ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INGEGNERIA MECCANICA

INGEGNERIZZAZIONE DEL MOTOVEICOLO M

TESI DI LAUREA

in
Motori a combustione interna e propulsori ibridi

Re-ingegnerizzazione di un modello di controllo per pompa ad alta pressione per motori GDI

CANDIDATO:
Riccardo Albanese

RELATORE:
Chiar.mo Prof. Nicolo' Cavina

CORRELATORI:
Filippo Cavanna
Gaetano Divieste
Leonardo Pecile
Prof. Enrico Corti
Prof. Davide Moro

Anno Accademico 2019/20

Sessione III

ABSTRACT

Il lavoro di tesi è basato su una ricerca esplorativa riguardo gli effetti che si hanno nel cambio di approccio di modellazione per un modello in SIMULINK passando da una gestione dei calcoli in una configurazione a Stati a una configurazione a Runnable. L'interesse nell'analisi di questo nuovo approccio è determinato dalla necessità di rendere i modelli di controllo più adatti ad uno sviluppo SW conforme alle specifiche architetturiali espresse secondo il formalismo AUTOSAR. Il tipo di modellazione che si vuole ottenere è con una forma dove tutti i calcoli relativi a un singolo task siano svincolati dai calcoli relativi ad altri task.

Attualmente, i modelli sono divisi nelle loro logiche interne per "scopi", in ogni blocco si hanno tutte le logiche relative a uno scopo (esempio il calcolo degli angoli di attivazione, la definizione dei parametri di esecuzione, etc.). Questi blocchi contengono tutte le logiche di calcolo necessarie allo scopo e vengono attivati totalmente o in parte secondo l'evento chiamante. Con il nuovo approccio di modellazione si intende svincolare le logiche di calcolo tra loro in modo da ottenere all'interno dello stesso modello dei blocchi che eseguono totalmente la catena di calcolo relativa a un singolo evento (con terminologia AUTOSAR i blocchi che rappresentano le diverse Runnable).

L'obiettivo, quindi, consiste nell'analizzare problematiche relative alla modellazione con questo nuovo approccio e fornire tramite dei metodi comparativi delle informazioni qualitative riguardanti il codice e la sua efficienza.

IL CONTROLLO NEI COMPONENTI MECCANICI

Attualmente tutti i sistemi meccanici necessitano di un sistema di controllo al fine di ottenere prestazioni sempre più elevate. Le logiche di controllo impiegate, per poter essere efficienti, richiedono la conoscenza approfondita di ogni componente nelle sue caratteristiche funzionali e costruttive. Nel mercato attuale, l'integrazione fisica e logica dei sistemi elettromeccanici viene raggiunta tramite la cooperazione di più attori nella fase produttiva e di sviluppo.

Per questo motivo è necessario avere un grado di portabilità sempre maggiore a disposizione dei clienti-produttori finali che agevoli il dialogo nella fase di sviluppo e di utilizzo del componente sia per quanto riguarda l'aspetto fisico che l'aspetto logico software.

Nel mondo Automotive produttori e fornitori hanno trovato una sintesi comune con il consorzio AUTOSAR e la definizione di specifiche per la modellazione e la codifica dell'architettura SW. Nell'ultimo decennio, tale standardizzazione si è imposta come una delle principali per lo sviluppo di SW[1].

La scelta di sviluppo SW tramite un approccio model based si è rivelato nel passato come scelta vincente nella progettazione dei componenti SW.

L'interesse di Marelli è quello di poter trovare un percorso di integrazione e dialogo tra lo sviluppo model based e le specifiche AUTOSAR. In questo scenario di evoluzione, è di interesse provare un ulteriore step evolutivo, infatti, con opportuni accorgimenti nella modifica dello sviluppo delle logiche model based è possibile ottenere un'esecuzione del codice più efficiente in un'architettura MultiCore (tipica delle elettroniche più recenti).

[1] SW = Software

IL COMPONENTE

Il modello di controllo che si vuole re-ingegnerizzare è quello di una pompa ad alta pressione a pistone per motori GDI.

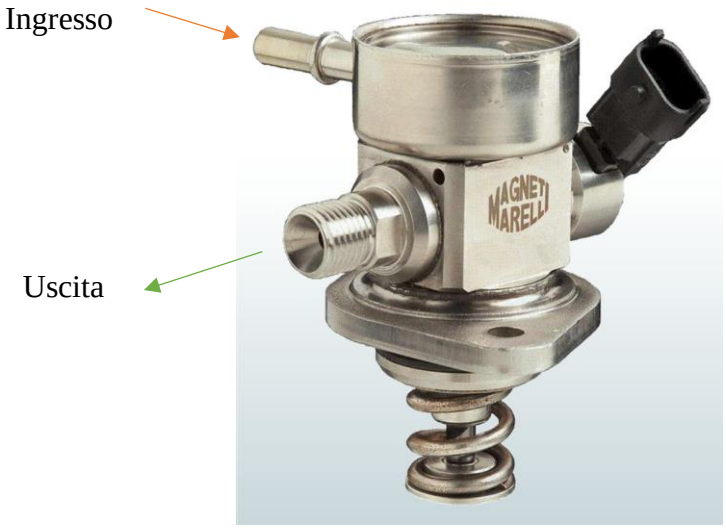


Figura 1 - Pompa alta pressione Magneti Marelli

Le caratteristiche della pompa sono:

- Pressione nominale di utilizzo fino a 200 bar
- Motorizzazioni GDI dove viene impiegata:
  - Applicazioni ad alta potenza (fino a 230 l/h)
  - Elevati regimi di rotazione (fino a 9000 rpm)
  - Qualsiasi tipo di combustibile (anche per motori flex-fuel)

Figura 2- spaccato della pompa

Il piattello inferiore viene azionato da una camma (solitamente con numero di lobi uguale al numero di cilindri), solidale all'albero a camme, che fa muovere il pistone interno per la movimentazione del fluido. Il PMS del pistone della pompa varia rispetto all'angolo dell'albero motore nel caso di motori con sistema VVT a causa del calettamento della camma di comando all'albero a camme.

La valvola di mandata è una valvola normalmente chiusa che si apre per effetto della differente pressione tra la camera e il rail. La presenza di una pressione minima necessaria all'apertura della valvola di mandata impone la determinazione di un angolo minimo di compressione utile all'apertura della valvola. In caso contrario, si avrebbe solo una compressione e successiva espansione del fluido che potrebbe generare cedimenti nel lungo periodo a causa di surriscaldamenti e sollecitazioni indesiderate.

La valvola di ingresso, formata da un disco e una membrana mobile, è controllata tramite uno spillo movimentato da un solenoide (attuatore). La posizione è normalmente aperta e pertanto l'azione dello spillo è quella di rimanere in battuta contro la membrana se non viene azionato elettromagneticamente. Nel caso di attuazione, lo spillo si muove e lascia libera la membrana di muoversi, in questo caso, la membrana è soggetta al flusso che la attraversa: in fase di aspirazione si apre e in fase di compressione si chiude. Una molla garantisce il ritorno dello spillo nella sua posizione originale.

Figura 3 - Dettaglio flusso combustibile

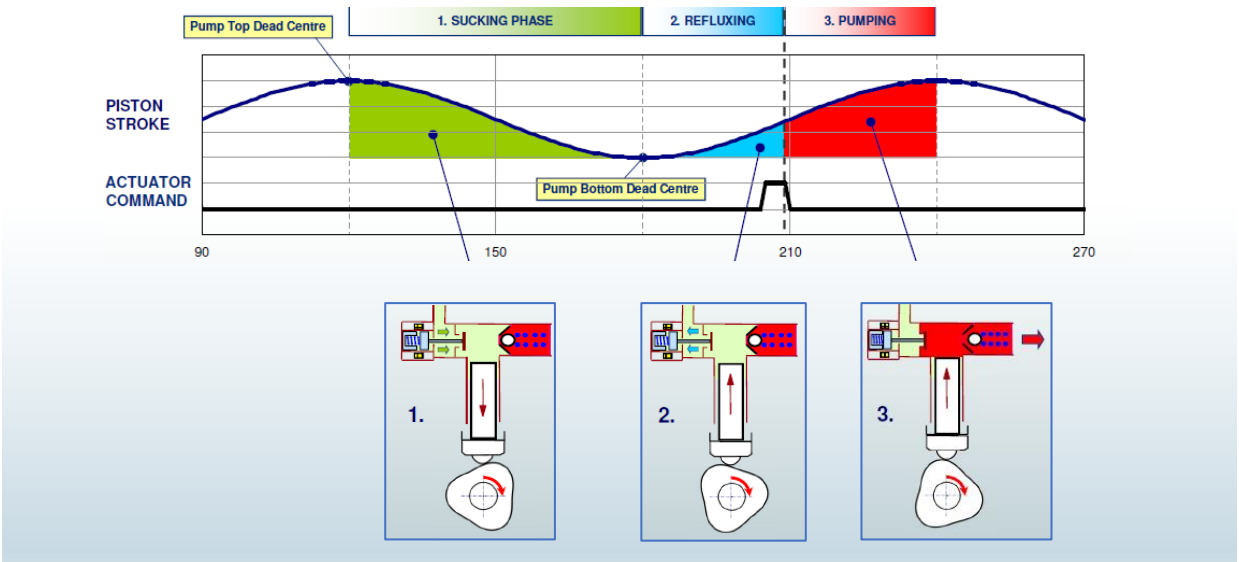


Figura 4-Schema attuazione in funzione dell'angolo dell'albero a camme

Durante la fase discendente del pistone si ha l'ingresso del fluido nella camera della pompa (1 - aspirazione), successivamente, durante il moto di risalita non essendo la valvola di ingresso chiusa si ha l'effetto di reflusso (2). Dopo l'attuazione la valvola si chiude, il fluido viene compresso e arrivato alla pressione adeguata all'apertura della valvola di scarico inizia a defluire (3 – pompaggio). Solitamente il tempo della corsa di salita è superiore del tempo della corsa di discesa per avere il lavoro di compressione distribuito su un angolo maggiore e garantire un funzionamento più affidabile. Una volta azionata la chiusura della valvola non è necessario mantenere attivo l'attuatore per tutta la fase di pompaggio (electrical closing), in quanto è presente un punto dove si riesce a garantire la valvola chiusa per effetto della sola pressione in camera (pressure closing).

Un aspetto importante da considerare invece è il ritardo nell'attuazione (TC, Figura 5) che è dovuto all'intensità della corrente che attraversa l'attuatore e la velocità di rotazione del motore. Tutti i calcoli di attuazione sono in funzione del PMS della pompa. Il momento di chiusura della valvola di aspirazione è determinato dal quantitativo di combustibile che si desidera pompare nel rail, quindi è funzione della portata richiesta.

Di seguito lo schema del comando elettrico (ideale e reale) e della chiusura della valvola rispetto alle fasi della pompa.

Figura 5- Fasi dell'attuazione rispetto al ciclo funzionale

IL COMANDO DI ATTUAZIONE

Il comando di attuazione è di tipo PEAK & HOLD in questo modo si evita il surriscaldamento del componente durante l'esercizio rispetto a un comando di corrente costante per effetto joule e ottengo inoltre un minore consumo energetico.

Figura 6 - Esempio tensione e corrente del comando attuazione

All'inizio dell'attuazione applicando la tensione si ha un incremento di corrente per effetto dell'alimentazione del solenoide, quando lo spillo inizia a muoversi per effetto magnetico si nota una diminuzione della corrente di alimentazione (1). La diminuzione prosegue fino al momento in cui la membrana impatta con il disco e quindi ferma la corsa dello spillo (2). A questo punto l'intensità della corrente ricomincia a salire. La corrente applicata rimane elevata (PEAK) per un tempo necessario a garantire che la movimentazione dello spillo sia completa, successivamente viene mantenuta a un valore di mantenimento minore (HOLD) fino alla disattivazione dell'attuatore. Come già detto precedentemente, la disattivazione non coincide con la fine della pompata ma con il raggiungimento di una pressione interna alla camera che garantisce il mantenimento della valvola chiusa.

Ci sono vari aspetti da considerare per definire il momento e la durata del comando:

- La fase di utilizzo del motore (Cranking, Sincrono) e la quantità di carburante da pompare.
- Il ritardo per effetto tensione batteria → minore è la tensione della batteria più tempo sarà richiesto per completare la movimentazione dello spillo (la curva dell'intensità di corrente è meno ripida).
- Gli effetti del moto del fluido rispetto la membrana → il moto di reflusso diminuisce il tempo necessario alla chiusura della membrana, ad esempio, mentre il moto di aspirazione lo rallenta.

- La temperatura del carburante → in base alla temperatura si hanno differenti valori di viscosità che applicano resistenze differenti al moto della membrana.
- Le caratteristiche meccaniche della pompa.

Tutti questi aspetti incidono sui parametri del comando PEAK & HOLD riassunti nello schema

Figura 7 - Dettaglio fasi comando PEAK & HOLD

La durata del comando può essere riassunta in due tipi di gestione principale: con motore in fase di Cranking o sincronizzato (Running).

- Cranking: Prima che la ECU riesca a trovare la fase del motore, la pompa fornisce carburante al rail a intervalli regolari in modo da garantire la massima portata finché la pressione del rail è quella designata come ideale e successivamente diventa la minima disponibile quando la pressione è prossima a questo valore.
- Running: il valore della durata è funzione della pressione nel rail e della velocità del motore.

Vi sono inoltre altri aspetti del controllo aggiuntivi che possono esser utilizzati per ridefinire i parametri di comando:

- Low Fuel Delivery Quantity Strategy: quando il quantitativo di carburante da pompare richiesto è contenuto non si esegue il comando e si aggiorna il quantitativo richiesto al successivo ricalcolo di portata richiesta. I motivi di questa scelta sono due: il primo legato al fatto che con piccole portate non si riesce a garantire un comando regolare, il

secondo è che in questo modo si riduce lo stress del componente in quanto una pompata a bassa portata comporterebbe un incremento delle temperature del componente per l'assenza di un passaggio adeguato di fluido "fresco".

- Anti Noise Strategy: Si modificano i valori di corrente di eccitazione del solenoide al fine di ridurre la velocità di impatto dello spillo e conseguentemente il rumore.
-

IL MODELLO DI CONTROLLO

L'obiettivo dell'algoritmo di controllo consiste nel calcolare l'esatta fasatura della chiusura della valvola di aspirazione. Per ottenere questo target è necessario calcolare la portata necessaria al raggiungimento del valore di pressione desiderato (utilizzando un sistema in closed-loop) e a partire da questo, grazie alla caratteristica inversa della pompa, definire la fasatura.

La fasatura viene calcolata quindi considerando l'angolo necessario all'attuazione (TC) e l'angolo desiderato di pompata. Un'errata definizione della dinamica di chiusura o della modellazione inversa comporta una portata differente da quella richiesta con un errore sempre maggiore all'aumentare del regime del motore (per questo motivo son presenti le strategie di self learning precedentemente menzionate che aggiornano queste caratteristiche).

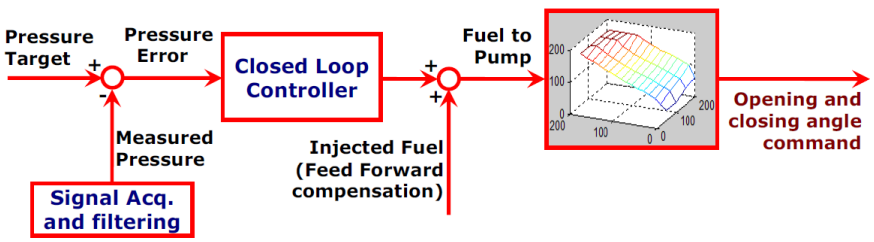


Figura 8 - Possibile architettura di controllo

Il modello d'interesse per questa tesi è la parte dell'architettura di controllo legata al calcolo degli angoli di fasatura dell'attuazione.

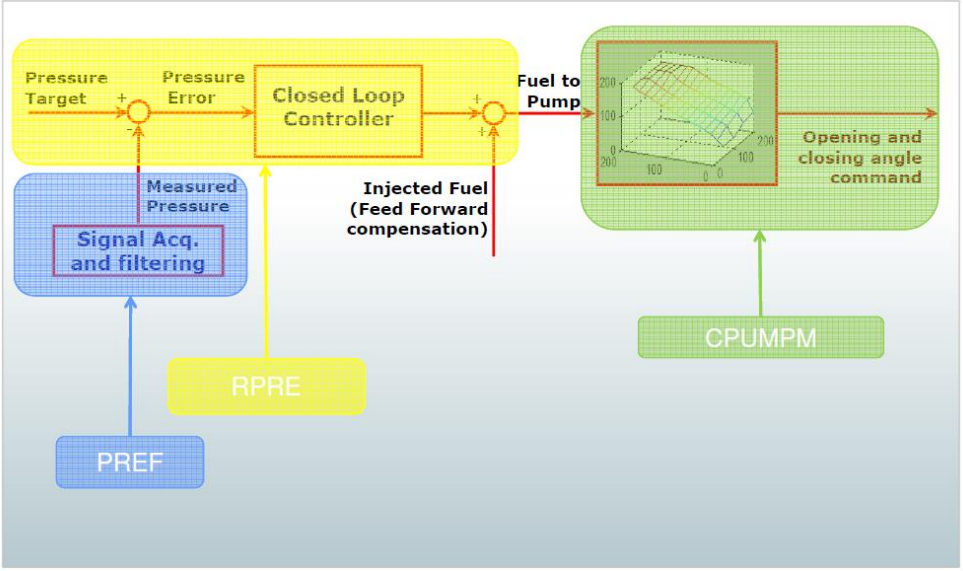


Figura 9 - Schema del modello

Il modello di controllo è composto da più componenti[2]:

- SW-C PREF = Dove è presente la logica di lettura e filtraggio del segnale di pressione nel rail.
- SW-C RPRE = Parte del modello che fornisce la portata ideale per l'attuazione. In questa parte si considerano la pressione letta nel rail, la pressione target determinata dalla strategia di guida e la portata necessaria al prossimo ciclo di iniezione.
- SW-C CPUMPM = In questa parte del modello ci sono tutte le logiche di calcolo per il calcolo dei parametri di attuazione e gli angoli di esecuzione. Fornisce l'angolo di attuazione grazie al modello inverso della pompa al suo interno. Da questo modulo inoltre si effettua l'attuazione del comando verso la centralina tramite le chiamate ai Device Driver specifici.

[2] SW-C = Software Components (specifica AUTOSAR)

IL MODELLO CPUMPM

Il modello CPUMPM è la parte più complessa. Infatti, rispetto alle altre parti che eseguono solo dei calcoli prestabiliti, presenta logiche di calcolo differenti in funzione dello stato del sistema e delle condizioni di input presenti.

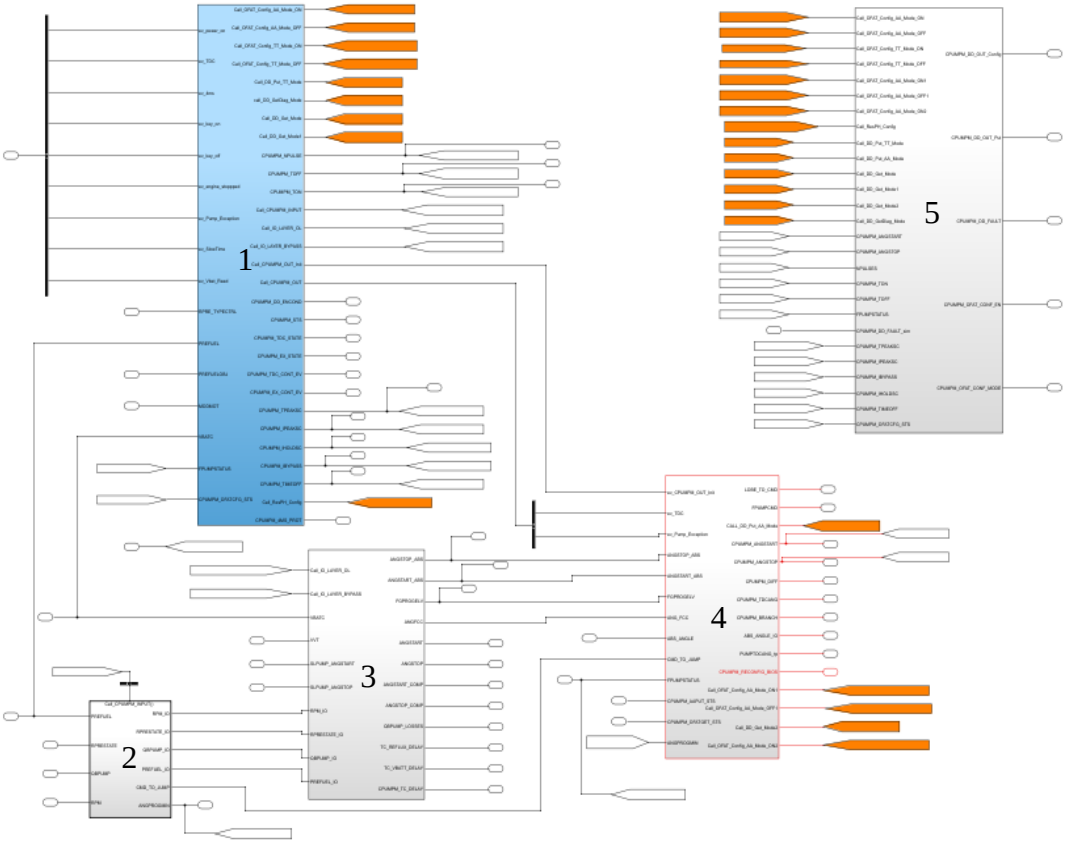


Figura 10 – Blocchi principali logica CPUMPM

È composto da 5 blocchi che (partendo da quello blu in senso antiorario) hanno differenti finalità:

1. ***Schedulatore*** = In funzione dello stato del motore e dell'evento "chiamante", definisce le strategie di calcolo e quali informazioni fornire come output.
2. ***Input Management*** = Raccoglie gli input generati dalle altre parti del modello.
3. ***Calcolo angoli relativi di inizio e fine comando*** = Fornisce in funzione degli input quali sono gli angoli di attuazione rispetto all'angolo della pompa e poi li trasforma in relazione al riferimento dell'angolo motore
4. ***Pianificazione esecuzione*** = Dai valori definiti pianifica il momento di esecuzione ed esegue le chiamate per l'esecuzione.

5. ***Device Driver Layer*** = È il blocco che rappresenta la comunicazione degli output e dei comandi che sono destinati alla centralina.

Il modello è attivo in tutte le fasi del motore, la maggior parte della logica è dedicata a quando il motore è in condizione di sincronia ma anche in fase di cranking ha parti di logica che garantiscono il funzionamento del componente.

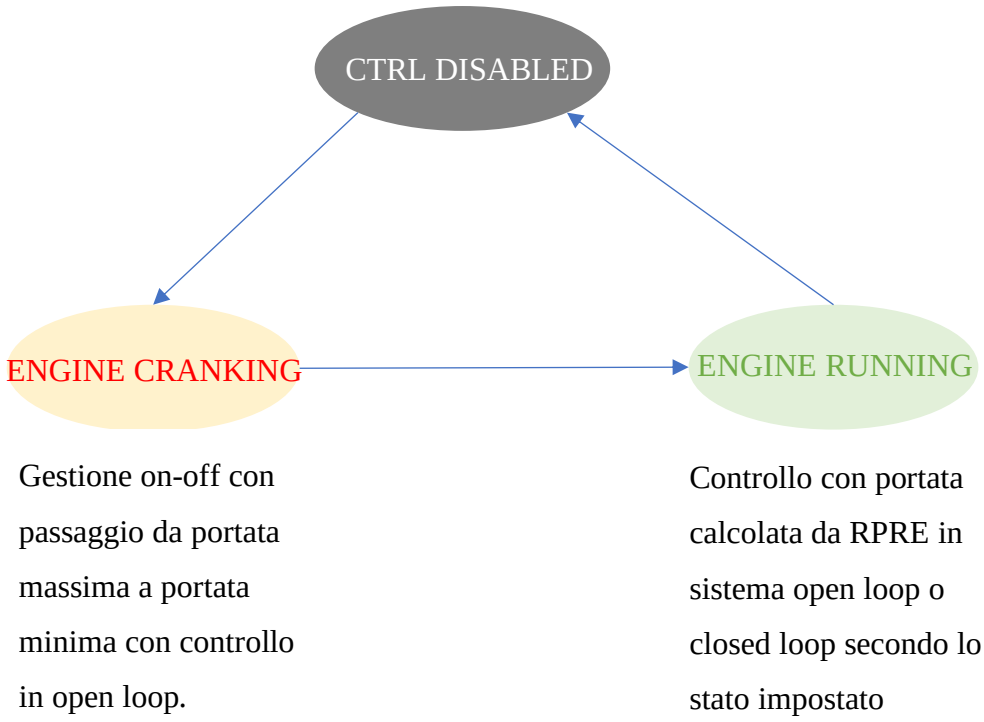


Figura 11 - Diagramma delle transizioni e delle condizioni di utilizzo

Gli obiettivi del modello sono:

- Gestione della precedenza di calcolo.
- Calcolare gli angoli di start e stop del comando di attuazione in funzione della portata richiesta correggendoli rispetto ai ritardi fisiologici derivati dalla tensione della batteria e della forza del flusso in ingresso.
- Ridefinire gli angoli ottenuti in valori di angolo motore.
- Individuare eventuali comandi non attuati per effetti elettrici (ad esempio, batteria troppo scarica).
- Comunicazione dei parametri di attuazione tramite tre tipi di comando ai Device Driver:
    - **Config**: configurazione dei parametri
    - **Put**: chiamata di nuova attuazione di comando
    - **Get**: controllo di feedback sull'attuazione del comando

METODOLOGIA USATA PER LA MODIFICA

Come detto in precedenza, l'obiettivo di questa tesi è la modifica della struttura del modello di controllo sviluppato in Simulink/Steteflow finalizzata ad ottenere un nuovo modello, isofunzionale rispetto all'originale, con struttura a runnable (l'imposizione degli stessi output in funzione di un certo tipo di input è importante per poter eseguire delle verifiche relative alla capacità di esecuzione delle logiche). Essendo un percorso complesso, si è deciso di procedere per step di rimodellazione. La prima parte è consistita nella ridefinizione delle logiche stateflow legate allo schedulatore, successivamente ci si è focalizzati sulla ridefinizione dei subsystem utili ai calcoli.

Ad ogni modifica sostanziale del modello si è effettuata una validazione della modifica implementata, applicando di volta in volta metodi differenti.

Solo una volta ottenuto un modello funzionante, si è passati alla fase di codifica in linguaggio ANSI C, eseguita tramite il motore di generazione assistita del codice dSpace TargetLink, ridefinendo eventualmente le variabili e/o le loro caratteristiche, caratterizzando adeguatamente i blocchi del modello ed effettuando simulazioni comparative.

Quando anche questo ultimo passaggio ha fornito un feedback positivo, si è passati alla valutazione quantitativa dell'efficienza delle modifiche effettuate.

Data la complessità del modello in esame, si è deciso di testare preliminarmente metodo su un modello semplificato nelle logiche (UTET) in modo da poter concentrare le attività di analisi dei problemi più su aspetti concettuali riguardanti le modifiche che su errori di modellazione generati dalla ridefinizione di calcoli complessi.

IL METODO DI VALIDAZIONE

Il percorso di reingegnerizzazione del modello ha seguito uno schema rigido di validazione a tappe in modo da garantire un sistema di controllo solido ed efficiente.

La prima fase durante la modellazione ha principalmente due tipi di verifica:

- ISOFUNZIONALITÀ: Il nuovo modello deve poter eseguire le stesse azioni con gli stessi output del modello precedente, pertanto, una modifica si può considerare efficace

solo quando gli output del modello sono uguali per tutti gli scenari considerati (confronto MIL-MIL[3]).

- ANALISI COVERAGE: Si esprime in percentuale e rappresenta quanti percorsi logici possibili sono stati eseguiti almeno una volta durante le simulazioni dei vari scenari.

Tendenzialmente queste verifiche si eseguono ogni qual volta le modifiche al modello sono sostanziali, ad esempio la modifica della logica di uno stateflow, la ridefinizione della logica di un subsystem o la modifica dell'esecuzione dei calcoli.

Una volta che questi controlli danno esito positivo, si passa alla seconda fase legata alla definizione e scrittura del codice. In questa fase, la realizzazione del codice è effettuata come già detto attraverso un tool, che attraverso la caratterizzazione (targettatura) del modello, produce un codice C auto-generato con aritmetica di calcolo a punto fisso (fixed point).

- CONFRONTO MIL-SIL[4]: consiste nel verificare se i risultati ottenuti dalla simulazione SIL sono compatibili con le simulazioni MIL, essendo presenti nel SIL dei troncamenti, dovuti alla presenza di un calcolo aritmetico differente, si possono avere degli scostamenti con i risultati delle simulazioni MIL. Pertanto è necessario stabilire una tolleranza tra i risultati ottenuti.
- CONFRONTO SIL-SIL: ultimo passaggio di verifica, si controlla che i risultati ottenuti dalle simulazioni SIL del modello originale e del nuovo modello siano uguali.

Nelle prossime pagine si espone nel dettaglio ogni tipo di verifica sopra citata.

ISOFUNZIONALITÀ

Ad ogni simulazione si registrano tramite le funzionalità di *signal logging* di Simulink alcune variabili in output che possono essere di due tipi:

- *OUTPUT* se corrispondono ai risultati di calcolo del modello.
- *TESTPOINT* variabili che non rappresentano un risultato finale della catena di calcolo del modello ma fungono da verifica se una parte di logica viene eseguita correttamente o meno.

[3] MIL = Model In the Loop
[4] SIL = Software In the Loop

Questi output selezionati non corrispondono a tutte le variabili implicate nei calcoli presenti nel modello ma quando queste variabili sono uguali si può dedurre con certezza che tutte le altre variabili presenti siano congruenti. Il confronto può esser fatto in due modi: grafico e analitico.

Per motivi funzionali all'analisi, si è deciso di utilizzare uno script in linguaggio MATLAB che confrontasse le variabili in output tra i due modelli e qualora non fossero congrue, stampasse a video due grafici: il primo di confronto del risultato tra i due modelli e il secondo dell'errore assoluto tra simulazione originale e nuova. Inoltre, lo script permette di scegliere se fare il confronto rispetto a un solo scenario di simulazione o a tutti gli scenari possibili. Nel secondo caso si è deciso di mettere a video solo il grafico del confronto tra le due grandezze e non quello dell'errore per non sovraccaricare la schermata inutilmente.

La verifica di un solo scenario è utile quando si fa una modifica al modello e ci si vuole focalizzare sulla funzionalità della modifica, la verifica di tutti gli scenari serve invece per poter avere la visione ampia e riassuntiva, quindi il dettaglio dell'errore non è importante.

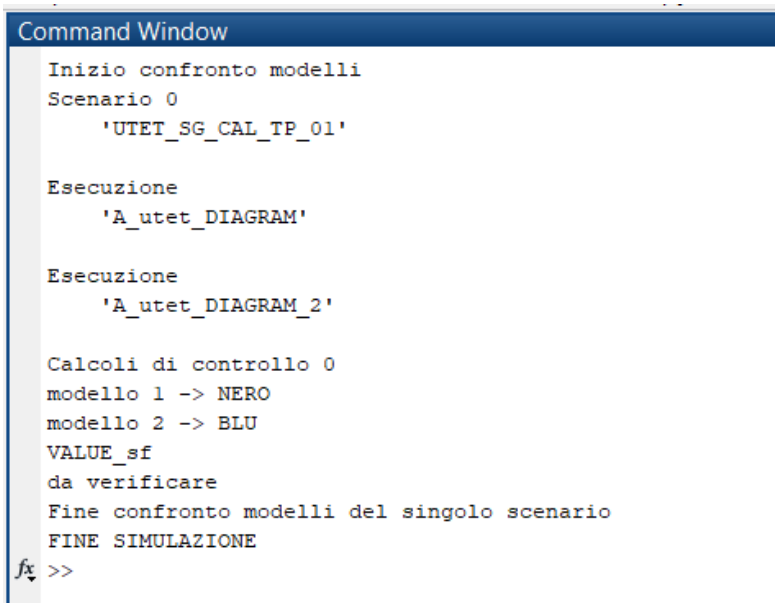


Figura 12 - Schermata di comunicazione

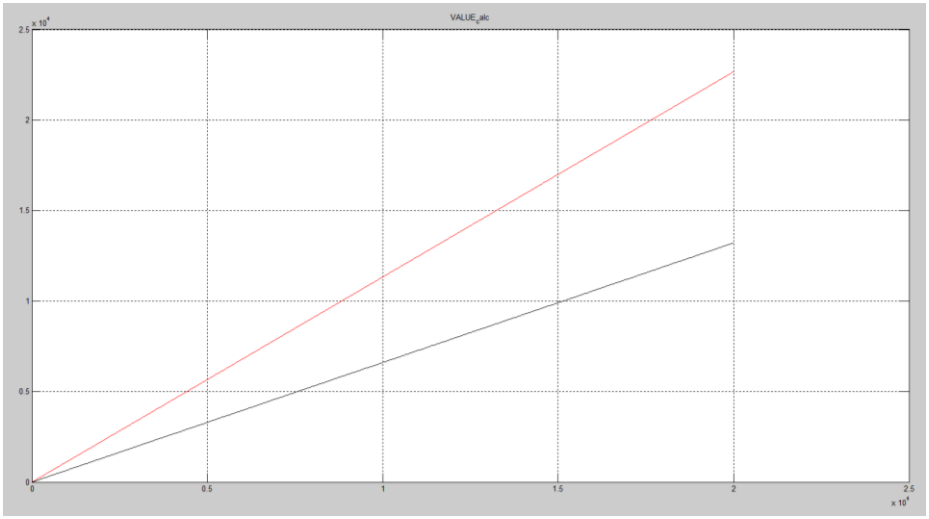


Figura 13 - Esempio di grafico di confronto tra due output non congruenti

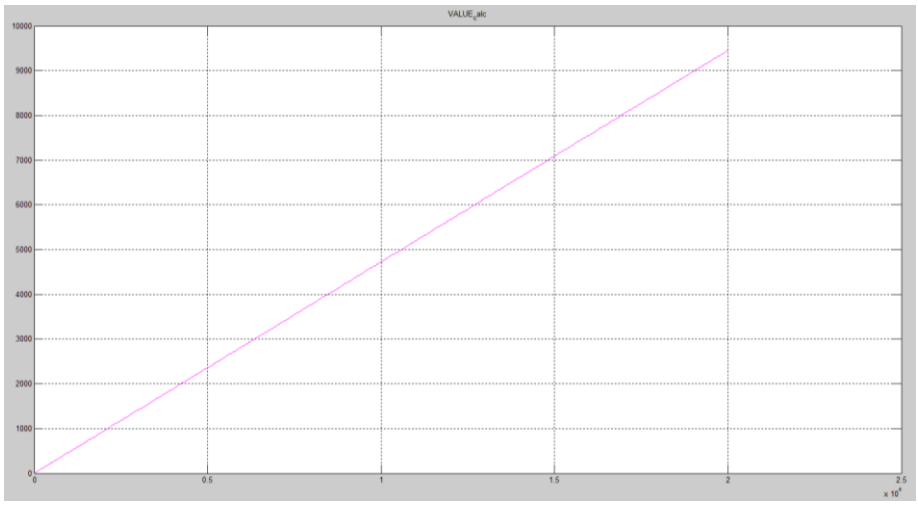


Figura 14- Grafico errore tra i due segnali

IL COVERAGE

È l'analisi che si può fare tramite l'apposito tool interno a Simulink la quale verifica che tutte le logiche presenti nel modello siano state eseguite, infatti tramite gli input e le variabili imposte di ogni scenario di simulazione (chiamato TEST PATTERN) si verifica se tutti i blocchi (nel caso di un modello) o le linee di codice (nel caso di un sorgente C) sono state eseguite e se tutte le espressioni sono state verificate in tutte le loro possibili combinazioni. I tipi di verifica sono tre:

- DECISION: serve a verificare la funzionalità dei blocchi con logiche interne di tipo "if", "while", "for" e i blocchi switch, ovvero tutti i punti decisionali tra logiche alternative, assicurando che tutti i possibili esiti decisionali siano stati effettuati almeno una volta.

- CONDITION: serve a verificare che ogni condizione logica sia stata testata in tutte le sue possibili combinazioni di valori.
- MC\DC: verifica, nel caso di espressioni decisionali composte, che si sia valutata ogni singola condizione che la compone in modo tale che la sua sola variazione faccia ottenere un risultato differente all'intera espressione.

```
if  (A||B) && C > 0
     out= 1;
else
     out =0;
end
```

La logica considerata fornisce un differente output in base alla soluzione ottenuta dall'equazione logica (funzione delle tre variabili A, B, C). Vi sono 8 possibili combinazioni che possono assumere le tre variabili, ma bastano 4 combinazioni per testare tutti e tre gli aspetti del Coverage per verificare completamente la logica.

Di seguito, l'esempio dei tre test pattern con l'evoluzione dei tre parametri ad ogni esecuzione.

TEST PATTERN	TP1	TP2	TP3	TP4
A	0	0	0	1
B	0	1	1	0
C	1	1	0	1
out	0	1	0	1
DECISION	50%	100%	100%	100%
CONDITION	50%	66%	83%	100%
MC\DC	0%	33%	66%	100%

Tra il TP1 e il TP2 si è verificato completamente il DECISION in quanto vengono ottenuti i risultati di entrambi i percorsi possibili dalla logica "IF", con il terzo TP si incrementa al percentuale di analisi del CONDITION e si ha la prima percentuale di verifica dell'MC\DC. Con i successivi TP, si ottiene la copertura totale di analisi.

È fondamentale, pertanto, avere dei test pattern adeguati a ridurre al minimo le simulazioni necessarie al completamento dell'analisi di Coverage. Una strategia è quella di ipotizzare ogni test pattern come uno "scenario" possibile dove si verificano tutte le funzionalità di quella condizione.

Quando il Coverage è completo alla fine della simulazione grazie alla funzionalità di Simulink Coverage si ottiene il blocco colorato di verde e si può creare un file contenete il rapporto riassuntivo delle percentuali coperte come si può vedere di seguito.

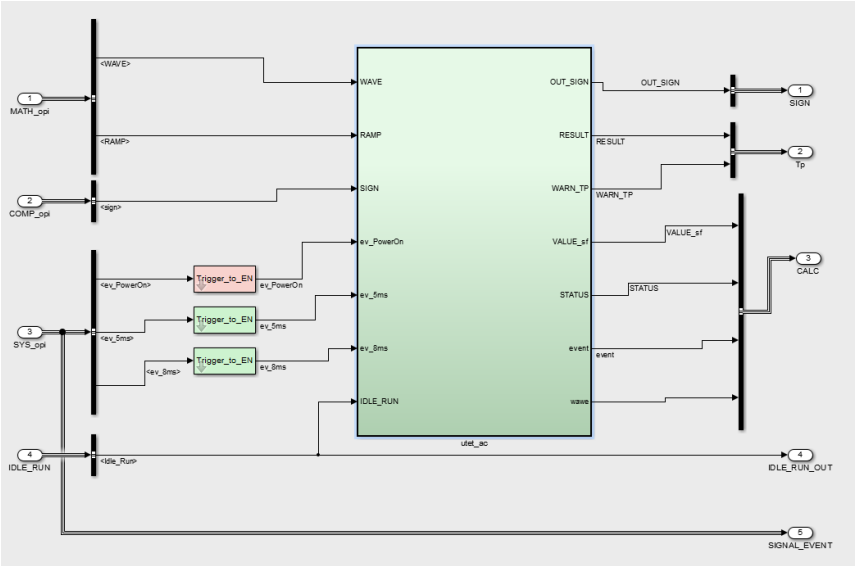


Figura 15 - Blocco con 100% Coverage

		D1		C1		MCDC		D1		C1		MCDC		D1		C1		MCDC	
46	utet_ac	51	90%		58%		0%		1%		0%		0%		100%		100%		100%
47	En_to_FC	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
48	Function Call Generator	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
49	Buffering	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
50	Call_Runnable	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
51	Posting	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
52	En_to_FC1	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
53	Function Call Generator	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
54	Buffering	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
55	Call_Runnable	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
56	Posting	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
57	En_to_FC2	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
58	Function Call Generator	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
59	Buffering	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
60	Call_Runnable	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
61	Posting	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
62	Stateflow DIAGRAMMI	35	87%		58%		0%		2%		0%		0%		100%		100%		100%
63	DIAGRAMMA_run8_idle1	17	81%		75%		0%		3%		0%		0%		100%		100%		100%
64	SF: DIAGRAMMA_run8_idle1	16	80%		75%		0%		3%		0%		0%		100%		100%		100%
65	Inizializzazione	2	100%		NA		NA		0%		NA		NA		100%		NA		NA
66	Filter_Init	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
67	Math_init	1	100%		NA		NA		0%		NA		NA		100%		NA		NA
68	calcoli originali	16	93%		50%		0%		0%		0%		0%		100%		100%		100%
69	Filter	7	85%		50%		0%		0%		0%		0%		100%		100%		100%
70	evento 5ms	2	75%		50%		0%		0%		0%		0%		100%		100%		100%
71	evento 8ms	2	75%		50%		0%		0%		0%		0%		100%		100%		100%
72	Math_calc	5	100%		NA		NA		0%		NA		NA		100%		NA		NA
73	Strategy_calc	4	100%		NA		NA		0%		NA		NA		100%		NA		NA

Figura 16 - Diagramma riassuntivo Coverage

L'analisi del Coverage va eseguita con l'obiettivo di ottenere la copertura totale unicamente per la parte del modello interessata dalla traduzione in codice C in quanto è l'unica parte che

successivamente verrà rielaborata per esser inserita in centralina. È quindi possibile avere parti di modello non completamente verificate al di fuori di questa parte del modello.

L'utilizzo del Coverage oltre a fornire un feedback sulla copertura del modello è importante al fine di dare maggiore consistenza alle verifiche comparative MIL-MIL di isofunzionalità, infatti, solo grazie a una copertura totale del codice si può garantire la totale isofunzionalità dei modelli comparati.

TARGHETTATURA

Il processo di ***targhettatura*** consiste in una serie di operazioni che consentono al tool **TargetLink** (TL) di "tradurre" il modello **Simulink** (SL) in sintassi di codice C. In questo modo si possono eseguire delle simulazioni SIL in ambiente Matlab.

Questa trasformazione avviene con la modifica dei blocchi nel modello con blocchi compatibili con TargetLink. Non è necessario sostituire i blocchi presenti con quelli della libreria di TL, infatti è presente il comando di trasformazione dal menu a tendina del blocco SL. Dopo la trasformazione, il blocco non perde le sue funzionalità per quanto riguarda la simulazione ma guadagna altre opzioni, tra cui la possibilità di inserire informazioni che forniscono indicazioni per la generazione del codice.

Le impostazioni più importanti sono quelle che configurano le informazioni SW riguardanti la variabile C che nel codice generato rappresenterà il valore in uscita dal blocco che può essere effettuata tramite l'esplicita definizione di un oggetto Variabile di TL o tramite le sue sole caratteristiche SW (es. Tipo, Range, LSB, offset).

Una buona targhettatura fornisce un codice più snello (minori righe di codice) che corrispondono a un tempo di esecuzione più contenuto. La definizione accurata delle caratteristiche delle variabili invece, fornisce una minore occupazione di memoria RAM\ROM e quindi anche in questo caso performance migliori in caso di sistemi real-time con limitate risorse[5].

[5] Il codice viene scaricato nelle centraline in uno spazio che si chiama Memoria FLASH. Mentre le variabili a seconda delle loro caratteristiche in una memoria Static ROM, o in una memoria dinamica RAM. Minore è il quantitativo di memoria occupata maggiore è l'efficienza di esecuzione.

Le caratteristiche delle variabili vengono definite dall'architetto software e dal modellista. Le descrizioni delle variabili possono essere riassunte in questi aspetti principali:

- TIPO: definisce il tipo di variabile (es. booleana, fixed point, float -non utilizzata perché occupa troppa memoria-) e la sua dimensione in bit.
- RANGE: descrive qual è l'intervallo di grandezza allinterno del quale ci sono tutti i valori che la grandezza può assumere.
- LSB[6]: Corrisponde al concetto di risoluzione in quanto definisce la quantità minima rappresentabile per la grandezza descritta.

Volendo fare un esempio, si consideri una variabile *A* con un Range nell'intervallo [-50 150] dove la precisione richiesta è di 0,05.

$$A\ bit_{min} = \log_2\big(150 - (-50)\big) + \log_2\left(\frac{1}{0.05}\right) = 8 + 5 = 13\ bit$$

I risultati vanno arrotondati per eccesso all'intero superiore, avendo ottenuto un numero minimo di 13bit ed essendo i tipi di base disponibili solo di 8,16 o 32 bit, la dimensione della variabile *A* sarà quindi di 16bit. Avendo un range con anche numeri negativi il Tipo della variabile in targhettatura sarà INT16, dove un bit viene dedicato al segno della variabile (nel caso di variabili intere il tipo è identificato come UINT).

Nel caso di una ri-modellazione la definizione di una variabile è un'attività inusuale in quanto teoricamente son già state determinate tutte le grandezze e le variabili del modello.

Tutte le informazioni riguardanti le variabili sono raccolte nel TLDD (Target Link Data Dictionary) dove per ragioni di organizzazione dei dati vengono divise secondo il tipo di funzione che hanno nel modello (input, output, intermediate, constant e lookup)[7].

Per la generazione del codice da utilizzare in centralina è preferibile avere una definizione delle variabili in virgola fissa (fixed point)[8].

[6] LSB = Least Significant Bit
[7] Convenzione interna a Marelli per migliorare la gestione
[8] Alcuni processori possono non supportare calcoli in virgola mobile. Nella maggior parte dei casi è possibile ma si preferisce mantenere un approccio a virgola fissa in quanto i calcoli risultano essere più rapidi.

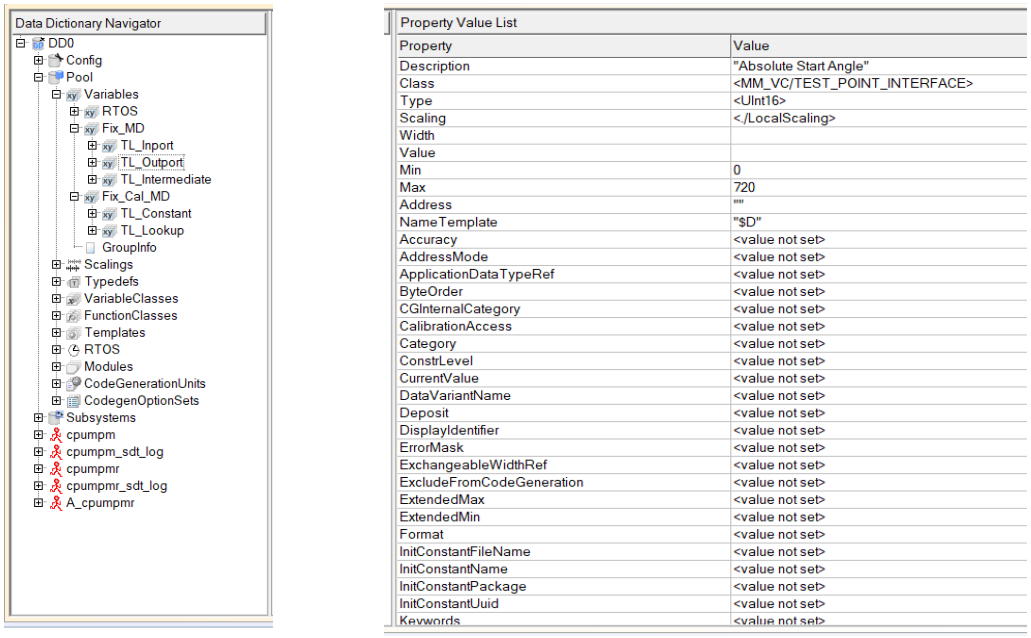


Figura 17 - Dettaglio della descrizione della variabile

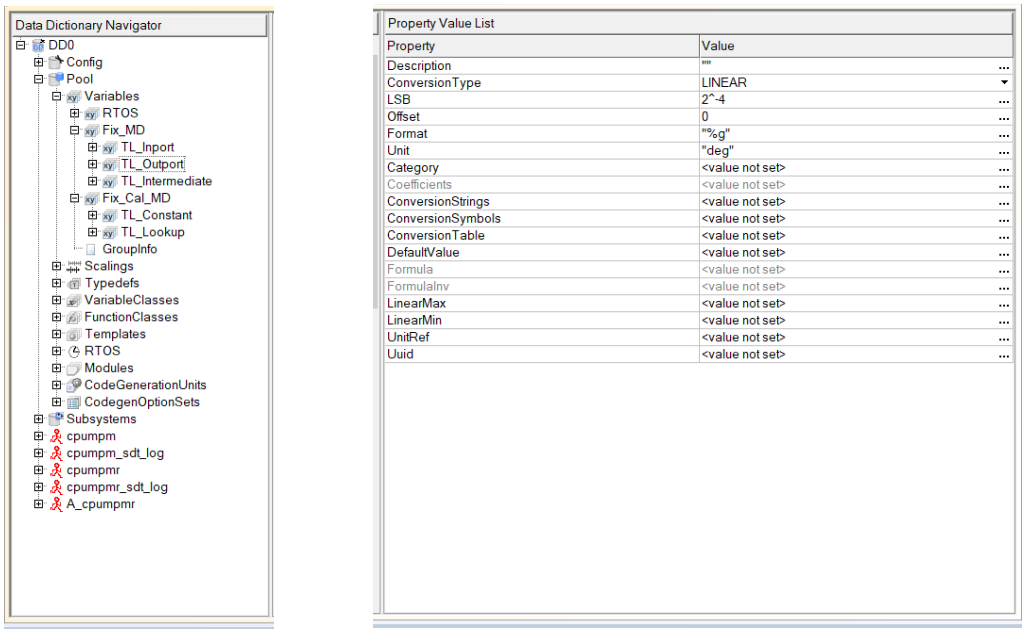


Figura 18 - Dettaglio della scalatura della variabile

VERIFICA MIL-SIL

Per la verifica MIL-SIL si può fare in due modi: o confrontando i risultati tramite i log di TargetLink o utilizzando uno script di Matlab sviluppato internamente a Marelli chiamato “auto_sil”.

L’analisi dei risultati tramite il tool Target Link è utile per indagini macroscopiche (ad esempio confrontare l’uscita di un contatore) ma per altre grandezze non permette di considerare intervalli di tolleranza per gli effetti di troncamento presenti nei calcoli a causa della differenza

tra calcoli in FLOAT64 (tipo utilizzato per i calcoli MIL, virgola mobile a 64bit) e i tipi scelti per esser utilizzati nella simulazione SIL (argomento trattato in introduzione di Target Link). In questo caso lo script "auto_sil" fornisce il supporto adeguato in quanto per tutte le variabili in output è stata definita una tolleranza entro la quale lo scostamento tra MIL e SIL è accettabile (solitamente corrisponde al valore corrispondente della precisione legata a LSB).

Lanciando lo script si ha un'interfaccia che chiede quale modello e su quale scenario si vuole fare il confronto. Dopo i calcoli si ottiene il confronto tra i risultati riassunto nella seguente schermata.

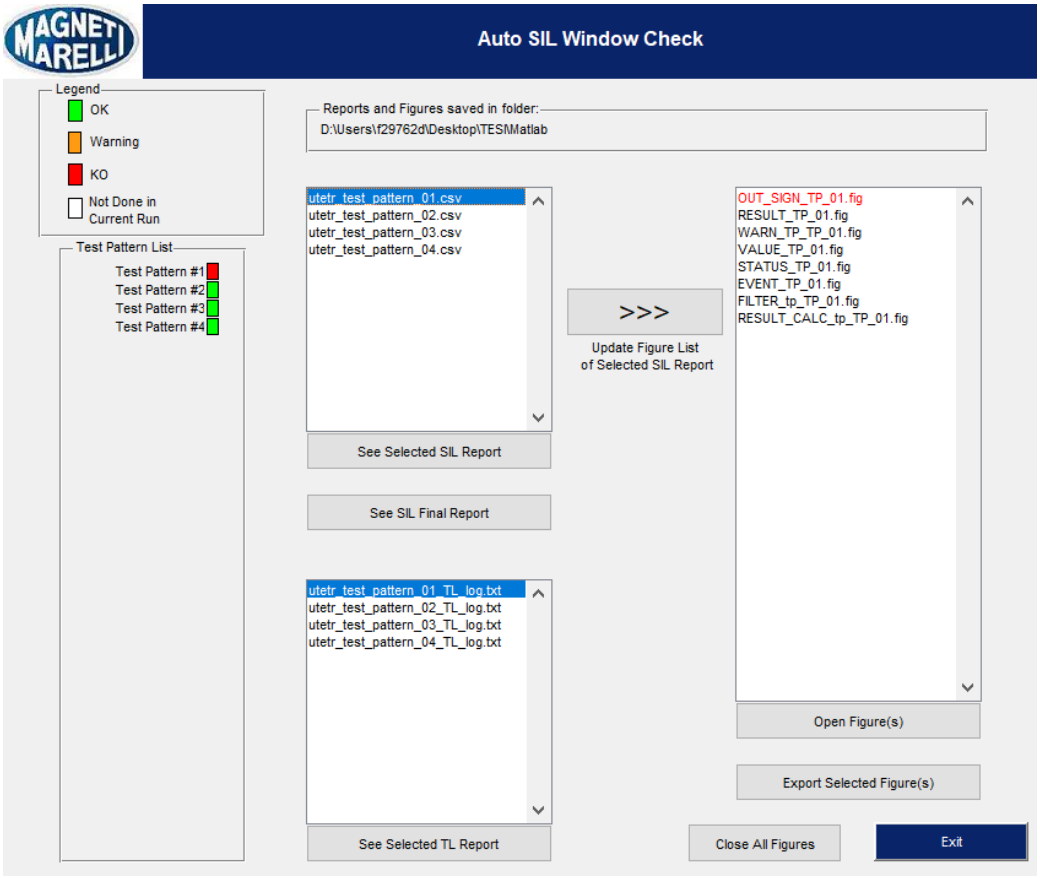


Figura 19 - Schermata riassuntiva risultati auto_sil

A destra si ha il "semaforo" sullo stato dei risultati:

verde = ok, arancio = valore vicino alla soglia, rosso = incongruenza tra risultati, **bianco** = non eseguito nell'ultima verifica.

Si può selezionare il test pattern desiderato e vedere quali variabili non sono congruenti (in questo caso OUT_SIGN_TP per lo scenario 1). Si può aprire la finestra di confronto dove è presente un grafico di sovrapposizione delle variabili e un grafico relativo all'errore presente.

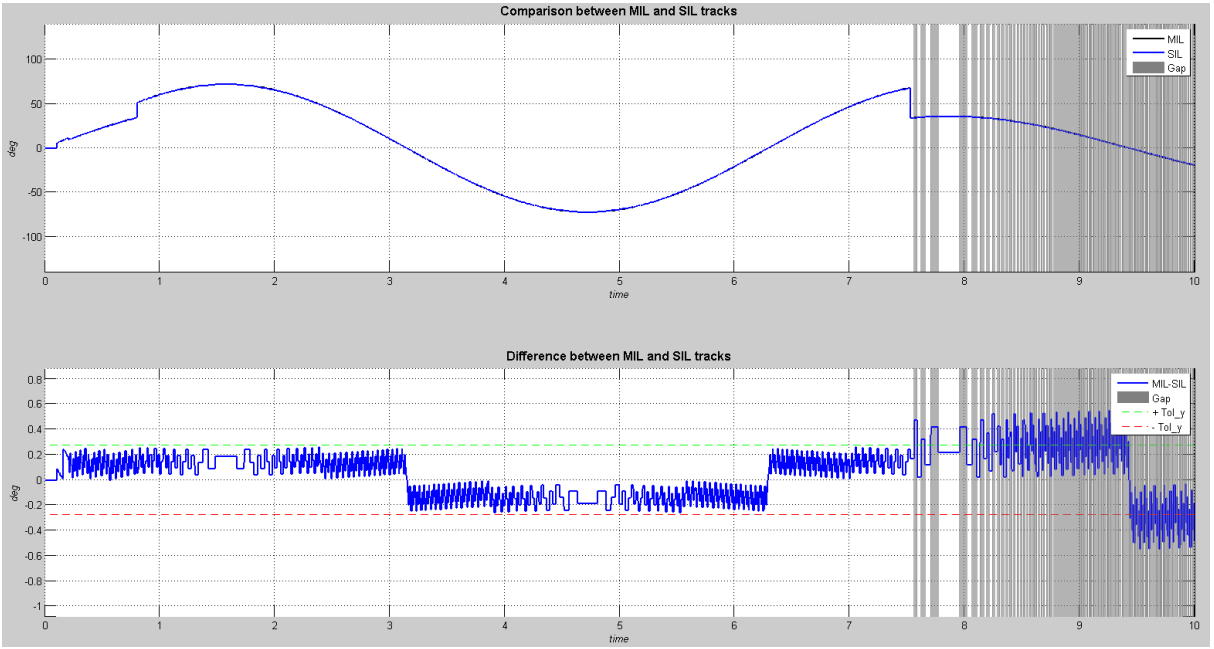


Figura 20 - Esempio di confronto segnali

Ad ogni punto della simulazione dove il risultato è oltre alla soglia massima di errore ho il grafico evidenziato in grigio. Anche se non necessario per i modelli originali, è sempre importante fare questa simulazione in quanto potrebbero esserci delle incongruenze presenti nel modello originale che durante la sua modellazione sono stati considerati ugualmente accettabili. In questo caso è impossibile poter ottenere risultati ottimali solo tramite una rimodellazione delle logiche.

Qualora vi siano delle grandezze incongruenti, non si agisce sul codice generato ma solamente sulla targhettatura dei blocchi del modello e nella ridefinizione eventualmente dei Tipi e scalature delle variabili.

CONFRONTO SIL-SIL

Per eseguire il confronto tra i risultati SIL di due modelli per uno stesso Test Pattern si è deciso di usare uno script apposito simile a quello di confronto MIL-MIL. In questo caso, avendo le differenze tra MIL e SIL contenute all'interno della tolleranza imposta dalla verifica auto_sil, tendenzialmente la differenza tra i due modelli sono accettabili.

PARAMETRI ACCETTABILITÀ

L'obiettivo principale è quello di ottenere un modello con lo stesso comportamento del modello esistente, pertanto, non sono ammessi nessun tipo di variazione tra i risultati di confronto tra modello originale e nuovo.

METRICHE DI COMPARAZIONE

Una volta terminato il processo di rimodellazione e ottenuto risultati di simulazione e qualitativi congruenti con quanto desiderato si è passati alla definizione di alcune metriche di valutazione sugli effetti che questa nuova modellazione fornisce. Sono state individuati i seguenti metri di paragone:

- Lunghezza codice in linguaggio C
- Tempo di simulazione MIL
- Tempo di simulazione SIL
- Tempo di generazione e compilazione codice
- Tempo di verifica da parte del tool "auto_sil" per tutti i Test Pattern

LA STRUTTURA DEGLI SCRIPT

Durante lo sviluppo della tesi è stato necessario sviluppare degli script che rendessero possibile l'analisi dei risultati ottenuti e le simulazioni in maniera accurata e automatica. Gli script che sono stati scritti sono principalmente 2. La scrittura degli script è stata fatta in maniera che con la modifica di poche righe di codice presenti all'inizio di essi si possano eseguire per entrambi i modelli.

SCRIPT DI CONFRONTO

Lo script di confronto è stato usato per le fasi di verifica di isofunzionalità MIL-MIL e per il Coverage, inoltre è stata integrata anche la possibilità di eseguire una semplice simulazione. Per rendere più facile da utilizzare, all'inizio dell'esecuzione sono state inserite alcune domande che consentono la configurazione del tipo di test che si desidera fare. La struttura è descritta di seguito.

Struttura dello script

- Scelta del tipo di test {Simulazione, Confronto modelli, Coverage}

  Simulazione
  - Scelta del modello
  - Scelta scenario di simulazione {Uno, Tutti}

    UNO / TUTTI
    - Esecuzione della simulazione e salvataggio su WS dei risultati come struttura
    - Termine della simulazione o esecuzione per nuovo scenario

  Confronto modelli
  - Scelta dei modelli
  - Scelta scenario di simulazione {Uno, Tutti}

    UNO / TUTTI
    - Disattivazione animazioni ed eliminazione breakpoint
    - Esecuzione della simulazione del modello 1 e salvataggio risultati
    - Esecuzione della simulazione del modello 2 e salvataggio risultati
    - Confronto risultati per ogni variabile tramite differenza tra due vettori
    - Salvataggio WS di tutti i risultati come struttura
    - Stampa a video del grafico dei risultati a confronto e dell'errore se la differenza non è nulla per tutti gli elementi del vettore
    - Termine del confronto o esecuzione test per nuovo scenario

  Coverage
  - Scelta del modello
  - Disabilitazione di animazioni ed eliminazione dei breakpoint
  - Esecuzione della simulazione per tutti gli scenari
  - Salvataggio in WS dei risultati del Coverage
  - Apertura HTML con resoconto simulazione

SCRIPT RILEVAMENTO TEMPISTICHE

Nello script si esegue un ciclo per ogni Test Pattern dove si eseguono prima delle simulazioni MIL in loop per tutti i Test Pattern, successivamente si esegue in loop la generazione e la compilazione del codice, dopo l'ultima compilazione si esegue il loop di esecuzione delle simulazioni SIL, a questo punto si cambia Test Pattern e si rieseguono i loop di generazione, compilazione e simulazione SIL. Una volta terminati tutti i loop si esegue l'analisi dei tempi.

A causa di processi concorrenti nel pc, i tempi di esecuzione presentano a volte delle tempistiche anormali. Si è deciso pertanto di considerare tra tutte le tempistiche rilevate solo quelle con uno scarto adeguatamente contenuto tra loro in modo da avere il calcolo della media più preciso possibile.

Nel dettaglio, per ogni gruppo di tempistiche di una simulazione per Test Pattern si esegue la differenza tra una grandezza e tutte le altre dello stesso scenario. Successivamente si valutano le differenze relative, se questa differenza è inferiore alla tolleranza massima ammessa per un valore superiore al 60% del numero totale di simulazioni allora la tempistica viene ammessa nel calcolo della media.

Imponendo una tolleranza del 5% rispetto al minor tempo rilevato, il numero di simulazioni necessarie per poter definire la media come descritto è stato attestato ad almeno 15.

Struttura dello script

La struttura dello script è stata ispirata dallo script di comparazione dati.

- Lancio dello script con pulizia Workspace e chiusura del modello
    - Simulazioni MIL
        - Caricamento dei parametri di simulazione del primo Test Pattern
        - Loop di simulazioni MIL per n-volte
        - Caricamento nuovo Test Pattern o passaggio successivo
    - Simulazioni SIL
        - Caricamento dei parametri di simulazione del primo Test Pattern
        - Loop di generazione codice per n-volte
        - Loop di simulazioni SIL per n-volte
        - Pulitura Workspace e passaggio a simulazione per altro Test Pattern o fine esecuzione simulazioni
    - Rielaborazione dei tempi rilevati tramite la media dei tempi considerati "consistenti"

UTET

UTET – Il modello palestra originale

Il concetto di modello palestra è quello di avere un modello con gli stessi accorgimenti nella modellazione presenti in CPUMPM ma con una logica semplificata in modo da potersi focalizzare su errori concettuali di modellazione anziché su problemi legati alla forma.

La struttura è uguale al modello CPUMPM dove ho 3 subsystem principali:

- Generazione input (SignalGenerator)
- Calcolo (opi)
- Rielaborazione risultati (SimulationResult).

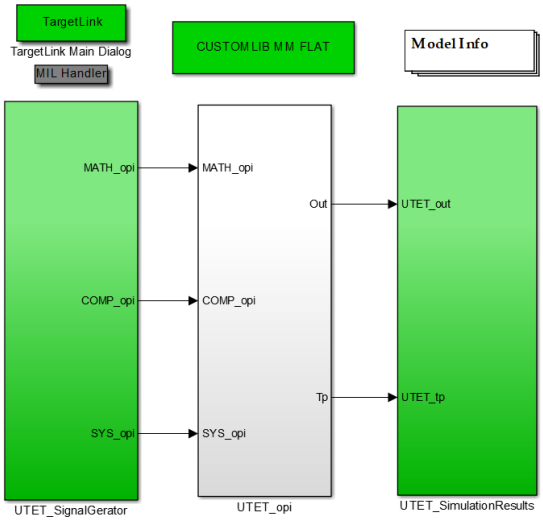


Figura 21 - Livello superiore modello UTET

Analizzando il subsystem UTET_SignalGenerator si notano tre tipi di input:

- Eventi: ev_Poweron ed ev_MediumTime, il primo corrisponde in un segnale gradino mentre il secondo in un'onda quadra con ampiezza 5ms e durata della pulsazione per 2ms.
- Segnali: RAMP e WAVE, il primo è una rampa che ad ogni simstep incrementa di 1 partendo da 0 e il secondo un segnale sinusoidale con range [-30; 30] e periodo 1 rad/sec
- Acquisizioni: che corrispondono a dei vettori presenti nel workspace.

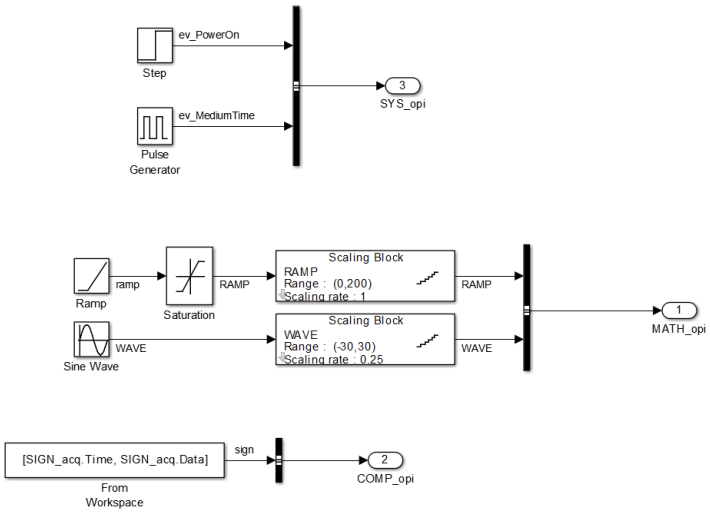


Figura 22 - Livello "UTET_Signalgen"

Il subsystem UTET_SimulatioResult è semplicemente un blocco dove sono disponibili i grafici degli output del sistema. Gli output sono divisi in due tipi “output” e “testpoint”, il primo tipo è il risultato che si desidera ottenere dal modello mentre i testpoint corrispondono a delle grandezze di controllo che si usano per monitorare il corretto funzionamento del modello.

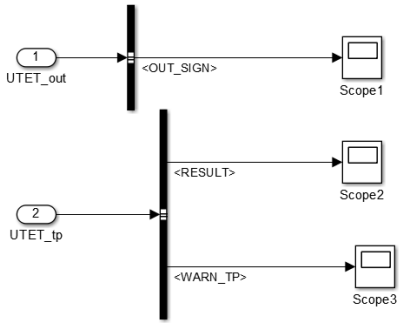


Figura 23 - Livello "UTET_SimulationResult"

Il subsystem centrale UTET_opi è il cuore del modello e ha dei livelli interni con particolari utilità.

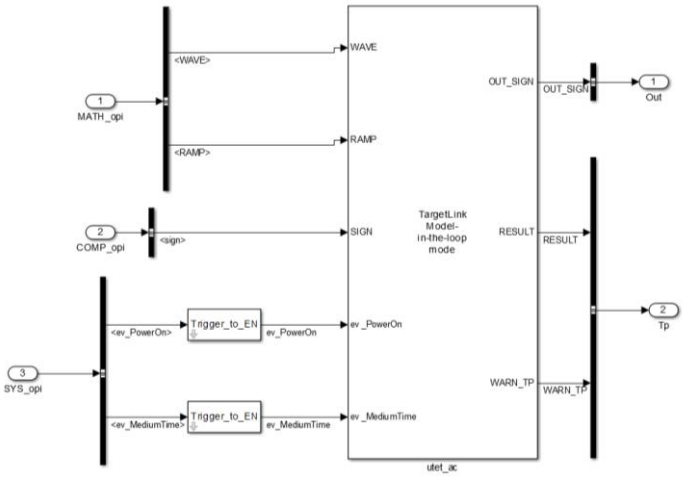


Figura 24 - Livello "UTET_opi"

Al primo livello è presente il subsystem “utet_ac” dove tutto quello che verrà modellato al suo interno verrà successivamente trasformato in codice C, a questo livello pertanto sono presenti solamente dei blocchi di trasformazione del segnale degli eventi i quali trasformano il segnale da onda quadra a un segnale booleano. Tutte le evoluzioni del modello una volta definiti gli eventi saranno circoscritte all’interno di solo questo subsystem.

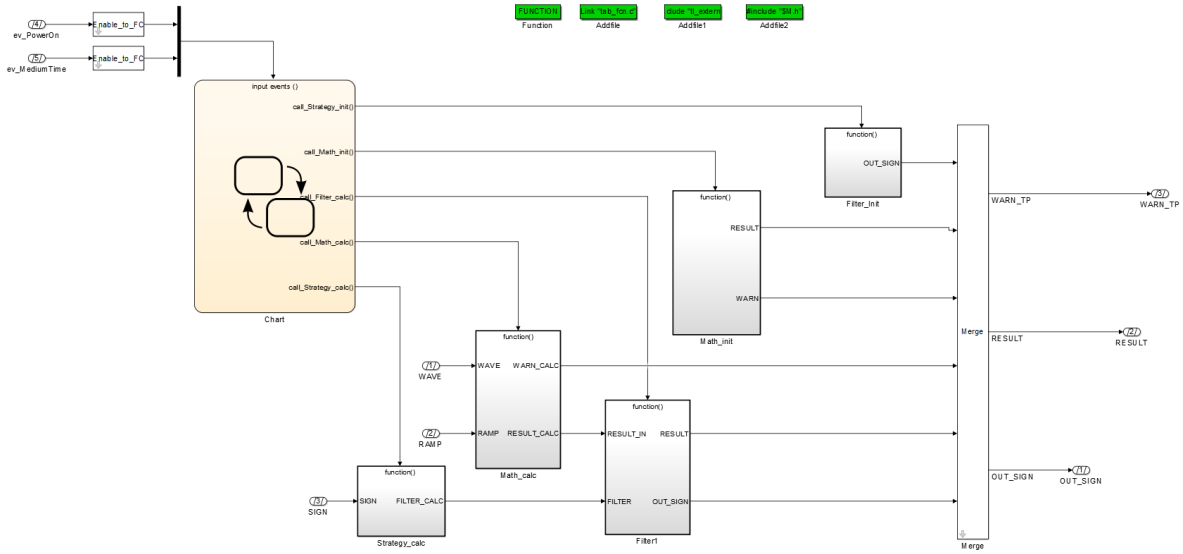


Figura 25 - UTET_ac

All’interno del subsystem utet_ac oltre ai blocchi che implementano la logica del modello sono presenti 4 blocchi (in verde) che servono a inserire dipendenze da file e informazioni aggiuntive per la generazione del codice.

La struttura del modello è composta da uno schedulatore (chart in stateflow) e 6 subsystem: 3 di calcolo (strategy_calc; Math_calc; Filter), 2 di inizializzazione (Filter_init; Math_init) e uno utilizzato per fare il merge di segnali calcolati in più di un Function Call subsystem (Merge). Da questo livello, e per tutti i livelli inferiori, tutti i blocchi presenti devono esser descritti con le opzioni di Target Link per poter ottenere il codice.

Di seguito il dettaglio dei blocchi

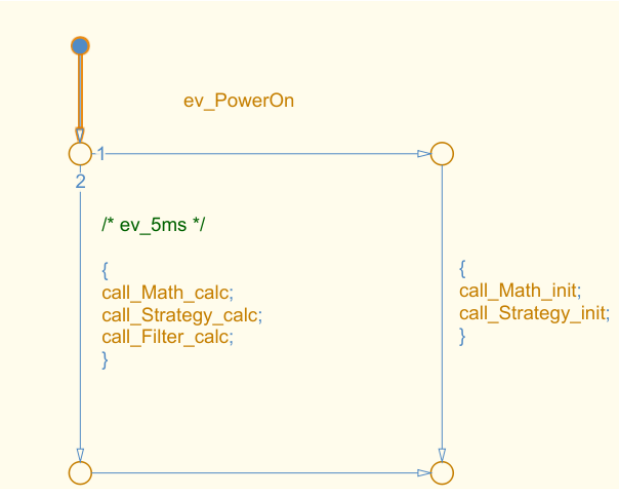
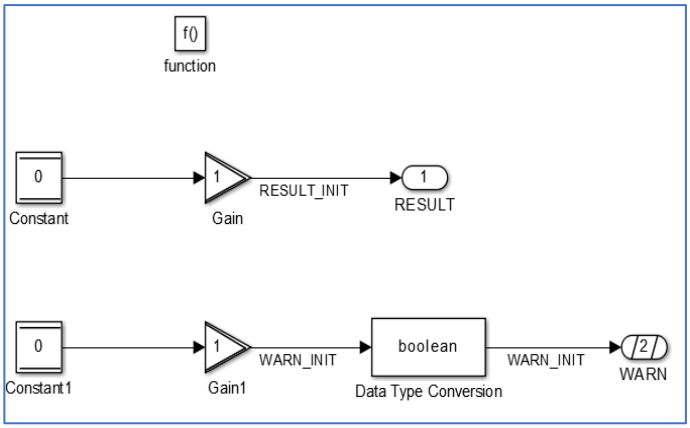
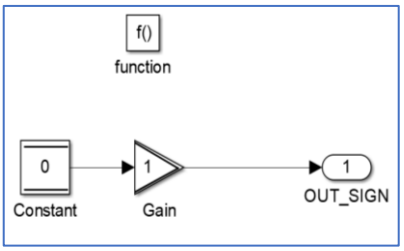


Figura 26 - Logica Stateflow UTET

Il blocco ***STATEFLOW*** è composto da una logica ad albero dove in base al tipo di evento (PowerOn e 5ms) si esegue l'inizializzazione delle variabili o i calcoli nell'ordine Math, Strategy e Filter.





I blocchi di ***INIZIALIZZAZIONE*** quando attivati restituiscono il valore nullo per la grandezza.

Figura 27 - Blocchi INIZIALIZZAZIONE modello UTET

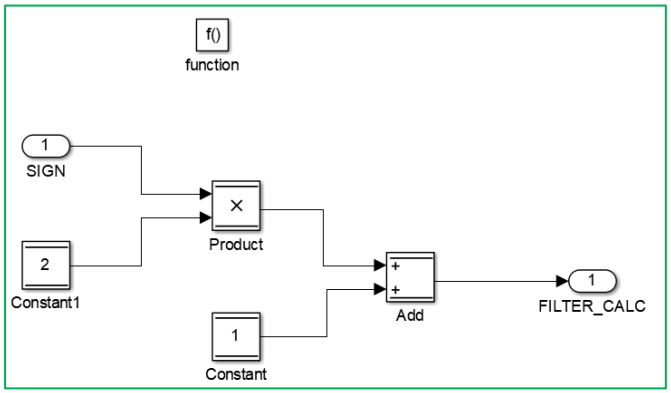


Figura 28- Blocco STRATEGY modello UTET

Il blocco ***STRATEGY*** prendendo il valore del segnale SIGN viene rielaborato con dei passaggi algebrici per ottenere la variabile FILTER_CALC.

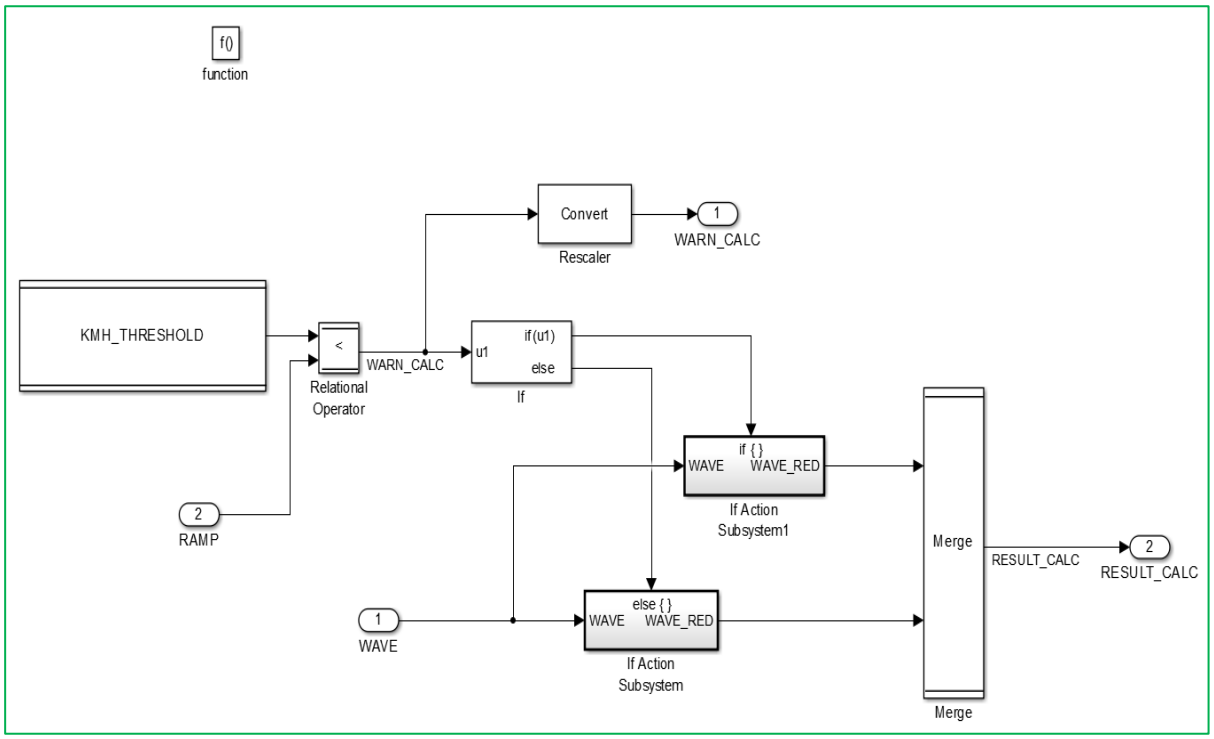


Figura 29-Blocco MATH modello UTET

Il blocco ***MATH*** verifica se il valore di rampa è inferiore a una soglia rappresentata da KMH_THRESHOLD, qualora la condizione venisse verificata altera il valore del segnale WAVE moltiplicandolo per una costante numerica (0.5, non ha un nome specifico), altrimenti trasmette il valore di WAVE senza alterazione. In base a questa valutazione, si ottiene il valore RESULT_CALC.

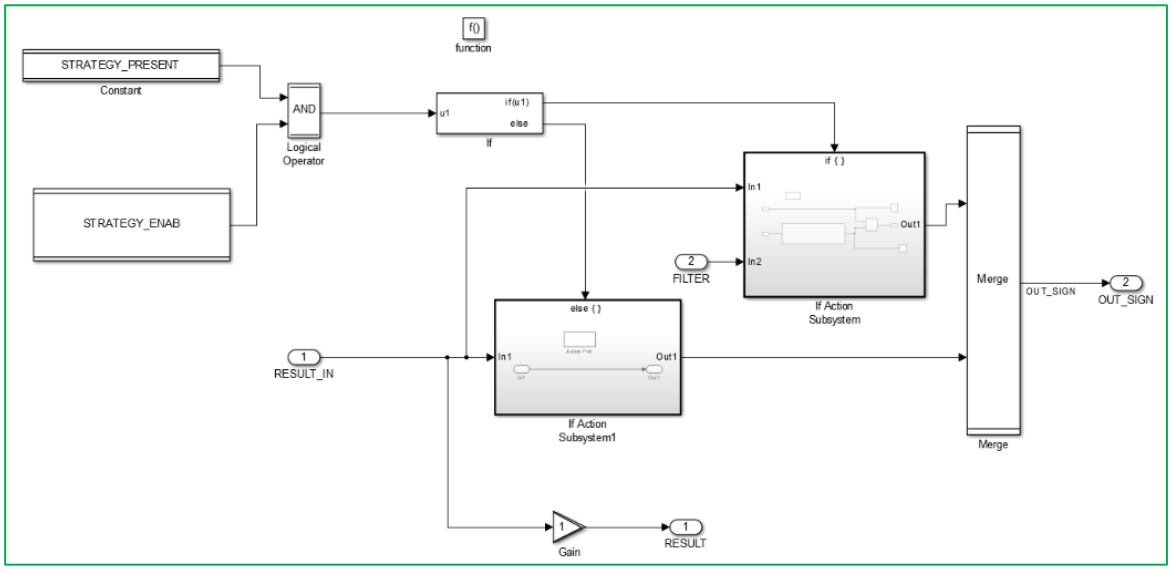


Figura 30 - Blocco FILTER modello UTET

Il blocco ***FILTER*** rielabora le grandezze RESULT_CALC e FILTER_CALC calcolate dai blocchi precedenti. L'output RESULT è la ridefinizione di RESULT_CALC, mentre il valore di OUT_SIGN è definito secondo la verifica logica del valore di STRATEGY_PRESENT e

STRATEGY_ENAB. In caso positivo il valore di filter viene interpolato in una mappa a un ingresso e il risultato viene moltiplicato per la variabile RESULT_IN, altrimenti OUT_SIGN sarà semplicemente uguale alla variabile RESULT_IN.

LE MODIFICHE

Per ottenere un modello il più simile possibile nel concetto funzionale a CPUMPM sono state definite alcune modifiche da apportare al modello originale:

- **Logica a stati** → La logica dello schedulatore va modellata a stati dove in base al tipo di evento si entra nello stato corrispondente dopo aver eseguito la chiamata di esecuzione di tutte le logiche di calcolo.
- **Eventi** → Oltre all'evento di PowerOn e 5ms viene aggiunto un evento 8ms che eseguirà la chiamata per le stesse logiche di calcolo dell'evento 5ms.
- **Stato del sistema** → È stato necessario aggiungere una variabile di sistema che cambia nel tempo e che viene aggiornata durante l'evento 5ms o 8 ms. Per rendere la logica più complessa si è deciso che la variabile di stato (che rappresenta idealmente la condizione di IDLE e quella di RUN) quando assume valore *IDLE* esegue il suo aggiornamento all'evento 5ms mentre se assume valore *RUN* l'aggiornamento sarà eseguito all'evento 8ms
- **Sommatoria** → È stata aggiunta una variabile di output identificata in una sommatoria che cambia il valore da aggiungere in funzione dello stato del sistema e dell'evento che ha generato la chiamata del calcolo.

	IDLE	RUN
5_ms	+1	+5
8_ms	+3	+8

- **Simstep** → Si è deciso di non aver la presenza di eventi concorrenti in quanto il risolutore di matlab poteva decidere di applicare diversamente la risoluzione del modello. È stato deciso di sfasare l'evento di 8ms di 0.5ms, pertanto il simstep è stato modificato da 1 ms a 0.5ms.
- **Alterazione dei calcoli** → Per poter verificare il corretto funzionamento delle logiche si è deciso di alterare i calcoli degli output RESULT e OUT_SIGN rispetto al modello originale, l'alterazione è presente sia per il tipo di stato che per il tipo di evento chiamante. In ogni caso, si possono sempre confrontare i risultati con il modello

originale in quanto sono uguali al caso di evento chiamante 5ms e stato del sistema in *IDLE*.

- **Rielaborazione dei risultati** → Sono stati aggiunti dei Testpoint, ma soprattutto è stato aggiunto un blocco di salvataggio dei valori degli output nel workspace in quanto si è reso necessario uno script per poter confrontare i risultati ottenuti dal modello. Altre opzioni presenti nello script sono la verifica del Coverage e la semplice esecuzione della simulazione del modello.
- **Nome** → Per poter differenziare i due modelli si è deciso di chiamare il modello con la configurazione a stati **UTETS** mentre il modello diviso per runnable **UTETR**

Per poter eseguire la verifica completa del modello e di tutte le sue logiche sono stati identificati 4 scenari di input (Test Pattern). Questi scenari variano alcune caratteristiche degli input tra cui l'inizio dell'esecuzione del segnale di onda quadra e il valore delle variabili "STRATEGY_ENAB" e "STRATEGY_PRESENT".

VARIABILE	UTET_TP_01	UTET_TP_02	UTET_TP_03	UTET_TP_04
SIMSTEP	0.0005	0.0005	0.0005	0.0005
TIMESTART	0	0	0	0
TIMESTOP	10	10	10	10
STRATEGY_PRESENT	1	1	0	0
STRATEGY_ENAB	1	0	1	0
REDUCTION	0.5	0.5	0.5	0.5
delay_IDLE_RUN	0.0170	0.0170	0.0070	0.0070
delay_5ms	0.0100	0.0150	0.0150	0.0010
delay_8ms	0.0105	0.0105	0.0105	0.0015
delay_PowerOn	0.0050	0.0050	0.0050	0.0050
KMH_TRESHOLD	150	150	150	150

Gli eventi chiamanti sono modellati come eventi temporali, l'ordine della loro presenza non segue in tutti gli scenari un senso logico temporale (in TP_02 e TP_03 ho prima la presenza dell'evento 8ms) questo perché si devono pensare come due possibili eventi con presenza randomica.

UTET STATI – UTETS

SIGNAL_GENERATION

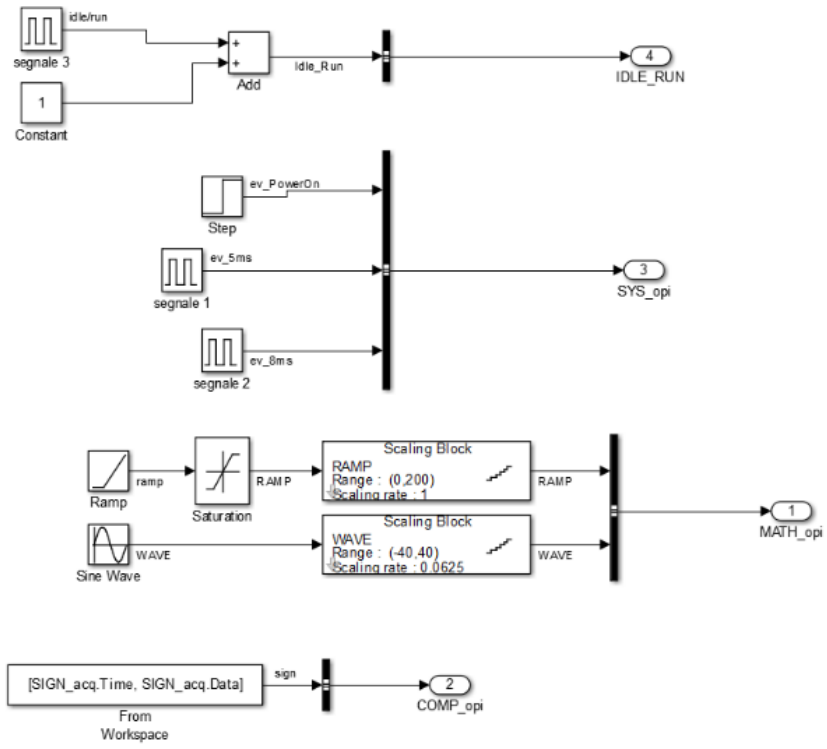


Figura 31 - UTETS SignalGeneration

La modellazione degli input per quanto riguarda i segnali già presenti non è stata modificata, mentre il segnale di IDLE_RUN è stato definito come un'onda quadra che varia da 1 a 2, di periodo 34ms e il picco corrispondente allo stato di Run di 17 ms. La scelta della variazione tra 1 e 2 è stata fatta in modo da avere un valore di output pari a 0 quando non è ancora stato valutato lo stato del sistema. L'evento 8ms invece è un'onda quadra con periodo di 8ms e picco di 4ms.

SIMULATION RESULT

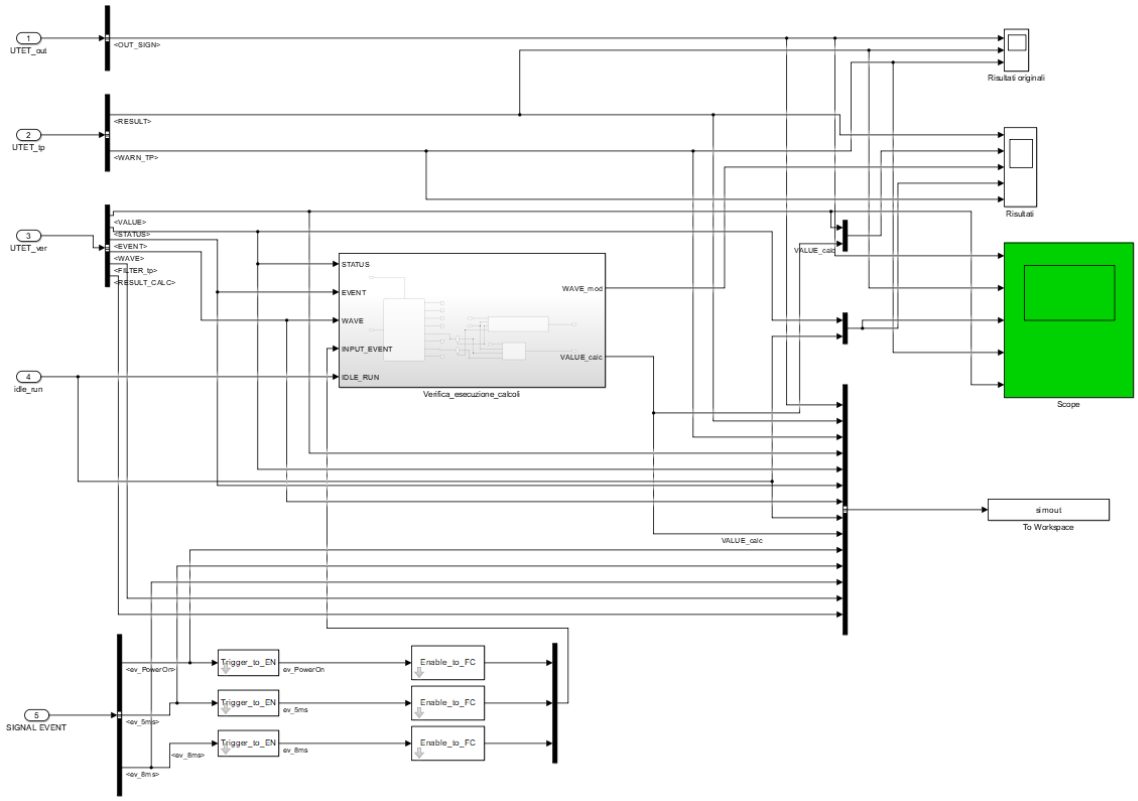


Figura 32 - UTETS SimulationResult

Il blocco *RESULT* è stato ampliato in modo da avere la possibilità di controllare meglio gli effetti delle modifiche e raccogliere tutti i vettori di output e testpoint utili all'analisi della funzionalità del modello (blocco "To_Workspace"), gli output del modello sono stati verificati tramite il blocco Scope verde.

I cinque grafici che si visualizzano sono relativi alle grandezze più significative nel modello:

- OUT_SIGN → Output principale del modello, il suo valore viene calcolato ad ogni evento, viene alterato in funzione dello stato del sistema
- RESULT → Testpoint principale, viene definito ad ogni evento, il suo valore viene alterato sia in funzione dello stato del sistema che in funzione dell'evento.
- IDLE_RUN → rappresenta lo stato del sistema effettivo in rosso mentre in blu è rappresentato lo stato del sistema secondo le logiche di calcolo.
- WARN_TP → rappresenta l'attivazione della logica di warning quando il valore di RAMP è superiore a KMH_TRESHOLD.
- VALUE → risultato della nuova variabile generata dalla sommatoria in funzione dell'evento e in funzione dello stato

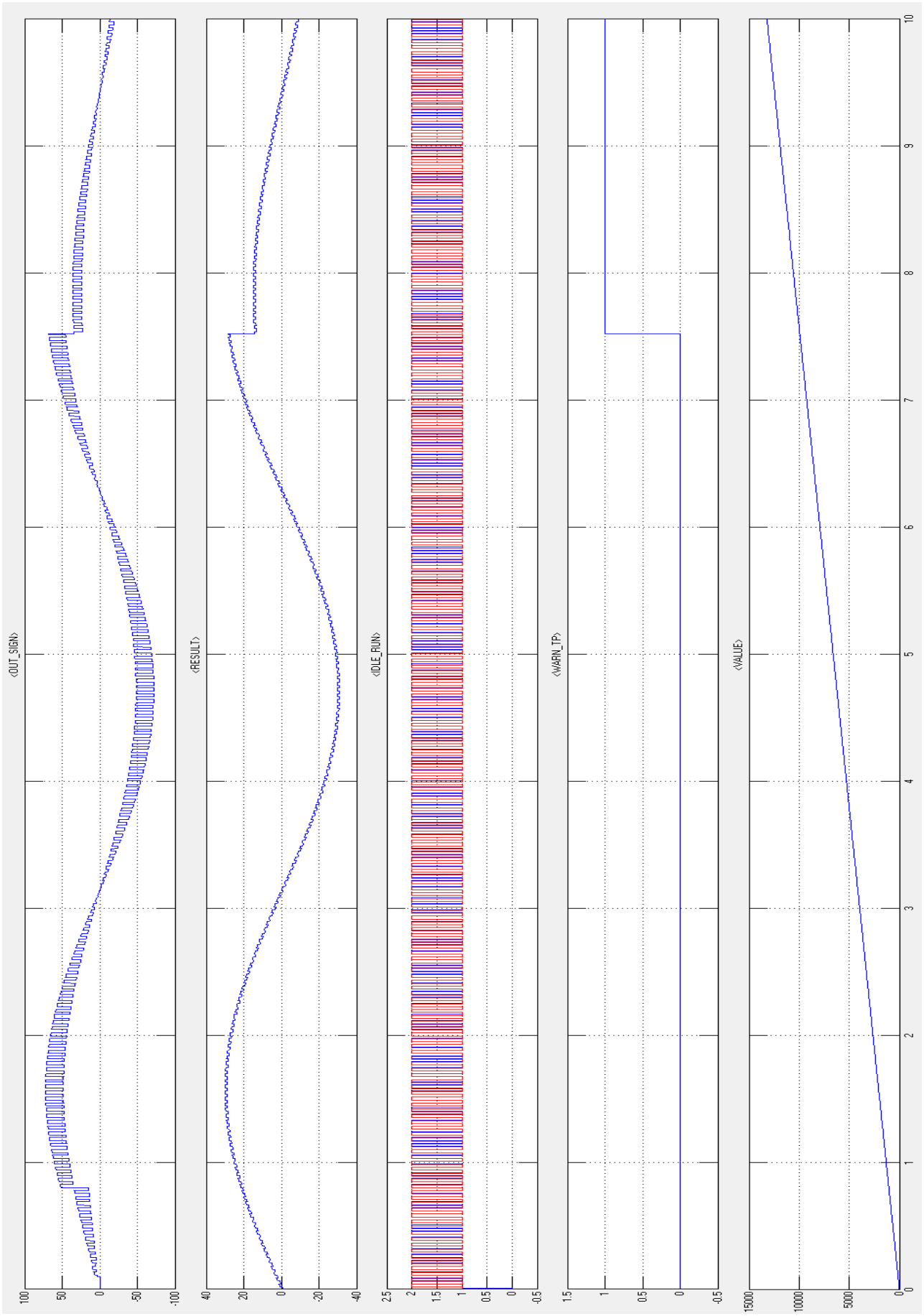


Figura 33 - UTETS esempio grafici output

Di seguito, il dettaglio dei grafici.

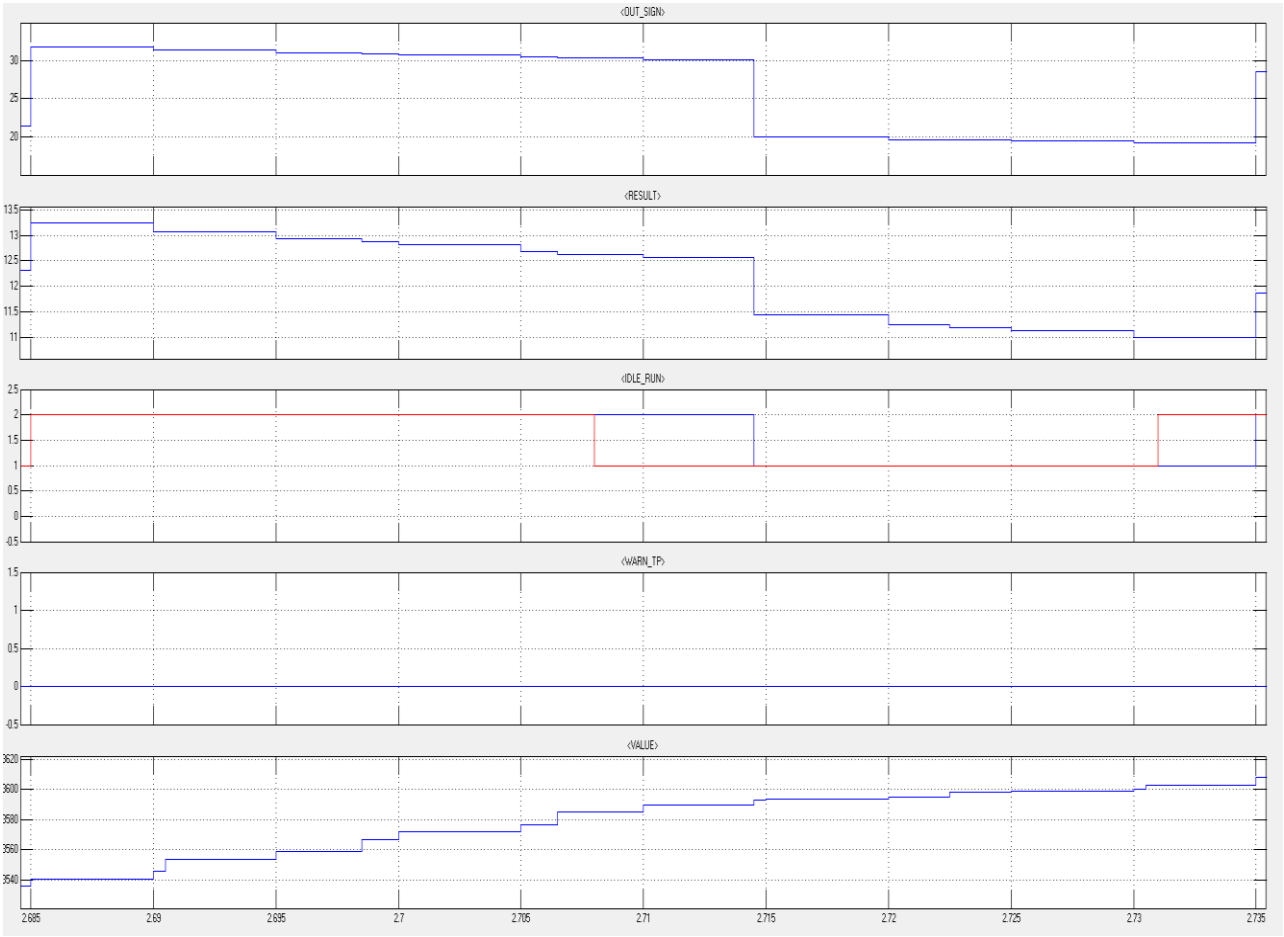


Figura 34 - Dettaglio grafici output

Si possono notare gli effetti della variazione dello stato da *RUN* a *IDLE*, si può vedere come la curva OUT_SIGN presenti un gradino al momento dell'aggiornamento del sistema dello stato. Lo stesso gradino è presente sul grafico RESULT. É inoltre possibile apprezzare la variazione in funzione dell'evento chiamante, e le stesse considerazioni si possono fare per il grafico VALUE. L'aggiornamento dello stato si può notare sia avvenuto con un ritardo rispetto alla variazione "reale", questo perché il primo evento dopo la variazione è un 5ms che quando lo stato del sistema è in RUN non aggiorna la variabile di stato (cosa che invece avviene all'evento successivo).

Per completezza, si riportano i risultati ottenuti dai singoli Test Pattern. Si può notare come la simulazione più interessante sia quella del primo Test Pattern, per motivi pratici pertanto tutti i prossimi grafici di output saranno inerenti al primo scenario.

UTETS_TP_1

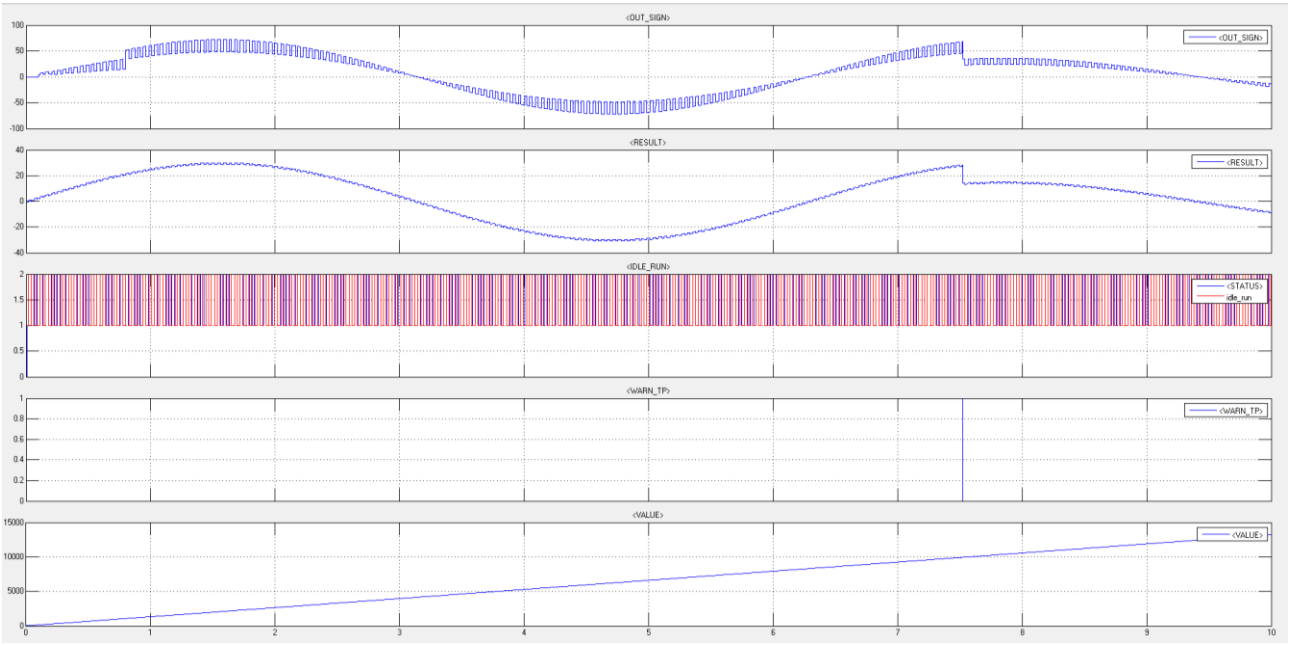


Figura 35 - Grafici output TestPattern 1

UTETS_TP_2

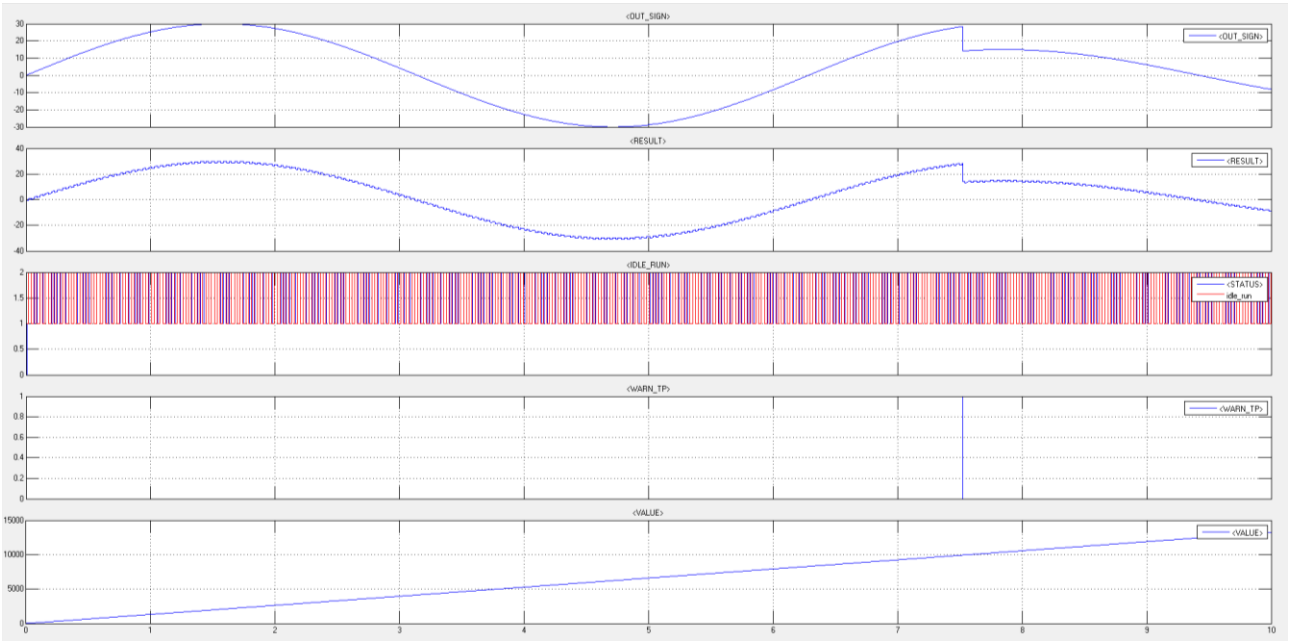


Figura 36 - Grafici output TestPattern 2

UTETS_TP_3

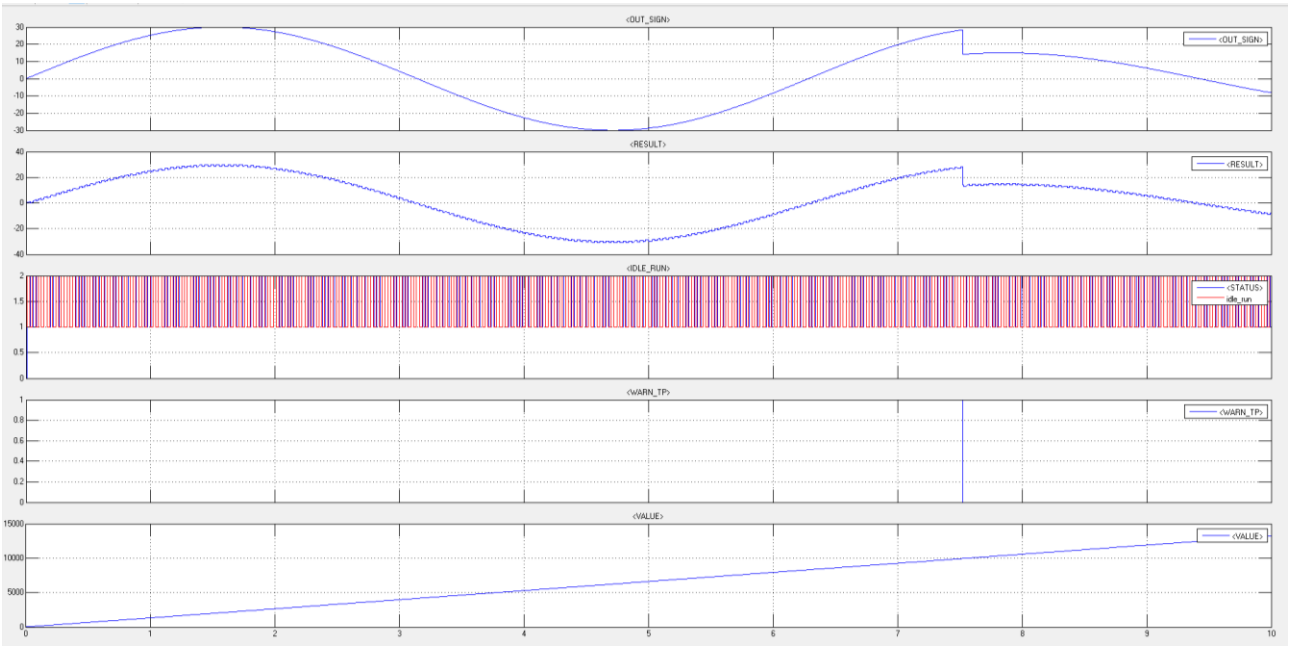


Figura 37 - Grafici output TestPattern 3

UTETS_TP_4

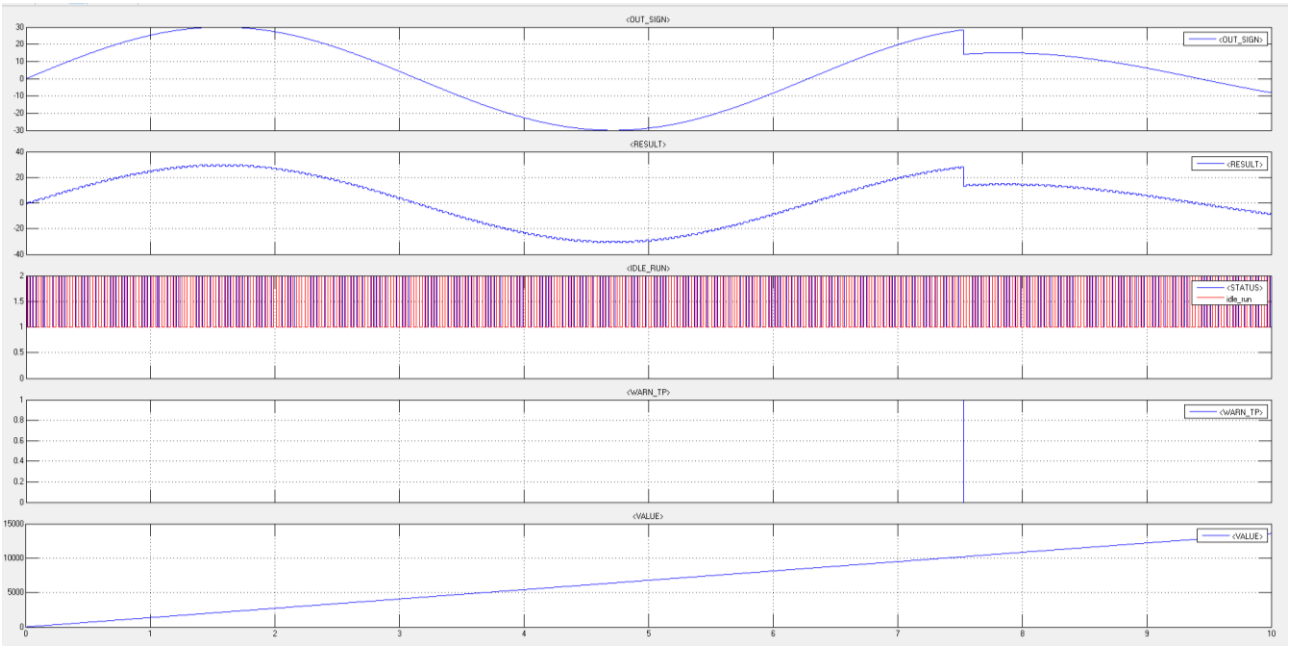


Figura 38 - Grafici output TestPattern 4

UTETS_ac

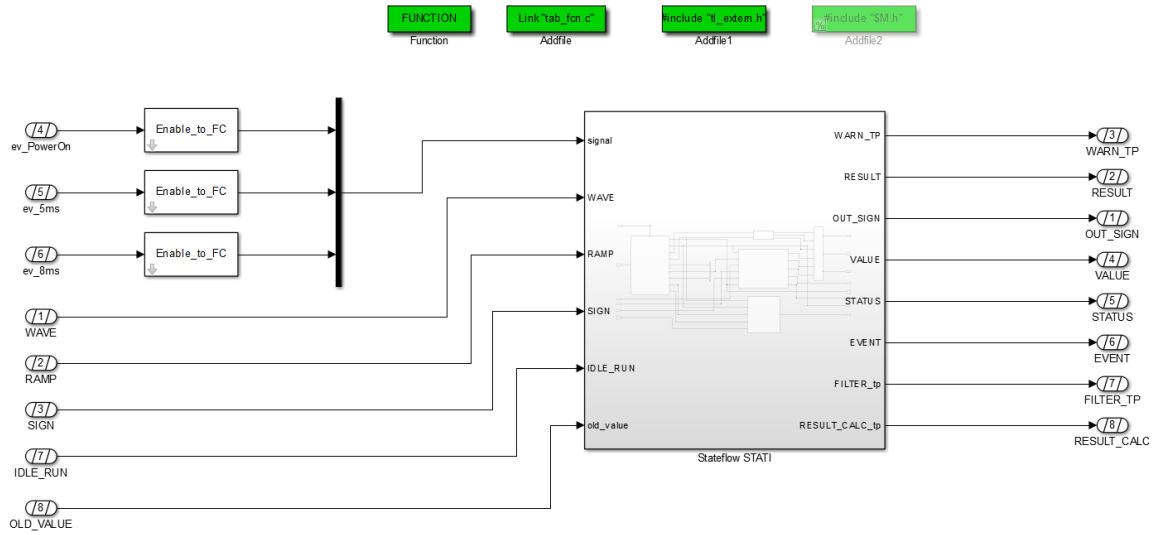


Figura 39 - UTETS_ac

Questo livello differentemente dal caso originale è stato condensato in un unico subsystem chiamato Stateflow_STATI. Questo subsystem ha tutte le modifiche di calcolo descritte precedentemente.

Sono presenti al suo interno 4 blocchi (oltre al subsystem di Merge):

- STATEFLOW => usato come schedulatore con logica a stati
- INIZIALIZZAZIONE => serve ad inizializzare le grandezze del modello originale
- CALC => contiene tutte le logiche di calcolo del modello originale modificate
- SOMMATORIA => è un blocco che esegue il calcolo della sommatoria

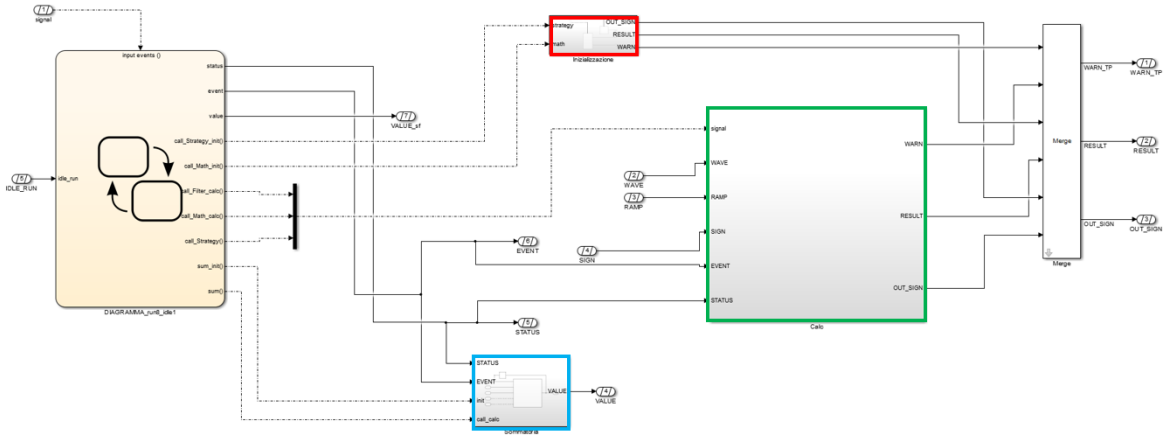


Figura 40 - Blocco Stateflow STATI

Di seguito si espongono nel dettaglio i quattro blocchi presenti.

La chart STATEFLOW stati

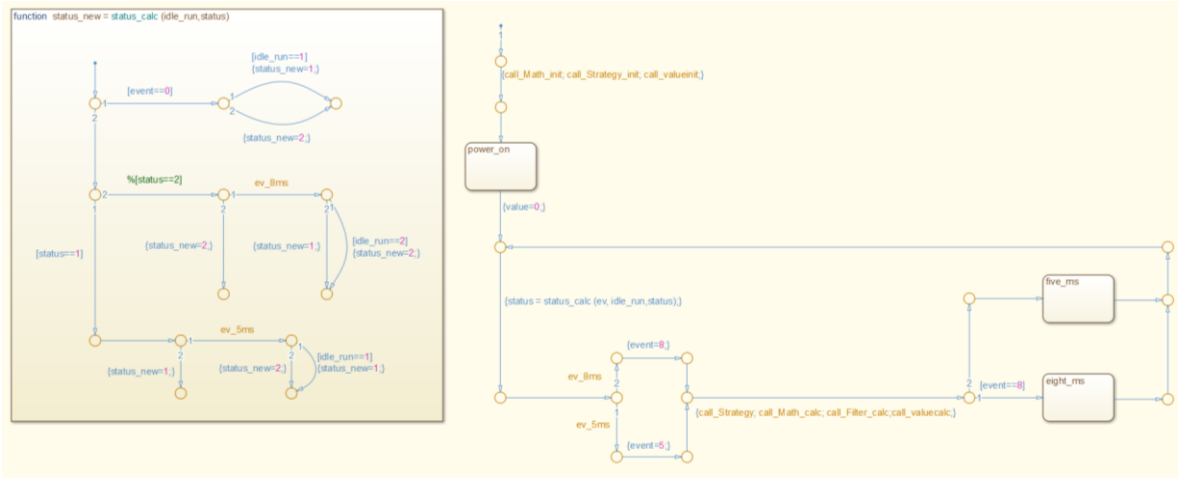


Figura 41 - Chart STATI

ha la presenza di 3 stati (power_on, five_ms, eight_ms) e una function utile per l'aggiornamento dello stato del sistema (Idle-Run). Al primo evento ho l'ingresso nello stato di power_on con l'inizializzazione di tutte le grandezze. Successivamente con la presenza di un evento eseguo prima l'analisi dello stato del sistema, definisco il tipo di evento che è avvenuto ed eseguo i calcoli previsti. Come ultimo passaggio entro nello stato collegato all'evento chiamante in attesa di un nuovo evento dove rieseguo l'eventuale aggiornamento dello stato del sistema e successivamente i calcoli.

Il blocco di INIZIALIZZAZIONE

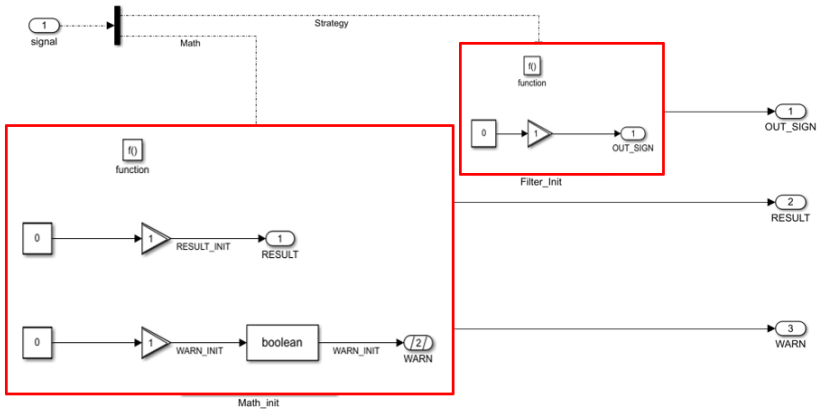


Figura 42 - UTETS - Blocchi inizializzazione

Non è cambiato dal modello originale. L'inizializzazione della sommatoria non è stata impostata in questo blocco perché al momento della modellazione aggiuntiva era più facile da gestire tutto in un unico blocco.

Il blocco di SOMMATORIA

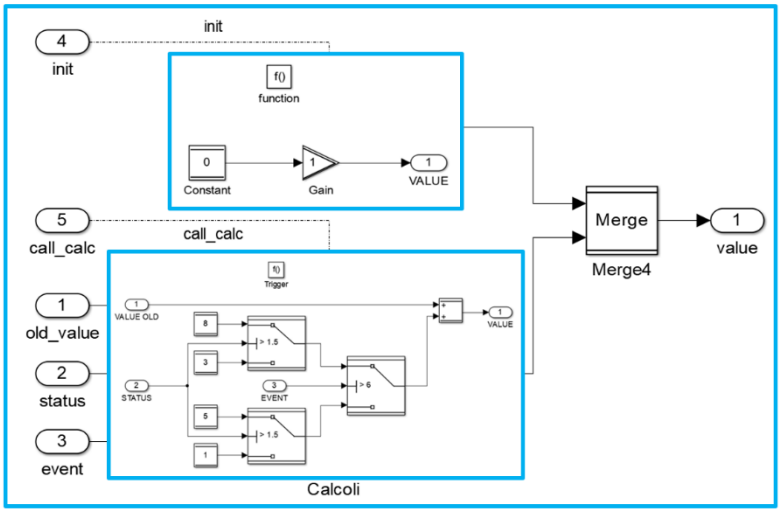


Figura 43 - UTETS Blocco SOMMATORIA

È stato modellato con questa forma: la chiamata attiva il blocco dove in funzione del valore logico dello stato e dell'evento si esegue la somma del valore calcolato al simstep precedente (che viene fatto ricircolare con il blocco unit-delay, presente al di fuori del blocco "ac").

Il blocco CALC

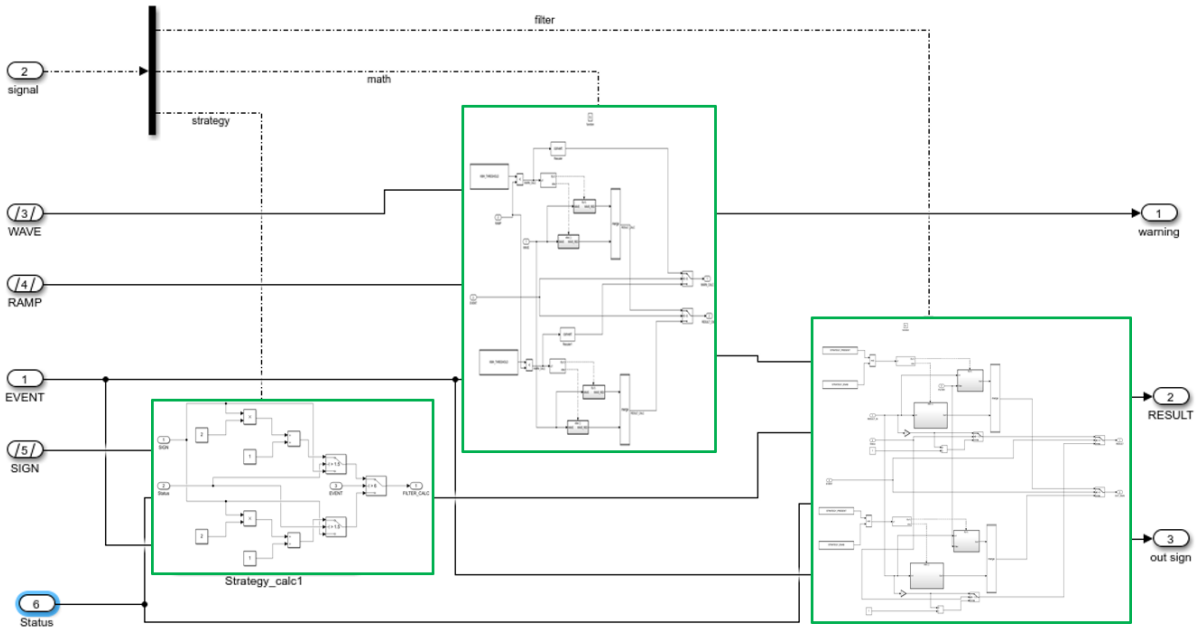
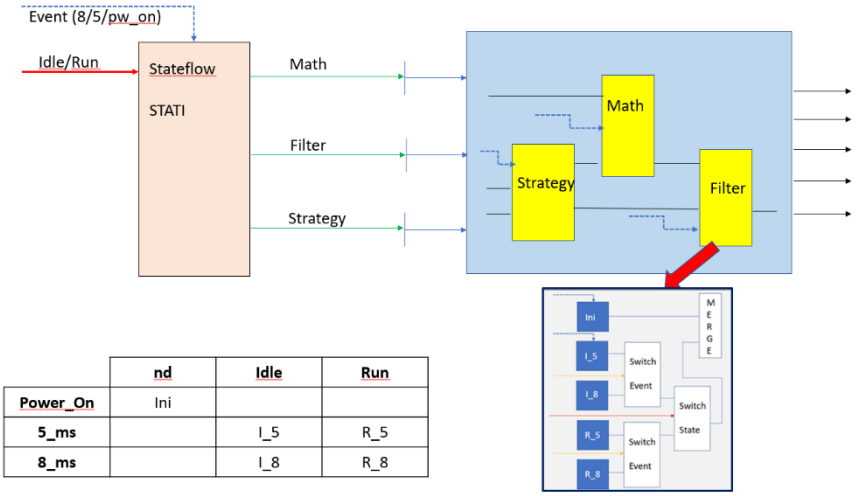


Figura 44 - UTETS blocco CALC

È il blocco con i calcoli originali divisi in tre subsystem "raddoppiati" internamente in modo da avere la presenza in parallelo dello stesso tipo di calcolo (con opportune variazioni) in funzione dell'evento chiamante. Prima dell'output ho la presenza di uno switch che determina quale è il risultato da comunicare in funzione dell'evento mentre l'alterazione dei calcoli in funzione dello stato è presente all'interno della logica dedicata all'evento.

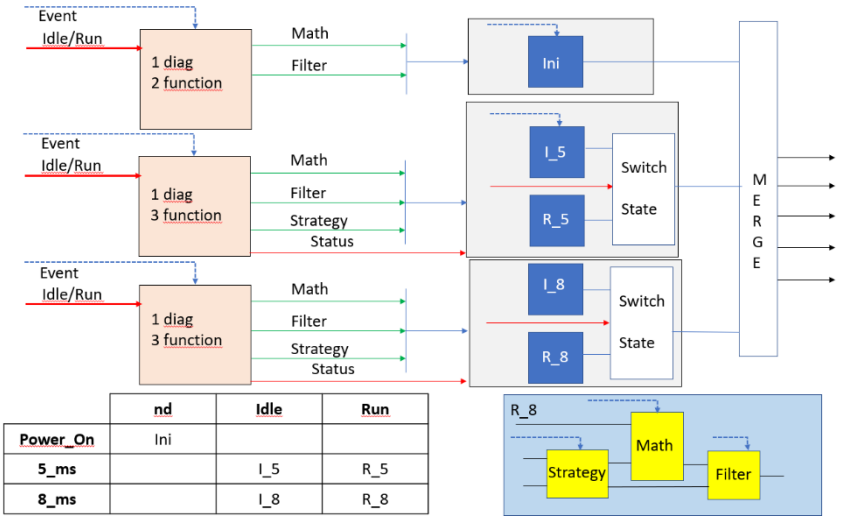
UTET – evoluzione del modello da stati a runnable



	nd	Idle	Run
Power_On	Ini		
5_ms		I_5	R_5
8_ms		I_8	R_8

Figura 46 - Schema esecuzione calcoli logica a stati

Il concetto di evoluzione a Runnable partendo da una logica a Stati richiede una modifica profonda del design del modello in termini architetturali ed implementativi. Infatti, considerando la struttura presente nella prima immagine, si può raffigurare la funzione dello stateflow come quella di un imbuto che riceve tutti gli eventi e successivamente fa eseguire le chiamate secondo la logica interna attivando i subsystem di calcolo (che forniscono un risultato in funzione dello stato del sistema e dell'evento chiamante), questo implica l'impossibilità di avere logiche indipendenti tra loro. Nella seconda immagine si può notare come la catena di calcolo di un singolo evento non si va a intersecare mai con la logica di calcolo relativa ad un altro evento. Ovviamente questa indipendenza richiede un percorso di divisione che non faccia perdere informazioni importanti tra un evento e l'altro (pensiamo ad esempio all'aggiornamento dello stato, che avviene su più eventi) e una struttura solida della catena di calcolo.



	nd	Idle	Run
Power_On	Ini		
5_ms		I_5	R_5
8_ms		I_8	R_8

Figura 45 - Schema esecuzione calcoli logica a runnable

Pertanto, partendo dal modello a stati si è deciso di eseguire la modifica del modello per gradi, apportando prima delle modifiche allo stateflow e successivamente modificando i blocchi di calcolo.

La prima operazione necessaria all'evoluzione della modellazione è stata quella di ottenere una chart di logica stateflow senza la presenza di Stati che rispecchiasse il più possibile la logica già presente.

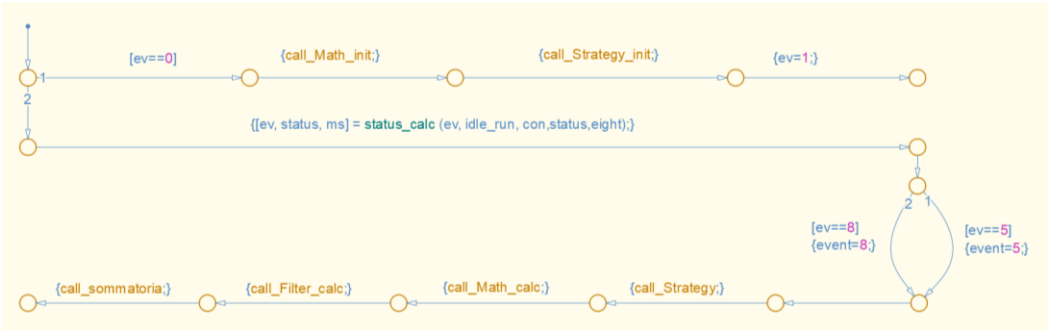


Figura 47 - Logica stateflow a diagramma

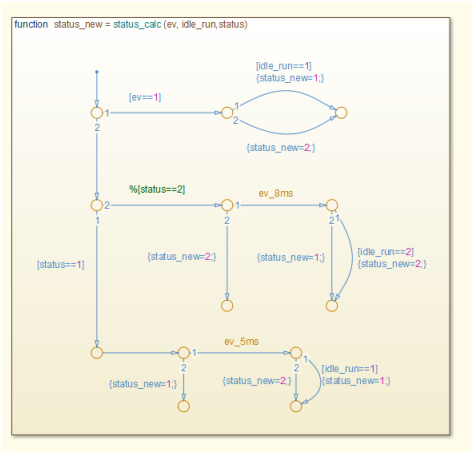


Figura 48 - Stateflow Function

Il tipo di ramo da percorrere è definito dalla variabile "ev", infatti all'inizio devia l'esecuzione verso il ramo dell'inizializzazione del sistema che avviene con la presenza del primo evento. I calcoli invece vengono eseguiti solamente in presenza di un evento 5ms o 8ms, anche in questo caso è stato più utile raggruppare la logica di analisi dello stato del sistema con una function. Questa forma intermedia ha permesso la realizzazione della forma MULTIALBERO senza particolari complicazioni. Benché possa sembrare più "dispersiva" la logica MULTIALBERO consente un passaggio importante nella divisione delle catene di calcolo. Infatti, nello stateflow sono presenti 3 logiche separate di calcolo, ciascuna attivata esclusivamente da un unico evento.

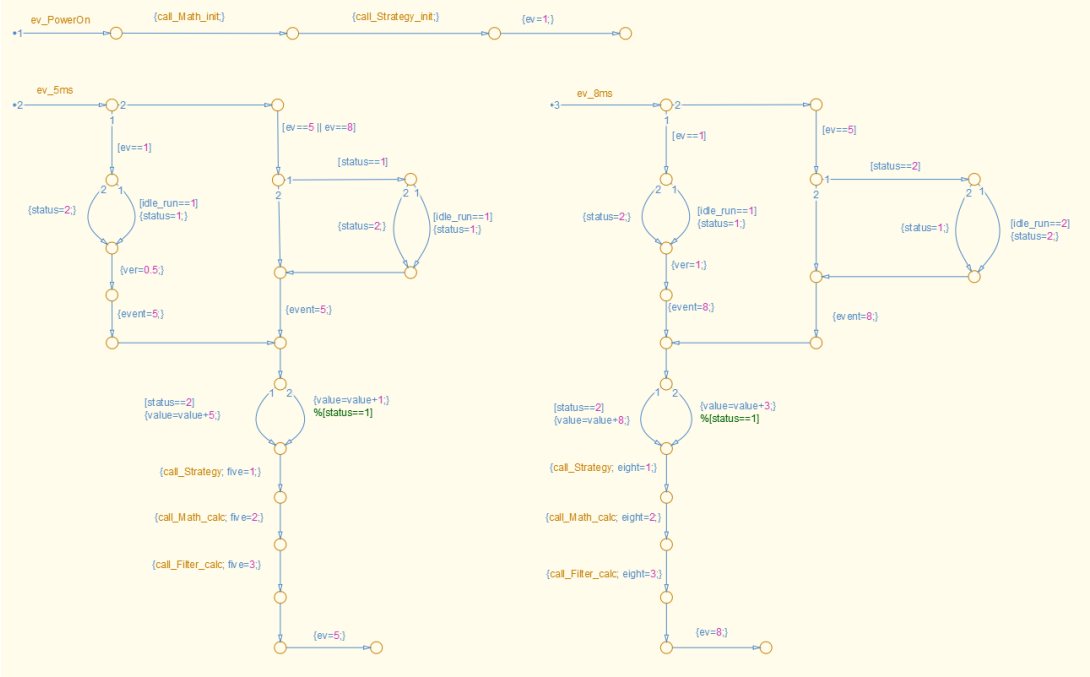


Figura 49 - Logica Multialbero

Questo comporta la definizione di quali parti di logica sono da attivare per il singolo evento e impone la necessità di individuare eventuali variabili condivise tra i diagrammi in modo da poter avere "memoria" delle logiche eseguite in un altro albero.

Il rischio che si corre nella modellazione di una logica multialbero solitamente è quello di ottenere una logica dispersiva in quanto si tende a perdere di vista eventuali rami condivisi. Un esempio di questo problema è rilevabile nella figura seguente: in tutte le parti evidenziate sono presenti le stesse Function Call che attivano i medesimi subsystem nello stesso ordine. Pertanto i tre rami possono esser addensati in uno unico al fine di evitare una modellazione rindondante.

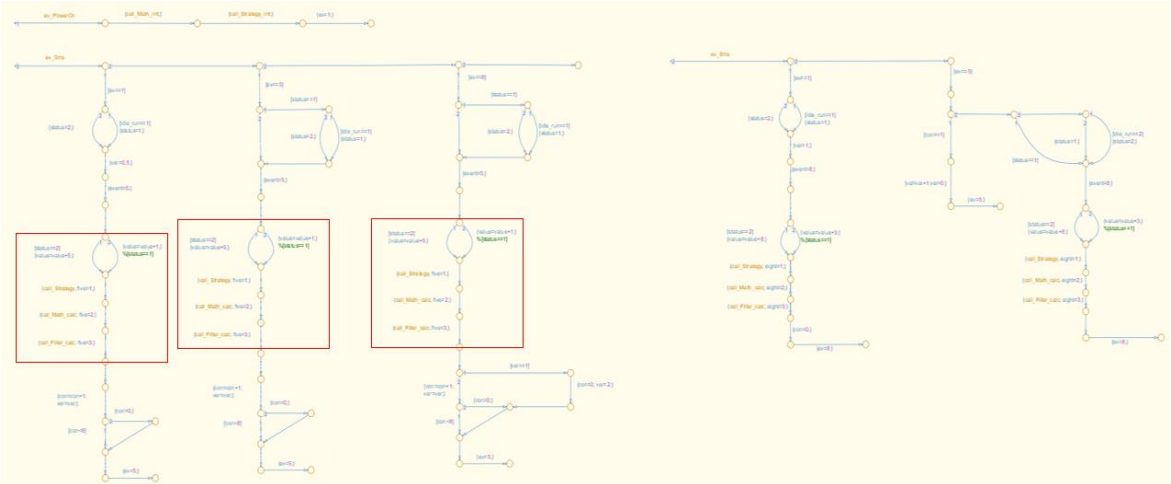


Figura 50 - Logica multialbero non ottimizzata

Con la logica multialbero è possibile quindi ottenere una chart stateflow dedicata per ogni singolo evento. Ottenuta questa combinazione si è passati alla modellazione anche dei subsystem di calcolo in maniera da ottenere la catena di calcolo indipendente. Al fine di semplificare ulteriormente la logica si è deciso di portare al di fuori dello stateflow qualsiasi tipo di calcolo che non fosse l'esecuzione delle chiamate delle function.

Successivamente sono state individuate le variabili, di cui è necessario conoscere il risultato ottenuto all'evento di calcolo precedente, in modo da risolvere le operazioni in modo coerente:

- STATUS = che descrive lo stato del sistema (Idle o Run)
- VALUE = che è il valore della sommatoria

L'ultimo passaggio è stato quello di dividere i subsystem di calcolo in blocchi dedicati al calcolo di una precisa Function Call. In questo modo si è ottenuto un modello con un esploso di tutti i calcoli presenti.

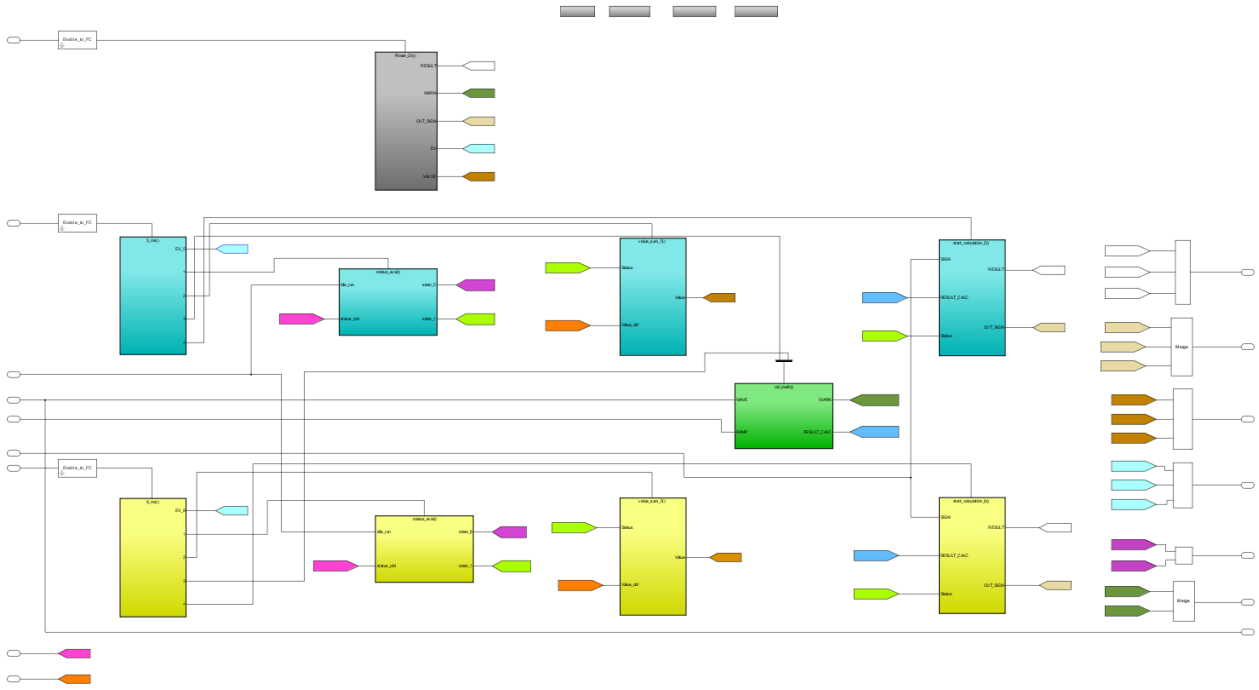


Figura 51 - Divisione preliminare runnable

Le scelte cromatiche e di disposizione dei blocchi sono state fatte al fine di rendere più intuitiva la lettura del modello.

I tre colori principali (grigio, azzurro e giallo) rappresentano le catene di calcolo relative agli eventi *PowerOn*, *5ms* e *8ms*. Il blocco verde invece è una logica comune in tutti i suoi elementi tra la catena di calcolo *5ms* e *8ms*. l'esecuzione della catena di calcolo è organizzata con il seguente ordine:

- L’evento entra nel primo blocco e attiva la logica stateflow e aggiorna il valore di ev
- Si esegue la chiamata del blocco STATUS dove si aggiorna lo stato del sistema
- Si esegue la chiamata del blocco sommatoria e si ottiene il valore di VALUE
- Si esegue il calcolo del blocco MATH_CALC
- Si esegue il calcolo del blocco STRATEGY e FILTER, in questo caso sono stati inglobati in un unico blocco

La presenza di un blocco comune, anche se le catene di calcolo non possono intrecciarsi, rappresenta un “collo di bottiglia” nella modellazione. La soluzione è quella di sdoppiarlo in quanto nella modellazione per runnable non sono ammesi blocchi condivisi nelle chiamate di esecuzione da due runnable differenti. Dovendo mantenere “l’identità di un blocco condiviso” e volendo ottenere all’interno del codice generato una function dedicata è stato deciso di definirlo come blocco di libreria. In questo modo, qualsiasi variazione nella sua modellazione viene aggiornata in tutto il modello e non si rischia di avere blocchi con la stessa funzione che eseguono calcoli in maniera differente tra loro.

UTET RUNNABLE - UTETR

Il modello finale ottenuto ha le parti di signal generation e result uguali al modello a stati, l'unica parte modificata è quella relativa al blocco "utet_ac". In questa nuova forma, tutte le catene di calcolo sono state divise e si possono notare i tre blocchi distinti delle runnable con all'interno tutta la logica di calcolo dedicata.

La struttura della modellazione è definita con questa distribuzione spaziale per una migliore interpretazione, a sinistra son presenti tutti gli input al modello, separati dagli input sono stati messi gli eventi chiamanti collegati direttamente con il subsystem dedicato, nella zona centrale vi sono i blocchi con la catena di calcolo mentre a destra sono presenti tutti i blocchi di aggiornamento della variabile tramite blocchi Merge.

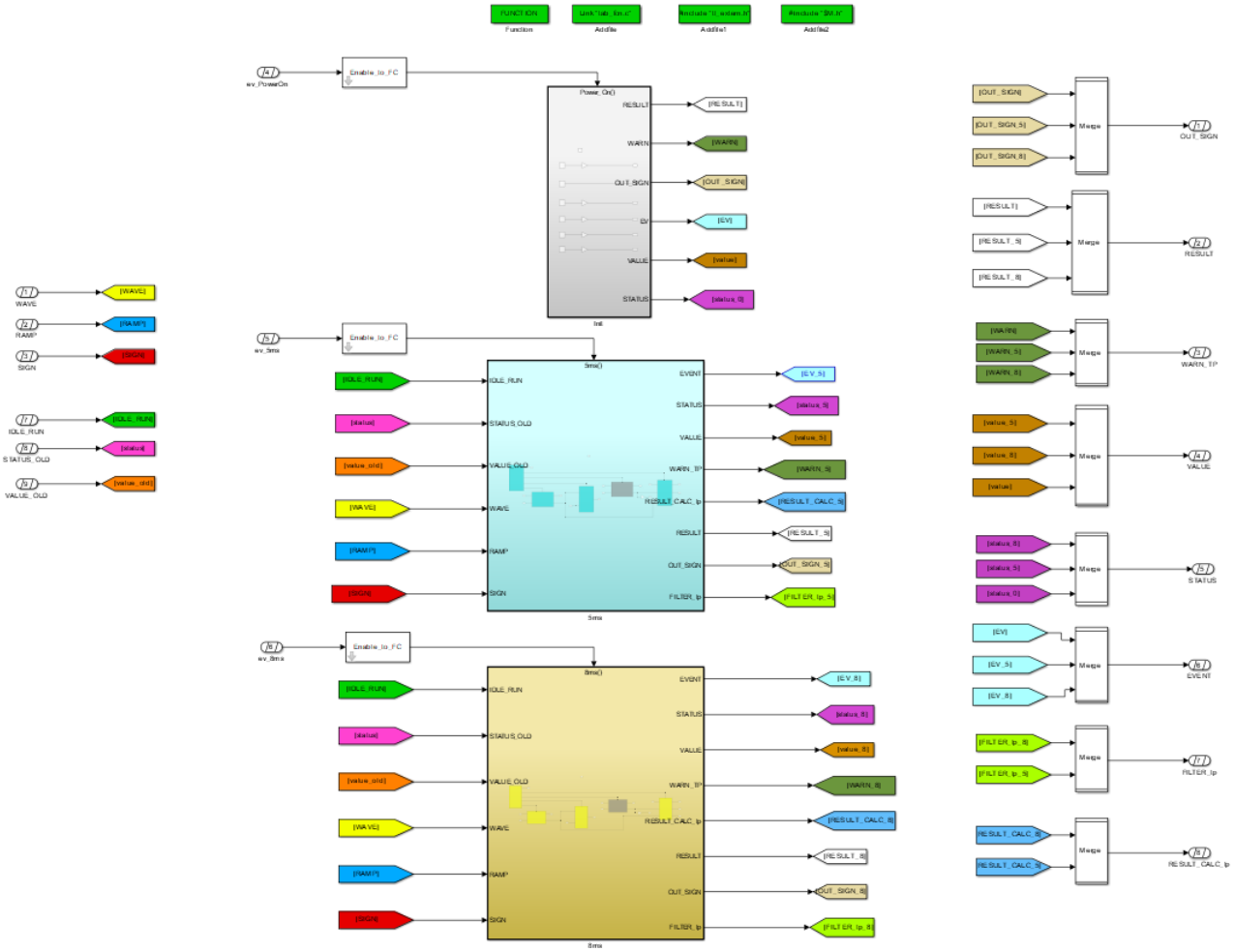


Figura 52 - Divisione runnable definitiva

Come per il modello a Stati, si procede con l'analisi dei singoli subsystem

Il blocco di INIZIALIZZAZIONE

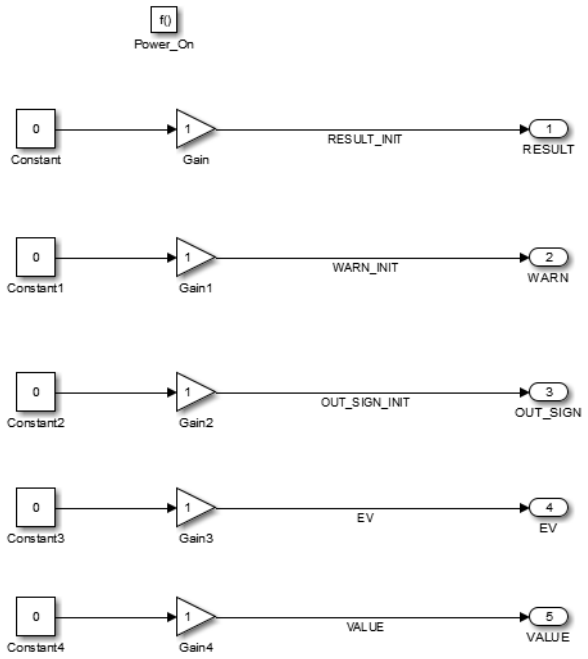


Figura 53 - Blocco inizializzazione

Come per il modello a stati e per il modello originale il blocco di inizializzazione viene attivato per effetto di un evento power_on che impone solamente tutte le grandezze a 0.

Il blocco 5ms e 8ms

Questi blocchi sono uguali, entrambi hanno la stessa struttura formata da un primo blocco che ha il compito di aggiornare la variabile “ev” e definire l’ordine delle chiamate dei vari subsystem. Con questa struttura è stata garantita sia la precedenza logica di calcolo in quanto il blocco viene tutto attivato solamente alla presenza della Function Call dedicata e non alla presenza dell’evento chiamante.

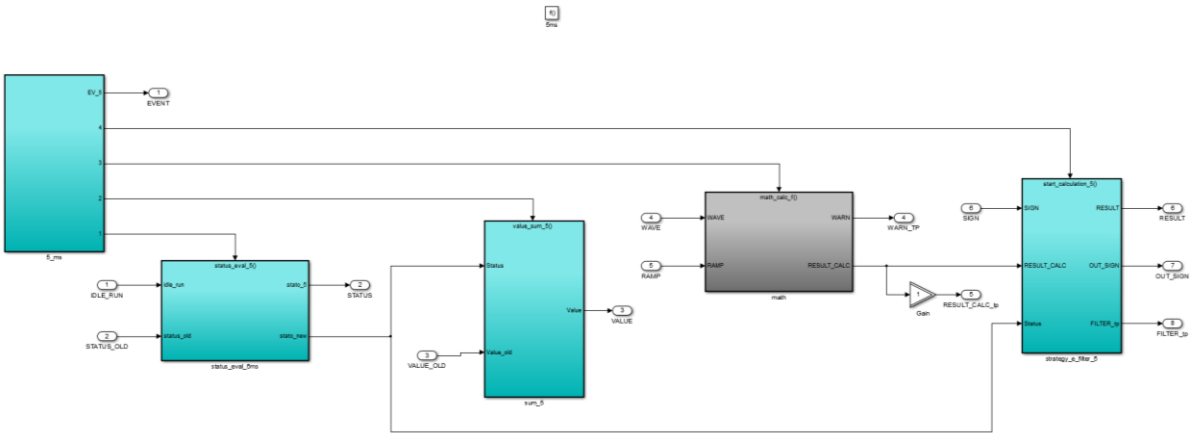


Figura 54 - Distribuzione blocchi calcolo in una runnable

L’ordine di attivazione è quello da sinistra a destra, il primo blocco (unico attivato dall’evento) esegue l’aggiornamento della variabile “ev” e schedula le chiamate per i blocchi successivi. Il primo blocco attivato è quello relativo all’aggiornamento dello stato del sistema, il blocco successivo si occupa dell’aggiornamento del valore della sommatoria. Il blocco grigio (terzo nell’ordine di attivazione) è la parte di logica comune tra le due runnable e corrisponde al blocco math_calc del modello originale. L’ultimo blocco invece racchiude gli ultimi due blocchi appartenenti al modello originale: Strategy e Filter. Questi due blocchi vengono attivati nell’ordine desiderato grazie ad uno stateflow che determina le precedenze.

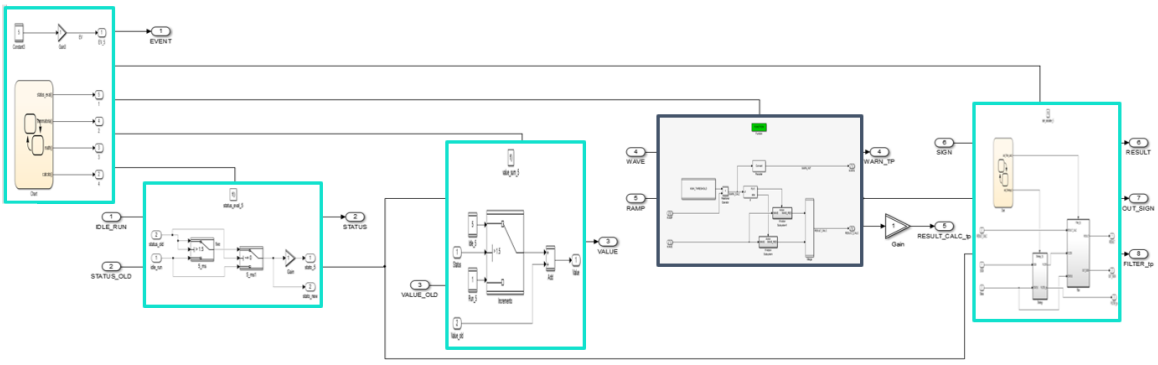


Figura 55 - Dettaglio del contenuto dei blocchi della runnable

PRIMO BLOCCO

Il primo blocco viene attivato dall’evento chiamante della runnable, ha due obiettivi: aggiornare il tipo di evento presente e schedulare l’ordine della catena di calcolo. Per definire l’ordine di esecuzione è stato usato un blocco stateflow con solamente le Function Call nell’ordine desiderato.

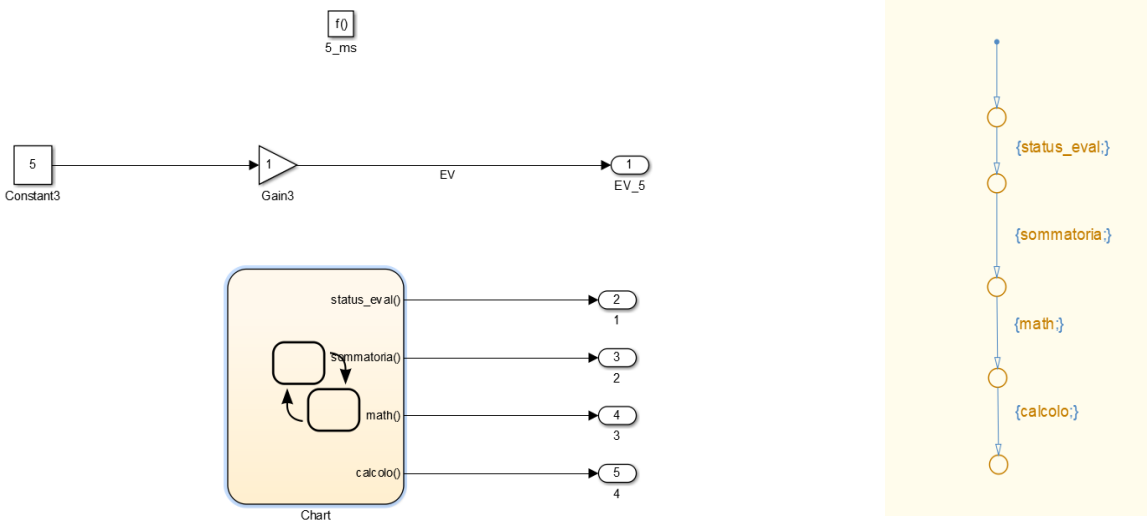


Figura 56 - Primo blocco

SECONDO BLOCCO

Il blocco “status_eval_5ms” ha lo scopo di verificare e aggiornare lo stato del sistema. In funzione dello stato presente al simstep precedente (status_old) si definisce se aggiornare lo stato (switch “5_ms”), se non è mai stato definito lo stato (status_old è nullo) con lo switch “init” si ottiene lo stato del sistema. Le uscite sono due, una come che fungerà da testpoint per il controllo e una per comunicare lo stato presente per i calcoli successivi[9].

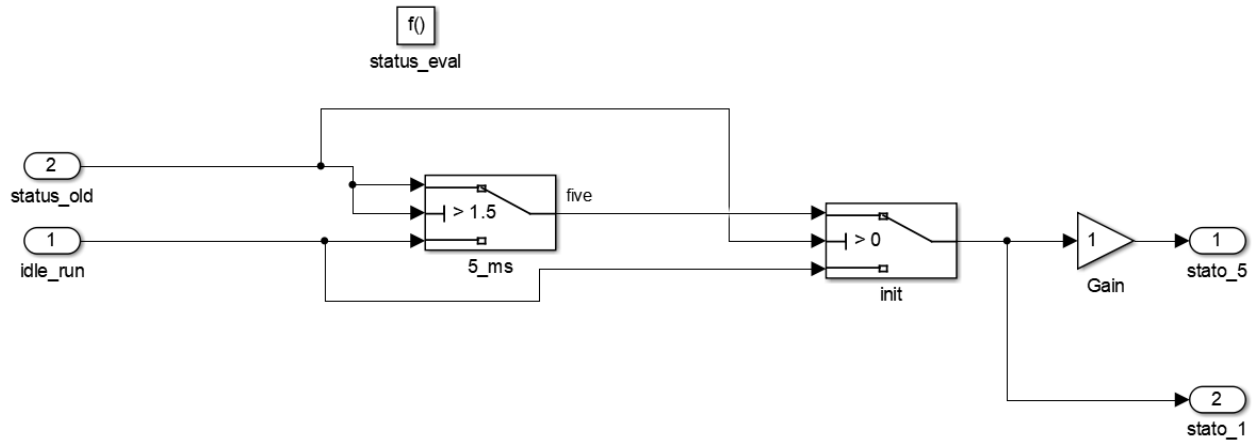


Figura 57 - Logica secondo blocco

TERZO BLOCCO

Il blocco “value_sum” è la parte che esegue la sommatoria, in funzione dello stato e dell’evento chiamante. Nel dettaglio il blocco inerente all’evento 5ms.

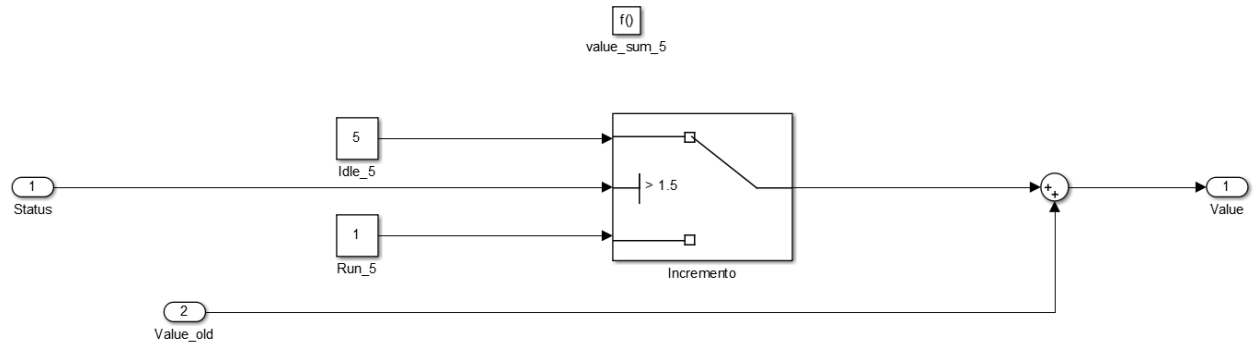


Figura 58 - Logica terzo blocco

[9] Quando un segnale deve essere portato ad un Merge non può essere usato anche in catene di calcolo parallele rispetto al merge. Si deve separare i segnali (con un blocco di Gain o di Data Conversion), portandone uno al Merge e l’altro nel continuo della catena di calcolo.

QUARTO BLOCCO

Il blocco corrisponde al blocco math (condiviso tra le runnable 5ms e 8ms) è stato trasformato in function condivisa. Esegue il calcolo della variabile WARN e RESULT_CALC come già descritto precedentemente.

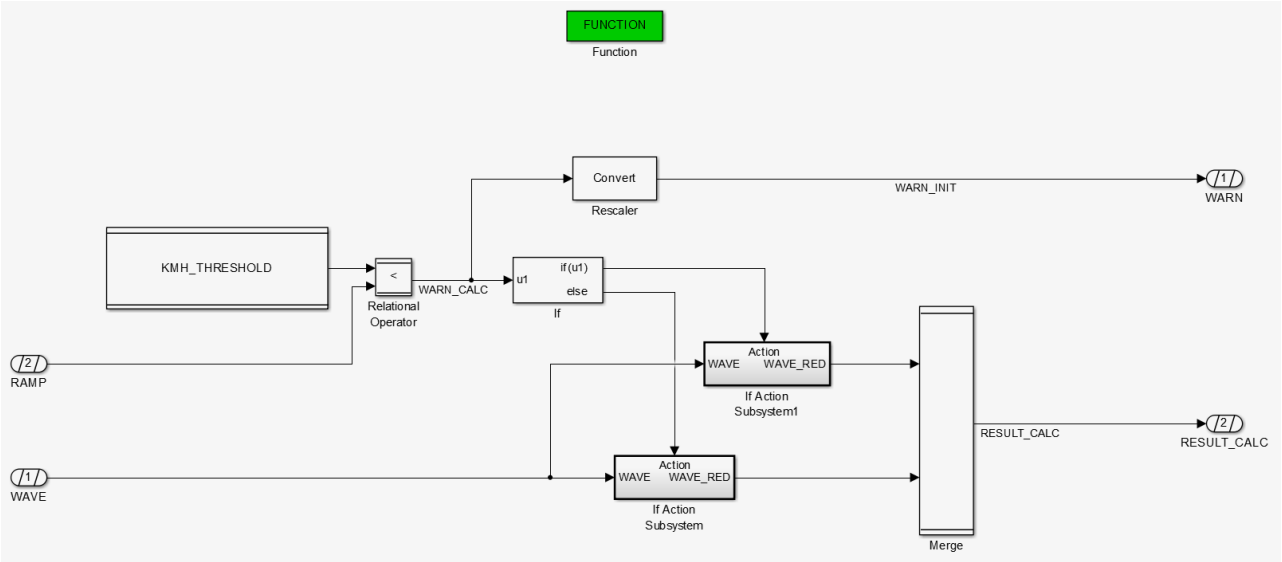


Figura 59 - Logica quarto blocco (condiviso)

QUINTO BLOCCO

Il blocco CALCOLI è composto da due blocchi del modello originale: STRATEGY e FILTER. Sono stati gestiti tramite un blocco Stateflow interno che esegue semplicemente le chiamate di calcolo.

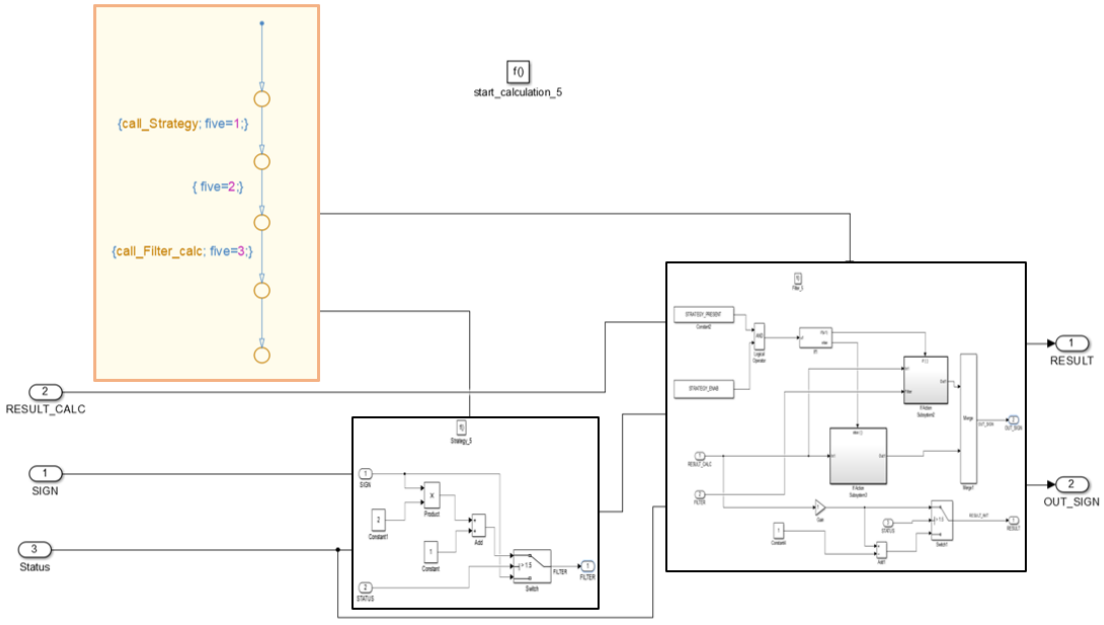


Figura 60 - Logica quinto blocco

UTET – Verifica rimodellazione

ISOFUNZIONALITÀ

Il modello ottenuto diviso per runnable è stato messo a confronto con il modello a stati, sono state eseguite delle verifiche sugli output ottenuti dai modelli per tutti gli scenari. Tutte le variabili di output sono state verificate in modo che il valore ottenuto dalle simulazioni MIL dei due modelli fossero uguali.

Di seguito si riportano per motivi di sintesi i grafici ottenuti dallo scenario 1 per le variabili VALUE, OUT_SIGN, RESULT. Per ogni variabile si mostra la sovrapposizione tra il risultato dei due modelli (primo grafico) e il valore della differenza tra i due valori (secondo grafico).

VALUE

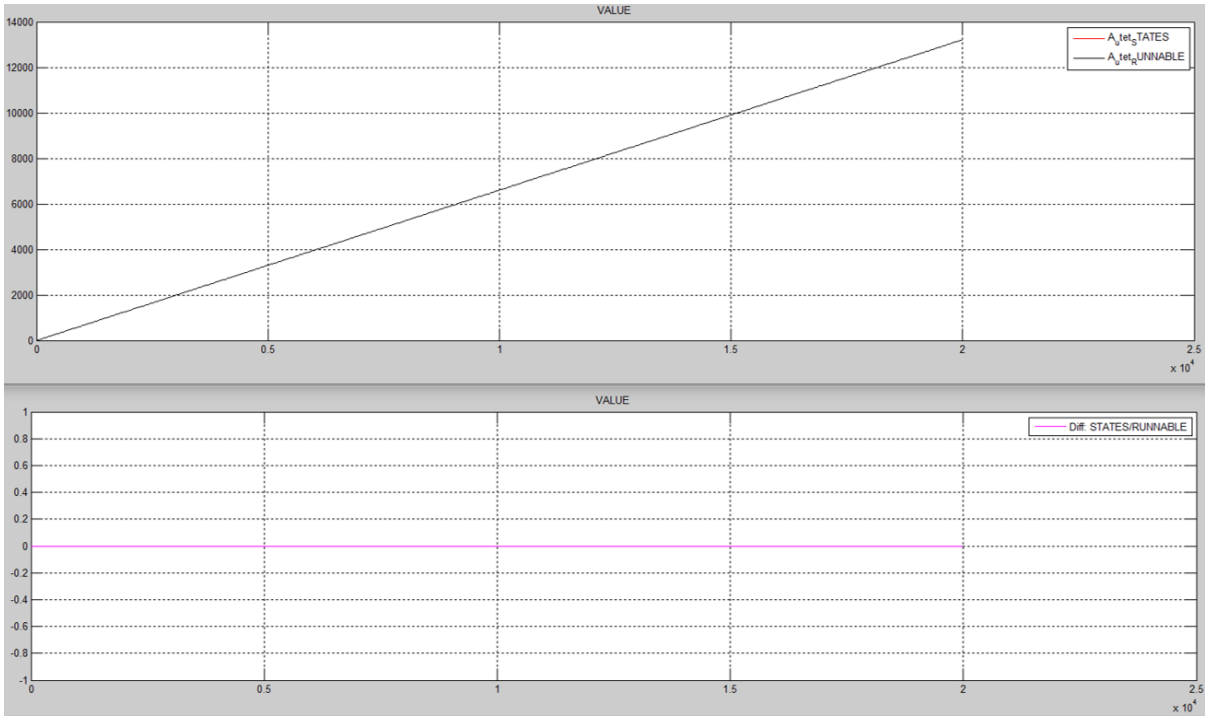


Figura 61 - Confronto VALUE

La sommatoria essendo coerente dimostra che l'aggiornamento della variabile di stato e la sua comunicazione è avvenuta correttamente.

RESULT

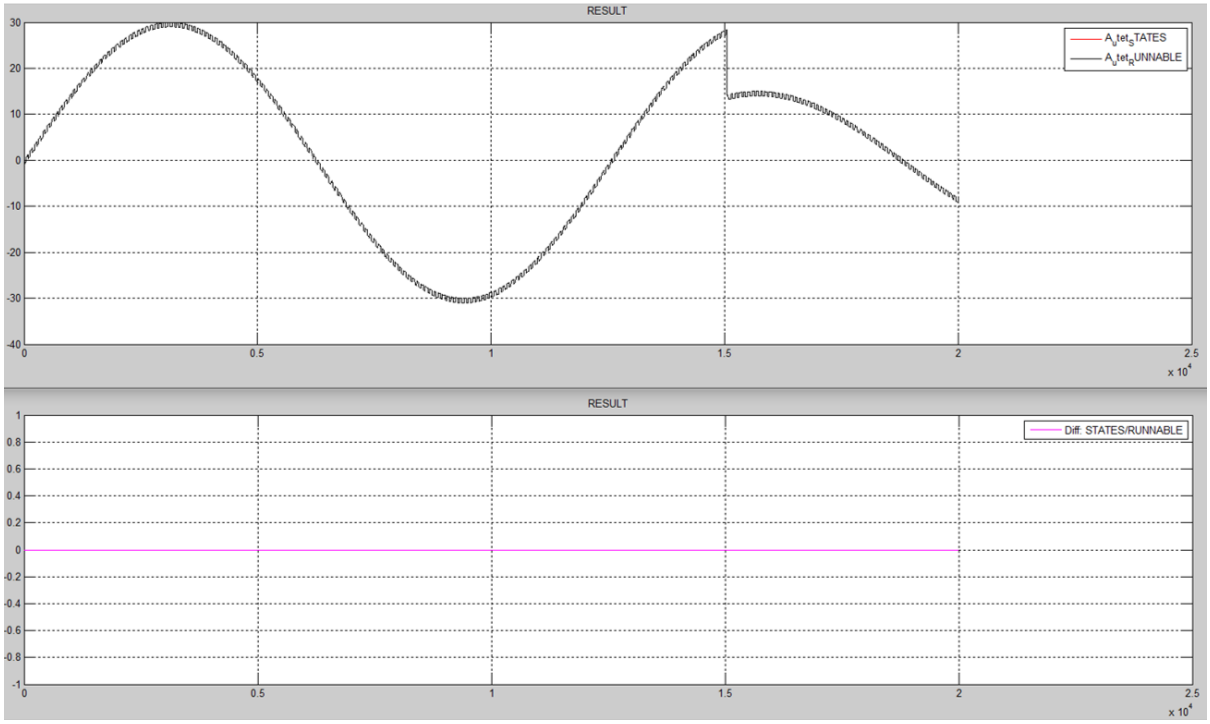


Figura 62 - Confronto RESULT

OUT_SIGN

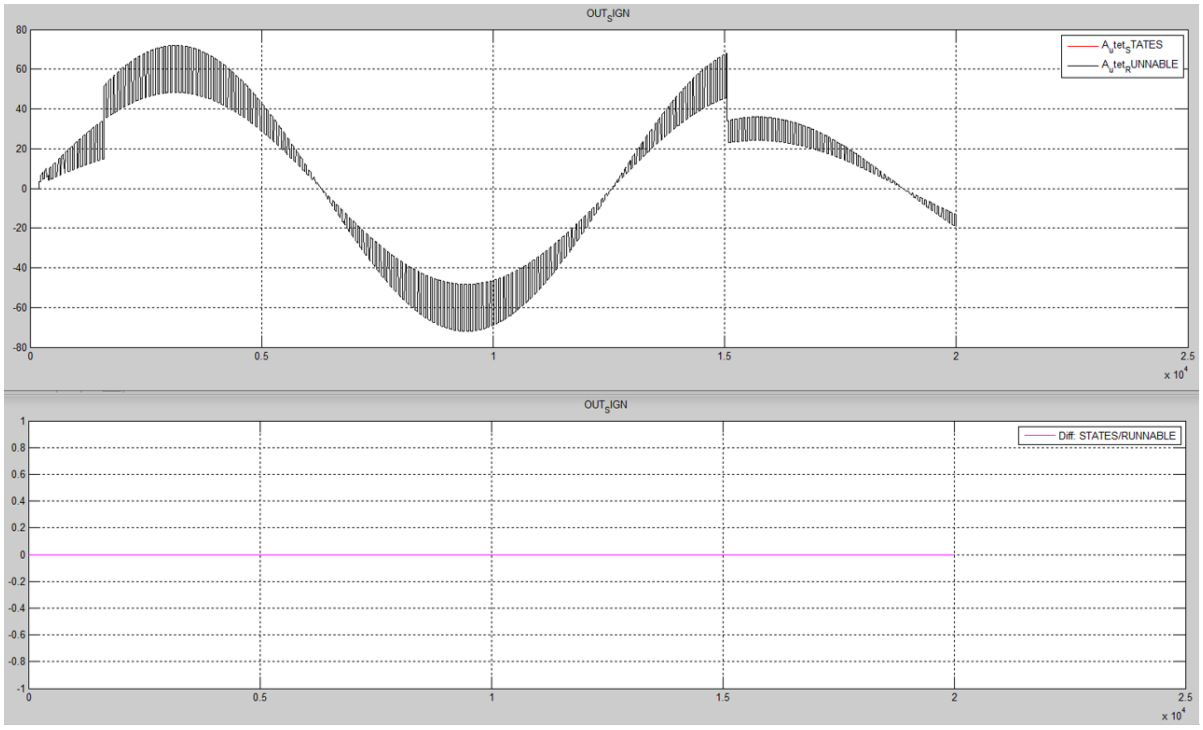


Figura 63 - Confronto OUT_SIGN

Con questo grafico si può notate che tutte le catene di calcolo forniscono risultati coerenti con il modello a stati.

Come si può notare per tutti e tre i casi l'errore è nullo quindi i due modelli si possono definire *isofunzionali*.

COVERAGE

La seconda verifica che è stata eseguita è quella del Coverage. In questa verifica si testa se tutte le logiche di calcolo sono state eseguite almeno una volta per tutte le possibili combinazioni.

In questo caso non è stato fatto un raffronto tra dei risultati di simulazione ma solamente la verifica dell'esecuzione delle logiche.

<u>Risultati UTET_STATES</u>

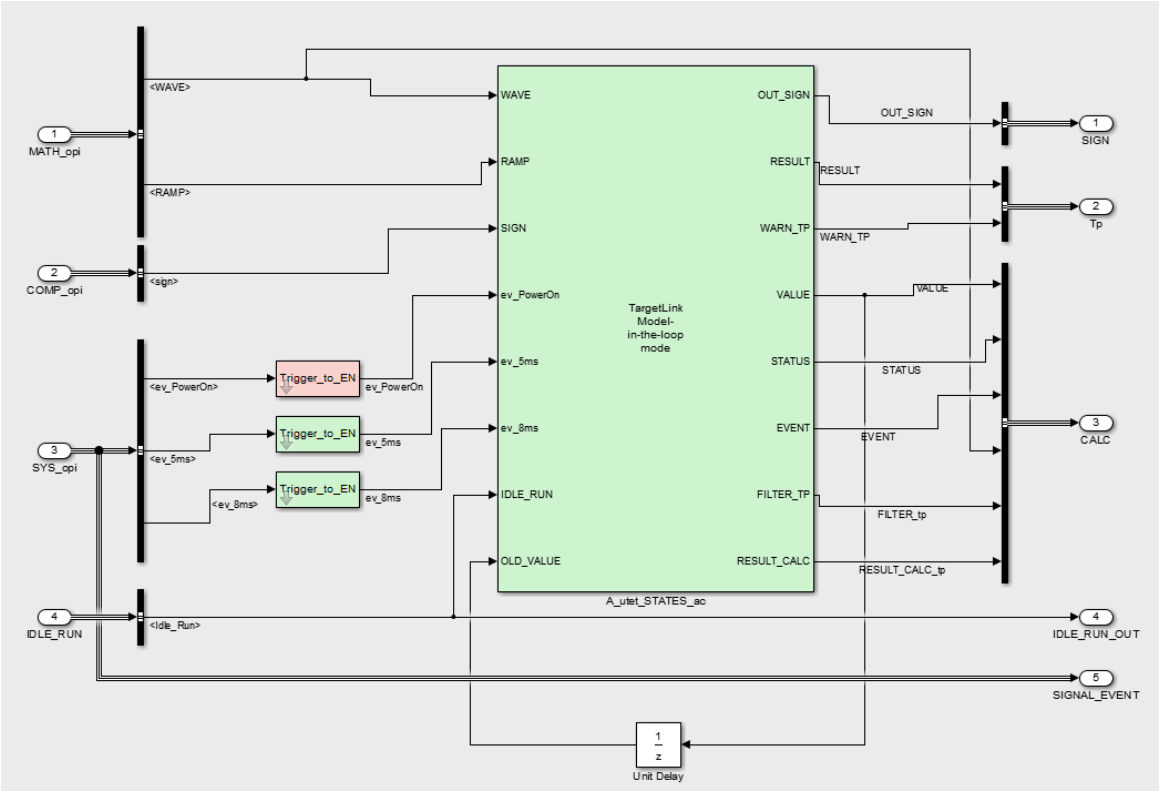


Figura 64 - Risultato grafico Coverage stati

Come spiegato precedentemente, se il modello viene colorato di verde, ha ottenuto la completa verifica delle logiche presenti.

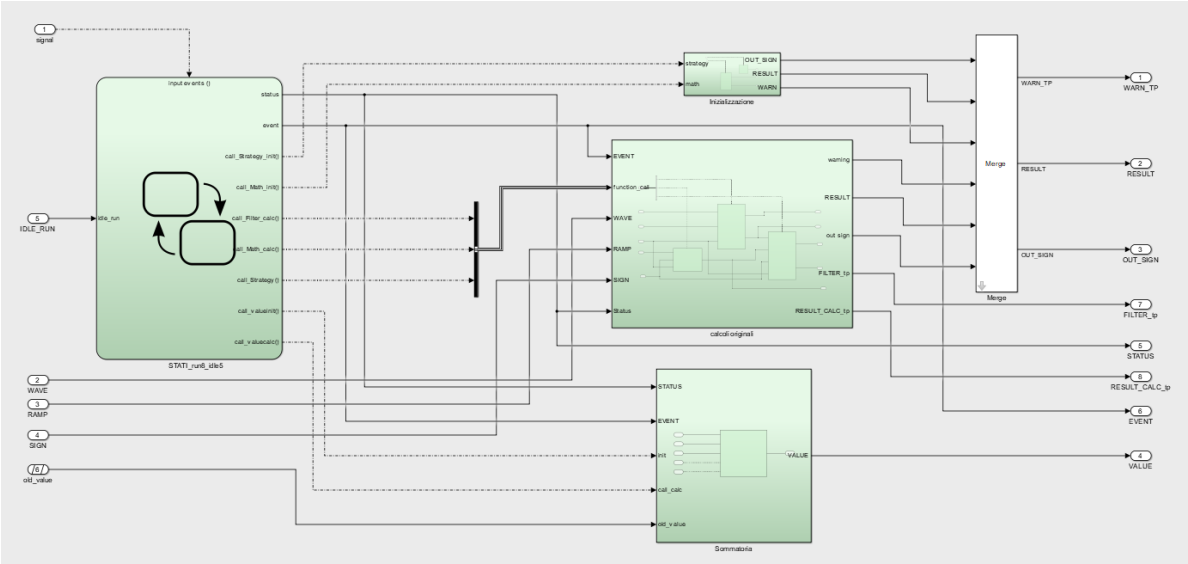


Figura 65 - Dettaglio blocco ac

		D1	C1	MCDC	D1	C1	MCDC	D1	C1	MCDC
1. A_utet_STATES	**117**	95%	53%	7%	0%	0%	0%	98%	65%	43%
2. A_UTET_STATES_opi	59	97%	64%	0%	0%	0%	0%	100%	93%	100%
3. A_utet_STATES_ac	53	96%	50%	0%	0%	0%	0%	100%	100%	100%
4. Subsystem	53	96%	50%	0%	0%	0%	0%	100%	100%	100%
5. A_utet_STATES_ac	**53**	96%	50%	0%	0%	0%	0%	100%	100%	100%
6. En_to_FC	5	100%	NA	NA	0%	NA	NA	100%	NA	NA
7. Function Call Generator	**5**	100%	NA	NA	0%	NA	NA	100%	NA	NA
8. Buffering	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
9. Call_Runnable	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
10. Posting	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
11. En_to_FC1	5	100%	NA	NA	0%	NA	NA	100%	NA	NA
12. Function Call Generator	**5**	100%	NA	NA	0%	NA	NA	100%	NA	NA
13. Buffering	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
14. Call_Runnable	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
15. Posting	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
16. En_to_FC2	5	100%	NA	NA	0%	NA	NA	100%	NA	NA
17. Function Call Generator	**5**	100%	NA	NA	0%	NA	NA	100%	NA	NA
18. Buffering	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
19. Call_Runnable	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
20. Posting	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
21. Stateflow STATI	37	95%	50%	0%	0%	0%	0%	100%	100%	100%
22. STATI_run8_idle5	**14**	96%	NA	NA	0%	NA	NA	100%	NA	NA
23. SF: STATI_run8_idle5	13	96%	NA	NA	0%	NA	NA	100%	NA	NA
24. SF: status_calc	7	93%	NA	NA	0%	NA	NA	100%	NA	NA
25. Inizializzazione	2	100%	NA	NA	0%	NA	NA	100%	NA	NA
26. Filter_Init	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
27. Math_init	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
28. Sommatoria	5	100%	NA	NA	0%	NA	NA	100%	NA	NA
29. SUM	5	100%	NA	NA	0%	NA	NA	100%	NA	NA
30. Calcoli	4	100%	NA	NA	0%	NA	NA	100%	NA	NA
31. Filter_Init	**1**	100%	NA	NA	0%	NA	NA	100%	NA	NA
32. calcoli originali	16	93%	50%	0%	0%	0%	0%	100%	100%	100%
33. Filter	**7**	85%	50%	0%	0%	0%	0%	100%	100%	100%
34. evento 5ms	2	75%	50%	0%	0%	0%	0%	100%	100%	100%
35. evento 8ms	2	75%	50%	0%	0%	0%	0%	100%	100%	100%
36. Math_calc	**5**	100%	NA	NA	0%	NA	NA	100%	NA	NA
37. Strategy_calc	4	100%	NA	NA	0%	NA	NA	100%	NA	NA

Figura 66 - Report coverage dei blocchi interni al blocco ac

Il report finale restituisce la percentuale verificata di ogni singolo blocco. Osservando la riga 3 si ha la valutazione totale del blocco “ac”, mentre dalla riga 4 alla riga 37 il dettaglio dei singoli componenti della logica.

Risultati UTET_RUNNABLE

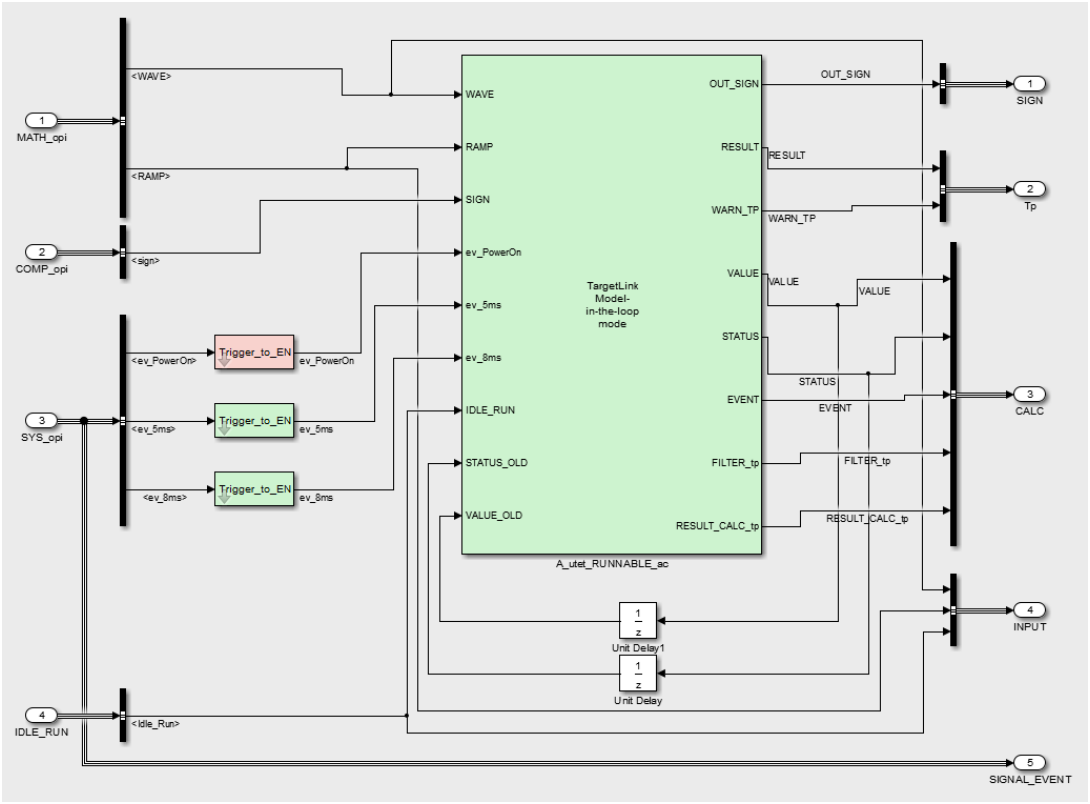


Figura 67 - Risultato grafico Coverage runnable

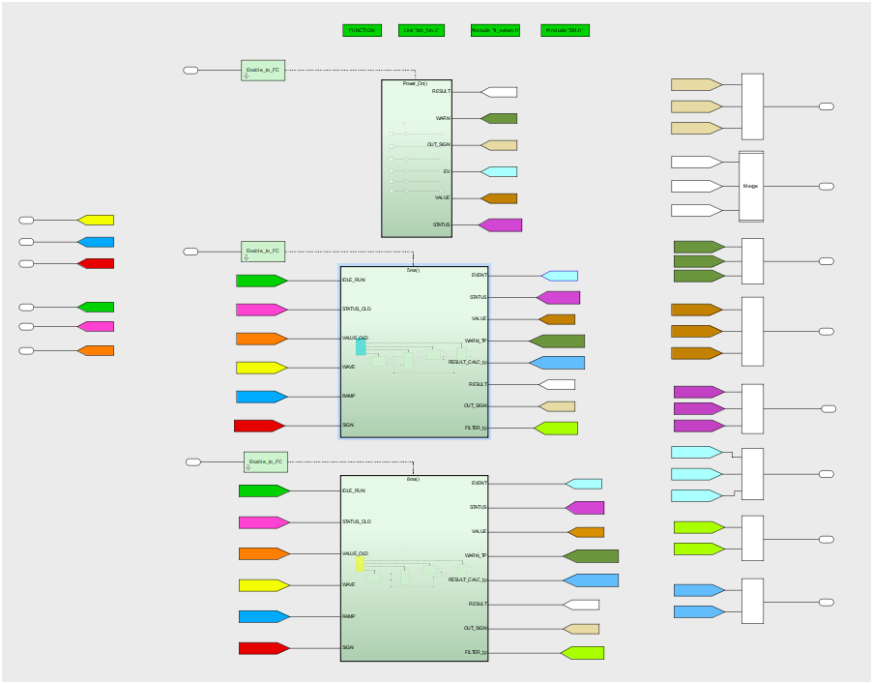


Figura 68 - Dettaglio livello ac

Graficamente avendo tutti i subsystem verdi si può dedurre che il coverage è completo per tutti e tre i parametri (Condition, Decision, MCDC). Nel dettaglio il grafico riassuntivo, le

ultime tre colonne rappresentano lo stato del Coverage del modello, nella riga 3 si ha la panoramica del blocco “ac”, mentre dalla riga 4 alla riga 37 il dettaglio di tutti i subsystem.

		D1		C1		MCDC		D1		C1		MCDC		D1		C1		MCDC	
1.	A_utet_RUNNABLE	**111**	91%		53%		0%		1%		0%		0%		97%		65%		43%	
2.	A_UTET_RUNNABLE_opi	57	97%		64%		0%		0%		0%		0%		100%		93%		100%	
3.	A_utet_RUNNABLE_ac	51	97%		50%		0%		0%		0%		0%		100%		100%		100%	
4.	Subsystem	51	97%		50%		0%		0%		0%		0%		100%		100%		100%	
5.	A_utet_RUNNABLE_ac	**51**	97%		50%		0%		0%		0%		0%		100%		100%		100%	
6.	5ms	**17**	95%		50%		0%		0%		0%		0%		100%		100%		100%	
7.	math	**2**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
8.	Subsystem	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
9.	status_eval_5ms	**3**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
10.	strategy_e_filter_5	7	89%		50%		0%		0%		0%		0%		100%		100%		100%	
11.	Filter	**3**	80%		50%		0%		0%		0%		0%		100%		100%		100%	
12.	Strategy	**2**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
13.	sum_5	**2**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
14.	8ms	**17**	95%		50%		0%		0%		0%		0%		100%		100%		100%	
15.	math	**2**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
16.	Subsystem	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
17.	status_eval_8ms	**3**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
18.	strategy_e_filter_8	7	89%		50%		0%		0%		0%		0%		100%		100%		100%	
19.	Filter	**3**	80%		50%		0%		0%		0%		0%		100%		100%		100%	
20.	Strategy	**2**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
21.	sum_8	**2**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
22.	En_to_FC	5	100%		NA		NA		0%		NA		NA		100%		NA		NA	
23.	Function Call Generator	**5**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
24.	Buffering	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
25.	Call_Runnable	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
26.	Posting	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
27.	En_to_FC1	5	100%		NA		NA		0%		NA		NA		100%		NA		NA	
28.	Function Call Generator	**5**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
29.	Buffering	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
30.	Call_Runnable	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
31.	Posting	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
32.	En_to_FC2	5	100%		NA		NA		0%		NA		NA		100%		NA		NA	
33.	Function Call Generator	**5**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
34.	Buffering	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
35.	Call_Runnable	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
36.	Posting	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	
37.	Init	**1**	100%		NA		NA		0%		NA		NA		100%		NA		NA	

Figura 69 - Resoconto blocchi interni a livello ac

TARGHETTATURA

Il passaggio successivo per ottenere un codice C funzionante con variabili ad aritmetica a punto fisso (“fixed point”) per poter sostenere le simulazioni SIL è quello di “targhettare” tutte le variabili nel modello con le caratteristiche adeguate. Il modello UTET non aveva una targhettatura completa per poter compilare correttamente con il tool Target Link. È stato quindi deciso di definire la targhettatura di tutte le nuove variabili inserite nella modellazione focalizzandosi sul definire le caratteristiche di ogni singola variabile in funzione del range e della precisione desiderata.

La descrizione ottenuta per tutte le variabili è riassumibile nella seguente tabella.

VARIABILE	CARATT.	min	Max	RANGE	RANGE_bit	precisione	PREC_bit	LSB	TYPE
IDLE_RUN	INPUT	0	2	2	2	1	0	2^0	UINT8
STATUS (old)	INPUT	0	2	2	2	1	0	2^0	UINT8
RAMP	INPUT	0	200	200	8	1	0	2^0	UINT16
WAVE	INPUT	-30	30	60	6	0,5	2	2^-2	INT16
SIGN	INPUT	0	5	5	3	1	0	2^0	UINT8
VALUE_OLD	INPUT	0	15000	15000	13	1	0	2^0	UINT16
STATUS	TEST_PIONT	0	2	2	2	1	0	2^0	UINT8
RESULT	TEST_POINT	-35	30	60	6	0,01	4	2^-4	INT16
FILTER	TEST_POINT	0	11	11	4	1	0	2^0	UINT8
EVENT	TEST_POINT	0	8	8	4	1	0	2^0	UINT8
VALUE	OUTPUT	0	15000	1500	13	1	0	2^0	UINT16
WARN	OUTPUT	0	1						BOOL
OUT_SIGN	OUTPUT	-80	80	160	8	0,01	4	2^-4	INT16

Questa scelta non è stata vincente, infatti, durante i calcoli si presentavano dei troncamenti dovuti a ridefinizione del tipo o calcoli con precisione troppo basse (questi problemi sono tipici delle fasi preliminari di codifica fixed step). In questo modo si otteneva una propagazione di errore che forniva un risultato tra MIL e SIL totalmente differente.

L’esempio più eloquente è rappresentato nella differenza tra gli output del segnale OUT_SIGN.

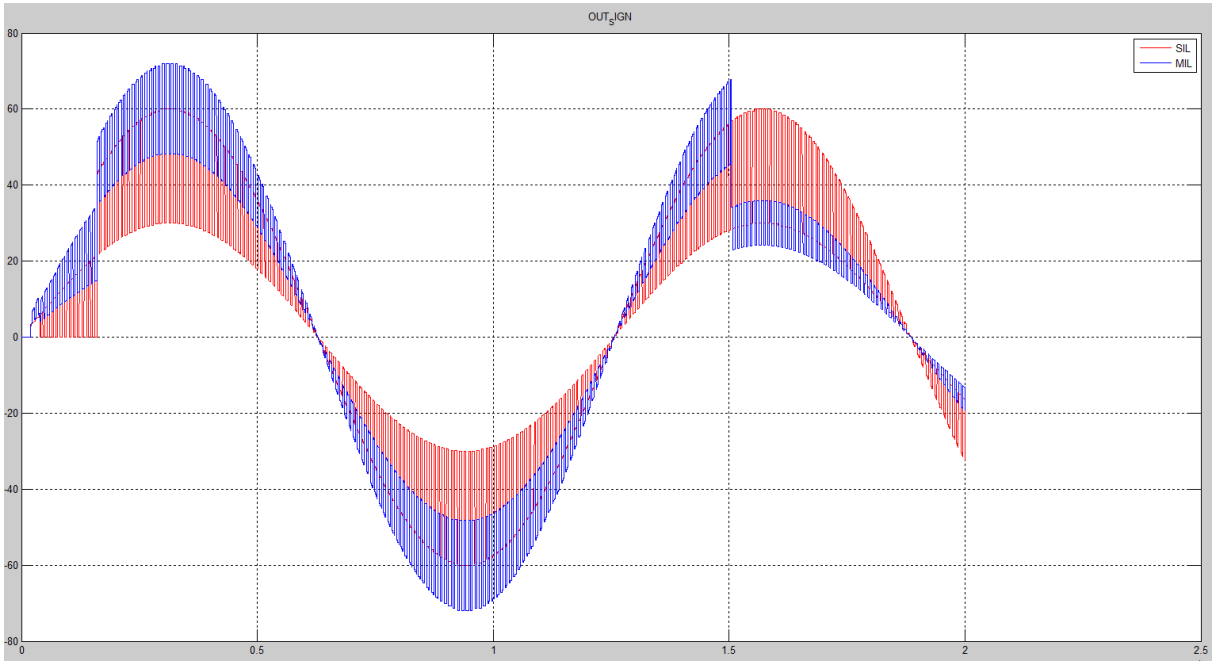


Figura 70 - Confronto MIL-SIL segnale OUT_SIGN

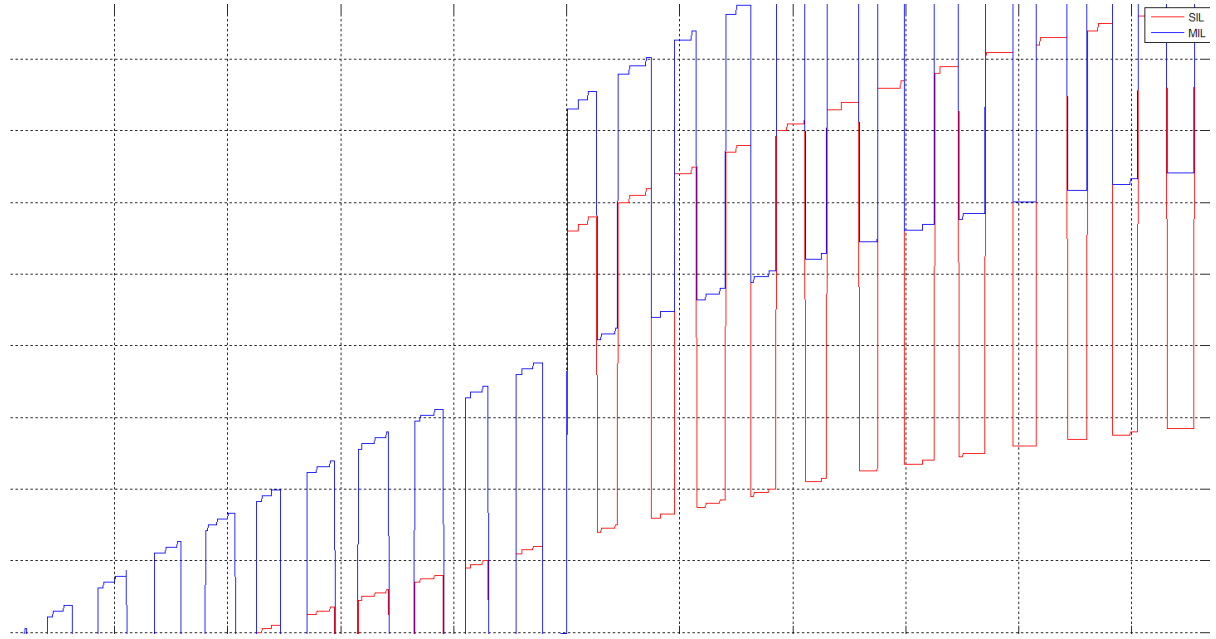


Figura 71 - Dettaglio del grafico di confronto

Dall’analisi dei calcoli e delle targhettature applicate sono stati individuati i calcoli che comportavano i troncamenti indesiderati, in questo modo si è passati alla ridefinizione delle caratteristiche di alcune variabili aumentando il numero di bit dedicati alla definizione della variabile.

VARIABILE	IDLE_RUN	RAMP	WAVE	SIGN	STATUS	RESULT	EVENT	FILTER	OUT_SIGN	WARN	VALUE
TIPO_iniz	UINT_8	UINT_16	INT16	UINT8	UINT8	INT16	UINT8	UINT8	INT16	BOOL	UINT16
TIPO	\\	INT16	INT16	INT16	\\	\\	\\	INT16	\\	\\	\\
LSB_iniz	2^0	2^0	2^-2	2^0	2^0	2^-4	2^0	2^0	2^-4		2^0
LSB	\\	2^-4	2^-4	2^-4	\\	\\	\\	2^-4	\\		\\

Nonostante vi fosse la possibilità internamente a TargetLink di eseguire la verifica tra MIL e SIL è stato usato il tool “auto_sil” in quanto è in grado considerare una tolleranza tra i risultati ottenuti e fornire un metro di accettabilità in automatico.

AUTO_SIL

Come nel caso del Coverage non è stata eseguita una verifica diretta tra i due modelli ma un raffronto tra le verifiche dei due processi di verifica tra Runnable e Stati.

UTETS

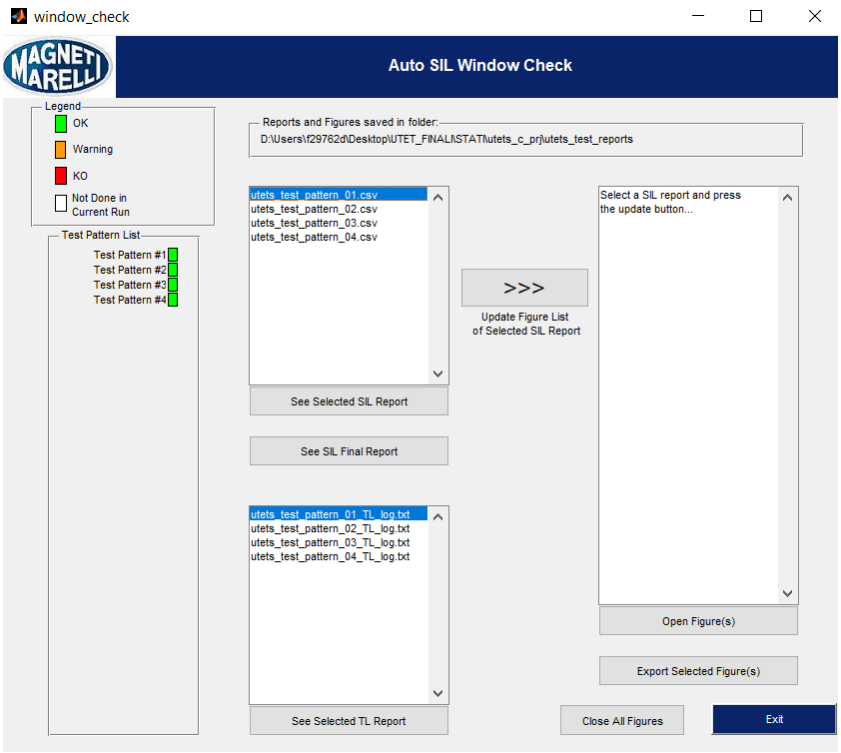


Figura 72 - Finestra resoconto auto sil modello stati

Dalla finestra di dialogo si nota come per tutti i Test Pattern i risultati ottenuti siano congruenti tra simulazioni MIL e SIL o, in caso di scostamento tra i segnali, la differenza sia minore della tolleranza imposta.

Nei grafici seguenti si ha la comparazione tra MIL e SIL nel grafico superiore e nel grafico inferiore l'errore tra le due simulazioni, è presente inoltre il segno in verde e rosso della tolleranza superiore e inferiore che rende accettabile lo scostamento tra le due simulazioni.

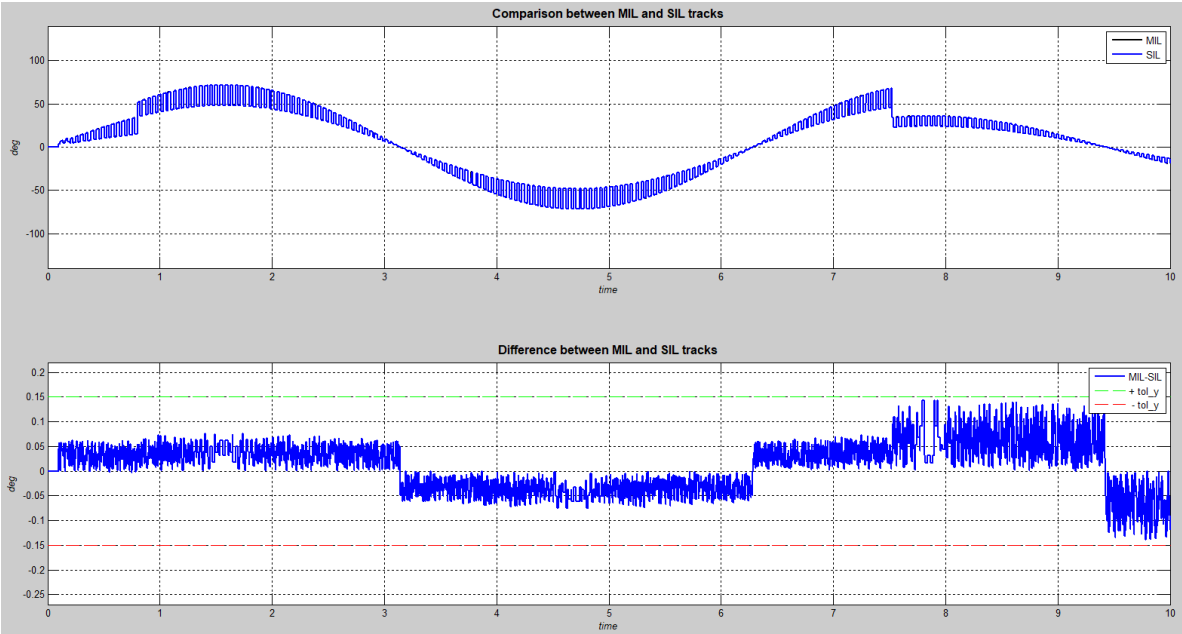


Figura 73 - Risultato auto sil segnale OUT_SIGN

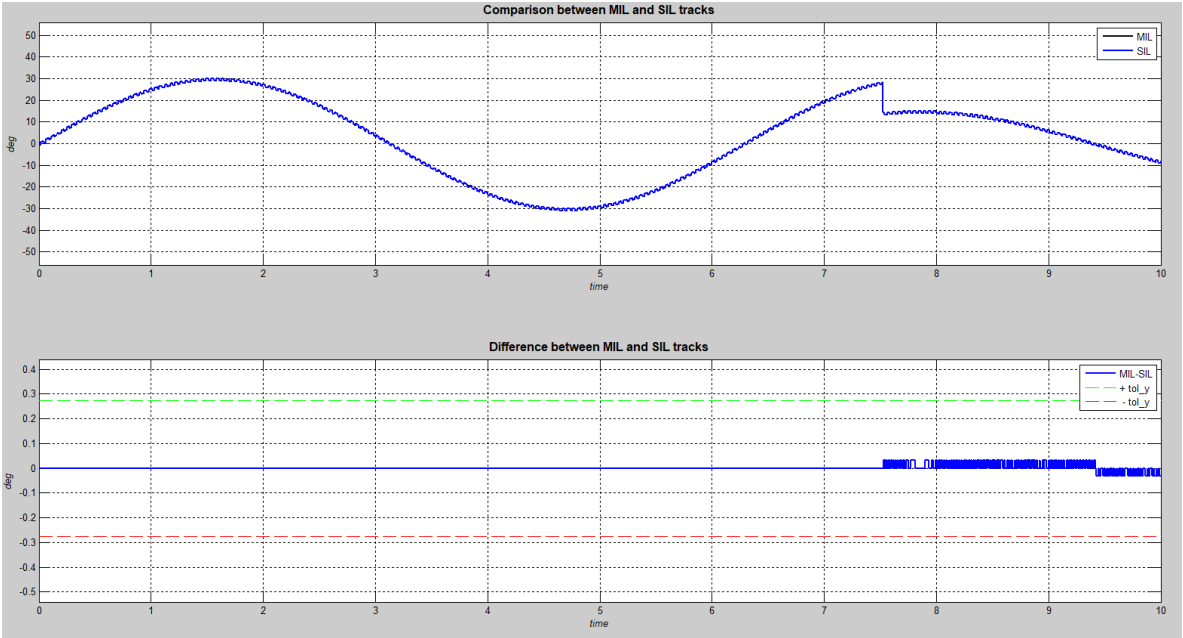


Figura 74 - Risultato auto sil segnale RESULT

Questi scostamenti sono presenti a causa di troncamenti o variazioni di Tipo della variabile durante i calcoli nel SIL.

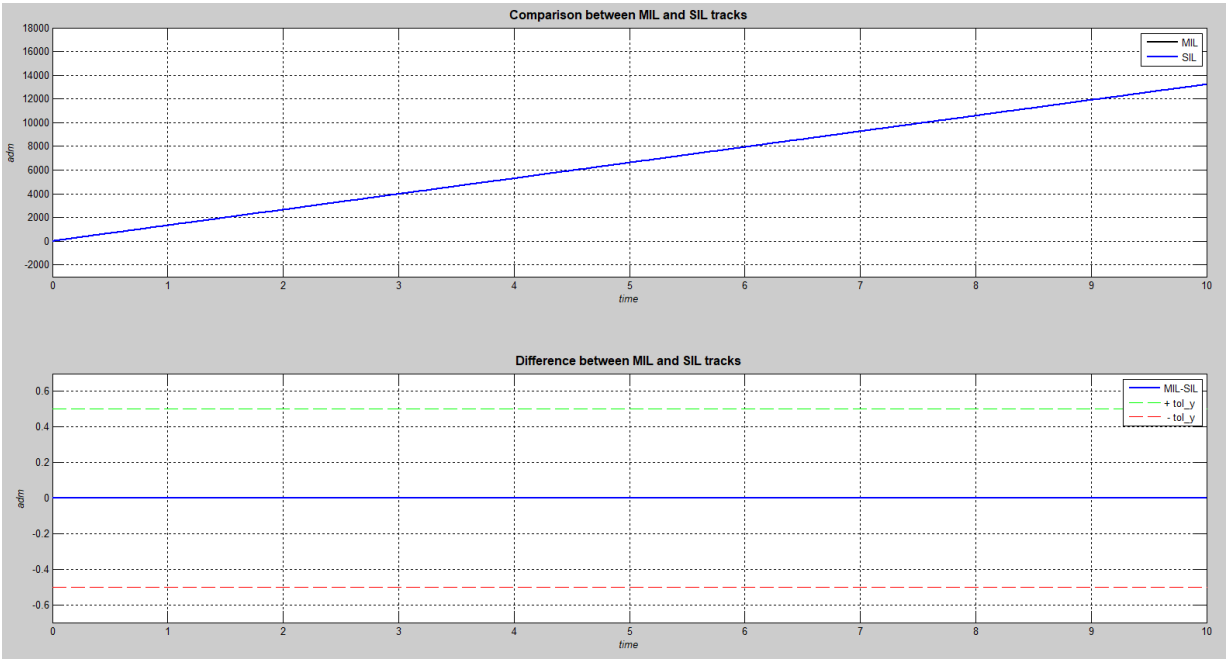


Figura 75 - Risultato auto sil VALUE

Per quanto riguarda invece il risultato di VALUE, essendo una variabile che non presenta particolari calcoli non ho differenza nel risultato tra MIL e SIL.

UTETR

Come per UTETS , anche i risultati di UTETR forniscono dei risultati coerenti tra simulazioni MIL e SIL.

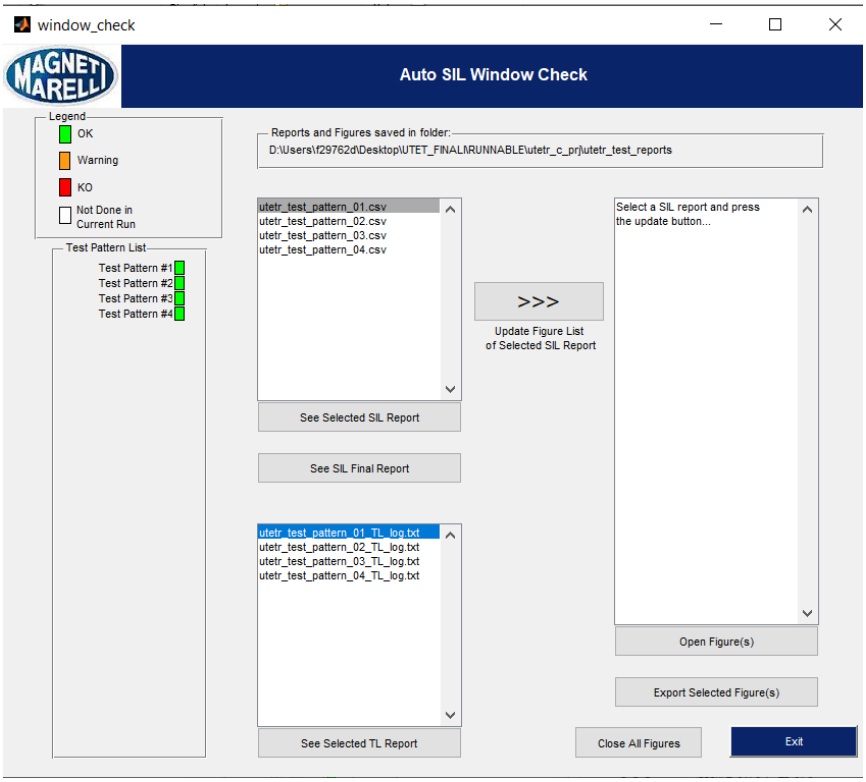


Figura 76 - Finestra resoconto auto sil modello runnable

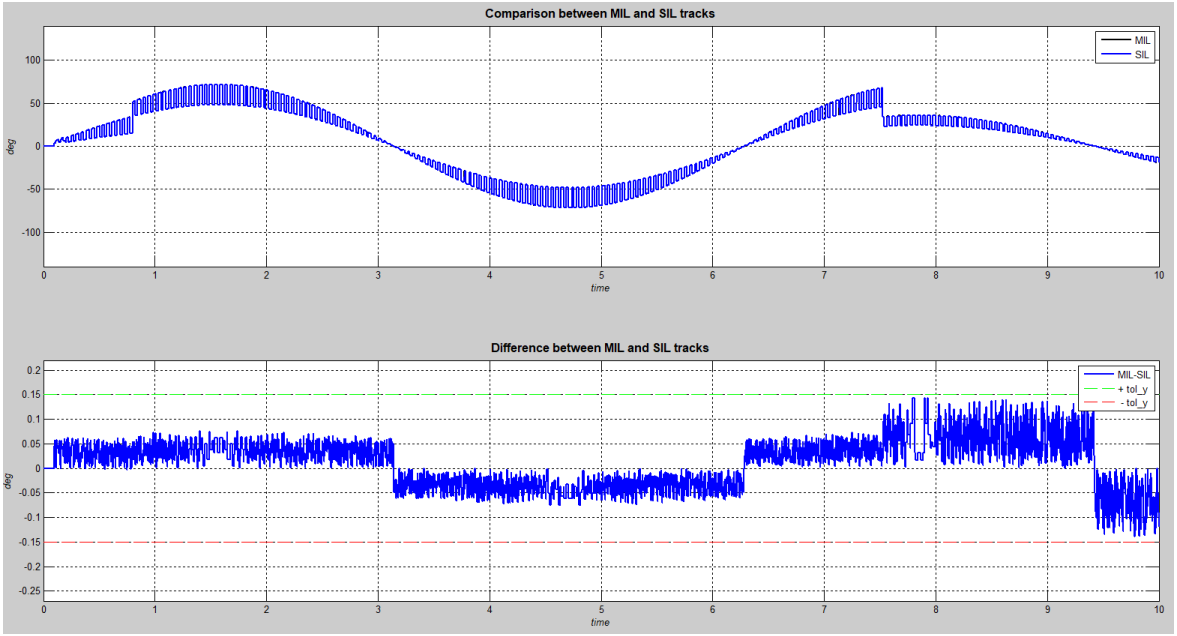


Figura 77 - Risultato auto sil OUT_SIGN

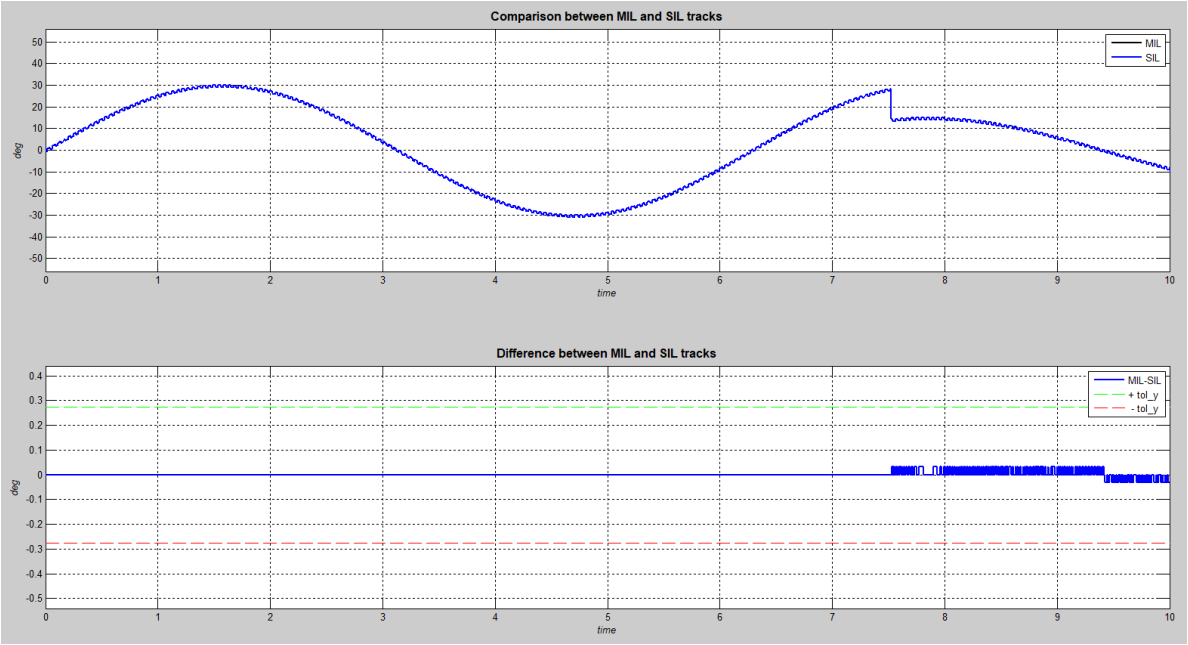


Figura 78 - Risultato auto sil RESULT

Come per il caso del modello a Stati i risultati hanno queste divergenze causate da troncamenti presenti ne calcoli.

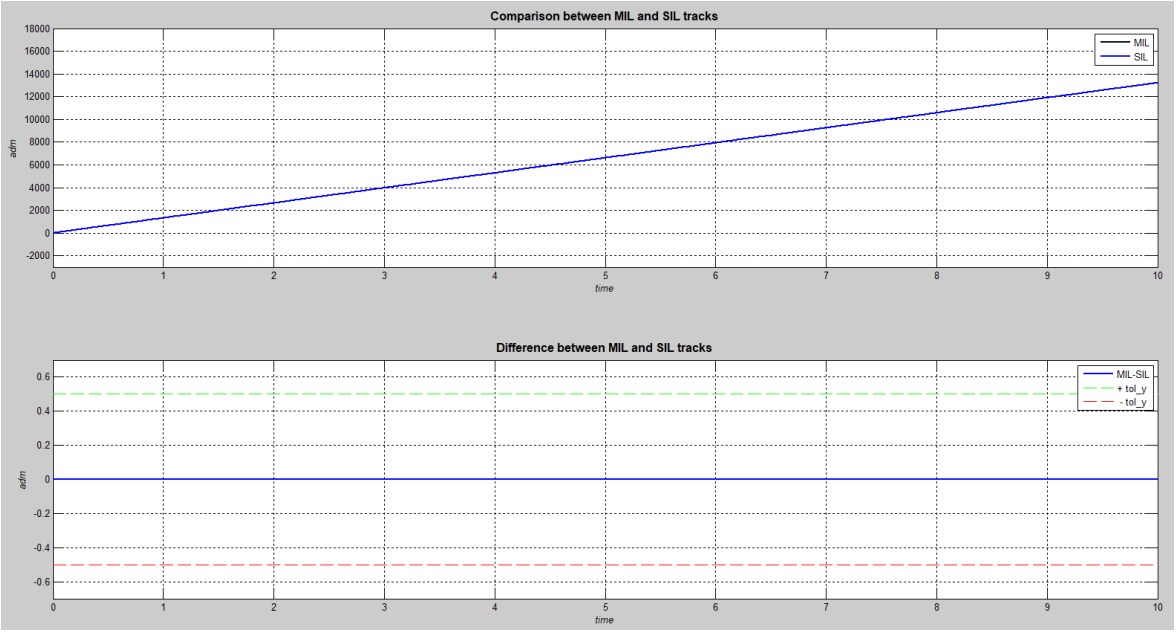


Figura 79 - Risultato auto sil VALUE

VERIFICA SIMULAZIONI SIL-SIL

L'ultima verifica prima di passare alle analisi delle metriche è quella di ottenere dei risultati simili tra gli output delle simulazioni SIL tra i due modelli.

In generale, avendo simulazioni MIL coerenti e verifica dell'auto_sil con tutti gli output dentro alla tolleranza desiderata è è possibile affermare che i due SIL producano risultati uguali o comunque con differenze accettabili (in quanto sotto tolleranza).

UTET – ANALISI DEI RISULTATI

Dopo aver appurato che le modifiche di entrambi i modelli abbiano fornito un modello solido e isofunzionale si è passati alla verifica delle metriche al fine di ottenere una valutazione sull'efficienza della rimodellazione.

Le metriche scelte per il confronto sono:

- Lunghezza del codice in righe
- Tempo di simulazione MIL
- Tempo di compilazione
- Tempo di simulazione SIL
- Tempo esecuzione verifica "auto_sil"

LUNGHEZZA DEL CODICE

La lunghezza del codice è la metrica più semplice da verificare. Sono stati analizzati i file "model.c" sia per le versioni a stati che runnable.

	Righe codice	
STATES	1524	
RUNNABLE	825	699

Con la forma a RUNNABLE si è potuto ottenere un codice più snello con un risparmio di 699 righe.

TEMPI

La definizione delle tempistiche è stata eseguita con l'utilizzo di uno script dedicato esposto precedentemente. Si ricordano parametri di accettabilità dei tempi rilevati:

Per ogni Test Pattern si hanno a disposizione 15 tempi rilevati per ogni aspetto (MIL, SIL, generazione), ho pertanto una matrice 3x15 per ogni Test Pattern. Si esegue la differenza tra una grandezza e tutte le altre della stessa simulazione. Si valutano le differenze relative, se

questa differenza è inferiore alla tolleranza massima ammessa (imposta al 5% del tempo minimo rilevato per la simulazione in analisi) per una numerabilità superiore al 60% del numero totale di simulazioni allora la tempistica analizzata viene ammessa nel calcolo della media.

Di seguito la tabella riassuntiva dei tempi medi necessari per le simulazioni **MIL**, **SIL** e **COMPILAZIONE** di entrambe le configurazioni (States e Runnable) per ogni Test Pattern. Affianco ai tempi relativi al modello Runnable sono riportate le differenze tra i tempi states e runnable (positivo, negativo).

		TP_1		TP_2		TP_3		TP_4	
STATES	MIL	3,06		3,01		2,97		2,98	
	SIL	2,29		2,32		2,29		2,29	
	COMP.	12,82		12,76		12,76		12,87	
RUNNABLE	MIL	3,17	0,11	3,11	0,10	3,10	0,13	3,15	0,17
	SIL	2,37	0,08	2,34	0,02	2,39	0,10	2,34	0,05
	COMP.	12,39	0,43	12,47	0,29	12,50	0,26	12,33	0,54

L'ultima tempistica considerata è quella necessaria al test **auto_sil**.

	Tempo [sec]	
STATES	172,95	
RUNNABLE	161,18	11,77

RESOCONTO RISULTATI OTTENUTI DAL MODELLO PALESTRA

La modellazione per Runnable è fattibile a patto di gestire accuratamente alcuni aspetti (blocchi condivisi con function e ricircolo di variabili), fornisce un modello isofunzionale dove le tempistiche di simulazione rimangono pressochè invariate con il vantaggio di ottenere un codice più snello.

Alla luce di questi risultati si è passati alla rimodellazione di CPUMPM.

CPUMPM

Il modello CPUMPM è il modello dove è stata concentrata la maggior parte dell'attenzione dell'attività di tesi in quanto è un modello realmente in uso. Tutte le informazioni ottenute dall'esperienza sul modello UTET sono servite a semplificare il processo di reingegnerizzazione.

CPUMPM - STRUTTURA DEL MODELLO ORIGINALE

Di seguito si esegue l'analisi del modello originale nel dettaglio come per il modello UTET.

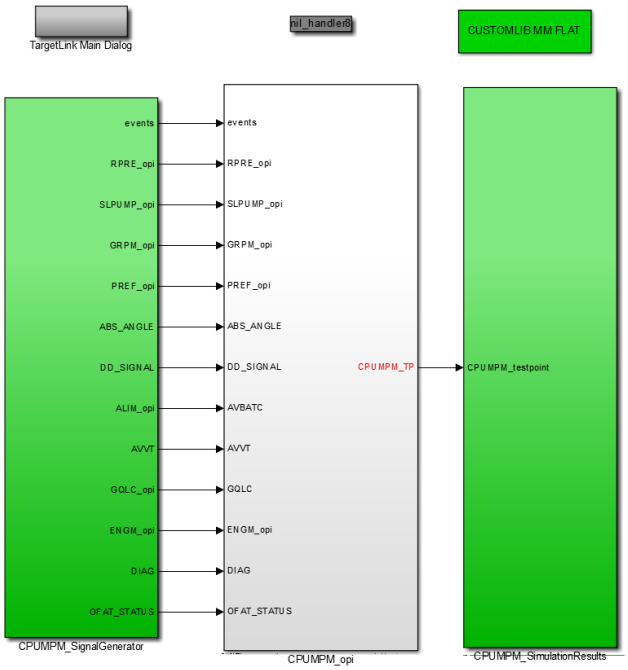


Figura 80 – CPUMPM - Livello superiore

Come con UTET il modello presenta tre subsystem principali: generazione input, esecuzione calcoli, generazione output.

CPUMPM_SIGNALGENERATOR

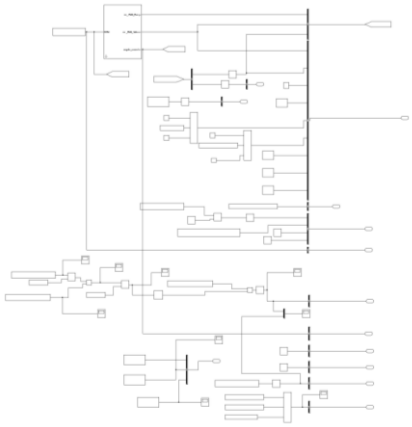


Figura 81 - CPUMPM_SignalGenerator

Il blocco di generazione dei segnali è diviso in 2 parti, la zona superiore è dedicata alla generazione di tutti gli eventi chiamanti del modello, ognuno caratterizzato in maniera da poter rappresentare al meglio un caso reale nella presenza di successione degli eventi (Onde quadre, scalini e calcolo di soglia per il trigger) .

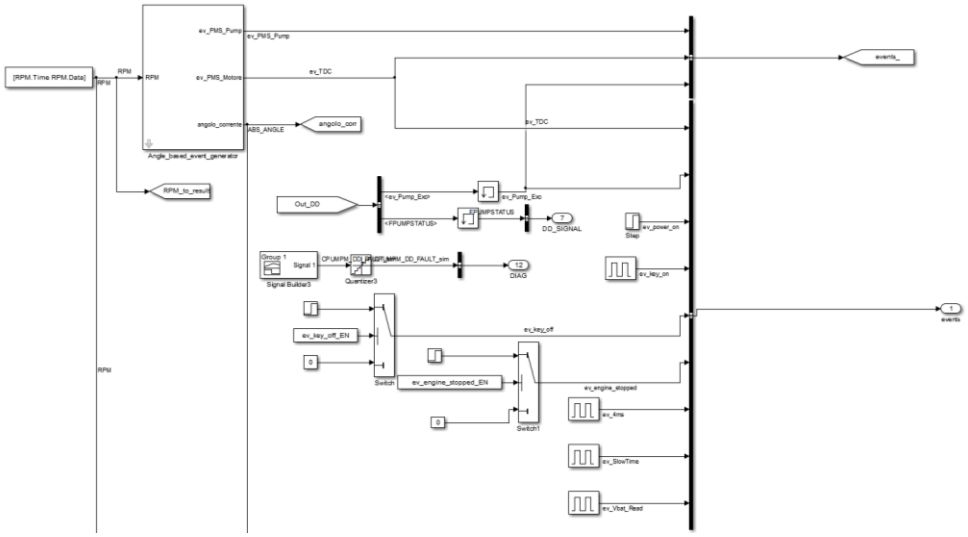


Figura 82 - dettaglio CPUMPM_SignalGenerator – Eventi

Gli eventi presenti sono:

- PowerOn: accensione del sistema
- TDC: quando un pistone arriva al punto morto superiore
- PumEx: quanto viene eseguita una pompata
- 4ms: evento temporale che si ripete ogni 4 millisecondi
- SlowTime: evento temporale che si ripete ogni 100 millisecondi
- VbatRead: evento temporale che si esegue ogni 12 millisecondi

- KeyOn: rappresenta l'evento di accensione tramite il "giro" della chiave
- keyOff: rappresenta quando si "stacca" la chiave
- EngineStopped: rappresenta quando il motore si stoppa

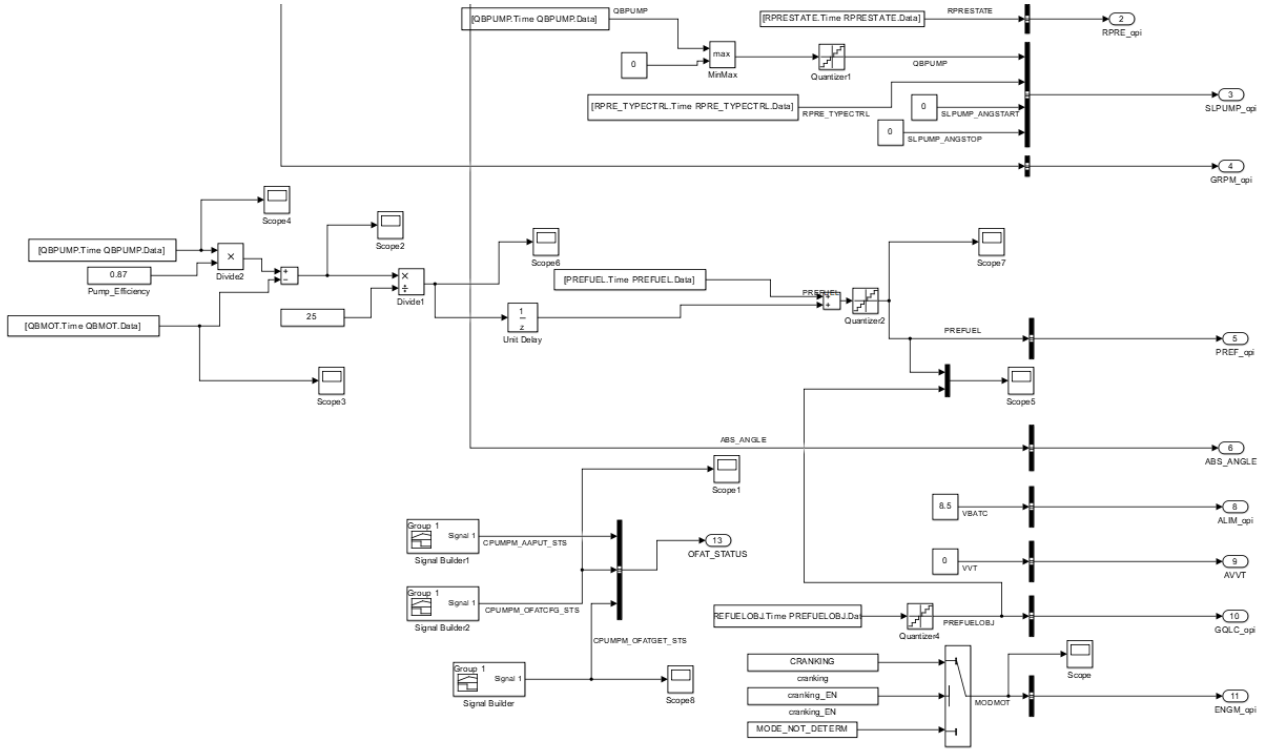


Figura 83 - Dettaglio CPUMPM_SignelGenerator - Altri segnali

La seconda parte è dedicata alla lettura e definizione di parametri utili alla simulazione (esempio, costanti o parametri e definizione di variabili con variazione non schematica nel tempo). In generale, la definizione degli input è molto simile nel concetto e nella struttura al modello UTET.

L'unica variabile di input che viene influenzata dai risultati del modello è l'evento PumpEx. Questo segnale viene generato nel subsystem SimulationResult tramite un blocco che simula il Device Driver (blocco DD a sinistra).

CPUMPM_SIMULATIONRESULT

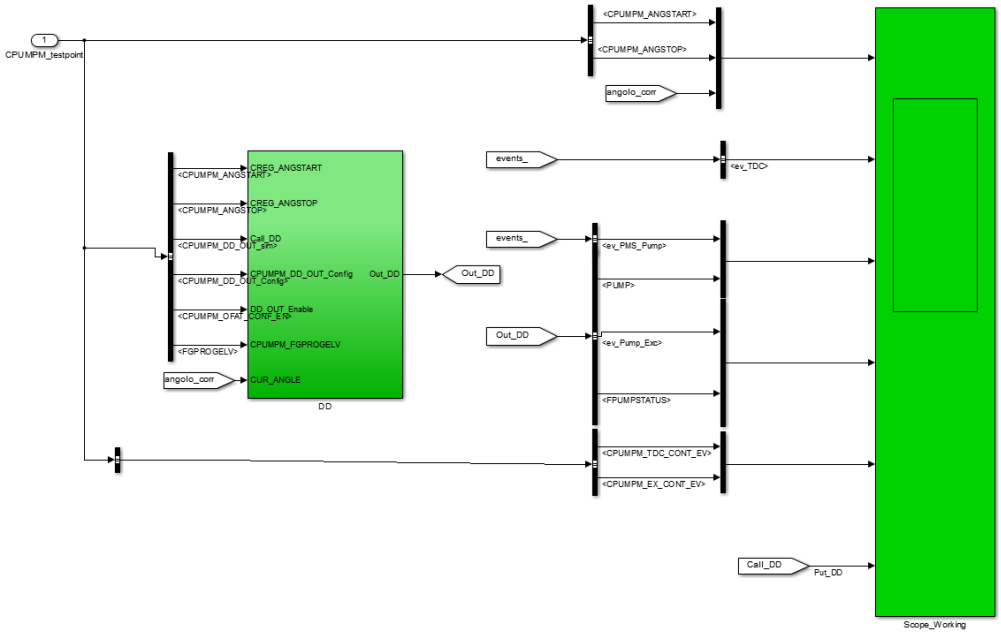


Figura 84 - CPUMPM_SimulationResult

All'interno del blocco che simula il Device Driver sono presenti delle logiche che simulano parti esterne al modello ma che hanno interazioni che condizionano i risultati del modello in base ai loro output.

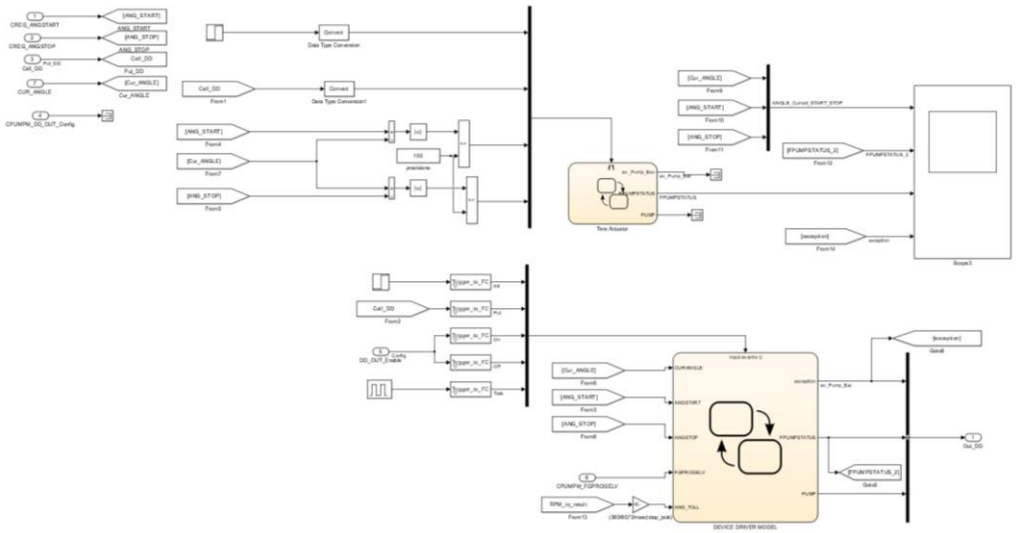


Figura 85 - Blocco DD

Il blocco che definisce l'input del PumpEx è lo stateflow in basso. Lo Stateflow è composto da una macchina a stati che aggiorna il suo stato ad ogni simstep tramite l'evento "Task", dopo aver abilitato la possibilità di eseguire pompate (grazie al comando "ConfigOn" presente nelle logiche di inizializzazione) si entra nel primo stato NOT PROGRAMMED, la macchina non

cambia il suo stato finché non arriva un segnale di “Put” proveniente dalle logiche di calcolo legate alla definizione o programmazione dell’evento. A questo punto si entra nello stato PROGRAMMED. Per poter entrare nello stato di PROGRESS si deve aspettare che la differenza tra l’angolo motore (calcolato nel blocco input) e l’angolo di inizio esecuzione raggiunga un valore inferiore del valore di tolleranza imposto. Si uscirà da questo stato solo quando la differenza tra l’angolo dell’albero motore e l’angolo di fine esecuzione sarà di un valore inferiore al valore di tolleranza predefinito. Usciti da questo stato si possono raggiungere due stati NOT PROGRAMMED o WAIT in base allo stato della variabile FGPROGELV che rappresenta il minimo tempo disponibile per programmare un nuovo evento (1 possibile, 0 no). Se entro in WAIT, si attende un evento di “Put” per poter entrare nello stato di PROGRAMMED altrimenti, qualora l’angolo motore sia troppo vicino all’angolo di start precedentemente definito si va in NOT PROGRAMMED.

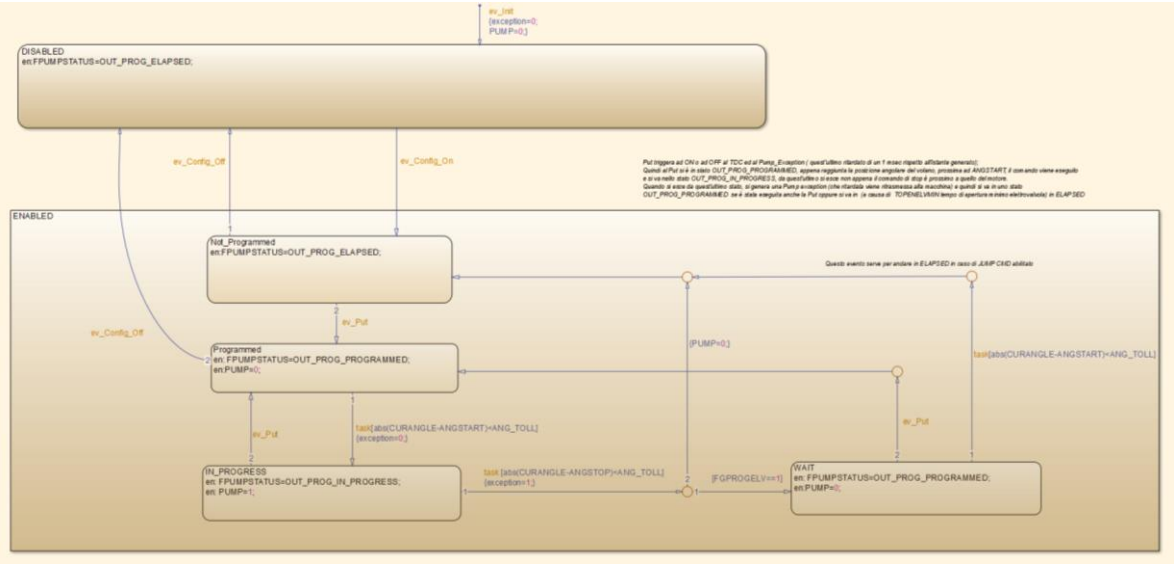


Figura 86 - Macchina a stati Device Driver Model

GRAFICI OUTPUT

Il grafico riassuntivo di output generato dal blocco Scope verde fornisce il grafico di vari aspetti importanti per la supervisione di un possibile corretto funzionamento del modello.

Vi sono 6 grafici che raffigurano:

1. Angolo motore (CURRANGLE), Angolo inizio esecuzione (CPUMPM_ANGSTART), Angolo fine esecuzione (CPUMPM_ANGSTOP).
2. Evento TDC, il grafico mostra il valore 1 quando è presente un TDC (ev_TDC)
3. Evento PMS pompa (ev_PMS_Pump) ed esecuzione pompata (PUMP)
4. Esecuzione pompata (ev_Pum_Ex) e stato del comando (FPUMPSTATUS)

5. Conteggio degli eventi TDC (CPUMPM_TDC_CONT_EV) e degli eventi di Pompata (CPUMPM_EX_CON_EV)
6. Evento “Put” in ingresso nel DeviceDiverModel (PUT_DD)

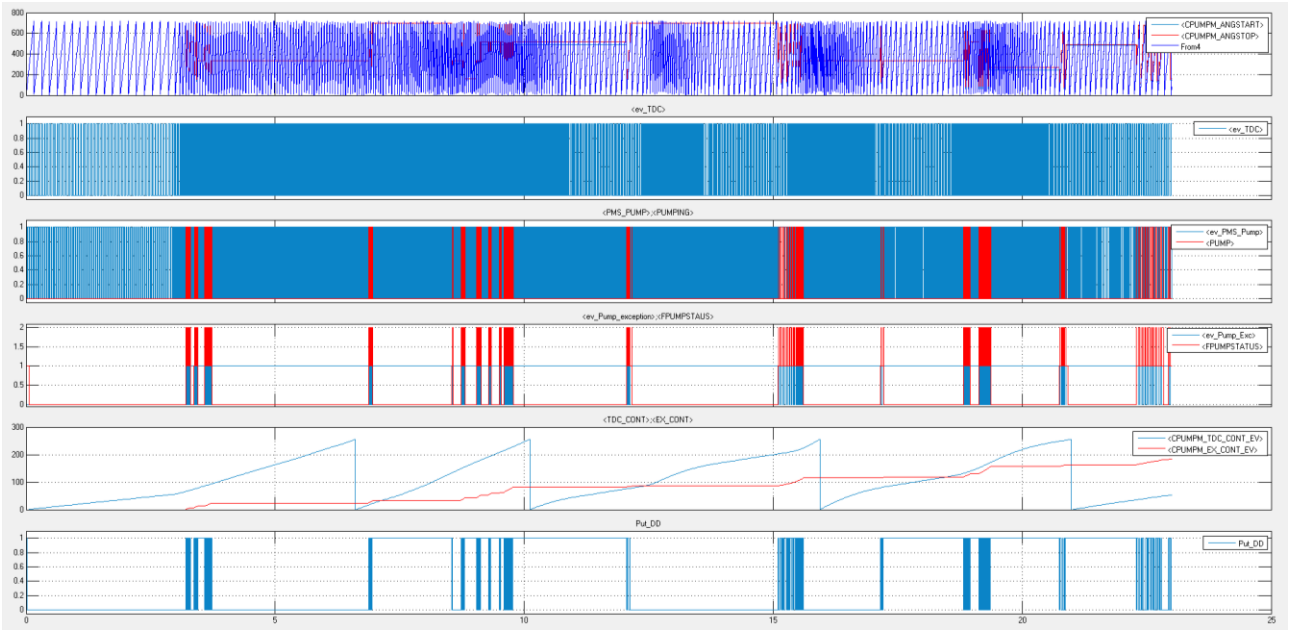


Figura 87 - Esempio grafico Scenario 1

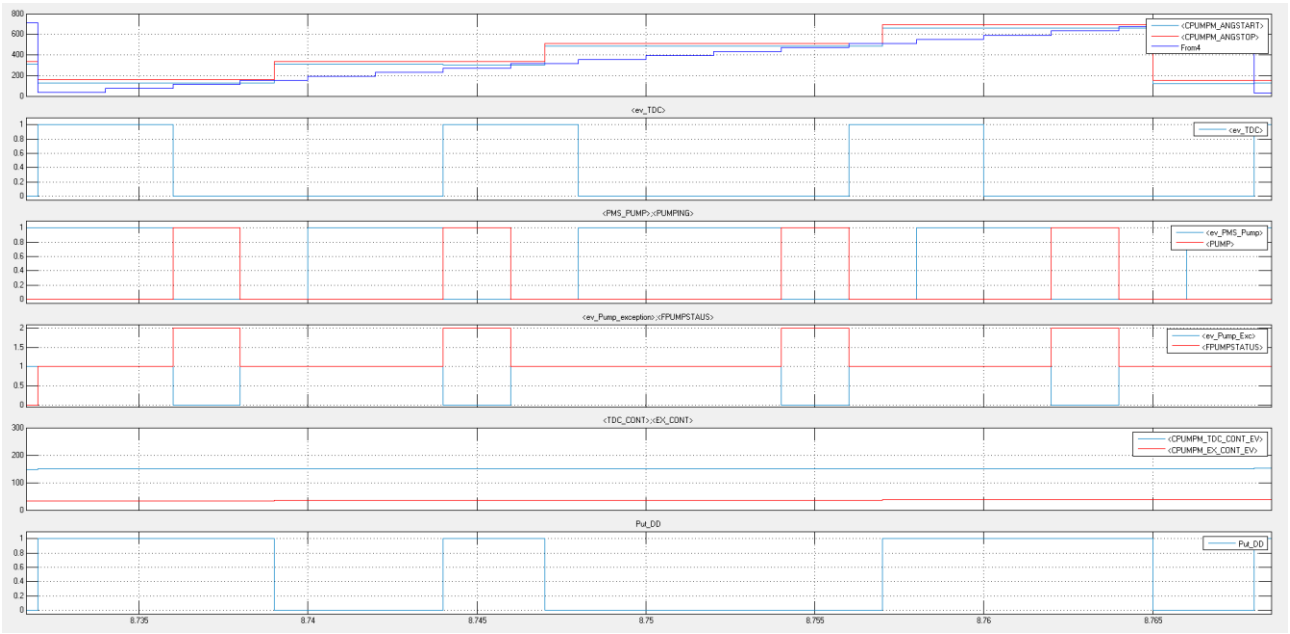


Figura 88 - Dettaglio grafico Scenario 1

In questo caso, al contrario di UTET i Test Pattern sono stati definiti con degli scopi ben precisi dove l’obiettivo è quello di simulare con accuratezza un aspetto particolare del modello.

Di seguito le descrizioni dei singoli Test Pattern e relativo grafico generale.

CPUMPM_TP_01 = Analisi di inizializzazione, fasabilità del valore di iniezione, switch tra i modi EOI e SOI (grafico già esposto)

CPUMPM_TP_02 = utilizzo della mappa RPRE MAP come Input

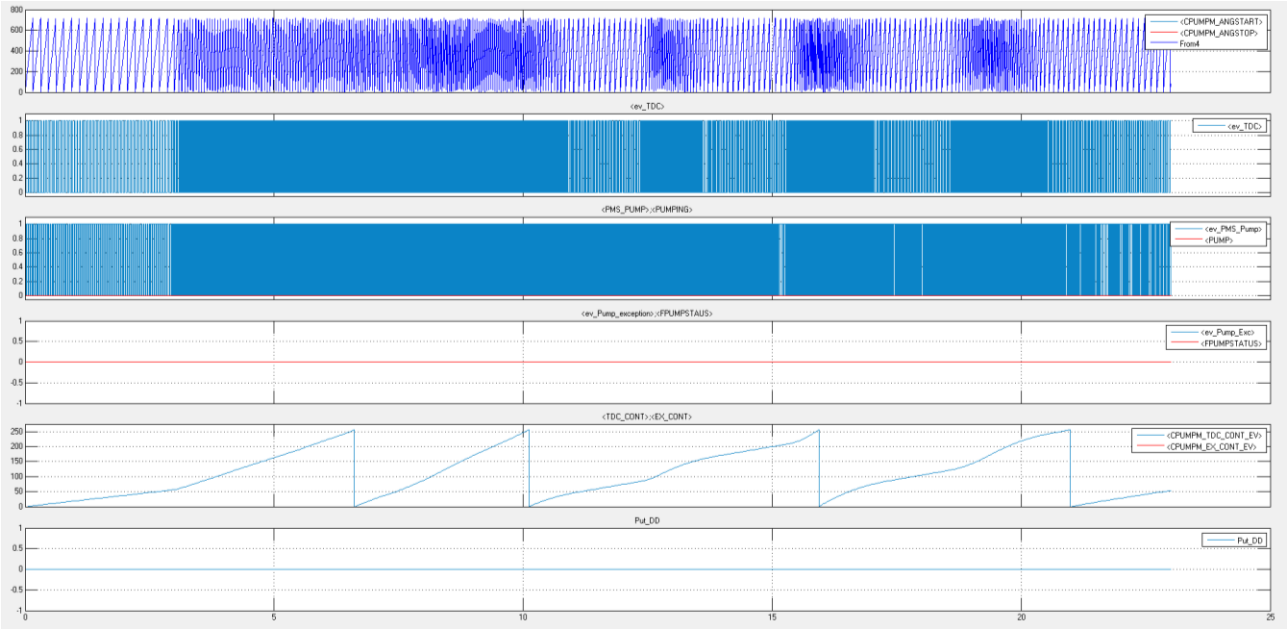


Figura 89 - Grafico Scenario_2

CPUMPM_TP_03 = Strategia di salto commando e utilizzo di mappe per ANGSTART e ANGSTOP

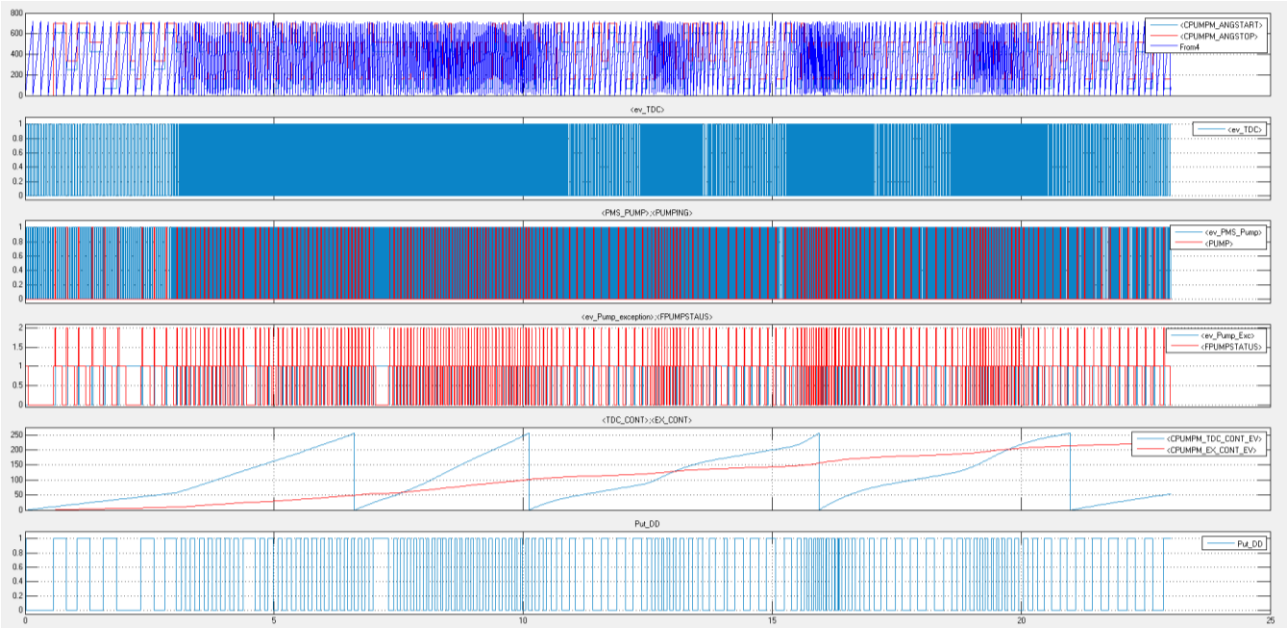


Figura 90 - Grafico Scenario_3

CPUMPM_TP_04 = Key_off, engine stopped, variazione di RPRE_TYPECTRL e RPRESTATE

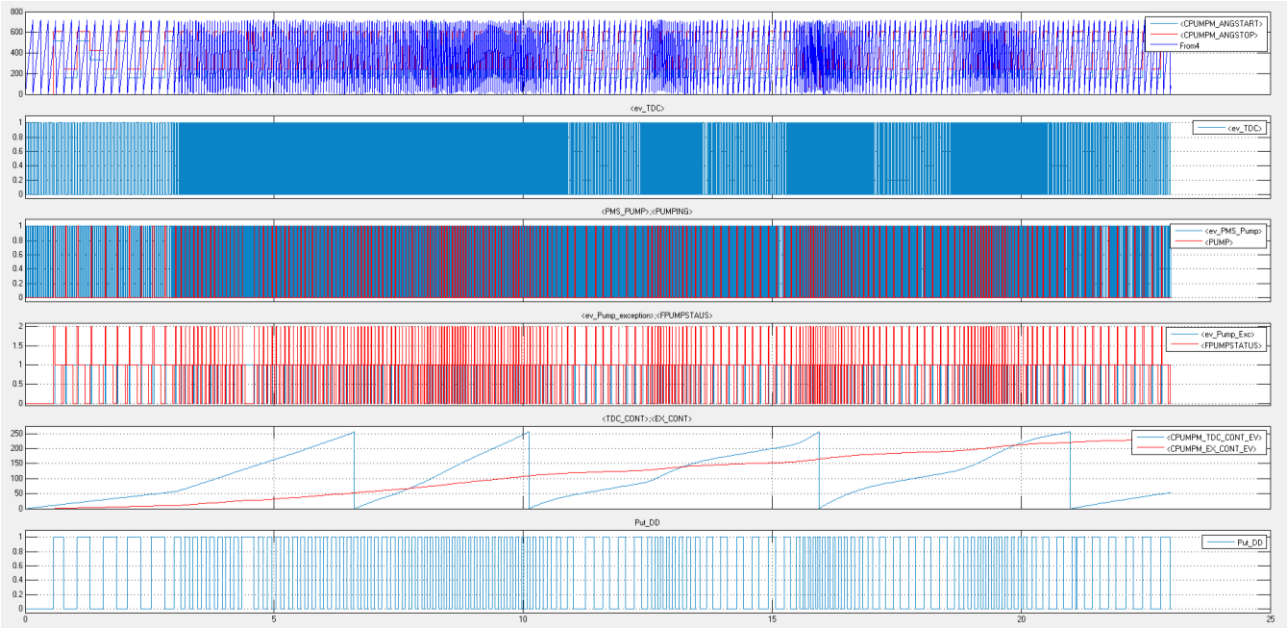


Figura 91 - Grafico Scenatio_4

CPUMPM_TP_05 = Valvola Normalmente chiusa, OFA Device Driver

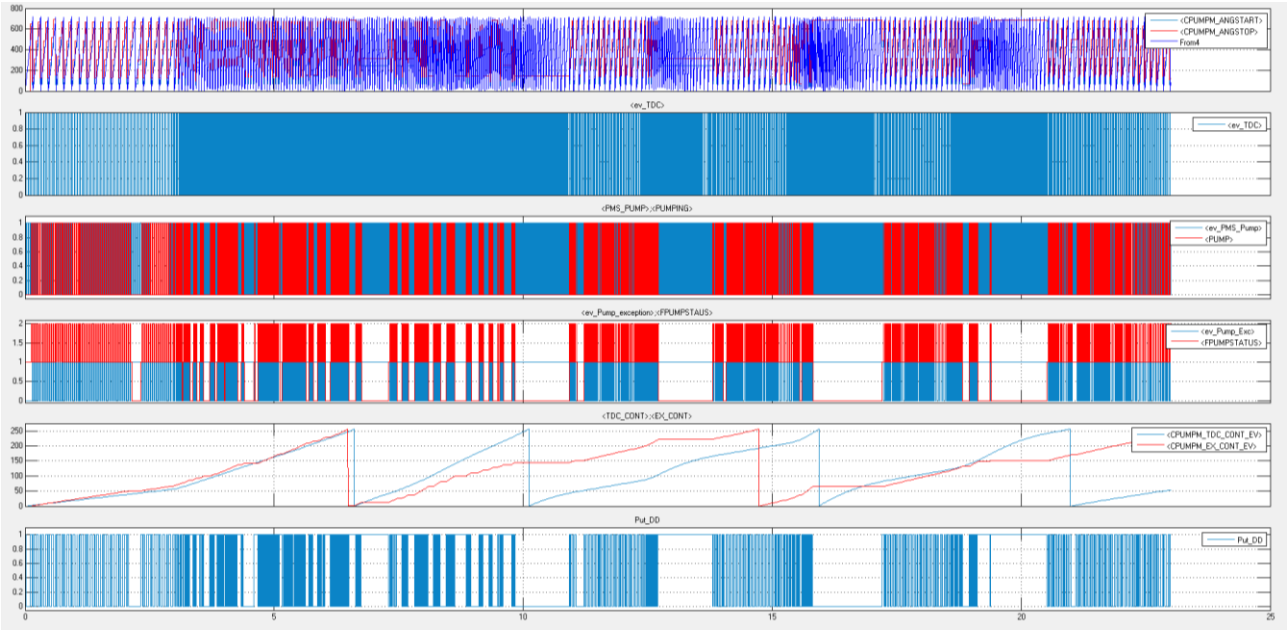


Figura 92 - Grafico Scenario_5

CPUMPM_TP_06 = Valvola Normalmente Aperta, OFA Device Driver, Hardware Type On/Off Command

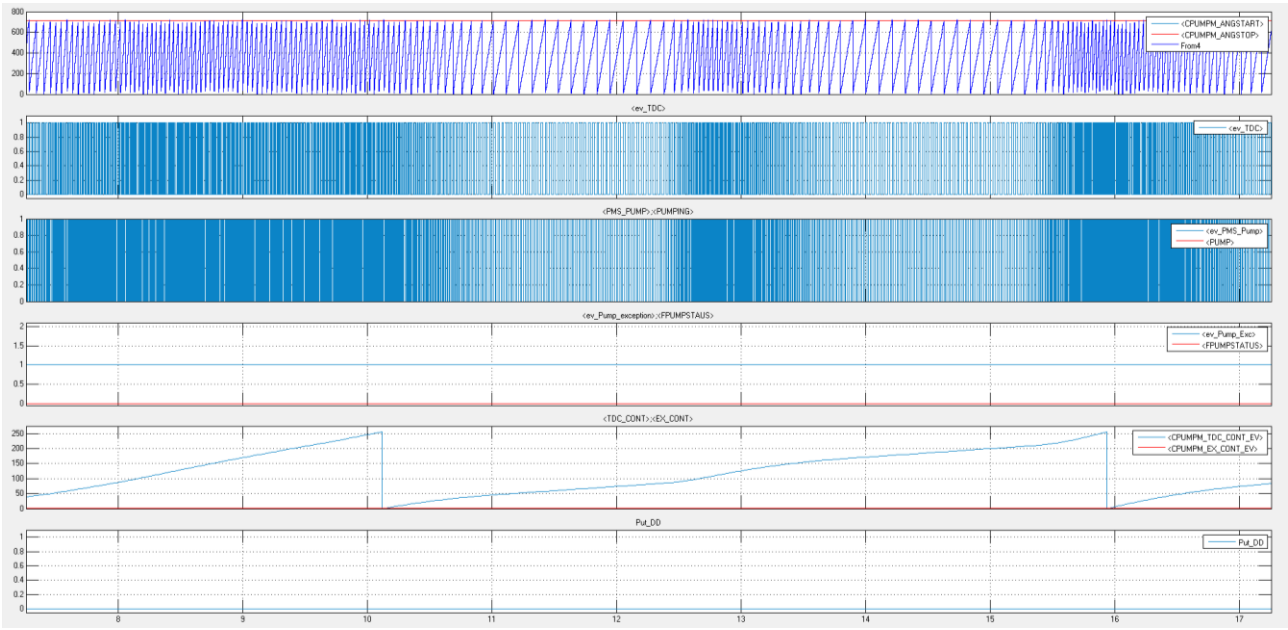


Figura 93 - Grafico Scenario_6

CPUMPM_OPI e CPUMPM_AC

In questo blocco sono presenti tutte le logiche di calcolo che verranno poi trasformate in codice C.

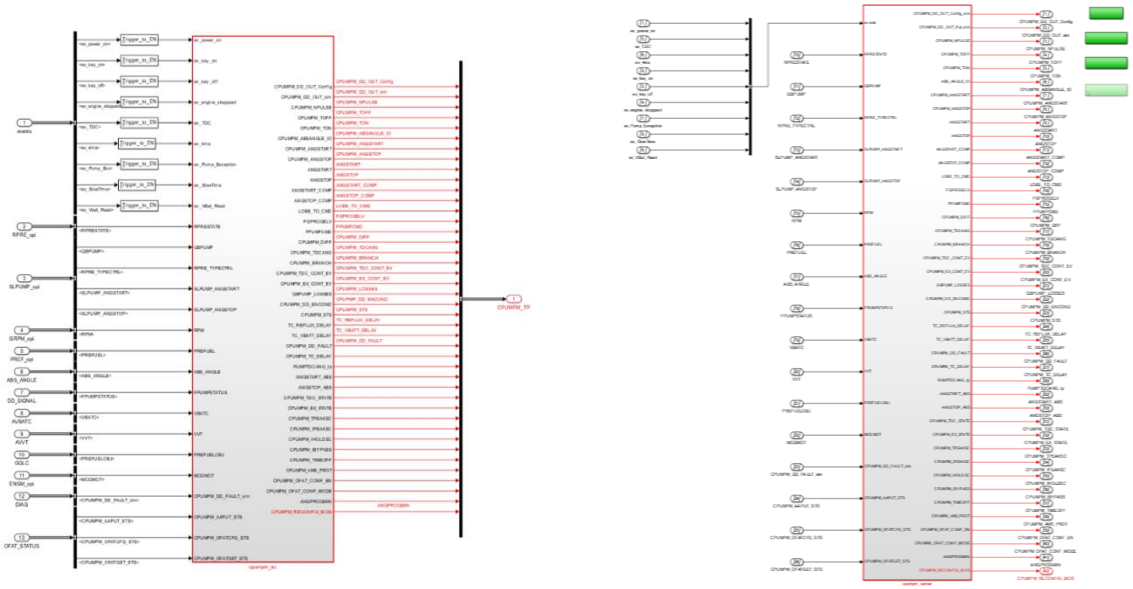


Figura 94 - Livello CPUMPM_opi (sinistra) e CPUMPM_ac (destra)

I livelli “opi” e “ac” non hanno funzioni inerenti alla generazione del codice. Il blocco OPI raccoglie in maniera ordinata input e output dalla cella di sintesi, il livello del blocco AC è il

primo dove si trovano delle variabili targhettate e altri blocchi esclusivi di Target Link (verdi) che configurano le dipendenze del codice generato da altri file e il nome della funzione C principale.

CPUMPM_SERVER

Il livello successivo CPUMPM_Server invece inizia a presentare una divisione per blocchi. Contrariamente dalla struttura presente in UTET, non è presente un unico blocco di Merge per le grandezze in quanto vengono aggiornate in punti differenti all'interno dei vari blocchi.

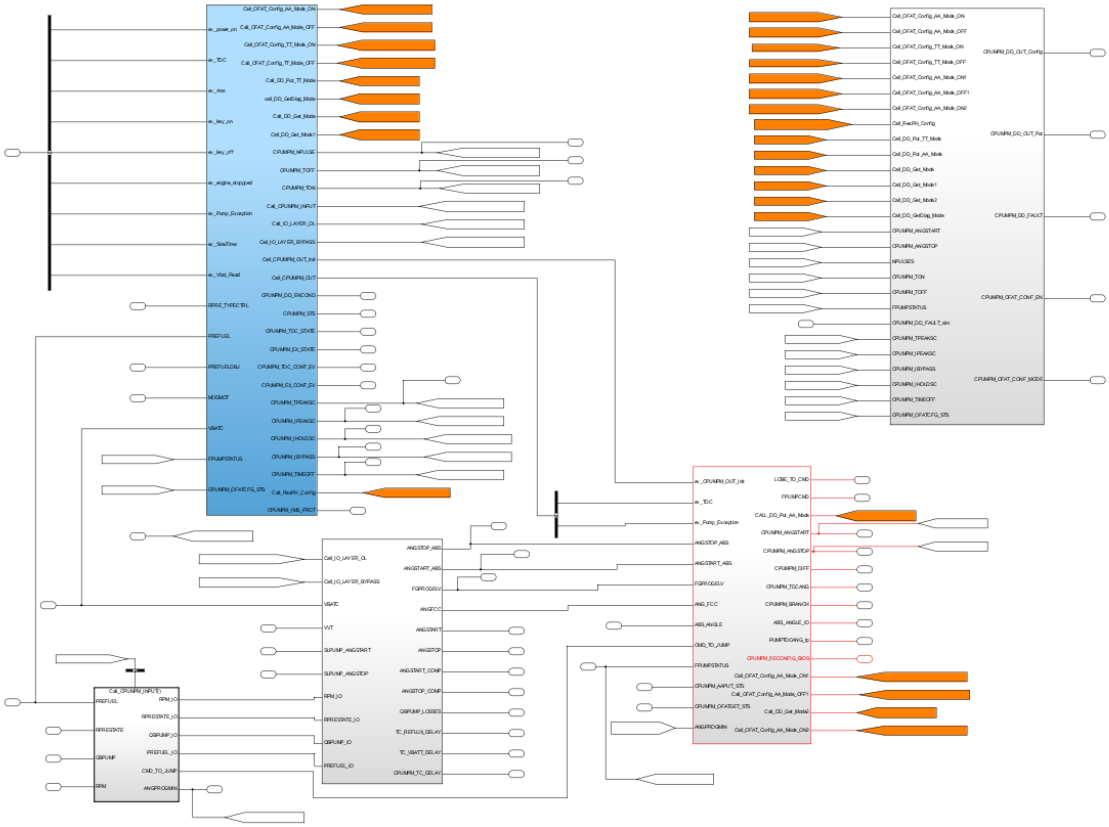


Figura 95 - Livello CPUMPM_Server

Al contrario del capitolo introduttivo, di seguito si analizza nel dettaglio ogni singolo subsystem partendo dal blocco azzurro e procedendo in senso antiorario.

SCHEDULER

Il primo blocco è chiamato **1-Scheduler**, ha come scopo quello di smistare le chiamate degli eventi fornendo loro un ordine di esecuzione qualora siano presenti più eventi contemporaneamente.

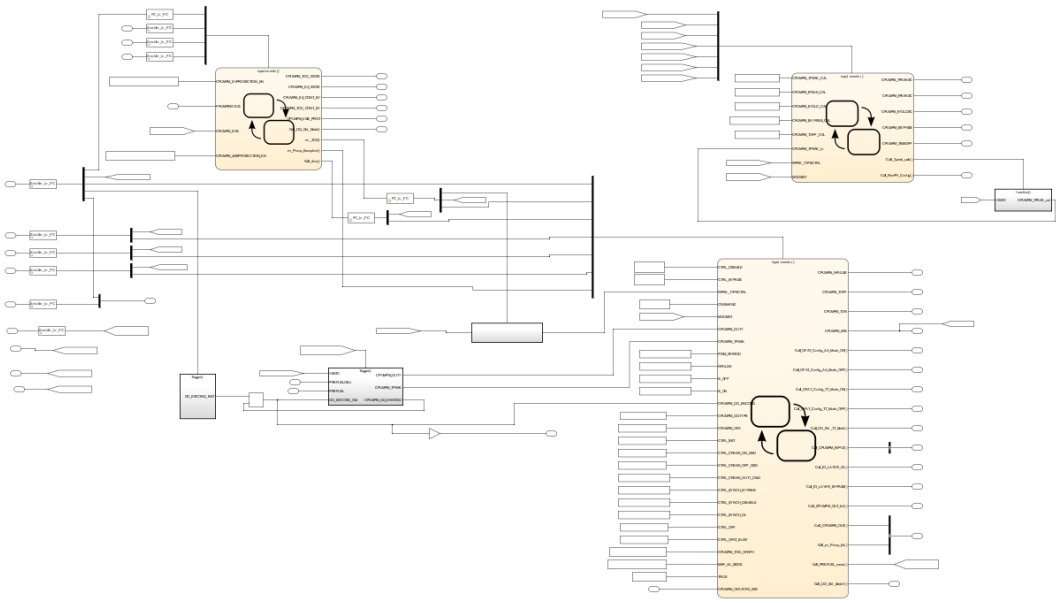


Figura 96 - Blocco 1-Scheduler

Il blocco è composto da 3 logiche Stateflow e 4 Subsystem per alcuni calcoli di rielaborazione di segnali.

Analizzando i blocchi Stateflow, quello in alto a sinistra (**1.1-EvProtection**) ha lo scopo di gestire la presenza di eventi concorrenti durante la simulazione che incidono nel calcolo dei parametri di esecuzione dell'attuazione.

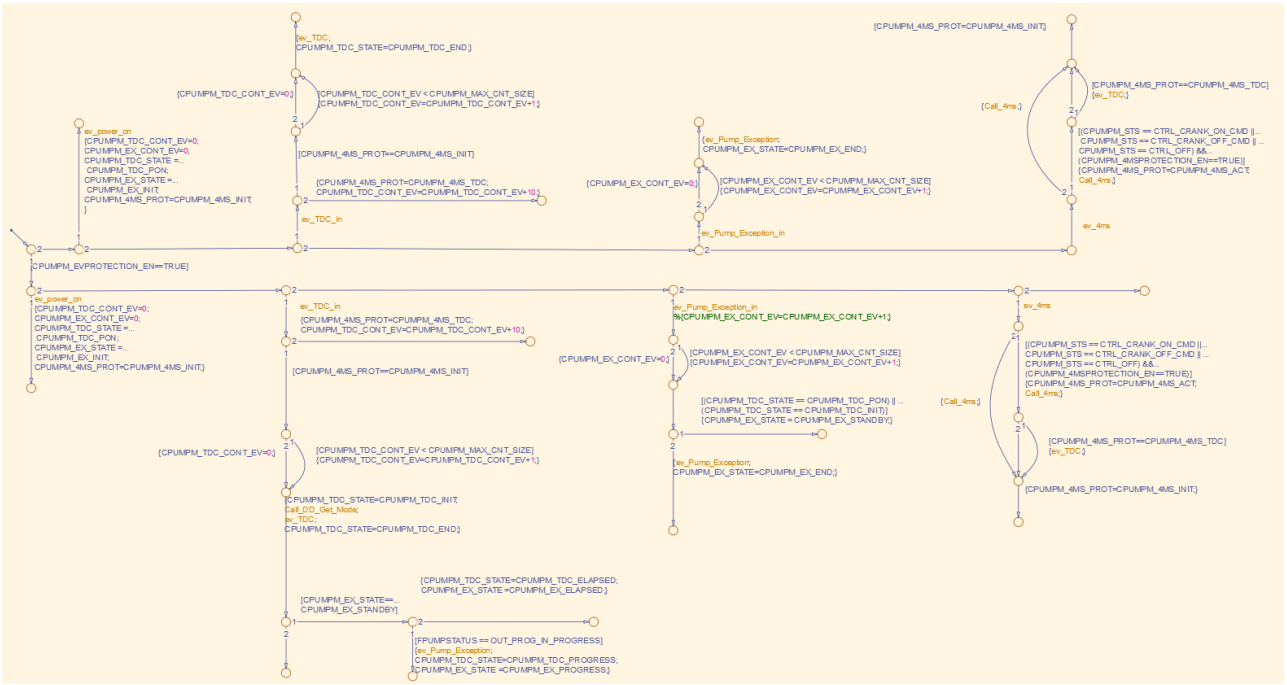


Figura 97 - Logica 1.1-EvProtection

Quando è presente un evento tra PumpEx, TDC e 4ms si esegue una verifica se è presente un altro evento tra quelli citati, in questo caso si esegue prima la logica di calcolo dell'evento con priorità maggiore e successivamente gli altri. Con le nuove logiche di modellazione AUTOSAR per runnable questo tipo di accorgimenti vengono tolti dalla logica del modello e impostati esternamente in fase di "integrazione" del componente SW nel sistema operativo.

Il blocco in alto a destra (**1.2_ResPH_mgm**) si occupa della gestione e del calcolo dei parametri legati all'attuazione del comando Peak & Hold. Tali parametri sono legati all'attuazione quindi a valori di tensione, corrente e tempo al fine di garantire una corretta attuazione. I tempi sono solamente relativi alle caratteristiche elettromagnetiche in questa fase. I ritardi dovuti al flusso vengono definiti in un'altra parte del modello.

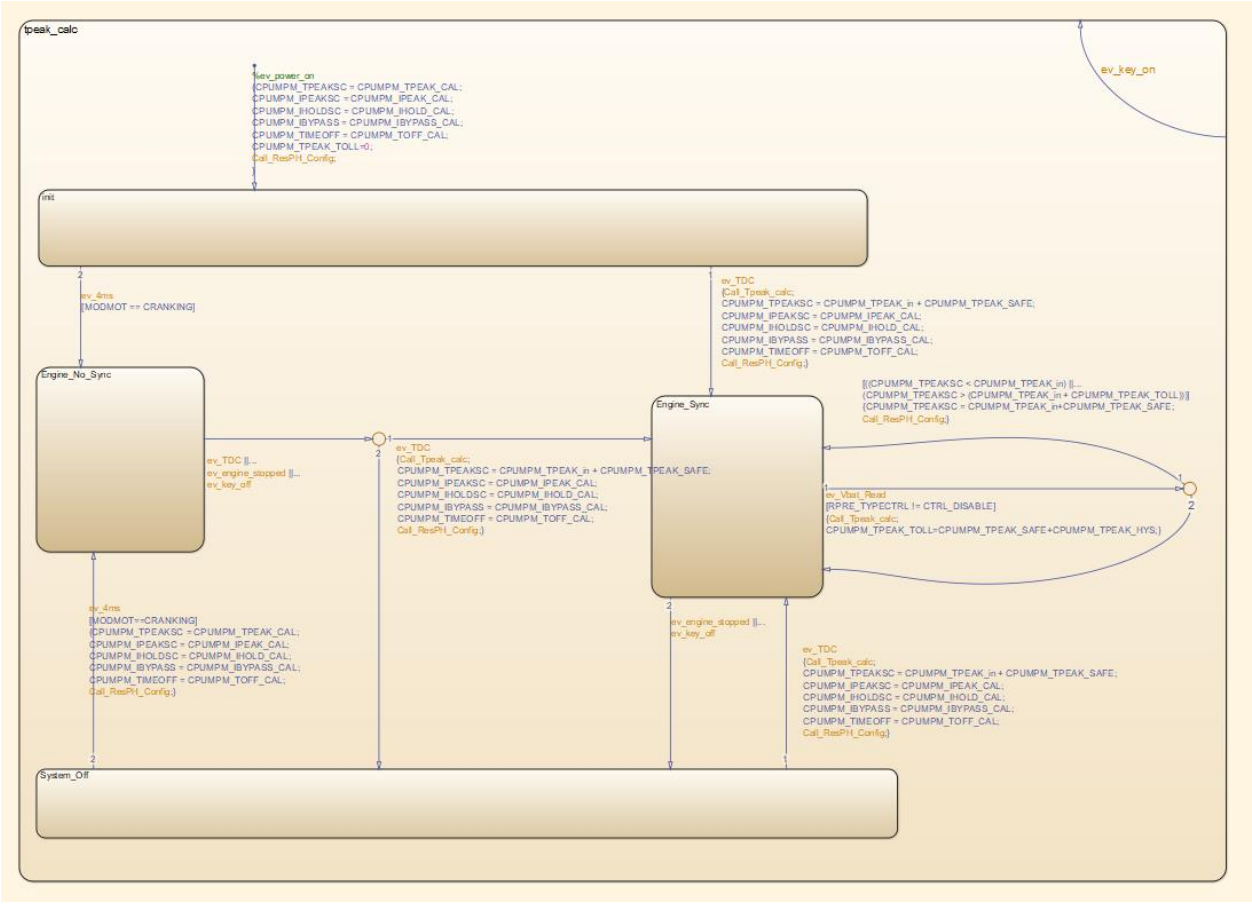


Figura 98 - Logica 1.2_ResPH_mgm

La gestione dei calcoli è definita da una macchina a stati divisa in 4 casi: Inizializzazione, Motore in Cranking, Motore Sincronizzato e Sistema disattivato. Quando il motore non è sincronizzato i valori sono determinati come valori di default. Nel caso invece sia nella condizione di sincronia, ogni volta che si ha un evento di lettura tensione batteria si aggiornano i parametri. L'aggiornamento dei parametri viene eseguito dal subsystem 1.2.1_TPeakNomCal

dove tramite il calcolo del valore presente all'interno di una mappa si definisce il valore del tempo di picco (Tp) del comando.

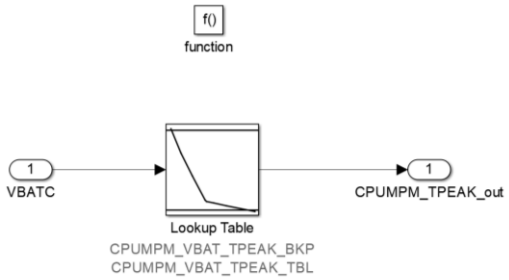


Figura 99 - 1.2.1_TPeakNomCal

Il blocco Stateflow presente in basso invece si occupa della gestione delle chiamate dei calcoli da eseguire in funzione dello stato del sistema.

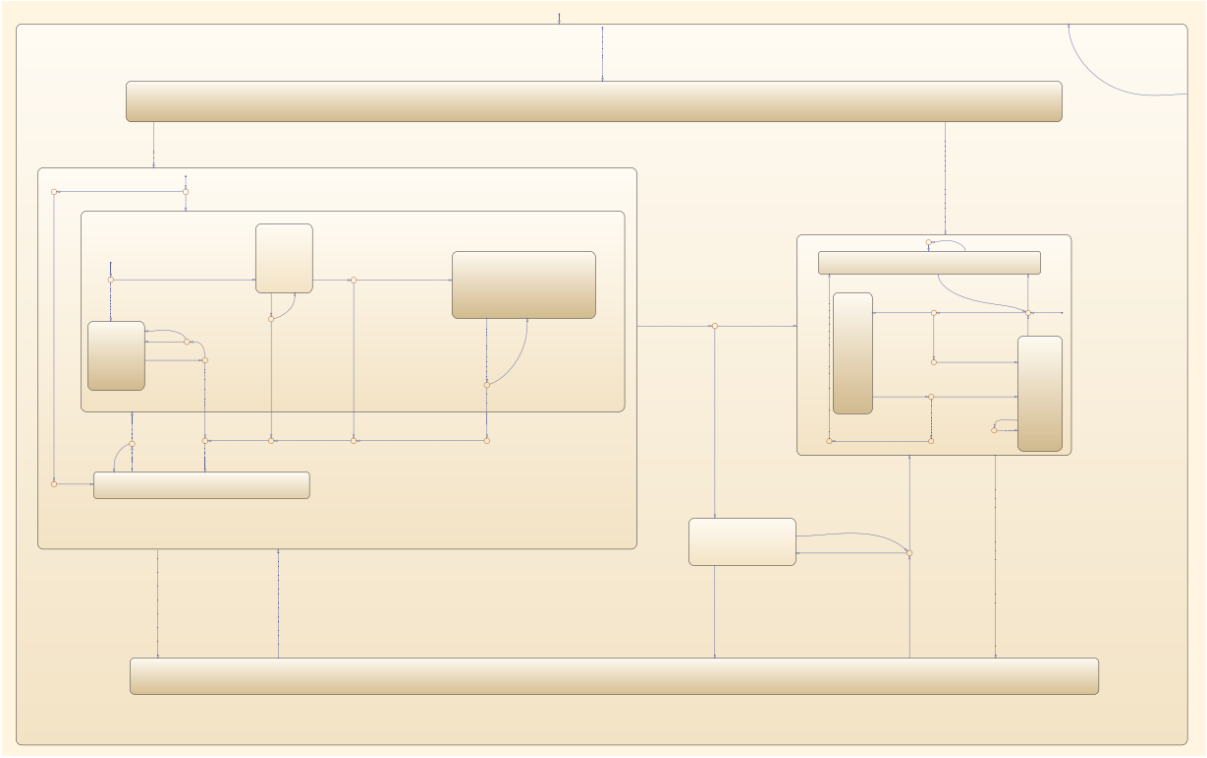


Figura 100 - 1.6_DD_Cal_Scheduler

Anche in questo è presente una macchina a stati con una divisione principale che ricorda la struttura della logica a stati del blocco 1.2, infatti si ha nella zona superiore lo stato di inizializzazione, nella parte di sinistra le logiche legate alla condizione di motore non in sincronia, nella parte di destra la logica legata al caso sincrono e nella parte bassa la logica di disabilitazione. Il blocco aggiuntivo presente tra lo stato di sincronia e quello di cranking è un blocco intermedio con lo scopo di gestire eventuali momenti di sovraccarico di eventi ed è

attivabile solo quando si sta passando da una condizione di cranking verso la condizione di sincronia e il Device Driver risulta occupato.

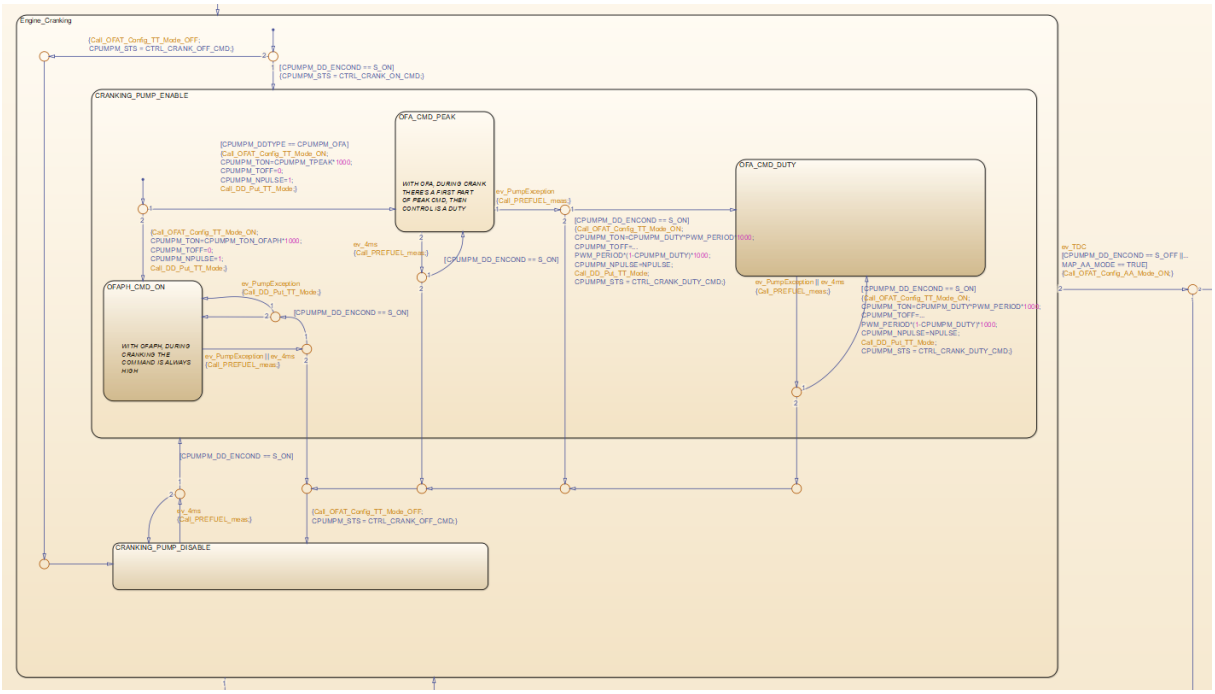


Figura 101 - dettaglio logica cranking

Durante la fase cranking si possono avere due condizioni principali: comando *attivato* o *disattivato*.

Quando si è nella condizione di comando attivato, si hanno tre possibili condizioni:

- **Comando attivo** con calcolo dei parametri di attuazione.
- **Comando attivo con valori massimi di attuazione** (in questo caso non eseguo il calcolo dei tempi di picco).
- **Condizione di comando forzato** (in questo caso si eseguono le verifiche e i calcoli dei parametri ad ogni evento 4ms o PumEx).

Quando previsto, si esegue una chiamata sulla pressione nel rail in modo da conoscere quanto si è lontani dal target.

Il calcolo viene eseguito dal blocco 1.5_PwdDutycicleCalculation il quale tramite delle mappe ridefinisce i tempi di picco e se è necessario eseguire la pompata.

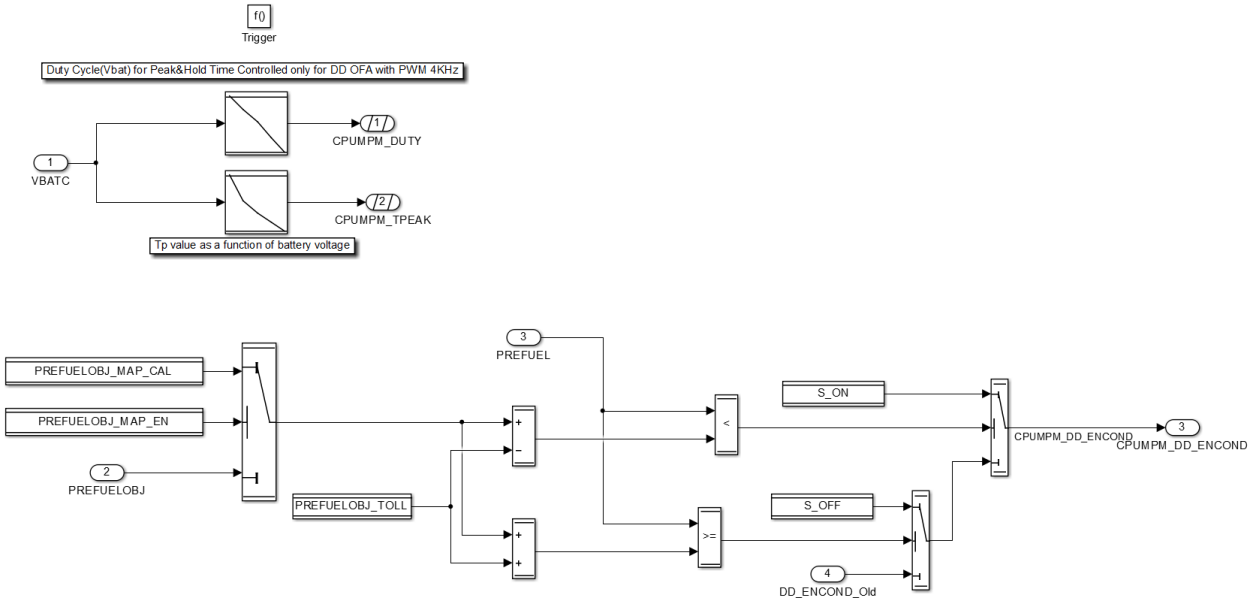


Figura 102 - 1.5_PwdDutycicleCalculation

L'uscita da uno di questi stati interni verso altri avviene solo tramite la presenza di un evento 4ms o PumEx, in corrispondenza del primo evento TDC si passa allo stato di sincronia dove sono presenti 3 stati interni:

- **Bypass** = non eseguo i calcoli di apertura e chiusura in funzione del modello inverso ma solamente tramite dei valori prestabiliti.
- **OpenLoop** = corrisponde alla configurazione dove vengono eseguiti tutti i calcoli riguardanti gli angoli di apertura e chiusura.
- **Disable** = Il comando è disabilitato e non si eseguono i calcoli di aggiornamento.

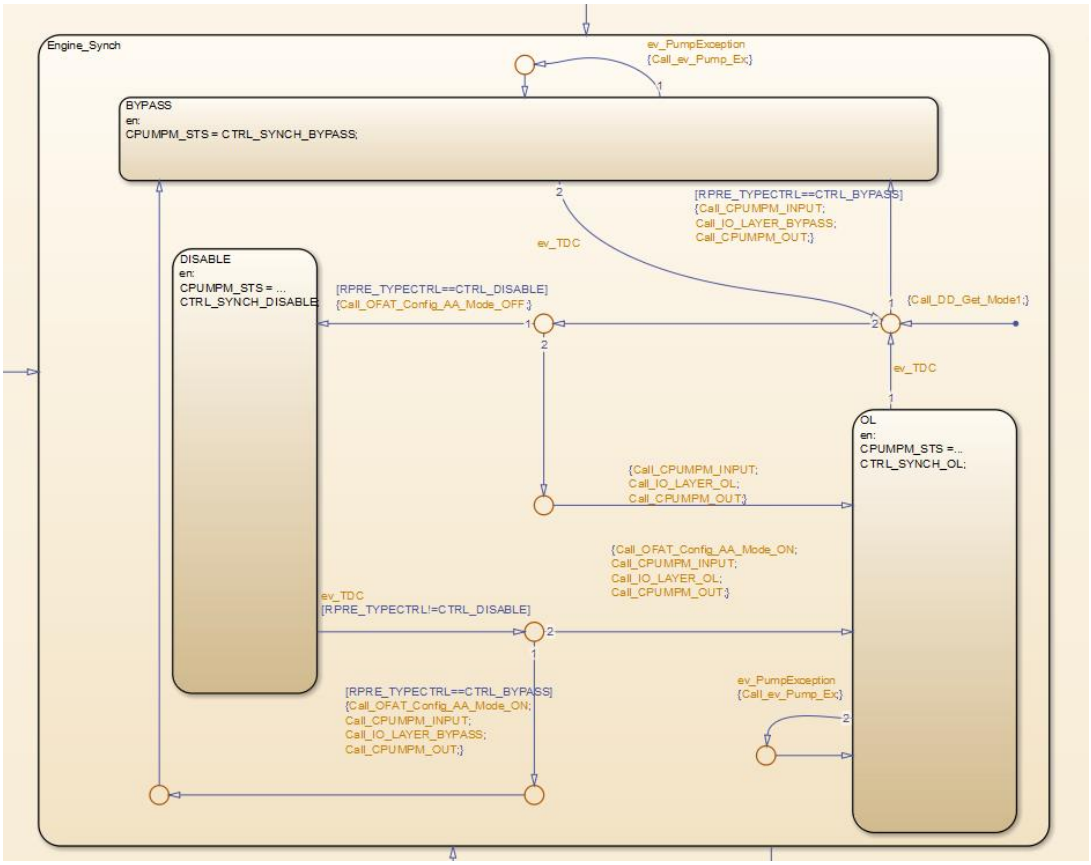


Figura 103 - Dettaglio logica Sincrona

Il passaggio tra due stati interni avviene per effetto di un evento TDC o PumEx e qualora si fosse in condizione di Bypass o OpenLoop si eseguono i calcoli dedicati.

INPUTLAYER

A questo punto l'analisi della logica si sposta al blocco 2.InputLayer, questo viene attivato dalla Function Call "Call_CPUMPM_INPUT" che è presente nel blocco 1.6_DD_Cal_DeviceDriver dopo un evento TDC.

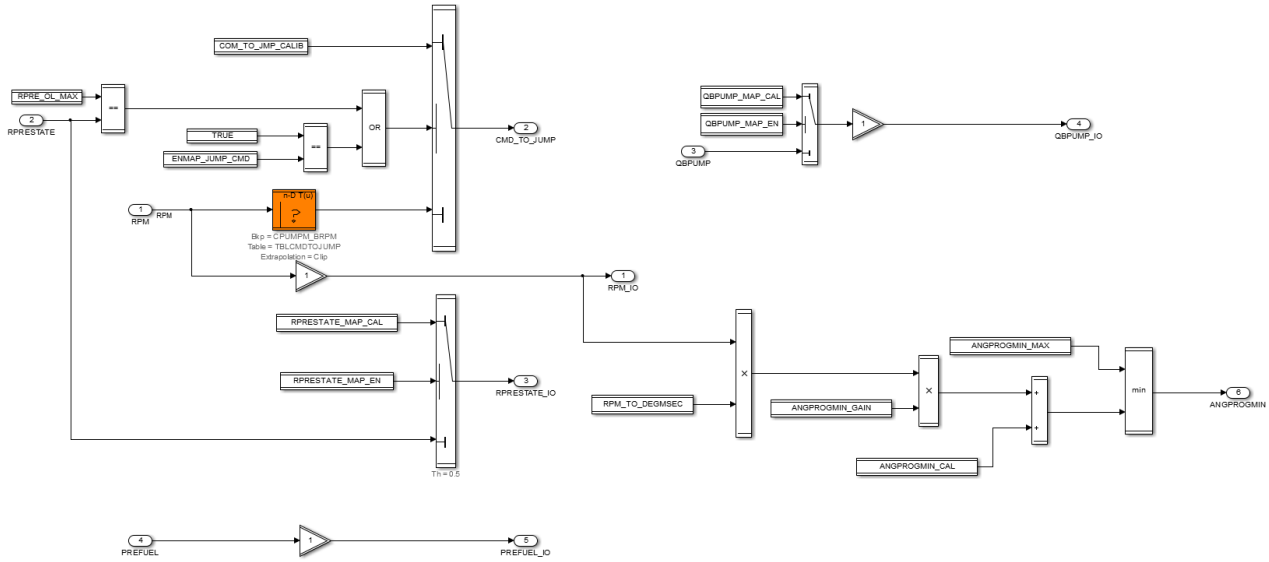


Figura 104 - Blocco 2.InputLayer

Questo blocco serve come inizializzazione della catena di calcolo per la definizione degli angoli di attuazione. Fornisce il valore dell'angolo minimo per eseguire la programmazione, i giri motore, la portata da pompare e la pressione nel rail. Inoltre, fornisce due flag: se il comando è da ignorare e lo stato del regolatore di pressione nel rail.

Dopo aver inizializzato i calcoli si passa al blocco successivo 3.IO_Layer. In questo blocco avvengono i calcoli in valore relativo degli angoli di inizio e fine comando.

IO_LAYER

Al suo interno vi sono più logiche, in quanto viene attivato sia se il sistema è nello stato di controllo sincrono OpenLoop sia se il controllo è nello stato Bypass. Da notare che contrariamente a quanto presente in UTET, la scelta della variabile più recente viene eseguita comunque tramite un blocco Merge ma non è stato deciso di costruire un subsystem dedicato.

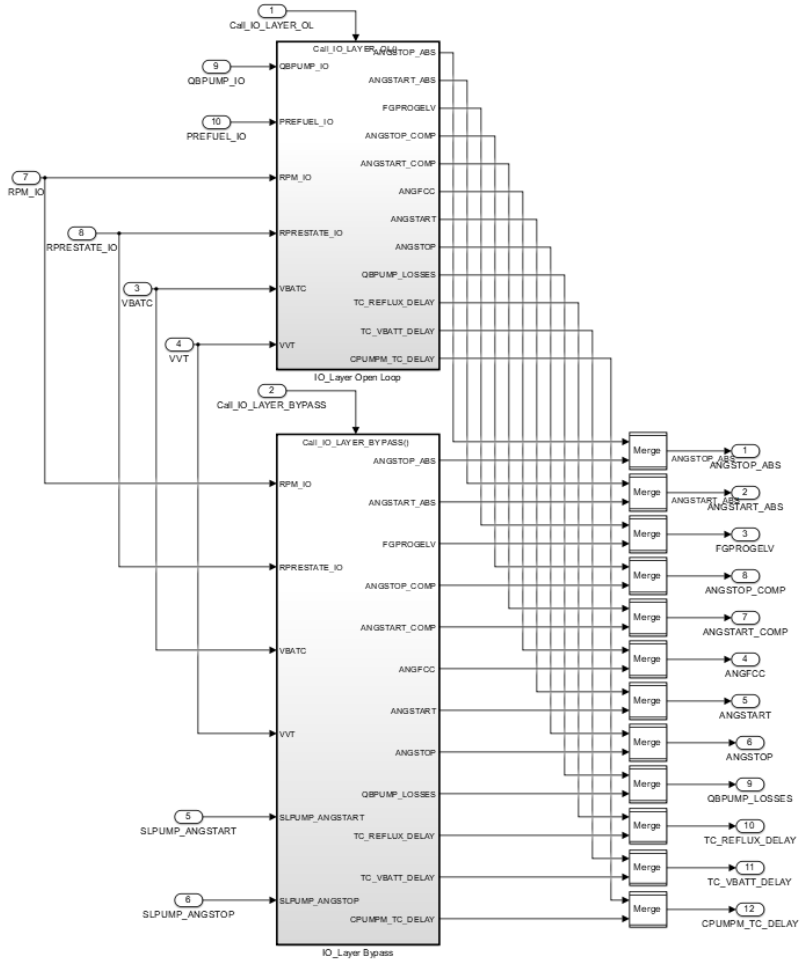


Figura 105 - Blocco 3.IO_Layer

LOGICA OPEN_LOOP

Analizzando la logica Open Loop si trovano tre subsystem al suo interno ognuno con uno scopo differente: angolo di inizio comando relativo alla pompa, angolo di fine comando relativo alla pompa e la trasformazione da angolo relativo alla pompa ad angolo assoluto motore.

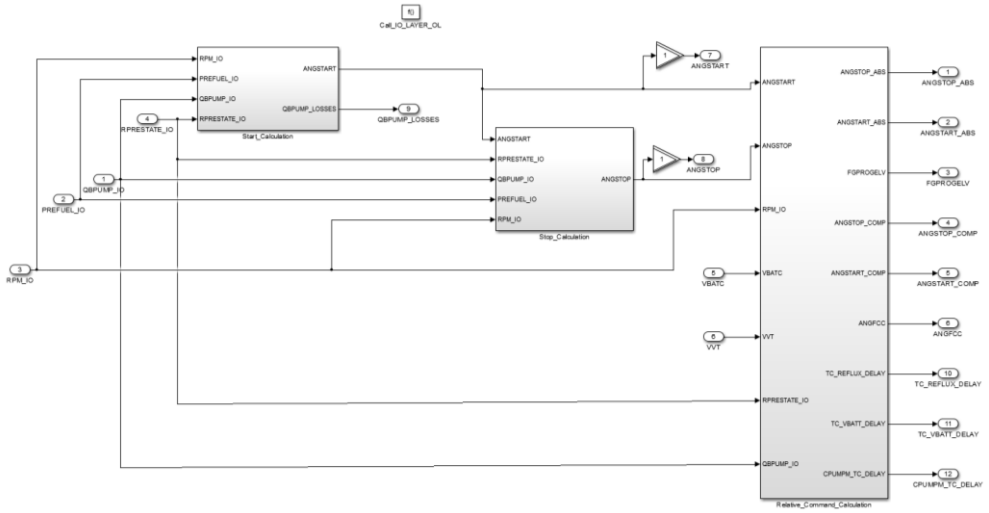


Figura 106 - Blocco IO_Layer_OpenLoop

Considerando il blocco inerente al calcolo dell'angolo di apertura, si può notare che nella logica sono presenti due subsystem, uno che fa riferimento alla valvola normalmente aperta (il componente che vogliamo gestire) e uno che fa riferimento a una valvola normalmente chiusa. L'attenzione nella descrizione pertanto sarà solamente per il caso di valvola normalmente aperta, relativa al componente da controllare.

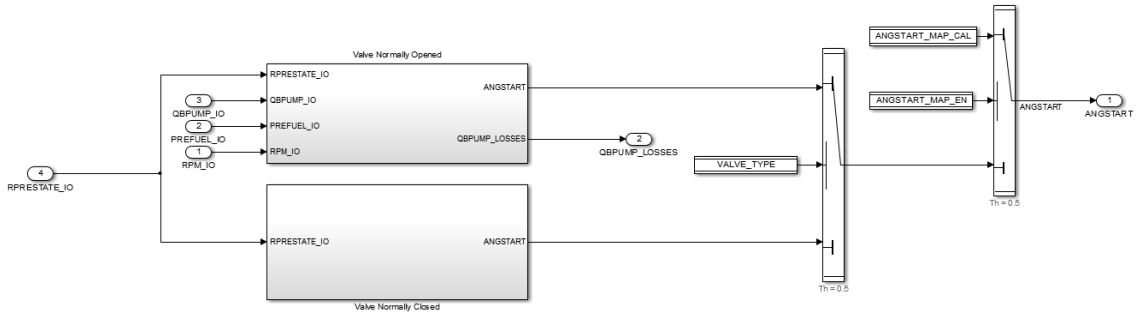


Figura 107 - Logica ANGSTART

La logica per definire gli angoli di apertura partendo dalla portata desiderata si basa sulla definizione della portata effettiva necessaria considerando le perdite di trafilamento interne,

successivamente, tramite una mappa che considera la portata richiesta e la velocità di rotazione ottengo l'angolo necessario per l'inizio dell'attuazione.

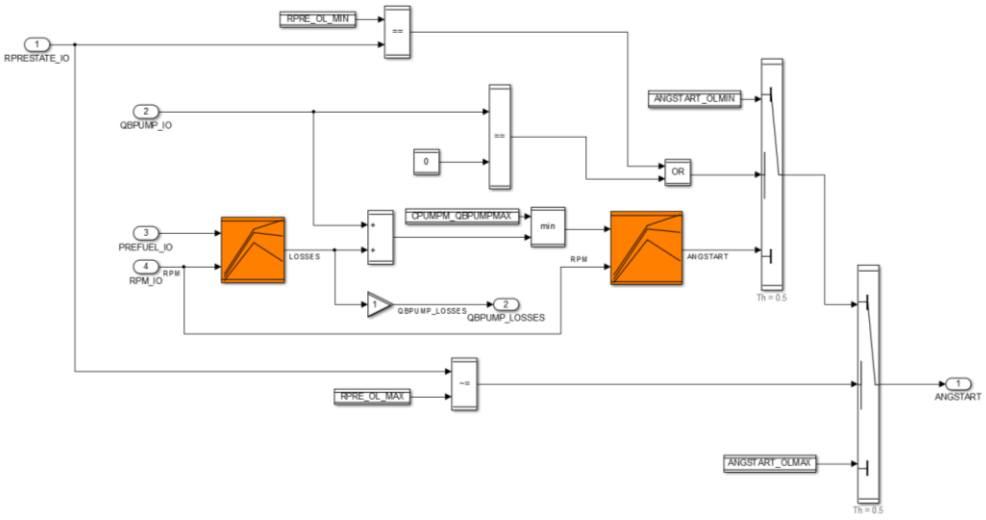


Figura 108 - Logica calcolo ANGSTART valvola normalmente aperta

Analogamente al caso della logica di calcolo dell'angolo di inizio attuazione anche per il blocco di calcolo dell'angolo di fine comando sono presenti i due blocchi di logica relativi al tipo di valvola.

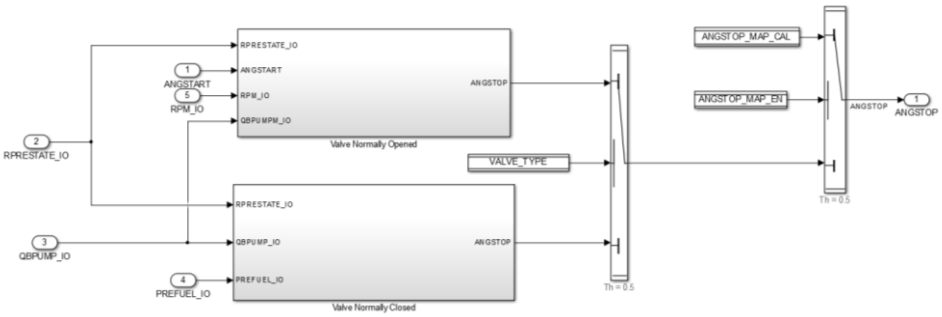


Figura 109 – Logica ANGSTOP

Il calcolo della logica dell'angolo di fine comando è più semplice in quanto devo solo definire il tempo necessario per garantire la chiusura per sovrapressione del piattello e tradurlo in angolo motore.

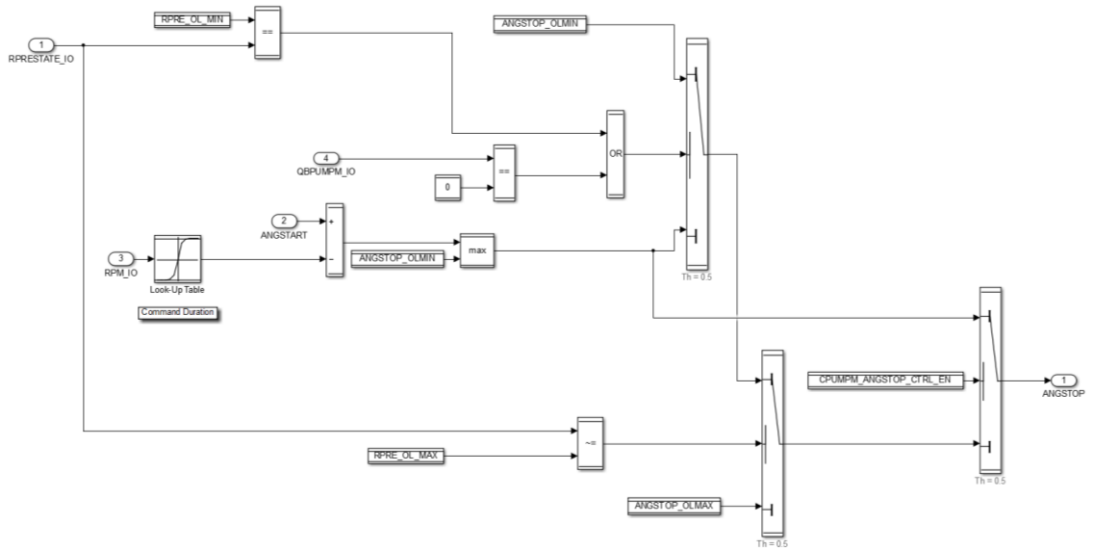


Figura 110 - Logica di calcolo ANGSTOP valvola normalmente aperta

In entrambe le logiche di calcolo si può notare la presenza di alcune verifiche di "soglia", infatti quando i valori sono superiori o inferiori al range di calibrazione di default si assumono i valori di massimo o minimo.

Ottenuti i valori di inizio e fine comando rispetto all'angolo di ciclo del pistone della pompa (0-360), è necessario trasportare questo valore in funzione dell'angolo assoluto del motore (0-720). Contrariamente al caso di calcolo degli angoli di inizio e fine comando, la definizione degli angoli assoluti non ha due logiche separate in funzione del tipo di valvola considerato.

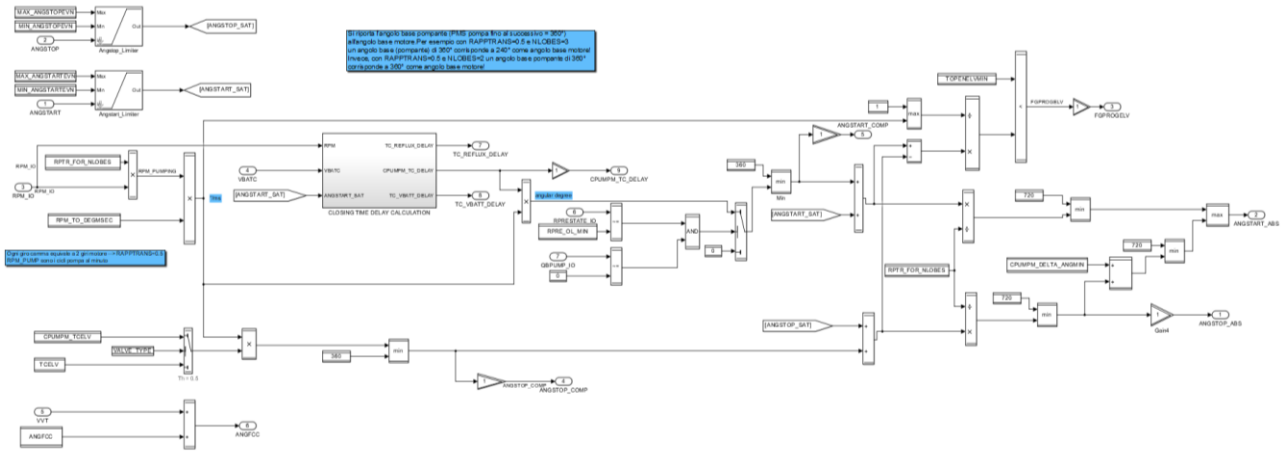


Figura 111 – logica di calcolo ABS_ANGLE

La struttura della logica di calcolo si basa sul definire come prima cosa il tempo di ritardo di attuazione in funzione della velocità di rotazione del motore e della tensione della batteria (in termine di angolo).

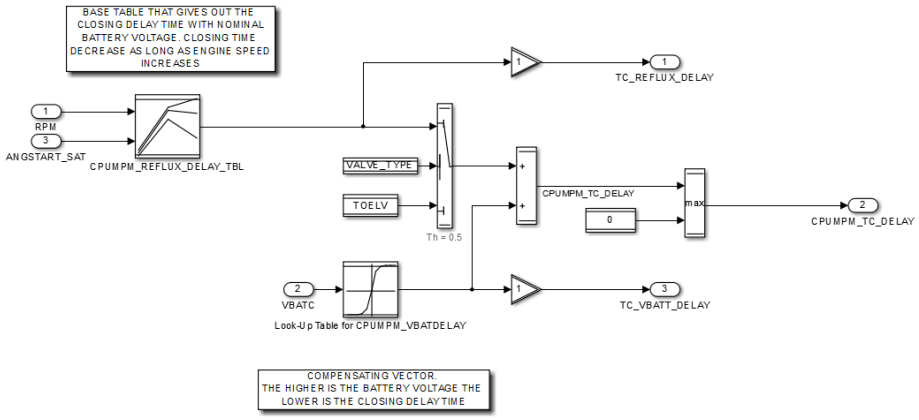


Figura 112 - Dettaglio blocco calcolo ritardo di attuazione

La logica per definire l'angolo assoluto di inizio e fine comando hanno due percorsi differenti.

Analizzando il percorso dell'angolo di inizio, dopo aver calcolato gli angoli di ritardo dell'attuazione, si esegue una verifica logica se il comando "è necessario" e "attuabile". Da queste valutazioni si ottiene il valore logico della variabile FGPROGELV.

Nel calcolo degli angoli assoluti, per quanto riguarda l'angolo di inizio attuazione si sommano i valori di angolo di ritardo nell'attuazione e dell'angolo calcolato precedentemente che garantisce la portata desiderata. La trasformazione del valore assoluto avviene considerando il rapporto di trasmissione dell'albero a camme e il numero di lobi della camma di attuazione.

La trasformazione dell'angolo di chiusura avviene come nel caso dell'angolo di apertura considerando il rapporto di trasmissione tra albero motore e albero a camme e il numero di lobi.

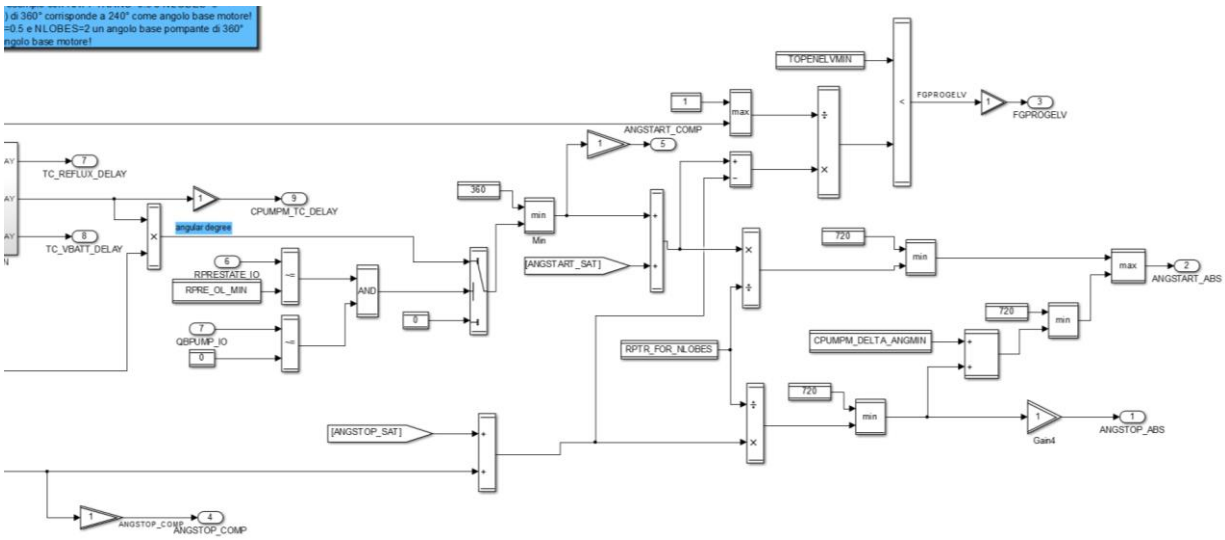


Figura 113 - Dettaglio logica calcolo angoli assoluti

A questo punto con i valori assoluti degli angoli di comando e l'utilità di eseguire il comando si passa alla definizione nel dettaglio del comando nel blocco successivo.

LOGICA BYPASS

Tornando al livello iniziale del subsystem 3.IO_Layer, si vuole passare all'analisi del subsystem di calcolo degli angoli di esecuzione comando per quanto riguarda la condizione di Bypass.

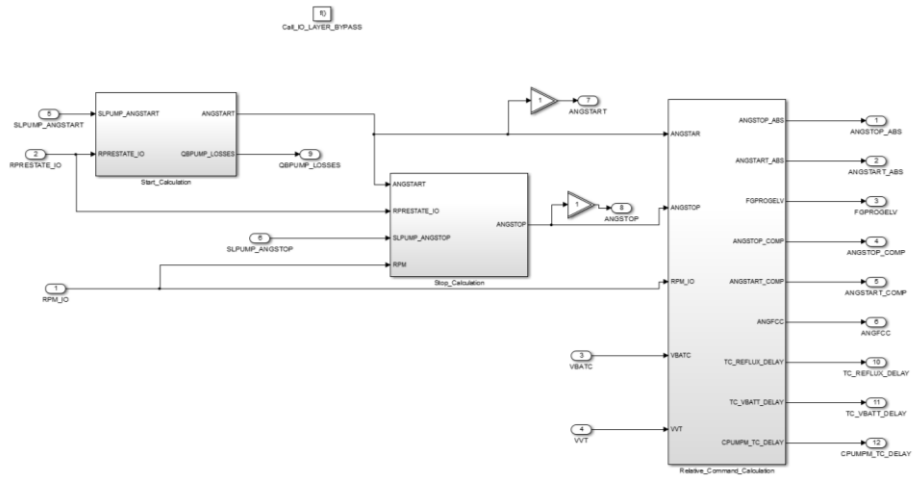


Figura 114 - Blocco 3.IO_Layer – bypass

Come per il caso OpenLoop anche in questo livello sono presenti i tre blocchi di calcolo relativi alla definizione di angolo di inizio e fine comando e il blocco di trasformazione dal valore relativo a quello assoluto.

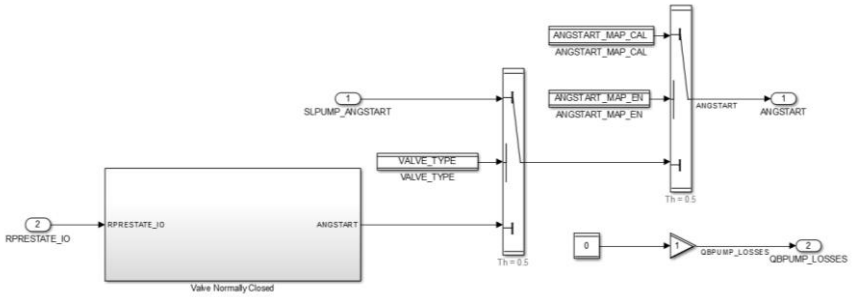


Figura 115 – Logica ANGSTART

Nella logica di ANGSTART non sono presenti calcoli particolari, nel caso di valvola normalmente aperta il valore viene semplicemente definito dalla costante SLPUMP_ANGSTART.

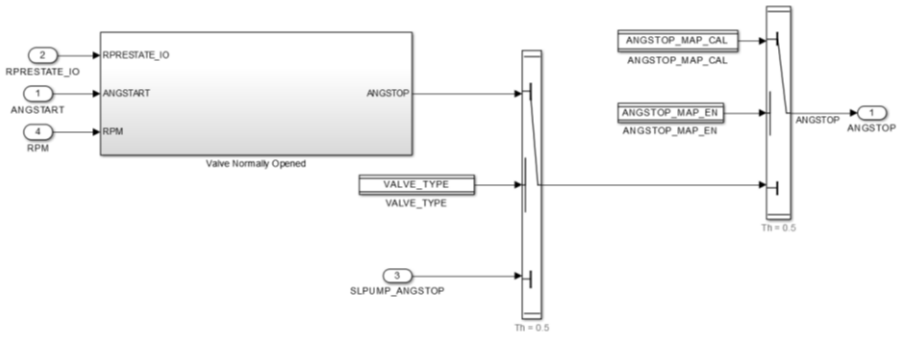


Figura 116 - Logica ANGSTOP

La logica di calcolo di ANGSTOP è la stessa presente nel calcolo della logica OpenLoop vista in precedenza. Analoga considerazione per il blocco di definizione degli angoli assoluti riportato di seguito.

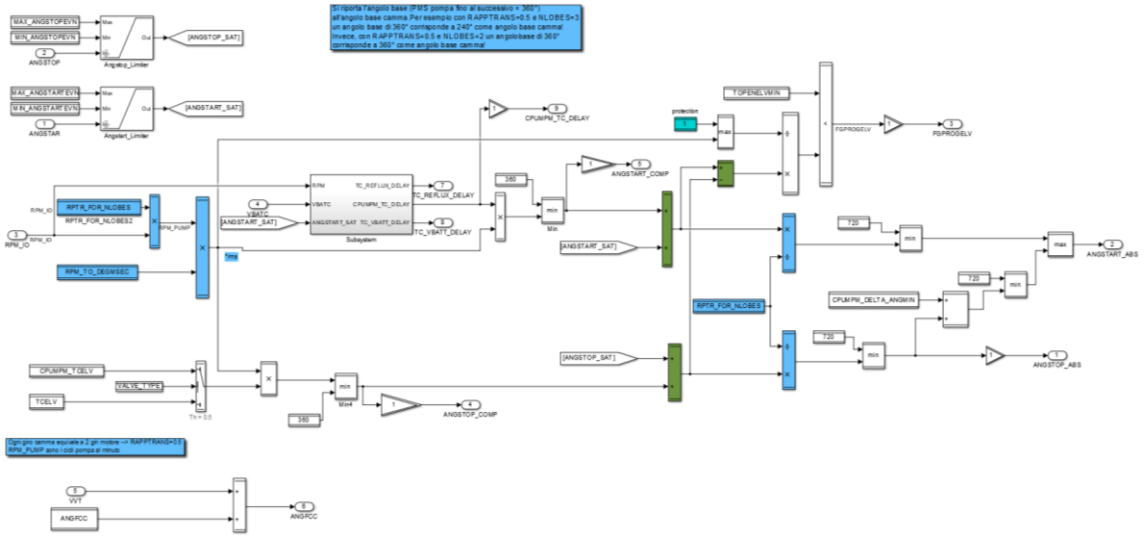


Figura 117 - logica ABS_ANGLE

Come per il caso OpenLoop con la definizione degli angoli assoluti si completano i calcoli logici presenti nel blocco 3.IO_Layer.

OUTPUT_LAYER

Il blocco 4.OUTPUT_Layer è l'ultimo blocco di calcolo presente nel modello e serve a definire tutte le grandezze da comunicare verso l'esterno del modello ed esegue le Function Call che attivano le parti di logica che comunicheranno con il Device Driver.

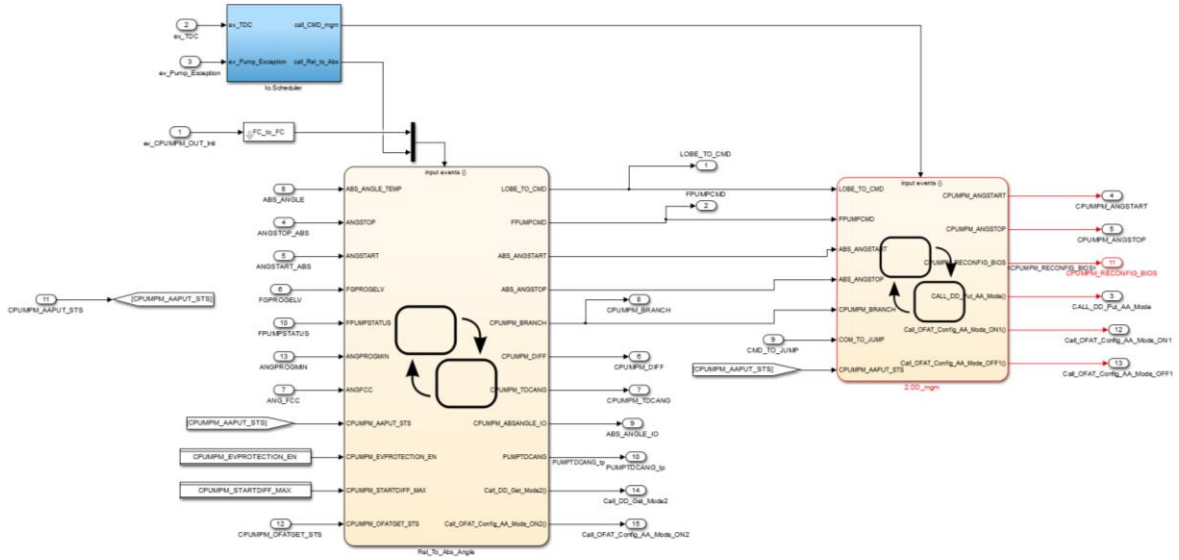


Figura 118 - Blocco 4.OUTPUT_Layer

Il blocco azzurro è un blocco di schedulazione senza particolari logiche, contiene al suo interno dei blocchi di Mux (come quello sopra allo stateflow di sinistra) utile solo a definire la posizione d'ingresso nella logica stateflow degli eventi TDC e PumEx.

Il blocco stateflow di sinistra è il primo che viene attivato e ha al suo interno delle logiche ad albero.

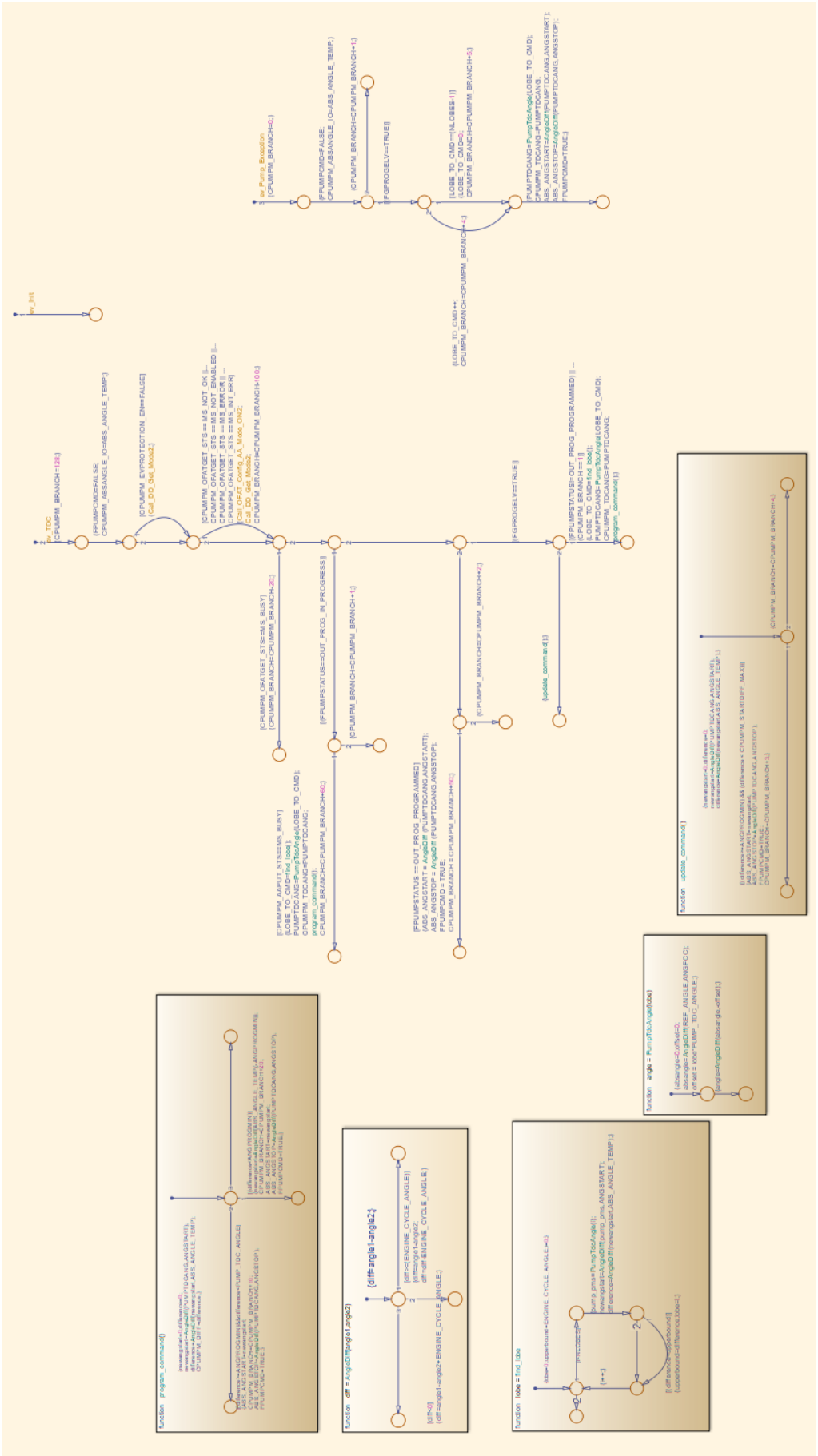


Figura 119 - Logica albero Stateflow 4.1_Rel_To_Abs_Angle

In questo blocco in funzione dell'evento TDC o PumEx (il terzo evento è il PowerOn e ha lo scopo di inizializzare la logica) esegue una serie di operazioni volte a definire quale è l'angolo di inizio e fine comando da comunicare al Device Driver. Oltre alla logica ad albero si può notare che per rendere più leggibile il grafico Stateflow sono state usate anche delle Stateflow graphical function.

Considerando la logica del TDC, una volta inizializzato se necessario, vi sono 3 possibili percorsi principali per la definizione degli angoli. La discriminante per il percorso da intraprendere è lo stato del sistema e se è già stata programmata una pompata. La logica di esecuzione è basata sulla determinazione del lobo che si intende usare per il comando, definire quindi l'angolo motore necessario all'esecuzione della pompata, il tutto verificando la possibilità di esecuzione in funzione della velocità di generazione e attuazione del comando.

La logica PumEx è più semplice e viene usata per ridefinire gli angoli di comando in funzione di nuovi rilevamenti.

In questa fase ho ultimato i calcoli e le logiche decisionali riguardanti il comando di attuazione.

L'ultimo passaggio sta nel definire le grandezze da trasmettere per poter eseguire il comando e attivare le function esterne al modello per l'esecuzione. Queste operazioni vengono eseguite dal blocco 4.2_DD_mgm.

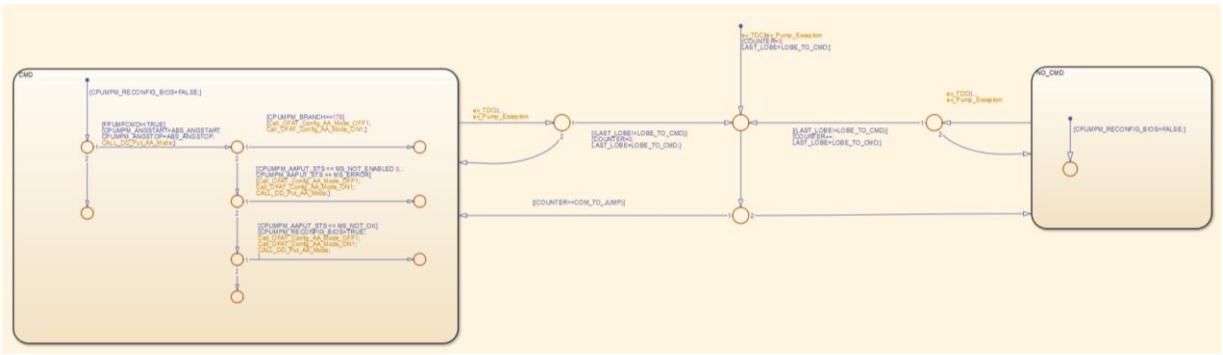


Figura 120 - Logica 4.2_DD_mgm

Questo blocco è una macchina a stati. L'ingresso in uno dei due stati è deciso dalla presenza della necessità di eseguire un "salto" di comando. Quando il comando non viene eseguito perché "da saltare" si entra nello stato di destra, altrimenti si entra nello stato di sinistra.

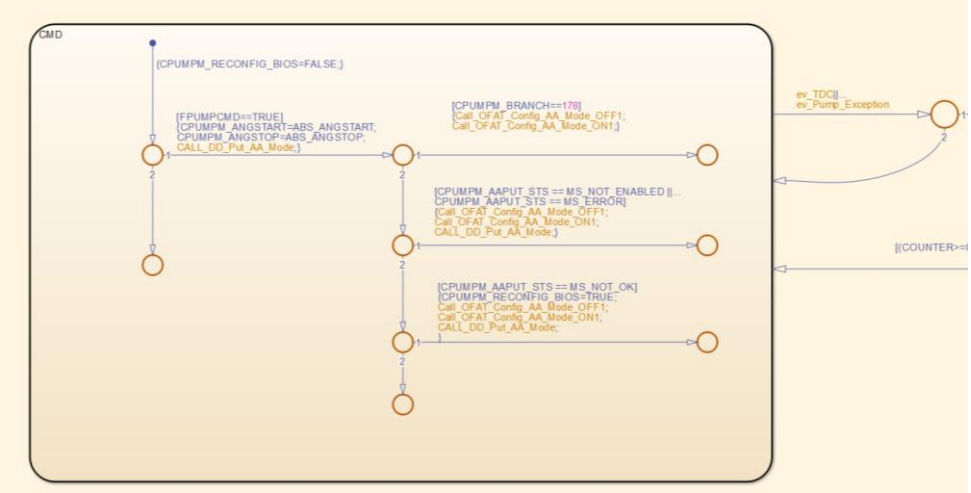


Figura 121 - Dettaglio logica

Se lo stato della pompa è adeguato si passa alla definizione degli angoli di inizio e fine comando e si esegue il comando “Put”. In caso di feedback negativo di azionamento dovuto a uno stato di errore o di altra natura si esegue un comando di spegnimento e riaccensione del Device Driver e successivamente, grazie a questo reset, si riesegue la “Put” di comando.

La descrizione delle logiche di calcolo e di chiamata del modello finiscono con il blocco 4. L’ultimo blocco è quello dedicato alla modellazione necessaria all’interfaccia logica tra questo modello e altre parti.

CPUMPM_DD_LAYER

Il blocco 5.CPUMPM_DD_Layer ha un aspetto particolare rispetto agli altri blocchi. Sono presenti delle logiche al suo interno che non hanno lo scopo di attivare dei calcoli ma di attivare dei blocchi di “comunicazione” che non hanno utilità a livello funzionale, ma sono utili per definire l’interfaccia necessaria con il Device Driver nelle fasi successive di sviluppo del Software. Questi blocchi di “interfaccia” sono quelli con la mascheratura arancione.

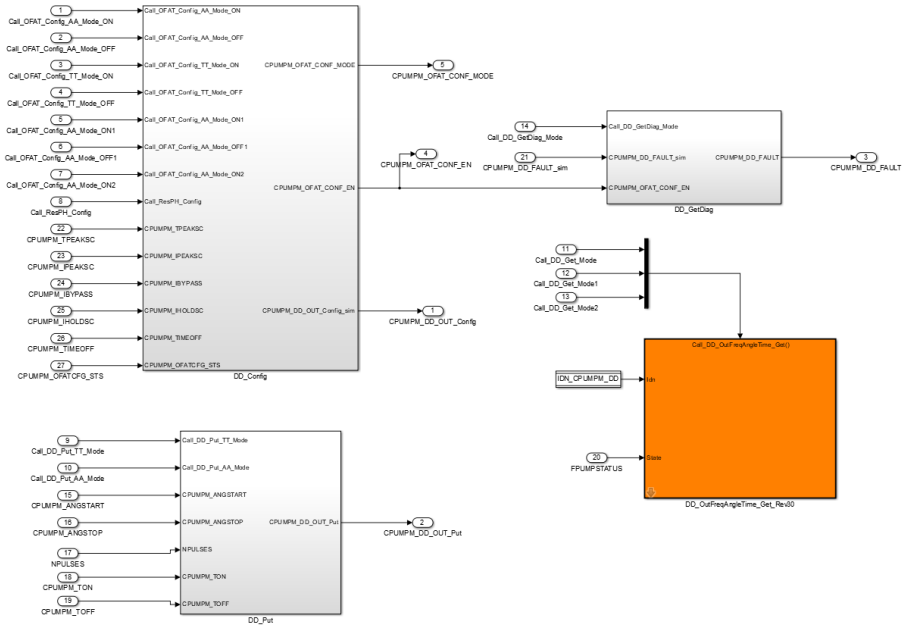


Figura 122 - 5.CPUMPM_DD_Layer

I quattro Subsystem presenti al suo interno hanno scopi di comunicazione differenti.

Il primo blocco (visibile nel livello) DD_OutFreqAngleTime_Get_Rev30 rappresenta la comunicazione del comando Get.

DD_Config: esegue le Call per quanto riguarda la configurazione del Device Driver per quanto riguarda il tipo di configurazione del sistema (a base angolo o base tempo) e definisce i parametri di esecuzione del comando per quanto riguarda tempi, tensioni e correnti.

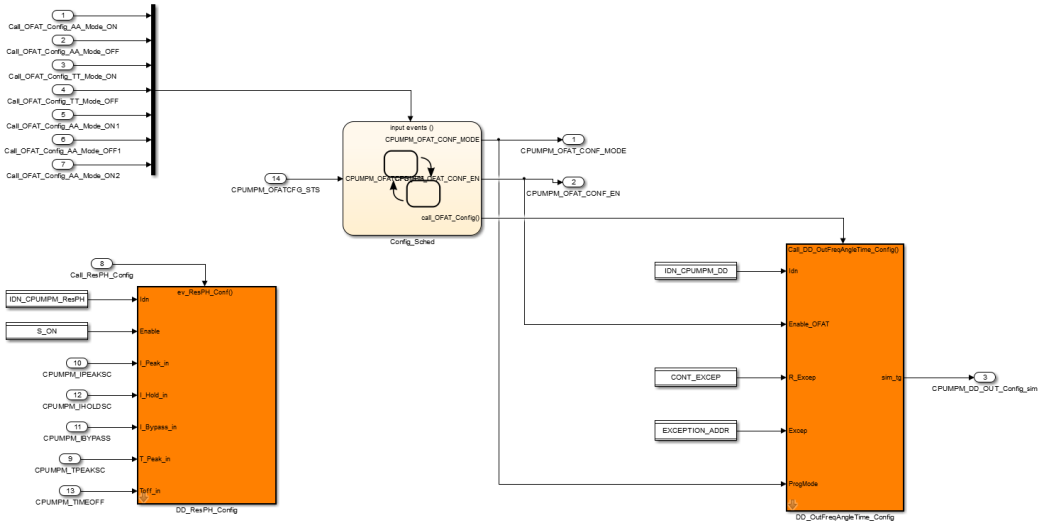


Figura 123 - DD_Config

In contatto a cascata è presente il blocco DD_GetDiag che restituisce la comunicazione di Fault.

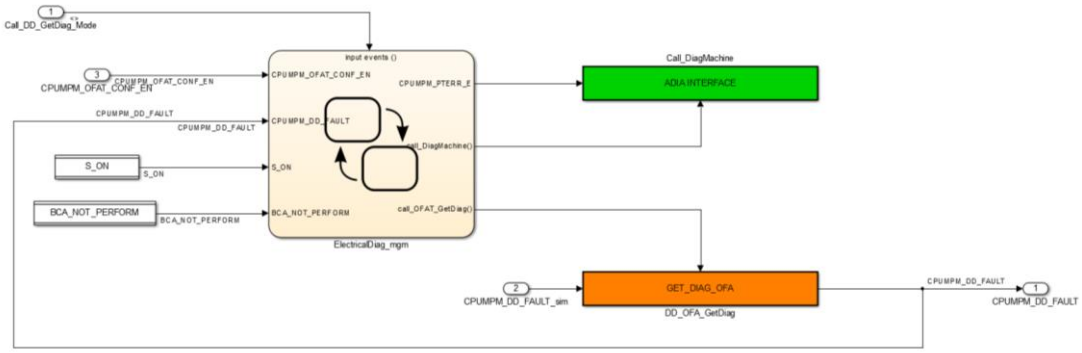


Figura 124 - Dettaglio blocco DD_GetDiag

DD_Put: è l'interfaccia relativa alla comunicazione del momento d'inizio e fine esecuzione del comando, è divisa tra la logica di definizione in base angolo e in base tempo.

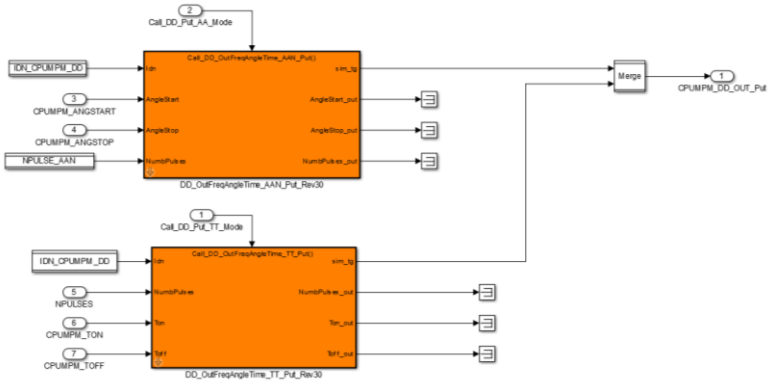


Figura 125 - DD_Put

L'aspetto di questi blocchi è particolare, in quanto tendenzialmente i loro input sono collegati con dei blocchi Terminator che ne interrompono il percorso. Servono in fase successive alla generazione del codice C a dare un'indicazione all'architetto software su quale è il tipo di codice da inserire per avere la corretta interfaccia con altre parti di codice.

Un aspetto importante è che questi blocchi vengono modellati come "Function" e, al loro interno sono presenti gli specifici blocchi di Target Link (verdi) che sono utilizzati per la configurazione e la rigenerazione della funzione C chiamata. In questo modo nel codice generato non è presente la logica integrata alle logiche di calcolo dell'evento ma è solo presente al loro interno la chiamata a una funzione. Il questo modo la function è presente una sola volta nel codice generato e riduce i punti dove dovrà intervenire l'architetto software nelle fasi successive di implementazione.

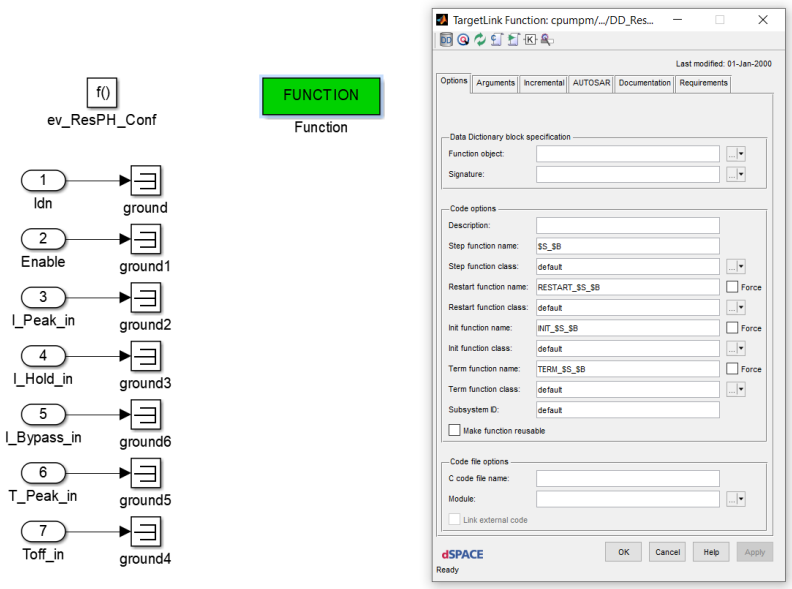


Figura 126 - Dettaglio blocco ResPH_Config con finestra di dialogo di configurazione Function

CPUMPM – IL PERCORSO DI MODIFICA

Come si può notare il modello è decisamente più complesso nelle istruzioni di calcolo rispetto al modello UTET ma entrambi hanno degli aspetti comuni:

- Hanno una logica a stati che determina quando e che calcoli eseguire
- Sono presenti più eventi che alterano lo “stato” del modello
- Vi sono parti di logica in comune tra più eventi

Il percorso di modifica ha essenzialmente seguito lo stesso sviluppo del modello UTET.

La prima fase è stata quella di garantire la presenza di un solo evento per simstep. Questo aspetto è molto importante in quanto nel caso di presenza di eventi concorrenti si ha il rischio di una gestione differente nell’esecuzione dei calcoli in funzione del tipo di modellazione che è stata fatta. Per questo, al fine di garantire l’isofunzionalità del modello, sia la versione originale che la versione modificata hanno avuto una modifica nel blocco CPUMPM_SignalGen.

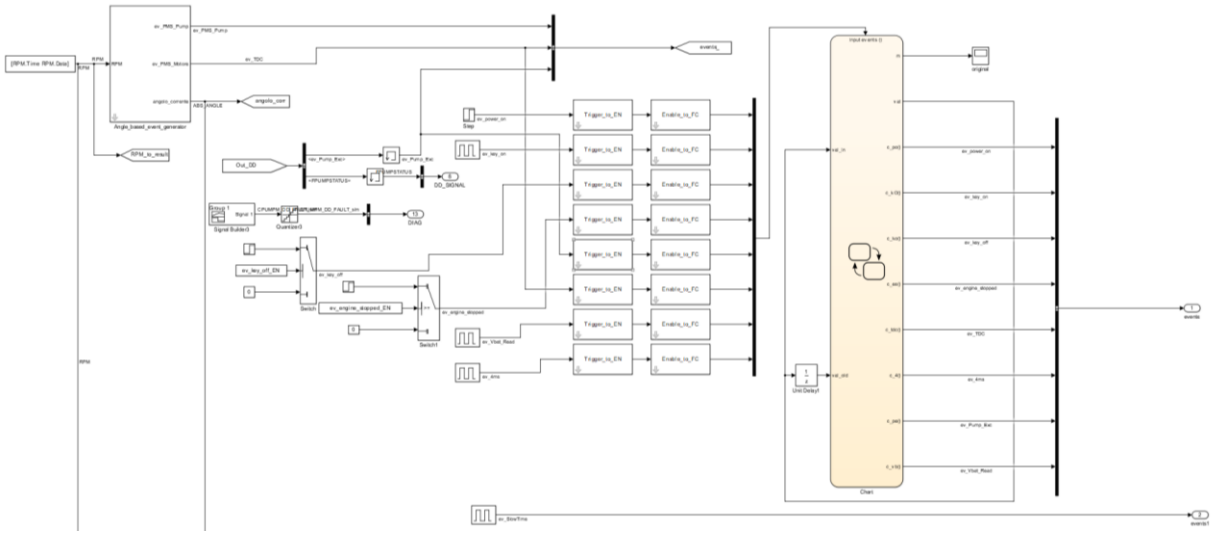


Figura 127 - CPUMPM_SignalGen

Nel blocco è stato aggiunto uno stateflow dove è presente una logica semplice che durante un simstep fa attivare una Function Call relativa al primo evento entrante, tutti gli altri eventi presenti nello stesso simstep vengono invece bloccati e non eseguiti.

L'ordine di precedenza di ingresso nello stateflow è quello utilizzato per tutte le logiche: Eventi on-off, eventi asincroni (secondo loro priorità) eventi temporali (dal meno frequente al più frequente).

L'ordine pertanto degli eventi è:

1. PowerOn
2. EngineStopped
3. KeyOn
4. keyOff
5. PumEx
6. TDC
7. VbatRead
8. 4ms

L'evento SlowTime non è stato inserito in quanto non partecipa attivamente alle logiche di comando ma è presente solo per attivare il blocco DD_GetDiag presente nel subsystem 5.CPUMPM_DD_Layer e quindi non incide nelle logiche della simulazione.

EVOLUZIONE SCHEDULATORE

Il primo step evolutivo è stato quello di modificare la logica presente nel blocco 1.Scheduler integrando i blocchi 1.4 e 1.6 in un unico subsystem. La logica del blocco 1.1 invece non è stata integrata in quanto per la natura evolutiva del tipo di gestione del modello e degli eventi non è utile.

Il secondo step è quello di definire una logica a stati dove non son presenti macrostati. Per poter iniziare il processo è necessario pertanto definire il numero di soto-stati presenti nella logica (ad esempio per la logica di motore sincronizzato sono 3: OpenLoop, Disable e Bypass). Successivamente sono stati individuati tutti gli eventi chiamanti che potevano generare un'uscita da ogni singolo stato e seguendo le logiche possibili si è costruito il percorso verso i possibili stati di arrivo. La logica ottenuta è poco intuitiva e di difficile interpretazione ma cruciale per poter proseguire con il percorso per ottenere una logica multi-albero.

Figura 128 - Processo evolutivo logica a stati

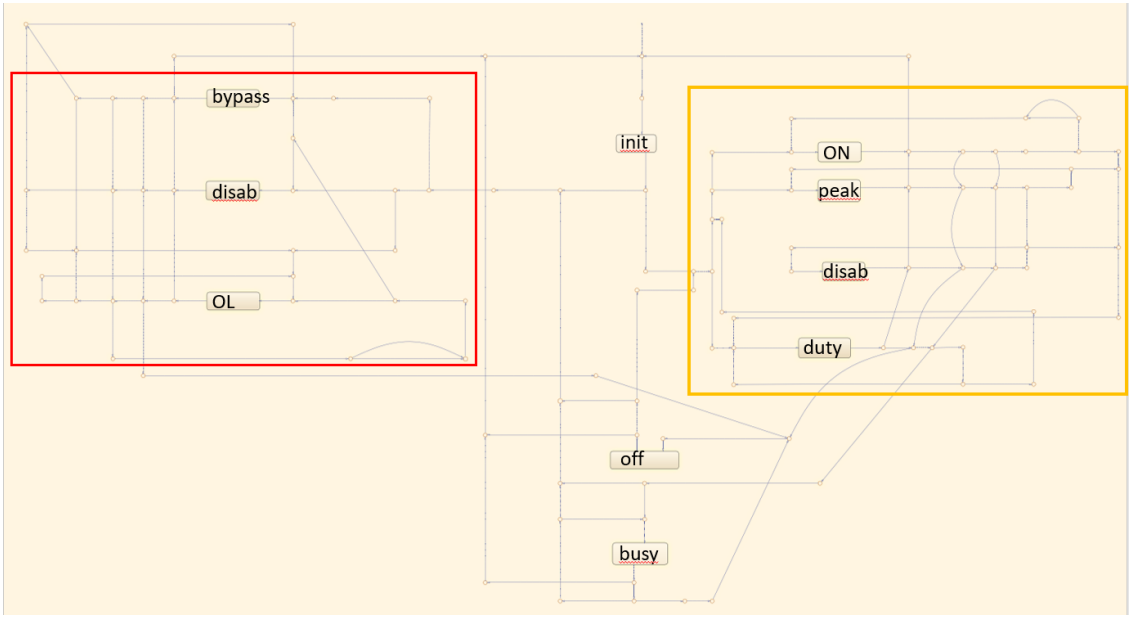


Figura 129 - Dettaglio della logica multistato

Nella logica esposta si possono notare tutti e 10 gli stati del sistema, nella zona rossa sono presenti le logiche riguardanti la condizione di sincronia, in quella arancione sono presenti gli stati relativi alla condizione di Cranking mentre gli stati esterni sono quelli relativi alle condizioni di inizializzazione e blocco. Questa configurazione logica è riconducibile alla condizione iniziale della logica a stati di UTET.

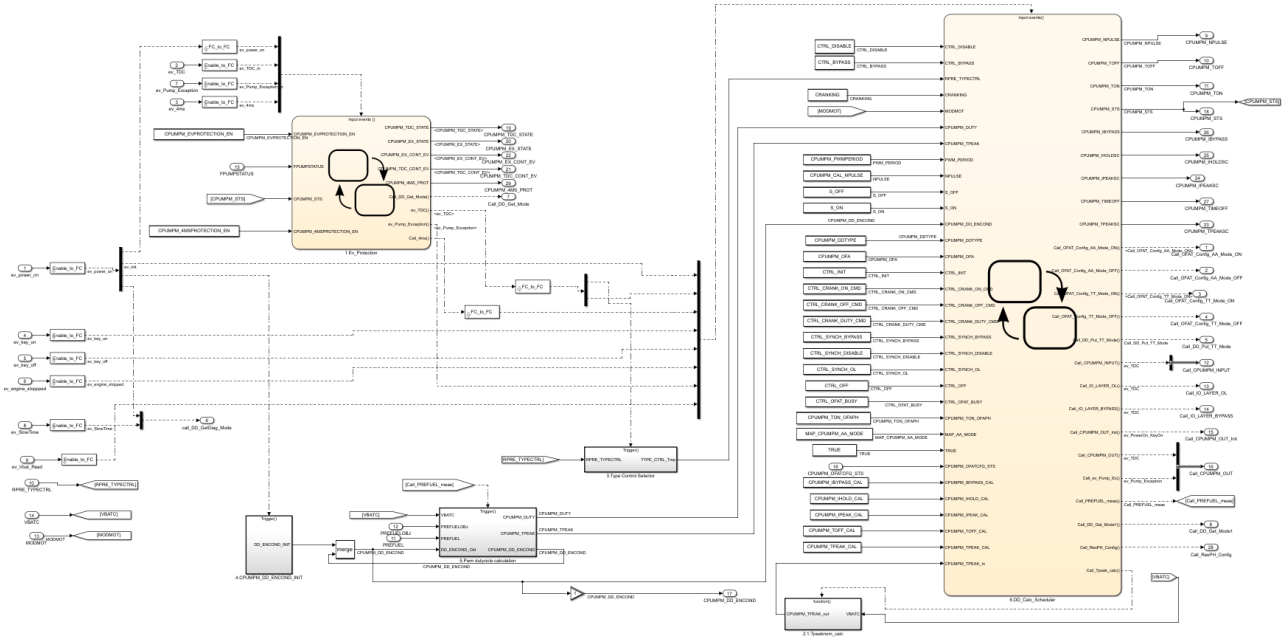


Figura 130 - Blocco 1.Scheduler dopo modifica

Lo step successivo è stato quello di analizzare ogni singolo evento chiamante, definire da quale stato potesse uscire e che percorso logico sviluppasse fino a raggiungere ogni possibile nuovo stato. Per poter mantenere la condizione di “stato del sistema” si è deciso di utilizzare una

variabile chiamata CPUMPM_STS. Una volta definite le logiche sono state analizzate quali avessero parti comuni e sono state concatenate. Si è ottenuta una logica multi-albero all'interno dello stateflow.

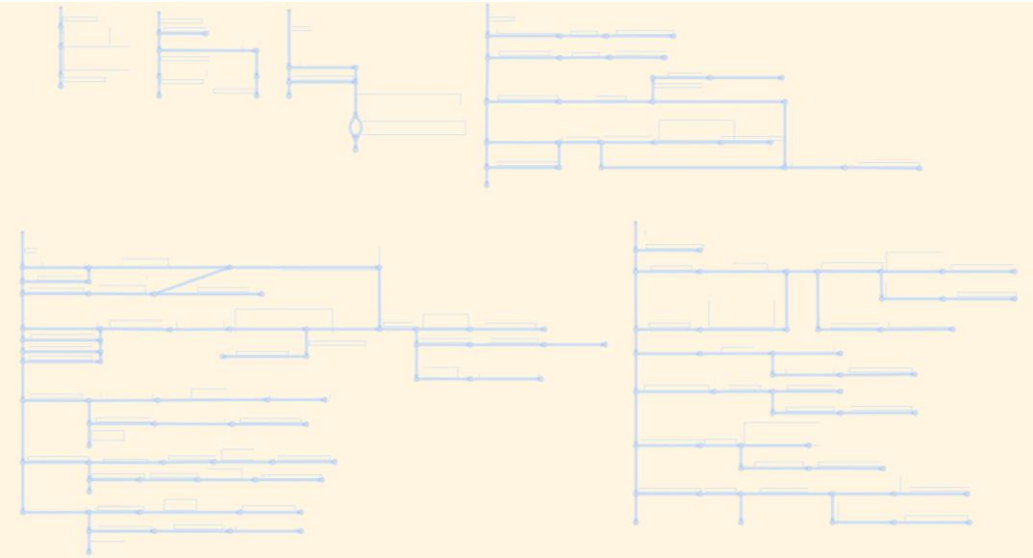


Figura 131 - Vista complessiva della logica multi-albero

Sono presenti in questa fase solo 6 logiche anziché 8 in quanto erano presenti per lo stesso evento logiche esattamente uguali e in questa fase si è deciso di "compattarle" in un unico albero.

In questa fase è emersa un'incongruenza sulla modellazione nella logica a stati, infatti la variabile CPUMPM_STS assumeva lo stesso valore sia per il caso di stato PEAK che per lo stato ON. Nell'immagine seguente è stato evidenziato in maniera grafica quando e con che valori la variabile viene aggiornata.

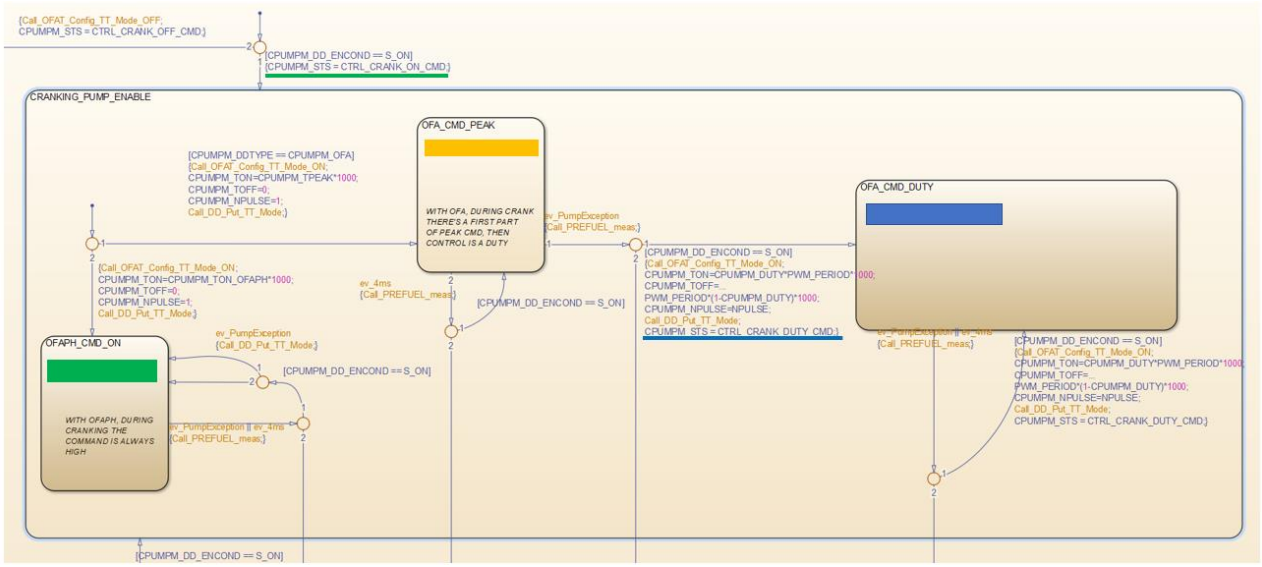


Figura 132 - Dettaglio della logica

Questa cosa non è possibile per la modellazione ad albero in quanto i percorsi sono differenti in uscita da questi stati. A causa di questa incongruenza impossibile da gestire in altro modo è stato deciso che è accettabile la differenza tra modello originale e modello a runnable esclusivamente per la variabile CPUMPM_STS quando il sistema passa per lo stato OFA_CMD_PEAK.

Ottenuta la separazione delle logiche in versione multi-albero si è passati alla separazione del blocco unico che era stato ottenuto. In modo da avere all'interno del blocco di schedulazione i blocchi stateflow dedicati ad ogni singolo evento con all'interno solo la logica ad albero di quell'evento.

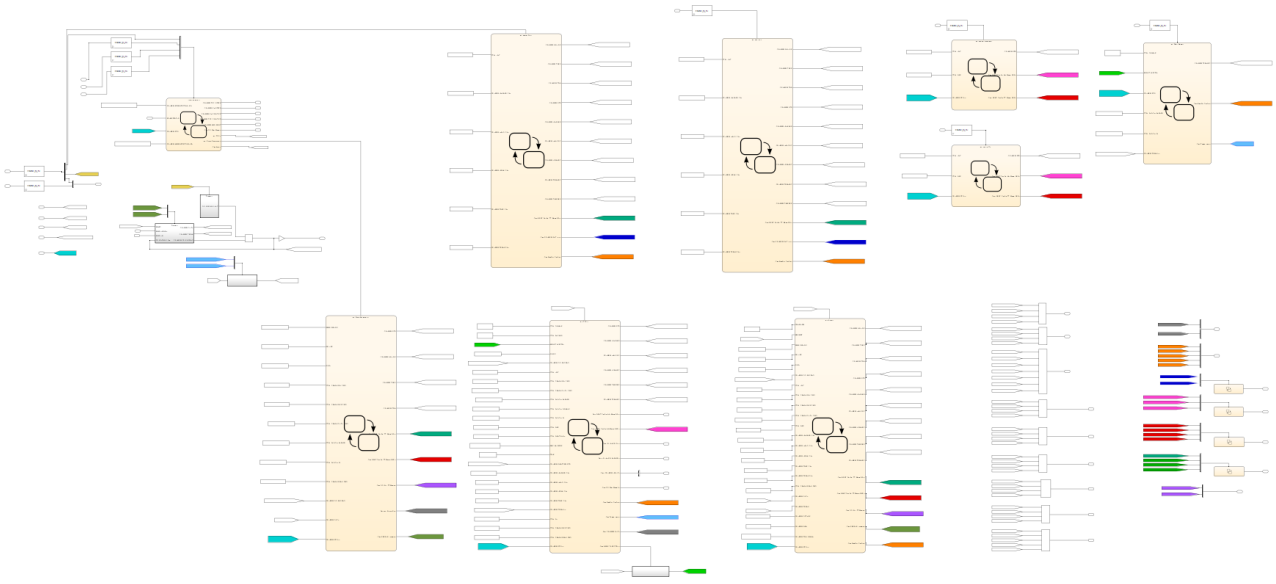


Figura 133 - Schedulatore diviso per eventi

In questa fase si può riconoscere lo schema utilizzato anche in UTET dove a destra è presente la raccolta di variabili da mergiare, i “goto – from” colorati sono invece le Function Call verso blocchi esterni di ogni singolo evento.

EVOLUZIONE BLOCCHI 2, 3, 4

A questo punto si è passati all’analisi degli altri blocchi logici in modo da definire quali eventi attivavano le parti di logica presenti negli altri blocchi. Di seguito si riassume schematicamente quali sono gli eventi che attivano parte della logica presente nei blocchi.

Evento	Blocco 2	Blocco 3	Blocco 4	Blocco 5
PowerOn			V	V
EngineStopped				V
KeyOn				V
keyOff				V
PumEx			V	V
TDC	V	V	V	V
VbatRead				V
4ms				V

Si è deciso di procedere con la modifica del blocco 4 alla stessa maniera, separando tra loro i blocchi con le logiche multi-albero. Per il blocco 4.2 (macchina a stati) è stato necessario definire una logica ad albero in grado di sostituirlo.

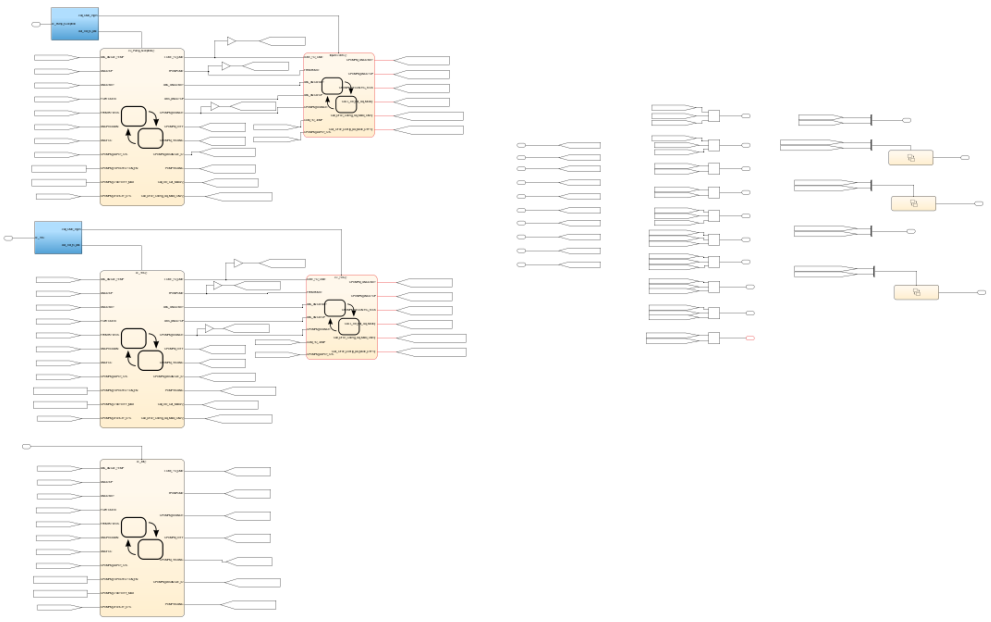


Figura 134 - Blocco 4.DD_Layer

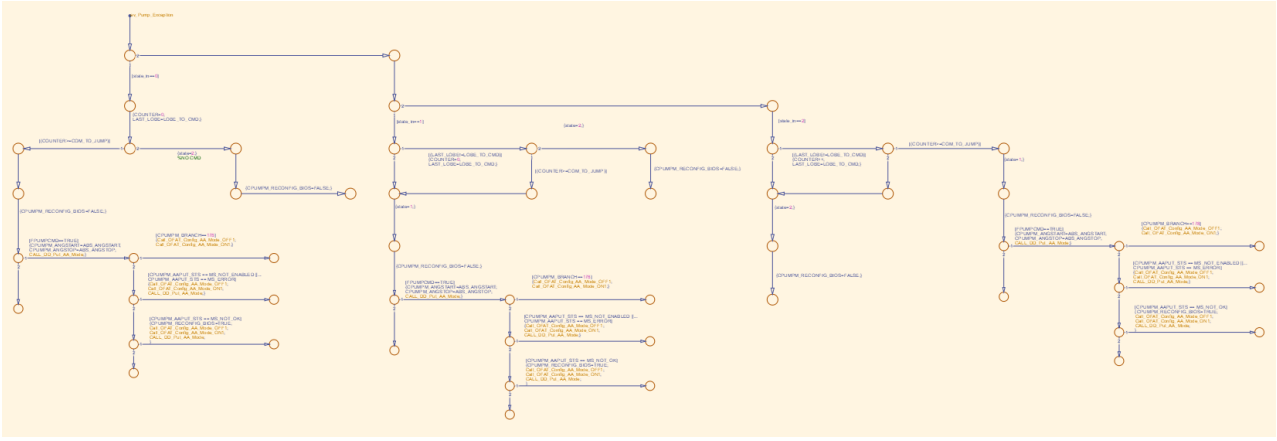


Figura 135 - Logica ad albero blocco 4.2_DD_mgm

Si può notare che in questa fase la logica ad albero non era stata ancora ottimizzata per il blocco 4.2, l'ottimizzazione è stata eseguita in una fase successiva, dopo la divisione del blocco 5.

A questo punto (tolto il blocco 5) sono presenti in ogni blocco logico le logiche divise per evento. Il passaggio naturale, quindi, è quello di portare a livello server tutte le logiche e ottenere una prima divisione grafica delle runnable.

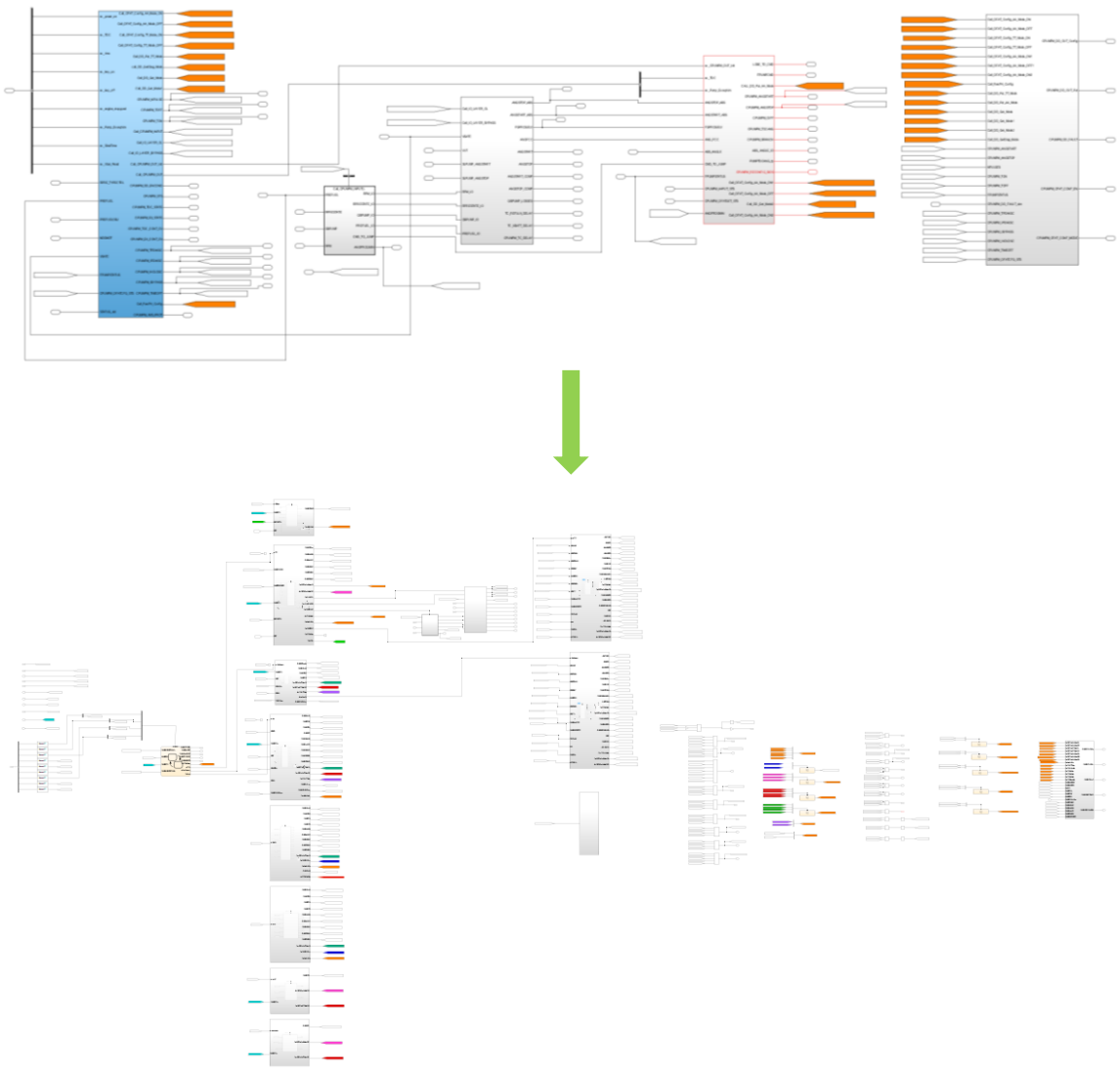


Figura 136 - Evoluzione del livello Server

In questa divisione grafica si possono notare da sinistra verso destra per ogni colonna: gli input, i blocchi relativi allo schedulatore, il blocco 2, blocco 3, i blocchi relativi al blocco 4, grandezze da mergiare in uscita dai blocchi dello schedulatore, raccolta Function Call presenti in uscita dai blocchi di schedulazione, grandezze da mergiare in uscita dal blocco 4, Function Call in uscita dal blocco 4, blocco 5 (non ancora diviso).

Si noti che nella divisione del blocco 4 vi sono 3 subsystem, due con molti input e output mentre il terzo con un solo input. Quel blocco corrisponde alla logica relativa al blocco 4.1 attivata dall'evento PowerOn. Non essendo utile al modello e nemmeno all'interazione con il Device Driver questa parte di logica è stata cancellata.

EVOLUZIONE BLOCCO 5

L'ultimo step evolutivo prima di passare alla modifica del blocco 5 è stata quella di togliere la logica 1.1 ancora presente a monte della divisione, questo perché al suo interno erano presenti delle logiche utili al modello ma non necessarie al fine della generazione del codice. Pertanto, questo blocco non è stato cancellato ma spostato nel subsystem SignalGeneration dove comunicava direttamente al subsystem SimulationResult gli output necessari.

Fino a questo punto della modellazione non erano stati individuati blocchi con logiche di calcolo o esecuzione totalmente condivise, nel blocco 5 questo invece era molto frequente. La divisione del blocco 5 ha quindi subito un processo evolutivo differente. Sono state definite tutte le FUNCTION presenti nel blocco e sono state rese indipendenti tra loro in modo da avere dei blocchi comuni. A questo punto è stata definita la libreria "CPUMPM_Block" dove sono state ordinate tutte le logiche relative ai 4 blocchi presenti all'interno del blocco 5 (DD_Put, DD_GetDiag, DD_Config, DD_ResPH_Config)

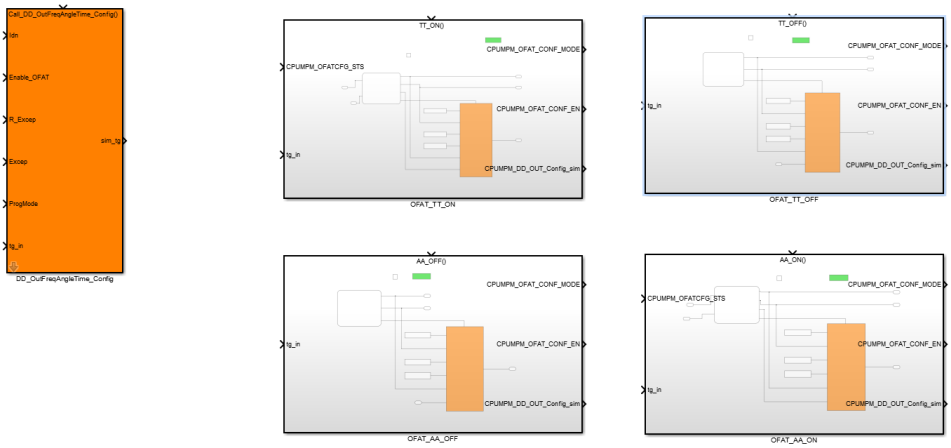


Figura 137 - Blocchi indipendenti DD_Config

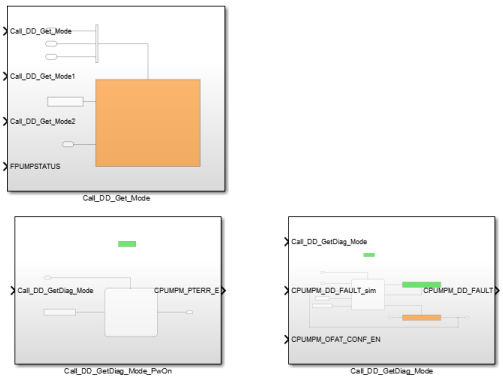


Figura 138 - Blocchi indipendenti DD_GetDiag

Figura 139 - Blocchi indipendenti DD_Put

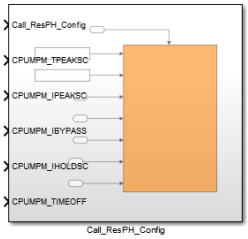


Figura 140 - Blocchi indipendenti DD_ResPH_Config

Utilizzando i blocchi di libreria nelle singole catene di calcolo si è ottenuta la configurazione finale divisa per runnable, senza dover replicare le stesse implementazioni in più parti del modello.

CPUMPMR – Divisione runnable

Il modello CPUMPM diviso per runnable è stato rinominato CPUMPMR. La sua struttura rimane invariata nel livello più alto. Con la presenza dei tre susbsystem.

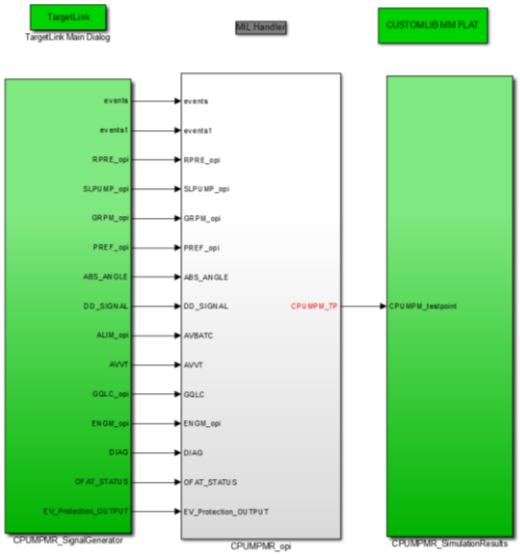


Figura 141 – CPUMPMR

Il subsystem CPUMPM_SignalGeneration ha subito solamente le modifiche al suo interno per garantire che per ogni simstep sia presente un solo evento e l'aggiunta della logica del blocco 1.1 per poter comunicare alcuni output al blocco SimulationResult. Il blocco CPUMPMR_SimulationResult invece è rimasto totalmente invariato.

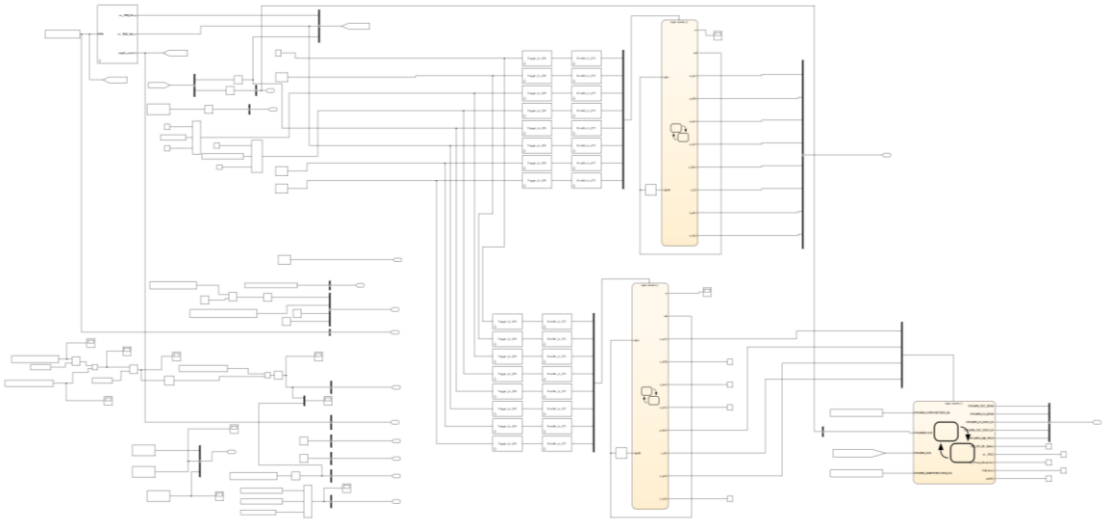


Figura 142 - CPUMPMR_SignalGeneration

Per quanto riguarda invece il livello CPUMPMR_OPI e CPUMPMR_ac sono state aggiunte delle variabili di ricircolo ed eseguiti i collegamenti con la logica 1.1 per poter avere gli stessi output in calcolo.

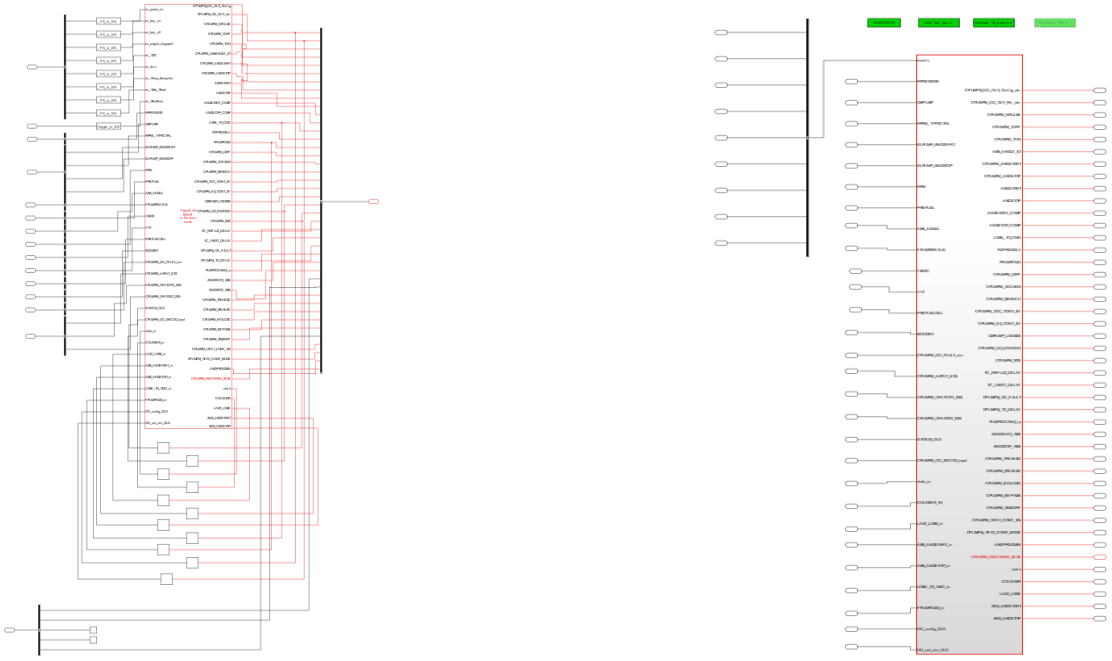
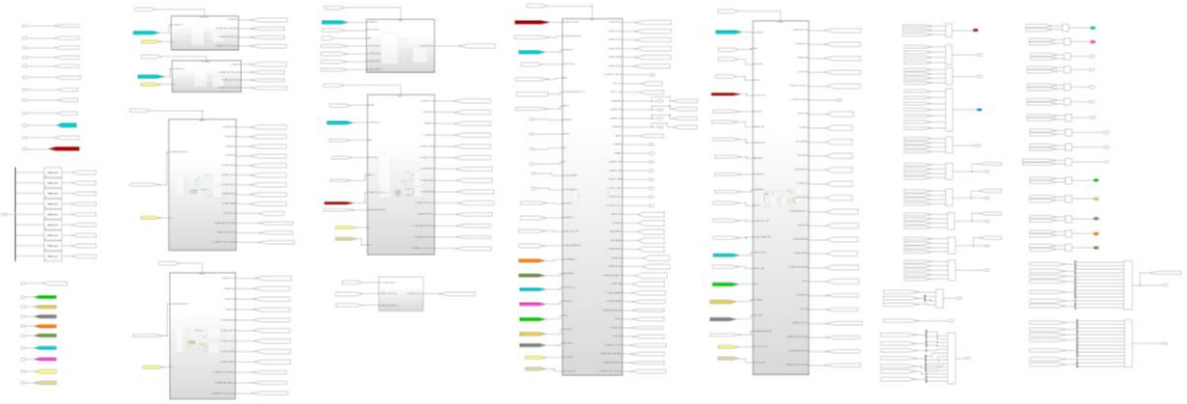


Figura 143 - CPUMPMR_opi e CPUMPMR_ac

Con la modellazione per runnable non si ha più la presenza degli stati, è necessario però mantenere traccia di questa struttura. La soluzione è quella di trasformare il concetto di stato in una variabile che viene condivisa tra le varie runnable. In ogni runnable si riceve l'aggiornamento della variabile e si esegue l'eventuale modifica. Nella terminologia AUTOSAR, questo tipo di variabili sono identificate come *Inter-runnable variables*. In

CPUMPM sono presenti più inter-runnable variables e si possono notare con il ricircolo presente nel livello “opi”.

Il livello Server è stato riorganizzato mettendo a sinistra tutti gli input, al centro i subsystem relativi ad ogni evento e a destra nel primo raggruppamento variabili e Function Call relativi alle logiche del vecchio blocco di schedulazione mentre nell’ultima colonna tutte le variabili aggiornate dal blocco 4 e le chiamate relative. Al contrario delle configurazioni precedenti, i “Goto – From” colorati rappresentano le variabili in ricircolo.



Ogni evento è collegato direttamente al subsystem dedicato nel quale al primo livello è stato inserito il blocco Function Call per poter attivare le logiche della runnable. Di seguito si espongono le strutture delle runnable ottenute nell’ordine di priorità di esecuzione.

POWER_ON

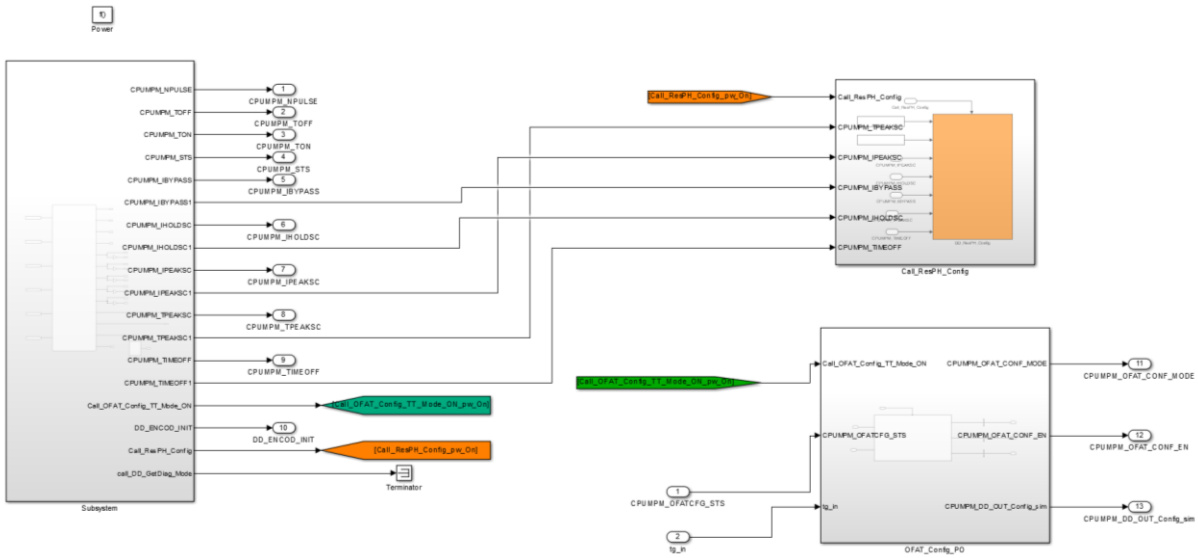


Figura 144 – PowerOn

La logica di PowerOn presenta solamente la logica stateflow di inizializzazione di alcune variabili e le chiamate verso il Device Driver di configurazione. Si fa notare la presenza doppia

delle variabili che entrano nel blocco ResPH_Config, questo è dovuto al fatto di dover imporre come già visto su UTET delle variabili non condivise che entrano nel blocco Merge.

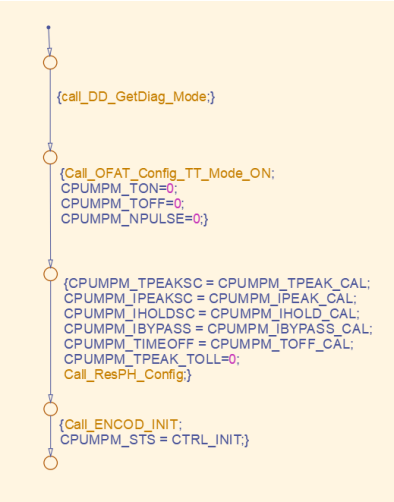


Figura 145 - Logica ad albero PowerOn

ENGINE_STOPPED

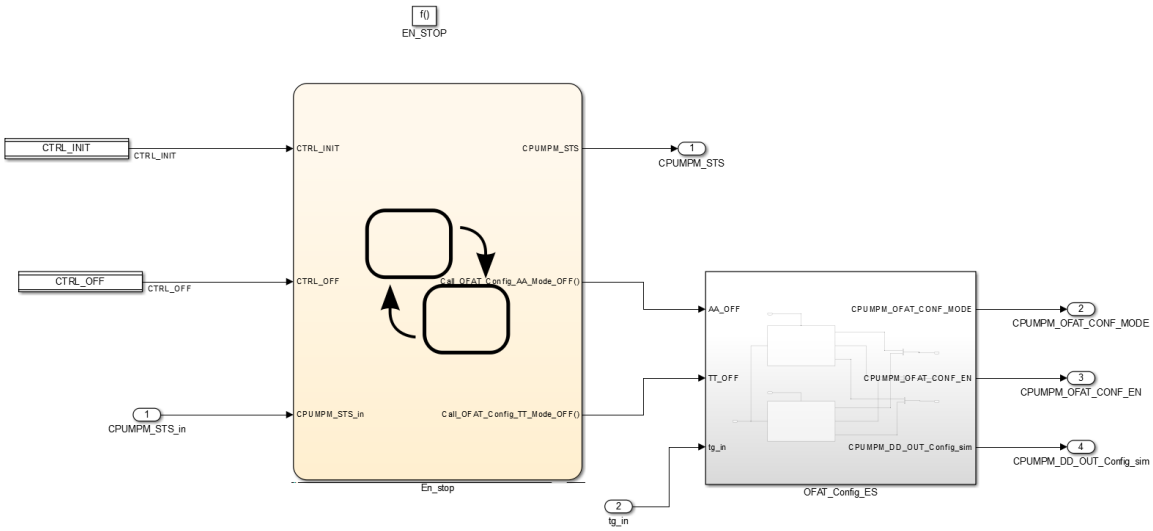


Figura 146 - Logica EngineStopped

La logica è molto simile al caso del PowerOn lo stateflow della logica presente nello schedulatore e i blocchi relativi della logica del blocco 5.

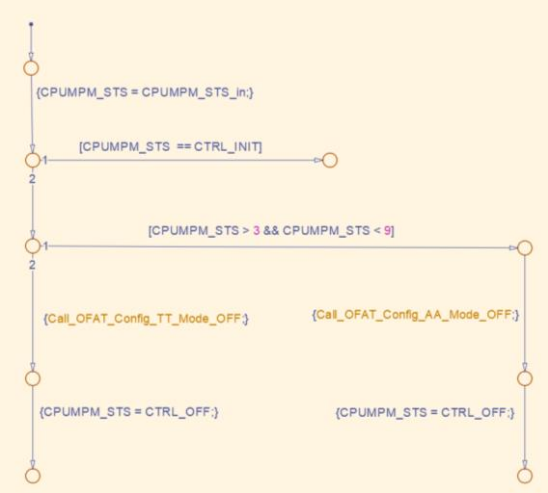


Figura 147 - Logica ad albero EngineStopped

KEYON

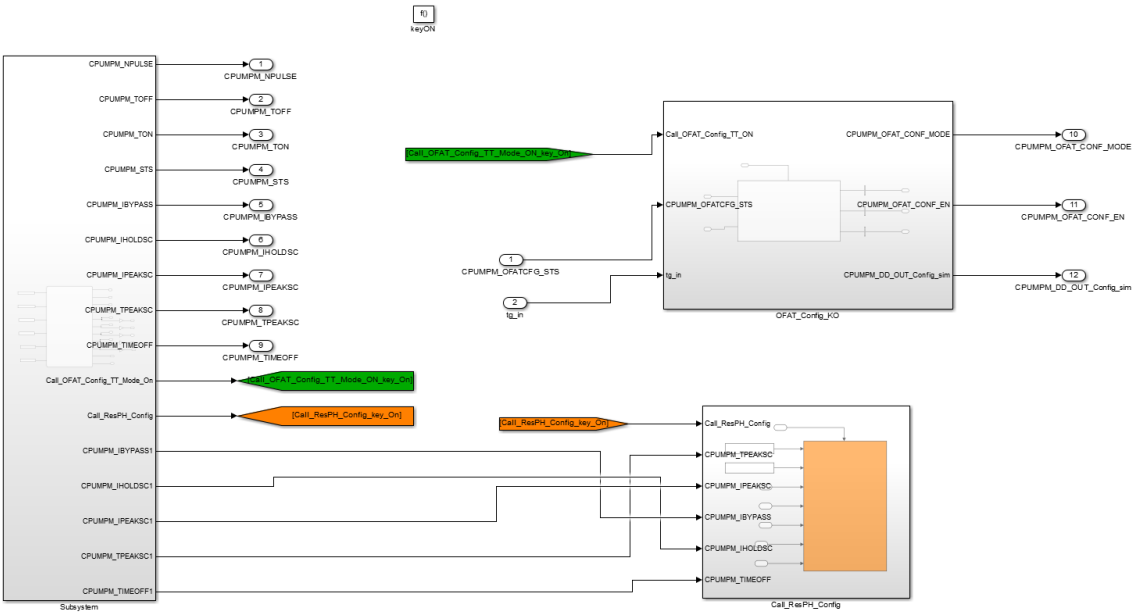


Figura 148 – Logica KeyOn

La logica dell'evento KeyOn è molto simile alla logica usata per il PowerOn in quanto entrambi eseguono l'inizializzazione del sistema, l'unica differenza è presente nell'albero stateflow dove se messo a confronto con quello del PowerOn mancano alcune istruzioni.

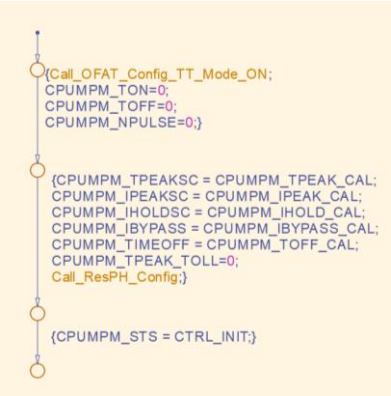


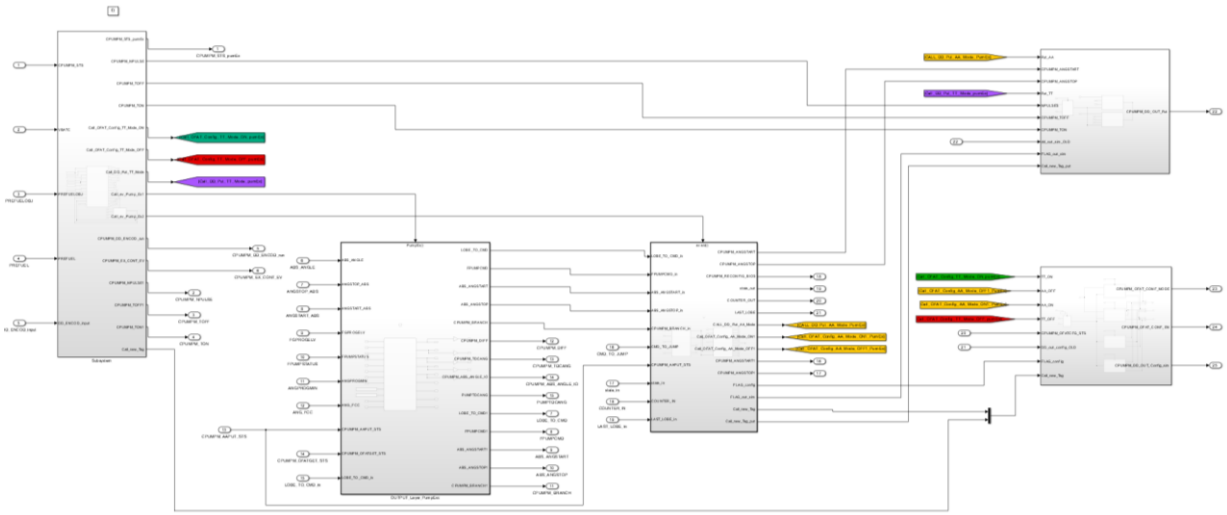
Figura 149 - Logica ad albero KeyOn

KEYOFF

La logica del keyOff è totalmente uguale a quella presente nella runnable EngineStopped

PUMP_EX

La logica del PumpEx è una delle due logiche più articolate presenti nel modello.



Graficamente si può notare da sinistra a destra per ogni colonna i blocchi relativi ai subsystem Scheduler, 4.1Rel_to_Abs, 4.2DD_mgm e i blocchi di comunicazione con il Device Diriver.

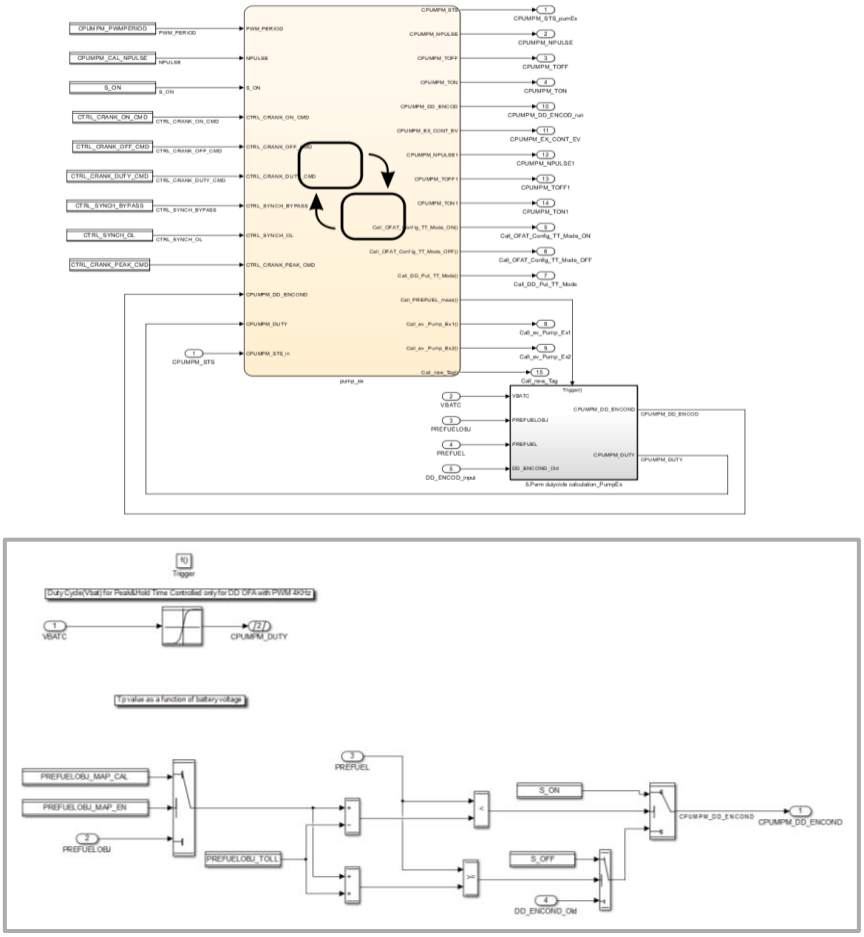


Figura 150- Blocco schedulazione e dettaglio blocco 1.5 PumEx

Il blocco di logica della schedulazione ha presente oltre allo stateflow il blocco 1.5 presente nello schedulatore del modello originale. Questa logica non è stata modificata.

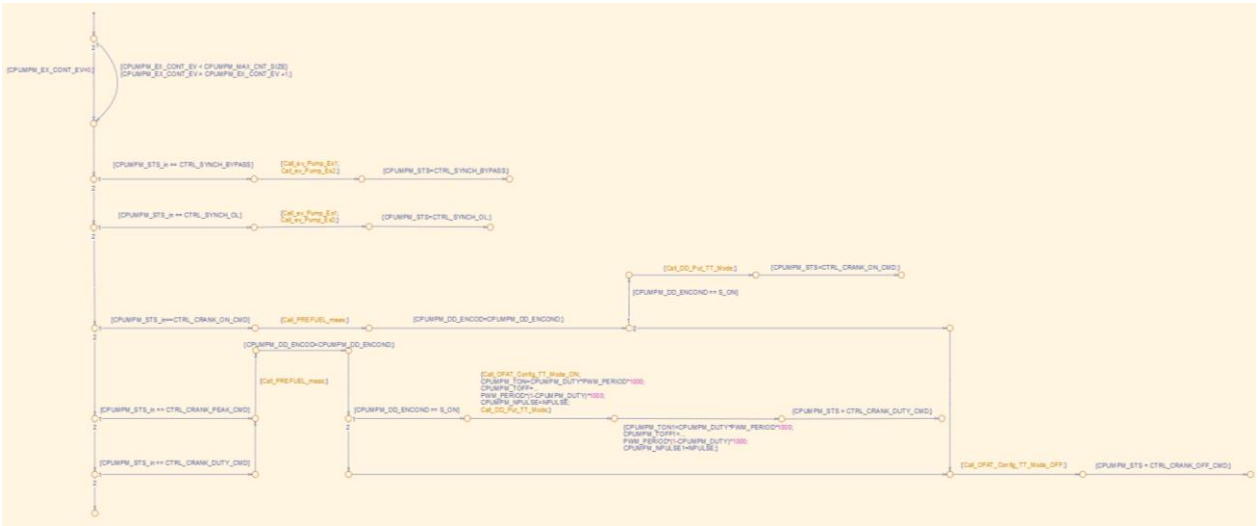


Figura 151 - Logica albero PumEx

Per quanto riguarda il blocco 4.1Rel_To_Abs è stato solamente isolato l'albero logico riguardante l'evento PumEx.

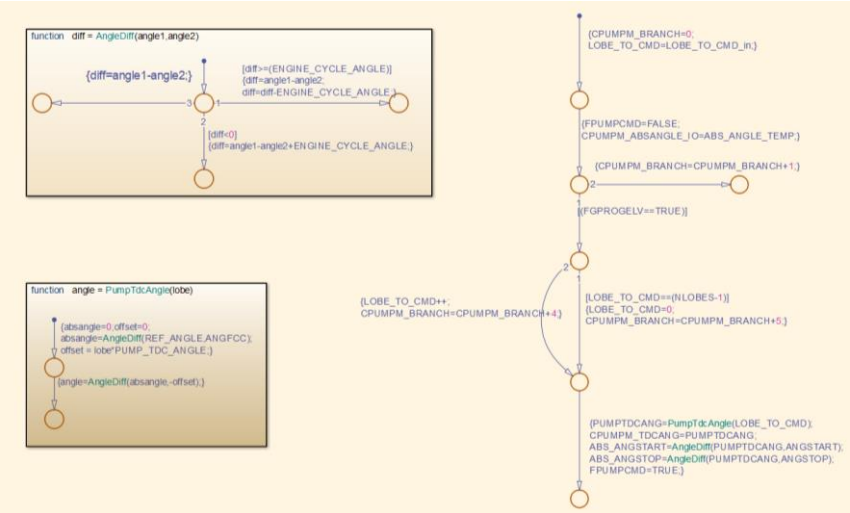


Figura 152 - Logica ad albero 4.1Rel_To_Abs PumEx

Il blocco 4.2 precedentemente esposto in una versione ad albero poco compatta è stato ridefinito.

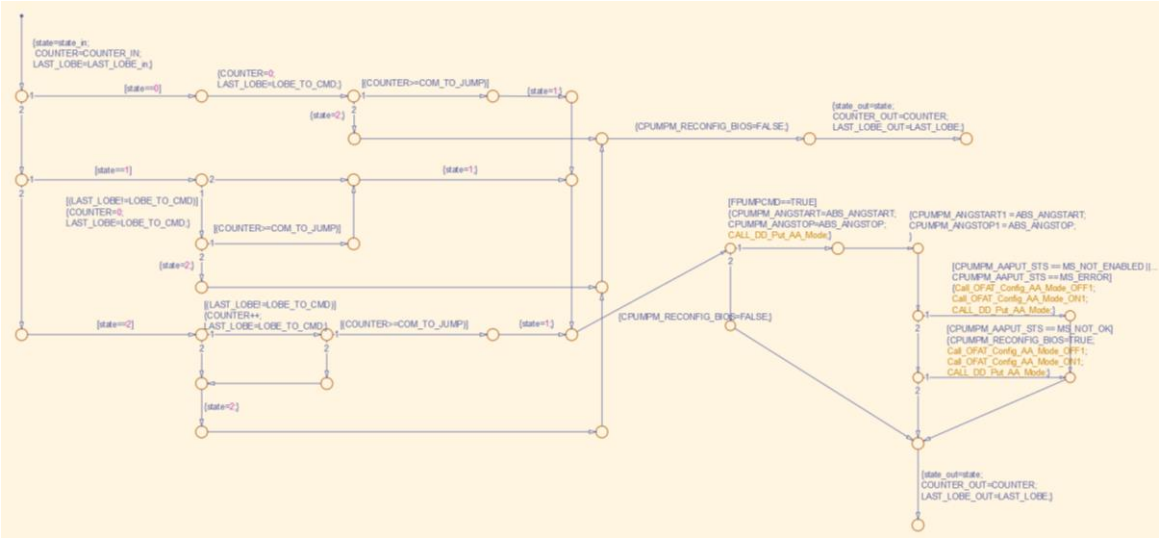


Figura 153 - Logica ad albero 4.2DD_mgm PumEx

Le logiche relative al blocco 5 sono state aggiornate inserendo a monte delle logiche comuni dei blocchi di verifica inerenti a quante volte il comando veniva chiamato nella runnable. Questo accorgimento è stato introdotto solamente per poter ottenere un modello isofunzionale in quanto le chiamate delle "Put" incidono sull'input PumEx, pertanto era necessario che la logica di trigger avesse le stesse dinamiche del modello originale. Nella pratica, l'azione essendo collegata a un blocco di interfaccia con il Device Driver non ha la stessa funzionalità. Si può dire pertanto che questa parte di logica serva solamente al corretto funzionamento del

modello Simulink. Una volta verificata l'isofunzionalità è stato possibile togliere questa parte di logica per affinare i rilevamenti delle metriche, ma questo verrà spiegato successivamente.

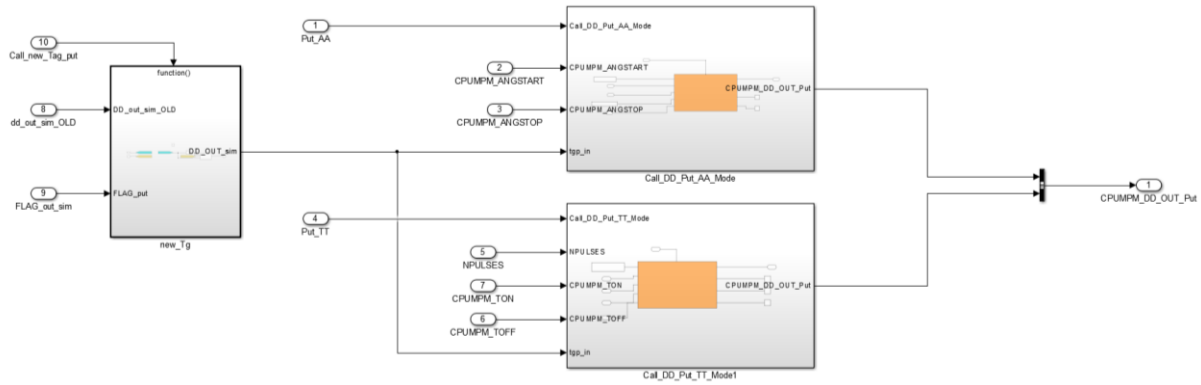


Figura 154 - Blocco Device Driver PumEx

TDC

Il blocco TDC è quello con la logica più complessa e articolata del modello. Da sinistra a destra sono presenti i seguenti blocchi: 1.Scheduler, 2.Input_Layer, 3.IO_Layer, 4.1Rel_To_Abs, 4.2DD_mgm, Blocchi di comunicazione con il Device Driver.

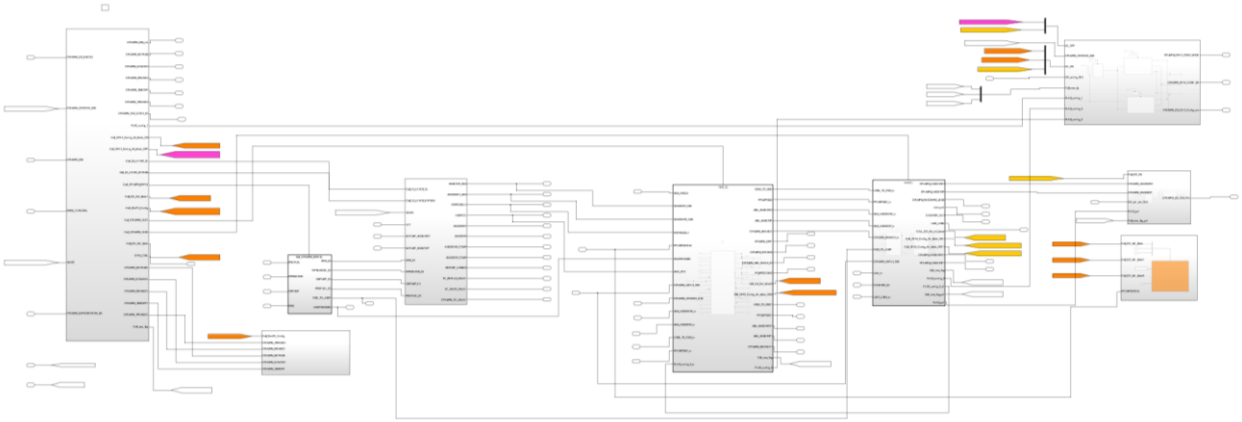


Figura 155 - Logica TDC

La logica dello Scheduler è abbastanza complessa, ha quasi tutti gli stati possibili presenti al suo interno.

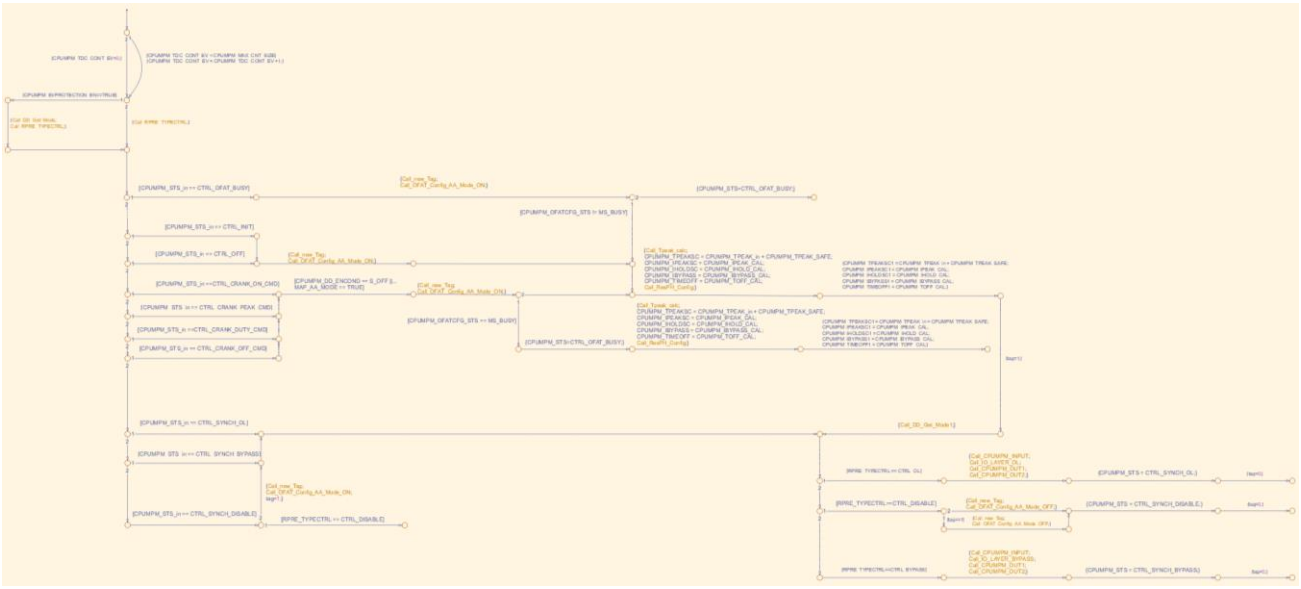


Figura 156- Logica Scheduler TDC

I blocchi 2 e 3 non sono stati cambiati e sono uguali alla versione originale.

Come per il caso dell'evento PumEx il blocco 4.1 è stato ridefinito isolando solamente la logica relativa all'evento TDC.

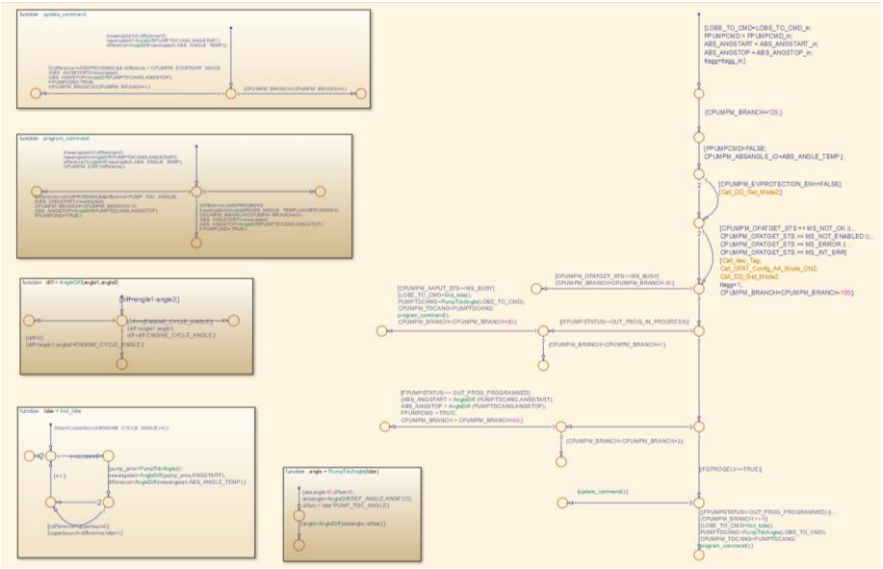


Figura 157 - Logica 4.1Rel_To_Abs TDC

Il blocco 4.2DD_mgm invece è stato ridefinito nella logica come per il caso dell'evento PumEx

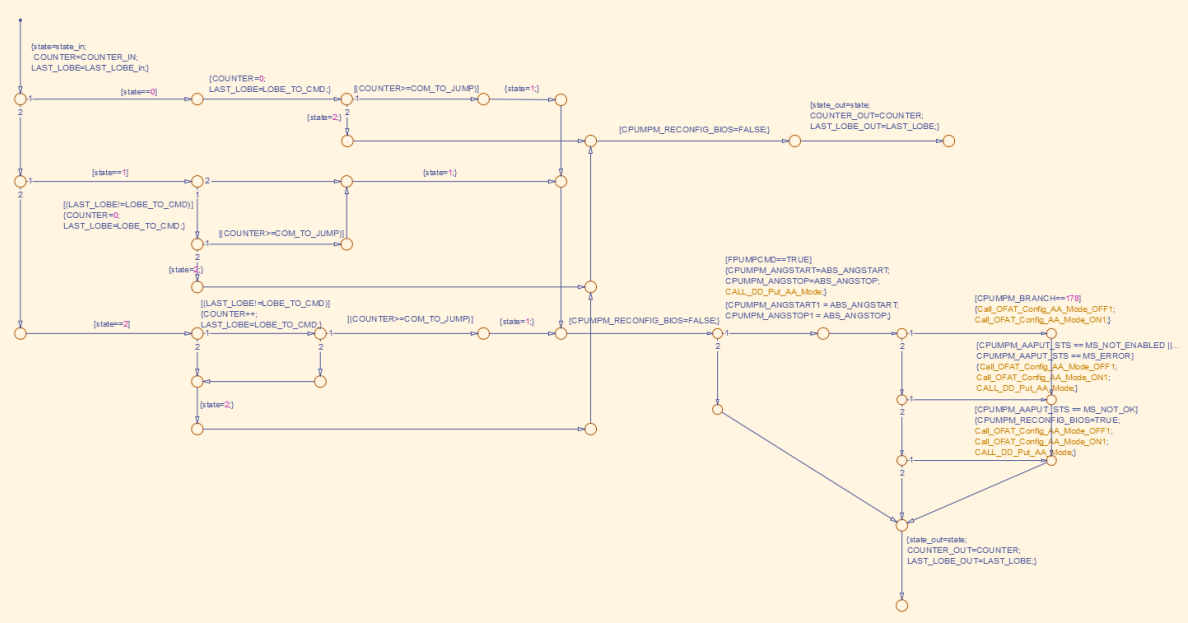


Figura 158 - logica 4.2DD_mgm TDC

Anche in questo caso avendo più chiamate della funzione OFAT_Config e Put_Mode che fan parte delle chiamate verso il Device Driver ed attivano parte di logica nella parte del SimulationResult che partecipa attivamente alla generazione degli input, è stato necessario inserire il contatore che rendesse possibile l'aggiornamento delle logiche.

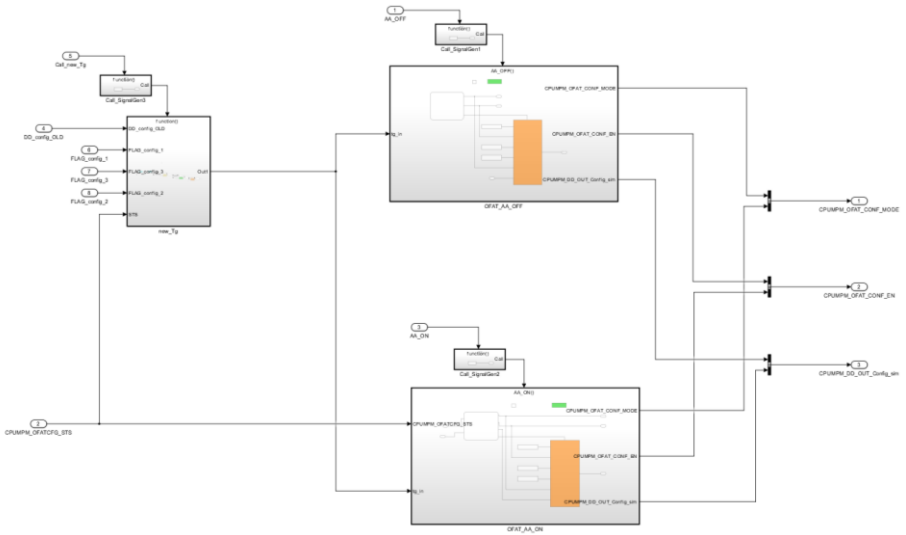


Figura 159 - Logica Device Driver 5.1 TDC

VBATT_READ

La logica relativa all'aggiornamento della tensione della batteria viene attivata ogni 12ms, non è presente una logica complessa, infatti è determinata solo dalla presenza dello stateflow e dal blocco di dialogo con il Device Driver.

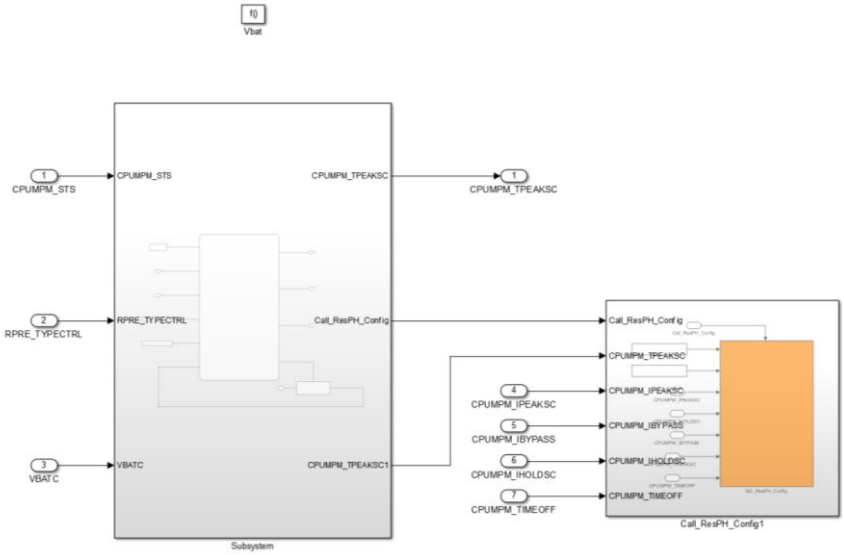


Figura 160 - Logica VbatRead

Come nel modello originale sono presenti le logiche di aggiornamento dei parametri relativi all'attuazione.

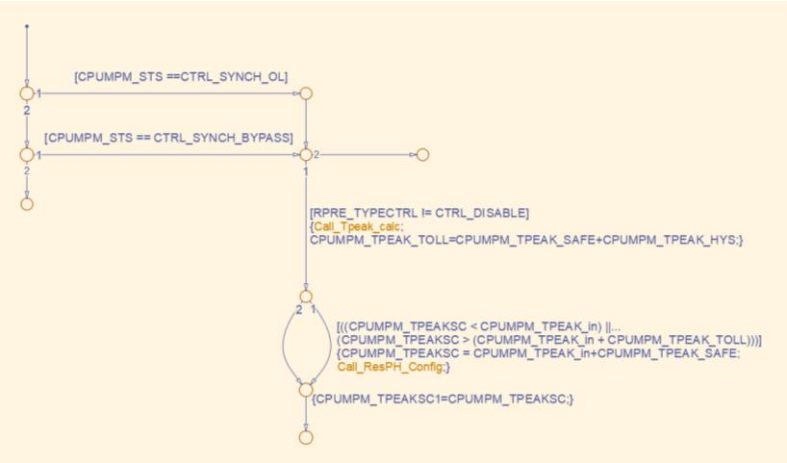


Figura 161 - Logica albero VbatRead

4MS

La logica del 4ms è legata principalmente all'esecuzione dei comandi relativi alla gestione del comando in fase di Cranking.

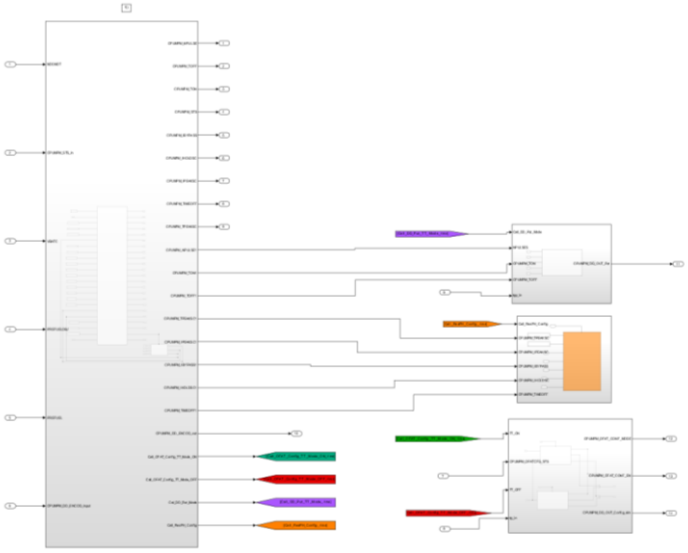


Figura 162 - Logica 4ms

Nella logica stateflow sono presenti solo gli stati inerenti alla condizione di Cranking e quelle di inizializzazione.

Figura 163 - Logica albero 4ms

Al contrario del caso dei blocchi TDC e PumEx non sono presenti doppie chiamate degli eventi pertanto non è stato necessario inserire il blocco di verifica di chiamata già eseguita.

SLOWTIME

L'evento SlowTime come già detto non ha nessuna interazione all'interno del modello, è solamente collegato a un blocco di dialogo con il Device Driver.

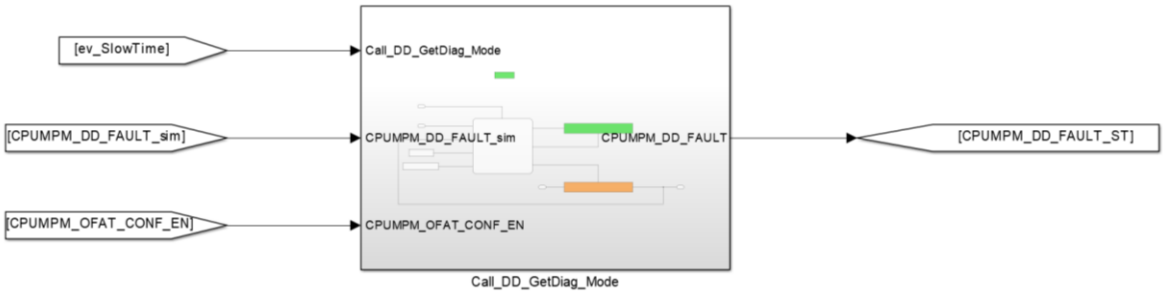


Figura 164 - logica SlowTime

CPUMPMR – Analisi comparativa

ISOFUNZIONALITÀ

È stata eseguita la verifica dell'isofunzionalità del modello diviso per runnable, sono stati individuati i canali più importanti per la verifica della corretta esecuzione del modello:

- CPUMPM_EX_CONT_EV = segnale relativo al Counter degli eventi di esecuzione pompate
- CPUMPM_ANGSTART = segnale degli angoli di start calcolati
- CPUMPM_ANGSTOP = segnale degli angoli calcolati per la fine del comando
- LOBE_TO_CMD = segnale che aggiorna su quale lobo della camma va eseguito il comando
- CPUMPM_STS = segnale che descrive lo stato del sistema

In questo modo si possono monitorare se le esecuzioni sono le stesse e con le stesse caratteristiche e se l'aggiornamento dello stato del sistema è uguale.

Grazie allo script comparativo si può notare che non vi sono differenze tra i due modelli e quindi di possono definire come *isofunzionali*.

Tutti i grafici fanno riferimento allo scenario del Test Pattern 1, non vengono riportati i grafici degli altri scenari per motivi di sintesi tranne che per il segnale CPUMPM_STS dove verranno evidenziate le incongruenze dovute all'aggiunta del valore logico all'interno della logica di Cranking.

CPUMPM_EX_CONT_EV

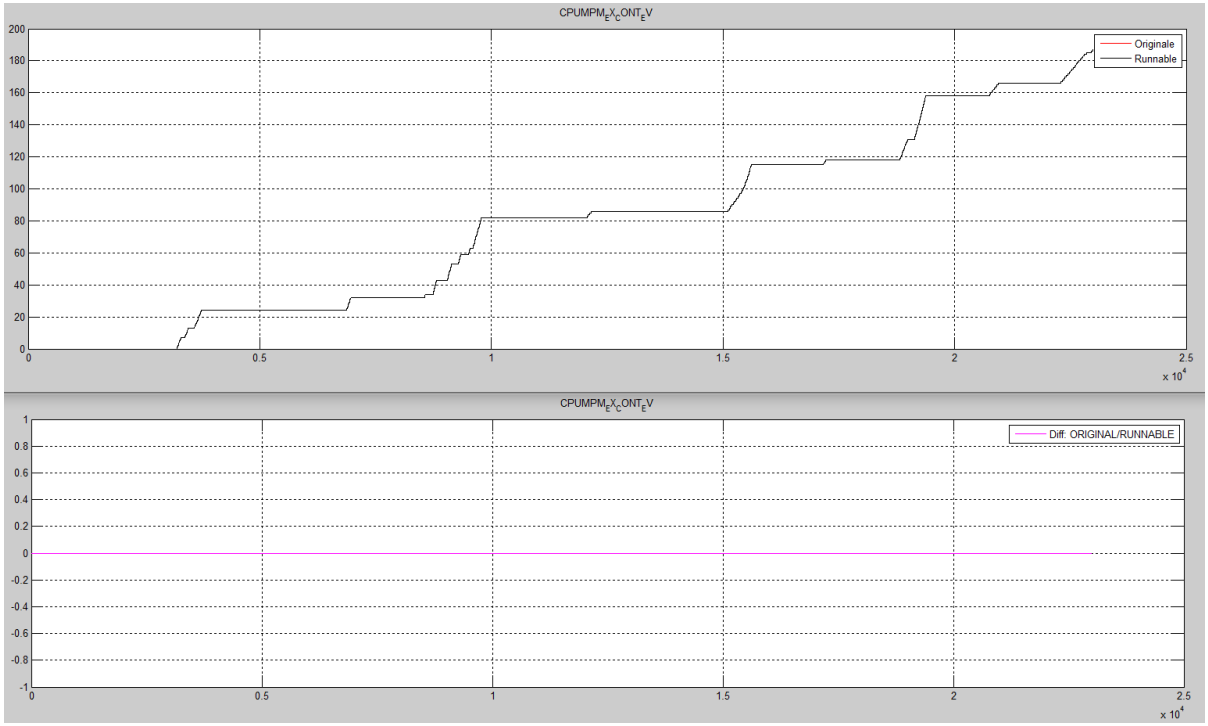


Figura 165 - Confronto CPUMPM_EX_CONT_EV

CPUMPM_ANGSTART

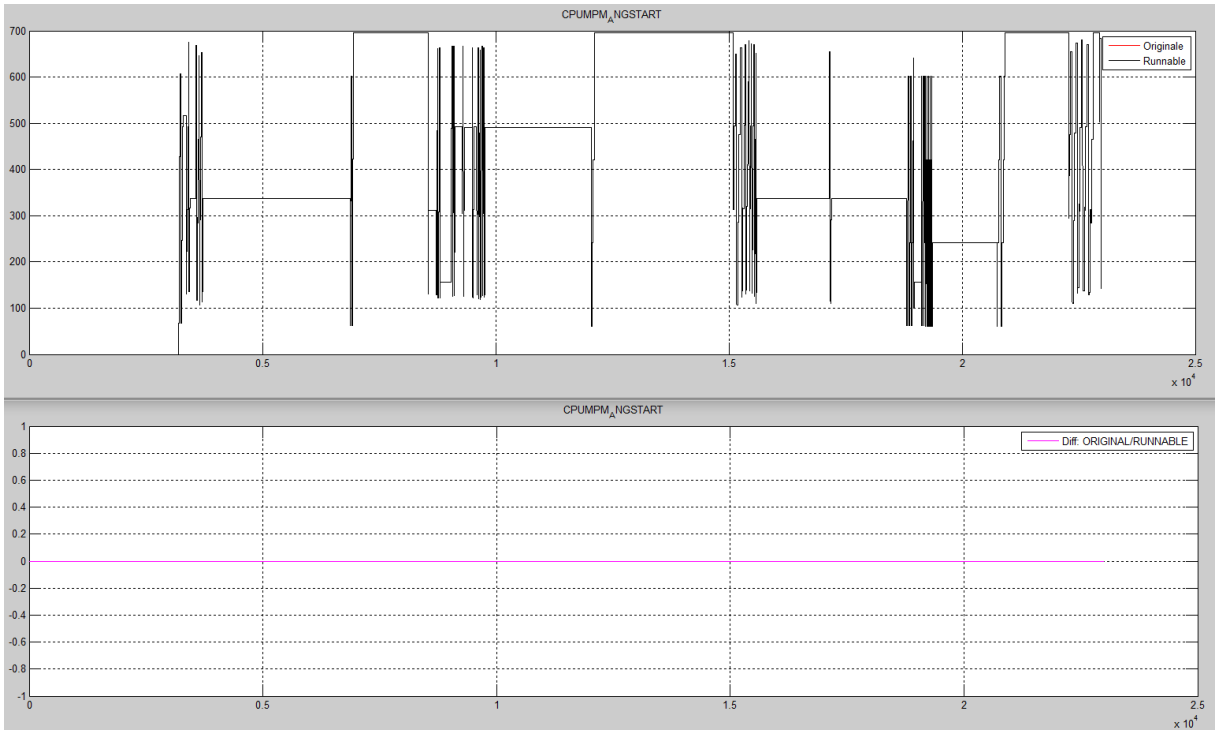


Figura 166 - Confronto CPUMPM_ANGSTART

CPUMPM_ANGSTOP

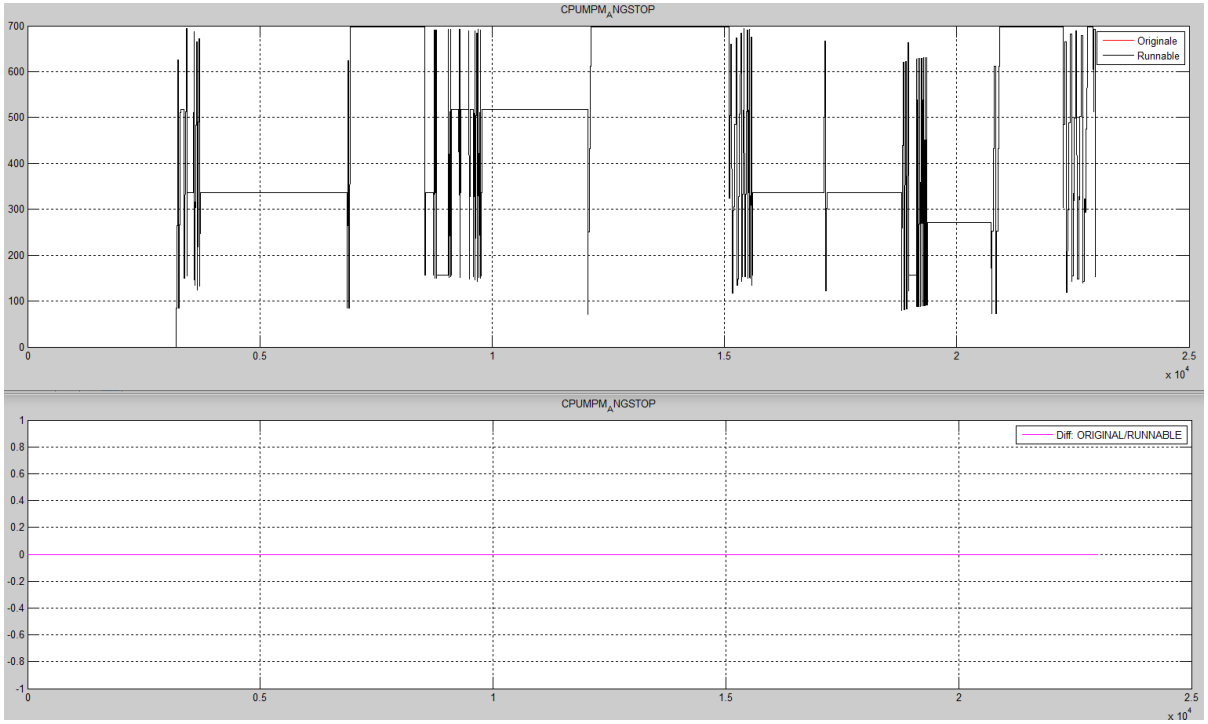


Figura 167 - Confronto CPUMPM_ANGSTOP

LOBE_TO_CMD

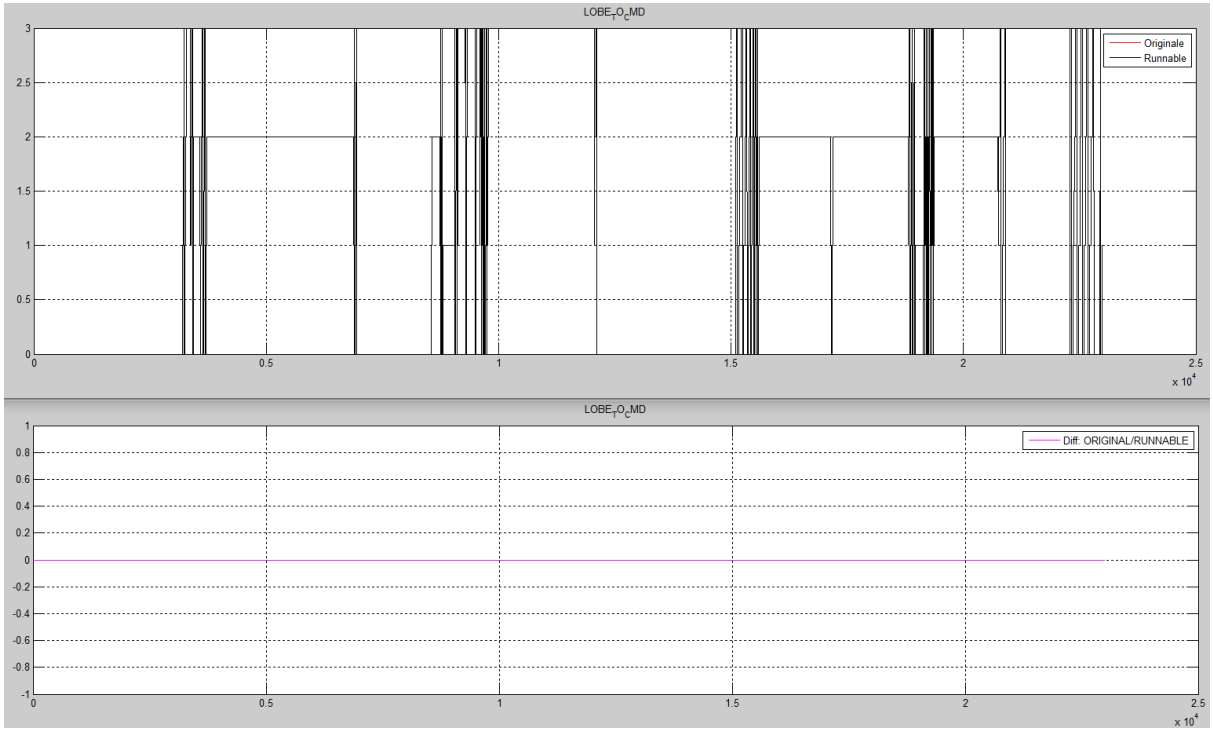


Figura 168 - Confronto LOBE_TO_CMD

CPUMPM_STS

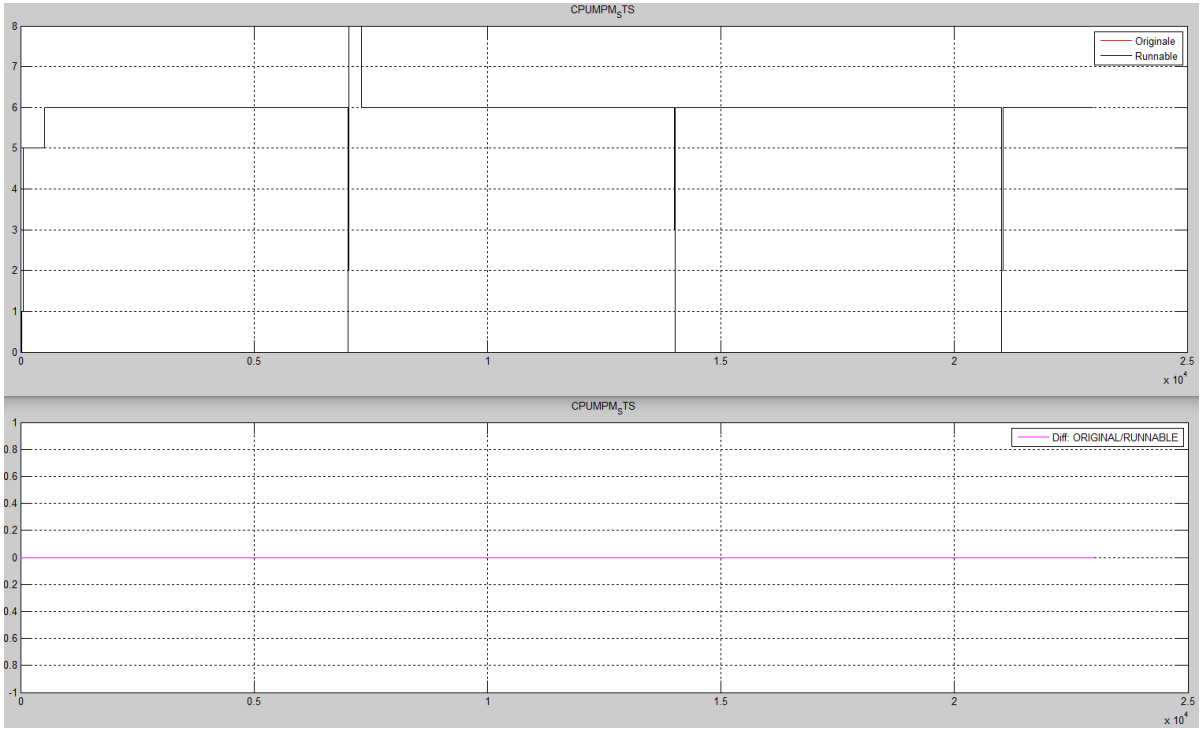


Figura 169 - Confronto CPUMPM_STS (TP_1)

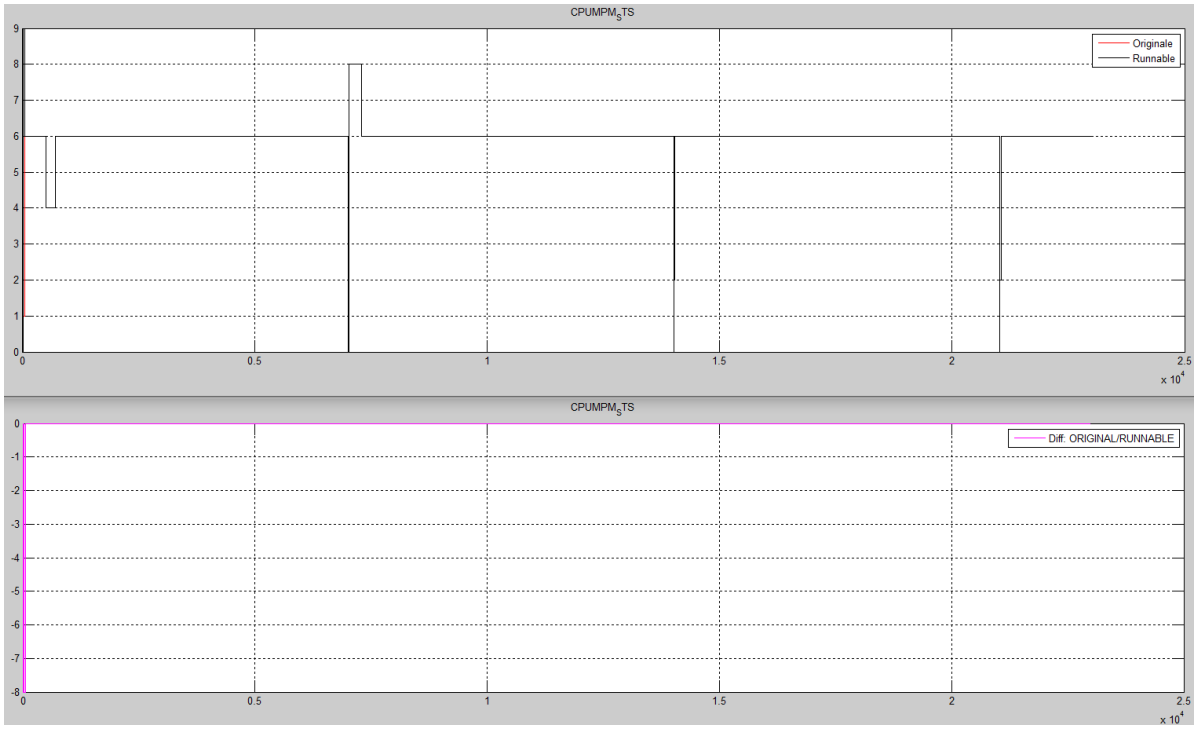


Figura 170 - Confronto CPUMPM_STS (TP_5)

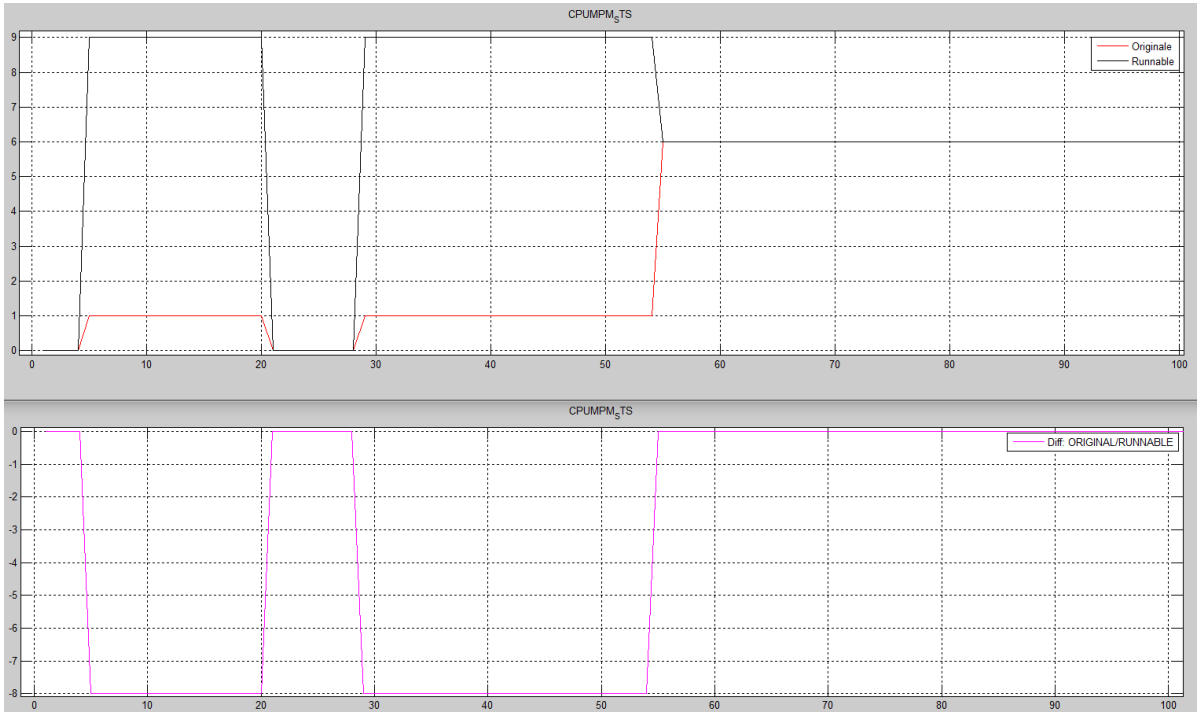


Figura 171 - Dettaglio incongruenza CPUMPM_STS stati e runnable

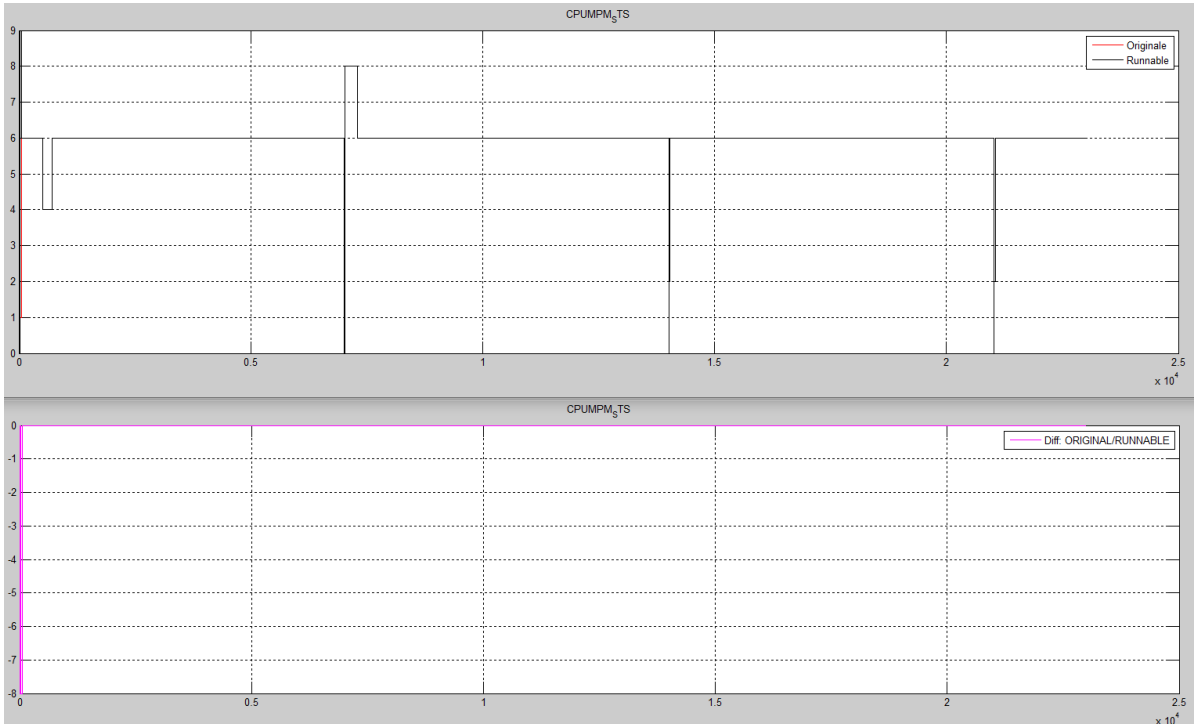


Figura 172 - Confronto CPUMPM_STS (TP_6)

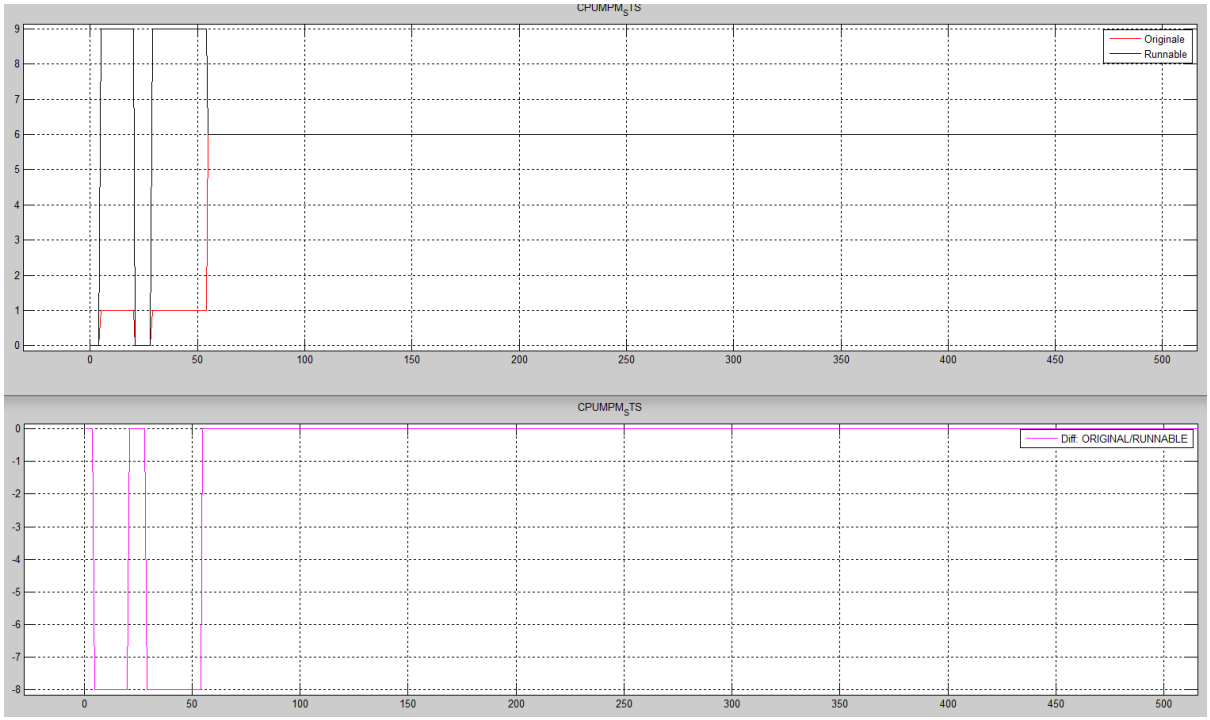


Figura 173 - Dettaglio incongruenza CPUMPM_STS stati e runnable

IL COVERAGE

Il Coverage che è accessorio alla verifica dell'isofunzionalità, viene usato per garantire che non vi siano possibili parti di logica non testate che possono quindi generare differenti output indesiderati. Preventivamente è stato eseguito il Coverage del modello originale riscontrando una non completa copertura da parte dei Test Pattern per tutte le logiche presenti.

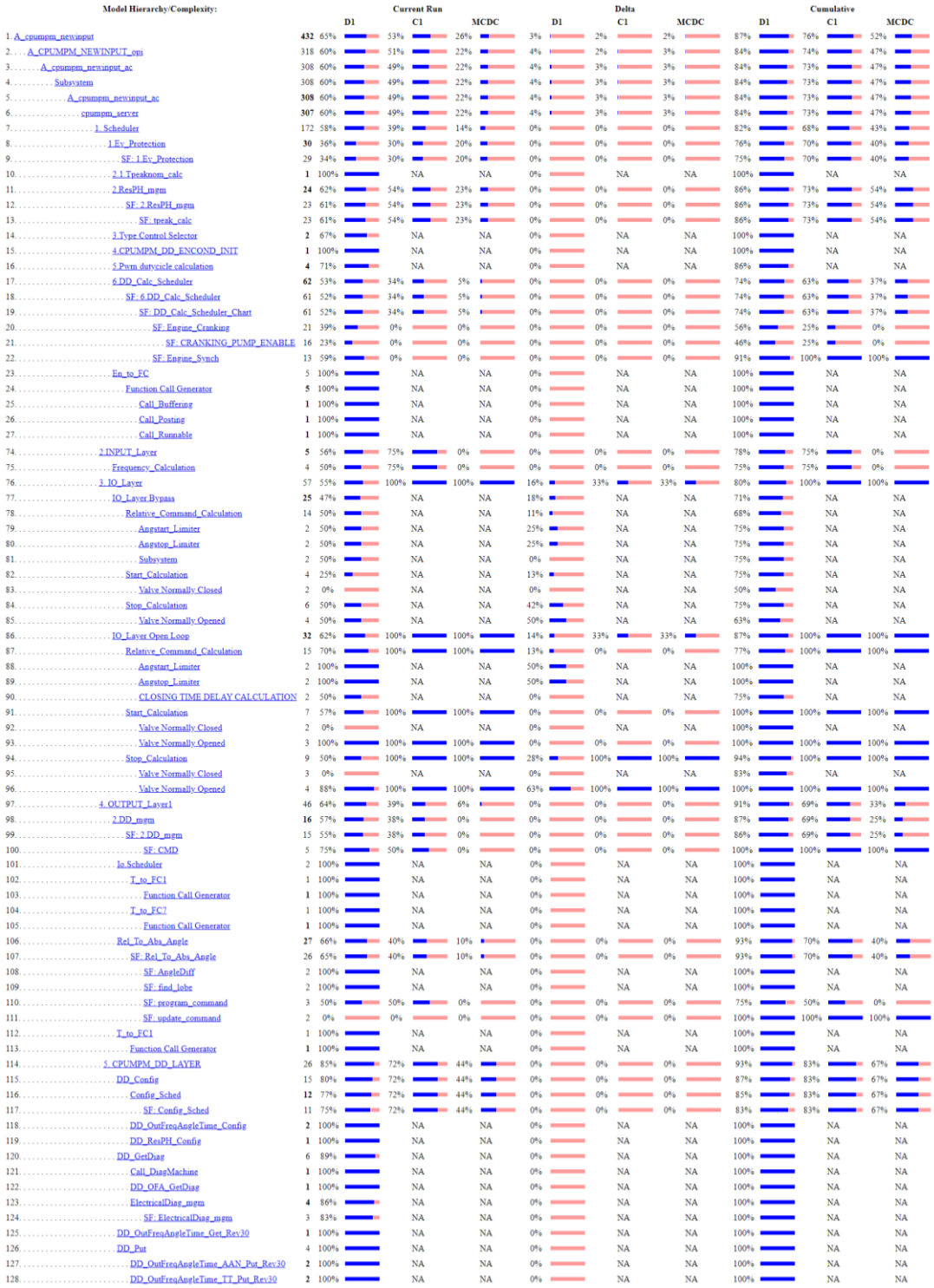


Figura 174 - Resoconto Coverage del blocco ac

Si possono notare che in tutti i blocchi sono presenti parti di logica non analizzate. Riassumendo i valori ottenuti per il blocco “ac” si ha:

DECISION	CONDITION	MC/DC
84%	73%	47%

Nota questa non totale copertura da parte del modello originale è stata eseguita l'analisi per il modello diviso per runnable

Model Hierarchy/Complexity:	Current Run						Delta						Cumulative					
		D1		C1		MCDC		D1		C1		MCDC		D1		C1		MCDC	
1. cpumpmr	597	59%		48%		23%		2%		2%		3%		84%		73%		45%	
162. cpumpmr_ac	377	51%		41%		18%		4%		4%		5%		78%		65%		39%	
163. Subsystem	377	51%		41%		18%		4%		4%		5%		78%		65%		39%	
164. cpumpmr_ac	377	51%		41%		18%		4%		4%		5%		78%		65%		39%	
165. cpumpmr_server	376	51%		41%		18%		4%		4%		5%		78%		65%		39%	
166. Call_DD_GetDiag_Mode	4	80%		NA		NA		0%		NA		NA		100%		NA		NA	
167. Call_DiagMachine	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
168. DD_OFA_GetDiag	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
169. ElectricalDiag_mgm	2	67%		NA		NA		0%		NA		NA		100%		NA		NA	
170. SF: ElectricalDiag_mgm	1	50%		NA		NA		0%		NA		NA		100%		NA		NA	
171. En_to_FC	5	100%		NA		NA		0%		NA		NA		100%		NA		NA	
172. Function Call Generator	5	100%		NA		NA		0%		NA		NA		100%		NA		NA	
173. Buffering	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
174. Call_Runnable	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
175. Posting	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
216. Engine_stopped	15	0%		0%		0%		0%		0%		0%		46%		50%		0%	
217. OFAT_Config_ES	10	0%		NA		NA		0%		NA		NA		38%		NA		NA	
218. OFAT_AA_OFF	5	0%		NA		NA		0%		NA		NA		75%		NA		NA	
219. DD_OutFreqAngleTime_Config	3	0%		NA		NA		0%		NA		NA		67%		NA		NA	
220. fc2tg	2	0%		NA		NA		0%		NA		NA		50%		NA		NA	
221. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		50%		NA		NA	
222. OFAT_TT_OFF	5	0%		NA		NA		0%		NA		NA		0%		NA		NA	
223. DD_OutFreqAngleTime_Config	3	0%		NA		NA		0%		NA		NA		0%		NA		NA	
224. fc2tg	2	0%		NA		NA		0%		NA		NA		0%		NA		NA	
225. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		0%		NA		NA	
226. engine_stop	4	0%		0%		0%		0%		0%		0%		50%		50%		0%	
227. SF: engine_stop	3	0%		0%		0%		0%		0%		0%		50%		50%		0%	
228. Key_On	11	70%		50%		0%		0%		0%		0%		70%		50%		0%	
229. Call_ResPH_Config	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
230. DD_ResPH_Config	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
231. OFAT_Config_KO	8	63%		50%		0%		0%		0%		0%		63%		50%		0%	
232. OFAT_TT_ON	8	63%		50%		0%		0%		0%		0%		63%		50%		0%	
233. Config_TT_ON	4	25%		50%		0%		0%		0%		0%		25%		50%		0%	
234. SF: Config_TT_ON	3	25%		50%		0%		0%		0%		0%		25%		50%		0%	
235. DD_OutFreqAngleTime_Config	3	100%		NA		NA		0%		NA		NA		100%		NA		NA	
236. fc2tg	2	100%		NA		NA		0%		NA		NA		100%		NA		NA	
237. SF: fc2tg	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
238. Keyoff	15	0%		0%		0%		0%		0%		0%		46%		50%		0%	
239. OFAT_Config_Ko	10	0%		NA		NA		0%		NA		NA		38%		NA		NA	
240. OFAT_AA_OFF	5	0%		NA		NA		0%		NA		NA		75%		NA		NA	
241. DD_OutFreqAngleTime_Config	3	0%		NA		NA		0%		NA		NA		67%		NA		NA	
242. fc2tg	2	0%		NA		NA		0%		NA		NA		50%		NA		NA	
243. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		50%		NA		NA	
244. OFAT_TT_OFF	5	0%		NA		NA		0%		NA		NA		0%		NA		NA	
245. DD_OutFreqAngleTime_Config	3	0%		NA		NA		0%		NA		NA		0%		NA		NA	
246. fc2tg	2	0%		NA		NA		0%		NA		NA		0%		NA		NA	
247. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		0%		NA		NA	
248. key_off	4	0%		0%		0%		0%		0%		0%		50%		50%		0%	
249. SF: key_off	3	0%		0%		0%		0%		0%		0%		50%		50%		0%	
250. Power_On	12	64%		50%		0%		0%		0%		0%		64%		50%		0%	
251. Call_ResPH_Config	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
252. DD_ResPH_Config	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
253. OFAT_Config_PO	8	50%		50%		0%		0%		0%		0%		50%		50%		0%	
254. OFAT_TT_ON	8	50%		50%		0%		0%		0%		0%		50%		50%		0%	
255. Config_TT_ON	4	25%		50%		0%		0%		0%		0%		25%		50%		0%	
256. SF: Config_TT_ON	3	25%		50%		0%		0%		0%		0%		25%		50%		0%	
257. DD_OutFreqAngleTime_Config	3	67%		NA		NA		0%		NA		NA		67%		NA		NA	
258. fc2tg	2	50%		NA		NA		0%		NA		NA		50%		NA		NA	
259. SF: fc2tg	1	50%		NA		NA		0%		NA		NA		50%		NA		NA	
260. Subsystem	2	100%		NA		NA		0%		NA		NA		100%		NA		NA	
261. 4.CPUMPM_DD_ENCOND_INIT	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
260. Subsystem	2	100%		NA		NA		0%		NA		NA		100%		NA		NA	
261. 4.CPUMPM_DD_ENCOND_INIT	1	100%		NA		NA		0%		NA		NA		100%		NA		NA	
262. PumpException	71	12%		0%		0%		0%		0%		0%		60%		56%		50%	
263. DD_Put_PE	8	0%		0%		NA		0%		0%		NA		67%		50%		NA	
264. Call_DD_Put_AA_Mode	3	0%		NA		NA		0%		NA		NA		100%		NA		NA	
265. DD_OutFreqAngleTime_AAN_Put_Rev30	3	0%		NA		NA		0%		NA		NA		100%		NA		NA	
266. fc2tg	2	0%		NA		NA		0%		NA		NA		100%		NA		NA	
267. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		100%		NA		NA	
268. Call_DD_Put_TT_Mode1	3	0%		NA		NA		0%		NA		NA		0%		NA		NA	
269. DD_OutFreqAngleTime_TT_Put_Rev30	3	0%		NA		NA		0%		NA		NA		0%		NA		NA	
270. fc2tg	2	0%		NA		NA		0%		NA		NA		0%		NA		NA	
271. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		0%		NA		NA	
272. new_Tg	2	0%		0%		NA		0%		0%		NA		100%		50%		NA	
273. DD_mgm_func	14	22%		0%		0%		0%		0%		0%		74%		100%		100%	
274. DD_mgm_PE	13	18%		0%		0%		0%		0%		0%		73%		100%		100%	
275. SF: DD_mgm_PE	12	18%		0%		0%		0%		0%		0%		73%		100%		100%	
276. OFAT_Config_PE	29	0%		0%		0%		0%		0%		0%		50%		40%		25%	
277. Call_SignalGen3	1	0%		NA		NA		0%		NA		NA		100%		NA		NA	
278. OFAT_AA_OFF	5	0%		NA		NA		0%		NA		NA		75%		NA		NA	
279. DD_OutFreqAngleTime_Config	3	0%		NA		NA		0%		NA		NA		67%		NA		NA	
280. fc2tg	2	0%		NA		NA		0%		NA		NA		50%		NA		NA	
281. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		50%		NA		NA	
282. OFAT_AA_ON	8	0%		0%		0%		0%		0%		0%		88%		75%		50%	
283. Config_AA_ON	4	0%		0%		0%		0%		0%		0%		75%		75%		50%	
284. SF: Config_AA_ON	3	0%		0%		0%		0%		0%		0%		75%		75%		50%	
285. DD_OutFreqAngleTime_Config	3	0%		NA		NA		0%		NA		NA		100%		NA		NA	
286. fc2tg	2	0%		NA		NA		0%		NA		NA		100%		NA		NA	
287. SF: fc2tg	1	0%		NA		NA		0%		NA		NA		100%		NA		NA	

Figura 175 -Resoconto Coverage runnable

288	OFAT_TT_OFF	5	0%		NA		NA	0%		NA		NA	0%		NA		NA
289	DD_OutFreqAngleTime_Config	3	0%		NA		NA	0%		NA		NA	0%		NA		NA
290	fc2tg	2	0%		NA		NA	0%		NA		NA	0%		NA		NA
291	SF: fc2tg	1	0%		NA		NA	0%		NA		NA	0%		NA		NA
292	OFAT_TT_ON	8	0%		0%		0%	0%		0%		0%	0%		0%		0%
293	Config_TT_ON	4	0%		0%		0%	0%		0%		0%	0%		0%		0%
294	SF: Config_TT_ON	3	0%		0%		0%	0%		0%		0%	0%		0%		0%
295	DD_OutFreqAngleTime_Config	3	0%		NA		NA	0%		NA		NA	0%		NA		NA
296	fc2tg	2	0%		NA		NA	0%		NA		NA	0%		NA		NA
297	SF: fc2tg	1	0%		NA		NA	0%		NA		NA	0%		NA		NA
298	new_Tg	2	0%		0%		NA	0%		0%		NA	100%		50%		NA
299	OUTPUT_Layer_PumpExc	6	22%		NA		NA	0%		NA		NA	100%		NA		NA
300	Rel_To_Abs_Angle	5	13%		NA		NA	0%		NA		NA	100%		NA		NA
301	SF: Rel_To_Abs_Angle	4	13%		NA		NA	0%		NA		NA	100%		NA		NA
302	SF: AngleDiff	2	0%		NA		NA	0%		NA		NA	100%		NA		NA
303	Subsystem	13	13%		NA		NA	0%		NA		NA	39%		NA		NA
304	5.Pwm dutycicle calculation_PumpEx	4	0%		NA		NA	0%		NA		NA	0%		NA		NA
305	pump_ex	9	19%		NA		NA	0%		NA		NA	56%		NA		NA
306	SF: pump_ex	8	19%		NA		NA	0%		NA		NA	56%		NA		NA
307	TDC	155	61%		53%		29%	7%		7%		8%	86%		73%		54%
308	2.INPUT_Layer	5	56%		75%		0%	0%		0%		0%	78%		75%		0%
309	Frequency_Calculation	4	50%		75%		0%	0%		0%		0%	75%		75%		0%
310	3. IO_Layer	57	55%		100%		100%	16%		33%		33%	80%		100%		100%
311	IO_Layer Bypass	25	47%		NA		NA	18%		NA		NA	71%		NA		NA
312	Relative_Command_Calculation	14	50%		NA		NA	11%		NA		NA	68%		NA		NA
313	Angstart_Limiter	2	50%		NA		NA	25%		NA		NA	75%		NA		NA
314	Angstop_Limiter	2	50%		NA		NA	25%		NA		NA	75%		NA		NA
315	Subsystem	2	50%		NA		NA	0%		NA		NA	75%		NA		NA
316	Start_Calculation	4	25%		NA		NA	13%		NA		NA	75%		NA		NA
317	Valve Normally Closed	2	0%		NA		NA	0%		NA		NA	50%		NA		NA
318	Stop_Calculation	6	50%		NA		NA	42%		NA		NA	75%		NA		NA
319	Valve Normally Opened	4	50%		NA		NA	50%		NA		NA	63%		NA		NA
320	IO_Layer Open Loop	32	62%		100%		100%	14%		33%		33%	87%		100%		100%
321	Relative_Command_Calculation	15	70%		100%		100%	13%		0%		0%	77%		100%		100%
322	Angstart_Limiter	2	100%		NA		NA	50%		NA		NA	100%		NA		NA
323	Angstop_Limiter	2	100%		NA		NA	50%		NA		NA	100%		NA		NA
324	CLOSING TIME DELAY CALCULATION	2	50%		NA		NA	0%		NA		NA	75%		NA		NA
325	Start_Calculation	7	57%		100%		100%	0%		0%		0%	100%		100%		100%
326	Valve Normally Closed	2	0%		NA		NA	0%		NA		NA	100%		NA		NA
327	Valve Normally Opened	3	100%		100%		100%	0%		0%		0%	100%		100%		100%
328	Stop_Calculation	9	50%		100%		100%	28%		100%		100%	94%		100%		100%
329	Valve Normally Closed	3	0%		NA		NA	0%		NA		NA	83%		NA		NA
330	Valve Normally Opened	4	88%		100%		100%	63%		100%		100%	100%		100%		100%
331	Call_DD_Get_Mode	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
332	DD_OutFreqAngleTime_Get_Rev30	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
333	Call_ResPH_Config	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
334	DD_ResPH_Config	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
335	DD_Put_TDC	5	83%		0%		NA	0%		0%		NA	100%		100%		NA
336	Call_DD_Put_AA_Mode	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
337	DD_OutFreqAngleTime_AAN_Put_Rev30	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
338	fc2tg	2	100%		NA		NA	0%		NA		NA	100%		NA		NA
339	SF: fc2tg	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
340	new_Tg	2	67%		0%		NA	0%		0%		NA	100%		100%		NA
341	DD_mgm_func	15	56%		50%		0%	0%		0%		0%	92%		100%		100%
342	DD_mgm_TDC	14	54%		50%		0%	0%		0%		0%	92%		100%		100%
343	SF: DD_mgm_TDC	13	54%		50%		0%	0%		0%		0%	92%		100%		100%
344	OFAT_Config_TDC	22	69%		29%		0%	0%		0%		0%	85%		43%		50%
345	Call_SignalGen1	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
346	Call_SignalGen2	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
347	Call_SignalGen3	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
348	OFAT_AA_OFF	5	75%		NA		NA	0%		NA		NA	100%		NA		NA
349	DD_OutFreqAngleTime_Config	3	67%		NA		NA	0%		NA		NA	100%		NA		NA
350	fc2tg	2	50%		NA		NA	0%		NA		NA	100%		NA		NA
351	SF: fc2tg	1	50%		NA		NA	0%		NA		NA	100%		NA		NA
352	OFAT_AA_ON	8	63%		50%		0%	0%		0%		0%	100%		75%		50%
353	Config_AA_ON	4	25%		50%		0%	0%		0%		0%	100%		75%		50%
354	SF: Config_AA_ON	3	25%		50%		0%	0%		0%		0%	100%		75%		50%
355	DD_OutFreqAngleTime_Config	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
356	fc2tg	2	100%		NA		NA	0%		NA		NA	100%		NA		NA
357	SF: fc2tg	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
358	new_Tg	6	64%		20%		NA	0%		0%		NA	64%		30%		NA
359	OUTPUT_Layer_TDC	23	68%		40%		10%	0%		0%		0%	94%		70%		40%
360	Rel_To_Abs_Angle_TDC	22	67%		40%		10%	0%		0%		0%	93%		70%		40%
361	SF: Rel_To_Abs_Angle_TDC	21	67%		40%		10%	0%		0%		0%	93%		70%		40%
362	SF: AngleDiff	2	100%		NA		NA	0%		NA		NA	100%		NA		NA
363	SF: find_lobe	2	100%		NA		NA	0%		NA		NA	100%		NA		NA
364	SF: program_command	3	50%		50%		0%	0%		0%		0%	75%		50%		0%
365	SF: update_command	2	0%		0%		0%	0%		0%		0%	100%		100%		100%
366	Subsystem	25	64%		75%		0%	0%		0%		0%	91%		75%		0%
367	2.1.Tpeaknom_calc_TDC	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
368	3.Type Control Selector	2	67%		NA		NA	0%		NA		NA	100%		NA		NA
369	tdc	22	63%		75%		0%	0%		0%		0%	90%		75%		0%
370	SF: tdc	21	63%		75%		0%	0%		0%		0%	90%		75%		0%
371	Vbat	9	73%		50%		0%	0%		0%		0%	73%		50%		0%
372	Call_ResPH_Config1	1	0%		NA		NA	0%		NA		NA	0%		NA		NA
373	DD_ResPH_Config	1	0%		NA		NA	0%		NA		NA	0%		NA		NA
374	Subsystem	7	78%		50%		0%	0%		0%		0%	78%		50%		0%
375	2.1.Tpeaknom_calc_Vbat	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
376	v-bat	6	75%		50%		0%	0%		0%		0%	75%		50%		0%
377	SF: v-bat	5	75%		50%		0%	0%		0%		0%	75%		50%		0%
378	ev_4ms	38	60%		63%		25%	0%		0%		0%	71%		63%		25%
379	Call_ResPH_Config1	1	0%		NA		NA	0%		NA		NA	0%		NA		NA
380	DD_ResPH_Config	1	0%		NA		NA	0%		NA		NA	0%		NA		NA
381	DD_OUT	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
382	Call_DD_Put_TT_Mode1	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
383	DD_OutFreqAngleTime_TT_Put_Rev30	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
384	fc2tg	2	100%		NA		NA	0%		NA		NA	100%		NA		NA
385	SF: fc2tg	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
386	OFAT_Config_4ms	13	67%		50%		0%	0%		0%		0%	67%		50%		0%
387	OFAT_TT_OFF	5	100%		NA		NA	0%		NA		NA	100%		NA		NA
388	DD_OutFreqAngleTime_Config	3	100%		NA		NA	0%		NA		NA	100%		NA		NA
389	fc2tg	2	100%		NA		NA	0%		NA		NA	100%		NA		NA
390	SF: fc2tg	1	100%		NA		NA	0%		NA		NA	100%		NA		NA
391	OFAT_TT_ON	8	50%		50%		0%	0%		0%		0%	50%		50%		0%
392	Config_TT_ON	4	25%		50%		0%	0%		0%		0%	25%		50%		0%
393	SF: Config_TT_ON	3	25%		50%		0%	0%		0%		0%	25%		50%		0%
394	DD_OutFreqAngleTime_Config	3	67%		NA		NA	0%		NA		NA	67%		NA		NA
395	fc2tg	2	50%		NA		NA	0%		NA		NA	50%		NA		NA
396	SF: fc2tg	1	50%		NA		NA	0%		NA		NA	50%		NA		NA
397	Subsystem	20	54%		75%		50%	0%		0%		0%	71%		75%		50%
398	4ms	16	50%		75%		50%	0%		0%		0%	68%		75%		50%
399	SF: 4ms	15	50%		75%		50%	0%		0%		0%	68%		75%		50%
400	5.Pwm dutycicle calculation_pumEx	4	71%		NA		NA	0%		NA		NA	86%		NA		NA

Si può notare che la lista di analisi sia molto più lunga rispetto al caso della divisione per stati in quanto è cambiata completamente la struttura del modello.

I risultati ottenuti dall'analisi sono:

DECISION	CONDITION	MC/DC
65%	78%	39%

Questa flessione delle percentuali è dovuta al fatto che le logiche sono state "duplicate". Infatti, per ogni singolo blocco comune, se prima era sufficiente l'attivazione tramite un qualsiasi task per renderlo "verificato" ora non è possibile in quanto nell'analisi è definito come blocco a parte. Avendo inoltre già una mancanza di copertura totale nel modello originale, con la duplicazione delle logiche si è ridotta maggiormente la percentuale di Coverage. In uno step di analisi successivo potrebbe essere di interesse raggiungere con opportuni Test Pattern la copertura totale del modello originale. Attualmente la verifica dell'isofunzionalità del nuovo modello è garantita per l'84% del modello originale.

AUTO SIL

Successivamente è stata eseguita la verifica dell'auto sil. Anche in questo caso non è presente una totale coerenza tra i risultati delle simulazioni MIL e SIL sin dal modello originale. Questi errori rilevati sono stati considerati accettabili, pertanto l'obiettivo è stato quello di ottenere dei risultati uguali e non peggiorativi rispetto al caso originale.

Per motivi di sintesi di riportano di seguito nel dettaglio solamente i risultati ottenuti per l'auto sil del Test Pattern 1 dei cinque segnali designati per la verifica. Per gli altri due scenari che risultano "Ko" si fornisce solo la lista dei segnali non congrui.

ORIGINALE

RUNNABLE

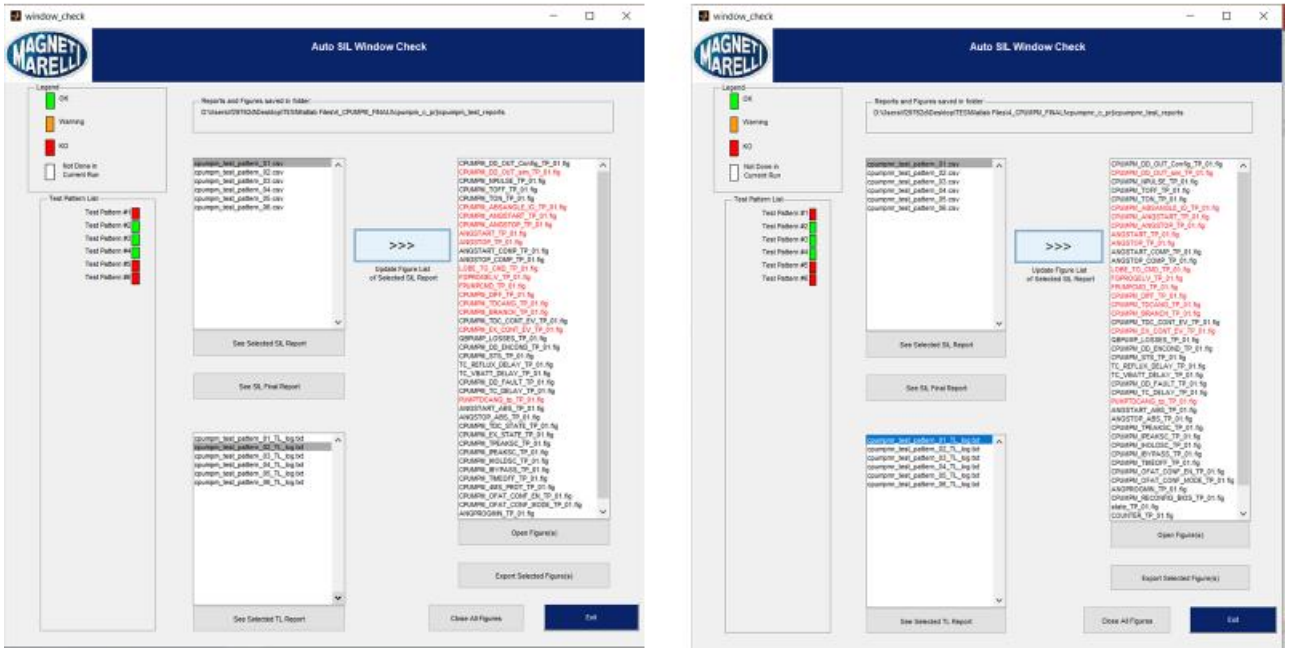


Figura 176 - Confronto auto sil TestPattern 1

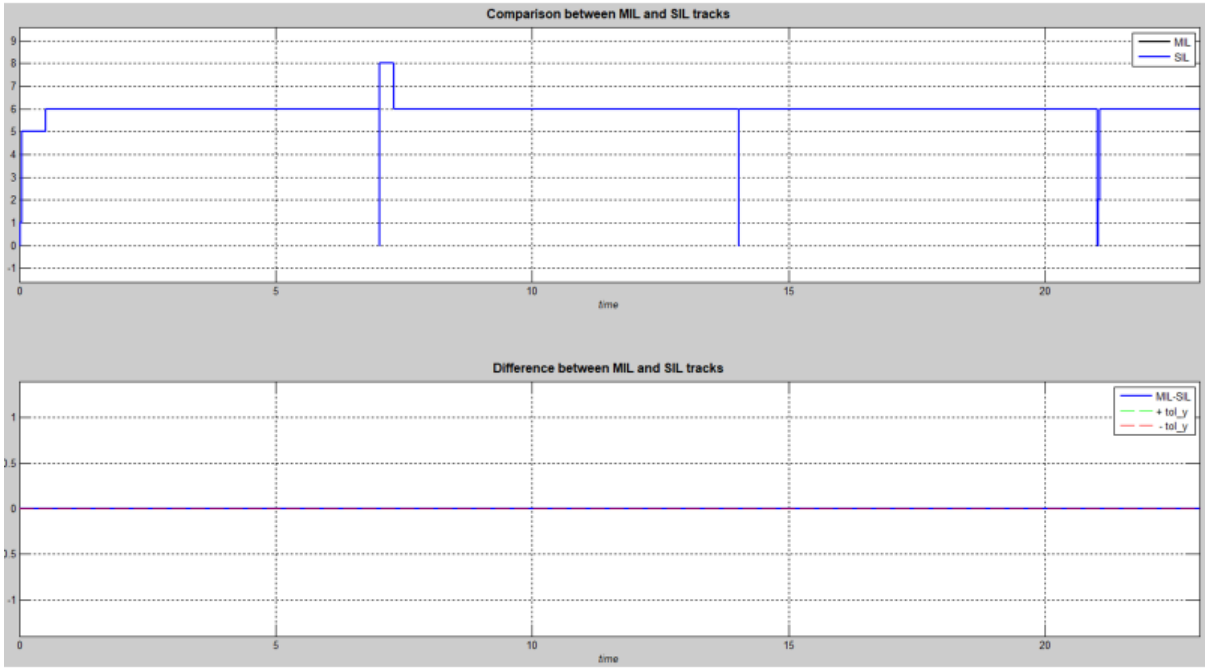


Figura 177 – Confronto auto sil CPUMPM_STS

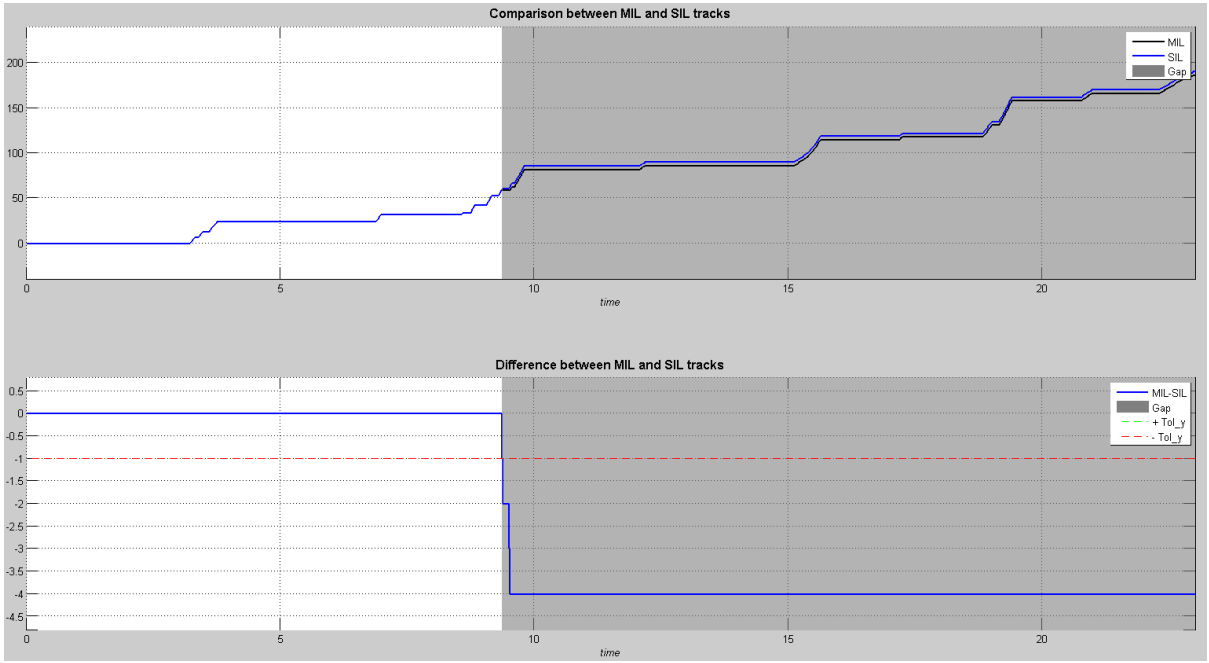


Figura 178 – Confronto auto sil CPUMPM_EX_CONT_EV

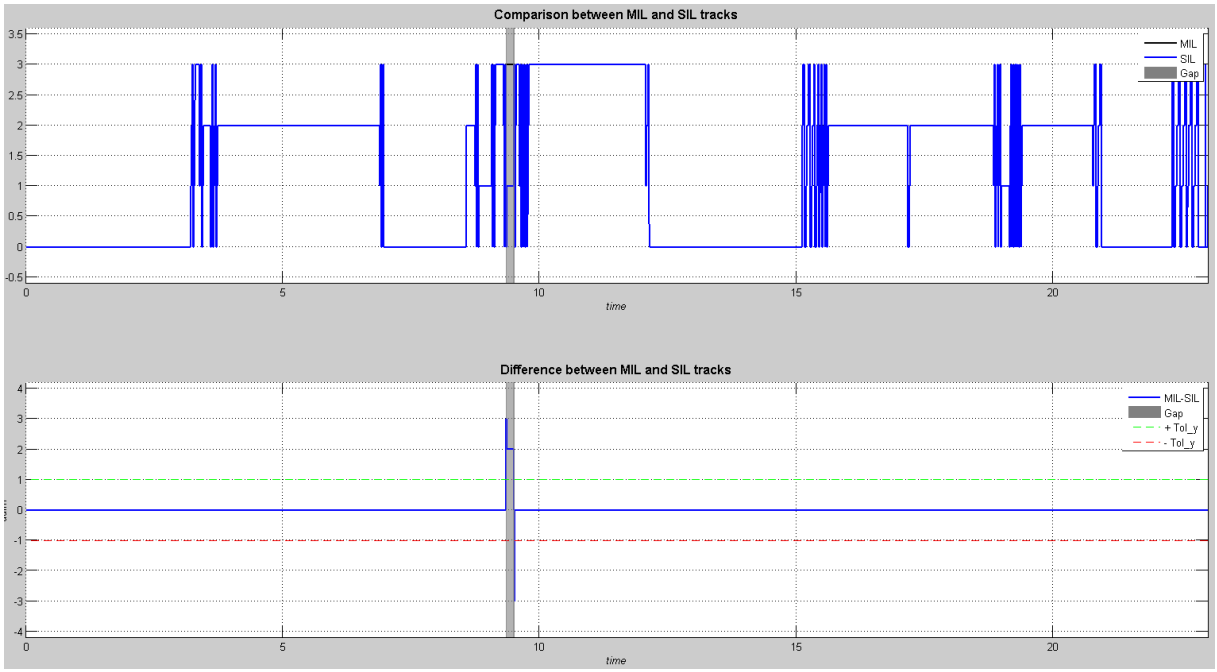


Figura 179 - Confronto auto sil LOBE_TO_CMD

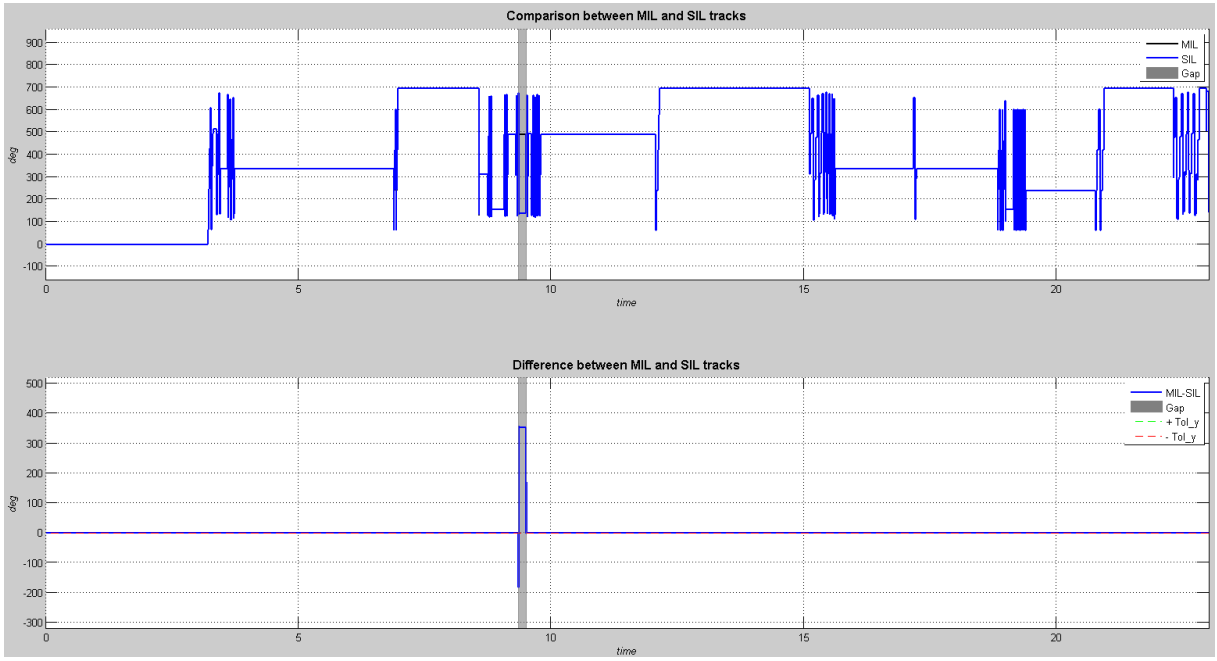


Figura 180 - Confronto auto sil CPUMPM_ANGSTART

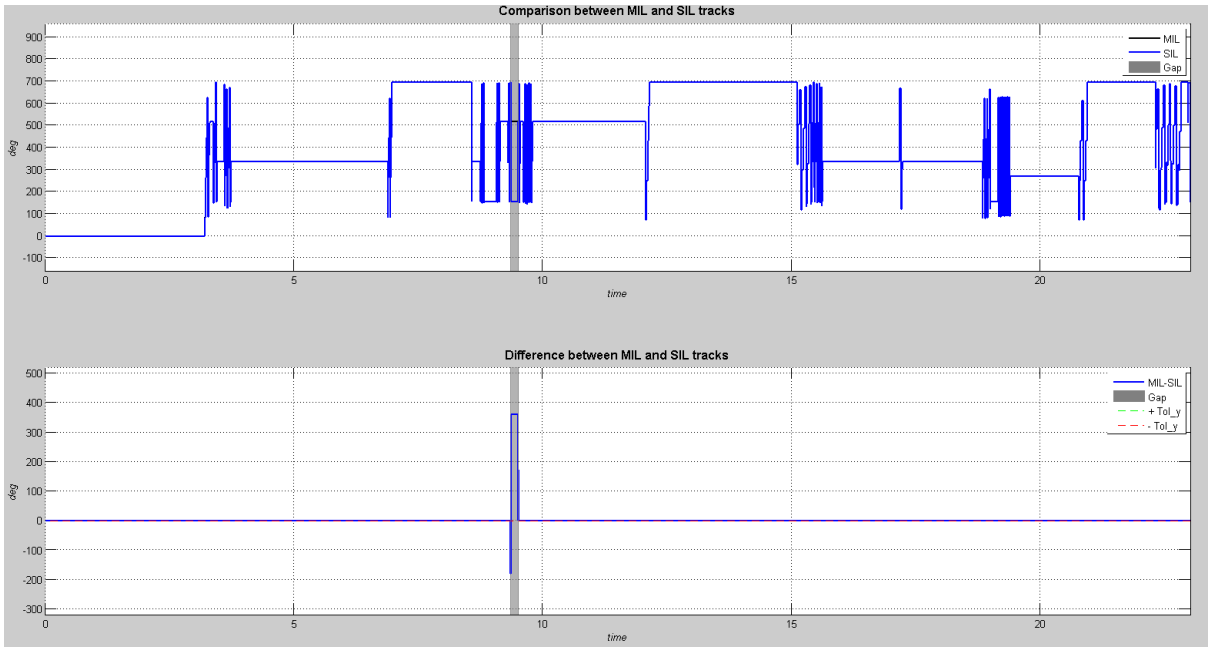


Figura 181 - Confronto auto sil CPUMPM_ANGSTOP

ORIGINALE

RUNNABLE

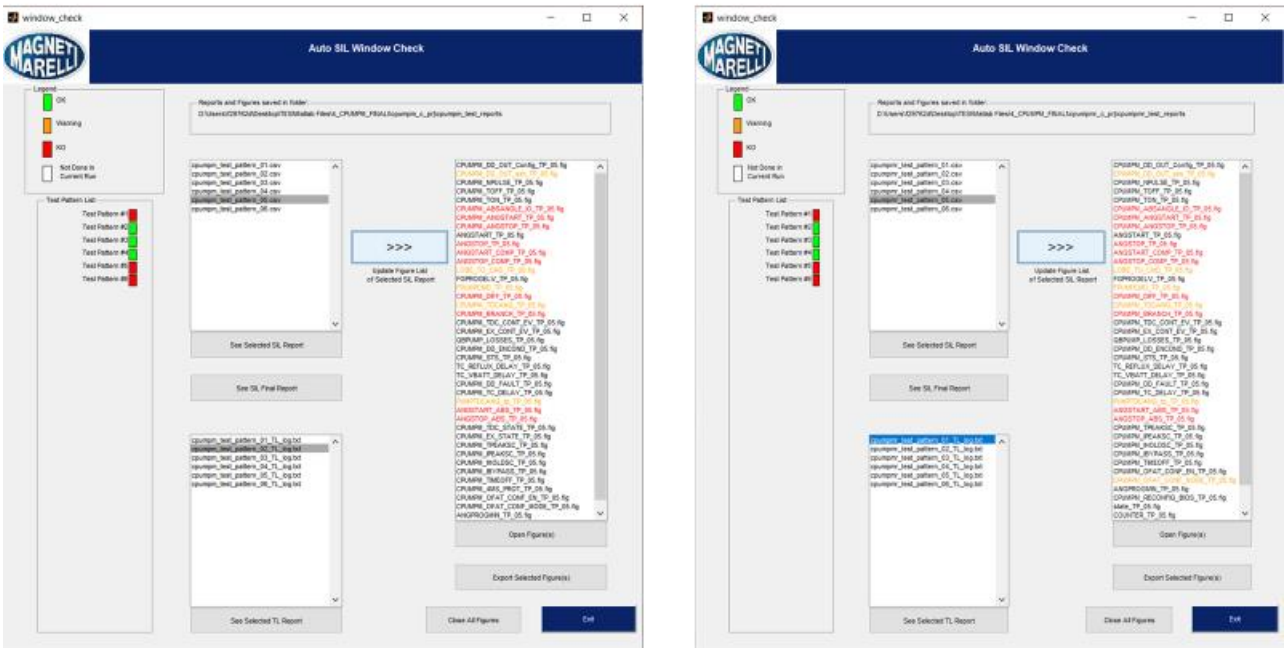


Figura 182 - confronto auto sil TestPattern 5

ORIGINALE

RUNNABLE

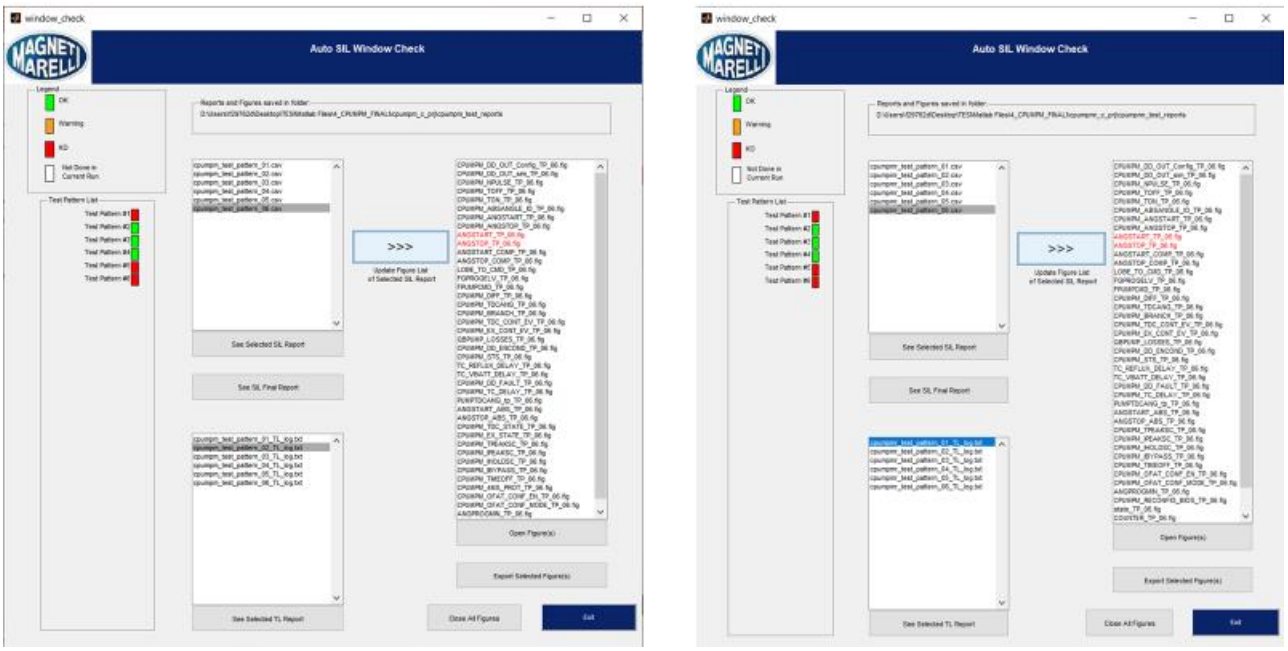


Figura 183 - Confronto TestPattern 6

VERIFICA SIMULAZIONI SIL-SIL

L'ultima verifica prima di passare alle analisi delle metriche è quella di ottenere dei risultati simili tra gli output delle simulazioni SIL tra i due modelli.

In generale, avendo simulazioni MIL coerenti e verifica dell'auto_sil con tutti gli output dentro alla tolleranza desiderata non è possibile vi siano grandi differenze tra i risultati ottenuti nelle simulazioni SIL dei due modelli. Infatti, avendo le verifiche precedenti con risultati uguali lo scostamento massimo potrà risultare del valore di tolleranza scelto per l'auto sil.

CPUMPM – ANALISI DEI RISULTATI

I risultati ottenuti sono soddisfacenti, pertanto si è deciso di eseguire la verifica secondo le metriche già viste per UTET:

- Lunghezza del codice in righe
- Tempo di simulazione MIL
- Tempo di compilazione
- Tempo di simulazione SIL
- Tempo esecuzione verifica "auto sil"

LUNGHEZZA DEL CODICE

La lunghezza del codice è la metrica più semplice da verificare. Sono stati analizzati i file "<model>.c" ottenuti dai due modelli in versione a stati e a runnable.

	Lunghezza codice	
Originale	6058	
Runnable	6367	309

Al contrario del caso di UTET, la lunghezza del codice è a favore della configurazione originale. Questo è causato dal fatto che sono state impiegate delle logiche particolari al fine di ottenere un modello isofunzionale inserendo dei blocchi logici aggiuntivi utili solo al funzionamento del modello in ambiente Matlab. Altre possibili cause sono l'inserimento di controlli aggiuntivi delle variabili inter-runnable.

TEMPI

La definizione delle tempistiche è stata eseguita nello stesso metodo esposto nel capitolo riguardante il modello UTET.

Di seguito la tabella riassuntiva dei tempi medi necessari per le simulazioni **MIL**, **SIL** e **GENERAZIONE** codice e **COMPILAZIONE** di entrambe le configurazioni (States e Runnable) per ogni Test Pattern. A fianco ai tempi relativi al modello Runnable sono riportate le differenze tra i tempi states e runnable (positivo, negativo).

		TP_1		TP_2		TP_3		TP_4		TP_5		TP_6	
Originale	MIL	21,15		22,03		21,62		22,36		21,84		21,56	
	SIL	15,77		15,79		15,61		15,78		15,70		15,80	
	COMP.	26,61		26,90		26,76		27,21		26,79		27,12	
Runnable	MIL	26,73	5,58	28,82	6,79	27,39	5,77	27,42	5,06	27,54	5,73	27,31	5,75
	SIL	17,63	1,86	17,61	1,82	17,73	2,12	17,71	1,93	17,77	2,07	17,68	1,88
	COMP.	46,64	20,03	46,66	19,76	46,39	19,63	46,55	19,34	46,93	20,14	46,92	19,80

L'ultima tempistica considerata è quella necessaria al test **auto sil**.

	Tempo [sec]	
ORIGINALE	1063.19	
RUNNABLE	1119.17	55.79

Anche nell'analisi dei tempi si è rilevata una tendenza opposta a quella osservata nel caso del modello UTET. Si è pertanto deciso di eseguire delle indagini per poter trovare possibili cause di questi comportamenti.

Si è iniziato ad analizzare il codice nelle sue configurazioni, l'aspetto più rilevante era la modifica necessaria per garantire l'isofunzionalità delle chiamate "Put" e "Config". A questo punto, avendo definito la robustezza del sistema e la sua isofunzionalità è stato deciso di togliere la parte di logica inerente al controllo del numero di esecuzioni e semplificare anche le logiche presenti nel blocco di comunicazione con il Device Driver.

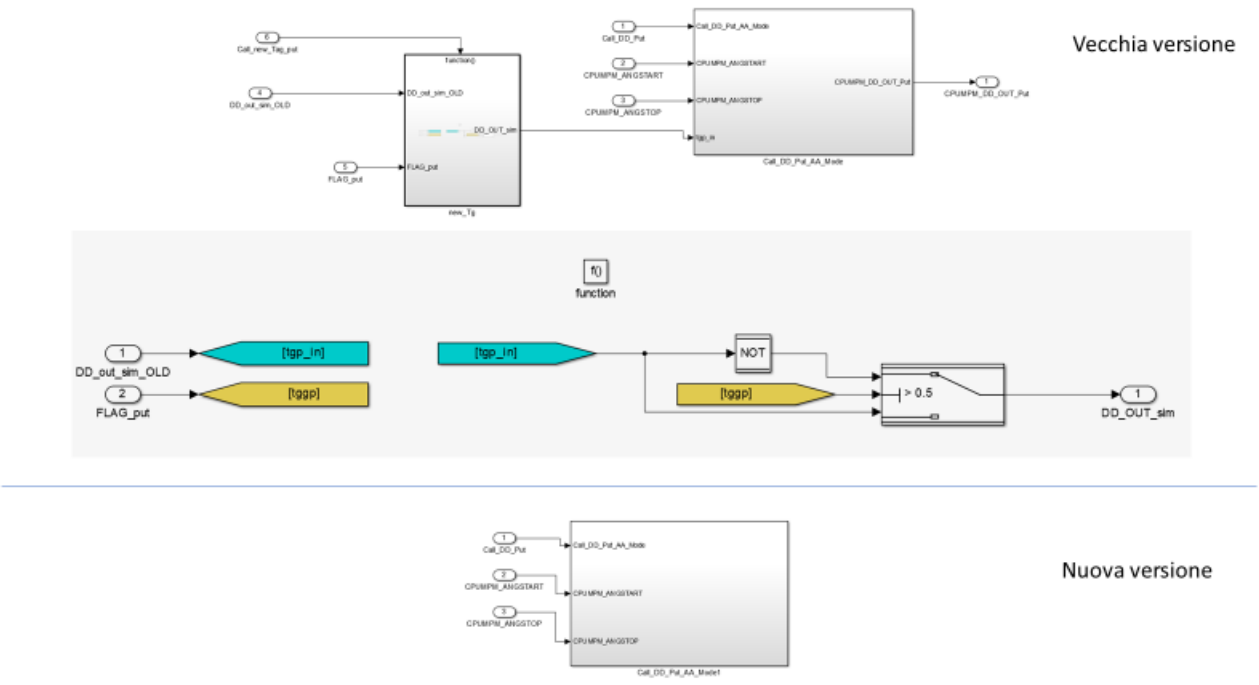


Figura 184 - Dettaglio modifica con eliminazione blocco

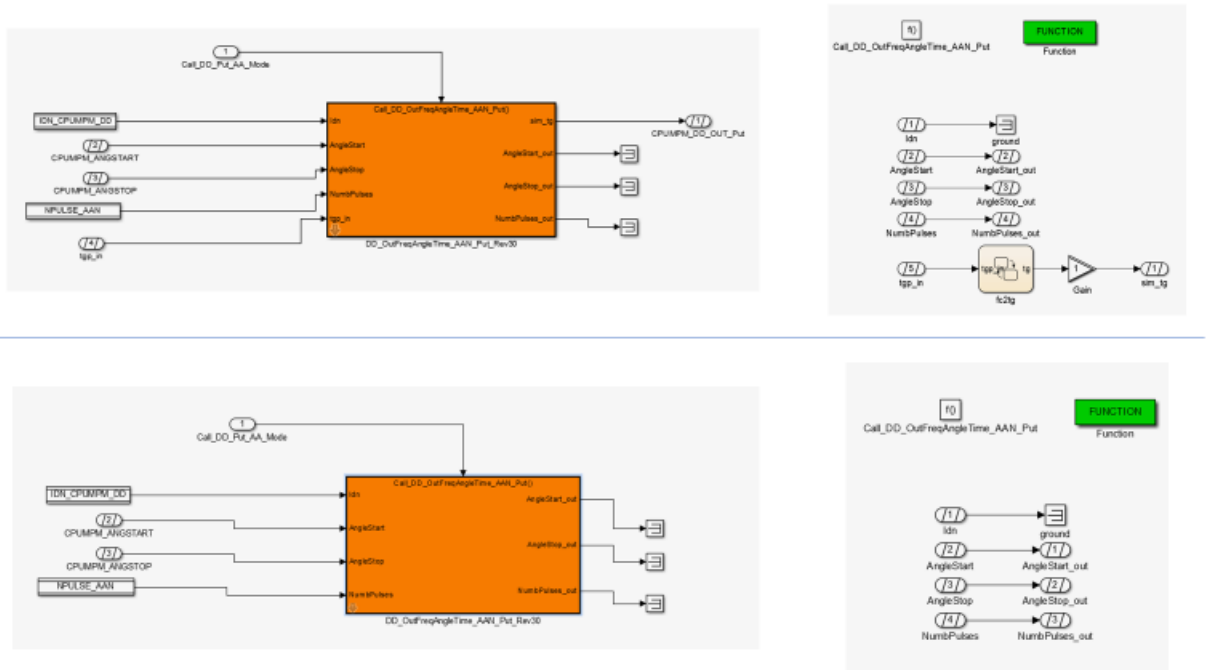


Figura 185 - Confronto della modifica del blocco FUNCTION

Con queste modifiche però si sono rimossi alcuni segnali che garantivano il funzionamento del modello. Si è deciso di eseguire quindi una modifica anche al blocco SimulationResult. È stata eseguita la simulazione del modello originale di tutti i Test Pattern registrando il valore assunto dalle variabili CPUMPM_DD_out_sim e CPUMPM_DD_out_Config in quanto queste variabili erano funzionali solo per il funzionamento del modello e non erano utili per

l'implementazione successiva del codice C. A questo punto la matrice ottenuta è stata salvata in un file di estensione ".mat". Nel nuovo modello è stato sostituito l'input del Device Driver Model con un blocco di lettura di vettori. Ad ogni simulazione, quindi, sono stati caricati nel WorkSpace i vettori relativi a quel Test Pattern per le due variabili.

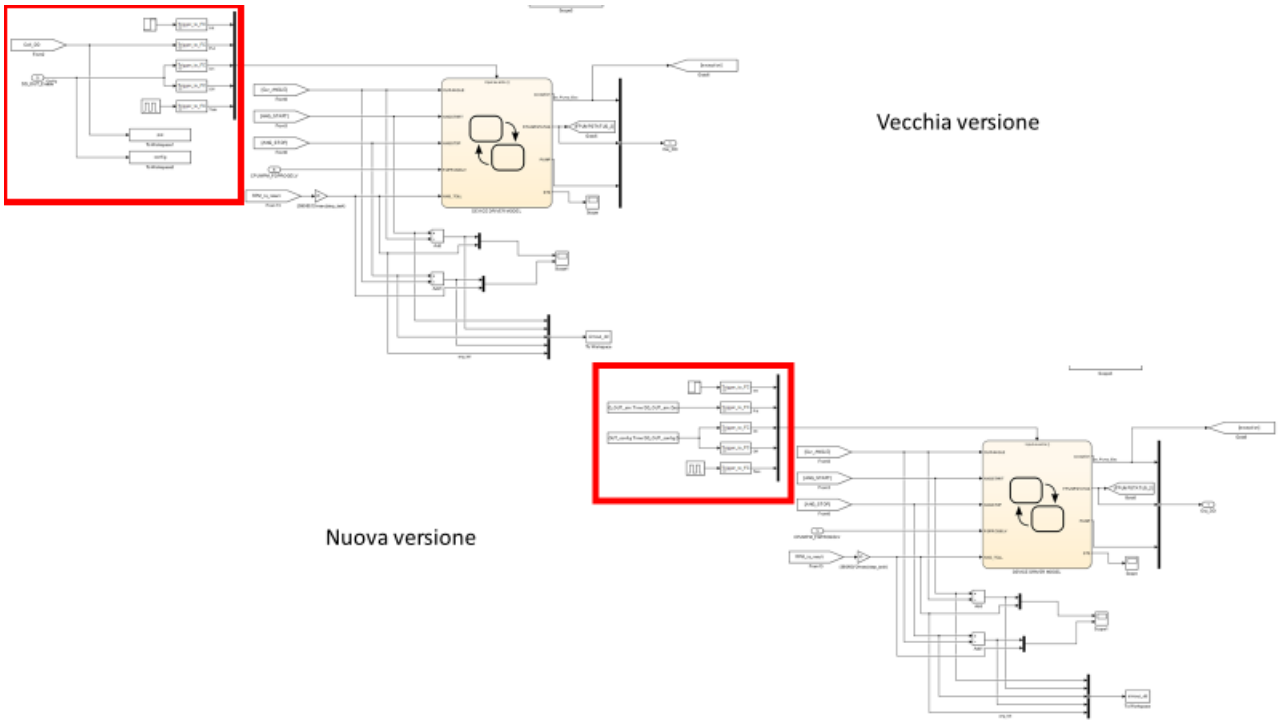


Figura 186 - Modifica blocco SimulationResult

Le simulazioni in questo modo risultavano avere gli stessi output alle versioni precedenti tranne per le grandezze CPUMPM_DD_out_sim e CPUMPM_DD_out_Config. I modelli pertanto si possono ancora considerare isofunzionali. Questa modifica per poter rendere la simulazione e il confronto più reale è stata fatta anche al modello originale.

Nuovi risultati

Si è andati a verificare come prima cosa l'effetto sul codice della nuova forma di modellazione ed è stata riscontrata, come previsto, una riduzione delle linee di codice per entrambi i modelli (originale e runnable). Infatti, l'eliminazione di blocchi logici presenti nel blocco "ac" per entrambi i modelli comporta una riduzione del codice autoprodotto dal tool di generazione. Questa volta la configurazione a runnable è risultata in grado di generare un codice più compatto.

	Lunghezza codice	
Originale	5883	
Runnable	5803	80

Dall'analisi dei tempi invece non sono stati rilevati miglioramenti per quanto riguarda la durata delle simulazioni MIL o SIL, il divario rimane tale. È stata notata invece una contrazione della fase di generazione del codice per il modello a runnable. In generale, i tempi richiesti per l'esecuzione delle prove è rimasto a favore del modello originale.

		TP_1		TP_2		TP_3		TP_4		TP_5		TP_6	
Originale	MIL	20,77		20,86		21,05		20,96		21,09		20,97	
	SIL	16,08		15,98		16,06		16,47		16,12		16,01	
	GEN.	25,63		25,59		25,63		25,65		25,82		26,05	
Runnable	MIL	25,99	5,22	26,37	5,51	26,93	5,88	27,09	6,13	26,91	5,82	28,16	7,19
	SIL	18,10	2,02	17,88	1,90	17,99	1,93	18,06	1,59	17,81	1,69	17,87	1,86
	GEN.	39,24	13,61	37,74	12,15	37,81	12,18	37,77	12,12	37,98	12,16	37,99	11,94

L'ultima prova, quella dell'auto sil, ha fornito anche in questo caso una tempistica comparabile al caso pre-modifica e con un divario di circa lo stesso ordine di grandezza.

	auto sil	
Originale	1075,15	
Runnable	1089,17	25,79

Al contrario dell'indicazione delle righe di codice, l'analisi dei tempi di simulazione è un indicatore parziale. Infatti, non è certo che una volta inserito il codice in centralina sia presente lo stesso trend peggiorativo nell'esecuzione delle logiche.

CONCLUSIONI

A seguito del lavoro di indagine e dall'esperienza ricavata nella modellazione vi sono più aspetti da considerare e portare in evidenza.

- La modifica di un modello da una configurazione a Stati a una per Runnable comporta una modifica totale del modello, anche nella gestione degli input in quanto è obbligatorio garantire la presenza univoca per ogni simstep di un solo e unico evento. Solo in questo modo si possono verificare se le modifiche eseguite hanno prodotto un modello isofunzionale.
- Tutte i simboli nelle logiche Stateflow corrispondenti a variabili interne o che descrivono gli stati devono esser trasformate in variabili di ricircolo. Il ricircolo di variabile è sempre necessario quando in due logiche Stateflow separate è presente come output una variabile che è anche di input, soprattutto se può esser stata modificata da un altro evento.
- La duplicazione di blocchi con la stessa logica è inevitabile quando vi sono logiche condivise da più task. In questo caso al fine di non rischiare di eseguire le modifiche a una sola parte del modello che usa quella logica è consigliabile l'implementazione tramite blocchi di libreria identificati come FUNCTION. Questo tipo di modellazione è lievemente più complesso della modellazione originale per quanto riguarda la gestione della targhettatura in quanto è possibile eseguirla solo dopo aver "rotto" il link con la libreria e per aggiornarlo su tutto il modello è necessario "risolvere" il link.
- La modellazione è graficamente intuitiva e di immediata lettura per quanto riguarda la catena di calcolo relativa a un task, si perde la visione d'insieme di quanti task eseguono lo stesso tipo di calcolo (precedentemente era il contrario, si riuscivano a capire che task eseguivano il comando ma era difficile seguire quali operazioni faceva il singolo task).
- In fase di rimodellazione non ho possibilità di eseguire verifiche se la traduzione della logica è avvenuta in maniera completa prima della totale rimodellazione del blocco, questo implica la possibilità di "perdere" parti di logica che però verranno evidenziate durante la verifica dell'isofunzionalità.
- Il numero di variabili presenti nel modello è probabile aumenti, in quanto la logica a stati viene sostituita dall'utilizzo di variabili "globali".
- La tempistica di simulazione e generazione del codice risulta essere leggermente peggiorata, ma essendo un'indicazione ottenuta dalle simulazioni a pc potrebbe fornire risultati differenti nel caso del target reale.

- La lunghezza del codice prodotto risulta essere uguale o minore. Al contrario dell'indicazione dei tempi di simulazione, la riduzione della lunghezza di codice è sicuramente indicativa di un miglioramento sul target reale in quanto si riduce la memoria occupata dal SW.