmio, in termini di costi, che si ottiene con MBD.

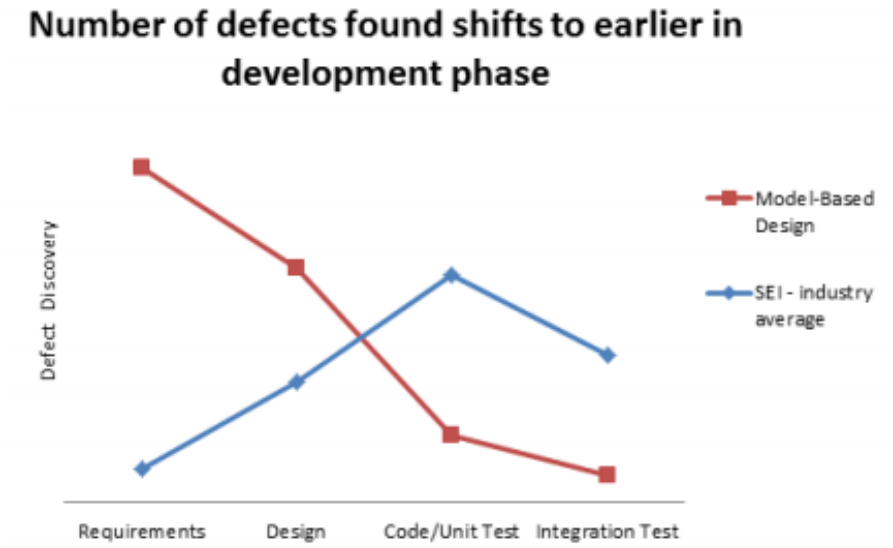

Figura 1.4: Vantaggi MBD

	Auto MBD	Auto Not MBD
Devel time Months	11.6	12.5
% behind schedule	43.1%	52.1%
Months behind	3.4	3.2
Ave Delay Months	1.47	1.67
% cancelled	11.0%	10.6%
Months lost to cancellation	3.6	4.1
SW Developers/proj	10.6	13.2
HW Developers/proj	6.8	9.1
Total project developers	17.4	22.2
Average Developer months/project	201.4	278.0
Developer months lost to schedule	25.7	37.1
Developer months lost to cancellation	6.9	9.6
Total developer months/ project	234.0	324.7
At $10,000/developer month		
Average developer cost/project	$2,014,003	$2,779,762
Average cost to delay	$256,594	$371,316
Total developer cost/project	**$2,270,597**	**$3,151,078**
	MBD Adv	**38.8%**

Figura 1.5: Comparazione ROI

1.4 Diagramma a V

Nello sviluppo del software, lo schema a V in figura 1.6, rappresenta un processo di sviluppo che può essere considerato un'estensione del modello tradizionale visto nei

capitoli precedenti, [3], [4]. Invece di scendere in modo lineare, le fasi del processo vengono piegate verso l'alto dopo la fase di codifica, per formare la tipica forma a V. Lo schema a V, come si vede in figura 1.6, prevede delle relazioni tra ogni fase del ciclo di sviluppo e la fase di test ad essa associata. Gli assi orizzontali e verticali rappresentano rispettivamente il tempo o la completezza del progetto (da sinistra a destra) e il livello di astrazione (astrazione a grana più grossa in alto).
Più in generale, il Software Development Life Cycle (SDLC) consiste in una metodologia per pianificare, creare, testare e distribuire un sistema informatico. All'interno del SDLC possiamo applicare lo schema a V di progettazione e sviluppo per ottenere un prodotto finito. Per cui la metodologia SDLC è un concetto più esteso e al suo interno troviamo la tecnica in esame ovvero il Model Based Design o MBD.

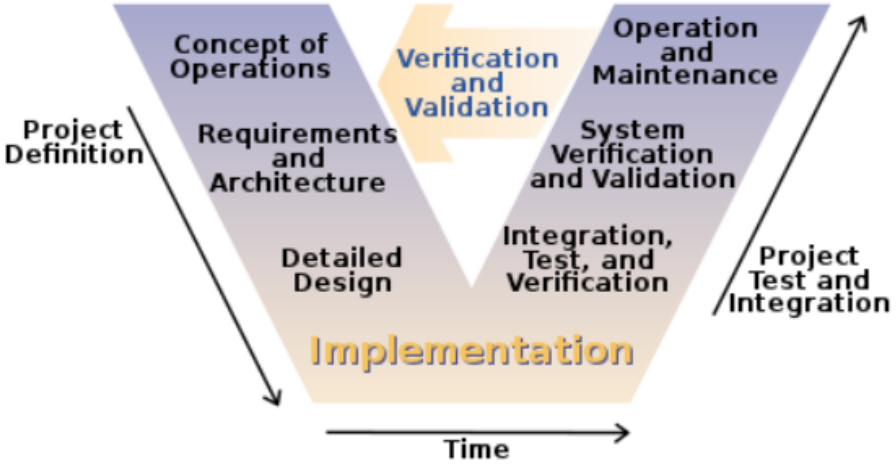

Figura 1.6: Schema a V - SDLC

1.4.1 Descrizioni delle fasi

Analisi dei Requisiti (Requirements analysis)

Nella fase di analisi dei requisiti, il primo passo nel processo di verifica, i requisiti del sistema vengono raccolti analizzando le esigenze dei clienti che commissionano il progetto. Questa fase si occupa di stabilire ciò che il sistema ideale deve svolgere. Tuttavia, non determina come verrà progettato o realizzato il software. Di solito, gli utenti vengono intervistati e viene generato un documento chiamato documento dei requisiti utente. Questo descriverà in genere i requisiti funzionali, di interfaccia, di prestazioni, di dati, di sicurezza, ecc. Viene utilizzato dagli analisti aziendali per comunicare agli utenti la loro comprensione del sistema. In una fase successiva, gli

utenti esaminano il documento redatto dagli analisti poiché questo fornirà le linee guida per i progettisti. Di conseguenza se i requisiti sono espressi in modo errato o contengono delle imprecisioni, queste si ripercuoteranno in modo diretto sul prodotto finale. Esistono diversi metodi per raccogliere i requisiti tra cui: interviste, questionari, analisi documentale, osservazione, prototipi usa e getta, use cases.

Progettazione del sistema (System design)

La fase di design del sistema è il punto in cui gli ingegneri analizzano e comprendono l'attività del sistema proposto studiando il documento dei requisiti utente generato nella fase precedente. Scopo principale di questa fase è definire quali saranno le metodologie per implementare in modo corretto e coerente con il documento dei requisiti le funzionalità desiderate dal cliente. Se uno qualsiasi dei requisiti non è fattibile, l'utente viene informato del problema. Viene trovata una risoluzione a livello di requirements analysis e il documento prodotto da questa fase viene aggiornato di conseguenza. Output della fase di design è un documento di specifica del software che funge da modello per la fase di sviluppo. Questo documento contiene l'organizzazione generale del sistema. In questa fase verrà prodotta anche altra documentazione tecnica come diagrammi di entità, dizionario dei dati.

Progettazione dell'architettura e dei componenti (Architecture design e module design)

La fase in esame può essere definita anche progettazione di alto livello. La linea di base nella scelta dell'architettura è la seguente: elencare i vari moduli del sistema, descrivere in breve le funzionalità di ciascun modulo, le loro relazioni di interfaccia, dipendenze, tabelle di database, diagrammi dell'architettura, dettagli tecnologici, ecc. In questa fase verranno definiti i test case per le fasi successive. In una fase successiva si entra nel dettaglio di ciascun modulo e si esegue sullo stesso la fase di design, in modo da ottenere una progettazione di basso livello del sistema complessivo. Si definiscono inoltre, i test case per verificare le funzionalità dei singoli moduli.

Implementazione (Implementation)

Dopo aver definito per ogni singolo modulo che cosa esso deve svolgere e come deve farlo, si procede nella fase di scrittura codice vera e propria. In questa fase i progettisti devono attenersi in modo stringente ai requisiti delineati nelle fasi superiori per non aggiungere o modificare funzionalità non richieste dal cliente.

Integrazione (Integration)

Questa fase verrà discussa nel dettaglio nello schema proposto per il Model Based Design. Essa coinvolge tutte le fasi di integrazione del codice sviluppato all'interno del sistema hardware partendo dal Software in the loop (SIL), Processor in the loop (PIL) e in fine Hardware in the loop (HIL).

Fase di validazione (Validation phases)

La differenza alla base tra la fase di verifica e validazione è che la prima, generalmente, è il controllo che il prodotto ottenuto al termine di una fase sia congruente con il semilavorato avuto come punto di partenza di quella fase. Per esempio, nella realizzazione di un modulo, è una tipica verifica il controllo che le specifiche del modulo siano state rispettate sia come interfaccia che come funzionalità. La validazione invece, è un'attività di controllo mirata a confrontare il risultato di una fase del processo di sviluppo con i requisiti del prodotto – tipicamente con quanto stabilito dal contratto o, meglio, dal documento di analisi dei requisiti. Un comune esempio di validazione è il controllo che il prodotto finito abbia funzionalità e prestazioni conformi con quelle stabilite all'inizio del processo di sviluppo, [5].
Nel modello V, ogni fase della fase di verifica ha una fase corrispondente nella fase di validazione, nello specifico si ha:

Unit testing Gli Unit Test Plan (UTP) sono sviluppati durante la fase di progettazione del modulo. Questi UTP vengono eseguiti per eliminare i bug a livello di codice o a livello di unità. Un'unità è l'entità più piccola che può esistere indipendentemente, ad esempio un modulo che compone il programma. Il test delle unità verifica che l'entità più piccola possa funzionare correttamente quando è isolata dal resto dei codici/unità.

Integration testing Questi test verificano che le unità create e testate in modo indipendente possano coesistere e comunicare tra loro. I risultati dei test sono condivisi con il team del cliente.

System testing Questa tipologia di test vengono sviluppati durante la fase di progettazione del sistema. A differenza degli UTP e dei test di integrazione, i piani di test di sistema sono composti dal team aziendale del cliente. Questa fase garantisce che le aspettative dell'applicazione sviluppata siano soddisfatte. L'intera applicazione viene testata per la sua funzionalità, interdipendenza e comunicazione. Il test di sistema verifica che i requisiti funzionali e non funzionali siano stati soddisfatti.

User Acceptance Testing I piani di User Acceptance Test (UAT) vengono sviluppati durante la fase di analisi dei requisiti. I test cases sono formulati da

utenti business. UAT viene eseguito in un ambiente utente che assomiglia all'ambiente di produzione, utilizzando dati realistici. UAT verifica che il sistema consegnato soddisfi i requisiti dell'utente e che il sistema sia pronto per l'uso in tempo reale.

1.5 Schema a V applicato a MBD

Lo schema 1.6 descritto nelle sezioni generali, proponeva una metodologia generica applicabile ad un qualunque sistema informativo. In questa sezione verrà specializzato per il suo utilizzo in ambito Model Based Design. Lo schema proposto è quello in figura 1.7. La struttura di questo schema è la medesima di quello

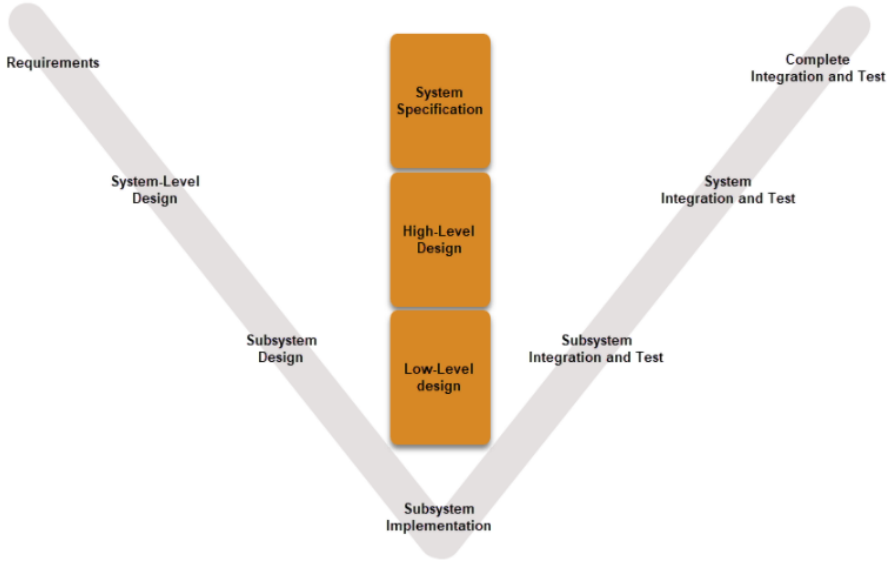

Figura 1.7: Schema a V - MBD

generico proposto in precedenza. Viene mantenuta la struttura a "V" in cui sul lato sinistro troviamo le fasi di analisi dei requisiti e le fasi di design, sulla base la fase implementativa e infine sul lato destro tutte le fasi di verifica e validazione.
Se applichiamo questa metodologia all'approccio MBD è possibile creare una rappresentazione virtuale di un sistema del mondo reale. Inoltre grazie alla simulazione è possibile vedere la risposta del sistema sotto diverse condizioni di lavoro per valutare il comportamento reale del sistema fisico.

La figura 1.8 rappresenta il flusso di lavoro che propone MathWorks per lo sviluppo di sistemi basati sulla metodologia MBD.
Nelle sezioni successive, seguendo il work flow proposto in figura 1.8, si analizzeranno in modo dettagliato le singole fasi.

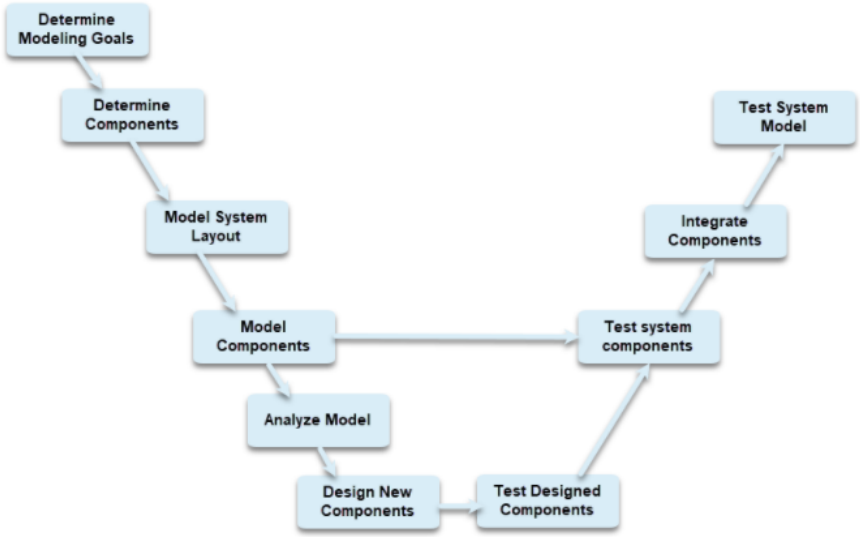

Figura 1.8: Work flow - MBD

1.5.1 Determinazione degli scopi della modellazione (Determine Modelling Goals)

Si tratta della fase iniziale comune ad ogni progetto ovvero l'analisi dei requisiti. Nel nostro caso specifico i requisiti coinvolgono il modello in esame ovvero quell'entità che deve essere controllata o gestita in qualche modo.
La raccolta e analisi dei requisiti è un'attività che vede coinvolti il Project Manager, il Team di progetto e gli stakeholder del progetto stesso. L'analisi dei requisiti è quindi il processo di definizione delle aspettative degli stakeholder riguardanti i prodotti di un determinato progetto. Nel caso del MBD i requisiti tipici comprendono, ad esempio, quali sono le azioni che il sistema deve compiere a fronte di un determinato input, che tipologia di input il sistema è in grado di tollerare, come il sistema deve adattarsi al variare delle condizioni ambientali ecc. Per definire in modo puntale, preciso e non ambigua ciò che il sistema deve fare è necessario redigere un documento più schematico possibile, il quale conterrà i requisiti stessi, in modo da evitare ambiguità nell'interpretazione.
In ottica MBD possiamo trovare, come in tutti i progetti, innumerevoli stakeholders. La natura di questi varia dal cliente stesso, ovvero colui che commissiona il progetto, all'utilizzatore del software o hardware frutto del progetto. I fornitori dei componenti facenti parte del progetto stesso sono considerati stakeholders. La fase che precede la stesura del documento dei requisiti è quella di raccolta delle informazioni: si intervistano tutte le persone o entità che entrano in gioco durante l'intero ciclo di vita del prodotto con lo scopo di capire che cosa e come il prodotto deve risolvere dei problemi dati. Di seguito viene riportato un elenco di alcune

delle metodologie usate per raccogliere le informazioni.

Interviste one-to-one Si tratta di incontri faccia-a-faccia con ciascun stakeholder per raccogliere le aspettative e i requisiti funzionali di ciò che dovrà essere realizzato dal progetto dal punto di vista di ciascun stakeholder. E' importante che vengano raccolte indicazioni rilevanti per gli obiettivi del progetto. In questi incontri è necessario precisare che non verrà eseguita una fase di analisi dei requisiti ma verranno solo discusse le problematiche e le possibili soluzioni che il progetto porterà.

Focus group A differenza delle interviste one-to-one, viene coinvolto un gruppo di persone. Durante questi incontri viene utilizzata la tecnica del brainstorming per ricavare più informazioni possibili. Condizione necessaria per l'utilizzo di tale tecnica è che i partecipanti abbiano un pensiero completamente aperto in modo da confrontarsi costruttivamente con gli altri individui.

Workshop Analogo della metodologia precedente, con la differenza che anche lo staff del progetto viene incluso. Ciò consente non solo di individuare le possibili problematiche ma anche di proporre possibili soluzioni. Questo fatto porta ad un maggior coinvolgimento degli stakeholders in quanto risulta meno asettico del focus group e si entra maggiormente sulle questioni di dettaglio.

Osservazione diretta Questa modalità implica un sopralluogo in situazioni in cui è possibile prendere visione diretta di quanto richiesto. Nell'ambito MBD, ovvero in tutte quelle situazioni in cui il prodotto finale deve andare a risolvere una problematica, è utile che lo staff del progetto veda con i propri occhi il problema da arginare in modo da rendersi conto in prima persona dell'entità della richiesta.

Processo di analisi dei requisiti

Quando si è ultimata la fase di raccolta delle informazioni e delle aspettative che i vari stakeholder hanno, si devono tradurre queste in concetti più formali e meno ambigui ovvero i requisiti veri e propri.

Se vi dovessero essere problemi relativi all'ambiguità di qualche informazione, questi devono essere risolti prima di passare alla modellazione dei requisiti. Inoltre per raccogliere le informazioni e verificare se qualcuna di essa è in contrasto è utile modellare le richieste degli stakeholders con diagrammi dei casi d'uso.

Esistono diverse tecniche utilizzate per l'analisi dei requisiti. Di seguito è riportato un elenco di alcune di queste:

Diagrammi di flusso Questa tecnica viene utilizzata per rappresentare visivamente sistemi e processi che sono complessi e difficili da descrivere in modo

discorsivo. In figura 1.9 è riportato un esempio di diagramma che ha lo scopo di definire un requisito di tipo accettazione. Nello specifico il requisito definito tramite diagramma, se espresso nel linguaggio naturale sarebbe:

"All'avvio del sistema si entra nella fase di login, se l'utente non è autorizzato si torna nella fase di login. Se l'utente ha tentato l'accesso non autorizzato per tre volte si torna nella fase iniziale. Se l'utente è autorizzato accede al sistema. Se l'utente desidera uscire dal sistema si torna nella fase iniziale altrimenti si rimane dentro il sistema."

Confrontando la stessa richiesta espressa sia in modo visivo, sia testuale è evidente il vantaggio che si ottiene esprimendo i requisiti in un linguaggio visivo, come i diagrammi, ovvero la semplicità di interpretazione e la non ambiguità.

Il diagramma descrive varie entità o componenti del progetto e le loro relazioni con l'aiuto di notazioni e simboli standardizzati. Avendo una rappresentazione grafica risulta più semplice individuare e correggere eventuali carenze o ridondanze.

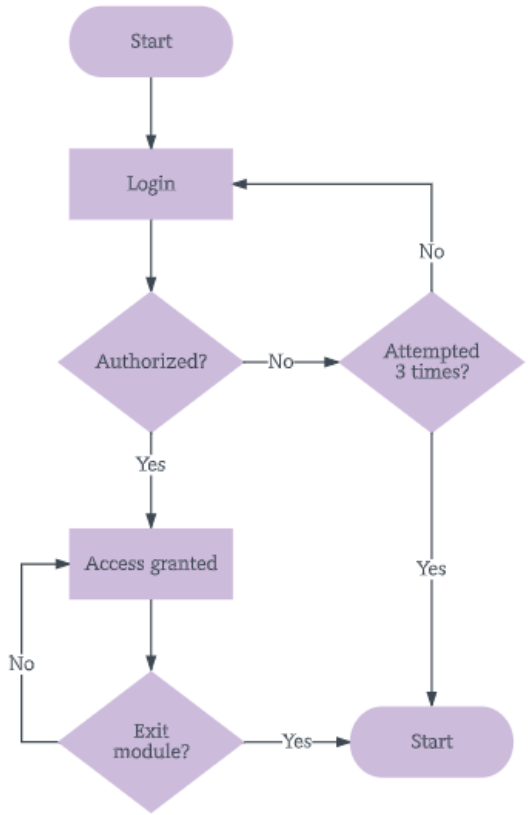

Figura 1.9: Esempio di diagramma dei requisiti

Role Activity Diagrams (RAD) In questa tipologia di diagrammi vengono evidenziati quali sono i ruoli a cui ciascuna entità deve adempiere. Il classico diagramma RAD è composto da archi, ai quali sarà associata un'etichetta di tipo verbale (e.g. una azione) e degli stati che rappresenteranno i vari attori. In figura 1.10 un esempio del diagramma RAD in cui il requisito espresso va a definire i ruoli degli attori buyer, broker, seller.

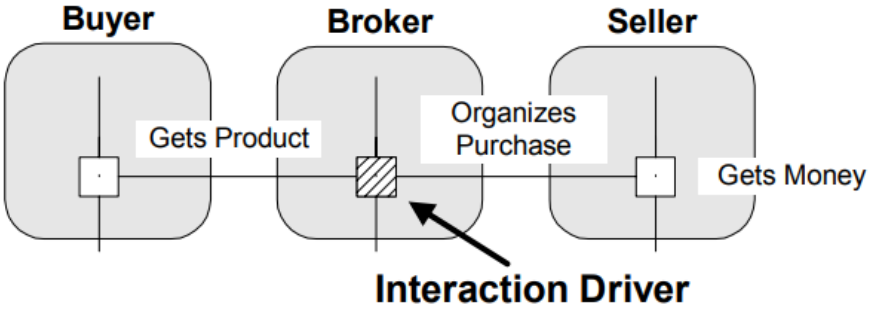

Figura 1.10: Esempio Role Activity Diagrams

Gap Analysis Una gap analysis è uno studio formale di ciò che un'organizzazione sta attualmente facendo, dove vuole andare e come è possibile colmare il divario fra queste due cose. Consente quindi di capire le misure da adottare per garantire che tutti i requisiti aziendali vengano soddisfatti.

Brainstorming Come nella fase di raccolta, anche nell'analisi dei requisiti può essere opportuno svolgere delle riunioni in cui confrontarsi con i diversi stakeholder per la definizione dei criteri di rappresentazione dei requisiti e scegliere la metodologia più adatta al singolo problema.

1.5.2 Determinazione delle componenti (Determine Components)

Scopo di questa fase è quella di definire i componenti che formeranno il prodotto finito. Partendo dall'analisi dei requisiti fatta precedentemente si devono individuare gli attori, ovvero quelle entità software e hardware, che collaborano durante l'esecuzione.

Un attore è un elemento del modello che interagisce con un sistema. Un elemento del modello, può essere una persona astratta (es. "cliente") o un altro sistema esterno. Nella pratica della modellazione si raccomanda di definire il ruolo in modo univoco con stereotipi - ad es. "Utente", "Stakeholder" o "Sistema esterno".

In questa fase è possibile utilizzare dei diagrammi di tipo Use Cases formulati in SysML (System Modeling Language), vedi Fig. 1.11, in modo da rendere più

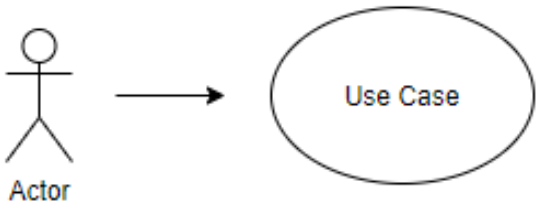

Figura 1.11: Esempio di SysML - Actor

evidenti chi sono gli attori e come essi vengono collocati all'interno dei singoli casi d'uso.

Una corretta individuazione degli attori significa trovare il numero giusto di attori: un numero troppo elevato potrebbe portare ad un "passaggio di responsabilità" troppo pronunciato. In questo caso si creerebbe una catena di deleghe in cui ogni componente lascia la responsabilità al componente a valle perdendo dunque il senso logico dell'azione comandata. In Fig. 1.12 si vede come i due attori centrali sono superflui a patto che l'attore all'estrema sinistra possa logicamente compiere l'azione nei confronti dell'attore più a destra con l'azione individuata dalla freccia tratteggiata.

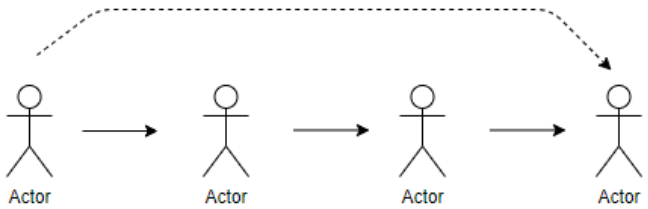

Figura 1.12: Responsabilità tra attori

D'altro canto, individuare un numero di attori non sufficiente rende poco espressivo il funzionamento del sistema in quanto si rischia di attribuire delle azioni a degli attori che logicamente non possono compiere l'azione stessa.

1.5.3 Struttura del modello (Model System Layout)

Dopo aver definito quali sono gli attori, ovvero i componenti del sistema, è necessario creare dei legami tra di essi. Come nella fase precedente è utile utilizzare un linguaggio di tipo SysML per la schematizzazione del sistema nel complesso. Questa fase ha lo scopo di creare legami tra gli attori individuati in precedenza, in particolare questi legami rappresentano ciò che essi devono compiere per adempiere ai requisiti imposti dal cliente.

1.5.4 Modellazione delle componenti (Model Components)

La fase di modellazione dei componenti implica che essi siano stati collocati in modo corretto all'interno del sistema complessivo, definito nel passo precedente. In questo stato si considera ogni singolo componente e se ne modella il funzionamento intrinseco. Rappresenta una delle fasi più delicate di tutto il processo, poiché racchiude il concetto di Model Based Design, ovvero partendo da una analisi dei principi fisici alla base del problema, se ne ricostruisce un modello matematico equivalente in grado di simulare il funzionamento reale. Se il modello non rispecchia a pieno la realtà, si ottiene un prodotto finito non coerente con i risultati attesi. In Fig. 1.13 è riportato un esempio di un modello dinamico di un sistema con doppia massa, molla e smorzatore realizzato utilizzando i software della MathWorks, in particolare Matlab e Simulink. Per realizzare il modello si sono, in una prima fase, ricavate le equazioni fisiche alla base del sistema reale per poi implementarle tramite l'utilizzo di uno schema a blocchi all'interno dell'interfaccia di Simulink. Nella parte destra della Fig. 1.13 è stata realizzata una versione equivalente utilizzando le librerie di Simscape, presenti dentro il software Simulink, le quali permettono la modellazione di sistemi reali non partendo dalle equazioni differenziali, bensì da componenti fisici direttamente presenti all'interno delle librerie stesse.

Considerando lo schema a V principale in Fig. 1.8, si nota come la fase di modellazione dei singoli componenti sia collegata anche ad alla fase di test dei singoli componenti. Un vantaggio del MBD è proprio che ogni singolo componente, in generale, è un'entità auto-contenuta ed indipendente per cui è possibile verificare il suo funzionamento indipendentemente dalla realizzazione dei componenti a monte e a valle del componente in esame. Attuando una metodologia del tipo: sviluppo di un singolo componente e test dello stesso, si limita il fenomeno di propagazione degli errori.

1.5.5 Analisi del modello (Analyze Model)

Una volta che i singoli componenti sono stati modellati nel dettaglio e ne sono state testate le funzionalità, si procede con una prima fase di analisi, ovvero di verifica del funzionamento complessivo. L'operazione di analisi ha lo scopo di evidenziare eventuali lacune o ridondanze nel modello, infatti la fase a valle di quella in esame è la definizione e modellazione di eventuali nuovi componenti.

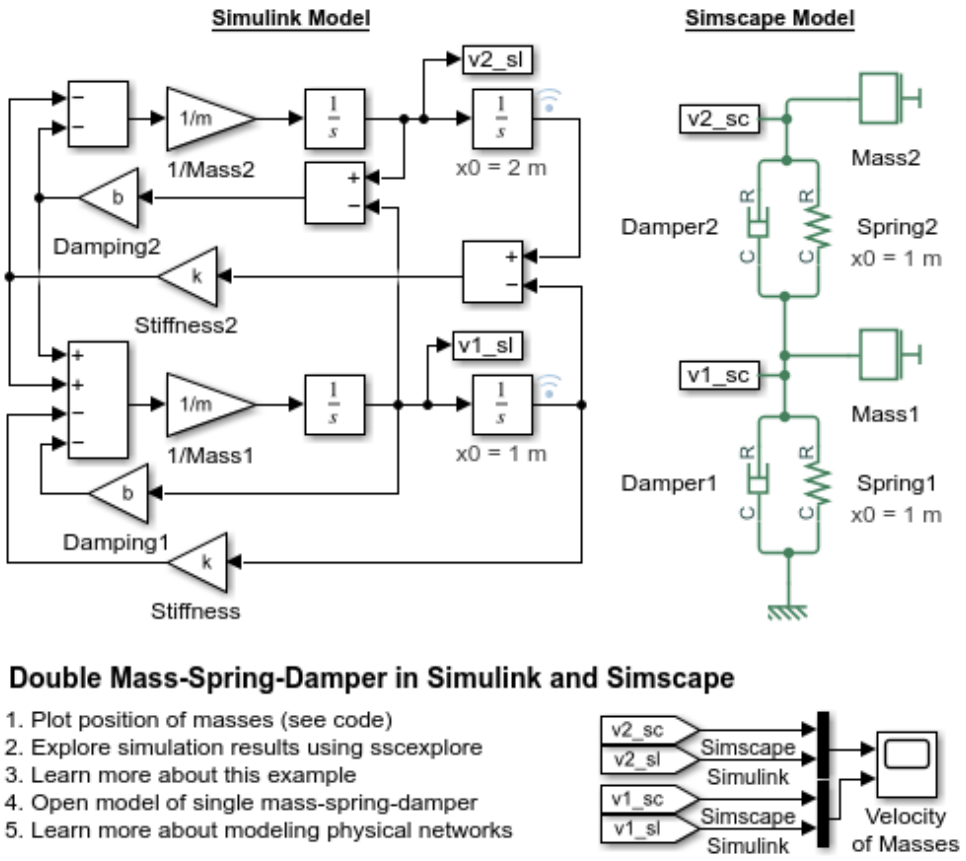

Figura 1.13: Esempio di un modello in Simulink

1.5.6 Progettazione di nuovi componenti (Design New Components)

Analoga alla fase Model Components, si definiscono e si modellano eventuali nuovi componenti risultati mancanti nella fase di analisi. Ovviamente lo schema si sviluppo a V non è unidirezionale: c'è sempre la possibilità di ritornare nelle fasi precedenti per rivedere eventuali scelte e correggere il comportamento del sistema complessivo.

1.5.7 Fase di test delle componenti (Test System Components)

Nel momento in cui la fase di analisi ha esito positivo, è necessaria una fase di test. I test possono essere formulati sia in fase di analisi dei requisiti, sia pensati ad-hoc

per le singole parti del modello. In Fig. 1.14 è riportato un esempio di un modello implementato in Simulink, in cui sono presenti dei blocchi in grado di compiere dei test sul sistema e verificare se questi sono stati passati con esito positivo o meno. Nell'ottica MBD, la fase di test, Fig. 1.3, è estesa lungo tutto il processo di sviluppo e non solo alla fine della progettazione dei singoli componenti. Ciò consente di rilevare gli errori già dalla fase di progettazione, prima ancora di aver implementato il sistema.

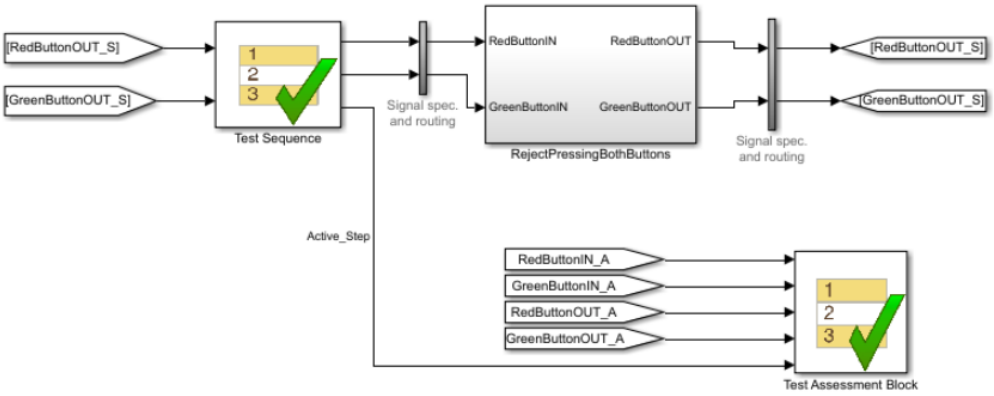

Figura 1.14: Esempio di test in Simulink

1.5.8 Integrazione delle componenti (Integrate Components)

In questa fase si integrano le varie parti del modello che sono state realizzate e si verifica la loro corretta interazione. Come in ogni fase è opportuno verificare, prima di procedere, che ciò che è stato realizzato sia coerente con i requisiti imposti a monte e che l'integrazione dei componenti non aggiunga funzionalità non richieste.

1.5.9 Fase di test del modello (Test System Model)

Questa fase ha lo scopo di simulare i componenti modellati per verificare il corretto funzionamento nel mondo reale. Molto spesso sì è obbligati a procedere per gradi, ad esempio non avendo a disposizione l'hardware, ovvero è necessario crearsi degli ambienti di simulazione ad-hoc.
Di seguito verranno illustrate le tre metodologie più diffuse per consentire la creazione di ambienti di simulazione, [6].

Model-in-the-loop

In questa configurazione, sia il modello di sistema che il controller girano nello stesso ambiente virtuale, con l'obiettivo di validare il controllo a ciclo chiuso proposto. Generalmente è il primo approccio di test che viene utilizzato, in quanto permette di verificare le funzionalità del controllore senza che questa sia "tradotto" in codice eseguibile. Questa metodologia è particolarmente utilizzata nel settore automotive poiché molte volte non è disponibile il veicolo fisico per la progettazione di una singola centralina, per cui è necessario simulare il funzionamento delle restanti centraline per testare quella in fase di realizzazione.

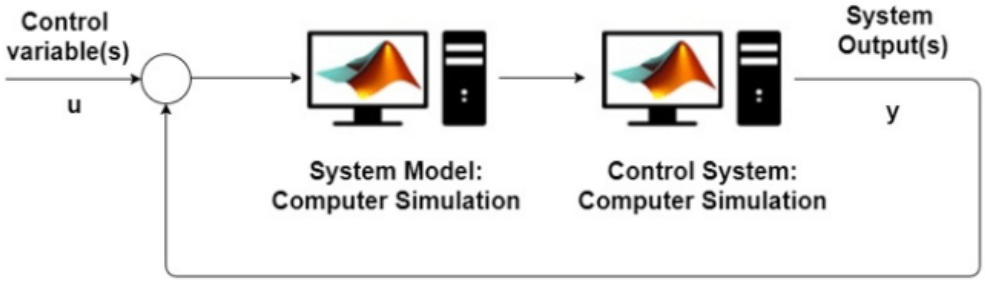

Figura 1.15: Model-in-the-loop

Software-in-the-loop

In questa configurazione, il controller è scritto o generato automaticamente nel software, viene posto esecuzione nello stesso ambiente virtuale del modello di sistema per convalidare l'algoritmo di controllo che sarà incorporato nell'hardware. A differenza del MIL, il controllore è sotto forma di codice eseguibile. Questo, se validato, sarà lo stesso codice che verrà caricato sulla centralina fisica. Con questa metodologia di simulazione riusciamo a verificare se il modello implementato a blocchi ha una corrispondenza univoca con il codice eseguibile.

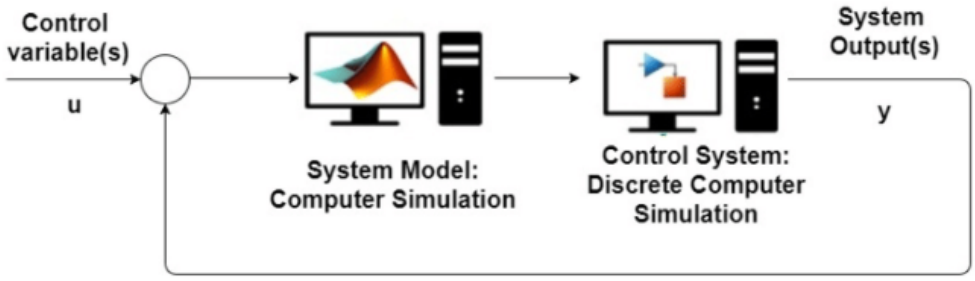

Figura 1.16: Software-in-the-loop

Hardware-in-the-loop

Se i risultati di MIL e SIL concordano tra loro, la fase successiva corrisponde al test HIL o Hardware-in-the-loop. Nella fase HIL, il codice del controller è incorporato nell'hardware e il modello del sistema è simulato in alcuni dispositivi in grado di funzionare in tempo reale, con ingressi e uscite simulati, con l'obiettivo di riprodurre il comportamento più vicino possibile al sistema reale.

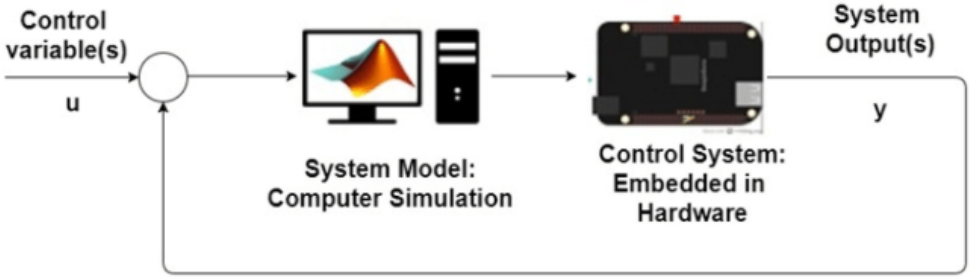

Figura 1.17: Hardware-in-the-loop

Una variante intermedia tra SIL e HIL è il Processor-in-the-loop (PIL). In questo caso, il codice del controller sviluppato utilizzando SIL è incorporato in un dispositivo hardware esterno (non necessariamente l'hardware finale per l'applicazione), mentre il modello di sistema rimane simulato in un ambiente virtuale (non necessariamente in tempo reale).

1.6 Funzionalità di MathWorks

In questa sezione verranno analizzati, in breve, i tool che MathWorks mette a disposizione per poter seguire un processo di sviluppo basato sulla metodologia Model Based Design. I tool proposti seguono l'ordine logico riportato nel work flow in Fig. 1.8.

1.6.1 Simulink Requirements

Consente di creare un set di requisiti, modificarli e gestirli all'interno di Simulink, così da poterli avere all'interno dello stesso ambiente di sviluppo, questo processo prende il nome di *authoring*. Inoltre, come si nota nella parte sinistra in Fig. 1.18, proprio per la caratteristica di condivisione dell'ambiente di esecuzione, è possibile associare ad ogni elemento di Simulink un requisito. Questo creerà un legame che consentirà di risalire al requisito che ha influenzato una determinata scelta di modellazione.
I requisiti, possono essere importati ed esportati nell'estensione standard: *ReqIF*,

acronimo Requirements Interchange Format, uno standard XML per l'interscambio di requisiti. Nella parte destra della Fig. 1.18 è presente lo stato di implementazione

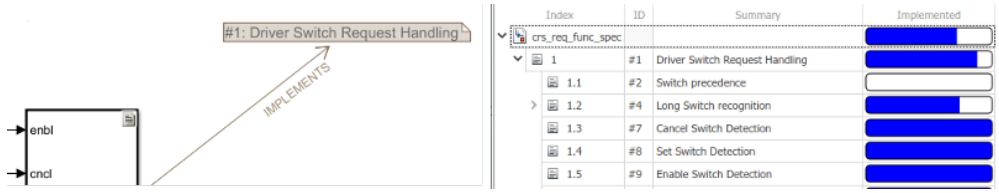

Figura 1.18: Simulink Requirements

del singolo requisito. Nel momento in cui associamo uno specifico requisito ad un elemento di Simulink esso risulta essere implementato. Inoltre, i requisiti possono essere espressi in modo gerarchico così da renderli più interpretabili.
Altra caratteristica di Simulink Requirement è la possibilità di creare dei test cases ad-hoc per verificare uno o più requisiti. Se il test dovesse ottenere esito positivo, il requisito corrispondente non solo sarà implementato, ma pure verificato. Questa caratteristica è utile poiché durante la fase di analisi dei requisiti, quando ancora il modello non è stato progettato, è possibile associare fin da subito dei test specializzati.

1.6.2 Matlab/Simulink/Stateflow

Matlab e Simulink sono i prodotti principali di MathWorks, il primo è un ambiente per il calcolo numerico e l'analisi statistica scritto in C, che comprende anche l'omonimo linguaggio di programmazione creato dalla MathWorks. Matlab consente di manipolare matrici, visualizzare funzioni e dati, implementare algoritmi, creare interfacce utente, e interfacciarsi con altri programmi. Simulink è un software per la modellazione, simulazione e analisi di sistemi dinamici, strettamente integrato con Matlab.

```
% Simple script in Matlab

M = [1 2 3; 4 5 6];
N = 2;

res = M.^N;
```

Figura 1.19: Script di Matlab

In Fig. 1.19 è riportato un semplice script che esegue la seguente operazione tra

matrici:

$$M = \begin{bmatrix} 1^2 & 2^2 & 3^2 \\ 4^2 & 5^2 & 6^2 \end{bmatrix} \tag{1.1}$$

In Fig. 1.20 è riportato un esempio di utilizzo di Simulink: un blocco che genera in output una funzione sinusoidale, la quale viene moltiplicata per due. All'output viene poi applicato un limite superiore e inferiore tramite un blocco che implementa una saturazione. Il vantaggio di utilizzare una tecnica di modellazione come quella implementata da Simulink, sta nella facilità di interpretazione, infatti lo stesso problema risolto da del codice sarebbe molto più complesso e meno intuibile.

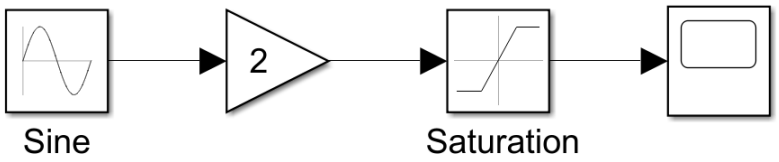

Figura 1.20: Esempio in Simulink

All'interno dell'ambiente Simulink, è possibile utilizzare dei blocchi ereditati da librerie le quali propongono soluzioni ai problemi più disparati.
Un'ulteriore feature che possiede Simulink è la perfetta integrazione con l'ambiente Matlab, infatti tramite dei blocchi ad-hoc chiamati *Matlab Function* è possibile integrare del codice scritto in Matlab all'interno di un modello rappresentato in Simulink.
Inoltre, grazie a Stateflow, Simulink permette la creazione di macchine a stati per lo sviluppo di applicazioni che ne richiedono l'uso. In Fig. 1.21 è stato utilizzato un blocco *State* il quale contiene una semplice macchina a stati per l'accensione di una lampadina. La macchina a stati, Fig. 1.22, viene realizzata nel canvas di Stateflow, che presenta un colore giallo chiaro a differenza del colore bianco dell'interfaccia di Simulink.

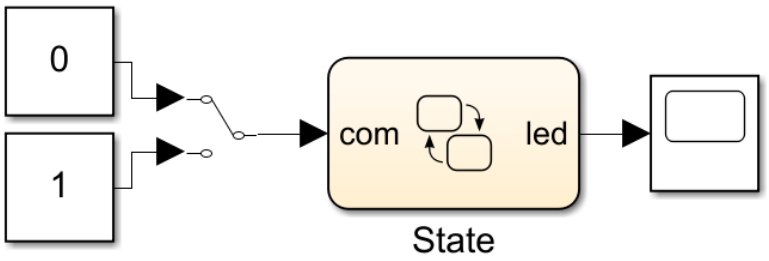

Figura 1.21: Simulink e Stateflow

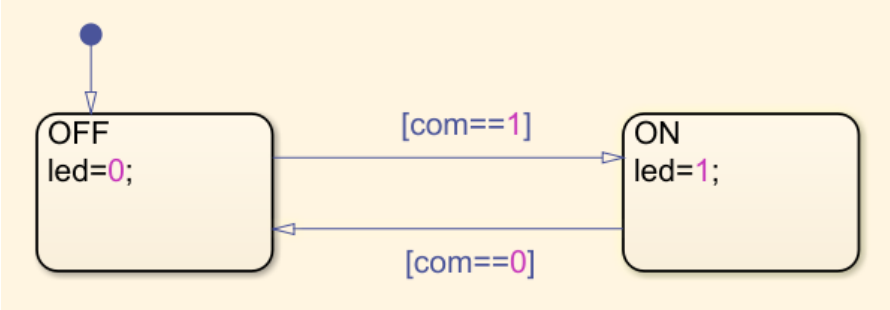

Figura 1.22: Stateflow

1.6.3 Simulink Coverage

Simulink Coverage è un tool che consente di eseguire l'analisi di copertura del modello e del codice autogenerato da esso. In Fig. 1.23 è riportato un esempio di

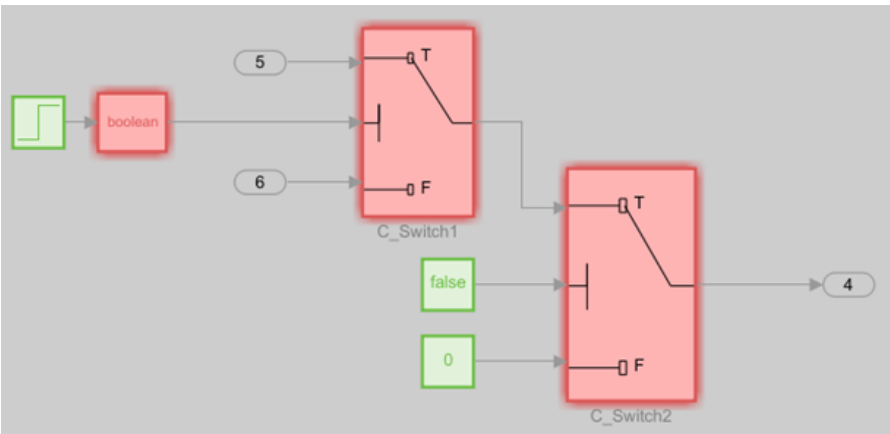

Figura 1.23: Simulink Coverage

analisi di copertura su un modello, le componenti in rosso non sono completamente coperte, mentre quelle in verde si.
Le tipologie di copertura analizzate dal tool sono molteplici, quelle più utilizzate sono le seguenti:

Execution Coverage (EC) è la forma più elementare di copertura. Per ogni elemento, EC determina se l'elemento viene eseguito durante la simulazione.

Decision Coverage (DC) analizza gli elementi che rappresentano i punti decisionali in un modello, come un blocco Switch o gli stati Stateflow. DC mira a garantire che ciascuna delle possibili diramazioni da ciascun punto di decisione venga eseguita almeno una volta e quindi garantendo che tutto il

codice raggiungibile venga eseguito. Cioè, ogni decisione viene presa in ogni modo, vero e falso. DC aiuta a convalidare tutti i rami nel codice assicurandosi che nessun ramo porti a un comportamento anomalo dell'applicazione.

Condition Coverage (CC) analizza i blocchi che emettono la combinazione logica degli loro ingressi (ad esempio, il blocco Operatore logico) e le transizioni Stateflow. Un test case ottiene una copertura completa CC quando fa sì che ogni input, a ciascuna istanza di un blocco logico nel modello, e ogni condizione su una transizione sia vera almeno una volta durante la simulazione e falsa almeno una volta durante la simulazione.

Modified Condition/Decision Coverage (MCDC) estende le capacità di copertura delle decisioni (DC) e delle condizioni (CC). MCMD analizza i blocchi che emettono la combinazione logica dei loro ingressi e le transizioni Stateflow per determinare la misura in cui il test case verifica l'indipendenza degli ingressi del blocco logico e delle condizioni di transizione.

Si hanno principalmente due casi in cui si ha piena copertura MCDC:

1. Un test case raggiunge la copertura completa per un blocco, quando una modifica in un ingresso, indipendentemente da qualsiasi altro ingresso, provoca una modifica nell'uscita del blocco.
2. Un test casa ottiene la copertura completa per una transizione Stateflow, quando c'è almeno una volta in cui un cambiamento nella condizione, attiva la transizione.

Simulink Coverage consente di creare dei report contenenti tutti i dettagli delle analisi svolte. Il tool può essere anche integrato all'interno di Simulink Requirement, Simulink Test e Simulink Design Verifier.

In Fig. 1.24, è riportato il flusso di lavoro da seguire per utilizzare al meglio i concetti di copertura del modello e del codice. Lo schema mostra come poter gestire una copertura non sufficiente rispetto a quella richiesta, proponendo delle soluzioni quali:

- Modifica il modello: il modello potrebbe contenere funzionalità indesiderate che non fanno parte del progetto richiesto. Rimuovere la funzionalità indesiderata.
- Modificare i requisiti: i requisiti potrebbero non dettagliare sufficientemente i test richiesti per mettere alla prova tutte le parti del modello.
- Creare test aggiuntivi: i test esistenti potrebbero non esercitare completamente gli input di simulazione previsti in base ai requisiti.

- Estendere i test esistenti: i test esistenti potrebbero non esercitare completamente tutte le parti del progetto. E' possibile utilizzare Simulink Design Verifier per generare automaticamente ulteriori test per esercitare parti non testate del tuo modello.
- Giustificare i risultati: parti del modello potrebbero non essere esercitate durante la simulazione, ad esempio i sottosistemi che si abilitano solo durante i guasti. Per ottenere una copertura completa in questi casi, è necessario giustificare la copertura mancante.

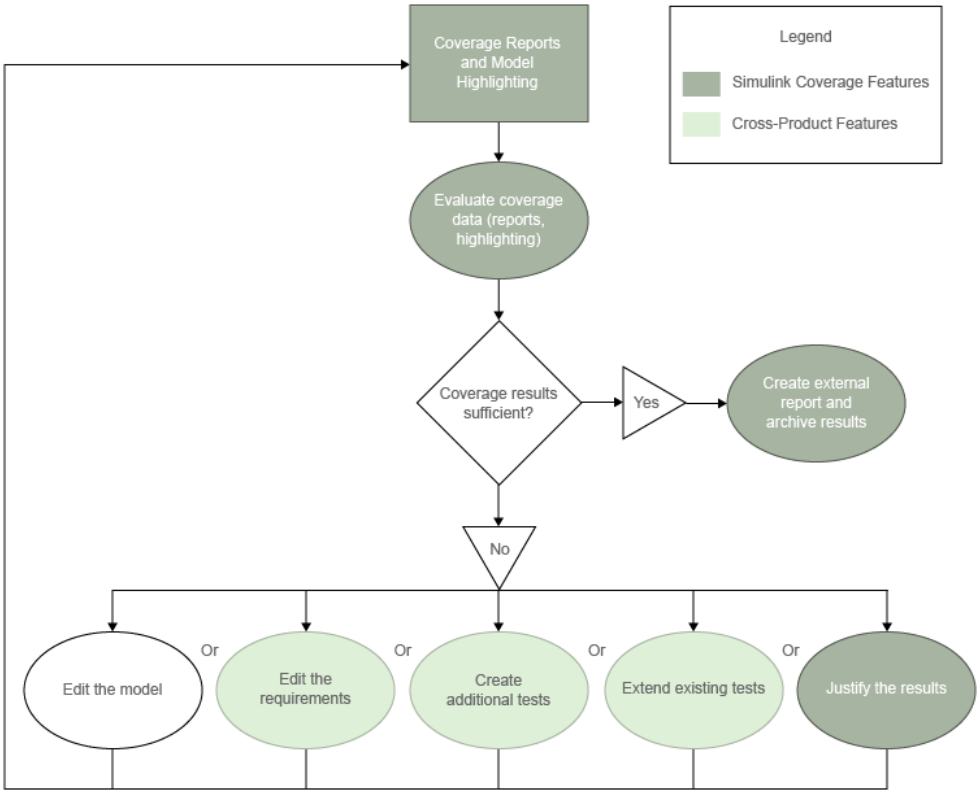

Figura 1.24: Simulink Coverage Workflow

1.6.4 Simulink Test

Simulink Test fornisce gli strumenti per la creazione, gestione ed esecuzione sistematica di test di modelli e codice generato. Con lo stesso tool è possibile implementare simulazioni di tipo software-in-the-loop (SIL), processor-in-the-loop (PIL) e hardware-in-the-loop (HIL).

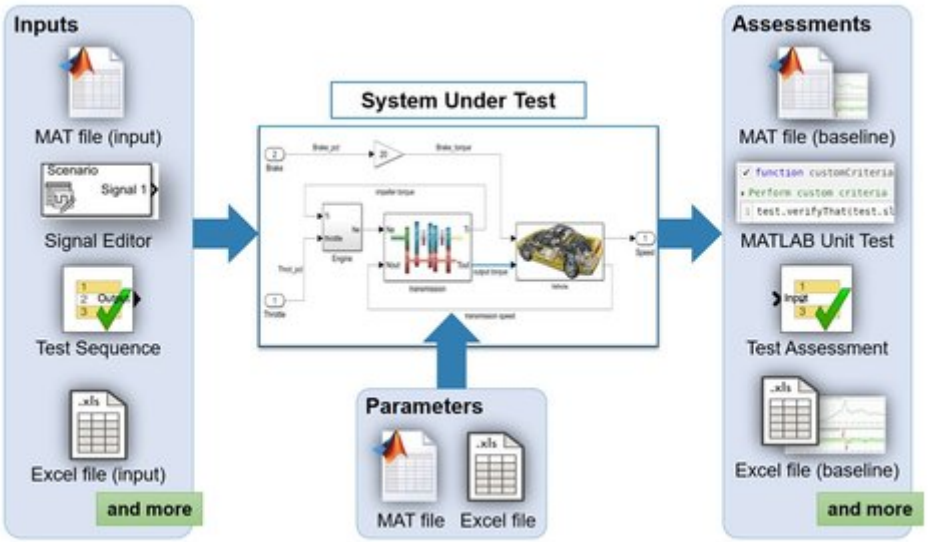

Figura 1.25: Simulink Test

Il funzionamento del tool è schematizzato in Fig. 1.25 in cui possiamo distinguere quattro componenti principali:

1. Input: Simulink Test è in grado di gestire simulazioni prendendo in input più tipologie di sorgenti come file di dati, blocchi Signal Builder e Test Sequence.

2. System Under Test (SUT): è il componente che si vuole testare. In Fig. 1.26 è visualizzato il concetto generale di *harness* specializzato nell'ambito del testing: il test harness non è altro che un ambiente isolato in cui i test vengono eseguiti e verificati da uno dei meccanismi proposti in Fig. 1.25, nel caso in esempio sono impiegati i blocchi *Test Sequence* e *Test Assessment*.

3. Parameter: sono dati necessari all'esecuzione del modello.

4. Assessment: corrisponde alla fase di valutazione del test, essa può essere implementata in diverse metodologie come l'impiego di blocchi di tipo *Test Assessment* o porzioni di codice Maltab.

Il modellatore ha a disposizione, diverse tipologie di test per la valutazione del modello creato. La scelta di una modalità rispetto ad un altra, viene generalmente specificata in fase di analisi dei requisiti in cui, associati ad essi, vengono definiti i relativi test. Nella Fig. 1.27 sono riportate le tre modalità:

Simulation Test è la metodologia più rapida per il test. Si basa sulla verifica dei risultati di una singola esecuzione, sulla base di criteri di accettazione definiti prima della simulazione stessa.

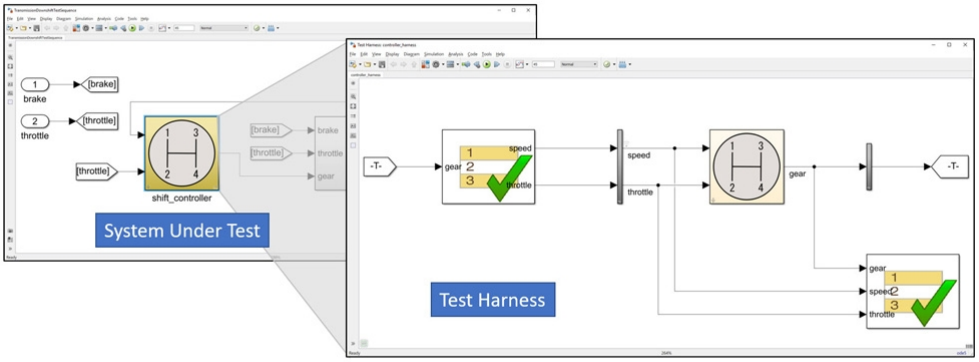

Figura 1.26: System Under Test

Baseline Test si confronta il risultato ottenuto da una particolare simulazione, con i risultati attesi forniti ad esempio da un file di dati. Il test ha esito positivo se la differenza tra i due è nulla o presenta una differenza tollerabile.

Equivalence Test come nella metodologia precedente si confrontano i risultati di due esecuzioni diverse. Viene utilizzato, ad esempio, per il test tra l'esecuzione di tipo MIL e SIL. Se il risultato tra le due forme di simulazioni è equivalente, l'esito del test sarà positivo.

1.6.5 Simulink Design Verifier

Simulink Design Verifier usa metodi formali per individuare errori di progettazione nascosti nei modelli. Per ciascun errore di progettazione o violazione dei requisiti, genera un test case di simulazione per il debug. Gli errori di progettazione, ad esempio, possono essere integer overflow, logiche morte, divisione per zero ecc. Per ogni condizione di errore estrapolata dal modello, il tool genera un test cases specializzato in grado di facilitare il debug e risolvere l'errore in considerazione.

Il tool può essere anche utilizzato per generare test case per ampliare i quelli esistenti basati sui requisiti. Questi test case stimolano il modello affinché soddisfi gli obiettivi di copertura definiti nelle sezioni precedenti. Oltre agli obiettivi di copertura, è possibile specificare obiettivi di test personalizzati per generare in modo automatico test case basati sui requisiti.

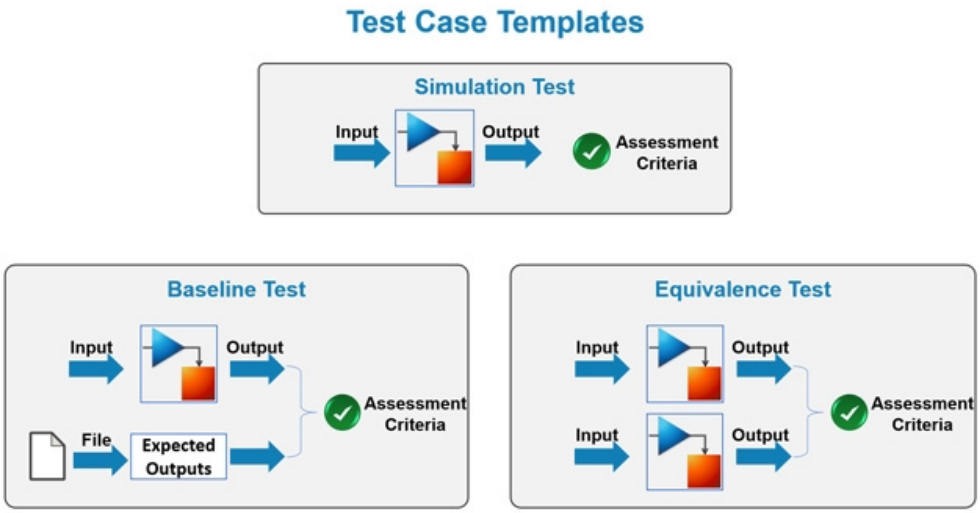

Figura 1.27: Tipologia di test

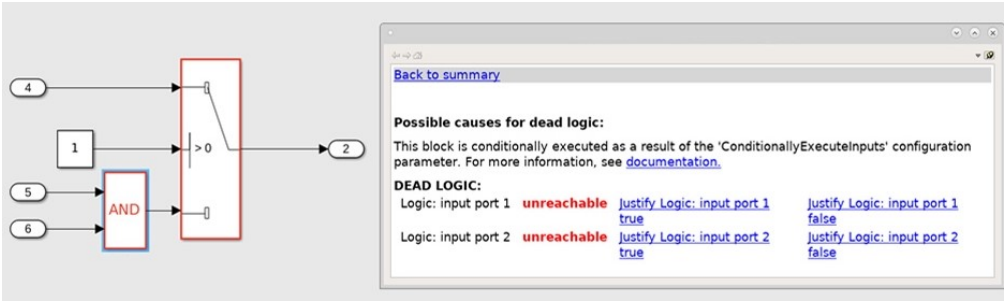

Figura 1.28: Simulink Design Verifier

In Fig. 1.28 è riportato un semplice diagramma a blocchi su cui è stato eseguito Simulink Design Verifier. Nel caso specifico è stata individuata una logica morta in quanto il ramo AND non viene mai eseguito. A destra è visualizzato il report che evidenzia il problema di non raggiungibilità, con lo scopo di aiutare il progettista a individuare e risolvere nel migliore dei modi la problematica.

1.6.6 Embedded Coder

Embedded Coder genera codice C e C++ a partire da modelli Simulink e codice Matlab. Il codice generato è ottimizzato per essere eseguito su dispositivi embedded. Il tool da la possibilità di personalizzare le caratteristiche del codice generato, come ad esempio la tipizzazione dei dati. Una volta che il codice è stato generato, è possibile, per ogni porzione di codice, risalire al blocco Simulink corrispondente,

facilitando la leggibilità. Con Embedded Coder possiamo utilizzare le modalità di test Software in the Loop (SIL) e Processor in the Loop (PIL) per provare il codice generato e confrontare i risultati ottenuti con quelli della simulazione effettuati da modello. Inoltre, il tool da la possibilità, sfruttando Simulink Check, di verificare se il codice generato rispetta le regole di buona programmazione, come le guidelines proposte da MISRA C. Al codice, utilizzando Simulink Coverage, può essere applicata l'analisi di copertura in tutte le modalità presentate nella sezione dedicata.

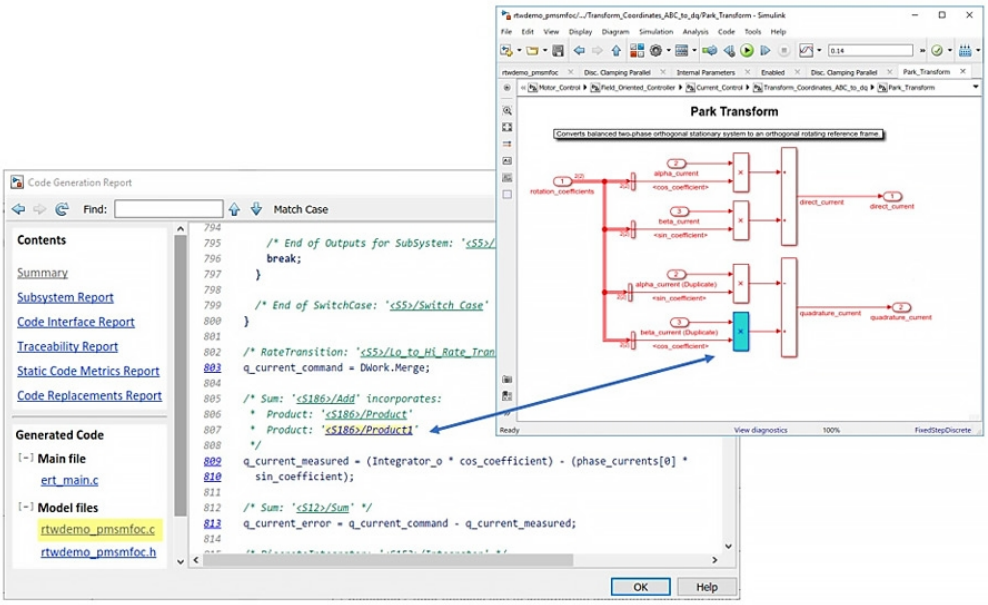

Figura 1.29: Tracciabilità tra codice e modello

Un'ulteriore feature che offre Embedded Coder è la possibilità di avere un collegamento tra il codice auto-generato e il modello (*traceability*). Questa caratteristica consente un'interpretazione più semplice del codice e permette, inoltre, di risolvere eventuali problemi in fase di debugging.

Capitolo 2

Caso di studio: On Board Charger

I motori, presenti all'interno dei veicoli elettrici, prelevano energia da delle batterie. Queste, come qualunque altra tipologia di accumulatore, devono essere ricaricate periodicamente per garantire il funzionamento del mezzo. In questo capitolo verrà fatta un'introduzione sui carica batterie posti a bordo del veicolo o On Board Charger (OBC), per poi applicare lo schema di sviluppo a V illustrato nei capitoli precedenti, con lo scopo di implementare, in un micro-controllore, la logica di supervisione per un OBC.

2.1 On Board Charger

L' On Board Charger, OBC, è un componente elettronico che viene posto a bordo del veicolo elettrico, con la funzione di ricaricare il pacco batterie del mezzo stesso. In Fig. 2.1 vengono schematizzate le due principali tipologie di carica che si hanno a disposizione. Le due si differenziano per la forma d'onda che la corrente assume. In presenza di un'adeguata corrente continua (Direct Current, DC), le batterie possono essere caricate direttamente dall'esterno senza che intervenga l'OBC. Questo perché il caricatore risulta essere esterno (off-charger) rispetto al mezzo. Generalmente i tempi di ricarica, sfruttando questa metodologia sono inferiori, in quanto si riescono ad applicare potenze maggiori, [7].

D'altro canto, se ad esempio si carica l'autovettura elettrica all'interno della propria abitazione, alimentata con una rete domestica in corrente alternata a 230V, è necessario utilizzare un caricatore in grado di convertire la corrente alternata (Alternative Current, AC), in corrente continua la quale andrà a ricaricare il pacco batteria. Quest'ultimo approccio prevede tempi di ricarica dilatati in quanto le potenze in gioco sono minori. In questa ultima strategia, se il caricatore è posto a

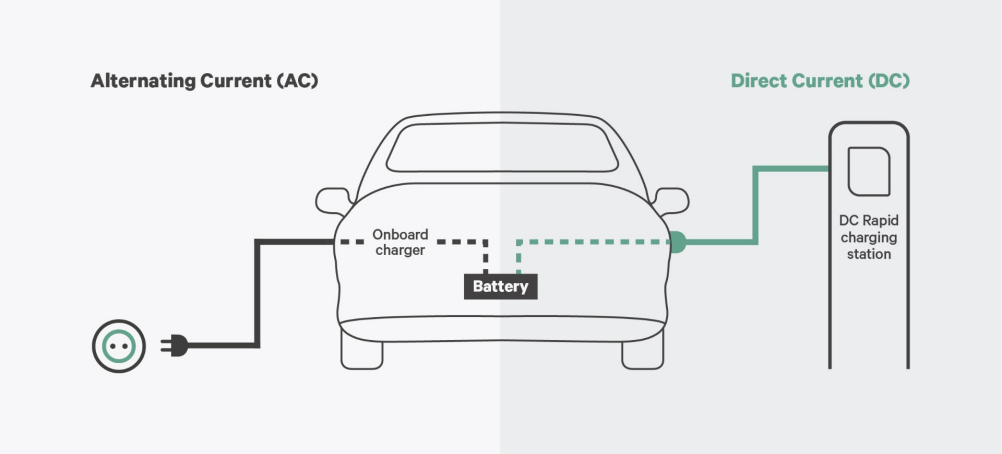

Figura 2.1: Tipologia di carica

bordo del veicolo, si parla di On Board Charger.
Nella pratica, l'OBC è un insieme di componenti elettronici, in Fig. 2.2 è raffigurato l'On Board Charger di una Tesla Model S, racchiusi all'interno di un case metallico.

Figura 2.2: On Board Charger di Tesla

2.1.1 Sistema di caricamento

Le parti che compongono un carica batterie *a due stadi*, vedi Appendice A, posto a bordo del veicolo sono schematizzate in Fig. 2.3. Gli autori, [8], dello schema prendono in considerazione un'alimentazione in corrente alternata AC, come quella

ad uso domestico. Analizzando lo schema proposto possiamo distinguere due macro componenti: *charging station* e *veicolo elettrico*.
Il primo rappresenta la colonnina di ricarica, che fornisce, tramite dei cavi specifici e standardizzati la sorgente di alimentazione AC al veicolo. Sempre nella charging station sono presenti dei moduli di controllo per garantire che la tensione e corrente in uscita siano coerenti con le forme d'onda richieste e, tramite un protocollo di comunicazione dedicato, la colonnina riesce a gestire eventuali problematiche che possono essere riscontrate durante il processo di ricarica, come ad esempio il raggiungimento della piena capacità o una disconnessione accidentale del cavo di ricarica.

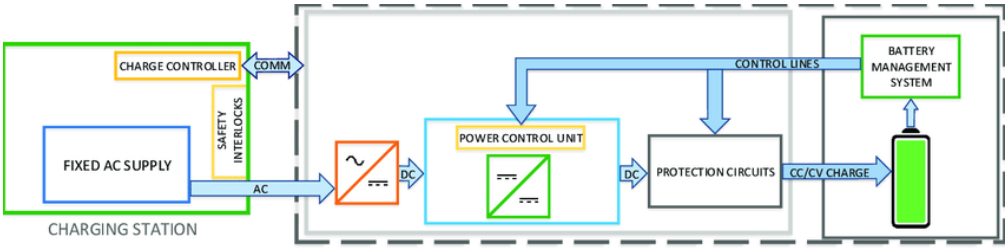

Figura 2.3: Schema a blocchi OBC

Il secondo macro-componente in Fig. 2.3 è il veicolo elettrico. All'interno di esso possiamo individuare il charger (OBC) e il sistema di gestione delle batterie o Battery Management System (BMS). Il charger, contornato di un grigio chiaro, contiene al suo interno più componenti elettronici nello specifico: convertitore da corrente alternata a corrente continua (e.g. raddrizzatore), convertitore da corrente continua a corrente continua (e.g. chopper) e una serie di circuiti di protezione per evitare danni causati da picchi non voluti. Un approfondimento teorico delle componenti che sono presenti all'interno di un OBC è discusso nell'appendice A. L'ultima componente all'interno del veicolo è il BMS. Questo racchiude al suo interno un modulo di gestione delle batterie e il pacco batterie stesso. Compito del BMS è, ad esempio, quello di determinate lo *State of Charge* (SOC) delle celle.

Uno schema più puntale e dettagliato di tutti i componenti è proposto da [9], in Fig. 2.4, in seguito discusso.

Lo schema a blocchi 2.4 vuole generalizzare la topologia di un generico On Board Charger, [9]. I componenti presenti in Fig. 2.3 vengono maggiormente dettagliati. Nello specifico si generalizza la provenienza della energia di alimentazione considerando che essa può derivare da fonti rinnovabili, dalla rete elettrica, da un meccanismo contactless di carica ecc. Inoltre, la parte del BMS viene espansa e vengono considerate le singole componenti del pacco batteria chiamate Smart Battery Module o SMB. Si nota il bus di comunicazione su cui scorreranno le informazioni riguardanti lo stato della carica, le tensioni e le correnti delle singole celle

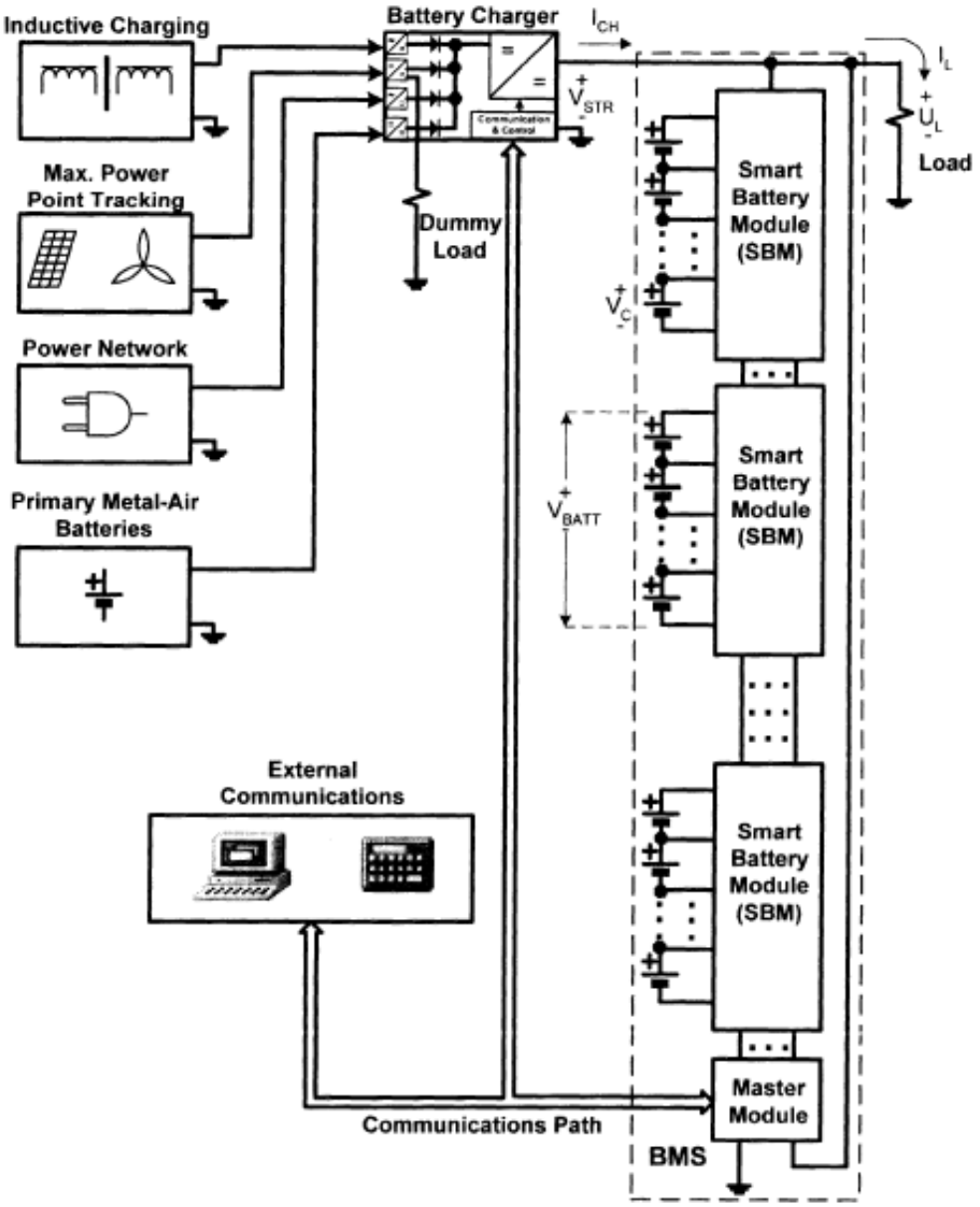

Figura 2.4: Schema completo ricarica

e tutti quei messaggi di gestione dell'intero processo di carica. Il Battery Charger o OBC prende in input una corrente da una qualunque sorgente precedentemente elencata e tramite una serie di componenti elettronici fornisce alimentazione in corrente continua (DC) alla componente di BMS, in figura viene isolata da una riga tratteggiata.

2.2 MBD applicato all'OBC

In questa sezione verrà seguita la metodologia del Model Based Design su ambiente MathWorks per l'implementazione della logica di controllo di un On Board Charger. La sezione sarà divisa in parti, ciascuna di essa rappresenta uno step fondamentale nel ciclo a V dello sviluppo software. L'ambiente di sviluppo sarà comune per tutte le fasi, tranne per la fase di test Hardware-in-the-Loop in cui verrà anche utilizzato l'IDE STM32CubeIDE per la comunicazione tra l'host e il micro-controllore.

2.2.1 System Requirements

Il progetto affrontato in questo lavoro di tesi, parte da alcuni requisiti industriali scritti all'interno di un file testuale. Il processo a monte dell'analisi degli stessi, ovvero tutte le metodologie presentate nel capitolo precedente per estrapolare, partendo dalle richieste del cliente, i requisiti stessi, non è stato trattato in questo elaborato.
Sfruttando il vantaggio del MBD, ovvero la condivisione degli ambienti di sviluppo, si è utilizzato il tool Simulink Requirements per importare i requisiti testuali di partenza, in un formato più strutturato ed interattivo presente all'interno del tool. In Fig. 2.5 è riportato un requisito specifico delle prime fasi del processo di interscambio di messaggi per avviare la fase di carica. Come si vede, è possibile integrare tabelle e altri oggetti grafici per aumentare la comprensione del requisito stesso. Il primo passo è quello di importare o riportare manualmente, i requisiti,

START CHARGING PROCESS

1. The Charger can read the plug in presence. When the plug is inserted, the Charger wakes up and sets **CHG_OUT.Proximity_detection = 1**;

2. The Charger sends to VCU a feedback where **CHG_OUT.Inlet_feedback = 0**;

3. If the VCU reads **CHG_OUT.Proximity_detection = 1**, the VCU sends the command to close the inlet to the Charger, that is **CHARGER _IN.Inlet_command = 2**;

4. If the Charger reads **CHARGER _IN.Inlet_command = 2**, the Charger will command the plug lock actuator with a 12V analog signal to lock the plug and set **CHG_OUT.Inlet_feedback = 1**;

NAME	TYPE (IN/OUT)
CHG_OUT.Proximity_detection	OUT
CHG_OUT.Inlet_feedback	OUT
CHARGER _IN.Inlet_command	IN

Figura 2.5: Requisito industriale

come quello in Fig. 2.5, all'interno del tool. Quest'ultimo consente di inserire al suo interno anche grafici, tabelle o immagini con lo scopo di descrivere con maggiore dettaglio ciò che si deve realizzare.

L'interfaccia che adotta Simulink Requirements è di facile utilizzo e comprensione: in Fig. 2.6 si nota come un insieme di requisiti o *Requirements Set* vanga indicizzato, caratterizzato da un custom ID, da una breve descrizione e da una barra di avanzamento denominata *Implemented*. E' possibile applicare una gerarchia ai requisiti in modo da dettagliare e scomporre caratteristiche richieste più complesse in parti più piccole e di facile implementazione. Un'ulteriore feature disponibile, già a livello di scrittura dei requisiti, è quella di associare a ciascun requisito un *test case* specifico con l'obiettivo di verificarne la corretta realizzazione. In quest'ultimo caso, a fianco alla barra di avanzamento *Implemented*, verrà visualizzata un'altra barra di avanzamento, chiamata *Verificated*.

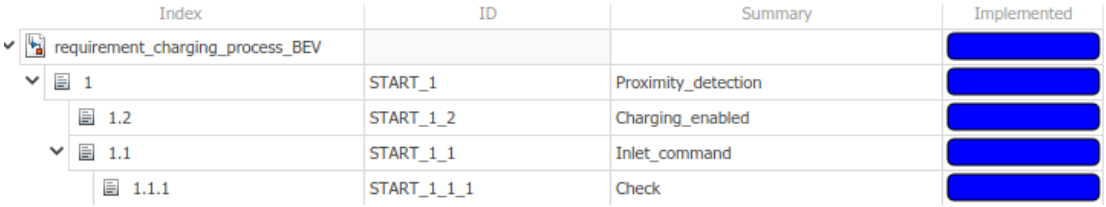

Figura 2.6: Gerarchia su Simulink Requirements

Espandendo un qualunque requisito all'interno della gerarchia in Fig. 2.6, verrà aperta una finestra con i dettagli relativi al requisito selezionato. In Fig. 2.7 è riportato il dettaglio di un requisito. Si nota come il livello di personalizzazione sia massimo: è possibile definire un ID personalizzato, un breve sommario e descrivere le richieste non solo con un formato testuale "di base", ma si possono utilizzare tutte le feature di un editor testuale, come Microsoft Word.

Una volta che i requisiti sono stati inseriti all'interno di un unico *Requirement Set* è possibile iniziare con la fase di design per poi modellare le varie richieste.

Durante la fase di modellazione, come riportato in Fig. 2.8, è di fondamentale importanza creare dei link tra il requisito che si sta implementando e la sua realizzazione. Come si vede, il requisito espresso in forma testuale, Fig. 2.5, è stato prima caricato sul tool di gestione, poi implementato su Stateflow e infine creato un link tra l'implementazione e il requisito. Questo meccanismo facilita sia la correzione di eventuali problemi, permettendo di risalire velocemente alle richieste del cliente, sia fornisce un ottimo strumento per misurare lo stato di avanzamento del progetto.

Caratteristica della metodologia Model Based Design è la possibilità di modificare in modo ciclico le caratteristiche del prodotto, ovvero aggiungere, togliere e

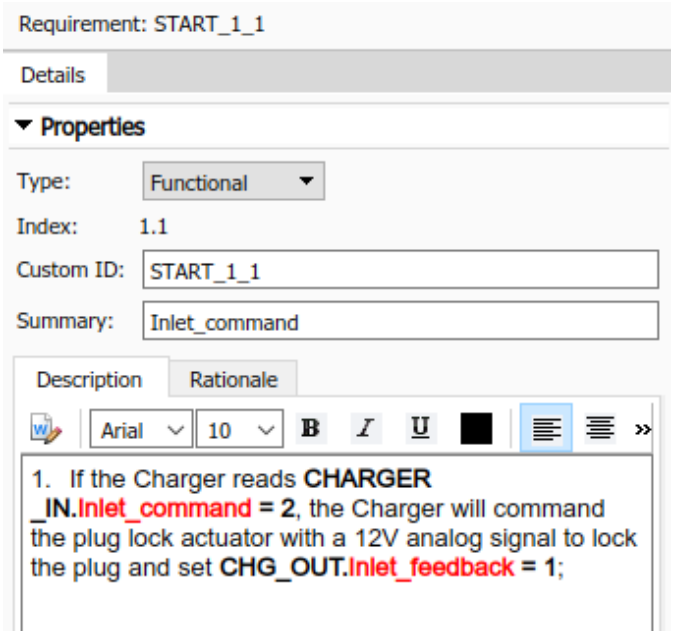

Figura 2.7: Rappresentazione su Simulink Requirements

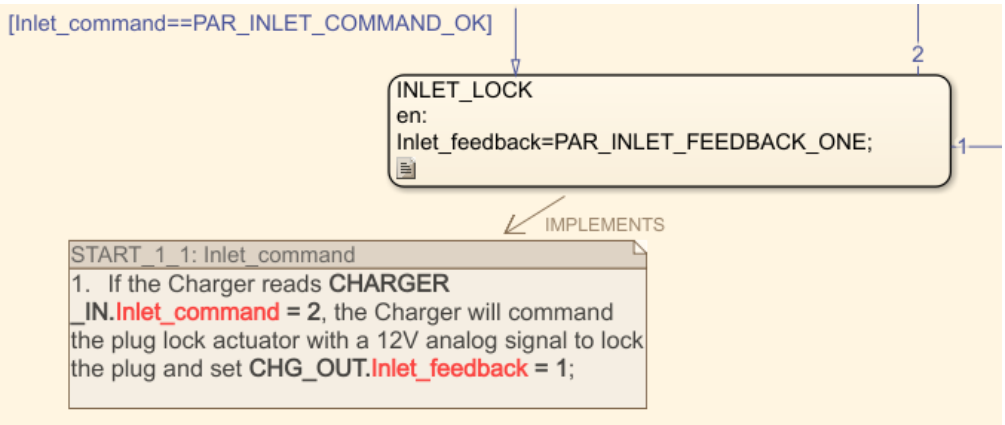

Figura 2.8: Link tra requisito e implementazione

modificare i requisiti cliente. Per questo motivo, una buona gestione dei requisiti evita la perdita di richieste durante lo sviluppo del progetto.

Essendo l'ambiente di sviluppo comune in tutte le fasi dello schema a V, Simulink Requirements è integrato con Simulink Check. Quest'ultimo è stato descritto nei capitoli precedenti. Nel caso particolare di verifica di regole relative ai requisiti, Fig. 2.9, è possibile verificare la consistenza dei requisiti. Nel modello realizzato, come si nota dalla Fig. 2.9, tutti i test sono stati passati con esito positivo. Ottenere dei

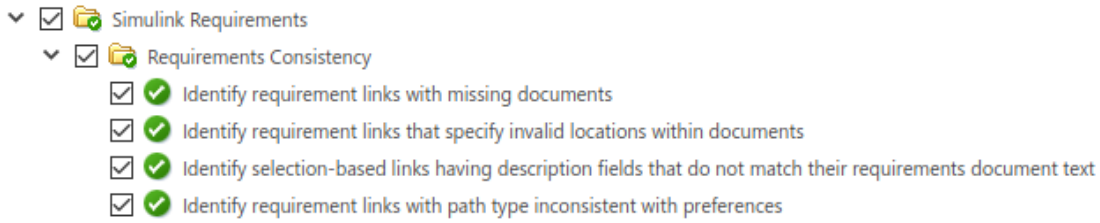

Figura 2.9: Model Advisor su Simulink Requirements

risultati positivi da Simulink Check è condizione necessaria per poter applicare dei test più rigorosi e formali come quelli per la verifica degli standard MISRA C, ISO 26262 ecc.

2.2.2 System Design: controller

Scopo di questa fase è la definizione delle componenti che formano il sistema complessivo. Nel caso in esame, i componenti da modellare sono solo 2. Il primo è il modulo di gestione della logica di supervisione, ovvero il componente principale di questo progetto.

Una seconda componente è il plant, che nel caso in esame sarà implementato tramite una curva di carica con la funzione di testare la logica. In Fig. 2.10 è

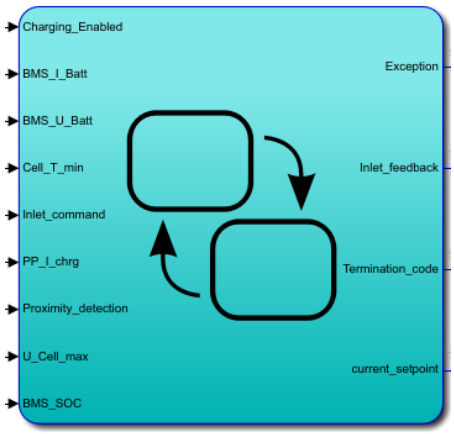

Figura 2.10: Supervisore logico - OBC

riportato il blocco di tipo *chart* che implementa la logica di supervisione tramite Stateflow. La scelta di utilizzare un tool che permettesse di implementare delle macchine a stati è stata facilitata dall'obiettivo di progetto: la realizzazione di un supervisore logico. Un'altra alternativa era quella di utilizzare dei blocchi "classici"

di Simulink, ottenendo però un modello più complesso da interpretare e più difficile da realizzare.

Come da requisiti, gli ingressi al modulo da progettare sono stati riportati nella tabella 2.1.

Nome	***Mittente***	***Descrizione***
Charging_Enabled	VCU	Flag di abilitazione alla carica
BMS_I_Batt	BMS	Corrente di batteria
BMS_U_Batt	BMS	Tensione di batteria
Cell_T_min	BMS	Temperatura delle celle
Inlet_command	VCU	Flag di bloccaccio spina esterna
PP_I_chrg	BMS	Corrente di riferimento per la carica
Proximity_detection	VCU	Flag di presenza spina
U_Cell_max	BMS	Tensione massima di cella
BMS_SOC	BMS	SOC di batteria

Tabella 2.1: Variabili in input

Gli output sono riportati in Tab. 2.2. In generale i segnali di Input/Output sono messaggi che viaggiano tra le centraline presenti all'interno del veicolo, sfruttando il protocollo CAN. Quest'ultimo non verrà approfondito in questo contesto, poiché nell'implementazione si è scelto di utilizzare un protocollo seriale.

Nome	***Destinazione***	***Descrizione***
Exception	VCU	Flag gestione eccezioni
Inlet_feedback	VCU	Flag di bloccaggio spina
Termination_code	VCU	Codice identificativo degli step di carica
current_setpoint	BMS	Corrente di carica delle celle

Tabella 2.2: Variabili in output

Il primo componente che è stato progettato è stata la macchina a stati, in Fig. 2.11. Essa contiene la logica di gestione del processo di carica. Come si nota, è stato utilizzato un linguaggio Matlab all'interno degli stati e delle transizioni. Le variabili in input e output sono state trattate come segnali separati e identificati da una porta. Nelle sezioni successive si commenteranno gli stati più importanti, per descrivere in modo più dettagliato il processo.

START_CHARGING

In Fig. 2.12 è riportato lo stato di inizio carica. In particolare a monte di esso c'è uno stato transiente, chiamato *INIT*, che rappresenta la situazione normale, ovvero

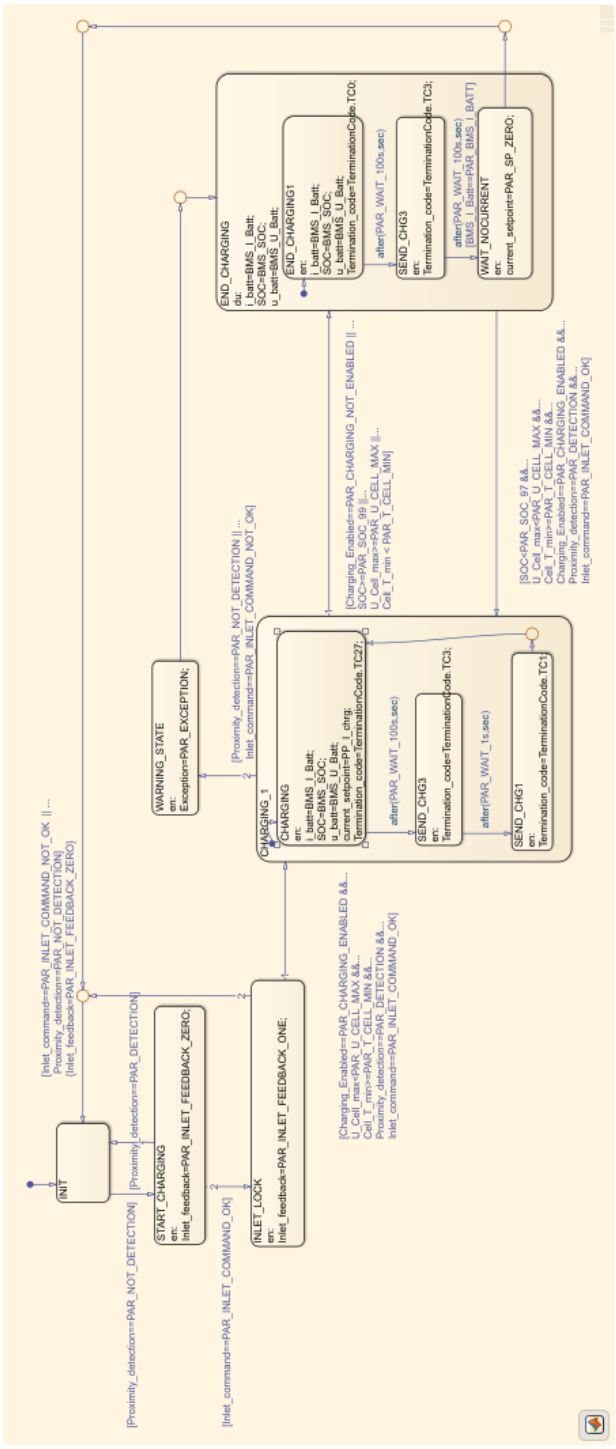

Figura 2.11: Logica completa OBC

quella in cui non si ha l'intenzione di procedere al caricamento del veicolo. La condizione per entrare nello stato in esame è che la spina deve essere stata rilevata dal sistema. Compito dello stato è quello di iniziare la procedura di bloccaggio della spina, inviando alla VCU il corrispettivo segnale. Se per un qualunque motivo, l'utente dovesse rimuovere la spina, prima che questa sia meccanicamente bloccata, il sistema ritorna nello stato transiente *INIT*.

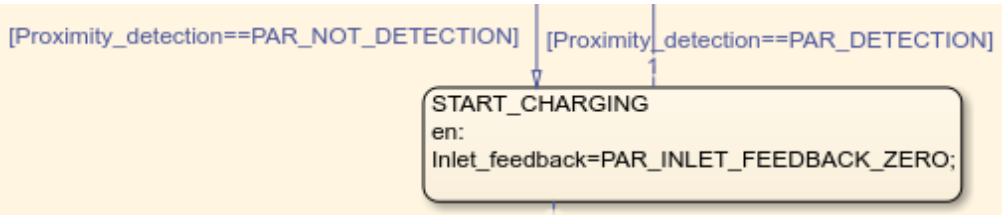

Figura 2.12: Stato: START_CHARGING

INLET_LOCK

Il sistema arriva nello stato di Fig. 2.13 quando la VCU ha ricevuto il segnale di bloccaggio e quando quest'ultimo è stato rimandato al mittente, ovvero il charger. In questo stato il charger conferma, tramite il settaggio della variabile *Inlet_feedback* l'effettivo bloccaggio della spina e il conseguente ingresso nello stato di carica, se le condizioni per poterlo eseguire sono verificate. In questo stato viene contemplato anche il caso in cui, nonostante la spina sia bloccata meccanicamente, l'utente riesca a disinserire il connettore stesso. In tale tale situazione il sistema torna nello stato iniziale senza aver iniziato il processo di carica.

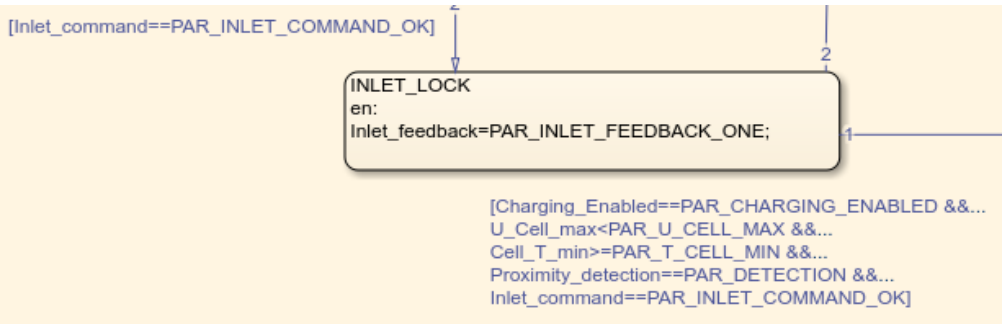

Figura 2.13: Stato: INLET_LOCK

CHARGING_1

In Fig. 2.14 è riportato uno degli stati principali dell'intero sistema di supervisione: quello di carica. Esso contiene al suo interno altri stati figli, con lo scopo di dettagliare il più possibile lo stato effettivo della macchina. Le condizioni per poter iniziare il processo di carica sono le seguenti:

1. Carica abilitata dalla VCU;
2. Tensione di cella minore di una soglia;
3. Temperatura di cella maggiore di una soglia;
4. Spina inserita, rilevamento della presenza;
5. Spina correttamente bloccata in modo meccanico.

Se le condizioni sopra dette sono tutte verificate è possibile iniziare il processo di ricarica. Al tempo stesso, se una di queste condizioni viene a mancare o se il SOC supera un determinato livello, si esce dallo stato di carica. L'uscita da questo stato può sia portare ad un processo di terminazione carica, nel caso in cui il SOC ha superato una soglia e dunque il caricamento è completato, oppure può portare in uno stato di allarme se ad esempio si rimuove in modo accidentale la spina avendo iniziato la carica. In questo ultimo caso è importante prevedere uno stato di allarme poiché, essendo iniziata la fase di carica, i componenti potrebbero condurre corrente pericolosa per gli umani.

All'interno del macrostato in Fig. 2.14 sono presenti tre stati figli:

CHARGING in questo stato vengono aggiornati i valori di corrente e tensione di cella e il corrispettivo SOC. Viene inoltrato alle altre centraline a bordo del veicolo, ed in particolare a BMS e VCU, il segnale *TerminationCode* che rappresenta lo stato di evoluzione del processo stesso. In particolare esso assume il valore 27 se è in corso la carica.

SEND_CHG3 si entra in questo stato dopo che sono trascorsi 100 secondi rispetto all'aggiornamento dei valori di carica. Qui si inviano alle altre centraline, un *TerminationCode* pari a 3.

SEND_CHG1 come nello stato precedente, dopo 1 secondo dall'invio del codice 3, si inoltra un *TerminationCode* pari a 1 per poi ritornare allo stato di aggiornamento dei valori di carica.

Durante l'esecuzione ciclica dei tre stati sopra descritti, è possibile uscire dal macro stato *CHARGING_1* interrompendo dunque il processo di carica. Inoltre, è possibile raggiungere lo stato di carica, non solo attraverso lo stato iniziale, ma

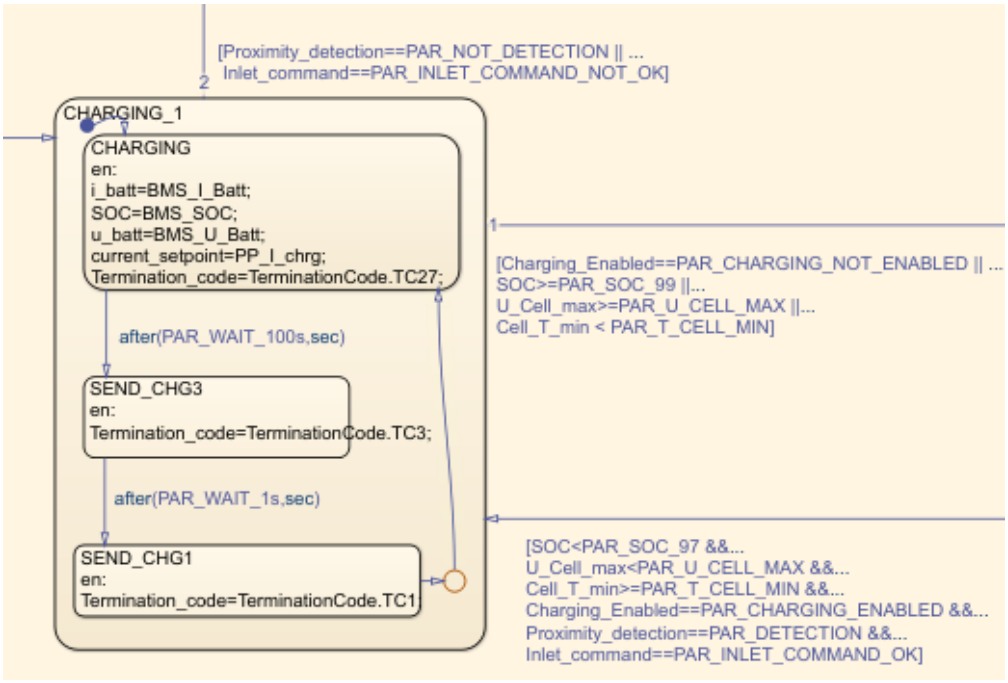

Figura 2.14: Stato: CHARGING_1

anche durante il processo di terminazione carica, se ci si rende conto che il SOC è effettivamente minore di 97. In questo caso, se le normali condizioni per accedere al processo di carica sono soddisfatte, si ricomincia il caricamento per ottenere il SOC massimo.

WARNING_STATE

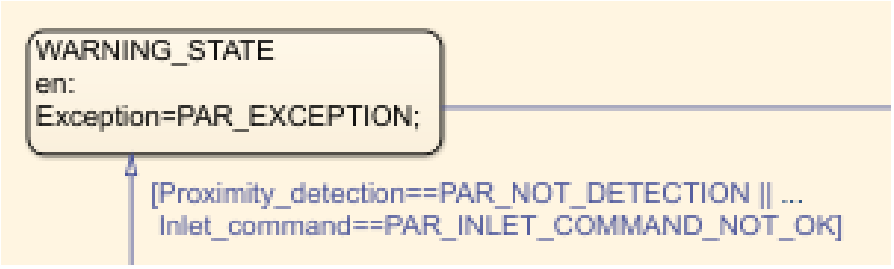

Figura 2.15: Stato: WARNING_STATE

Come detto precedentemente, se durante il processo di carica, dovesse esserci uno scollegamento forzato della spina nonostante il blocco meccanico, è necessario generare una condizione di allarme e di conseguenza progettare uno stato di

attenzione, come quello in Fig. 2.15. In questo stato non si fa altro che inviare alle centraline BMS e VCU il segnale di eccezione il quale verrà gestito al di fuori di questa logica, con ad esempio l'accensione di led di allerta ecc. L'uscita da questo stato è automatica e non presenta condizioni. Lo stato successivo a quello descritto è quello di terminazione del processo di carica.

END_CHARGING

In Fig. 2.16 è riportato il secondo stato più importante di tutta la logica di supervisione: il processo di fine carica. L'ingresso in questo stato può avvenire in due metodologie:

1. Se si proviene da uno stato di allarme;
2. Se la batteria ha raggiunto un SOC accettabile o non sono più verificate le condizioni necessarie alla carica.

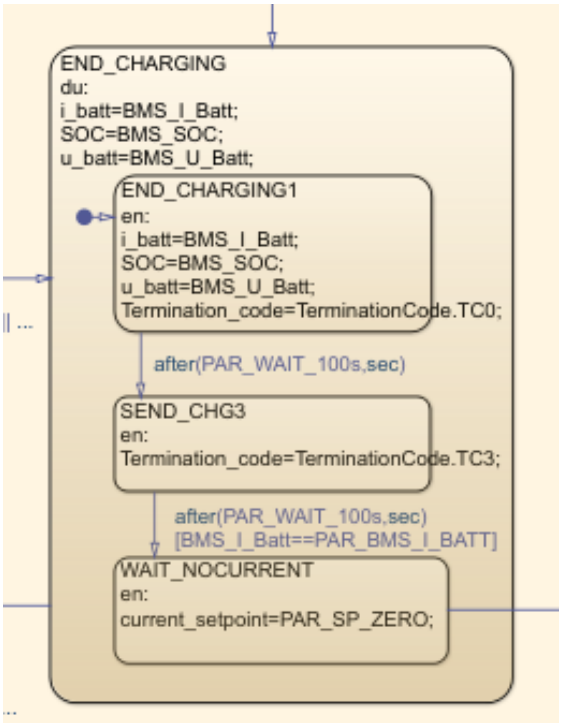

Figura 2.16: Stato: END_CHARGING

Entrati nello stato di fine carica, vengono aggiornati i valori di tensione e corrente delle celle per poi iniziare una serie di scambi messaggi tra BMS e charger. Come nello stato di caricamento, anche questo è formato da tre stati figli:

END_CHARGING1 in questo stato si aggiornano i valori caratteristici delle celle e si invia *TerminationCode* a 0, come ad indicare l'inizio del processo di fine carica;

SEND_CHG3 dopo 100 secondi dall'invio di *TerminationCode* a 0, si aggiorna il valore dello stesso segnale a 3;

WAIT_NOCURRENT trascorsi 100 secondi e verificato che la corrente di batteria sia nulla, si procede a impostare il valore di riferimento di carica batteria pari a zero così da permettere lo sbloccaggio meccanico della spina e il conseguente ritorno nello stato transiente *INIT*.

2.2.3 System Design: modello del plant

Il secondo componente che è stato progettato è un plant che simulasse una curva di carico. In Fig. 2.17 è implementata la gestione dell'andamento dello State of Charge. Lo schema a blocchi prende in input il *TerminationCode*, TC, e un SOC iniziale, ovvero il valore percentuale che la batteria possiede nell'istante in cui si vuole iniziare il processo di ricarica. Output dello schema è il SOC che il pacco batterie possiede.

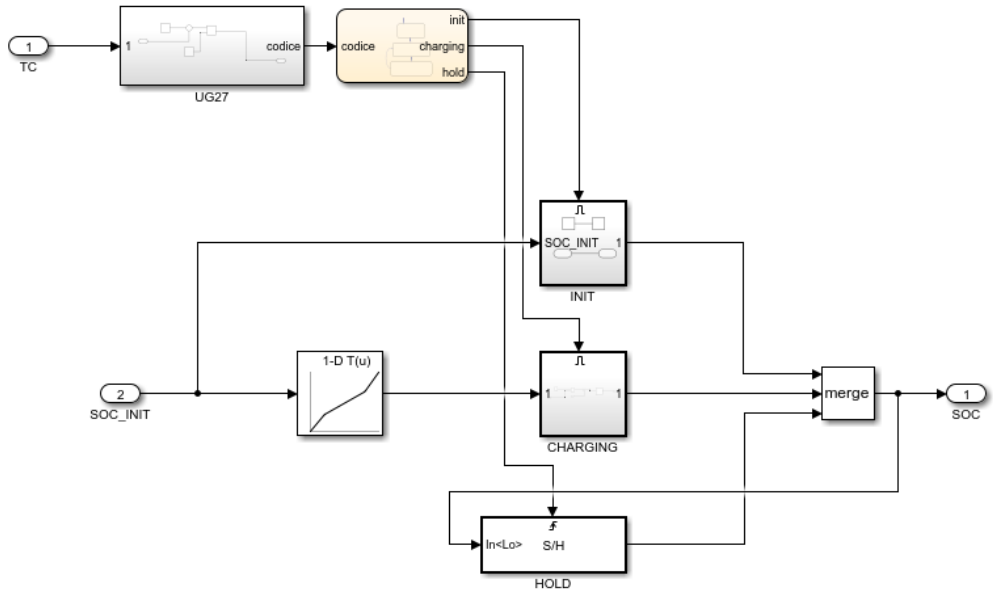

Figura 2.17: Plant: curva di carica

Il principio di funzionamento dello schema in Fig. 2.17 è il seguente: *partendo da un valore randomico di SOC, se TC è pari a 27, il SOC deve aumentare seguendo il profilo di carica presente all'interno di un look up table, se per un qualunque*

motivo TC è diverso da 27, il SOC deve rimanere invariato al valore precedente per poi riprendere ad aumentare, secondo la curva di carica, se TC dovesse tornare a 27.

Il requisito sopra citato, risulta essere di difficile comprensione se espresso sotto forma di linguaggio naturale. Per maggior chiarezza, il requisito, è stato tradotto in un flow chart in modo da facilitarne la comprensione, vedi Fig. 2.18. Tale metodologia di gestione dei requisiti è stata descritta nei capitoli precedenti.

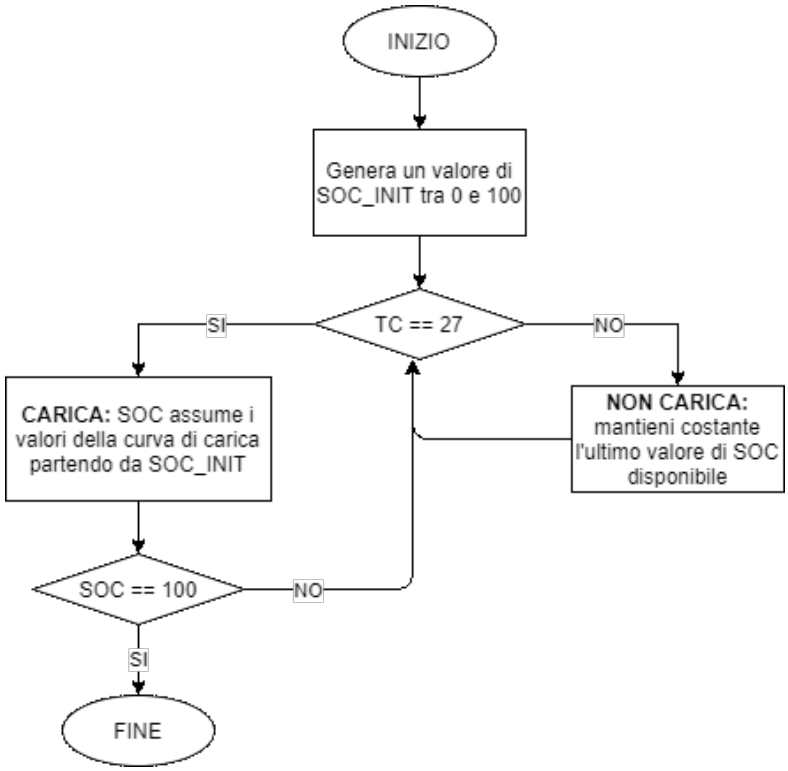

Figura 2.18: Requisito generazione SOC

Per poter implementare il requisito detto è stato necessario, in primo luogo, verificare il valore di *TerminationCode*, ciò è stato fatto con lo schema in Fig. 2.19 in cui si è adottato un approccio più elastico e adatto al fine della generazione di codice per piattaforme embedded: si definisce una soglia, nel caso in esame è stata posta pari a 0.0001, si opera una sottrazione tra il valore di riferimento (27) e il valore assunto della variabile. Se tale differenza è minore della soglia scelta allora si è verificata, con esito positivo, l'uguaglianza. Questo approccio consente di attribuire un esito positivo, rispetto all'operatore di uguaglianza, a variabili rappresentate su un numero finito di bit. Più in pratica, con questa trattazione, il valore 27.001 è uguale a 27, poiché abbiamo scelto una tolleranza di 0.0001.

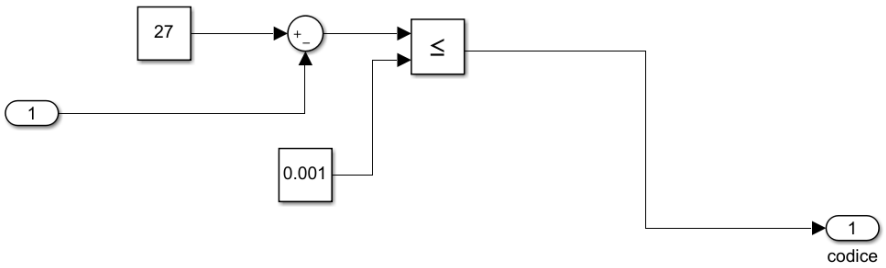

Figura 2.19: Verifica *TerminationCode*

Una volta che il *TerminationCode* è stato verificato, tramite la macchina a stati in Fig. 2.20 si determina la situazione, lo stato, in cui ci si trova:

INIT TC non ha mai assunto il valore 27, per cui si deve mantenere costante il valore iniziale;

HOLD se TC è diverso da 27, per cui non si è in fase di carica, si deve mantenere costante il valore attuale;

CHARGING se TC assume il valore 27, ovvero *codice == 1*, si entra nella fase di charging, per cui SOC deve seguire la curva di carico definita nella look up table.

La macchina a stati sopra descritta fornisce in output un ingresso di abilitazione a tre componenti dotati di un meccanismo di trigger. I primi due sono dei blocchi di tipo *If...Action* ovvero vengono eseguiti se l'ingresso di abilitazione viene attivato. L'ultimo componente è ereditato dalla libreria "DSP System Toolbox" e il suo funzionamento è il seguente: se l'ingresso di abilitazione viene attivato, mantiene costante il valore in input. Con questi tre blocchi si riescono a gestire i comportamenti descritti dalla macchina a stati.

Se la macchina a stati attiva il blocco di *CHARGING*, vedi Fig. 2.21, ciò che viene fornito in output è il corrispettivo valore di SOC associato al "tempo": viene utilizzato un integratore discreto per "contare" il tempo trascorso nello stato di charging, questo tempo viene sommato con il valore del "tempo" iniziale. Il valore del "tempo" iniziale viene ottenuto invertendo la curva presente all'interno di una look up table e passando a questa, come parametro in input, un SOC iniziale.

Finita la parte relativa alla modellazione, sono stati applicati, al modello, i tool descritti nei capitoli precedenti.

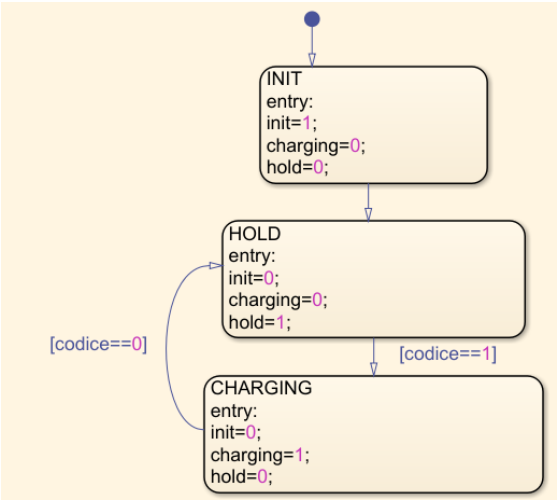

Figura 2.20: Gestione logica della ripresa di carica

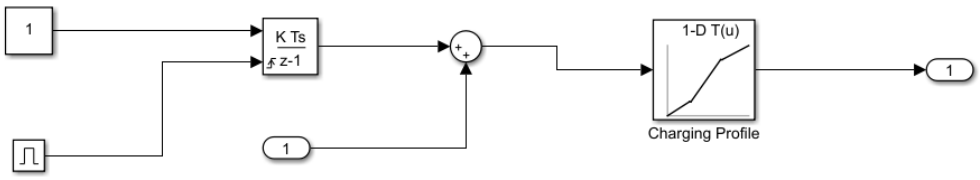

Figura 2.21: Generazione del valore di SOC

2.2.4 System Design: coverage della logica di supervisione

In Fig. 2.22 è stata riportato il report generato dal tool *Simulink Coverage*. Le tipologie di copertura che sono state utilizzate, per una valutazione delle prestazione del modello, sono Cyclomatic Complexity, Condition, Decision, MCDC. Le ultime tre sono state descritte teoricamente nei capitoli precedenti. Per quanto riguarda la Cyclomatic Complexity essa sta ad indicare il numero di cammini linearmente indipendenti attraverso il grafo di controllo del flusso, ovvero attraverso un grafo ottenibile considerando il flusso di informazioni del programma in cui gli archi sono le istruzioni e i nodi sono le decisioni, [10].

Analizzando i risultati ottenuti, notiamo che il modello è *full condition coverage* e *full MCMD*. Risultati comunque ottimi sono stati ottenuti per la decision coverage e per la complessità ciclomatica.

Child Systems: Chart

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	41
Condition	NA	100% (42/42) condition outcomes
Decision	NA	90% (38/42) decision outcomes
MCDC	NA	100% (21/21) conditions reversed the outcome

Figura 2.22: Copertura del modello

2.2.5 Software Design

Durante la modellazione, visto il fine del progetto, ovvero inserire il supervisore logico all'interno di un hardware, sono state utilizzate alcune feature che Matlab/Simulink mettono a disposizione e che offrono poi vantaggi nel momento in cui si genera codice automaticamente. Una di queste è l'utilizzo dei tipi di dato enumerato. Nella porzione di codice sotto riportata, è stata implementata una classe di tipo *IntEnumType* per la creazione del tipo enumerato. Questa tipologia di dati consente di risparmiare memoria quando si genera codice e facilita l'utilizzo e la lettura dello stesso.

```
classdef TerminationCode < Simulink.IntEnumType
    enumeration
        TC0(0)
        TC1(1)
        TC3(3)
        TC27(27)
    end
end
```

Nel caso particolare è stato associato un tipo enumerato alla variabile che rappresenta il *TerminationCode*, TC, in particolare:

- TC = 0 è utilizzabile tramite *TerminationCode.TC0*;
- TC = 1 è utilizzabile tramite *TerminationCode.TC1*;
- TC = 3 è utilizzabile tramite *TerminationCode.TC3*;
- TC = 27 è utilizzabile tramite *TerminationCode.TC27*;

La fase di progettazione del codice si è estesa anche alla personalizzazione del codice generato. Un esempio di personalizzazione effettuata è stata la scelta della tipologia delle variabili di I/0. Nel caso in esame la scelta è ricaduta su *External Global*. Con tale scelta il risultato è stato il seguente:

```
extern my_double Charging_Enabled;
extern my_double BMS_I_Batt;
extern my_double BMS_U_Batt;
extern my_double Cell_T_min;
extern my_double Inlet_command;
extern my_double PP_I_chrg;
extern my_double Proximity_detection;
extern my_double U_Cell_max;
extern my_double BMS_SOC;
extern my_double Exception;
extern my_double Inlet_feedback;
extern TerminationCode Termination_code;
extern my_double current_setpoint;
```

Le variabili sopra dichiarate forniscono i canali di Input e Output al modello. Si nota inoltre come sia stato utilizzato un tipo di dato personalizzato denominato *my_double*. Quest'ultimo fornisce una specializzazione al tipo predefinito *double*, o *float* in ambiente Matlab, in quanto consente di definire delle caratteristiche aggiuntive come range massimo e minimo, rappresentazione in virgola mobile ecc. Vista la scelta effettuata con la tipologia *External Global*, l'utilizzo delle variabili, quando si vuole integrare il codice auto-generato, in un codice scritto manualmente è semplice e diretta. Questa soluzione è un buon compromesso quando si hanno un numero limitato di variabili I/O, se il numero di I/O diventa elevato è necessario utilizzare strutture dati più complesse.

2.2.6 Auto Coding

Nel caso in esame, l'operazione di coding è stata svolta in due fasi:

1. Generazione automatica del codice con l'utilizzo di Embedded Coder;

2. Scrittura manuale del codice per implementare un protocollo seriale di comunicazione.

Nella prima fase, dopo aver personalizzato varie caratteristiche del codice, come l'uso di tipi enumerati o la scelta della tipologia di variabili, nel caso trattato External Global, si è semplicemente lanciato il tool Embedded Coder descritto nei capitoli precedenti. Quest'ultimo ha generato una cartella contente i file sorgente del modello di supervisione logica.

La cartella generata, Fig. 2.23, contiene al suo interno un numero variabile di file. Tale numero dipende dal livello di ottimizzazione che vogliamo attribuire al codice. Nel caso in esame si è scelto un basso livello di ottimizzazione per ottenere maggiore chiarezza e leggibilità al codice.

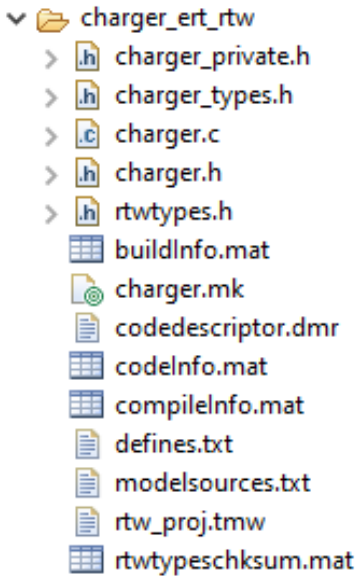

Figura 2.23: Gerarchia del codice auto-generato

I file più rilevanti per l'integrazione del codice auto-generato con altre porzioni del codice sono i seguenti:

charger.c è il file principale. Contiene le definizioni delle variabili di I/O e dei parametri. Al suo interno sono presenti tre funzioni che regolano l'evoluzione del modello:

```
1 extern void charger_initialize(void);
2 extern void charger_step(void);
3 extern void charger_terminate(void);
```

La prima e l'ultima servono rispettivamente all'inizializzazione e alla terminazione di tutto il processo. In questa fase vengono istanziati i parametri e definite le strutture. La funzione più importante è la step. Essa fa evolvere di un passo il modello producendo in output i risultati. Nel caso in esame, avendo scelto una tipologia External Global, la funzione non ha parametri in quando le variabili di I/O sono globali, come la funzione stessa.

charger.h file di libreria principale, contiene le firme delle funzioni di *charger.c* e la dichiarazione di variabili e parametri.

charger_types.h contiene eventuali tipi di dati dichiarati dall'utente, nel caso in esame, come definito in precedenza, è stato utilizzato un tipo di dato enumerato per rappresentare il Termination Code.

```
typedef enum {
    TC0 = 0,
    TC1 = 1,
    TC3 = 3,
    TC27 = 27
} TerminationCode;
```

rtwtypes.h contiene un ri-definizione dei tipi di dati utilizzati. Un esempio è nell'utilizzo del tipo di dato customizzato *my_doble* associato ad un *double*:

```
typedef double real_T;
typedef real_T my_double;
```

2.2.7 Manual Coding

La seconda fase di coding è stata la realizzazione di un protocollo di comunicazione basato su porta seriale. Per far ciò si è realizzata una funzione, vedi App. C, con la seguente firma:

```
int  processData ( uint8_t p_data );
```

Essa prende in input un carattere espresso sotto forma di *uint8* e restituisce un intero che rappresenta un codice di errore. La funzione rappresenta una macchina a stati che evolve in funzione dell'input riconoscendo quando il payload del pacchetto seriale è compromesso o meno. Il pacchetto risulta essere compromesso se non sono presenti i caratteri di inizio e di fine o se la lunghezza del payload è diversa da quella predefinita. Una trattazione più approfondita della comunicazione seriale implementata è discussa nelle sezioni successive.

Per quanto riguarda l'integrazione del codice auto-generato all'interno del micro-controllore, è stato realizzato un loop infinito che, ad ogni iterazione, richiama la funzione *charger_step()* per far evolvere la macchina a stati. Come si evince dal codice sotto riportato, prima della chiamata alla funzione, è necessario prelevare le variabili in input. Questo passaggio è stato realizzato tramite la funzione predefinita *HAL_UART_Receive* la quale legge un pacchetto completo (38 bytes) su seriale. Una volta che il pacchetto è stato letto, quest'ultimo viene considerato temporaneo in quanto non è ancora stata verificata la coerenza del messaggio. Se la ricezione ha avuto esito positivo, viene invocata la funzione per il processamento dei dati, la quale byte per byte, determina o meno la correttezza strutturale del payload. Se il payload viene dichiarato strutturalmente corretto, allora vengono attribuiti i valori

in input alle variabili external global del modello, poi viene eseguita la funzione di step. Dopo che il modello ha compiuto la sua evoluzione ad un passo, le variabili in output da esso, sempre di tipo external global, vengono inizializzate e copiate all'interno di un array che sarà poi spedito su seriale. L'array in questione verrà dotato di header e terminator per poter essere correttamente interpretato dallo schema a blocchi in Simulink, proposto in Fig. 2.26.

```
    while (1)
    {
        /* USER CODE END WHILE */

        /* USER CODE BEGIN 3 */

        ret_RX = HAL_UART_Receive(&huart3, (uint8_t *)pack2RX, sizeof
    (pack2RX),100);
        if(ret_RX == HAL_OK){
            for(x=0 ; x < sizeof(pack2RX) ; x++){
                ret = processData(pack2RX[x]);
            }

            //Input for the model
            Charging_Enabled = arrayPayload[0].number;
            BMS_I_Batt = arrayPayload[1].number;
            BMS_U_Batt = arrayPayload[2].number;
            Cell_T_min = arrayPayload[3].number;
            Inlet_command = arrayPayload[4].number;
            PP_I_chrg = arrayPayload[5].number;
            Proximity_detection = arrayPayload[6].number;
            U_Cell_max = arrayPayload[7].number;
            BMS_SOC = arrayPayload[8].number;

            // Step the model
            charger_step();

            //Output from the model
            arrayPayload2TX[0].number = Exception;
            arrayPayload2TX[1].number = Inlet_feedback;
            arrayPayload2TX[2].number = Termination_code;
            arrayPayload2TX[3].number = current_setpoint;

            pack2TX[0]=HEADER_VALUE;
            x=1;
            for(u=0 ; u < 4 ; u++){
                for(z=0 ; z < PAYLOAD_SIZE/BYTE_NUMBER ; z++){
                    pack2TX[x++] = arrayPayload2TX[u].bytes[z];
                }
            }
```

```
40                pack2TX[sizeof(pack2TX)-1]=TERM_VALUE;
41
42                ret_TX = HAL_UART_Transmit(&huart3, pack2TX, sizeof(pack2TX
    ),10);
43            }
44            /* USER CODE END 3 */
45        }
```

2.2.8 Software integration (SIL)

Parte fondamentale dell'approccio MBD è il processo di testing che può essere effettuato sfruttando il principio di Software-in-the-loop. Embedded Coder permette, tramite l'utilizzo di SIL/PIL Manager, di eseguire in codice generato al posto del modello. In questo modo si realizza una simulazione di tipo SIL. Utilizzando poi il Data Inspector, Fig. 2.24, in modalità compare è possibile confrontare i risultati ottenuti tramite il modello e quelli ottenuti tramite SIL, con lo scopo di verificare il corretto funzionamento del codice. Si nota come non ci stanno differenze, infatti eseguendo un test di tipo baseline, otteniamo per tutti i segnali un esito positivo. Nel segnale considerato, quello del SOC, si nota come i due segnali sono sovrapposti e dunque, nel grafico in basso, l'errore è costante e pari a zero.

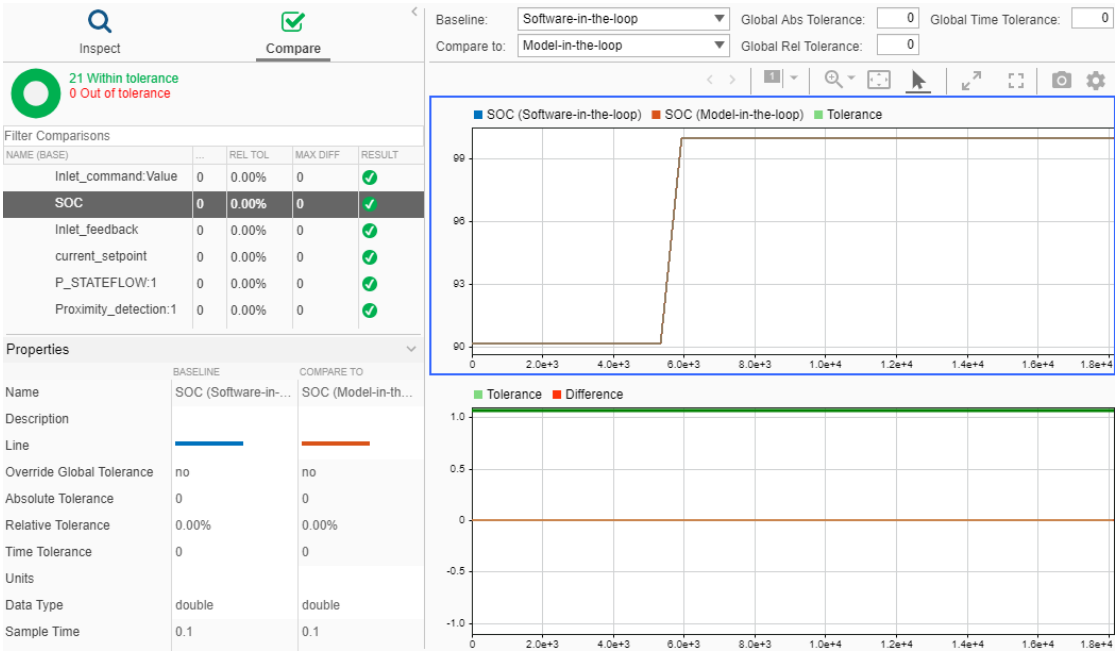

Figura 2.24: Comparazione SIL e MIL

2.2.9 Hardware/Software integration (PIL)

Come descritto nei paragrafi precedenti, il passo successivo per determinare la corretta esecuzione di ciò che è stato realizzato, dopo aver eseguito delle simulazioni di tipo SIL, è quello di integrare il software all'interno di qualche componente hardware. Nel caso in esame è stato realizzato uno schema a blocchi Simulink, Fig. 2.25, in grado di eseguire simulazioni di tipo Processor-in-the-loop. Nello specifico

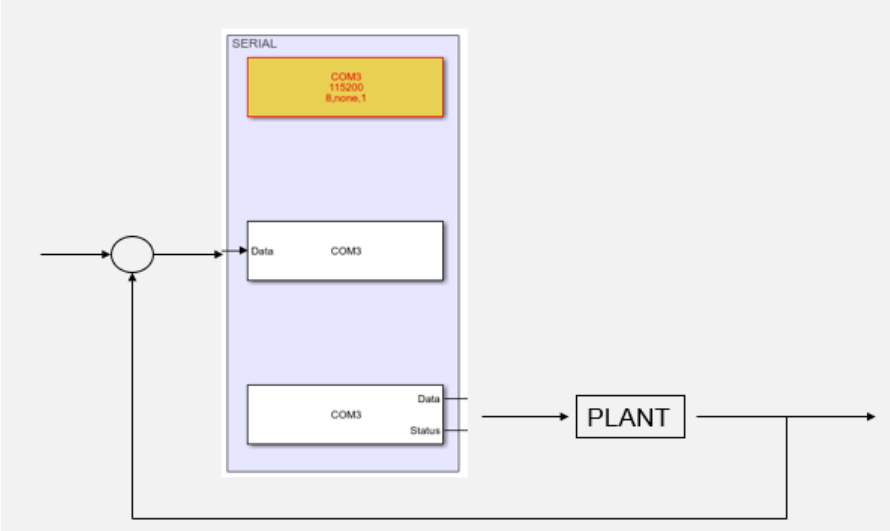

Figura 2.25: Processor-in-the-loop

si è utilizzata la libreria *Instrument Control Toolbox* di Simulink, la quale al suo interno contiene tre blocchi principali:

1. Serial Configuration: serve a configurare la porta seriale che si vuole utilizzare. Tramite questo blocco è possibile impostare il rate di lettura/scrittura, il timeout di fine lettura e l'eventuale presenza di un bit di parità;

2. Serial Send: permette di inviare tramite la porta individuata, un pacchetto di dati caratterizzato da un carattere di inizio e uno di fine. Qui sono stati scelti il carattere *"A"* come header e il carattere *"/n"* come terminatore.

3. Serial Receive: consente di ricevere pacchetti dalla porta seriale individuata. E' possibile, tramite la porta Status, capire se sono presenti nuovi messaggi sulla porta.

I tre blocchi sopra descritti, sono stati impiegati in Fig. 2.25 in modo tale da avere il codice auto-generato contenente la logica di supervisione all'interno del micro-controllore, e il plant, caratterizzato dalla curva di carica, all'interno del file Simulink.

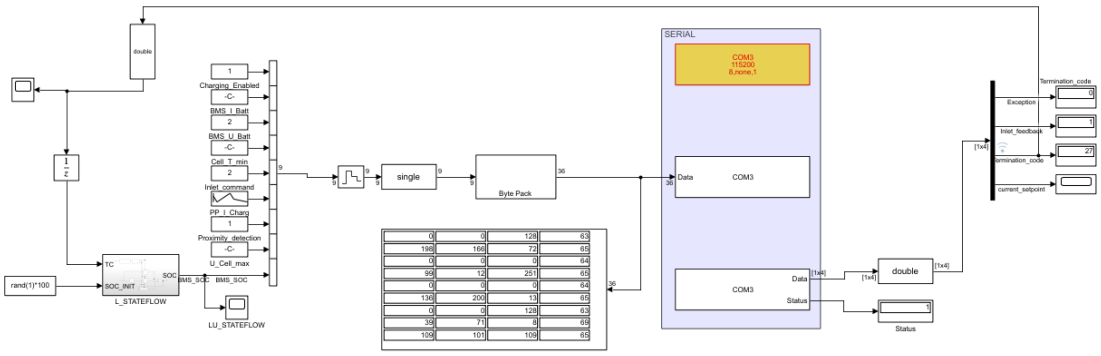

Figura 2.26: Schema di simulazione PIL

L'esecuzione di prove di funzionamento, basate su metodologia PIL, ha richiesto l'implementazione di una dashboard in grado di pilotare le variabili in input e visualizzare i valori in output. La parte di dashboard per l'invio delle informazioni al modello, tramite blocco Serial Send è riportata in Fig. 2.27. Tramite l'interfaccia creata è possibile simulare l'inserimento della spina all'interno della presa del veicolo (Proximity Detection), il comando di bloccaggio spina che la VCU invia al Charger (Inlet Command), l'abilitazione alla carica che la VCU invia al Charger (Charging Enabled). Inoltre, grazie a degli sliders, è possibile modificare alcuni dei parametri come la temperatura, la tensione e la corrente di carica.

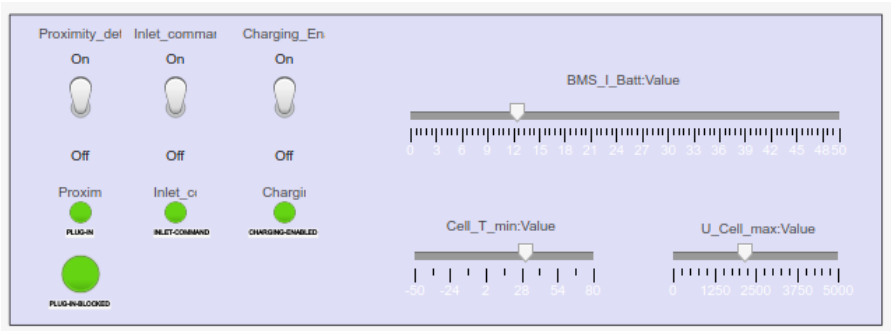

Figura 2.27: Dashboard di input

Una volta che il modello elabora le informazioni passate tramite protocollo seriale, esso risponde costituendo a sua volta un pacchetto seriale. Tale pacchetto viene interpretato dalla Serial Receive e le informazioni contenute al suo interno vengono visualizzate con la dashboard in Fig. 2.28.

Non avendo a disposizione l'hardware definitivo, presente all'interno del veicolo, per la gestione dell'OBC, non è stato possibile eseguire simulazioni di tipo Hardware-in-the-Loop.

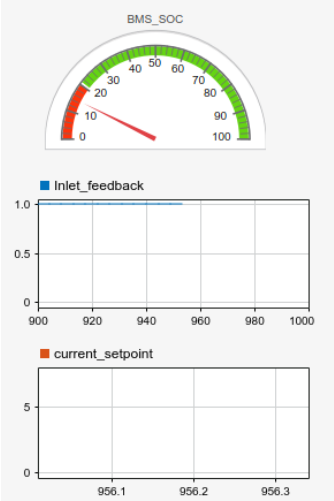

Figura 2.28: Dashboard di output

2.3 Software

Come descritto nelle sezioni precedenti, il codice si compone di due macro parti: quello auto-generato a partire dai modelli Simulink e quello scritto manualmente per la gestione della comunicazione seriale. Il software che è stato deciso di utilizzare per poter eseguire l'operazione di debug sul micro controllore, è STM32CubeIDE ovvero l'Integrated Development Environment sviluppato da STMicroelectronics.

2.3.1 STM32 CubeIDE

STM32CubeIDE è uno strumento di sviluppo multi-OS all-in-one, che fa parte dell'ecosistema software STM32Cube.

STM32CubeIDE è una piattaforma di sviluppo C/C++ avanzata che consente una configurazione delle periferiche in modo grafico, generazione automatica di codice, compilazione di codice e funzionalità di debug per microcontrollori e microprocessori STM32. Si basa sul framework Eclipse e sulla toolchain GCC per lo sviluppo e GDB per il debug. Dopo la selezione di un MCU o MPU STM32 vuoto, o di un microcontrollore o microprocessore preconfigurato dalla selezione di una scheda, viene creato il progetto e generato il codice di inizializzazione. In qualsiasi momento durante lo sviluppo, l'utente può tornare all'inizializzazione e alla configurazione delle periferiche e rigenerare il codice di inizializzazione senza alcun impatto sul codice scritto dall'utente.

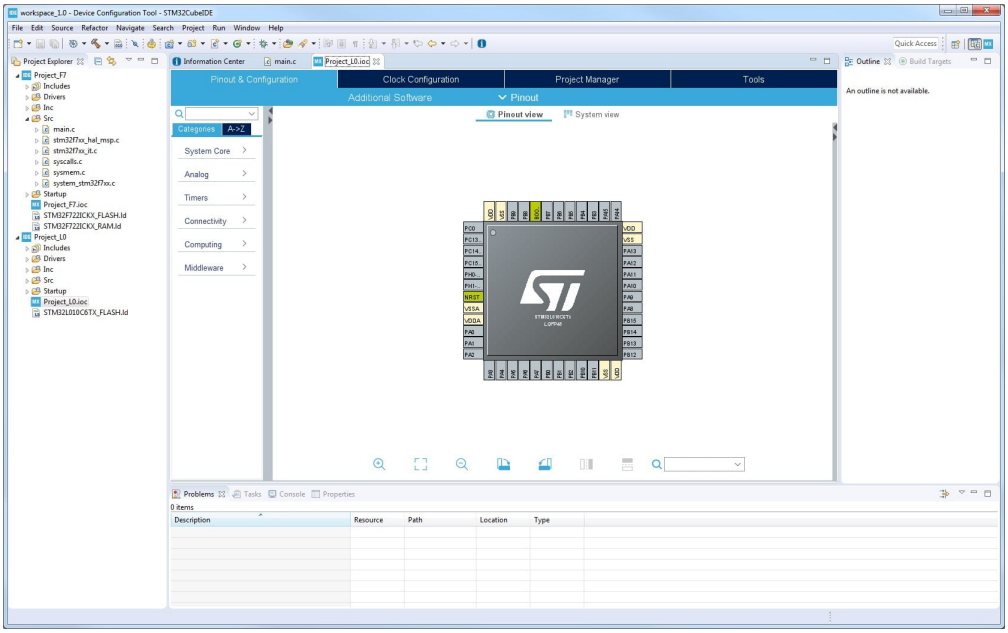

Figura 2.29: Interfaccia STM32 Cube IDE

2.3.2 Comunicazione seriale

Come già introdotto, la comunicazione tra la board STM32 e il PC Host avviene attraverso protocollo seriale. Si è scelto di utilizzare un cavo USB per la sua implementazione.

Nel caso in esame dell'OBC, si è scelto di rappresentare le variabili di I/O di tipo double o float, per cui rappresentabili ciascuna con 4 bytes. Avendo in input 9 variabili si ottiene un payload di 36 bytes. Come raffigurato in Fig. 2.30, al payload vanno aggiunti il carattere header e terminator, ciascuno rappresentabile con un byte per un totale di 38 bytes. Questa è la dimensione del pacchetto in input al modello. Per quanto riguarda quello in output, le variabili sono solo 4 per cui abbiamo una dimensione minore, ma la logica di costruzione strutturale del pacchetto rimane invariata.

La gestione software della comunicazione è stata riportata nell'appendice C per completezza. Il codice si basa sulla funzione *processData* la quale prendendo in input un singolo byte determina se il pacchetto letto su seriale è strutturalmente corretto. La funzione implementa una macchina a stati con il seguente funzionamento:

- Se il primo byte in input è il carattere header, la macchina a stati finiti (Finite State Machine), FSM passa dallo stato iniziale *STATE_HEADER* allo stato di lettura del payload *STATE_READ_PAYLOAD*. Se il primo byte non è il carattere iniziale la FSM rimane nello stato iniziale;

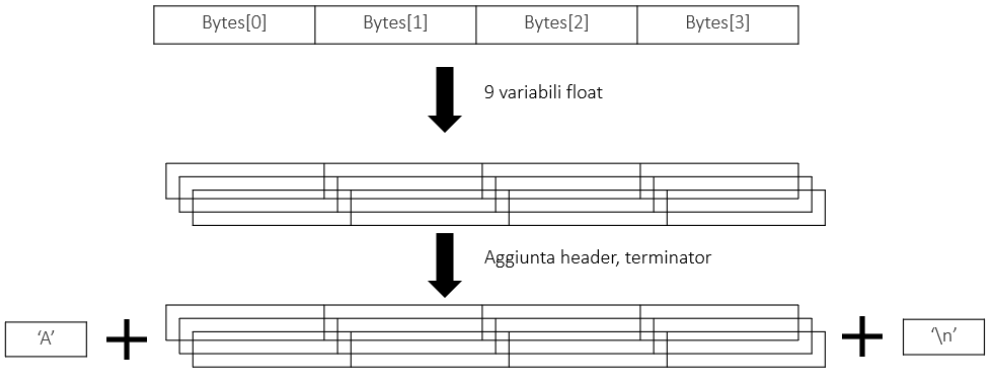

Figura 2.30: Pacchetto dati

- Se la macchina si trova nello stato *STATE_READ_PAYLOAD* per un numero di step pari alla lunghezza effettiva del payload, allora per ogni float letto, copia il valore di quest'ultimo in una struttura temporanea;
- Arrivati ad un numero di step pari alla dimensione del payload, lo stato successivo sarà *STATE_TERMINATOR*;
- Se la FSM si trova in *STATE_TERMINATOR* e non riceve in input il byte di terminazione allora scarta la struttura dati temporanea in cui sono stati memorizzate le variabili lette nel payload. Altrimenti copia la struttura dati temporanea in una struttura dati definitiva.

2.4 Hardware

L'hardware utilizzato per la fase di Processor-in-the-loop è la board *STM32 Nucleo-144* prodotta dalla STMicroelectronics. Tramite la board è possibile mettere alla prova il codice sviluppato dall'utente su un hardware personalizzabile sia per quanto riguarda le performance, sia per il consumo energetico. Essendo la board dotata di debugger integrato, il caricamento del codice al suo interno non richiede altri componenti esterni, è solo necessario collegare questa al PC host. Inoltre la board presente alcune feature caratteristiche come:

- STM32 microcontroller in LQFP144 package;
- 3 LED disponibili all'utente;
- 2 bottoni disponibili all'utente;

- 32.768 kHz crystal oscillator, per la generazione di timer;
- Diverse possibilità di alimentazione: tramite cavo di debug, ST-LINK o alimentazione esterna;
- Porta Ethernet con standard IEEE-802.3-2002
- Porta Micro-USB per gestione OTG.

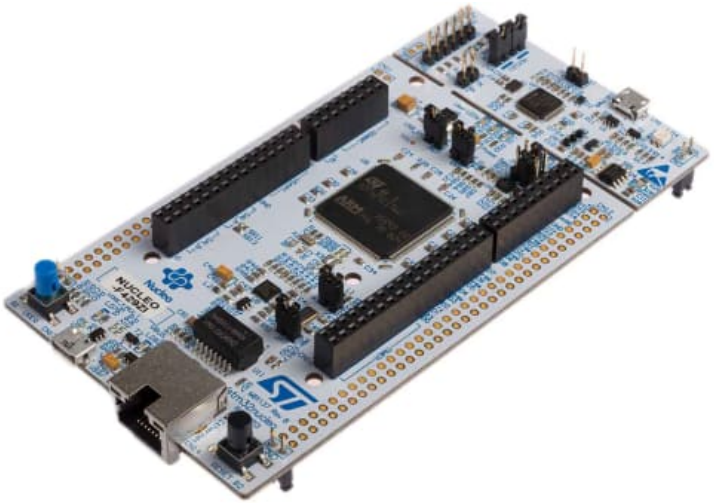

Figura 2.31: STM32 NUCLEO-F429ZI Nucleo-144 Development Board

2.5 Test

La fase di test, come prevede la metodologia del Model Based Design, è stata effettuata durante tutte le fasi di progettazione. In questa sezione verrà proposta solo la gestione delle fasi di test finale. In particolare la generazione di test cases aggiuntivi per la verifica della copertura, e l'implementazione di segnali per testare alcune funzionalità del modello.

2.5.1 Generazione di test cases

Come specificato nei capitoli precedenti, il modello, vedi Fig. 2.22, presenta una copertura condizionale e MCMD piena. Questi risultati sono stati ottenuti, non solo con le simulazioni progettate manualmente in cui si simulano situazioni previste o accidentali, ma sono state generate in modo automatico ulteriori casi di test per comprendere la reazione del modello. Queste simulazioni aggiuntive, sono frutto

dell'utilizzo si Simulink Design Verifier, tool introdotto nei capitoli precedenti. In Fig. 2.32 sono riportati alcuni dei Test Cases generati. Essi non necessariamente hanno un senso logico, ovvero potrebbero non essere fisicamente praticabili nella realtà. Questa loro caratteristica però, li rende adatti per testare il comportamento del modello.

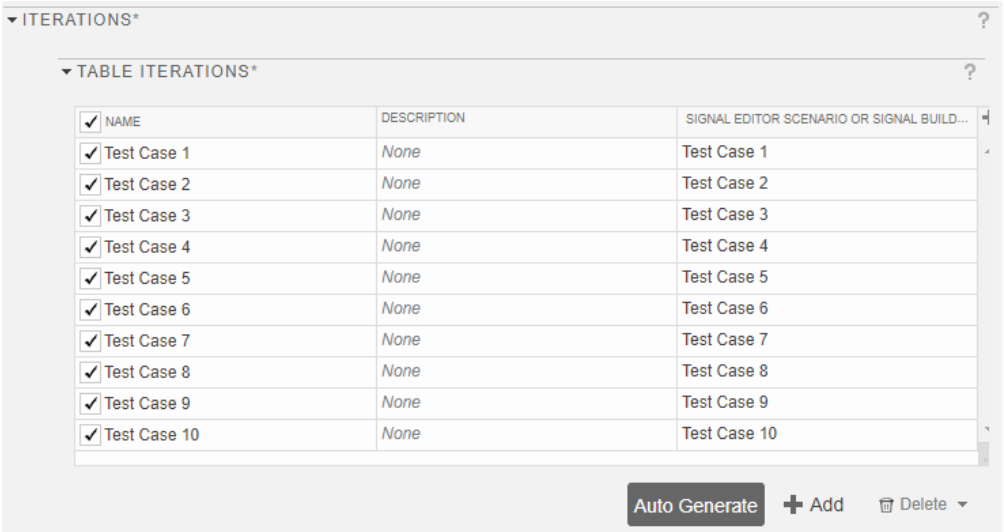

Figura 2.32: Generazione di test con Simulink Design Verifier

Per quanto riguarda i test che possiedono un riscontro reale, ovvero quelli che vogliono simulare una situazione fisicamente possibile, sono stati utilizzati due approcci:

1. Mediante l'utilizzo della dashboard. Il problema di questa tecnica è l'irripetibilità del singolo test;

2. Utilizzando il blocco Signal Builder, che consente la creazione di segnali in modo grafico, vedi Fig. 2.33. Con questa metodologia sono state simulate diverse situazioni. Quella riportata in Fig. 2.33 è il caso ideale di carica che porta la batteria ad un SOC maggiore di 99, senza problemi durante la carica.

I segnali in input, generati con un Signal Builder in Fig. 2.33 hanno portato ai risultati in Fig. 2.34. Nell'immagine sono stati riportati due grafici: quello in alto rappresenta l'evoluzione che è avvenuta tra i diversi stati della macchina, il secondo l'andamento del SOC. Osservando gli andamenti tra gli stati si nota come ci sia un "palleggiamento" tra gli stati *CHARGING*, *SEND_CHG3* e *SEND_CHG1*. Questa fase rappresenta la vera e propria fase di carica, infatti nel momento in cui il SOC raggiunge la soglia massima, lo stato in cui la FSM và è quello di *END_CHARGING* per poi attendere l'annullamento della corrente di carica nello

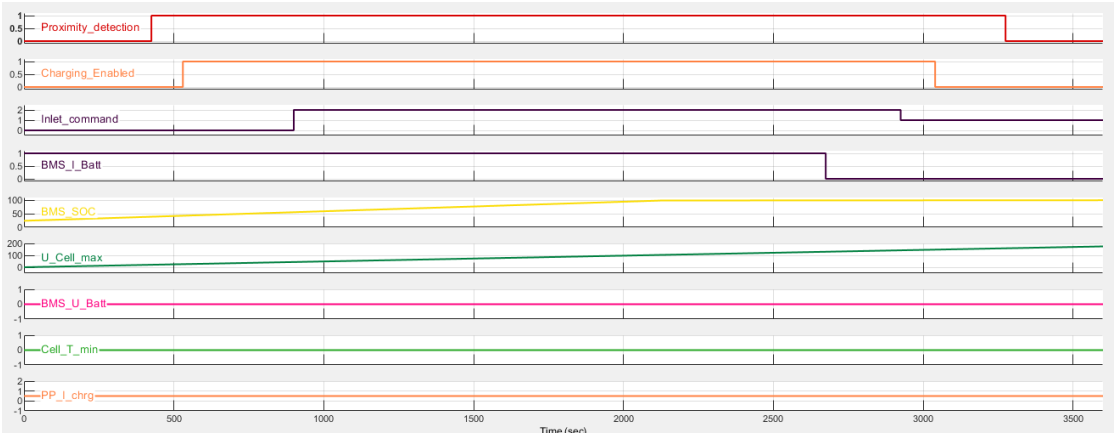

Figura 2.33: Generazione di test con Signal Builder

stato *WAIT_NOCURRENT*. Una volta che la corrente di batteria è posta a 0, il modello torna nello stato iniziale di *INIT*.

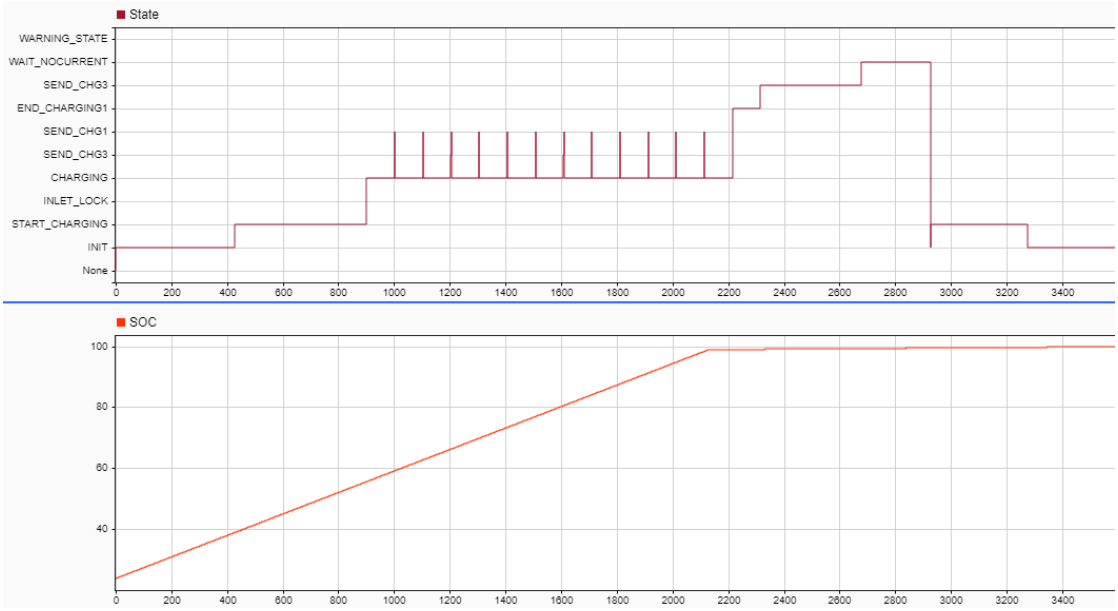

Figura 2.34: Risultato simulazione con Signal Builder

I risultati della simulazione sopra proposta, Fig. 2.34, come si può notare da grafico inferiore, non tengono conto della dinamica del SOC modellata e descritta nei capitoli precedenti. Infatti, la curva del SOC presenta un andamento non reale. I risultati di alcune simulazioni più concrete, utilizzando la metodologia del Processor-in-the-loop, sono stati riportati in Fig. 2.35 e in Fig. 2.36, in cui

non viene utilizzato un Signal Builder, ma la dashboard. Quest'ultima fornisce gli ingressi al modello che è caricato all'interno del micro-controllore.

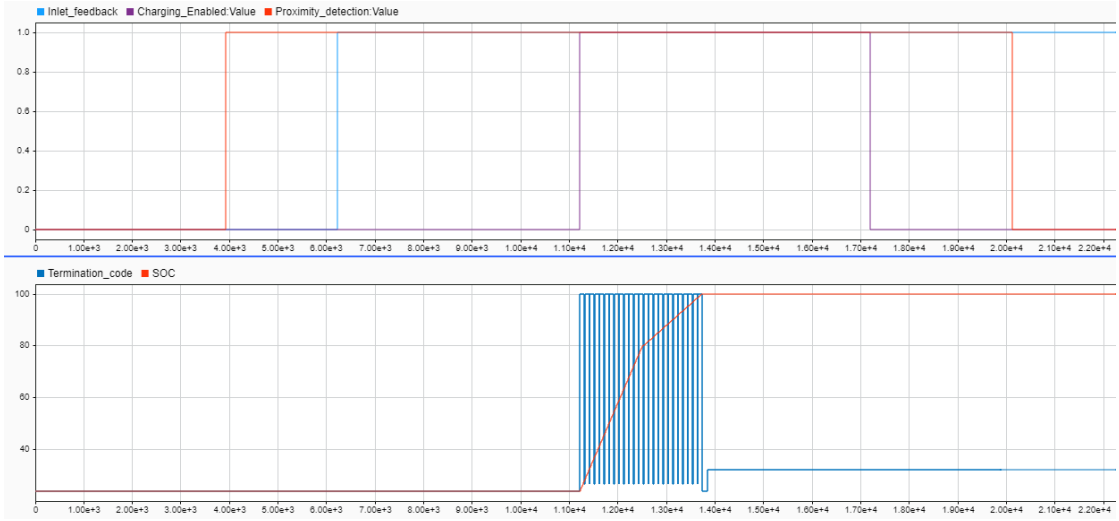

Figura 2.35: Data Inspector

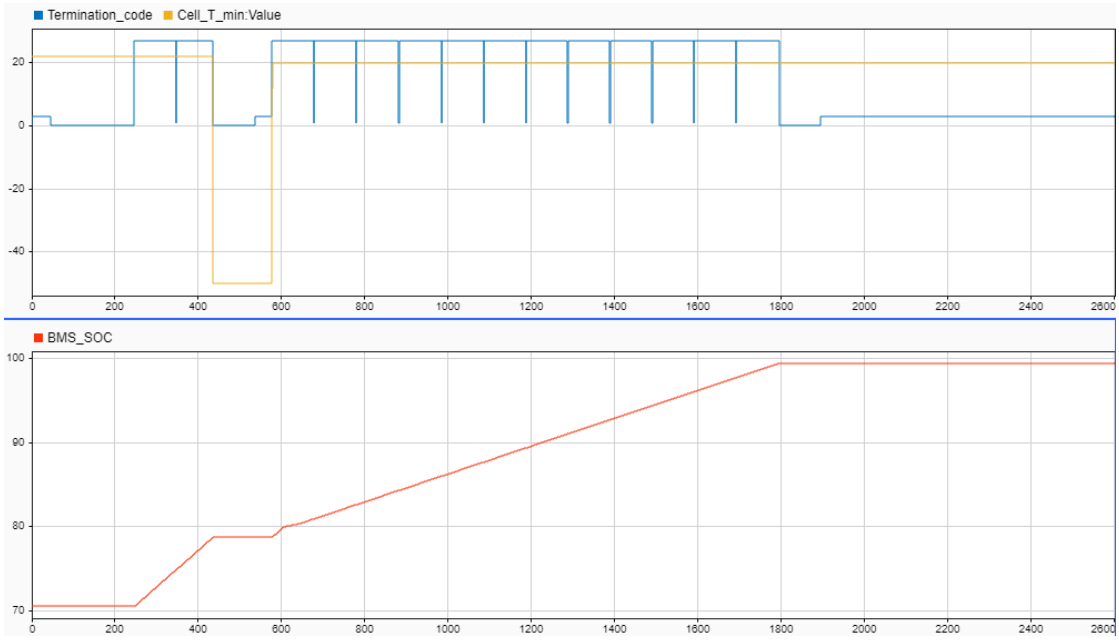

Figura 2.36: Data Inspector - 2

Il grafico superiore, in Fig. 2.35, rappresenta l'inserimento della spina, la rilevazione della spina e il segnale di abilitazione alla carica che la VCU invia al

charger. La parte sottostante riporta il Termination Code, ovvero il codice che identifica lo stato della FSM, e la curva di SOC reale. Si nota come la curva di SOC parte da un valore iniziale di circa 30 e rimane costante fino a quando il segnale di abilitazione dà l'autorizzazione. In questo grafico, come in quello precedente, si nota il "palleggiamento" tra i vari stati della FSM che rappresentano la carica. Solo in questa fase il SOC cresce fino alla sua soglia massima.

In Fig. 2.36 è riportato un ulteriore esempio in cui si va a simulare un calo della temperatura di cella al di sotto della soglia minima. Il grafico superiore raffigura il Termination Code e la temperatura di cella, quello inferiore riporta l'andamento del SOC. Si nota come, nel momento il cui la temperatura scende, il Termination Code assume un valore inferiore a quello che rappresenta lo stato di carica e di conseguenza, come si evince nel grafico del SOC, vi è una situazione di stallo. Quest'ultima termina quando la temperatura torna al proprio valore nominale e di conseguenza la macchina a stati rientra nella situazione di carica.

Capitolo 3

ISO 26262

3.1 La norma

Le pratiche di sicurezza stanno diventando più regolamentate man mano che le industrie adottano un insieme standardizzato di pratiche, per la progettazione e il test dei prodotti. ISO 26262 risponde alle esigenze di uno standard internazionale specifico per il settore automobilistico. ISO 26262 deriva dalla norma IEC 61508, lo standard generico per la sicurezza funzionale dei sistemi elettrici ed elettronici (E/E).

La crescente complessità in tutta l'industria automobilistica si traduce in maggiori sforzi per fornire sistemi conformi alla sicurezza. L'obiettivo della ISO 26262 è fornire uno standard di sicurezza unificante per tutti i sistemi E/E automobilistici.

Lo standard ISO 26262 fornisce regolamenti e raccomandazioni durante tutto il processo di sviluppo del prodotto, dallo sviluppo concettuale alla produzione in serie. Descrive in dettaglio come assegnare un livello di rischio accettabile a un sistema o componente e documenta il processo di test complessivo. In generale, la ISO 26262:

- Fornisce un ciclo di vita della sicurezza automobilistica (management, development, production, operation, service, decommissioning) e supporta la personalizzazione delle attività necessarie durante queste fasi del ciclo di vita;

- Fornisce un approccio basato sul rischio specifico per il settore automobilistico per la determinazione delle classi di rischio (Automotive Safety Integrity Level, ASIL);

- Utilizza ASIL per specificare i requisiti di sicurezza necessari per ottenere un rischio residuo accettabile;

- Fornisce i criteri per garantire il raggiungimento di un livello di sicurezza sufficiente e accettabile.

Automotive Safety Integrity Level (ASIL)

Definito il modo formale all'interno dello standard parte 9, ASIL si riferisce al livello di integrità della sicurezza automobilistica, [11]. Si tratta di un sistema di classificazione dei rischi definito dalla norma per la sicurezza funzionale dei veicoli stradali.

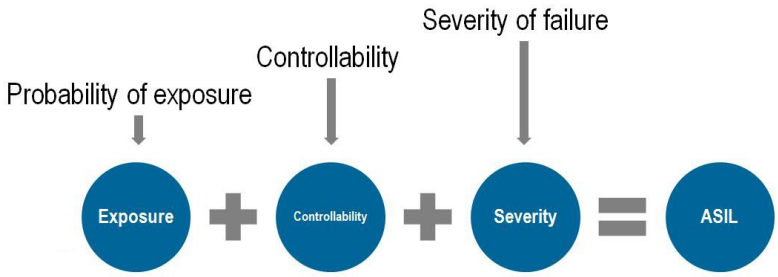

Figura 3.1: Determinazione del livello ASIL

La norma definisce la sicurezza funzionale come *"l'assenza di rischio irragionevole dovuto a pericoli causati da comportamenti di malfunzionamento dei sistemi elettrici o elettronici"*. I livelli ASIL stabiliscono requisiti di sicurezza, basati sulla probabilità e accettabilità del danno, affinché i componenti automobilistici siano conformi.

Ci sono quattro ASIL identificati da ISO 26262: A, B, C e D. ASIL A rappresenta il grado più basso e ASIL D rappresenta il grado più alto di rischio automobilistico.

Gli ASIL vengono stabiliti eseguendo l'analisi dei pericoli e la valutazione dei rischi. Per ogni componente elettronico di un veicolo, gli ingegneri misurano tre variabili specifiche:

1. Gravità (il tipo di lesioni al conducente e ai passeggeri);
2. Esposizione (quanto spesso il veicolo è esposto al pericolo);
3. Controllabilità (quanto può fare il conducente per prevenire l'infortunio).

La somma dei tre indicatori da origine al livello di *Automotive Safety Integrity*.

3.1.1 Struttura della norma

Lo standard, [11], è stato aggiornato nel 2018. Quello considerato in questo elaborato è la versione precedente del 2011. In questa versione sono presenti 10 parti:

1. Vocabulary;
2. Management of functional safety;
3. Concept phase;
4. Product development at the system level;
5. Product development at the hardware level;
6. Product development at the software level;
7. Production and operation;
8. Supporting processes;
9. Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analysis;
10. Guideline on ISO 26262.

Nella versione ISO 26262:2018 vengono aggiunte due ulteriori parti, Part 11 e Part 12, rispettivamente: *Guidelines on application of ISO 26262 to semiconductors* e *Adaptation of ISO 26262 for motorcycles*.

Part 1: Vocabulary

In questa prima parte, vengono date le definizioni formali di fault, failure, error, malfunction. Vengono introdotti, sotto forma di glossario, anche tutti gli altri termini utilizzati nella definizione delle altre parti della norma, in modo da limitare possibili ambiguità.

Part 2: Management of functional safety

La norma prevede uno standard, come insieme di regole, per l'ottenimento della sicurezza funzionale dei prodotti e dei processi nel settore automotive. In questa parte vengono introdotti i concetti di Hazardous Event, Safety Goals, Safety Requirements. Vengono gettate le basi per far si che le operazioni di management dietro ad un processo siano "safety".

Parts 3-7: Safety Life Cycle

Rappresentano le parti centrali della norma ISO 26262. Vengono identificati i rischi e i possibili requisiti di safety per ridurre questi sotto una soglia accettabile. Si trovano le nozioni per la progettazione di sistemi sicuri, hardware sicuri e software sicuri. La parte 7 illustra come i processi di produzione, service e decommissioning dovrebbero essere eseguiti per poter dichiarare il prodotto finito ISO 26262 compliance.

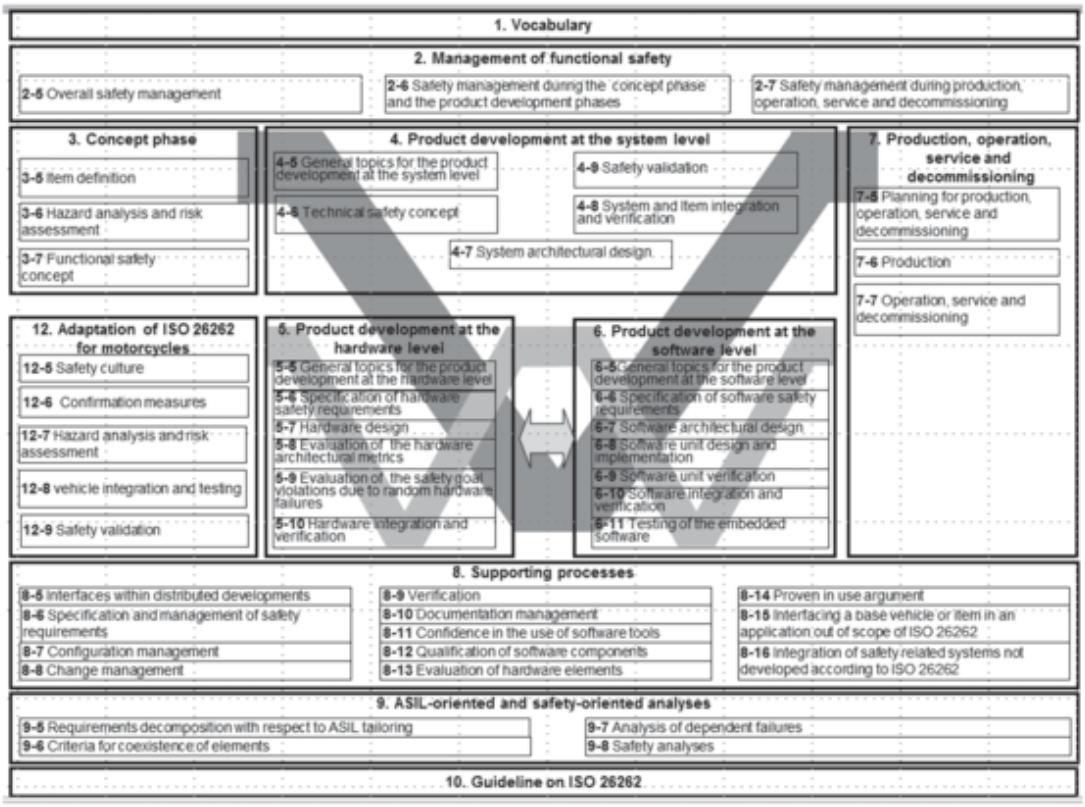

Figura 3.2: Struttura della norma ISO 26262

Part 8: Supporting Processes

Lo standard in esame copre tutto il ciclo di vita della sicurezza del prodotto. In particolare la fase di supporto ai distributori delle varie componenti del prodotto con una specificazione esplicita dei requisiti di sicurezza che anche essi devono rispettare. Altro aspetto discusso nella norma è la gestione formale dei requisiti: gestione delle modifiche e valutazione dell'impatto che esse hanno nei confronti dei requisiti funzionali di sicurezza.

Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analysis

Viene discussa in modo formale la classificazione ASIL, trattata nella sezione precendente, dandone una definizione ed un metodo di calcolo.

3.2 Model Based Design in ISO 26262

Il software in output da un processo di Model Based Design può essere dichiarato conforme allo standard ISO 26262. La norma prevede non solo requisiti per componenti software ma anche per parti hardware. Quest'ultime non saranno nominate in questo elaborato. La qualificazione dei componenti software implica attività quali la definizione dei requisiti funzionali, l'utilizzo delle risorse e la previsione del comportamento del software in situazioni di guasto e sovraccarico. Questo processo è notevolmente semplificato utilizzando software qualificato, come i prodotti proposti da MathWorks, durante lo sviluppo di un'applicazione. I componenti software qualificati sono generalmente prodotti ben consolidati che vengono riutilizzati nei progetti e includono librerie, sistemi operativi, database e driver software.

Per qualificare un componente software, lo standard richiede test in condizioni operative normali insieme all'inserimento di guasti nel sistema per determinare come reagisce a input anomali. Errori software come errori di runtime e dati vengono analizzati e affrontati durante il processo di progettazione.

3.3 ISO 26262 e MathWorks

In questa sezione verrà sintetizzato il flusso di lavoro, approvato da *TÜV SÜD*, per l'utilizzo della metodologia del Model Based Design compatibile con lo standard ISO 26262, [12]. In Fig. 3.3 è riportato il workflow che MathWorks propone.

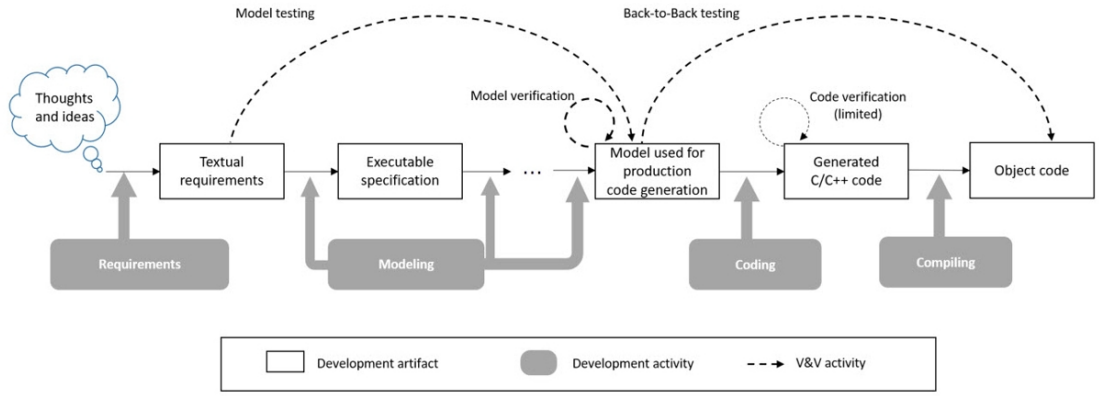

Figura 3.3: ISO 26262 - MathWorks workflow

Esso ha una struttura visivamente diversa dallo schema a V proposto nei capitoli precedenti, ma il comportamento è il medesimo. In questa sezione saranno discusse le fasi principali dello schema proposto, per ottenere un prodotto ISO 26262

compliance. Lo schema riportato si riferisce per la maggior parte dei concetti alla parte 6 della norma ISO 26262, [13], titolata *Product development at the software level*, che tratta la gestione dei requisiti funzionali di sicurezza per le componenti software.

Di seguito, saranno discussi brevemente i concetti da applicare nelle fasi principali dello schema a V, o analogamente al workflow di Fig. 3.3, per l'ottenimento di un software conforme allo standard in esame.

3.3.1 Requirements Development

Lo standard prevede la definizione di requisiti funzionali volti a garantire la safety del sistema. Questi devono presentare una tracciamento bidirezionale tra il requisito e la sua implementazione. Ciò può essere fatto, come illustrato nel capitolo dedicato a Simulink Requirements, con il tool che MathWorks mette a disposizione. Il tool permette anche l'integrazione con altri software di gestione dei requisiti come IBM Doors. Anche con l'utilizzo di software terzi, la tracciabilità è garantita. Inoltre al singolo requisito può essere associata una barra di avanzamento dell'implementazione per monitorare lo stato di avanzamento del progetto. Come già discusso, è possibile verificare con dei test case ad-hoc i singoli requisiti garantendo dunque la loro corretta implementazione. Simulink Requirements, in accordo con ISO 26262, consente non solo il tracciamento bidirezionale tra modello e requisito, ma anche tra codice auto-generato e requisito.

3.3.2 Design Modeling

In questa macro fase, che racchiude tutto il processo di progettazione e modellazione, in cui si ha come punto di partenza delle specifiche di alto livello e come punto di arrivo il modello definitivo, è possibile utilizzare i tool di MathWorks per adempiere agli obblighi imposti dalla norma. Ad esempio se una componente del sistema possiede un livello di integrità medio alto (ASIL B-D), è necessario che il modello realizzato sia coerente con le linee guida di modellazione. Un esempio di linee guida per una buona modellazione è MAAB. Queste possono essere verificate con l'utilizzo di Simulink Check. Altri esempi di modiche che portano da specifiche di alto livello al modello finale definitivo sono la conversione del dominio: dall'utilizzo di blocchi che sfruttano il tempo continuo, all'utilizzo di blocchi nel domino del tempo discreto. La conversione del domino può essere facilitata con Simulink Control Design. Oppure la conversione dei tipi di dato utilizzati in funzione dell'hardware di destinazione, operazione facilitata con il tool Fixed-Point-Designer.

3.3.3 Code Generation

All'interno della norma viene discussa la generazione automatica di codice e la traduzione di questo in codice oggetto. MathWorks con Embedded Coder, consente di generare codice C, C++ e AUTOSAR a partire dal modello creato. Tramite Simulink Check è possibile verificare se il codice generato soddisfa alcuni standard di buona programmazione come le linee guida MISRA C. La norma evidenzia come Embedded Coder sia un tool che soddisfi i requisiti imposti dalla ISO 26262. Utilizzando, un ulteriore tool Mathworks *IEC Certification Kit* è possibile qualificare il software realizzato.

3.3.4 Design Verification

La ISO 26262 raccomanda una serie di metodi statici e dinamici per verificare i modelli e il codice da essi derivati. Per il MBD, la norma afferma che "a seconda del processo di sviluppo del software, gli oggetti di test possono essere il codice derivato dal modello o il modello stesso". Con Simulink Test possiamo creare e gestire dei test sistematici basati sia su modello sia sul codice. I report generati da questo tool, e certificati con *IEC Certification Kit*, forniscono una garanzia rispetto ai requisiti funzionali di sicurezza imposti dalla norma stessa. ISO 26262 raccomanda di verificare la completezza del modello sfruttando metodologia di verifica della copertura, specie per le componenti ad alto livello di integrità (ASIL-D). Simulink Coverage, offre la possibilità di eseguire analisi di copertura con diverse tecniche, come ad esempio MDMC Coverage, fortemente raccomandata per componenti ASIL-D. Il tool di MathWorks, non solo esegue test di copertura sul modello ma anche sul codice auto-generato, come descritto nei capitoli precedenti. Sempre nella fase di *Design Verification*, utilizzando Simulink Check e l'interfaccia di Model Advisor, è possibile generare delle dashboard che evidenziano la complessità del modello e permettono al progettista di andare a ridurla in modo da semplificare la gestione e manutenzione dei singoli componenti. La norma ISO 26262 pone grande importanza alla progettazione di componenti compatti e di bassa complessità.

3.3.5 Code Verification

In questa fase si ispeziona il codice per verificare che esso non presenti comportamenti indesiderati. A questo problema il tool *IEC Certification Kit* fornisce una soluzione trovando, all'interno del codice, istruzioni non legate ad alcuna parte del modello. Un'ulteriore verifica può essere effettuata confrontando i risultati della simulazione SIL e MIL. Quest'ultima metodologia è stata percorsa nel case study considerato.

Utilizzando tool più complessi come *Polyspace Bug Finder* e *Polyspace Code Prover* è possibile dimostrare l'assenza degli errori che potrebbero verificarsi a runtime, come ad esempio una divisione per zero.

Ultima fase della Code Verification è la tipologia di test Processor-in-the-loop per la verifica che i risultati ottenuti sono coerenti con la simulazione Model-in-the-loop. La norma, specie per le parti con un alto livello di integrità (ASIL-C e ASIL-D) consiglia di utilizzare le metodologie di test in modo iterativo, come raffigurato in Fig. 3.3. Inoltre, lo standard ritiene positivo che l'ambiente di test condivida le stesse caratteristiche dell'hardware di destinazione in modo da limitare al massimo alcune differenze caratteristiche, come ad esempio il numero di bit necessari alla rappresentazione dei numeri.

3.3.6 Tool Qualification

La definizione proposta in [14] di Tool Qualification é la seguente: *"è un processo che ci consente di dimostrare che un tool può essere utilizzato nell'ambito della realizzazione di un'applicazione software con un determinato obiettivo di sicurezza"*.

Come riportato in [15], non tutti i tool utilizzati hanno la necessità di essere qualificati o meglio ancora la qualificazione di un determinato tool dipende dal progetto che si sta sviluppando. La qualificazione di un tool va eseguita se la seguente domanda ha una risposta positiva: *"Il tool potrebbe essere responsabile di un errore nel prodotto finale o il tool potrebbe non rilevare errori in una componente safety-critical?"*, (ISO 26262-8, 11.2). Come detto, se la risposta è positiva, allora dobbiamo porre grande fiducia sul tool che si sta utilizzando. La norma ISO 26262 definisce tre livelli di fiducia (Tool Confidence Level): per il livello più basso, TCL1, non è necessaria fiducia per cui non è necessaria alcuna qualifica. Come si vede in Fig. 3.4 per determinare il livello di confidenza dello strumento è necessaria un'analisi di altri due fattori: Tool Impact (TI) e Tool Error Detection (TD).

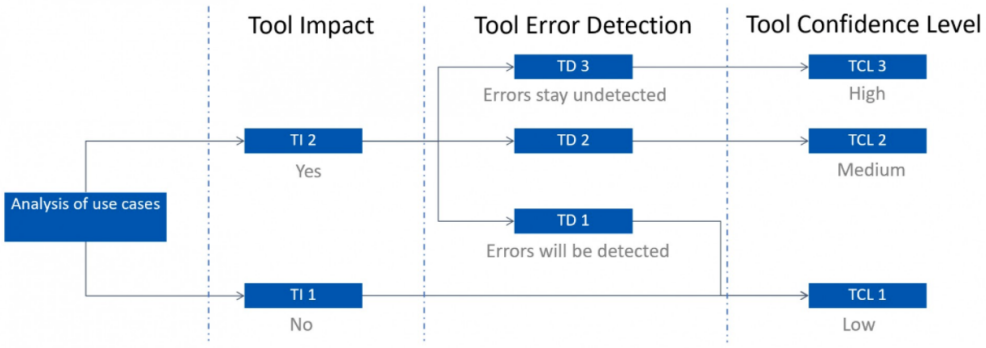

Figura 3.4: Tool Confidence Level

Tool Impact

Si hanno due possibili scelte: se uno strumento può "introdurre o non rilevare" errori, lo strumento avrà effettivamente un impatto sulla qualità del prodotto finale. Questo è quindi chiamato TI2. Se uno strumento non svolge alcun ruolo per quanto riguarda la qualità del prodotto finale, non vi è alcun impatto sullo strumento (TI1).

Tool Error Detection

Per gli strumenti che possono avere un impatto sulla sicurezza del prodotto finale (TI2), è sufficiente porsi una semplice domanda: supponendo che il tool abbia un malfunzionamento, qual è la probabilità di rilevarlo successivamente nel processo. Se c'è un'alta probabilità (TD1), non è necessario avere un'elevata fiducia nel comportamento corretto dello strumento. Se non dovessimo rilevare un malfunzionamento del tool nelle fasi successive del processo (TD2, TD3), dobbiamo avere un'elevata fiducia nel comportamento corretto dello strumento che si vuole qualificare.

Methods		ASIL			
		A	B	C	D
1a	Increased confidence from use	++	++	+	+
1b	Evaluation of the tool development process	++	++	+	+
1c	Validation of the software tool	+	+	++	++
1d	Development in accordance with a safety standard	+	+	++	++

Figura 3.5: TCL3: Qualificazione del software

Una volta stabilito il livello di confidenza del tool, se quest'ultimo ha un valore di TCL2 o TCL3 è necessaria la qualificazione del tool stesso. Nella sezione ISO 26262-8, 11.4.6, riportata in Fig. 3.5, sono riportate alcune metodologia per compiere la qualificazione. Nel caso particolare in cui il livello di confidenza del tool sia TCL3, ovvero il più alto, i metodi consigliati sono i seguenti:

Increased Confidence from Use: si sfrutta il fatto che il tool in questione è già stato utilizzato per altri progetti simili e non ha mai dato dei malfunzionamenti. Da notare come questo metodo si presta a bassi livelli di integrità e non è adatto per livelli ASIL-C o ASIL-D.

Evaluation of the Development Process: richiede un'analisi dettagliata del processo di sviluppo alla base del tool. Generalmente gli utilizzatori del tool non hanno accesso a questi dettagli, però le aziende produttori del tool

(e.g. MathWorks) possono pre-qualificare i propri strumenti appoggiandosi ad autorità come $T\ddot{U}V\ S\ddot{U}D$. Anche in questo caso questo metodo si applica per le classi di integrità AISL più basse.

Validation of the Software Tool: è uno dei metodi più robusti per la qualificazione del tool. Si crea una suite di test ad-hoc in grado di coprire tutti i casi d'uso del tool. Questa metodologia può essere applicata sia dal produttore del tool sia dall'utilizzatore, anche se quest'ultimo si vedrebbe aumentare notevolmente gli sforzi progettuali.

La scelta di un determinato metodo dipende dal particolare livello ASIL associato alla componente software in esame.

IEC Certification Kit è in grado di pre-qualificare gli strumenti della MathWorks fornendo casi d'uso tipici e report di qualificazione verificati da $T\ddot{U}V\ S\ddot{U}D$. Inoltre, $T\ddot{U}V\ S\ddot{U}D$ esamina e verifica lo sviluppo degli strumenti MathWorks e i processi di qualità, nonché le capacità di segnalazione dei bug e certifica i risultati a ogni rilascio del prodotto.

Capitolo 4

AUTOSAR

4.1 Introduzione

AUTomotive Open System ARchitecture (AUTOSAR) è un'architettura software open ed è standardizzata congiuntamente dai produttori di vetture, dai fornitori e dai sviluppatori di tool. La struttura della partnership comprende: Core partners, Premium Members e Associate Member.

In Fig. 4.1 è riportato un interessante grafico che mostra come la crescita della complessità all'interno del veicolo richieda un cambio delle architetture su cui il software si basa. All'inizio, quando la meccanica era preponderante, erano presenti centraline ECU indipendenti, in cui per ogni funzione corrispondeva una sola centralina, [16].

Con il crescere della complessità si è arrivati ad integrare le diverse funzionalità utilizzando un gateway centralizzato. Questa metodologia ha permesso di gestire funzioni più complesse e ha consentito una notevole diminuzione dei costi per la realizzazione del veicolo stesso. Ad oggi, oltre all'utilizzo di un gateway centralizzato, si sono iniziati a fondere i domini applicativi, riducendo notevolmente l'hardware dedicato ad una singola funzione elaborativa. Come mostra il grafico, nel 2020 i veicoli di nuova produzione devono soddisfare i requisiti fondamentali di performance, flessibilità e connettività. I requisiti sopra citati però devono adattarsi a soluzioni pre-esistenti e devono rispettare i requisiti di sicurezza sempre più stingenti come lo standard ISO26262.

Prima dell'architettura AUTOSAR, come si evince dall Fig. 4.2, non esisteva un netto confine tra i componenti hardware e quelli software. Ciò rendeva difficile integrare nuove funzionalità e modificare quelle pre-esistenti.

AUTOSAR ha portato un cambiamento in questo schema, introducendo un sistema a strati:

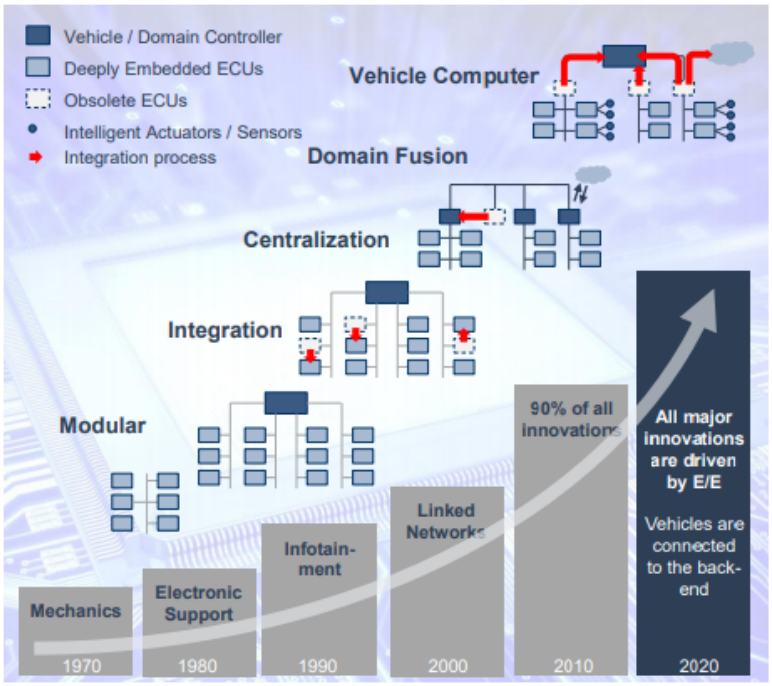

Figura 4.1: Cambiamenti nei sistemi E/E

1. Application Software: sono i moduli software custom scritti indipendentemente dall'hardware.
2. AUTOSAR: fornisce un' interfaccia in grado di comunicare con tutti i moduli software sopra descritti. E' lo strato che mette in comunicazione hardware e software.
3. Hardware: rappresenta i componenti hardware presenti all'interno del veicolo, come le varie ECU.

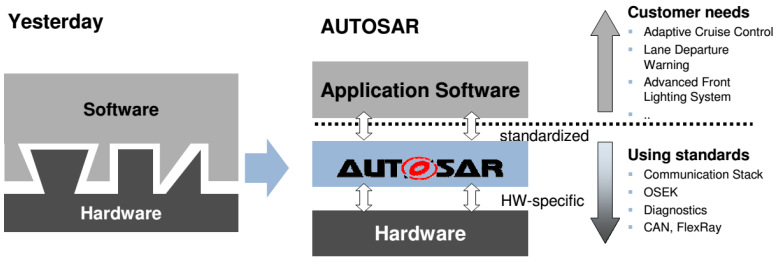

Figura 4.2: Yesterday vs Today

4.2 Layer Architecture

Come scritto prima, un tipico software basato su AUTOSAR in esecuzione su una ECU, è composto da strati, uno schema più completo degli strati è mostrato in Fig. 4.3. Fondamentalmente, ci sono tre livelli principali: *Software Components*, (SW-

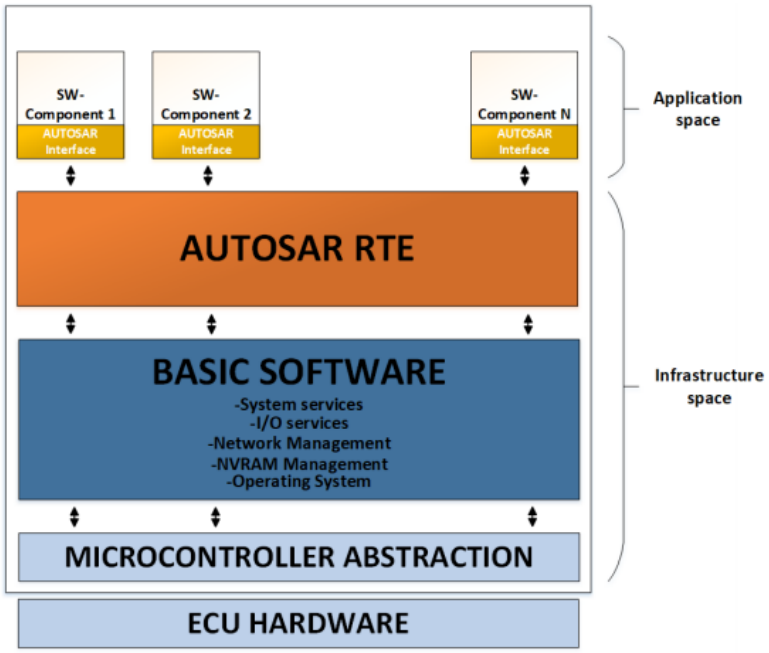

Figura 4.3: Organizzazione a livelli AUTOSAR

C), *AUTOSAR RTE* e *Basic Software*, (BSW). Possiamo individuare un'ulteriore stratificazione, considerando anche la separazione tra lo spazio *Applicazione* e lo spazio *Infrastruttura*. Il primo contiene le applicazioni SW (livello più in alto), mentre il secondo include gli strati rimanenti (livelli di gestione dell'hardware, I/O).

4.2.1 Software Component (SW-C)

Lo spazio applicativo è costituito da SW-C interconnessi, che rappresentano il SW applicativo, dotato di interfaccia AUTOSAR e descritto da un modello di componente software (SW-C), [17]. Un SW-C incapsula parte delle funzionalità dell'applicazione, pertanto potremmo avere anche più di un SW-C per realizzare l'applicazione SW complessiva. Un SW-C ha porte ben definite, attraverso le quali può interagire con altri componenti, attraversando lo strato AUTOSAR RTE, [18].

Le porte hanno la funzione di I/O rispetto il componente che si sta progettando. La struttura di questi componenti è di tipo modulare, così che esso può essere

impiegato più volte all'interno della stessa architettura. In Fig. **??** è riportato un esempio di architettura modulare tra componenti SW-C. Lo schema riporta la modellazione di un protocollo di comunicazione di tipo client-server. Le porte che possono essere utilizzate sono formalizzate e ciascuna di essa può adempiere ad un compito predefinito. Nell'esempio riportato, le porte sono di tipo Sender/Receiver.

4.2.2 AUTOSAR RTE

Questo strato, si basa su un concetto più astratto che prende il nome di *Virtual Functional Bus (VFB)*, [19]. Questo bus virtuale, senza entrare nel dettaglio teorico, fornisce uno spazio di comunicazione tra i vari SW-C. Il *Virtual Functional Bus* fornisce, inoltre, un meccanismo per descrivere i SW-C in modo indipendente dall'hardware rendendo questi più facilmente ricollocabili.

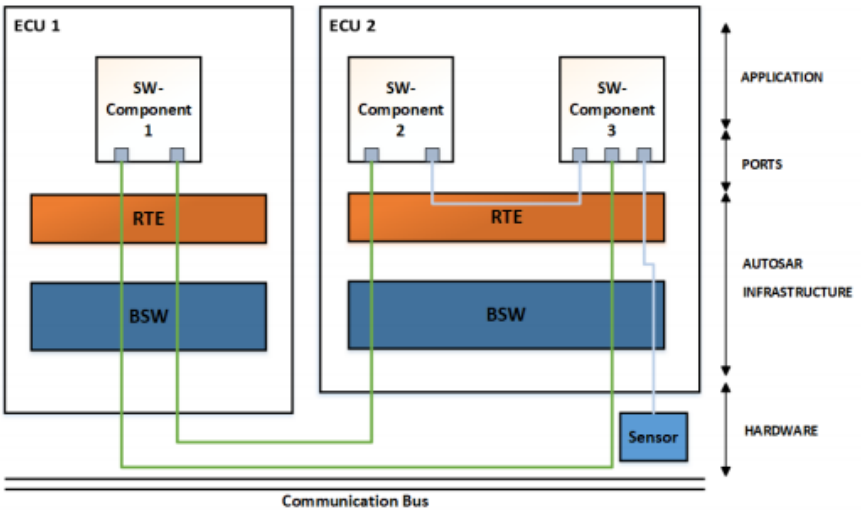

Figura 4.4: Esempio di comunicazione tra ECU e SW-C

AUTOSAR Runtime Enviroment (RTE), è una specializzazione del VFB per una singola ECU. L'RTE ha il compito di fornire un'interfaccia AUTOSAR comune per tutti i SW-C che saranno riferiti all'ECU considerata, [20].

In Fig. 4.4 è schematizzato il funzionamento del meccanismo di comunicazione tra due ECU, le quali singolarmente implementano la stratificazione di Fig. 4.3. Da notare come uno scambio di messaggi tra due componenti SW-C della stessa ECU, non necessità un passaggio all'interno del bus fisico di comunicazione (ad esempio CAN), ma resta confinato all'interno del runtime enviroment della stessa centralina.

4.2.3 Basic Software (BSW)

Il software di base è responsabile della gestione delle funzionalità di base della singola ECU. Questo software non comprende le funzionalità applicative, che sono implementate all'interno di SW-C. Alcune funzionalità di BSW sono: gestione I/O, System Service, gestione dello scheduling ecc. In pratica BSW gestisce la comunicazione e l'interfacciamento tra il software di basso livello e quello di alto livello in cui si "appoggia" il livello RTE AUTOSAR.

4.3 AUTOSAR Blockset per il MBD

Sfruttare la metodologia del MBD, descritta nei capitoli precedenti, per la progettazione di componenti software compatibili con AUTOSAR, porta non solo ai ben noti vantaggi del MBD, specie ad un notevole risparmio di tempo grazie alla generazione automatica di codice, ma anche alla relativa semplicità di applicazione delle regole AUTOSAR, [21]. Componenti software AUTOSAR, consistono non solo nel codice conforme allo standard, che è fondamentalmente un codice C con macro RTE speciali per consentire la comunicazione tra SW-C, ma ogni AUTOSAR SW-C comporta anche un cosiddetto software standardizzato di descrizione del componente che elenca tutte le caratteristiche del componente in esame. Questa descrizione serve per integrare la componente software in un applicazione AUTOSAR più grande.

AUTOSAR Blockset fornisce blocchi e costrutti per le routine delle librerie AUTOSAR e per i servizi software di base (BSW), tra cui NVRAM e diagnostica. Simulando i servizi BSW insieme al modello del software dell'applicativo, è possibile verificare il software AUTOSAR ECU senza uscire da Simulink. AUTOSAR Blockset consente di creare modelli di architettura AUTOSAR in Simulink o di importarne di pre-esistenti sfruttando il file che descrive la struttura, generalmente in formato ARXML. Dai modelli creati è possibile generare automaticamente codice conforme allo standard.

In Fig. 4.5 è riportato un esempio in cui, grazie ad AUTOSAR Blockset, è stato generato il layer di Basic Software (BSW) per simulare il comportamento del modello, realizzato per essere coerente con lo standard, direttamente da Simulink.

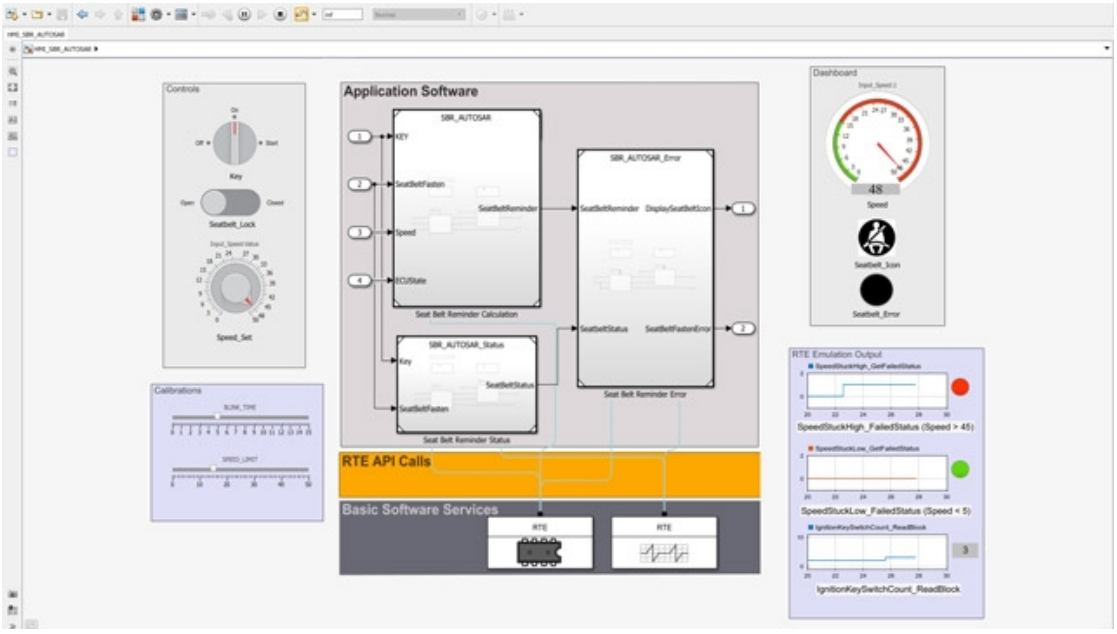

Figura 4.5: Simulazione di una ECU con AUTOSAR Blockset

Capitolo 5

Conclusioni

Obiettivo di questo elaborato era quello di progettare un supervisore logico per un on-board charger di un generico veicolo elettrico. La progettazione e implementazione di quest'ultimo ha sfruttato i concetti teorici della metodologia del Model Based Design, seguendo step-by-step il ciclo a V dello sviluppo software. La prima fase ha permesso, tramite una trattazione puramente teorica, di comprendere a pieno i principi cardini della metodologia utilizzata e di conseguenza di rendere più fluida la parte di progettazione e sviluppo del caso di studio preso in considerazione. Durante questa seconda fase, i vantaggi illustrati nelle parti introduttive di questo lavoro di tesi sono emersi e hanno permesso la realizzazione, in modo semplice ed efficiente, di un modello che rispettasse i requisiti iniziali. In particolar modo, tramite l'utilizzo dei soli tool forniti da MathWorks, l'esecuzione step-by-step dello schema a V è stata facile e intuitiva. I meccanismi di authoring e traceability dei requisiti hanno consentito di effettuare modifiche e valutazioni del progetto in corso d'opera, senza influenzare la pianificazione di tutto il ciclo di sviluppo. Questo importante vantaggio, portato dalla metodologia del Model Based Design, consente una gestione intelligente dei requisiti, delle relative implementazioni e dei test cases per la verifica e validazione di ciò che è stato realizzato, nonostante le dimensioni importanti del progetto in esame. Un altro vantaggio riscontrato durante le fasi finali del ciclo a V è stata la possibilità di generare in modo automatico codice C direttamente dal modello realizzato. La generazione automatica di codice, ha ridotto drasticamente i tempi di sviluppo dell'intero progetto, rendendo disponibili più spazi temporali dedicabili alle fasi di test. La metodologia utilizzata ha fornito, in merito alle fasi di test delle componenti progettate, la possibilità di verificare a partire dal modello il soddisfacimento dei requisiti, senza dover attendere la realizzazione del codice sorgente. Questo aspetto è l'ulteriore vantaggio riscontrato nel Model Based Design e ha consentito la risoluzione di problematiche ed errori, in anticipo rispetto alla prototipazione del codice sorgente sul micro-controllore.

Il soddisfacimento delle specifiche iniziali e i risultati positivi ottenuti durante

la fase di test di tipo Processor-In-the-Loop, hanno dimostrato che il prodotto ottenuto da questo elaborato può essere considerato a tutti gli effetti coerente con le aspettative. Negli ultimi capitoli sono stati illustrati i principi alla base di alcune norme presenti nel settore automotive: ISO 26262 e AUTOSAR. In questo lavoro non si sono applicate le suddette norme al caso di studio in esame. Un possibile sviluppo futuro potrebbe essere l'implementazione del medesimo modello secondo l'architettura AUTOSAR, andando dunque a generare un componente software (SW-C) compatibile con il layer di interfaccia utilizzato nello standard. Un ulteriore step, potrebbe essere quello di verificare che il componente software implementato e tutto il processo di sviluppo che ha portato ad esso, sia compatibile con le norme riportate all'interno dello standard ISO 26262. Se queste due ulteriori strade venissero percorse, si otterrebbe un componente software perfettamente integrabile all'interno di un'autovettura, realizzato in tempi ridotti rispetto ad una programmazione di tipo tradizionale e più robusto dal punto di vista strutturale.

Appendice A

Two Stage Charger

La seguente appendice analizza i componenti presenti all'interno di un carica batterie a due stadi, [22]. Per ogni parte verrà proposta una formulazione circuitale semplificata con lo scopo di illustrare i principi fisici alla base del funzionamento del componente.

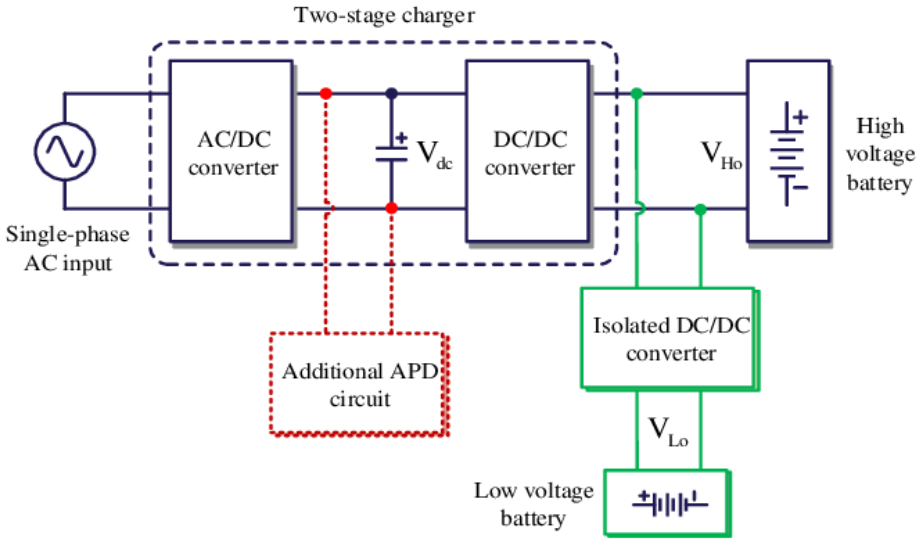

Figura A.1: Two-stage charger

Come riportato in Fig. A.1, un carica batterie a due stadi è composto da due componenti principali:

1. AC/DC converter;
2. DC/DC converter.

A.1 AC/DC converter: raddrizzatori

Il raddrizzatore è un componente che permette di convertire un segnale alternato in un segnale continuo. Il funzionamento di questo componente si basa principalmente sul comportamento del diodo. Quando il diodo viene polarizzato direttamente, ovvero tensione ai suoi capi positiva, esso viene posto in conduzione. Il caso opposto, ovvero se la tensione ai capi del diodo è negativa, è quando il diodo è in interdizione ovvero presenta una resistenza infinita dunque non scorre corrente su di esso. Il comportamento sopra citato è valido per un diodo ideale.
Di seguito sono riportate tre configurazioni che permetto di ottenere, in modo incrementale, risultati migliori.

Raddrizzatore a singola semionda

Nella prima parte del circuito è rappresentato un trasformatore, esso serve ad adattare il valore di tensione disponibile sulla rete elettrica col valore desiderato in uscita dal circuito. La tensione V1 in uscita dal trasformatore è di tipo alternato. Quando in uscita dal trasformatore è presente una semionda positiva, il diodo risulta essere polarizzato direttamente, e quindi si ha una circolazione di corrente che dal trasformatore mediante il diodo raggiunge la resistenza di carico. Invertendosi poi la polarità della tensione in uscita dal trasformatore, il diodo risulta essere polarizzato inversamente e quindi non lascia circolare corrente (la sua resistenza è infinita), e sulla resistenza di carico non circola corrente. Se si mettono in successione queste due considerazioni si nota che sulla resistenza di carico circola corrente solo quando il diodo è polarizzato direttamente, e la tensione ai capi della resistenza assume l' andamento della V2, cioè una corrente pulsante.

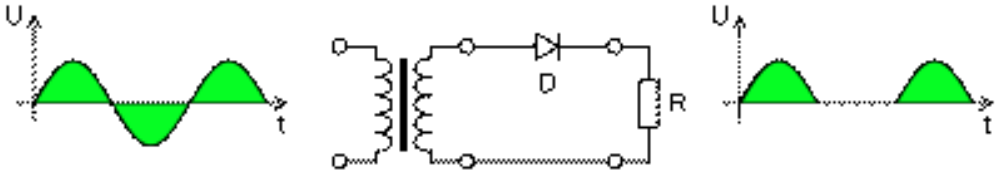

Figura A.2: Raddrizzatore a singola semionda

Raddrizzatore a doppia semionda

Il raddrizzatore a semionda non è certo il tipo migliore tra quelli realizzabili, difatti utilizzando un trasformatore leggermente diverso ed aggiungendo un diodo si ottiene un effetto migliore. Il trasformatore ha la particolarità di avere due avvolgimenti secondari uguali, collegati tra di loro dal morsetto centrale (morsetto di zero), la caratteristica di questo trasformatore, è quella di dare due tensioni in

uscita completamente uguali di valore, ma sfasate tra di loro di 180 gradi. Sulla resistenza di carico giungono due semionde, la prima arriva dal diodo D1 che conduce quando è polarizzato direttamente, contemporaneamente il diodo D2 è polarizzato inversamente e non conduce e viceversa. Questo raddrizzatore risulta migliore dei quello a singola semionda, in quanto al carico giunge una tensione con valore medio maggiore.

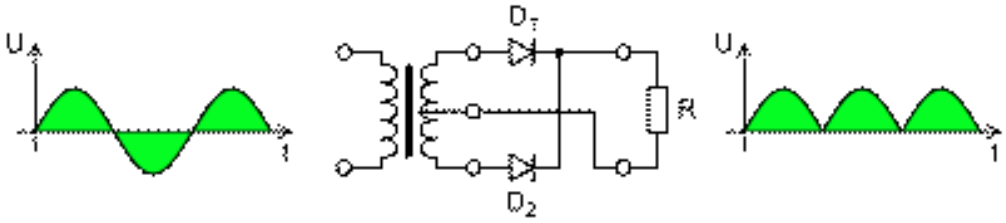

Figura A.3: Raddrizzatore a doppia semionda

Ponte di Graetz

Esso è costituito da 4 diodi opportunamente collegati e non necessita di un trasformatore dotato di morsetto zero. Per comprendere il funzionamento del ponte si considerano due fasi: la prima è quando la tensione positiva è nella parte alta del trasformatore. In questo caso il primo e l'ultimo diodo sono in conduzione. Nel secondo caso, i diodi in conduzione sono quelli centrali. Con questo meccanismo di alternanza si ottengono delle tensioni pulsanti come per il raddrizzatore a doppia semionda. Principale svantaggio di questo metodo è di avere una caduta di tensione pari a quella di due diodi in serie, quindi di circa 1,4 volt e dunque nel raddrizzare tensioni molto piccole si ha una perdita eccessiva.

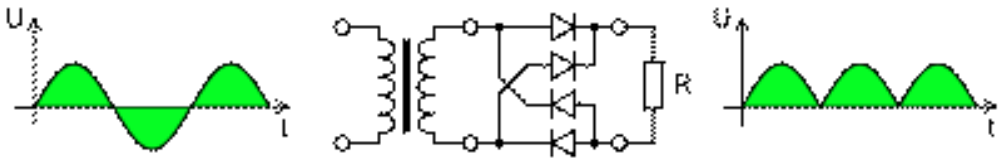

Figura A.4: Ponte di Graetz

A.2 DC/DC converter: chopper

Sono convertitori di potenza in grado di aumentare o diminuire il valore di una corrente continua in un altro. All'interno dei veicoli elettrici sono presenti tanti componenti e ciascuno di essi necessita di un'alimentazione diversa. Per rendere

ciò possibile è necessario modificare la tensione continua fornita dal pacco batteria in modo da alimentare con il giusto valore le singole parti.
I chopper possono essere di due tipologie:

Booster sono dispositivi in grado di aumentare il valore in input. Il principio base di funzionamento di un convertitore boost, [23], consiste in due stati distinti:

1. Nello stato "on", il commutatore S è chiuso, provocando un aumento di corrente nell'induttore;
2. Nello stato "off", il commutatore è aperto e l'unico percorso offerto alla corrente dell'induttore è attraverso il diodo D, la capacità C e il carico R. Ciò provoca il trasferimento dell'energia accumulata durante lo stato "on" dall'induttore verso la capacita'.

Buck sono dispositivi in grado di diminuire il valore in input. Anche questa tipologia di convertitori fa uso di un commutatore (in entrambi i casi è raffigurato un mosfet) che opportunatamente comandato da una sorgente PWM (Pulse Width Modulation) riesce a deviare il flusso di corrente. Nel caso del buck, si hanno due fasi:

1. Nello stato "on", il commutatore S è chiuso, provocando un aumento di corrente nell'induttore e la polarizzazione inversa del diodo. La corrente circola in L, C e sul carico;
2. Nello stato "off", il commutatore è aperto e poichè la corrente sull'induttore non può variare rapidamente (se accadesse la tensione ai capi dell'induttore sarebbe elevata e si rischierebbe la rottura dello stesso) il diodo viene ad essere polarizzato direttamente (diodo di fly-back) e permette di trasferire tutta l'energia accumulata nell'induttore verso il condensatore perciò la corrente sull'induttore decresce.

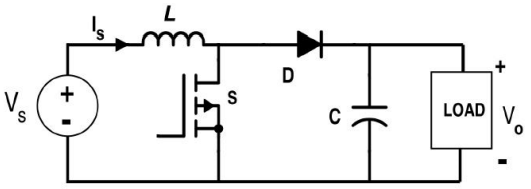

Figura A.5: Boost

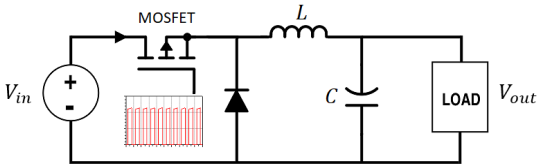

Figura A.6: Buck

Appendice B

Batteria al litio: modellazione matematica

La seguente appendice vuole illustrare il modello circuitale equivalente per una batteria al litio e le principali tecniche di determinazione dello State Of Charge (SOC)

Le batterie agli ioni di litio sono un tipo di accumulatore ricaricabile, utilizzati in diversi ambiti come l'elettronica, veicoli elettrici, industria ecc.

Esse usano un composto di litio sul catodo e grafite o titanato di litio sull'anodo. La tecnologia basata sul litio può costituire un pericolo per la sicurezza, poiché le batteria contengono un elettrolita infiammabile e se danneggiate o caricate in modo errato possono provocare esplosioni e incendi.

B.0.1 Modello circuitale equivalente

Visto l'approccio seguito in questo lavoro di tesi, è utile introdurre un modello circuitale equivalente in grado di simulare a pieno il comportamento tempo variante di un accumulatore al litio. Questi vengono modellati con dei semplici componenti elettronici, in particolare una cella al litio è composta da un condensatore e da una resistenza parallela ad esso, [24]. In Fig. B.1 è riportato il circuito equivalente di una batteria con tre diverse costanti di tempo, resistenza interna e potenziale a circuito aperto. I parametri del circuito vanno scelti in modo tale da riflettere il comportamento non lineare della batteria e le dipendenze su temperatura, SOC, SOH e corrente. Tali dipendenze sono proprie della chimica di ogni batteria e devono essere determinate utilizzando le misurazioni eseguite su celle della batteria dello stesso tipo di quelle per le quali il controller è stato progettato, [25]. Il circuito è stato realizzato con le librerie SimScape di Simulink.

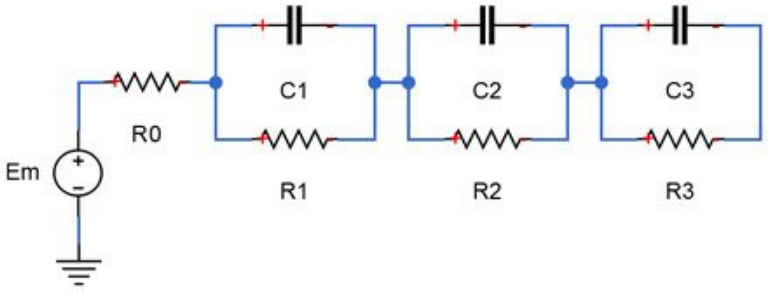

Figura B.1: Circuito equivalente di una batteria al litio

B.0.2 Curve di carica

Per poter testare il comportamento della logica di supervisione dell'OBC realizzata a partire dai requisiti, è stato necessario utilizzare una curva di carica/scarica, come quella proposta in Fig. B.2. La curva rappresenta nell'asse delle ascisse lo State of Charge (SOC) e nell'asse delle ordinate la tensione delle celle. Nella curva considerata i valori numerici presi in considerazione sono puramente esemplificativi, infatti nei requisiti non sono presenti specifiche riguardo alla tipologia di batteria e di conseguenza non vi sono informazioni relative alle tensioni e correnti nominali.

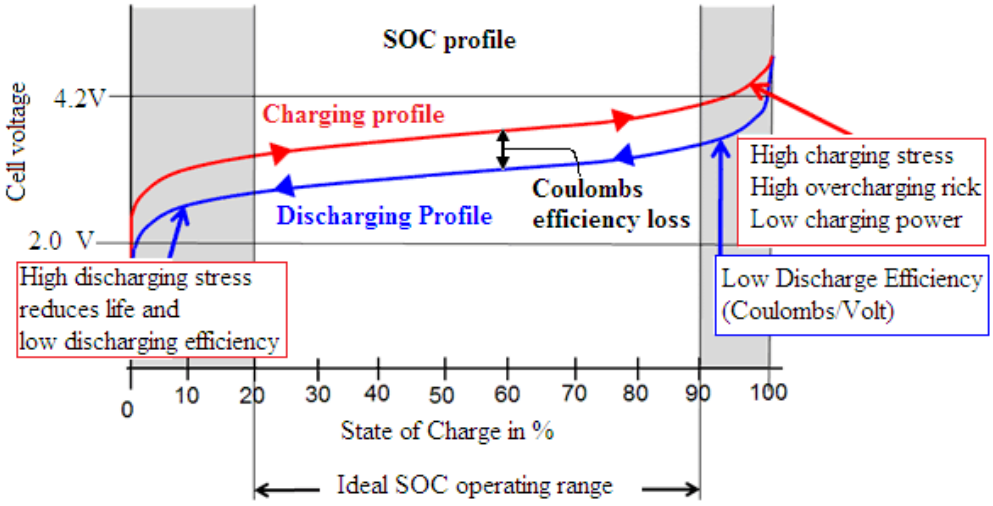

Figura B.2: Curva SOC - Volt

Nella curva B.2 si notano tre zone di lavoro. La prima, valori di SOC bassi, in cui si ha un forte stress nella fase di scarica il quale riduce la vita utile della batteria. La parte centrale presenta una crescita più lenta delle tensioni a fronte di un aumento significativo del SOC. Questa zona è quella in cui la batteria dovrebbe

lavorare poiché la tensione di cella presenta basse variazioni e dunque permette al carico di essere alimentato con valori di tensione costanti. La terza parte del grafico, quella con un SOC da 90-100%, presenta un forte stress durante la fase di carica, infatti la tensione sale oltre quella nominale. Da notare come la curva di carica, rispetto a quella di scarica, può erogare al carico una tensione maggiore a parità di State of Charge, [26]. Un esempio banale del comportamento sopra detto è verificabile durante l'utilizzo di un computer portatile: durante la fase di carica le prestazioni massime che esso raggiungerà saranno maggiori rispetto a quelle che potrebbe raggiungere se sconnesso dalla rete di alimentazione.

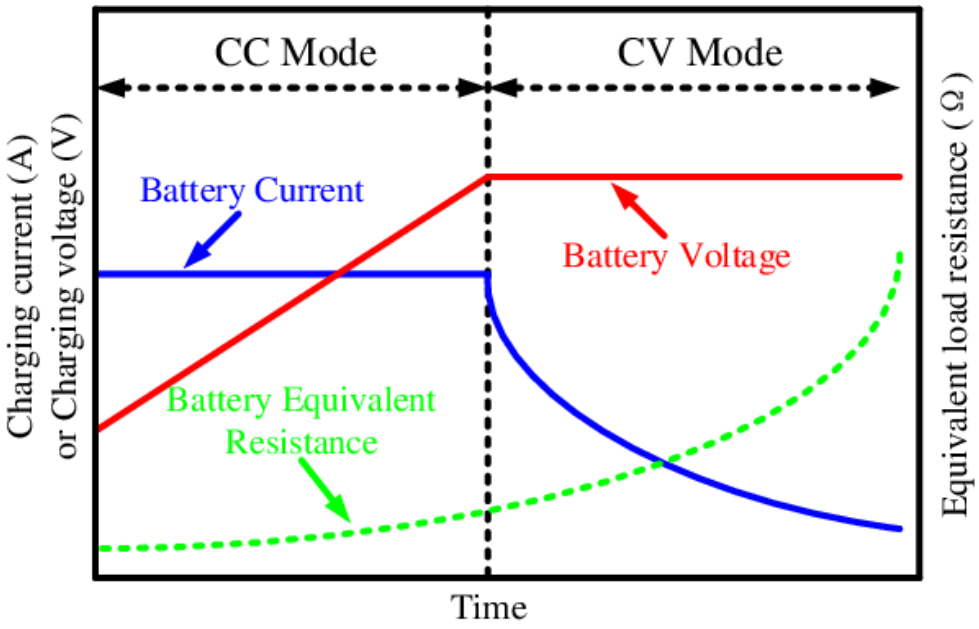

Figura B.3: Profilo di carica CC-CV

In Fig. B.3 è raffigurato il profilo di carica, [27]. Generalmente per batterie al litio viene utilizzato un algoritmo di tipo CC-CV: una prima fase di carica mantiene costante la corrente (fase CC) di batteria e aumenta la tensione in modo proporzionale, una seconda fase (fase CV) mantiene la tensione di batteria costante e fa diminuire in modo esponenziale la corrente di batteria. Durante l'intero processo di carica la resistenza equivalente della batteria cresce in modo proporzionale con lo State of Charge.

B.0.3 Stima dello stato di carica

Uno degli indicatori maggiormente utilizzato per la gestione delle batterie in qualunque ambito applicativo è lo State of Charge (SOC). Questo indica la percentuale

di energia residua contenuta all'interno dell'accumulatore. Nel caso in esame, ovvero considerando delle batterie al litio, l'energia viene immagazzinata tramite un processo chimico, questo non permette una misura diretta del SOC. Per tale ragione esso deve essere stimato a partire da altri indicatori.
La stima del SOC è un problema di rilevata importanza, proprio per il ruolo che lo State of Charge ha su tutti i componenti di gestione del sistema completo, come il Battery Management System o l'On Board Charger. Se specializziamo il problema in ambito automotive, più precisamente considerando veicoli elettrici o ibridi, lo State of Charge è l'indicatore principale che va a stimare il range ovvero l'autonomia, in termini chilometrici, che il veicolo possiede.
Abbiamo più metodologie per ottenere uno stima del SOC, [28]. Quella che permette di percepire meglio il funzionamento dell'indicatore è il metodo *Coulomb Counting Method*. Tale metodo è il più facile da implementare ma è il più soggetto a distorsione dei risultati nel momento il cui il SOC iniziale è sconosciuto o se le batterie in esame presentano un grosso fattore di auto-scarica.
Il metodo si basa su una semplice formula matematica, (B.1):

$$SOC = SOC(t_0) + \frac{1}{C_{rated}} \int_{t_0}^{t_0+\tau} (I_b - I_{loss})dt \tag{B.1}$$

in cui $SOC(t_0)$ è il SOC iniziale, C_{rated} è la capacità della cella, I_b è la corrente che scorre nella cella e I_{loss} è la corrente dispersa grazie alle perdite. Il metodo prende il nome di *Coulomb Counting Method* proprio perché si basa sul conteggio della quantità di carica che scorre attraverso le celle. Si parla anche di *conteggio ampere-ora*. L'accuratezza si basa principalmente su una misurazione precisa della corrente di batteria e stima accurata del SOC iniziale.

Con una capacità iniziale preconosciuta, che potrebbe essere memorizzata o inizialmente stimata dalle condizioni operative, è possibile calcolare il SOC di una batteria integrando le correnti di carica e scarica durante il periodo di funzionamento. Tuttavia, la carica rilasciabile è sempre inferiore a quella immagazzinata nel ciclo di carica e scarica precedente. In altre parole, ci sono perdite durante la carica e la scarica. Queste perdite, oltre a quelle dovute al fenomeno dell'auto-scarica, causano errori di accumulo in quanto la vita utile della batteria diminuisce. Per una stima più precisa, questi fattori dovrebbero essere presi in considerazione. Inoltre, il SOC dovrebbe essere ricalibrato regolarmente e la declinazione del la capacità rilasciabile dovrebbe essere considerata ad ogni ciclo di carica/scarica per una stima più precisa.

Appendice C

Codice per la gestione della comunicazione seriale

```
int processData(uint8_t p_data){
    static int fsmState = STATE_HEADER;
    static int j = 0;                   //Number of float
    static int i = 0;                   //Number of bytes
    static int count = 0;
    int retVal = 0;

    switch(fsmState){

    case STATE_HEADER:
        if (p_data == HEADER_VALUE){
            fsmState = STATE_READ_PAYLOAD;
            retVal = 1;
        }else{
            fsmState = STATE_HEADER;
            retVal = -1;
        }
        count=0;
        i=0;
        j=0;
        break;

    case STATE_READ_PAYLOAD:
        arrayPayload_tmp[j].bytes[i] = p_data;
        count++;
        if(count == PAYLOAD_SIZE){
            fsmState = STATE_TERMINATOR;
        }else{
```

```
29              if(i == (PAYLOAD_SIZE/BYTE_NUMBER - 1)){
30                  j++;
31                  i=0;
32              }else{
33                  i++;
34              }
35          }
36          retVal = 1;
37          break;
38
39      case STATE_TERMINATOR:
40          if (p_data == TERM_VALUE){
41              fsmState = STATE_HEADER;
42              memcpy(&arrayPayload , &arrayPayload_tmp, sizeof(
    arrayPayload_tmp));
43              retVal = 1;
44          }
45          else{
46              fsmState = STATE_HEADER;
47              retVal = -1;
48          }
49          break;
50
51      default:
52          break;
53      }
54      return(retVal);
55  }
```

Bibliografia

[1] MathWorks. «BAE Systems Achieves 80% Reduction in Software-Defined Radio Development Time with Model-Based Design». In: *MathWorks* (2020) (cit. alle pp. 2, 4).

[2] J. Krasner. «"Comparing embedded design outcomes with and without model based design"». In: *EMBEDDED MARKET FORECASTERS* (2011) (cit. alle pp. 4, 5).

[3] H. Mooz K. Forsberg. «"The Relationship of System Engineering to the Project Cycle"». In: *Proceedings of the First Annual Symposium of National Council on System Engineering* (1991) (cit. a p. 7).

[4] B. Marick. «"New Models for Test Development"». In: *Testing Foundations* (2000) (cit. a p. 7).

[5] L. Semini G.A. Cignoni C. Montangero. «"Il controllo del software: verifica e validazione"». In: *Corso di Ingegneria del software, Dipartimento di Informatica, Università degli Studi di Pisa* (2009) (cit. a p. 9).

[6] Alceu Farias, Reurison Rodrigues, Andre Murilo, Renato Lopes e Suzana Avila. «Low-Cost Hardware-in-the-Loop Platform for Embedded Control Strategies Simulation». In: *IEEE Access* PP (ago. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2934420 (cit. a p. 18).

[7] Alireza Khaligh e Michael D'Antonio. «Global Trends in High-Power On-Board Chargers for Electric Vehicles». In: *IEEE Transactions on Vehicular Technology* 68.4 (2019), pp. 3306–3324. DOI: 10.1109/TVT.2019.2897050 (cit. a p. 30).

[8] Deepak Ronanki, Apoorva Kelkar e Sheldon S. Williamson. «Extreme Fast Charging Technology—Prospects to Enhance Sustainable Electric Transportation». In: *Energies* 12.19 (set. 2019), pp. 1–17 (cit. a p. 31).

[9] J. Chatzakis, K. Kalaitzakis, N.C. Voulgaris e S.N. Manias. «Designing a new generalized battery management system». In: *IEEE Transactions on Industrial Electronics* 50.5 (2003), pp. 990–999. DOI: 10.1109/TIE.2003.817706 (cit. a p. 32).

[10] Thomas J McCabe. «A complexity measure». In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. a p. 47).

[11] Organización Internacional de Normalización. *ISO 26262: Road Vehicles : Functional Safety*. ISO, 2011. URL: https://books.google.it/books?id=3gcAjwEACAAJ (cit. a p. 65).

[12] MathWorks Tom Erkkinen. *How to Use Simulink for ISO 26262 Projects*. URL: https://it.mathworks.com/company/newsletters/articles/how-to-use-simulink-for-iso-26262-projects.html (cit. a p. 68).

[13] Organización Internacional de Normalización. *ISO 26262-6 Part 6: Product development at the software level*. ISO, 2011 (cit. a p. 69).

[14] Jean-Louis Boulanger. *Certifiable Software Applications 2: Support Processes*. Elsevier, 2016 (cit. a p. 71).

[15] BTC Embedded System Markus Gros. *When and how to qualify tools according to ISO 26262*. URL: https://www.btc-es.de/en/blog/when-and-how-to-qualify-tools-according-to-iso-26262.html (cit. a p. 71).

[16] AUTOSAR.org. *Automotive software and electrical/electronic architecture: Implications for OEMs*. URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/automotive-software-and-electrical-electronic-architecture-implications-for-oems (cit. a p. 74).

[17] AUTOSAR.org. *SW-C and System Modeling Guide*. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TR_SWCModelingGuide.pdf (cit. a p. 76).

[18] AUTOSAR.org. *Application Interface*. URL: https://www.autosar.org/standards/application-interface/ (cit. a p. 76).

[19] AUTOSAR.org. *Specification of the Virtual Functional Bus*. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_SWS_VFB.pdf (cit. a p. 77).

[20] AUTOSAR.org. *Technical Overview*. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/3-0/AUTOSAR_TechnicalOverview.pdf (cit. a p. 77).

[21] Ulrich Eisemann. «Model-Based Design for AUTOSAR Software Components». In: *Embedded Real Time Software and Systems (ERTS2008)*. 2008 (cit. a p. 78).

[22] Vu Nguyen, Dinh-Du To e Dong-Choon Lee. «Onboard Battery Chargers for Plug-in Electric Vehicles With Dual Functional Circuit for Low-Voltage Battery Charging and Active Power Decoupling». In: *IEEE Access* 6 (ott. 2018), pp. 70212–70222. DOI: 10.1109/ACCESS.2018.2876645 (cit. a p. 82).

[23] Saurabh Kumar, Rajat Kumar e Dr. Navdeep Singh. «Performance of closed loop SEPIC converter with DC-DC converter for solar energy system». In: mar. 2017. DOI: 10.1109/ICPCES.2017.8117668 (cit. a p. 85).

[24] Tarun Huria, Massimo Ceraolo, Javier Gazzarri e Robyn Jackey. «High fidelity electrical model with thermal dependence for characterization and simulation of high power lithium battery cells». In: *Electric Vehicle Conference (IEVC), 2012 IEEE International* (mar. 2012). DOI: 10.1109/IEVC.2012.6183271 (cit. a p. 86).

[25] Tarun Huria, Robyn Jackey, Javier Gazzarri, Michael Saginaw e Pravesh Sanghvi. «Battery Model Parameter Estimation Using a Layered Technique: An Example Using a Lithium Iron Phosphate Cell». In: vol. 2. Apr. 2013. DOI: 10.4271/2013-01-1547 (cit. a p. 86).

[26] C. Bharatiraja, P. Sanjeevikumar, Pierluigi Siano, K. Ramesh e Raghu Selvaraj. «Real Time Foresting of EV Charging Station Scheduling for Smart Energy System». In: *Energies* 10 (mar. 2017). DOI: 10.3390/en10030377 (cit. a p. 88).

[27] Yafei Chen, Hailong Zhang, Sung-Jun Park e Dong-Hee Kim. «A Switching Hybrid LCC-S Compensation Topology for Constant Current/Voltage EV Wireless Charging». In: *IEEE Access* PP (set. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2941652 (cit. a p. 88).

[28] Martin Murnane e Adel Ghazel. «A closer look at state of charge (SOC) and state of health (SOH) estimation techniques for batteries». In: () (cit. a p. 89).