UNIVERSITÀ POLMITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Corso di laurea magistrale in
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE
Dipartimento di ingegneria dell'informazione

MODELLAZIONE DI CONTESTI INDUSTRIALI MEDIANTE LINGUAGGIO AADL

Modellation of industrial contexts using AADL language

Relatore:
Prof. David **SCARADOZZI**

Laureando
Luca **PINCINI**

Correlatori:
*Dott. Nicolò **CIUCCOLI***
*Sig. Giuseppe **CINA'***

Anno accademico 2020-2021

Parte 2

RINGRAZIAMENTI

Questa tesi nasce dalla volontà di conoscere e imparare a sfruttare un nuovo strumento per la modellazione di sistemi e la loro analisi, sistemi semplificati oppure complessi e realistici.

Civitanavi System aveva come obiettivo quello di capire se AADL potesse essere un valido strumento da introdurre nel loro sistema di progettazione, ha avviato così una collaborazione col Professor Scaradozzi che ci ha proposto AADL come argomento di tirocinio e tesi.

Un grande ringraziamento va a David Scaradozzi per averci proposto questo lavoro che oltre ad averci aperto le porte ad un argomento incredibilmente utile e potente, ci ha dato l'opportunità di affacciarci su una realtà aziendale e lavorativa.

Un grande ringraziamento va a Civitanavi System per averci fornito il materiale relativo ad una loro IMU sulla quale applicare gli studi fatti su AADL, dandoci così la possibilità di modellarla e analizzarla.

Civitanavi deve essere ringraziata anche per averci dato la possibilità di visitare l'azienda e conoscere di persona una parte del loro organico, che ci hanno affiancato durante gli 8 mesi in cui si è svolto il lavoro.

I membri dell'organico di Civitanavi che ci hanno seguito e guidato sono Giuseppe Cinà e Dario Donati.

Un imprescindibile ringraziamento va proprio a Giuseppe Cinà che ci ha seguito fin dal primo giorno, restando sempre disponibile per qualsiasi informazione, fornendoci il materiale necessario rimanendo sempre comprensivo nei momenti in cui il lavoro veniva sospeso per portare avanti gli esami universitari rimanenti. Una persona incredibile dalle grandi conoscenze che ci ha fornito un supporto essenziale.

Ringraziamo Dario Donati per il supporto fornitoci in fase di analisi, procurandoci i dati utili a modellare il sistema in AADL.

Infine, ma non per importanza, un ringraziamento va a Veronica Bartolucci e Niccolò Ciuccoli per il supporto fornito durante tutto il percorso e in particolar modo durante gli esami.

Un ringraziamento, inoltre, va alla famiglia e agli amici sempre presenti in questo periodo.

Grazie.

INDICE

Introduzione

In fase di sviluppo, è chiaro che si venga a creare un gap tra il linguaggio di programmazione e il linguaggio macchina. Occorre dunque, attraverso differenti tecniche, costruire un ponte che renda possibile la comunicazione tra questi due differenti tipi di linguaggi.

Negli anni, gli ingegneri hanno sviluppato nuovi linguaggi che avvicinano il linguaggio macchina al linguaggio di programmazione, in modo da rendere più chiaro il funzionamento. Nella maggior parte dei casi, più si astrae il linguaggio, più facile risulta analizzare, capire e sviluppare programmi senza errori. Un alto livello di astrazione è utile per focalizzarsi su ciò che è realmente importante, il modello è un alto livello di astrazione del programma sulla quale è possibile eseguire studi preliminari dettagliati. Questo approccio è noto come Model Based Engineering (MBE), infatti il modello è al centro delle attività ingegneristiche, offrendo supporto in particolar modo ad analisi e verifica.

Un appropriato linguaggio di modellazione fornisce le tecniche necessarie di sintassi e semantica per descrivere correttamente e progettare gli aspetti chiave del sistema (concorrenza, organizzazione, modularità, scheduling…). Il livello di astrazione, così come la semantica del linguaggio sono fattori chiave di quali analisi possono essere eseguite tramite il linguaggio.

Il modello, va intese come astrazione del sistema che deve essere usato nella fase di progettazione per analizzare la correttezza del lavoro svolto e impostato. Utilizzando il modello e gli strumenti di analisi, è possibile scoprire in anticipo problemi che potrebbero, altrimenti, venire fuori in fasi successive. Ciò comporta una significativa riduzione di tempi e costi di sviluppo. Infatti, in questo modo, si evita di dover tornare alle fasi iniziali di sviluppo ogni volta che sorge o viene scoperto un nuovo problema. Ovviamente, condizione necessaria per far si che tutto funzioni, è quello di sviluppare un modello corretto e coerente con il lavoro da svolgere, ossia un modello che possa poi essere validato.

Al fine di avere un buon sviluppo MBE occorre:

- Una semantica ricca e precisa;
- Notazione grafica: una rappresentazione grafica risulta facile da capire e può essere riutilizzata per creare della documentazione;
- User_friendly tool: un buon tool può offrire esperienze di utilizzo migliori all'operatore e può cambiare radicalmente il modo in cui viene visto il modello.
- Un formato aperto e interoperabile: in questo modo cambiando tool è possibile riutilizzare e riadattare lo stesso modello.

Esistono già differenti tipi di linguaggi di modellazione e tools che si distinguono principalmente per il tipo di approccio utilizzato: quello funzionale e quello architetturale. Il modello funzionale cerca di rispondere alla domanda "cosa" (cosa fa il sistema?), mentre un linguaggio architetturale cerca di rispondere alla domanda "come" (come il sistema svolge e gestisce una determinata operazione). AADL rientra nella seconda casistica, ossia quella dei linguaggi architetturali.

Con "software architecture" ci si riferisce alla struttura fondamentale di un sistema software, ossia la disciplina di creare sia la struttura, che la documentazione della struttura stessa. Ogni struttura è composta da elementi software, relazionati tra loro, e proprietà sia degli elementi sia delle relazioni. Dunque, l'architettura software è in grado di mostrare come il sistema è organizzato, quali elementi sono connessi e quali parti sono state usate per realizzarlo.

Tra i linguaggi di tipo architetturale rientra AADL (Architecture Analysis and Design Language) con l'obiettivo del real time, per i sistemi embedded. Questo linguaggio si focalizza sul design di sistema usando un linguaggio ricco, semanticamente formale che può essere utilizzato per analizzare e generare il sistema. Tramite AADL, una volta modellato il sistema, è possibile analizzarlo, generare l'implementazione e derivare i vari test. Inoltre, attraverso l'analisi, la simulazione e una prototipazione rapida è possibile affinare il modello ed eventualmente procedere con una nuova implementazione.

Dunque, l'obiettivo che ci si è posti è quello di riuscire a creare un modello del sistema che possa essere analizzato ed ampliato garantendo il flow dei requisiti di sistema a Hardware e Software. Si procede con l'utilizzo di AADL in quanto:

- I sistemi che si progettano sono sempre più complessi e difficili da analizzare. Infatti, nonostante le normative applicabili agli sviluppi Hardware e Software siano rigorose, la fase di integrazione del sistema resta sempre complicata.
- Scoprire le anomalie o marginalità a valle dell'implementazione porta ritardi ed extracosti spesso non sostenibili.
- Validare l'architettura di un sistema safety critical richiede lo svolgimento di analisi dedicate che necessitano di una visione chiara del sistema.
- La disponibilità di un modello virtuale del sistema permette di effettuare analisi comparative, modificando in tempo reale il modello e verificarne il comportamento.
- La possibilità di procedere per livelli di dettaglio fa si che la verifica possa essere incrementale.
- Garantire un approccio uniforme, strutturato e coordinato al design di Sistema, Hardware e Software

Al di là dei vantaggi immediati derivanti dalla disponibilità di un gemello virtuale (Digital Twin) del sistema sul quale effettuare analisi e sperimentazioni, a costi contenuti sin dalle fasi iniziali del progetto, vi sono altri vantaggi derivanti dall'utilizzo di AADL nella progettazione del sistema:

- La possibilità di supportare le analisi di Safety – FHA (Functional Hazard Analysis), FMEA (Failure Mode and Effects Analysis), FTA (Fautl Tree Analysis), MTBF (Mean Time Between Failures) - documentando opportunamente il modello.
- Nel caso di modifiche limitate ad un sistema già modellato la possibilità di verificarne anticipatamente l'impatto a livello di sistema.
- Fornire input a tool di simulazione e di generazione automatica del codice.
- Svolge uno sviluppo 'Architecture driven', creando un percorso lineare ed omogeneo dalla fase di progettazione di sistema sino all'integrazione.
- Uso di una sintassi standardizzata e precisa per descrive il Sistema.
- Tracciare le evoluzioni del modello del Sistema lungo l'arco del Progetto.

L'utilizzo di AADL per Civitanavi rappresenta un passo avanti nel miglioramento e nell'efficientamento dei processi aziendali. La richiesta sempre più frequente di sistemi safety critical impone l'aderenza dei processi di sviluppo – Sistema, Hardware e Software- a standard stringenti, onerosi da soddisfare e da mantenere. Dunque, AADL costituisce un aspetto di innovazione:

- AADL è già utilizzato in ambito industriale, spazio ed avionico e quindi è un tool già noto.
- Promuove un concetto di Virtual Integration, dove il sistema viene modellizzato e verificato virtualmente prima di essere realizzato fisicamente.
- Facilitare l'applicazione di modifiche a sistemi già realizzati analizzandone gli impatti prima sul modello virtuale.

- AADL offre un percorso verso tool di modellazione di sistema e progettazione del software che possono rappresentare un'ulteriore evoluzione dei processi aziendali nel raccordare la progettazione di sistema con il design software.
- Favorire la creazione di un processo lineare ed omogeneo che parte dalla progettazione di sistema, arriva al design hardware/software passando attraverso la safety, garantendo sempre la massima visibilità sull'evoluzione del design, e consentendo di confrontarlo continuamente con requisiti applicabili.

Dunque, l'obiettivo che ci si prefigge è:

- quello di effettuare la modellazione di una IMU già sviluppata da Civitanavi arrivando sino al livello di dettaglio necessario per effettuare le analisi di timing e safety.
- Svolgere sul modello le analisi funzionali e di safety valutando i risultati con quelli già attenuti con metodologia 'classica'.
- Identificare un processo che consenta ai vari enti aziendali coinvolti nella progettazione di avere un ruolo attivo nello sviluppo del sistema, definendo la metodologia da adottarsi ed il livello di coinvolgimento di ogni singolo ente.

Una volta completata la fase di modellazione in AADL dell'IMU, il passo successivo è stato quello di andare a svolgere delle analisi specifiche di Latency e di Safety.

L'analisi della latenza viene eseguita su modelli AADL che includono flussi end-to-end e calcola la latenza minima e massima tenendo conto di un'ampia gamma di contributori di latenza.
Il calcolo della latenza si basa sui contributori assegnati ai diversi elementi architettonici e alle informazioni di progettazione, man mano che i progetti si evolvono. Il modello AADL può variare da un'architettura funzionale con budget di latenza a diversi livelli di scomposizione, a un'architettura di attività e comunicazione con velocità di esecuzione mappate su una piattaforma hardware che supporta il partizionamento. La fedeltà dell'analisi è determinata dai dettagli nel modello AADL.
I risultati vengono riportati in un formato di risultati comune e in un formato di foglio di calcolo insieme ai dettagli di ciascun contributore alla latenza. L'analisi della latenza può essere parametrizzata da alcune impostazioni delle preferenze che consentono agli utenti di esplorare le variazioni architettoniche senza dover modificare i dettagli nel modello (ad esempio se il sistema si comporta come un sistema asincrono o sincrono).

Nell'analisi di safety invece andiamo a vedere come si può eseguire un'analisi FHA (functional hazard assessment), FMEA (ex. fault impact analysis), FTA (fault tree analysis) di un sistema, tramite l'uso di AADL, Error Model V2 (EMV2) e OSATE.

Vedremo come AADL e EMV2 possono essere utilizzati in questo processo e dimostreremo l'uso di EMV2 a tre livelli di astrazione:

1. Propagazioni e flussi di errori: tramite un'analisi dell'impatto (FMEA) tipicamente da un'unica fonte, procedendo in avanti, si traccia l'impatto degli errori; un'analisi di propagazione all'indietro per identificare tutti i potenziali contributi a un evento catastrofico o importante.

2. Specifiche del comportamento degli errori dei componenti: identificazione delle modalità di guasto, tipi di guasti dei componenti (eventi di errore), logica del comportamento degli errori per riflettere la ridondanza dell'input esterno e la ridondanza dei sottocomponenti.

3. Generazione di un albero degli errori FTA con probabilità del guasto utile per determinare l'affidabilità iniziale del sistema. Può essere utilizzato per generare un albero dei guasti delle parti COMPOSITE.

Nei capitoli successivi verrà sintetizzato il lavoro svolto nella creazione del modello AADL del sistema fornito da Civitanavi. Nel primo capitolo verrà introdotto AADL per fornire delle nozioni base, utili alla comprensione dei capitoli successivi.

Buona lettura.

Capitolo 1: AADL

1.0 Introduzione al capitolo

In questo primo capitolo viene introdotto AADL e come modellare un sistema con esso. AADL è prima di tutto un linguaggio descrittivo di tipo testuale che contiene anche una notazione grafica per facilitare la comprensione delle architetture descritte. Vengono spiegate la semantica e la sintassi base, i costrutti fondamentali e la logica di AADL, elementi che lo rendono adatto alla descrizione di qualunque sistema embedded real time. Verranno illustrati gli elementi utilizzati per modellare il sistema IMU di Civitanavi e tutto ciò che è necessario modellare per poter eseguire in fase finale le Analisi volute.

1.1 Astrazioni del linguaggio AADL

In AADL i componenti sono definiti tramite la dichiarazione “Type” e “Implementation”.
Una dichiarazione **component type** definisce gli elementi dell'interfaccia e gli attributi osservabili esternamente (cioè caratteristiche che sono punti di interazione con altri componenti, specifiche di flusso e valori delle proprietà interne).
La **component implementation** definisce la struttura interna di un componente in termini di sottocomponenti, connessioni di sottocomponenti, sequenze di chiamate sottoprogramma, modalità, implementazioni e proprietà dei flussi.

Sono definibili anche set di proprietà e librerie di allegati che consentono a una finestra di progettazione di estendere il linguaggio e personalizzare una specifica AADL per soddisfare specifiche di progetto richieste.

1.2 Componenti

I componenti formano il vocabolario centrale di modellazione per l'AADL, devono avere un identificatore univoco (nome) e vengono dichiarati come type e implementation specificando, tramite un altro identificatore, la categoria a cui il componente appartiene.

I componenti sono raggruppati in tre categorie distinte (figura 1): *Hardware, Software e Composite.*

1. I componenti **Hardware** hanno una proprietà fisica, qualcosa che puoi toccare fisicamente o vedere:

    a. La categoria *Device* rappresenta un dispositivo, come un allarme o un sensore.
    b. La categoria *Processor* rappresenta un'unità di elaborazione che esegue il codice.
    c. La categoria *BUS* rappresenta una connessione fisica tra entità, ad esempio il bus Ethernet che connette i server in un data center o un bus che permette la comunicazione tra più processori.
    d. La categoria di *Memory* caratterizza una memoria hardware su un chip o sulla scheda madre

e. La categoria *Virtual BUS* rappresenta un protocollo utilizzato per la connessione logica tra componenti software.
f. La categoria *Virtual Processor* è una rappresentazione software di un elemento Hardware.

2. I componenti **Software** rappresentano artefatti non fisici:

    a. La categoria *Data* acquisisce un tipo di dati con le relative caratteristiche, ad esempio la dimensione, la clausola di rappresentazione o l'enconding. Un componente dati AADL può essere utilizzato in modi diversi: se associato a un'interfaccia componente, rappresenta il tipo di dati utilizzato su tale interfaccia. Se associato a un sottocomponente data, rappresenta il tipo di dati associato a una memoria condivisa.
    b. La categoria *Subprogram* specifica un blocco di codice eseguibile con la sua interfaccia.
    c. La categoria *Thread* rappresenta un'entità che esegue un flusso di istruzioni. Un thread è caratterizzato dai vincoli di temporizzazione e dal modo in cui viene eseguito.
    d. La categoria *Thread Group* rappresenta una collezione logica di thread contenuti nei processi.
    e. La categoria *Process* è la rappresentazione di uno spazio di indirizzi che contiene dati e sottocomponenti (thread, thread group…) contenenti gli elementi necessari per eseguire il software. Il processo è associato a un processore e a una memoria.

3. I **Composite**, componenti ibridi, non rappresentano nulla di concreto, ma sono costrutti convenienti per specificare un componente la cui categoria può essere definita in un secondo momento o per descrivere un sistema articolato, formato da più elementi delle precedenti due categorie.

    a. La categoria *Abstract* viene utilizzata per rappresentare componenti la cui categoria non è ancora specificata. Tale categoria è utile nella fase iniziale della progettazione del sistema, quando i requisiti non sono ancora completamente specificati e non è stata fatta alcuna scelta di implementazione. I progettisti possono specificare le interfacce esterne e le caratteristiche dei componenti e successivamente decidere la categoria appropriata.
    b. La categoria di *System* viene utilizzata per racchiudere i componenti, è un contenitore che raggruppa e configura i componenti necessari per implementare il sistema. I componenti di sistema AADL possono anche essere utilizzati per organizzare l'implementazione in modo gerarchico con i sottosistemi.

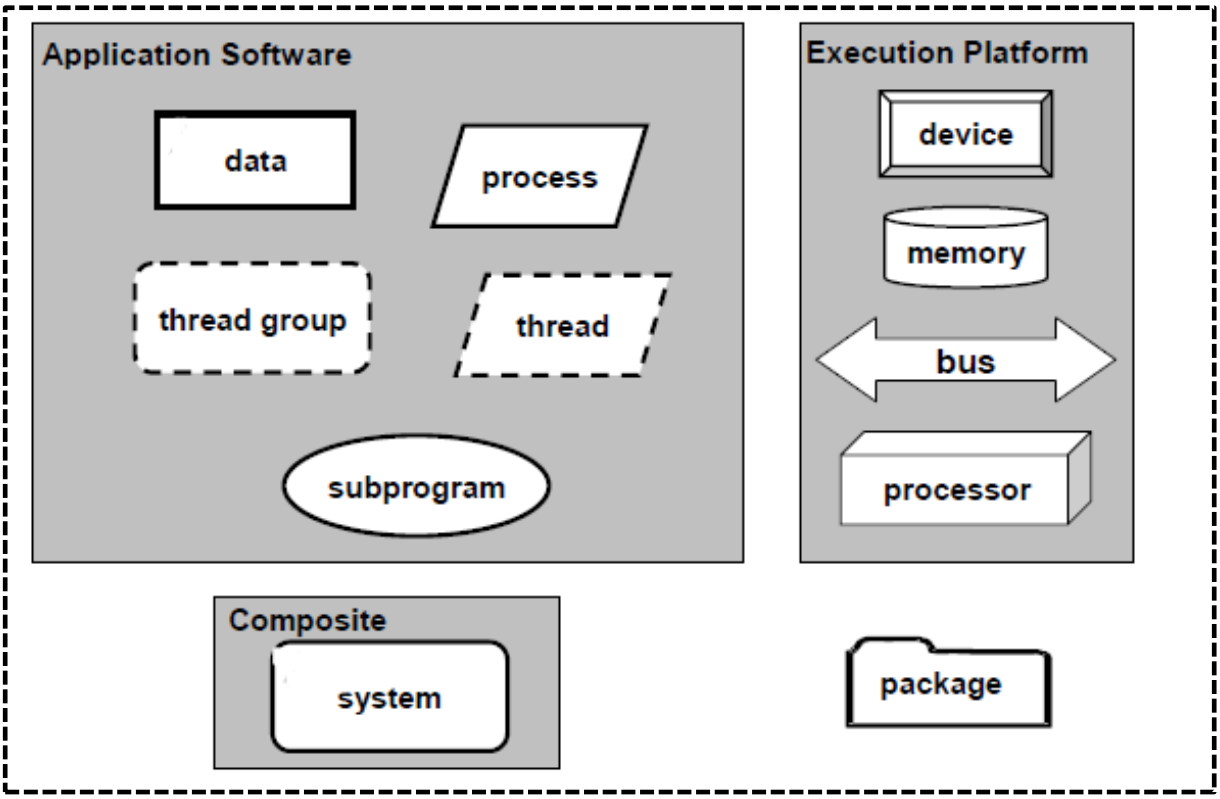


Figura 1 Suddivisione logica e grafica in AADL

1.3 Definizione componente

Un modello AADL, come specificato in precedenza, è definito tramite la *Type* e l'*Implementation*:

- La **type** è la rappresentazione esterna del componente. Definisce la categoria dei componenti e le sue interfacce esterne e mostra come comunica con il resto del mondo.
- L'**implementation** del componente definisce l'interno del componente: come le interfacce sono collegate ai sottocomponenti per fornire il servizio

Un tipo di componente può avere più implementazioni. Dopotutto, lo stesso servizio (tipo di componente) può essere implementato in modi diversi (implementazione dei componenti). Questo è un ottimo approccio quando si progetta un'architettura: la *component type* definisce il modo in cui viene utilizzato il componente e la *component implementation* può essere scelta quando si integrano i componenti insieme.

1.4 Component TYPE

In AADL una *component type* è caratterizzata da:

- Una categoria (obbligatoria): dispositivo, sistema, processore, processo, ecc.
- Identificatore univoco per identificare il componente (nome)
- Features (facoltativo) che definiscono le interfacce dei componenti disponibili per il mondo esterno

- Flussi (facoltativo) vengono specificati nel tipo di componente mostrando il modo in cui i dati fluiscono nelle interfacce esterne (features). Il flusso viene successivamente perfezionato nell'implementazione del componente e descritto più avanti in questa sezione.
- Proprietà (facoltative) che definiscono caratteristiche specifiche del componente e/o delle sue parti.
- Allegati (annex) AADL (facoltativi) che aumentano la descrizione del componente con altri linguaggi e librerie (es. EMV2 per la gestione degli errori).

Figura 2 Component Type

1.5 Component Implementation

La *component implementation* specifica una struttura interna in termini di sottocomponenti, interazioni (chiamate e connessioni) tra le funzionalità di tali sottocomponenti, passaggio di dati attraverso una sequenza di sottocomponenti e proprietà.

In AADL una *component implementation* è definita da:

- Categoria di componenti che specifica la categoria implementata. È la stessa cosa del tipo di componente implementato.
- Identificatore univoco per l'utilizzo e il riferimento all'implementazione del componente.
- Sottocomponenti (facoltativi) che definiscono il contenuto del componente: elementi contenuti nell'implementazione per fornire i servizi richiesti.
- Connessioni (facoltative) che definiscono il modo in cui le *features* del componente (definite nel tipo) sono collegate alle *features* dei sottocomponenti.
  Le connessioni mostrano come le interfacce esterne interagiscono con le interfacce situate all'interno del componente.
- Flussi, Proprietà e Allegati: come nella *component type.*

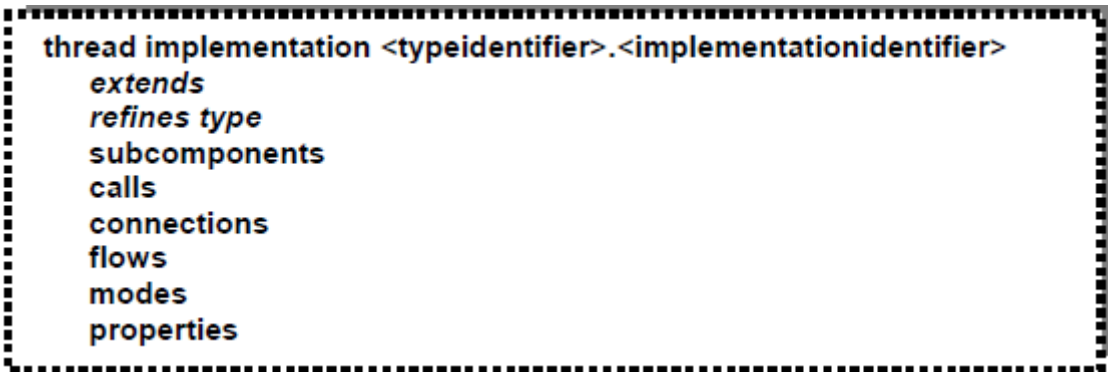


Figura 3 Component Implementation

```
125  ----------USER INTERFACE MAIN CONNECTOR----------------------------------------
126  device Main_Connector
127      features
128          CBIT: in data port [redacted]:Data_Types::FLOAT32_TYPE_AADL;
129          SYNC: out data port [redacted]:Data_Types::FLOAT32_TYPE_AADL;
130          RAW_Data: in data port [redacted]::Data_types::Message_ID_0x02;
131          SDLC_Data: in out data port [redacted]:Data_types::Message_ID_0x02;
132          UART_Data: in out data port [redacted]::Data_types::Message_ID_0x02;
133          V28_IN: in out event port;
134          MAINT: out event port;
135
136      flows
137          F1: flow sink RAW_Data;
138          F2: flow sink SDLC_Data;
139          F3: flow source MAINT;
140          F4: flow sink UART_Data;
141  end Main_Connector;
142
143  device implementation Main_Connector.impl
144
145  end Main_Connector.impl;
```

Figura 4 Rappresentazione di un dispositivo in AADL

1.6 Features

Le interfacce AADL mostrano ciò che il componente espone al mondo esterno, fisicamente o logicamente. Le caratteristiche (features) sono classificate in due categorie: *port* e *access*.

- La **Porta** rappresenta il punto di interazione per un evento, un trasferimento di dati o una combinazione di entrambi. È possibile specificare le porte per i componenti seguenti: *device*, *thread*, *process* e *system*. Esistono tre tipi di porte:

    - La **Event Port** rappresenta l'emissione/ricezione di una notifica e trasporta un segnale senza alcun dato.
    - La **Data Port** (scambio di dati) rappresenta un'interfaccia che riceve un flusso di dati e mantiene sempre i valori più recenti. Il valore della porta viene aggiornato non appena viene ricevuta una nuova istanza dei dati.

- La **Event-Data Port** (scambio di dati con notifica) combina la porta eventi e la porta dati nella stessa entità. La *Event-Data Port* trasporta un segnale (evento) associato ai dati.

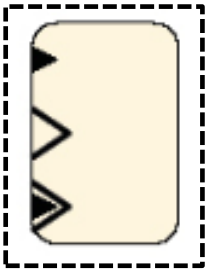


Figura 5 Rappresentazione grafica delle tipologie di porte

Le porte sono caratterizzate da una direzione:

- Porta in ingresso (**In Port**)
- Porta in uscita (**Out Port**)
- Porta bidirezionale (**In-Out Port**)

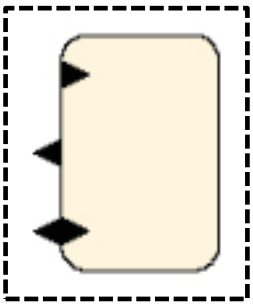


Figura 6 Rappresentazione grafica delle direzioni delle porte

Le porte possono essere raggruppate, anche se di tipi diversi, andato a formare i gruppi di porte.

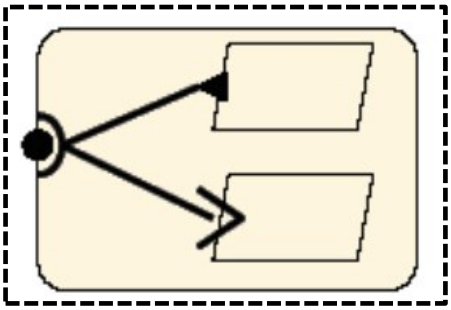


Figura 7 Port Group collegato in ingresso ad una data port e in uscita ad una event port

- **L'accesso** ai componenti specifica se un componente richiede o fornisce l'accesso a un determinato servizio.
  Esistono due tipi di accesso: **Bus Access** e **Data Access** e due tipi di direzione: *requires access* e *provides access*.

  Con il *Bus Access* i vari componenti che si interfacciano con il bus, possono essere resi accessibili ad altri componenti. I componenti possono richiedere di accedere a dei bus esterni, mentre se il bus è interno al componente, allora il componente fornisce l'accesso al bus ad altri componenti. Per distinguere questi due tipologie di accesso, si utilizzano gli attributi **requires** e **provides**

1.7 Packages, Property Sets, Annexes

Con *Property Sets* si intende un gruppo con un identificatore univoco (nome), costituito dalle tipologie, definizioni e le costanti che servono per completare le proprietà.
A queste proprietà viene fatto riferimento utilizzando il nome specifico del set e possono essere associate a componenti e altri elementi di modellazione (ad esempio porte o connessioni) all'interno di una specifica di sistema. La loro dichiarazione e il loro uso diventano parte della specifica.

Con *Annexes*, invece, si intende un'estensione del linguaggio di AADL con altri linguaggi e librerie, consentendo così l'incorporazione di notazioni aggiuntive e specifiche.

I *Package*, a loro volta, consentono di organizzare i tipi di componenti, implementazioni di componenti, tipi di gruppi di funzionalità e librerie di allegati in set di dichiarazioni.
I nomi dei pacchetti vengono costruiti utilizzando identificatori separati da due punti ("::").
I Package possono avere sezioni pubbliche e private. Nel caso di sezioni pubbliche, le componenti interne al pacchetto sono accessibili per altri pacchetti, mentre nel caso di sezioni private possono fare riferimento solo alle componenti private dello stesso pacchetto.

La visibilità di altri pacchetti è specificata da una dichiarazione **with**, questo operatore ha lo stesso significato dell'operatore **include** in C++, permette di utilizzare gli elementi di un Package richiamandoli all'interno di un altro Package. È possibile fare solo riferimento agli elementi interni dei pacchetti elencati nella dichiarazione **with**. Analogamente, lo stesso discorso può essere esteso ai **Property Set**

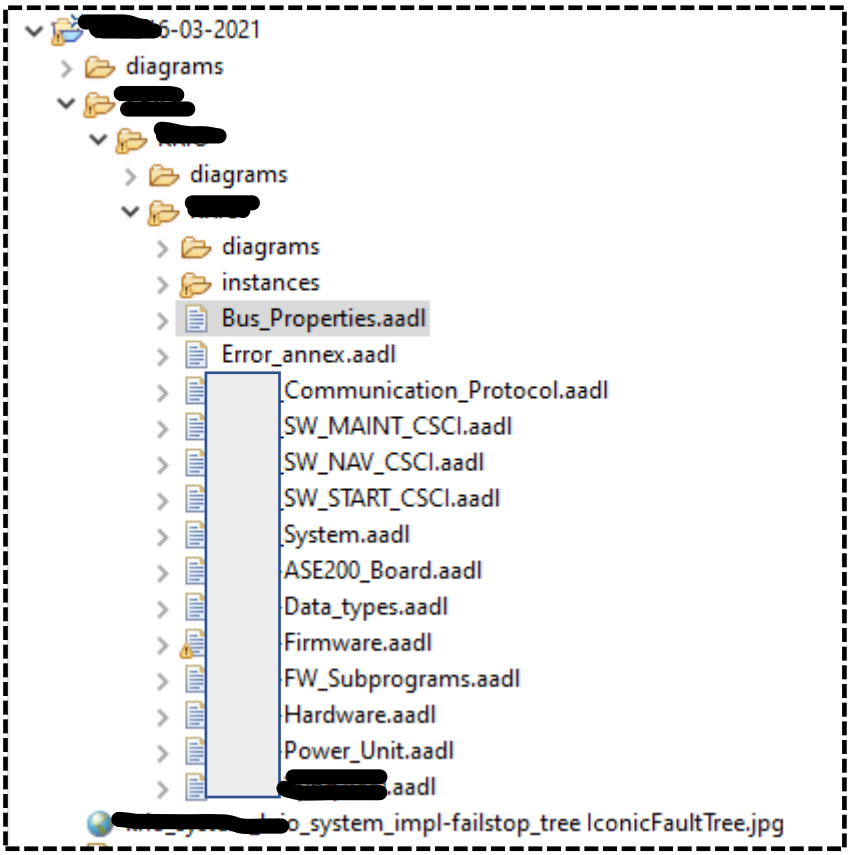


Figura 8 Cartella project e elenco dei Packages del modello del sistema

1.8 Flow

In AADL si fa riferimento al concetto di flusso. Con *Flow* si definisce un percorso che viene utilizzato per il trasporto dei dati all'interno del nostro modello: dalla loro creazione al loro utilizzo. Si viene dunque a dichiarare una catena di elementi che creano, trasportano, elaborano e consumano il dato. Nella specifica del flusso troviamo: una sorgente di flusso (il produttore), zero o più percorsi (trasportatori) e uno o più destinatari (consumatori).
La specifica del flusso consente di vedere dove passano i dati attraverso l'architettura di sistema.

I flussi vengono specificati nella rappresentazione testuale, in una sezione di componenti dedicata, attraverso:

- "Nome univoco" per distinguere il flusso all'interno del componente e la sua implementazione. Il nome dell'implementazione del flusso (in un *implementation*) deve essere uguale al nome della specifica di flusso (in una *type*) e mostrare come viene implementato il flusso.
- "Tipo" che spiega il flusso di dati all'interno di questo componente.

  Esistono tre tipi di flussi:

  - **flow source**: specifica l'origine di un flusso.
  - **flow sink**: specifica la terminazione di un flusso.
  - **flow path**: specifica l'attraversamento di un componente.

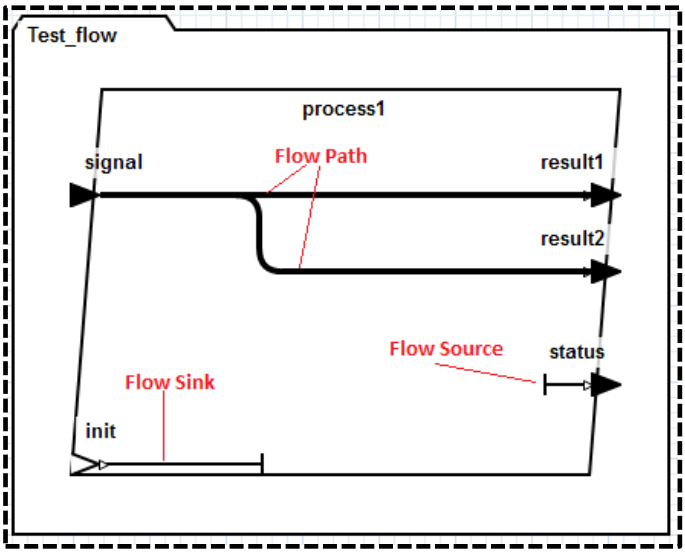


Figura 9 Rappresentazione grafica AADL dei flussi

- "Riferimento ad una feature": uno (per il *flow source* e *flow sink*) o due (per il *flow path*). Il flusso serve per mostrare i percorsi logici dei dati attraverso le varie porte.

I flussi sono rappresentati tramite: specifiche del flusso, implementazione del flusso e dichiarazioni di flusso **end-to-end**.

1.9 Dichiarazione di un flusso

Attraverso la dichiarazione di un flusso all'interno di una componente, si va a specificare come il percorso del flusso è visibile dall'esterno attraverso le porte, i gruppi porte e parametri. Un flusso che ha origine da un componente prende il nome di **flow source**. Un flusso che termina in un componente prende il nome di **flow sink**. Un flusso che attraversa un componente da una porta ad un'altra prende il nome di **flow path.**

1.10 Implementazione del flusso

Attraverso la *flow implementation* si dettaglia il percorso del flusso definito nella *type*, ossia come una **sequenza di *flow*** che attraversa le varie sottocomponenti, lungo l'intero percorso, dall'origine del flusso fino al suo esaurimento.

Le porte identificate dalla specifica del flusso non devono necessariamente avere lo stesso tipo di dati né devono essere lo stesso tipo. Ciò consente di utilizzare le specifiche del flusso per descrivere flussi logici di informazioni.

1.11 Flussi end-to-end

Un flusso end-to-end è un flusso logico attraverso una sequenza di componenti di sistema (*thread*, *device* e *processor*). Questa tipologia di flusso può essere dichiarata mediante la grammatica ***END-TO-END*** sia nella *component type*, sia nella *component implementation*.
Si noti che un flusso end-to-end è espresso in termini di specifiche di flusso dei suoi sottocomponenti. Di conseguenza, possiamo eseguire un'analisi dei flussi più superficiale, quelli visti dall'esterno specificati nelle *type*, oppure un'analisi più dettagliata tenendo in considerazione tutte le specifiche dei flussi dei sottocomponenti definite nelle *implementation*.

Maggiori dettagli sull'utilizzo dei flussi end-to-end e sulla loro implementazione verranno forniti nei capitoli successivi, quando si tratterà l'analisi di latenza.

In sintesi, i flussi end-to-end permettono di modellare in maniera rapida ed agevole la nascita, il percorso e la morte di un dato, di un qualsiasi sistema.

CAPITOLO 2: Modello AADL del sistema inerziale

2.0 Introduzione al Capitolo

In questa sezione verrà spiegato nel dettaglio come si è passati dal modello reale, descritto sulla documentazione PDF del IMU, al modello AADL. Tramite la Grammatica di AADL è stato costruito un modello del Sistema Inerziale abbastanza fedele da permettere delle analisi preliminari molto interessanti, che permettono un giudizio del sistema prima della realizzazione fisica. In questo modo si ha la possibilità di valutare l'efficienza e la sicurezza in anticipo permettendo al sistemista di modellare le specifiche necessarie direttamente in AADL.
L'ambiente di sviluppo OSATE2 fornisce la possibilità di modellare in forma di Codice, creando in automatico una versione sottoforma di Diagramma, con possibilità di sfruttare i comandi di disegno e disegnare il modello in forma di schema a blocchi (generando in automatico la relativa versione codice), questo grazie al fatto che AADL come linguaggio è pensato per trasformare in righe di codice analizzabili, gli Schemi a Blocchi descrittivi dell'intero sistema (con le relative connessioni, sottosistemi, processori, HW & SW, etc…).

2.1 Modello del sistema

Come prima cosa serve una visione complessiva del sistema ad Alto Livello in modo da poter vedere come il sistema è suddiviso in MACRO Sezioni e come sono collegate fra di loro. La modellazione in AADL avviene iniziando dalla creazione del package **IMU_System** che conterrà la *Component Type* e la *Component Implementation* del nostro sistema.
Entrambe queste dichiarazioni sono una rappresentazione del sistema, la **type** rappresenta il sistema visto dall'esterno, a scatola chiusa, l'**implementation** rappresenta il sistema visto dall'interno e quindi include i sottocomponenti che formano il sistema. A questo punto si considerano gli elementi **interni,** vengono modellati i singoli sottocomponenti creando per ognuno di essi un package che contenga le relative dichiarazioni.

In questo caso sono stati creati dei package personalizzati, come ad esempio il package **IMU::Hardware** che contiene sia la dichiarazione della **scheda principale** del sistema, sia le dichiarazioni di tutti gli elementi della parte Ottica e la sensoristica (compresi i relativi sottocomponenti). Bisogna creare una suddivisione degli elementi del sistema nelle classi fornite da AADL dividendo così gli elementi nelle tre macrosezioni: **Composite**, **Hardware**, **Software**.

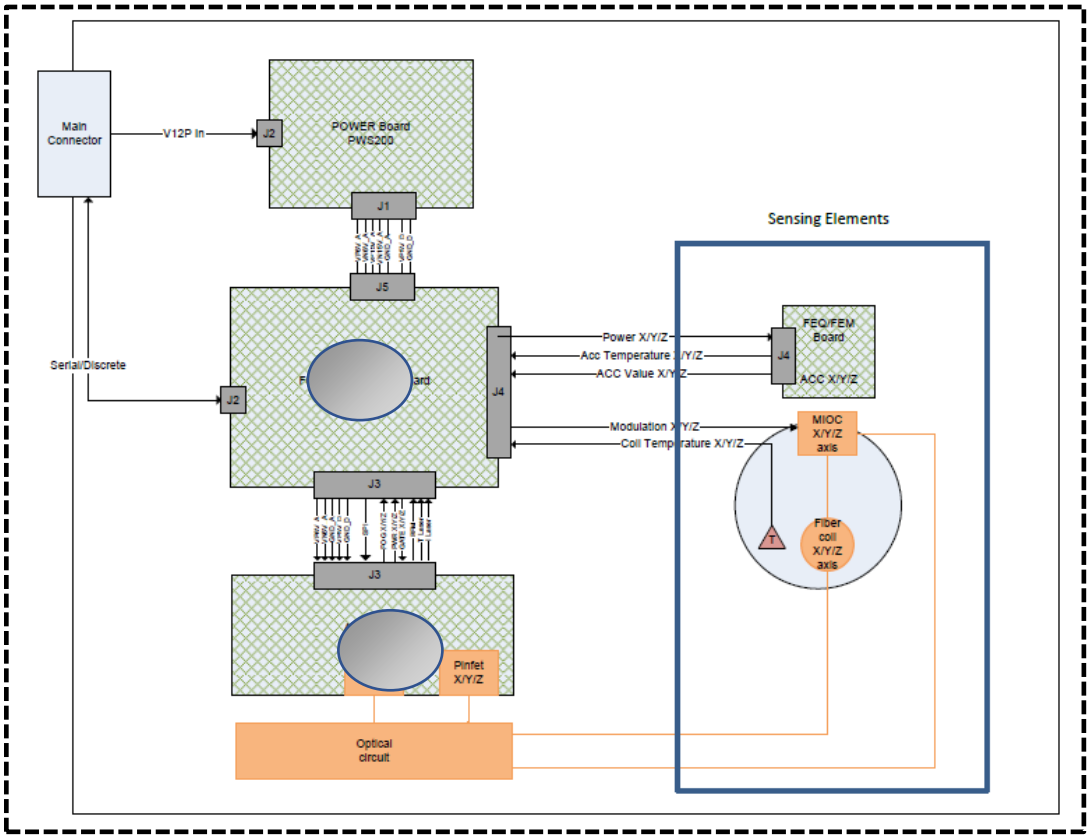


Figura 10 Diagramma a blocchi del sistema

```
system [redacted]

end [redacted]

system implementation [redacted]
      subcomponents

            FAM: system [redacted]:Hardware:[redacted]FAM300.impl;
            ASE200: system [redacted]::ASE200_Board::[redacted]ASE_Board.impl;
            Sensing_Unit: system [redacted]:Hardware::Sensing_Elements.impl;
            Optical_unit: device [redacted]ASE200_Board::Optical_Circuit.impl;
            POWER_Board: system [redacted]Power_Unit::Power_Board.impl;

            MAIN_CONNECTOR: device Main_Connector;

            FAM_Type: system [redacted]:Hardware:[redacted]FAM300;
            ASE200_Type: system [redacted]::ASE200_Board:[redacted]ASE_Board;
            Sensing_Unit_Type: system [redacted]Hardware::Sensing_Elements;
            Optical_unit_Type: device [redacted]ASE200_Board::Optical_Circuit;
            POWER_Board_Type: system [redacted]:Power_Unit::Power_Board;

            SPI1: bus [redacted]:Communication_Protocol::SPI;
            QSPI: bus [redacted]:Communication_Protocol::QSPI.impl;
            DDR3: bus [redacted]:Communication_Protocol::DDR3.impl;
            UART0: bu [redacted]::Communication_Protocol::UART_0;
            UART: bus [redacted]:Communication_Protocol::UART;
            SDLC: bus [redacted]:Communication_Protocol::SDLC;
```

Figura 11 Implementazione sistema in versione codice di AADL

Ognuno dei sottocomponenti ha il proprio package o è dichiarato nel package di un altro, come detto nei capitoli precedenti, questi package sono linkati nel primario tramite il comando **with**.

SI dichiarano le porte di input/output di ogni componente e le connessioni che sono presenti fra di loro, tra porta e porta. Si ottiene così un diagramma a blocchi primario rappresentativo del sistema (figura 12). Ogni sottocomponente verrà poi modellato nel dettaglio prendendo in considerazione tutti gli elementi al suo interno.

Gli elementi Macro sono 6:

- **FAM**: scheda principale che raccoglie i dati dalla sensoristica li elabora e li restituisce. Presenta al suo interno il system SoC, che a sua volta contiene i moduli CPU e FPGA.
- **ASE**: scheda adibita alla gestione dei dati provenienti dai giroscopi, in unione alla Optical_Unit fornisce i segnali, li riprende in ingresso elaborandoli e inviandoli successivamente alla FAM.
- **POWER_BOARD**: Componente di alimentazione dell'intero sistema, fornisce alimentazione a 15V, 6V e 5V alla FAM prendendo in ingresso 28V, dalla FAM l'alimentazione viene gestita e fornita a tutti gli altri componenti.

- **OPTICAL_UNIT**: Unità che riceve in ingresso il LASER dalla ASE, lo scompone in 3 fasci ognuno diretto ad un giroscopio, riprendendo i segnali in uscita dalle *Fiber Coil* e fornendoli alla ASE come *Pinfet X/Y/Z*.
- **SENSING_UNIT**: Unità che contiene la sensoristica fondamentale della IMU, Accelerometri e Giroscopi con i relativi sensori di temperatura.
- **MAIN CONNECTOR**: device che rappresenta la porta IN/OUT con la quale l'utente si interfaccia con il Sistema. Serve anche a livello semantico come pozzo in cui vanno a finire i dati, rappresentativo dell'utilizzo del dato da parte dell'utente. Al suo interno troveremo definiti i **flow sink** sulle porte in cui arrivano i dati provenienti dal Firmware.

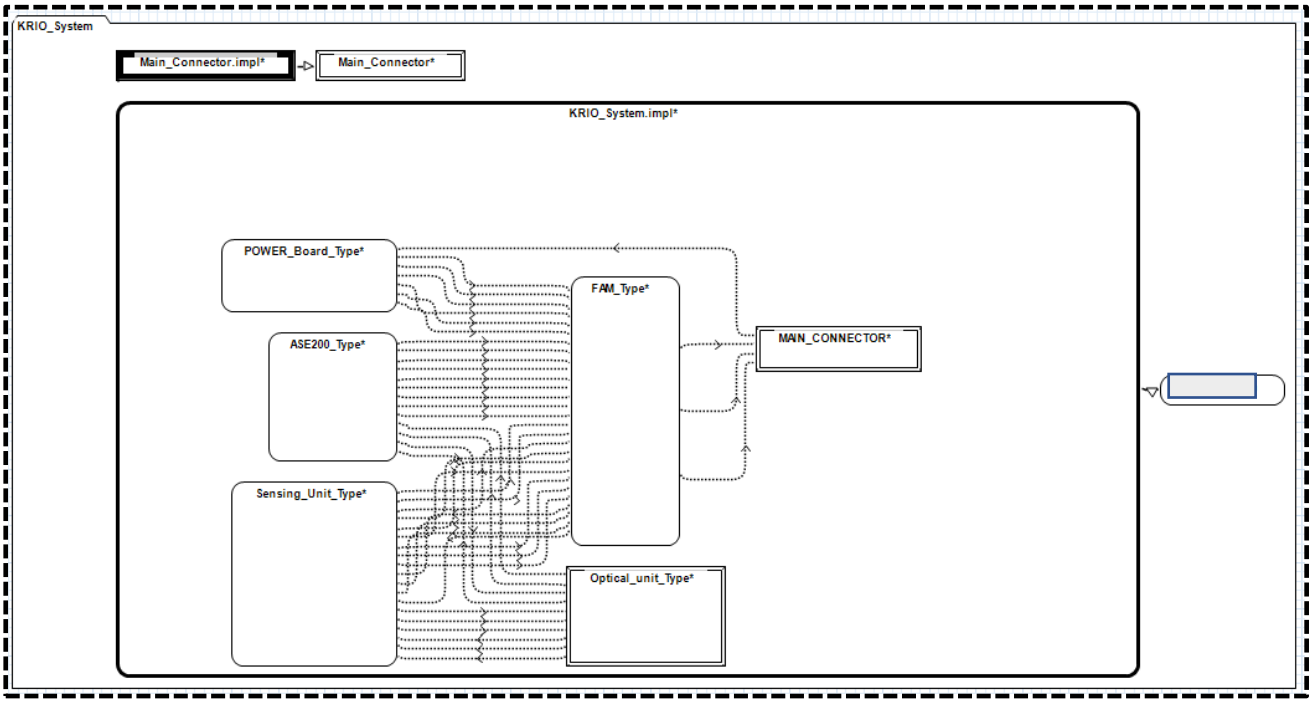


Figura 12 Schema a blocchi del sistema rappresentato con AADL

2.2 Scheda Principale: FAM

Nel package IMU_Hardware viene definita la scheda principale, cuore dell'intero sistema. La FAM è la scheda che fa da intermediario tra la sensoristica e l'utente, contiene il processore diviso nelle sue due componenti CPU e FPGA, i dispositivi di conversione dei segnali dati e le altre parti che permettono il funzionamento del sistema. La FAM viene modellata anch'essa come *composite,* venendo definita come *system* ed è attraversata da tutti i dati generati dal sistema.

Nella figura 13 viene mostrata la component type della FAM, sono mostrati solo alcuni degli elementi principali che sono stati modellati. Il modello, tra type e implementation, prevede oltre alla definizione di porte, connessioni, flussi e sottocomponenti, anche la modellazione degli errori verificabili, la loro tipologia e direzione di propagazione e la creazione della macchina a stati che gestisce gli stati di errore del sistema.

```
system KRIO_FAM300
    features
        SDLC_Data: in out data port [redacted]:Data_types::Message_ID_0x02;
        Accel_Value_x: in data port [redacted]:Data_types::Accell_Value_X;
        Accel_Temp_x: in data port K[redacted]Data_types::Accell_Temp_X;
        FOG_X: in data port KRIO::Da[redacted]pes::Gyro_FOG_X;
        Power_Good_15V: in event port;
        Power_Good_6V: in event port;
        Power_Good_5V: in event port;
        Power_15V: in data port;

    flows
        F1: flow sink Accel_Value_x;
        F13: flow source SDLC_Data;
        KRIO_FAM300_new_flow_spec2: flow source SDLC_Data;
        FProva: flow path Accel_Value_x -> SDLC_Data{latency => 30ms .. 100ms;};
        FProva2: flow path Accel_Temp_x -> SDLC_Data;

    annex EMV2 {**
            use types ErrorLibrary, Error_annex;
            use behavior Error_annex::ThreeState;

            error propagations
                Accel_Temp_x: in propagation {OutOfRange, ItemOmission};
                --...
                --...
            end propagations;

            component error behavior
                --...
                --...
            end component;

        properties
            emv2::OccurrenceDistribution => [ProbabilityValue => 3.01e-5;
                                             Distribution => Poisson;] applies to Failure;
            emv2::OccurrenceDistribution => [ProbabilityValue => 2.0e-7;
                                             Distribution => Poisson;] applies to Accel_Temp_x.OutOfRange;
            emv2::OccurrenceDistribution => [ProbabilityValue => 2.0e-9;
                                             Distribution => Poisson;] applies to Accel_Temp_x.ItemOmission;
            emv2::hazards => ([crossreference => "T_AccX_1.00 v2.34";
                    failure => "ItemOmission";
                    phases => ("all");
                    description => "No data from the Temp_Sensor_X";
                    comment => "Reset the sistem if the Temp_Sensor is not working as well";]) applies to Accel_Temp_x.itemomission;
            emv2::hazards => ([crossreference => "T_AccX_1.00 v3.34";
                    failure        => "OutOfRange";
                    phases         => ("all");
                    description    => "Data from the X-Accelerometer Temp Sensor is Out of Range";
                    comment        => "Reset the sistem if the accelerometer is not working as well";]) applies to Accel_Temp_x.OutOfRange;
        **};
end KRIO_FAM300;
```

Figura 13 Component Type della FAM, scheda di controllo principale del sistema

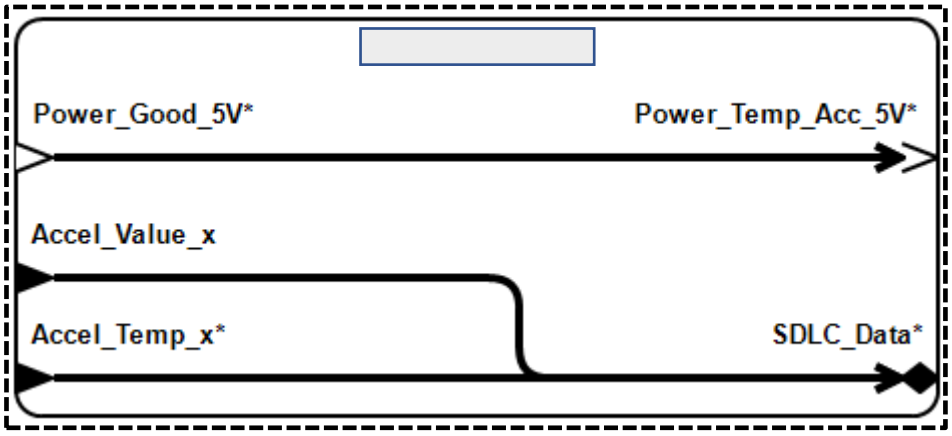


Figura 14 Diagramma della Component Type della FAM

Tra i sottocomponenti della FAM troviamo il processore SoC che verrà illustrato nel capitolo 3, le fasi di lettura e scrittura effettuate dal Firmware e quelle di elaborazione, compensazione e calibrazione effettuate dal Software vengono svolte all'interno della FAM, essendo il processore la piattaforma di

esecuzione. Il flusso dei dati esegue un percorso attraverso la scheda, dalle porte di ingresso alla porta di uscita SDLC_Data, vanno quindi modellate le connessioni che collegano tutti i sottocomponenti interessati durante il passaggio dei dati.

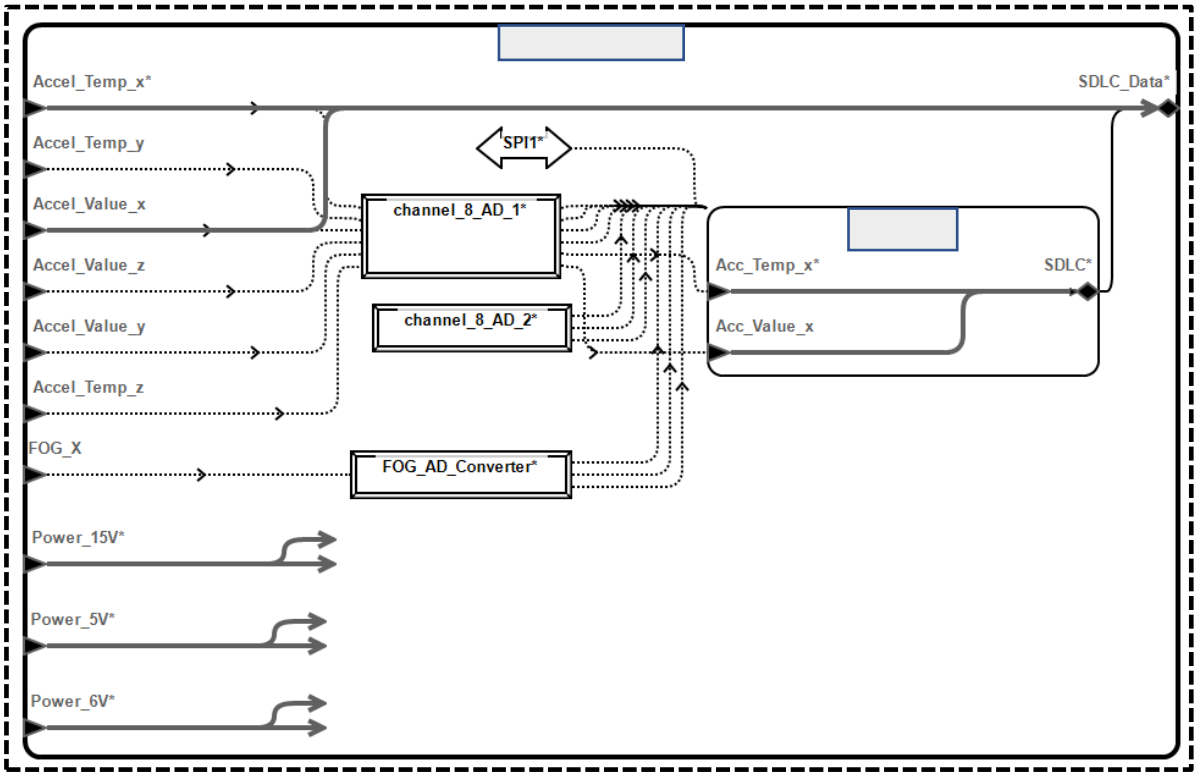


Figura 15 Diagramma della Component Implementation della FAM

Per semplificare il modello però, sono stati modellati nel dettaglio solo gli elementi fondamentali, tutti gli altri componenti hanno ricevuto una modellazione ad altissimo livello che tenga conto principalmente dei flow che li attraversano.
Un esempio è proprio AD Converte, modellato come semplice device con porte di ingresso, di uscita e i flow tra esse, nient'altro.

```
--------------------------8 channel A/D def/impl-------------------------
device channel_8_AD_Accell
    features
        Acc_Value_x: in data po          :Data_types::Accell_Value_X;
        Acc_Value_y: in data po          :Data_types::Accell_Value_Y;
        Acc_Value_z: in data po          :Data_types::Accell_Value_Z;
        Acc_Temp_x: in data por          Data_types::Accell_Temp_X;
        Acc_Temp_y: in data por          Data_types::Accell_Temp_Y;
        Acc_Temp_z: in data por          Data_types::Accell_Temp_Z;

        Out_Acc_Value_x: out data por          Data_types::Accell_Value_X;
        Out_Acc_Value_y: out data por          Data_types::Accell_Value_Y;
        Out_Acc_Value_z: out data por          Data_types::Accell_Value_Z;
        Out_Acc_Temp_x: out data port          ata_types::Accell_Temp_X;
        Out_Acc_Temp_y: out data port          ata_types::Accell_Temp_Y;
        Out_Acc_Temp_z: out data port          ata_types::Accell_Temp_Z;

    flows

        F1: flow path Acc_Value_x -> Out_Acc_Value_x;
        F2: flow path Acc_Value_y -> Out_Acc_Value_y;
        F3: flow path Acc_Value_z -> Out_Acc_Value_z;
        F4: flow path Acc_Temp_x -> Out_Acc_Temp_x;
        F5: flow path Acc_Temp_y -> Out_Acc_Temp_y;
        F6: flow path Acc_Temp_z -> Out_Acc_Temp_z;

end channel_8_AD_Accell;

device implementation channel_8_AD_Accell.impl
end channel_8_AD_Accell.impl;
```

Figura 16 Component Type di un AD_Converter a 8 canali

2.3 Scheda ASE e Optical_Unit

Queste due unità sono le responsabili della parte di sensoristica legata ai Giroscopi. La ASE è la scheda che genera il Laser inviato poi alla Optical_Unit, scomposto da quest'ultima in tre segnali diretti ognuno ad un giroscopio (presenti nella Sensing_Unit). I segnali vengono utilizzati dai giroscopi che restituiscono a loro volta tre segnali diretti nuovamente verso la Optical_Unit e ripresi infine dalla ASE che li filtra e li invia alla FAM per l'elaborazione.

Questi due dispositivi sono stati quasi del tutto ignorati poiché non si effettua nessuna analisi sui dati forniti dai giroscopi e l'attenzione è stata posta principalmente sui dati dagli accelerometri e dei sensori di temperatura; hanno, quindi, una modellazione più semplificata e ad alto livello rispetto ad altri elementi descritti con maggior dettaglio e maggiori informazioni.

La ASE viene descritta come *system* mentre la Optical_Unit come *device* non avendo alcun sottocomponente necessario da descrivere al suo interno, ma essendo solo un dispositivo che fraziona il segnale Laser in ingresso.

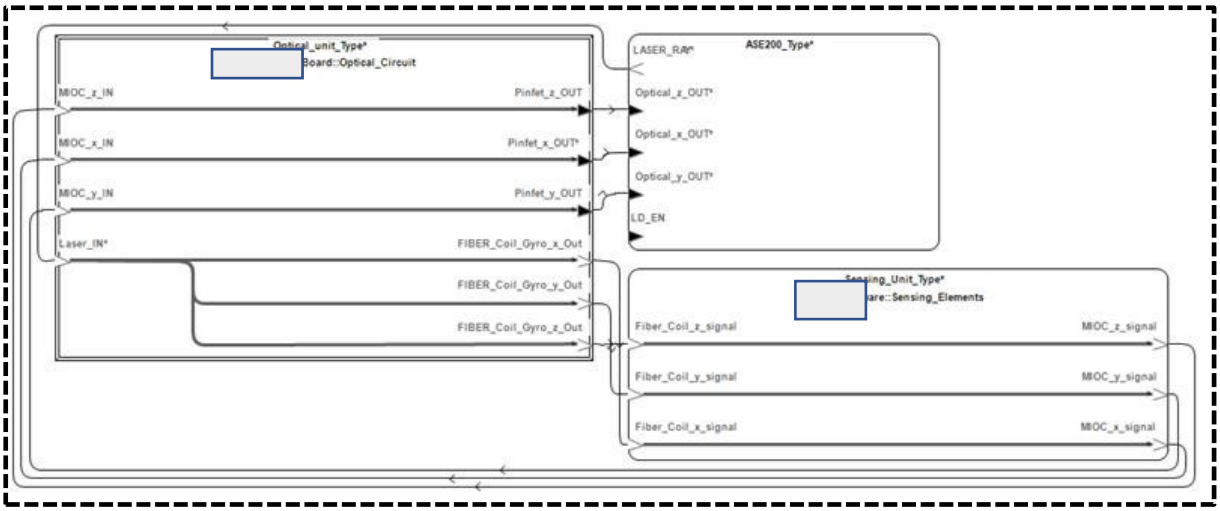


Figura 17 Interazione tra gli elementi della sensoristica

2.4 Sensing_Unit

La Sensing _Unit contiene come sottocomponenti gli accelerometri, i loro relativi sensori di temperatura, e un **system Gyroscope**, nella cui implementazione si trovano definiti come sottocomponenti i *device* che descrivono i Giroscopi X/Y/Z e i sensori di temperatura associati alle bobine di fibra ottica.

```
system implementation Sensing_Elements.impl
        subcomponents
            Gyroscope: system Gyro;
            accel_x: device Accel_X.impl;
            accel_y: device Accel_Y.impl;
            accel_z: device Accel_Z.impl;
            temp_accel_x: device Temperature_Sensor_Accel_X.impl;
            temp_accel_y: device Temperature_Sensor_Accel_Y.impl;
            temp_accel_z: device Temperature_Sensor_Accel_Z.impl;
            GEN_Bus: bus KRIO::Communication_Protocol::bus_generic;

        connections
            C1: port temp_accel_z.Temp_Data -> Accel_Temp_z;
            C2: port temp_accel_y.Temp_Data -> Accel_Temp_y;
            C3: port temp_accel_x.Temp_Data -> Accel_Temp_x;
```

Figura 18 Implementazione della Sensing Unit

```
system implementation Gyro.impl
    subcomponents
        Coil_x_Temp_Sensor: device Temperature_Sensor_Coil_X.impl;
        Coil_y_Temp_Sensor: device Temperature_Sensor_Coil_Y.impl;
        Coil_z_Temp_Sensor: device Temperature_Sensor_Coil_Z.impl;
        MIOC_x: device MIOC_XYZ.impl;
        MIOC_y: device MIOC_XYZ.impl;
        MIOC_z: device MIOC_XYZ.impl;
        Fiber_Coil_x: device Fiber_Coil.impl;
        Fiber_Coil_y: device Fiber_Coil.impl;
        Fiber_Coil_z: device Fiber_Coil.impl;
        GEN_Bus: bus KRIO::Communication_Protocol::bus_generic;

    connections
        C1: port Coil_y_Temp_Sensor.Temp_Data -> Gyro_Fiber_Temp_y;
        C2: port Coil_z_Temp_Sensor.Temp_Data -> Gyro_Fiber_Temp_z;
        C3: port Coil_x_Temp_Sensor.Temp_Data -> Gyro_Fiber_Temp_x;
```

Figura 19 Implementazione dei giroscopi

Successivamente sono stati modellati i vari sottocomponenti, le relative porte e proprietà, connessioni con altri componenti, i dataflow interni tramite i relativi flow e tutti gli altri elementi costitutivi del Sistema, compresi i moduli PS e PL, che contengono la parte Software e Firmware. Sono stati modellati i processori, le memorie e i Bus di comunicazione con relativi protocolli di comunicazione, oltre ai principali Software suddivisi tra processi, thread e dati. Ognuno di questi elementi deve essere completo di tutte le proprietà necessarie alle relative analisi**,** quindi vanno inserite anche le sezioni relative agli errori di ogni singolo componente. Tra le proprietà abbiamo anche specificato su che HW girano i Software e a quali protocolli di comunicazione sono legate le varie connessioni.

2.5 Sensori di temperatura

Il *Dataflow* su cui è stata effettuata l'analisi degli errori è quello relativo al dato di temperatura dell'accelerometro X.
Vengono definiti gli elementi principali, si deve associare ad ogni porta la tipologia di errore opportuna, in questo caso l'errore potrebbe nascere dal sensore e quindi affliggere il dato *Accel_Temp_X* uscente dalla porta *Temp_Data*, viene quindi impostata tale porta come **out propagation** dell'errore, dove quest'ultimo può essere una mancanza di dato {*ItemOmission*} o un fuoriscala {*OutOfRange*}. Viene definito anche un possibile errore in ingresso sull'alimentazione, errore che è la possibile causa di un errore sul dato. Questo propagarsi dell'errore dall'alimentazione al dato viene definito negli error path.
Ovviamente questo errore fluirà attraverso la Sensing Unit e lo troveremo in uscita diretto verso la FAM, poiché questo sensore è sottocomponente proprio della Sensing Elements.
Questo propagarsi dell'errore deve essere definito in ogni elemento che "interagisce" col dato considerato, per poter seguire così il percorso completo dalla nascita nel device TEMPERATURE_SENSOR_ACCEL_X fino alla morte nel MAIN CONNECTOR.

```
-----------------------------------Temp Sensor Accellerometer-------------------
device Temperature_Sensor_Accel_X
    features
        accel_failure: out data port;
        Temp_Data: out data port KRIO::Data_types::Accell_Temp_X;
        Power_5V: in event port;
    flows
        f1: flow source Temp_Data;

    annex EMV2 {**
            use types ErrorLibrary,Error_annex;
            use behavior Error_annex::ThreeState;

            error propagations
                accel_failure: out propagation {break, ItemOmission};
                Temp_Data: out propagation {OutOfRange, ItemOmission};
                Power_5V: in propagation {NoPower,LOW_Power};
            flows
                ef0: error source Temp_Data {OutOfRange, ItemOmission};
                F1: error path Power_5V{NoPower} -> Temp_Data{OutOfRange};
                F2: error path Power_5V{LOW_Power} -> Temp_Data{ItemOmission};
            end propagations;

            component error behavior
                events
                    Reset: recover event;
                transitions
                    t0: Operational-[accel_failure]-> FailStop;
                    t01: Operational-[Power_5V{LOW_Power}]-> CriticalModeFailure;
                    t02: Operational-[Power_5V{NoPower}]-> Failstop;
                    t03: NCritF_5V-[Reset]-> Operational;
                    t04: CritF_5V-[Broken]-> FailStop;
                    t05: CritF_5V-[Reset]-> Operational;
                    t1: FailStop -[Reset]-> Operational;
                    t2: Operational-[Critical_Failure]-> CriticalModeFailure;
                    t3: CriticalModeFailure-[Stop]-> FailStop;
            end component;

            properties

                emv2::OccurrenceDistribution => [ProbabilityValue => 0.01e-6; Distr
```

Figura 20 Component Type sensore di temperatura

2.6 Accelerometro X

L'accelerometro viene preso in considerazione poiché il dato sulla quale viene effettuata l'analisi di latenza è proprio *Accell_Value_X*, è quindi necessario esplicitare la porta da cui il dato esce, e modellarla come un flow source.
Anche gli altri accelerometri saranno modellati identicamente a questo, ma per semplicità si considera solo il dato proveniente da Accel_X
Vengono definite le proprietà del sensore che serviranno da contributori per l'analisi di latenza e si specifica la latenza prevista sull'uscita del dato dal sensore, sulla porta acc_out_data.

```
-------------------------------Accel Device Def/Impl----------------------
  device Accel_X
      features
          acc_out_data: out data port KRIO::Data_types::Accell_Value_X;
          Power: in event port;
      flows
          f1: flow source acc_out_data {latency => 1ms .. 2ms;};
          accel_new_flow_spec3: flow sink Power;
      properties
          Period => 500us;
      Dispatch_Protocol => Periodic;
      compute_execution_time => 900us .. 1ms;

  annex EMV2 {**
      use types ErrorLibrary;
      use behavior ErrorLibrary::FailStop;

      error propagations
          acc_out_data: out propagation{ItemOmission,OutOfRange};
      flows
          ef0 : error source acc_out_data{ItemOmission,OutOfRange};
      end propagations;
```

Figura 21 Component Type dell'accelerometro x

Sono stati modellati anche gli errori che potrebbero presentarsi sul device, ma solo per scopo illustrativo, poiché questo elemento non viene considerato durante le analisi di safety.
Tutti la sensoristica inerziale è sottocomponente della Sensing Elements, definita come system, e tutti i dati provenienti dai sensori sono letti e immagazzinati dal Firmware presente sulla FAM come sottocomponente software della parte FPGA del processore.

La FAM prende in ingresso le tre alimentazioni fornite dalla Power_Board, e le elargisce a tutta la componentistica del sistema, fornendo alimentazione, quindi, anche alla Sensing Unit e di conseguenza ai sensori di temperatura e agli accelerometri. Tra le *features* dei due *device* si notano le porte Power_5V e Power, entrambe settate come *in event port*, una che, come dice il nome, rappresenta l'alimentazione a 5V fornita ai sensori di temperatura mentre l'altra rappresenta l'alimentazione a 15V fornita agli accelerometri.

2.7 Unità di alimentazione del sistema: POWER_BOARD

L'alimentazione dell'intero sistema è gestita dalla Power_Unit che riceve in ingresso 28V e tramite filtri e DC/DC converter fornisce in uscita 15V, 6V e 5V per poter alimentare gli elementi presenti nel sistema con le corrette tensioni richieste.
Modellato come system, anche questo elemento necessita di essere dettagliato nella sua implementazione tramite la modellazione dei suoi sottocomponenti.
In questo dispositivo non vengono specificati elementi contributori alla latenza, ma viene dettagliata la parte degli errori che possono affliggere i vari sottocomponenti e di conseguenza le differenti alimentazioni.

```
----------------------------Power Board Unit def/impl------------
system Power_Board
    features
        Power_15V: out data port;
        Power_6V: out data port;
        Power_5V: out data port;

        V28_IN: in out event port;
        VP_15V_A: in out event port;
        VN_15V_A: in out event port;
        GND_A: in out event port;
        VP_6V_A: in out event port;
        VN_6V_A: in out event port;
        VP_5V_D: in out event port;
        GND_D: in out event port;
        Power_Good_15V: out event port;
        Power_Good_6V: out event port;
        Power_Good_5V: out event port;
    flows
        Power_Board_new_flow_spec: flow source Power_6V;
        Power_Board_new_flow_spec2: flow source Power_5V;
        Power_Board_new_flow_spec3: flow source Power_15V;
        Power_Board_new_flow_spec4: flow source Power_Good_15V;
        Power_Board_new_flow_spec5: flow source Power_Good_6V;
        Power_Board_new_flow_spec6: flow source Power_Good_5V;
```

Figura 22 Component Type della Power_Board

Come errori che possono presentarsi sull'alimentazione sono stati considerati:

- la mancanza di alimentazione (**NoPower**)
- un calo di tensione (**LOW_Power**)

Tramite la definizione degli error path, viene rappresentata l'influenza che questi errori hanno sulle tre alimentazioni in uscita nel caso in cui si presentassero sull'alimentazione generale a 28V.
Anche questo dispositivo è quindi stato dotato di una macchina a stati che descrive il comportamento in caso di errore.
Viene quindi definita, anche in questo caso, una **component error behavior** nella Type della Power_Board e una **composite error behavior** nella sua implementation, quest'ultima prende in considerazione anche gli stati dei sottocomponenti.

```
error propagations
    V28_IN: in propagation {NoPower,LOW_Power};
    Power_15V: out propagation {NoPower,LOW_Power};
    Power_Good_15V: out propagation {NoPower,LOW_Power};
    Power_5V: out propagation {NoPower,LOW_Power};
    Power_Good_5V: out propagation {NoPower,LOW_Power};
    Power_6V: out propagation {NoPower,LOW_Power};
    Power_Good_6V: out propagation {NoPower,LOW_Power};

    flows
        ef0: error source Power_15V {NoPower} when {HardwareFailure};
        ef1: error source Power_15V {LOW_Power} when {Power_Issue};
        EF3: error source Power_Good_15V {NoPower} when {HardwareFailure};
        EF2: error source Power_Good_15V {LOW_Power} when {Power_Issue};
        EF4: error source Power_Good_5V {NoPower} when {HardwareFailure};
        EF5: error source Power_Good_5V {LOW_Power} when {Power_Issue};
        ef6: error source Power_5V {NoPower} when {HardwareFailure};
        ef7: error source Power_5V {LOW_Power} when {Power_Issue};
        EF8: error source Power_Good_6V {NoPower} when {HardwareFailure};
        EF9: error source Power_Good_6V {LOW_Power} when {Power_Issue};
        ef10: error source Power_6V {NoPower} when {HardwareFailure};
        ef11: error source Power_6V {LOW_Power} when {Power_Issue};
        F1: error path V28_IN{NoPower} -> Power_Good_15V{NoPower};
        F2: error path V28_IN{LOW_Power} -> Power_Good_15V{LOW_Power};
        F3: error path V28_IN{LOW_Power} -> Power_15V{LOW_Power};
        F4: error path V28_IN{NoPower} -> Power_15V{NoPower};
        F5: error path V28_IN{LOW_Power} -> Power_Good_6V{LOW_Power};
        F6: error path V28_IN{LOW_Power} -> Power_6V{LOW_Power};
        F7: error path V28_IN{LOW_Power} -> Power_Good_5V{LOW_Power};
        F8: error path V28_IN{LOW_Power} -> Power_5V{LOW_Power};
end propagations;

component error behavior
    events
        Reset: recover event;
    transitions
        t0: Operational -[Broken]-> FailStop;
        t1: FailStop -[Reset]-> Operational;
        t2: Operational -[Critical_Failure]-> CriticalModeFailure;
        t3: CriticalModeFailure -[Stop or V28_IN{NoPower}]-> FailStop;
        t4: Operational -[V28_IN{NoPower}]-> FailStop;
        t5: Operational -[V28_IN{LOW_Power}]-> CriticalModeFailure;
        t6: Operational -[Broken]-> FailStop;
```

Figura 23 Annex EMV2 Power_Board

Tra i sottocomponenti sono presenti i tre DC/DC i quali sono dichiarati come device e al cui interno vengono modellati gli errori e le macchine a stati che influenzeranno poi la macchina a stati composite dell'intera unità.

```
system implementation Power_Board.impl
    subcomponents
        EMI_FLT: device EMI_Filter.impl;
        SURGE_STOP: device Surge_Stopper.impl;
        DCDC_15V: device DC_DC_15V.impl;
        DCDC_6V: device DC_DC_6V.impl;
        DCDC_5V: device DC_DC_5V.impl;

    connections
        Power_Board_impl_new_connection: port V28_IN -> EMI_FLT.V28_in;
        Power_Board_impl_new_connection2: port EMI_FLT.Filtered_V28 -> SURGE_STOP.Filtered_V28;
```

Figura 24 Component Implementation Power_Board

La figura 25 mostra, all'interno della *composite error behavior,* quali sono le condizioni modellate per cui l'intera Power_Unit può finire in uno stato di Failstop.
Come spiegato sopra queste condizioni dipendono dagli stati possibili dei singoli sottocomponenti. Questo significa che andrà modellata una *component error behavior* per ogni sottocomponente preso in analisi**.**

```
annex EMV2 {**
        use types ErrorLibrary;
        use behavior Error_annex::ThreeState;

        composite error behavior
            states
                [DCDC_15V.Error_15V_power]-> CritF_15V;
                [DCDC_6V.Error_6V_power]-> CritF_6V;
                [DCDC_5V.Error_5V_power]-> CritF_5V;
                [DCDC_15V.Error_15V_LOW_Power]-> NCritF_15V;
                [DCDC_6V.Error_6V_LOW_Power]-> NCritF_6V;
                [DCDC_5V.Error_5V_LOW_Power]-> NCritF_5V;
                [DCDC_15V.Failstop or DCDC_6V.Failstop or DCDC_5V.Failstop]-> Failstop;
                [EMI_FLT.Failstop or SURGE_STOP.Failstop]-> Failstop;

        end composite;

        **};
```

Figura 25 Composite error behavior Power Board

```
device DC_DC_15V
    features
        V28_FLT: in event port;
        VP_15V_A: in out event port;
        VN_15V_A: in out event port;
        GND_A: in out event port;
        Power_Good_15V: out event port;

    annex EMV2 {**
            use types ErrorLibrary,Error_annex;
            use behavior Error_annex::ThreeState;

            error propagations
                Power_Good_15V: out propagation {NoPower,LOW_Power};
                V28_FLT: in propagation {NoPower,LOW_Power};
            flows
                PF0: error sink V28_FLT{NoPower};
                PF01: error sink V28_FLT{LOW_Power};
                CRIT_PF1: error source Power_Good_15V{NoPower} when {HardwareFailure};
                WARN_PF2: error source Power_Good_15V{LOW_Power} when {Power_Issue};
                CRIT_F1: error path V28_FLT{LOW_Power} -> Power_Good_15V{LOW_Power};
            end propagations;

            component error behavior
                transitions
                    t01: Operational-[Broken]->Error_15V_power;
                    t0: Operational-[V28_FLT{NoPower}]->Error_15V_power;
                    t1: Operational-[V28_FLT{LOW_Power}]->Error_15V_LOW_Power;
                    t2: Error_15V_power-[V28_FLT{NoPower}]->Failstop;
                    t3: Error_15V_LOW_Power-[V28_FLT{NoPower}]->Failstop;
                    t4: Error_15V_LOW_Power-[Reset]->Operational;

                propagations
                    CRIT_Error: all -[V28_FLT{NoPower}]-> Power_Good_15V{NoPower};
            end component;

            properties
                emv2::OccurrenceDistribution => [ProbabilityValue => 3.01e-9; Distribution => Poisson;] applies to V28_FLT.NoPower;
```

Figura 26 Component Type DC/DC 15V

Si ricorda che ognuno degli stati, degli eventi e dei tipi di errori usati in questo lavoro sono stati dichiarati all'interno di un apposito package chiamato *Error_annex*, che contiene oltre allo stato di Failstop anche gli stati CritF_15V e NCritF_15V, oltre a molti altri che non vengono utilizzati.
Il motivo di queste aggiunte è mostrare come sia malleabile la modellazione degli errori e come si possano creare ad hoc secondo le proprie necessità di progettazione.
Si può quindi modellare un sistema semplice come uno molto articolato senza dover cambiare logica o semantica.

2.8 BUS e "*connection binding*"

All'interno del package "Communication_Protocol", vengono definiti i protocolli di comunicazione che saranno poi associati alle connessioni del sistema che sono interessate dall'analisi di latenza.
Questo serve a specificare come il bus, attraverso il quale passano i dati, contribuisce alla latenza dei dati stessi.

```
package [redacted]:Communication_Protocol
public
    with Bus_Properties;
    with SEI;
```

Figura 27 Package Communication Protocol

Per implementare l'effettiva comunicazione tra i componenti del sistema è necessario definire i bus poiché le informazioni di legame tra connessioni e bus sono usate per computare i tempi di trasporto dei dati fra nodi. Senza queste informazioni, non c'è modo per i tools di analisi di calcolare le tempistiche di trasmissione, necessarie per l'analisi di latenza oltre ad altri tipi di analisi.
Lo stesso concetto vale per la definizione delle piattaforme di esecuzione delle componenti Software; come verrà mostrato nel capitolo successivo, Software e Firmware devono essere legati ad un Hardware Processor che li esegue. Queste proprietà di legame sono definite in AADL come *deployment_properties*, mostrate nella figura 28.

In questo specifico caso le proprietà necessarie, usate per rendere completo il sistema sono:

- **Actual_processor_binding**: proprietà che si applica ad un AADL *process* o *thread* lega l'elemento SW ad un AADL processor o *virtual processor*.
- **Actual_connection_binding**: proprietà che si applica ad una *port connection* lega la connessione ad un *bus* o *virtual bus*.

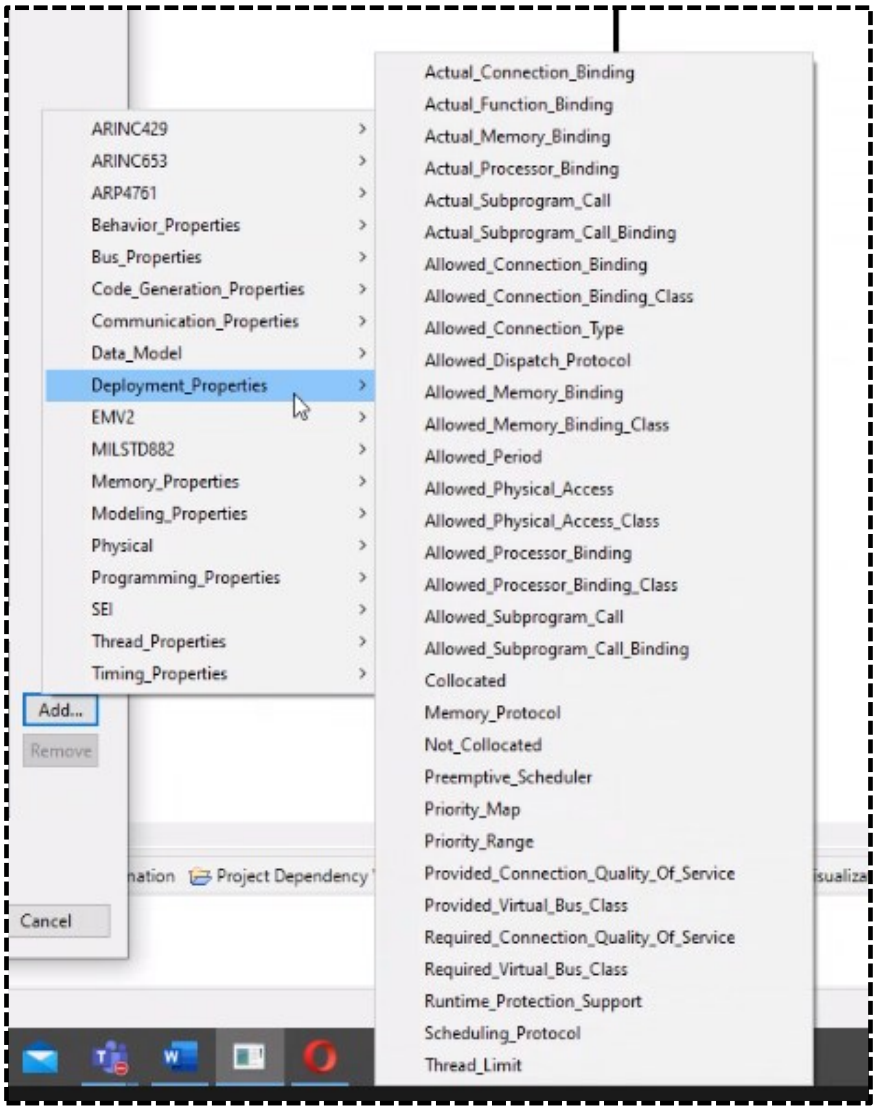


Figura 28 Deployment Properties

I principali bus definiti sono:

- **AXI BUS** - che mette in comunicazione i due moduli, PS e PL del processore. È attraverso questo bus che il Firmware e il Software si scambiano le informazioni.
- **Bus Generic** - utilizzato per le connessioni che non hanno un protocollo di comunicazione specifico, piuttosto che lasciarle scoperte e ottenere uno scambio istantaneo tra i dati, si è preferito modellare una minima presenza di latenza.
- **SPI** – che viene usato dal Firmware come canale di comunicazione per la lettura dei dati dai sensori inerziali, lega la PL con la Sensing Unit.

```
bus AXI
properties
    Latency => 1 Ms .. 1 Ms;
    SEI::BandWidthCapacity => 204800.0 bitsps;
    --Transmission_Time => [ Fixed => 10 ms .. 30ms;
                        --  PerByte => 1 us .. 10 us; ];
end AXI;

-------------------------------------------------
bus bus_generic
    properties
        Latency => 1 us .. 1 us;
        SEI::BandWidthCapacity => 2000000.0 bitsps;
        --Transmission_Time => [ Fixed => 1us .. 3us; PerByte => 1 us .. 10 us; ];

end bus_generic;


-------------Bus SPI 1 to F-RAM-------------------------

bus SPI
    properties
        Latency => 1 ms .. 2 ms;
        SEI::BandWidthCapacity => 2000.0 bitsps;
        Transmission_Time => [ Fixed => 500us .. 2ms;];
                        --   PerByte => 1 us .. 1 us; ];
end SPI;
```

Figura 29 Definizione dei tre protocolli di comunicazione del sistema

2.9 Data types

Nella modellazione del sistema è importante specificare anche quali sono le tipologie di dati presi in analisi, per esempio i dati forniti dagli accelerometri e dai sensori di temperatura sono dei float16 mentre i dati forniti dai giroscopi sono dei float32.
La dimensione del dato è una caratteristica fondamentale per l'analisi di latenza e non solo, è quindi opportuno creare un package in cui vengano definiti tutti i tipi di dati che sono presenti nel sistema che si sta modellando.

Si possono definire tutti i tipi comuni di dato, come integers di varie dimensioni (signed/unsigned 8, 16, 32 e 64 bits) oppure stringhe o valori a virgola mobile.

Accel Value X è il dato principale sul quale si esegue l'analisi di latenza. Nelle proprietà viene specificata la dimensione, essendo questa caratteristica quella che contribuisce alla latenza, poiché cambiano i tempi con cui un dato attraversa un bus in quanto i bus hanno una loro capienza e un loro tempo di trasmissione per bytes.

Quindi i dati forniti da accelerometri e sensori di temperatura avranno dimensione di 2 bytes, mentre i dati forniti dai giroscopi avranno dimensione di 4 bytes.

```
data Accell_Value_X
        properties
            Type_Source_Name => "__po_hi_float16_t";
            Source_Text => ("types");
            Data_Size => 2 Bytes;
end Accell_Value_X;


data Accell_Temp_X
        properties
            Type_Source_Name => "__po_hi_float16_t";
            Source_Text => ("types");
            Data_Size => 2 Bytes;
end Accell_Temp_X;


data Gyro_FOG_X
        properties
            Type_Source_Name => "__po_hi_float32_t";
            Source_Text => ("types");
            Data_Size => 4 Bytes;
end Gyro_FOG_X;
```

Figura 30 Definizione dei tipi di dato

Il dato subisce un'elaborazione all'interno del Software ma non ne modifica le dimensioni, però per chiarezza è stato creato un altro tipo di dato con nome diverso ma stessa dimensione per specificare se si tratta del dato pre o post elaborazione.

```
data Angular_Rate_Body_Axis_X
        properties
            Type_Source_Name => "__po_hi_float16_t";
            Source_Text => ("types");
            Data_Size => 2 Bytes;
end Angular_Rate_Body_Axis_X;
```

Figura 31 Tipo di dato post elaborazione

Tutti questi dati vengono poi raccolti in un unico pacchetto che avrà una dimensione data dalla somma delle dimensioni dei singoli dati che lo compongono.
In questo caso tutti i dati post elaborazione vengono aggregati in un pacchetto chiamato Message_ID_0x02 che contiene tutti gli elementi presenti in due pacchetti distinti.
A metà elaborazione viene prodotto un pacchetto chiamato *Message_ID_0x01*, mentre a fine elaborazione viene creato un pacchetto chiamato Message_IND, questi due pacchetti vengono poi uniti e forniti in uscita (*Message_ID_0x02*).

```
------------------Inertial Navigation Data--------------------------
data Message_IND
        properties
            Type_Source_Name => "__po_hi_long_t";
            Source_Text => ("types");
            Data_Size => 24 Bytes;
end Message_IND;

-----------------Flight Control Data & Inertial Navigation Data--------
data Message_ID_0x02
        properties
            Type_Source_Name => "__po_hi_long_t";
            Source_Text => ("types");
            Data_Size => 40 Bytes;
end Message_ID_0x02;

data implementation Message_ID_0x02.impl
    subcomponents
        Angular_Rate_X: data Angular_Rate_Body_Axis_X;
        Angular_Rate_Y: data Angular_Rate_Body_Axis_Y;
        Angular_Rate_Z: data Angular_Rate_Body_Axis_Z;
        Linear_Acc_X: data Linear_Acc_Body_Axis_X;
        Linear_Acc_Y: data Linear_Acc_Body_Axis_Y;
        Linear_Acc_Z: data Linear_Acc_Body_Axis_Z;
        Status_Word_1: data Status_Word_1;
        Status_Word_2: data Status_Word_2;
        Delta_Angle_X: data Delta_Angle_Axis_X;
        Delta_Angle_Y: data Delta_Angle_Axis_Y;
        Delta_Angle_Z: data Delta_Angle_Axis_Z;
        Delta_Velocity_X: data Delta_Velocity_Axis_X;
        Delta_Velocity_Y: data Delta_Velocity_Axis_Y;
        Delta_Velocity_Z: data Delta_Velocity_Axis_Z;

end Message_ID_0x02.impl;
```

Figura 32 Definizione dei pacchetti dati Message_IND e Message _ID_0x02

```
--------------------Flight Control Data--------------------------------
data Message_ID_0x01
        properties
            Type_Source_Name => "__po_hi_long_t";
            Source_Text => ("types");
            Data_Size => 16 Bytes;
end Message_ID_0x01;

data implementation Message_ID_0x01.impl
    subcomponents
        Angular_Rate_x: data Angular_Rate_Body_Axis_X;
        Angular_Rate_y: data Angular_Rate_Body_Axis_Y;
        Angular_Rate_z: data Angular_Rate_Body_Axis_Z;
        Linear_Acc_x: data Linear_Acc_Body_Axis_X;
        Linear_Acc_y: data Linear_Acc_Body_Axis_Y;
        Linear_Acc_z: data Linear_Acc_Body_Axis_Z;
        Status_Word_1: data Status_Word_1;
        Status_Word_2: data Status_Word_2;
end Message_ID_0x01.impl;
```

Figura 33 Definizione del pacchetto dati Message _ID_0x01

Nelle porte viene specificato quale pacchetto le attraversa inserendo il classificatore nella definizione della porta stessa. Se su una porta non viene specificato nessun classificatore allora tutti i tipi di dato possono attraversarla e può essere connessa con porte con classificatori diversi. Ma se il classificatore è definito allora solo quel tipo di dato potrà usarla e le connessioni saranno possibili solo con porte con stesso classificatore.

```
features
    Acc_Value_x: in data port      :Data_types::Accell_Value_X;
    Acc_Value_y: in data port      :Data_types::Accell_Value_Y;
    Acc_Value_z: in data port      :Data_types::Accell_Value_Z;
    Acc_Temp_x: in data port      Data_types::Accell_Temp_X;
```

Figura 34 Definizione delle porte con il classificatore specifico per il tipo di dato

Il pacchetto Message_ID_0x02 contiene tutti i dati forniti dal sistema per l'utente, e saranno accessibili dalla porta SDLC, per questo anche sulla porta SDLC è stato messo il classificatore opportuno.

```
Flight_Control_Data_Message: in data port      :Data_types::Message_ID_0x01;
Inertial_Data_Message: in data port      :Data_types::Message_ID_0x02;
SDLC: in out data port      :Data_Types::Message_ID_0x02;
```

Figura 35 Definizione delle porte con il classificatore specifico per il pacchetto considerato

CAPITOLO 3: Processore, Software e Firmware

3.0 Introduzione al capitolo

Essendo il processore un elemento articolato, composto da due parti ognuna con una piattaforma di esecuzione diversa, avente interazione con la parte Software del sistema e essendo il system su cui è stata effettuata l'analisi di latenza, è opportuno separarlo dal capitolo 2 e dedicargli una spiegazione più approfondita, illustrando inoltre come l'Hardware si interfaccia con il Software.

3.1 Processore

Il system SoC descrive il processore come l'unione della parte CPU alla parte FPGA, è diviso in due moduli: **PS_Module** e **PL_Module**, la PS è la porzione che gestisce la CPU e su cui gira il Software mentre la PL è la porzione che gestisce l'FPGA e su cui gira il Firmware.
Queste due sezioni comunicano tra di loro tramite un AXI BUS, sono definite come *system* e come *subcomponents* del processore SoC, vengono modellate le porte, le connessioni e i flow sia del system SoC sia dei Moduli PS e PL.

```
system implementation Zynq7000.impl
    subcomponents
        PS_MODULE: system [ ]PS_Module;
        PL_MODULE: system [ ]PL_Module;
        PS_MODULE_impl: system Z[ ]S_Module.impl;
        PL_MODULE_impl: system Z[ ]L_Module.impl;
        AXI_BUS: bus KRIO::Communication_Protocol::AXI;
```

Figura 36 Sottocomponenti del processore

Il valore dell'accelerometro X viene letto dalla PL per mezzo di un thread di lettura situato all'interno del Firmware, viene poi passato tramite AXI BUS alla PS e quindi al Software che lo elabora e compensa tramite due algoritmi di Coning and Sculling, successivamente viene rimandato alla PL per essere scritto in uscita dal Firmware sulla porta SDLC.

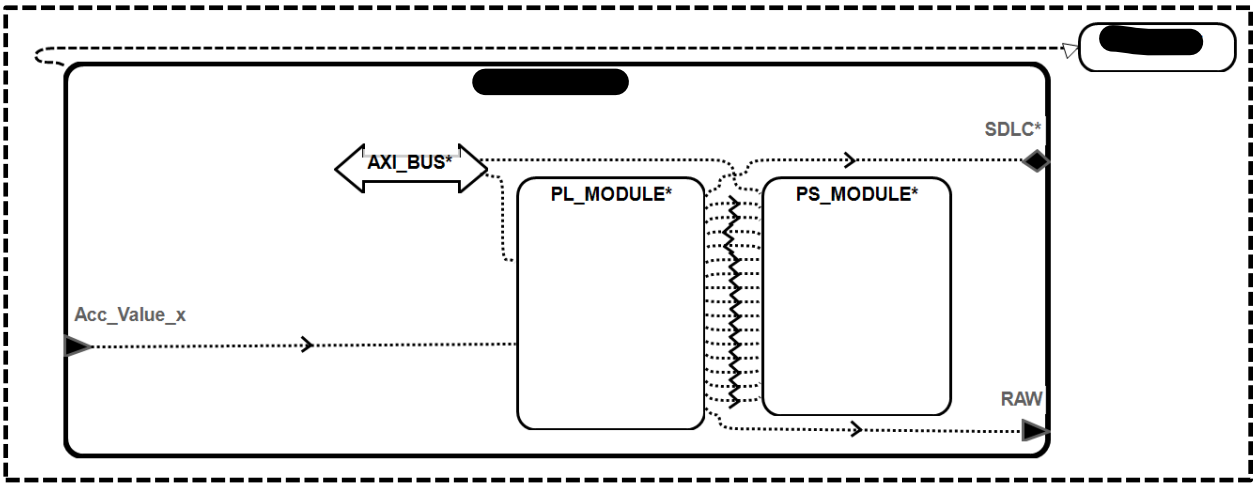


Figura 37 Rappresentazione grafica della definizione del processore

La PL prende in ingresso *Acc_Value_X* fornendolo in uscita sulla porta *OUT_ Acc_Value_X*, questa operazione viene eseguita dal Firmware che tramite il **READ_thread** e tramite un protocollo SPI legge il dato dall'accelerometro X per poi inviarlo alla PS. In uscita vi è anche la porta *RAW_Data* che rappresenta l'insieme dei dati grezzi previa qualsiasi elaborazione da parte del Software (figura 38).

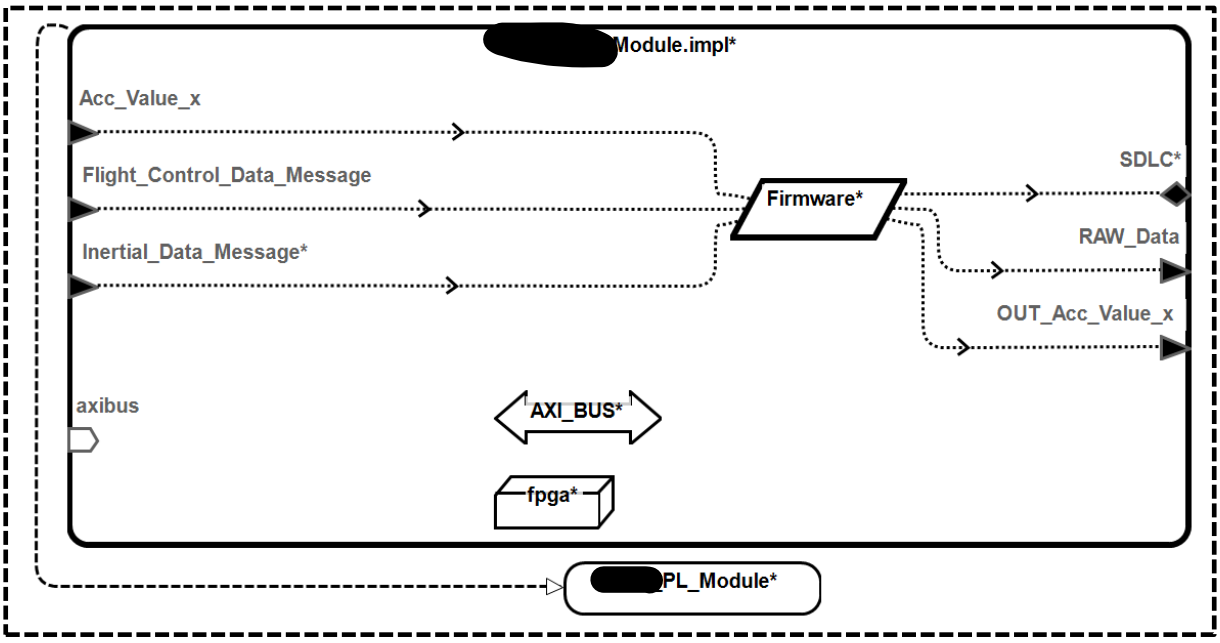


Figura 38 Rappresentazione grafica modulo PL

Il dato proveniente dalla PL viene acquisito dal software, che lo compensa e lo restituisce sulla porta *Inertial_Data_Message* (figura 39); su questa porta in realtà arriva un pacchetto dati creato dal Software e contenente tutti i valori fondamentali provenienti dalla sensoristica, calibrati e compensati, e pronti per essere utilizzati dall'utente.
Questo pacchetto viene rigirato dalla PS alla PL, il Software lo invia nuovamente al Firmware, che poi lo fornisce in uscita sulla porta SDLC pronto per essere letto dall'utente (figura 37).

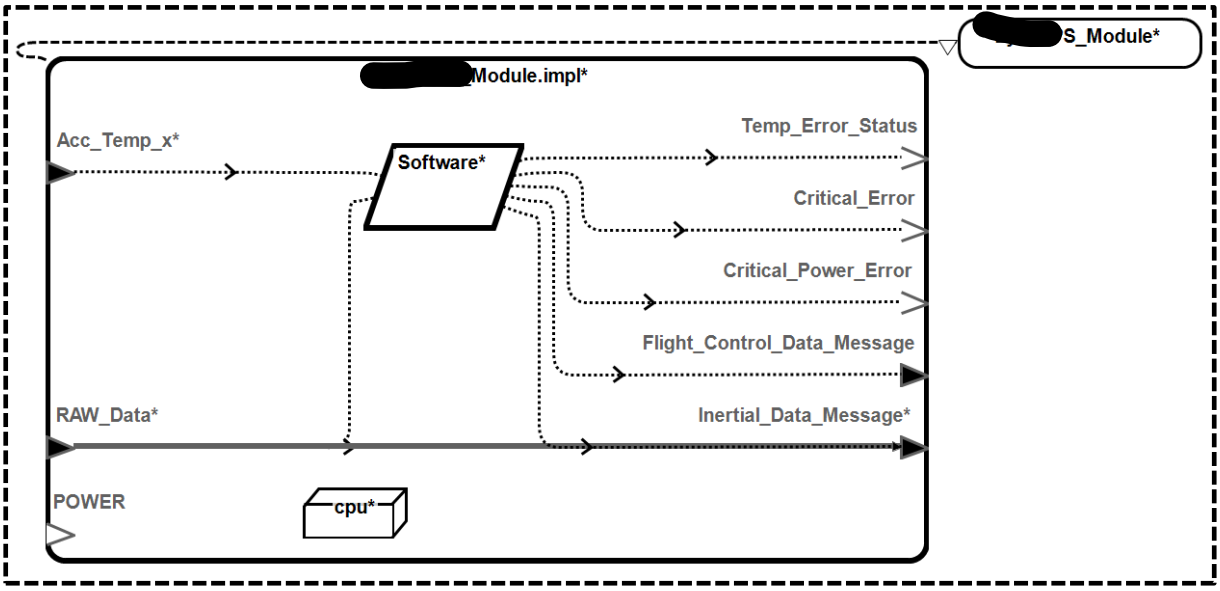


Figura 39 Rappresentazione grafica del modulo PS

Il percorso sopra descritto viene rappresentato attraverso un **end-to-end flow** dove vengono specificati il **flow source**(origine), il **flow path**, ossia il percorso attraversato e le connessioni utilizzate (C14, C16), e il **flow sink**(arrivo).
Le connessioni utilizzate vengono poi associate, tramite *properties*, al Bus che le descrive; la PS e la PL comunicano tramite AXI BUS e quindi lo scambio di dati tra di loro avviene attraverso quest'ultimo.

```
    ------------------------------End to End flow of Software Data elaboration---------------------------------------
    ETEF1: end to end flow PL_MODULE.F10 -> C14 -> PS_MODULE.IND_Flow -> C16 -> PL_MODULE.FF {latency => 1ms .. 5ms;};

properties
    actual_connection_binding => (reference (AXI_BUS)) applies to C14, C16;
```

Figura 40 Definizione del flusso end-to-end

All'interno della dichiarazione del system SoC vengono anche specificate le proprietà degli errori e la macchina a stati del componente stesso.
Tra gli elementi della macchina a stati del processore vediamo la presenza dello stato di *Failstop*, causato per esempio da una mancanza di alimentazione a 15V. Lo stato di *Failstop* è presente in tutti i componenti del sistema e verrà usato poi per lo studio dell'analisi dell'Albero degli Errori (FTA).

```
component error behavior
    events
        Failure: error event;
        Critical_Failure: error event;
        Reset: recover event;
        Stop: error event;
    transitions
        t0_1: Operational -[Power_15V{LOW_Power}]-> NonCriticalModeFailure;
        t0: Operational -[Acc_Temp_x{OutOfRange}]-> NonCriticalModeFailure;
        t1: Operational -[Power_15V{NoPower}]-> Failstop;
        t2: Operational -[Acc_Temp_x{ItemOmission}]-> CriticalModeFailure;
        t3: NonCriticalModeFailure -[ResetEvent]-> Operational;
        t4: NonCriticalModeFailure -[Power_15V{NoPower}]-> Failstop;
        t5: CriticalModeFailure -[Stop]-> Failstop;
        t6: CriticalModeFailure -[Reset]-> Operational;
        t7: Failstop -[Reset]-> Operational;

end component;
```

Figura 41 Macchina a stati del componente system SoC, definita nella Type

3.2 PL_Module

Questo modulo rappresenta la parte FPGA del sistema, il cui compito è la lettura dei dati dalla sensoristica inerziale, passare questi dati grezzi al Software, prelevarli dal Software dopo essere stati elaborati e consegnarli in uscita all'utente. Tutto questo viene svolto dal Firmware installato sulla PL.

```
system Zynq_PL_Module
      features
            axibus: requires bus access [redacted]:Communication_Protocol::AXI;
            FOG_X: in data port [redacted]ata_types::Gyro_FOG_X;
            Acc_Value_x: in data port [redacted]Data_types::Accell_Value_X;
            Acc_Temp_x: in data port [redacted]:Data_types::Accell_Temp_X;
            RAW_Data: out data port [redacted]:Data_types::Message_ID_0x02;
            OUT_Acc_Value_x: out data port [redacted]:Data_types::Accell_Value_X;
            OUT_Acc_Temp_x: out data port [redacted]::Data_types::Accell_Temp_X;
            OUT_FOG_X: out data port [redacted]Data_types::Gyro_FOG_X;
            Flight_Control_Data_Message: in data port [redacted]:Data_Types::Message_ID_0x01;
            Inertial_Data_Message: in data port [redacted]:Data_Types::Message_ID_0x02;
            SDLC: in out data port [redacted]:Data_Types::Message_ID_0x02;
            UART: in out data port [redacted]ata_Types::Message_ID_0x02;
      flows
            F1: flow sink Acc_Value_x;
            F4: flow sink Acc_Temp_x;
            F10: flow source RAW_Data;
            F11: flow source SDLC;
            FF: flow sink Inertial_Data_Message;
            FF1: flow path Acc_Temp_x -> RAW_Data;
            FF2: flow path Inertial_Data_Message -> SDLC;
            FF3: flow path Acc_Temp_x -> OUT_Acc_Temp_x;

end Zynq_PL_Module;
```

Figura 42 Component Type del modulo PL

All'interno della *Implementation* viene definito tra i sottocomponenti il processo Firmware, parte adibita alla lettura e scrittura ad alta frequenza dei dati.
Vengono definite anche le connessioni che legano il sottocomponente alle porte del sistema che lo contiene.

```
system implementation [redacted]PL_Module.impl
    subcomponents
        fpga: processor [redacted]00_fpga;
        Firmware: process [redacted]:Firmware::Firmware.impl;
        SPI: bus [redacted]:Communication_Protocol::SPI;
        UART0: bus [redacted]:Communication_Protocol::UART_0;
        UART_Bus: bus [redacted]:Communication_Protocol::UART;
        SDLC_Bus: bus [redacted]::Communication_Protocol::SDLC;
        GEN_Bus: bus [redacted]::Communication_Protocol::bus_generic;
        AXI_BUS: bus [redacted]:Communication_Protocol::AXI;


    connections
        C1: bus access fpga.axibus <-> axibus;
        C2: port Acc_Value_x -> Firmware.Acc_Value_x;
        C3: port Acc_Value_y -> Firmware.Acc_Value_y;
        C4: port Acc_Value_z -> Firmware.Acc_Value_z;
        C5: port Acc_Temp_x -> Firmware.Acc_Temp_x;
```

Figura 43 Component Implementation del modulo PL

Sono stati definiti i flow principali della componente PL (figura 42), definiti nella Type.
Gli stessi flow vengono dettagliati nella Implementation della PL (figura 44).
Anche in questo caso viene definito nelle proprietà su quale Platform gira il Firmware e a quale bus sono legate le connessioni interne al sistema.

```
flows
    F1: flow sink Acc_Value_x -> C2 -> Firmware.F1;
    F2: flow sink Acc_Value_y -> C3 -> Firmware.F2;
    F3: flow sink Acc_Value_z -> C4 -> Firmware.F3;
    F4: flow sink Acc_Temp_x -> C5 -> Firmware.F4;
    F5: flow sink Acc_Temp_y -> C6 -> Firmware.F5;
    F6: flow sink Acc_Temp_z -> C7 -> Firmware.F6;
    F7: flow sink Fiber_Temp_x_Raw -> C8 -> Firmware.F7;
    F8: flow sink Fiber_Temp_y_Raw -> C9 -> Firmware.F8;
    F9: flow sink Fiber_Temp_z_Raw -> C10 -> Firmware.F9;
    F10: flow source Firmware.FT -> C14 -> RAW_Data;

    FF: flow sink Inertial_Data_Message -> C15 -> Firmware.FF;

properties
    Actual_Processor_Binding => (reference (fpga)) applies to Firmware;
    actual_connection_binding => (reference (GEN_Bus)) applies to C14;
    actual_connection_binding => (reference (AXI_BUS)) applies to C15;--,C16;
    actual_connection_binding => (reference (SPI)) applies to C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13;
```

Figura 44 Dichiarazione dei flow nell' implementazione del modulo PL

Vengono specificati I flussi degli errori che attraversano la PL (figura 45), per definire in modo chiaro il percorso e quali porte in uscita sono influenzate dagli errori in ingresso.
L'errore che proviene dall'accelerometro si ripercuote in lettura all'interno del Firmware e se non viene compensato e corretto, si propagherà anche attraverso il Software, risultando presente all'interno del pacchetto dati fornito dal Software dopo l'elaborazione e verrà poi propagato verso l'utente tramite la porta SDLC.

```
ef1: error path Acc_Temp_x {OutOfRange} -> OUT_Acc_Temp_x {OutOfRange};
ef2: error path Acc_Temp_x {ItemOmission} -> OUT_Acc_Temp_x {ItemOmission};
ef3: error path Inertial_Data_Message {OutOfRange} -> SDLC {OutOfRange};
ef4: error path Inertial_Data_Message {ItemOmission} -> SDLC {ItemOmission};
```

Figura 45 Definizione della propagazione degli errori nella PL

3.3 PS_Module

Questo modulo rappresenta la parte di elaborazione dati del sistema, il tutto gestito da una CPU sulla quale gira il Software.
I dati vengono passati come un pacchetto di dati grezzi pronti per essere elaborati dagli algoritmi di Coning & Sculling presenti all'interno del Software. I dati *Acc_Temp_x* e *Raw_Data* seguono esattamente lo stesso percorso, poiché *Acc_Temp_x* si trova anche all'interno del pacchetto Raw_Data, sono stati separati per pura chiarezza esplicativa.

```
flows
    --  FCD_Flow: flow path RAW_Data -> Flight_Control_Data_Message;
    IND_Flow: flow path RAW_Data -> Inertial_Data_Message;
    F1: flow path Acc_Temp_x -> Inertial_Data_Message;
```

Figura 46 Flow principali della PS module definiti nella Type

All'interno della Type viene definito il flow *IND_Flow* che rappresenta il tragitto attraverso l'intero modulo PS, dalla porta d'ingresso RAW_Data a quella di uscita Inertial_Data_Message, verrà poi dettagliato nell'implementation specificando quali sottocomponenti attraversa.
Non può mancare la definizione dei flow degli errori attraverso la PS.

```
flows
    --ef0: error sink Acc_Temp_x {OutOfRange, ItemOmission};
    ef1: error path Acc_Temp_x {OutOfRange} -> Inertial_Data_Message {OutOfRange};
    ef2: error path Acc_Temp_x {ItemOmission} -> Inertial_Data_Message {ItemOmission};
    e1: error path RAW_Data {OutOfRange} -> Inertial_Data_Message {OutOfRange};
    e2: error path RAW_Data {ItemOmission} -> Inertial_Data_Message {ItemOmission};
    --              ef2: error sink Temp_3 {OutOfRange, ItemOmission};
```

Figura 47 Error flow definiti nella Component Type della PS

All'interno dell'*implementation* della PS viene definito con maggiore dettaglio il flusso dei dati grezzi, all'interno del quale c'è il dato preso in considerazione per l'analisi di latenza, cioè il Valore dell'accelerometro X, che attraversa il Software per poi uscire filtrato e compensato sulla porta *Inertial_Data_Message*.
Anche in questo caso viene specificato tra le proprietà dove il Software gira (Actual_Processor_Binding), mentre le connessioni interne, C11 e C13 non vengono associate a nessun bus, in questo modo AADL le identifica come istantanee e prive di fattori contributori alla latenza.

```
flows
    --  FCD_Flow: flow path RAW_Data -> C11 ->Software.Filtering_Path_FCD -> C12 ->Flight_Control_Data_Message;
    IND_Flow: flow path RAW_Data -> C11 -> Software.Filtering_Path_IND -> C13 -> Inertial_Data_Message;
--  F4: flow sink Acc_Temp_x;
properties
    --  Actual_Processor_Binding => (reference (cpu)) applies to KRIO_START_CSCI;
    --  Actual_Processor_Binding => (reference (cpu)) applies to KRIO_MAINT_CSCI;
    --  Actual_Processor_Binding => (reference (cpu)) applies to KRIO_NAV_CSCI;
    --  Actual_Processor_Binding => (reference (cpu)) applies to MIO_CONTROLLER;
    Actual_Processor_Binding => (reference (cpu)) applies to Software;
--actual_connection_binding => (reference (GEN_Bus)) applies to C11,C13;
```

Figura 48 Flow e Properties della Component Implementation della PS

All'interno della PS implementation viene definita la macchina a stati composita, che descrive il comportamento, il passaggio di stato, della PS a seconda dello stato in cui finisce il sottocomponente Software.
Anche in questo caso viene definito uno stato di *Failstop* (è stato definito all'interno di quasi tutti gli elementi analizzati), la PS finisce in uno stato di *Failstop* se il Software si ritrova in stato di *Failstop.*

```
composite error behavior
    states

        [Software.Operational]-> Operational;
        [Software.Failstop]-> Failstop;
        [Software.NonCriticalModeFailure]-> NonCriticalModeFailure;
        [Software.CriticalModeFailure]-> CriticalModeFailure;

end composite;
```

Figura 49 Composite error behavior, macchina a stati, della PS

3.4 Firmware

Il Firmware viene dichiarato come processo e ha come sottocomponenti dei thread adibiti alla lettura e alla scrittura. Un thread legge dai sensori e restituisce i dati in uscita, l'altro thread legge i dati elaborati dal Software e li restituisce all'utente.

```
----------------------------------------FIRMWARE FPGA PL Module-------
process Firmware
    features

        Acc_Value_x: in data port [redacted]::Data_types::Accell_Value_X;
        Acc_Value_y: in data port [redacted]::Data_types::Accell_Value_Y;
        Acc_Value_z: in data port [redacted]::Data_types::Accell_Value_Z;
        Acc_Temp_x: in data port [redacted]::Data_types::Accell_Temp_X;
        Acc_Temp_y: in data port [redacted]::Data_types::Accell_Temp_Y;
```

Figura 50 Porte della Component Type del Firmware

```
RAW_Data: out data port [redacted]::Data_types::Message_ID_0x02;
Flight_Control_Data_Message: in data port [redacted]:Data_types::Message_ID_0x01;
Inertial_Data_Message: in data port [redacted]:Data_types::Message_ID_0x02;
UART_Data: in out data port [redacted]:Data_types::Message_ID_0x02;
SDLC: in out data port [redacted]::Data_Types::Message_ID_0x02;
```

Figura 51 Porte principali della Component Type del Firmware

Dopo avere definito tutte le porte e le connessioni all'interno della *Type* e della *Impl*, sono stati definiti i flow principali con la loro relativa implementazione.
Anche in questo caso è importante definire le porte e le connessioni coinvolte, per poi proseguire definendo il propagarsi degli errori all'interno del Firmware e quindi all'interno dei thread.

```
process implementation Firmware.impl
    subcomponents
        READ_Thread: thread Read_Data.impl;
        READ_2_Thread: thread Read_2_Data.impl;

    connections
        C1: port Acc_Temp_x -> READ_Thread.Acc_Temp_x;
        C2: port Acc_Temp_y -> READ_Thread.Acc_Temp_y;
        C3: port Acc_Temp_z -> READ_Thread.Acc_Temp_z;
        C4: port Fiber_Temp_y_Raw -> READ_Thread.Fiber_Temp_y_Raw ;
        C5: port Fiber_Temp_z_Raw -> READ_Thread.Fiber_Temp_z_Raw ;
        C6: port Fiber_Temp_x_Raw -> READ_Thread.Fiber_Temp_x_Raw ;
        C7: port Acc_Value_z -> READ_Thread.Acc_Value_z ;
```

Figura 52 Component Implementation del Firmware e connessioni interne

```
flows
    F1: flow sink Acc_Value_x -> C9 -> READ_Thread.F1;
    F2: flow sink Acc_Value_y -> C8 -> READ_Thread.F2;
    F3: flow sink Acc_Value_z -> C7 -> READ_Thread.F3;
    F4: flow sink Acc_Temp_x -> C1 -> READ_Thread.F4;
    F5: flow sink Acc_Temp_y -> C2 -> READ_Thread.F5;
    F6: flow sink Acc_Temp_z -> C3 -> READ_Thread.F6;
    F7: flow sink Fiber_Temp_x_Raw -> C6 -> READ_Thread.F7;
    F8: flow sink Fiber_Temp_y_Raw -> C4 -> READ_Thread.F8;
    F9: flow sink Fiber_Temp_z_Raw -> C5 -> READ_Thread.F9;
    FT: flow source READ_Thread.F8 -> C13 -> RAW_Data;
    FF: flow sink Inertial_Data_Message -> C14 -> READ_2_Thread.FF;
```

Figura 53 Flow definiti nella Component Implementation del Firmware

Come si può notare nella sezione errori del Firmware (figura 54), vi è il flow path dell'errore **"ef1"**, in ingresso sulla porta *Acc_Temp_x* e che si propaga in uscita sulla porta *OUT_Acc_Temp_x*. Una volta elaborato dal Software ritorna nel Firmware sulla porta *Inertial_Data_Message*, che come già spiegato rappresenta che su questa porta entra un pacchetto dati il quale contiene anche il dato *Acc_Temp_x*, e quindi che l'errore, se non corretto, si è propagato anche sul pacchetto dei dati elaborati per l'utente.
L'errore poi si propaga sulla porta SDLC, come mostrato dall'error path **"ef2".**

```
flows
    --ef0: error sink Acc_Temp_x {OutOfRange, ItemOmission};
    ef1: error path Acc_Temp_x {OutOfRange, ItemOmission} -> OUT_Acc_Temp_x {OutOfRange, ItemOmission};
    ef2: error path Inertial_Data_Message {OutOfRange, ItemOmission} -> SDLC {OutOfRange, ItemOmission};
```

Figura 54 Error flow definito nella Implementation del Firmware

3.5 Read Thread

Le operazioni di lettura dei dati dalla sensoristica inerziale, come già detto, vengono eseguite dal Firmware, più precisamente dal sottoprogramma Read_Thread. Si tratta di una porzione di codice che tramite BUS SPI legge e immagazzina i dati per fornirli poi al Software

Questi thread come anche I thread di C&S hanno nella loro definizione le proprietà relative alle loro tempistiche di elaborazione, esecuzione e tempo vitale. Queste proprietà sono necessarie perché determinano parte dei contributori alla latenza, contributori che esistono a causa dei ritardi fisici esistenti sui dispositivi reali.

```
properties
    Dispatch_Protocol => Periodic;
    Period => 250us;
    compute_execution_time => 150us .. 300us;
    reference_processor => classifier (::platform::ecu);
    sei::instructionsperdispatch => 1.24 kipd .. 1.25 mipd;
annex EMV2 {**
        use types ErrorLibrary, Error_annex;
        use behavior Error_annex::ThreeState;

        error propagations
            Acc_Temp_x: in propagation {OutOfRange, ItemOmission};
            OUT_Acc_Temp_x : out propagation {OutOfRange, ItemOmission};
            Inertial_Data_Message: in propagation {OutOfRange, ItemOmission};


            flows
                --ef0: error sink Acc_Temp_x {OutOfRange, ItemOmission};
                ef1: error path Acc_Temp_x {OutOfRange, ItemOmission} -> OUT_Acc_Temp_x {OutOfRange, ItemOmission};


        end propagations;

  **};
```

Figura 55 Porzione degli errori della Component Type del Read Thread

3.6 Software

Anche il Software viene definito in AADL come un process, il quale contiene i sottocomponenti thread *CONING_SCULLING_Stage1* e *CONING_SCULLING_Stage2*. Il Monitor rappresenta un thread di prova per gestire la segnalazione in uscita della presenza di errori, ma non sarà trattato in questa tesi, poiché non utile alle analisi richieste.

```
process implementation Software.impl
    subcomponents
        CONING_SCULLING_Stage1: thread Coning_Sculling_S1.impl;
        CONING_SCULLING_Stage2: thread Coning_Sculling_S2.impl;
        Monitor: thread Monitor;

    connections
        C1: port RAW_Data -> CONING_SCULLING_Stage1.RAW_Data;
        C2: port CONING_SCULLING_Stage1.RAW_2_Data -> CONING_SCULLING_Stage2.RAW_2_Data;
        C3: port CONING_SCULLING_Stage2.Inertial_Data_Message -> Inertial_Data_Message;
        C4: port CONING_SCULLING_Stage1.Flight_Control_Data_Message -> Flight_Control_Data_Message;

        Software_impl_new_connection: port Acc_Temp_x -> Monitor.Acc_Temp_x;
```

Figura 56 Component Implementation del Software

Viene definito un flow path chiamato *Filtering_Path_IND* che rappresenta il percorso del dato all'interno della componente software, quindi da quando entra dalla porta *RAW_Data* a quando esce, verso il Firmware, sulla porta *Inertial_Data_Message*, attraversando i due stadi di C&S.

```
flows
    Filtering_Path_IND: flow path RAW_Data -> C1 -> CONING_SCULLING_Stage1.F2 -> C2 -> CONING_SCULLING_Stage2.F1 -> C3 -> Inertial_Data_Message {latency => 1ms .. 5ms;};
```

Figura 57 Flow path interno della Software Implementation

Ovviamente deve essere specificata anche la parte inerente agli stati in cui il componente Software può trovarsi, specificando che la macchina a stati *Composite* del Software dipende dagli stati possibili dei suoi sottocomponenti. Se uno o più thread finiscono nello stato di *Failstop*, allora anche il Software finirà in uno stato di Failstop (così è stato deciso di modellare il sistema, ma ogni sistema va modellato secondo le proprie esigenze e realtà).

```
annex EMV2 {**
        use types ErrorLibrary;
        use behavior Error_annex::ThreeState;

        composite error behavior
            states
                [1 ormore (CONING_SCULLING_Stage1.NonCriticalModeFailure, CONING_SCULLING_Stage2.NonCriticalModeFailure)]-> NonCriticalModeFailure;
                [1 ormore (CONING_SCULLING_Stage1.CriticalModeFailure, CONING_SCULLING_Stage2.CriticalModeFailure,
                           CONING_SCULLING_Stage1.Failstop, CONING_SCULLING_Stage2.Failstop)]-> Failstop;
                [Monitor.TransientFailure]-> NonCriticalModeFailure;
                [Monitor.Failed]-> Failstop;
                [Monitor.Warning_Error]-> NonCriticalModeFailure;
        end composite;

            **};
```

Figura 58 Composite error behavior del Software

Il primo stadio dell'algoritmo di C&S prende in ingresso i dati grezzi letti dal Firmware, esegue una prima elaborazione e passa i nuovi dati *RAW_2_Data* al secondo stadio dell'algoritmo.
Oltre a questo, fornisce in uscita anche un pacchetto Dati *Flight_Control_Data_Message* che vanno direttamente all'utente e saltano il secondo stadio di compensazione.

```
-------------------------------------Coning_Sculling_S1---------------------------------
thread Coning_Sculling_S1
    features
        RAW_Data: in data port KRIO::Data_Types::Message_ID_0x02;
        RAW_2_Data: out data port KRIO::Data_Types::Message_ID_0x02;
        Flight_Control_Data_Message: out data port [redacted]::Data_Types::Message_ID_0x01;

    flows
        F1: flow path RAW_Data -> Flight_Control_Data_Message;
        F2: flow path RAW_Data -> RAW_2_Data;
    properties
        Dispatch_Protocol => Periodic;
        Period => 250us;
        compute_execution_time => 150us .. 300us;
--      reference_processor => classifier (::platform::ecu);
--      sei::instructionsperdispatch => 1.24 kipd .. 1.25 mipd;

    annex EMV2 {**
            use types ErrorLibrary, Error_annex;
            use behavior Error_annex::ThreeState;

            error propagations
                RAW_Data: in propagation {OutOfRange, ItemOmission};
                RAW_2_Data : out propagation {OutOfRange, ItemOmission};

                Flight_Control_Data_Message : out propagation {OutOfRange, ItemOmission};

                flows
                    --ef0: error sink Acc_Temp_x {OutOfRange, ItemOmission};
                    ef1: error path RAW_Data {OutOfRange} -> RAW_2_Data {OutOfRange};
                    ef2: error path RAW_Data {ItemOmission} -> Flight_Control_Data_Message {ItemOmission};
                    ef3: error path RAW_Data {OutOfRange} -> RAW_2_Data {OutOfRange};
                    ef4: error path RAW_Data {ItemOmission} -> Flight_Control_Data_Message {ItemOmission};

            end propagations;
```

Figura 59 Component Type del Thread C&S S1

Anche i thread presenti all'interno del Software vanno modellati inserendo la parte degli errori e la macchina a stati che li descrive. Si è scelto di impostare lo stato di Failstop solo se un dato presente sul pacchetto/porta RAW_Data è mancante (ItemOmission), in questo caso il dato che è affetto da errore è quello di temperatura.

```
component error behavior
    events
        Reset: recover event;
    transitions
        t01: Operational-[RAW_Data{OutOfRange}]-> NonCriticalModeFailure;
        t02: Operational-[RAW_Data{ItemOmission}]-> CriticalModeFailure;
        t3: CriticalModeFailure-[RAW_Data{ItemOmission}]-> FailStop;
end component;

properties

    emv2::OccurrenceDistribution => [ProbabilityValue => 3.01e-7; Distribution => Poisson;] applies to RAW_Data.itemomission;
    emv2::OccurrenceDistribution => [ProbabilityValue => 2.01e-9; Distribution => Poisson;] applies to RAW_Data.OutOfRange;
    emv2::OccurrenceDistribution => [ProbabilityValue => 0.01e-10; Distribution => Poisson;] applies to RAW_2_Data.OutOfRange;
    emv2::OccurrenceDistribution => [ProbabilityValue => 0.01e-7; Distribution => Poisson;] applies to RAW_2_Data.itemomission;
    emv2::hazards => ([crossreference => "F_RAW_DATA 1.00";
            failure => "ItemOmission";
            phases => ("all");
            description => "No data from the one of the measurment device";
            comment => "Reset the sistem if lost data during acquisition";]) applies to RAW_Data.itemomission;
    emv2::hazards => ([crossreference => "F_RAW_DATA 2.00";
                failure        => "OutOfRange";
                phases         => ("all");
                description    => "Data from the one of the measurment device is Out of Range";
                comment        => "Look at alimentation or restart the system and recalibrate sensor";]) applies to RAW_Data.OutOfRange;
```

Figura 60 Component error behavior e proprietà degli errori del thread C&S S1

Il secondo stadio di C&S è modellato similarmente al primo appena mostrato, ovviamente con le sue porte, le sue connessioni, i suoi flow, i flow degli errori che affliggono i dati che lo attraversano e la sua macchina a stati.
Una volta che ogni elemento è stato modellato con i dettagli necessari, si deve controllare che tutte le connessioni siano legate ad un bus o ad un protocollo di comunicazione, e per ultimo ma non per importanza vanno specificate le proprietà di coloro che sono gli elementi contributori di Latenza.
Si può ora passare all'esecuzione delle varie tipologie di analisi. Il sistema andrà istanziato e sull'istanza verrà specificato se ci sono degli errori semantici non appena verrà creata. Se tutto si conclude senza intoppi si possono eseguire le analisi sull'istanza appena creata.
Come creare l'istanza e come eseguire le varie analisi, compresi chi sono e quali proprietà necessitano i contributori, verrà spiegato nei capitoli successivi.

3.7 Conclusione

AADL e il suo ambiente di sviluppo OSATE2 sono risultati essere uno strumento estremamente utile ed efficiente nello studio di sistemi safety-critical e non solo, permettendo di modellare qualsiasi tipo di sistema, mettendo in evidenza le componenti HW e SW e le interazioni fra di esse. Il linguaggio fornito da AADL permette la creazione di modelli estremamente realistici, poiché fornisce gli strumenti per una modellazione a bassissimo livello e quindi di realizzare modelli fedelissimi alla realtà.

I benefici nell'usare AADL includono:

1. Predizione e validazione di caratteristiche runtime come accessibilità, tempestività di risposta e sicurezza.
2. Validazione di architetture di sistema e implementazioni
3. Potenzia lo sviluppo di processi attraverso un singolo modello architetturale dettagliato
4. Portabilità dei modelli su altre piattaforme di lavoro e altri linguaggi
5. Permette analisi valide di sistemi real-time, embedded e ad alta fedeltà.

Le analisi eseguite sul modello della IMU di Civitanavi sono risultate ottimali rispettando le aspettative richieste.

Una volta compresa la grammatica di AADL risulta molto semplice creare modelli ad alto livello e questo velocizza le operazioni di progettazione, permettendo delle analisi preliminari senza troppo dispendio di energie e risorse.

Ma AADL risulta essere efficiente e valido anche nella modellazione a bassissimo livello e le conseguenti analisi, rendendolo così un utile e consigliatissimo strumento per la progettazione di sistemi di qualsiasi azienda o industria.

APPENDICE

APPENDICE 1: Tabelle riassuntive delle gerarchie fra elementi AADL

Category Group	Component Category	Permitted Subcomponents	Permitted Subcomponent of
Software	process	thread data thread group	system
	thread	data	process thread group
	data	data	process thread data thread group system
	thread group	data thread thread group	process thread group
	subprogram	None allowed	None
Execution Platform	processor	memory	system
	memory	memory	processor memory system
	bus	None allowed	system
	device	None allowed	system
Composite	system	process data processor memory bus device system	system

Feature		Allowed Feature of Component or Component Category
port **port group**	all port types	• system • process • thread • thread group • processor • device
	• event port • event data port • port group (events only)	subprogram (component)
subprogram	server	• system • process • thread • thread group • processor • device
	subprogram (data)	data
access	provides data	• system • process • thread • thread group • data
	requires data	• system • process • thread • thread group • subprogram (component)
	provides bus	system
	requires bus	• system • processor • memory • bus • device
parameter		subprogram (component)

Category Group	Component Category	Allowed Features
Software	process	• server subprogram • port/port group • provides data access • requires data access
	thread	• server subprogram • port/port group • provides data access • requires data access
	data	• subprogram • provides data access
	thread group	• server subprogram • port/port group • provides data access • requires data access
	subprogram	• out event port • out event data port • port group (event only) • requires data access • parameter
Execution Platform	processor	• server subprogram • port/port group • requires bus access
	memory	requires bus access
	bus	requires bus access
	device	port/port group • server subprogram • requires bus access
Composite	system	• server subprogram • port/port group • provides data access • provides bus access • requires data access • requires bus access

APPENDICE 2: Libreria EMV2 ridotta e modificata per il caso di studio

Pacchetto contenente la libreria degli errori modificata da librerie già esistenti, contiene le tipologie di errore e le macchine a stati relative al Sistema.
Sono stati creati sia gli eventi sia gli stati che descrivono nella maniera più realistica possibile il comportamento reale del Sistema, il tutto ovviamente semplificato per ragioni di studio.

```
package Error_annex
  public
       annex EMV2 {**
             error types
                   NoPower: type;
                   NoService: type;
                   ValueError: type;
                   NoValue: type extends ValueError;
                   IncorrectData: type;
                   PlatformFailure: type;
                   HardwareFailure: type extends PlatformFailure;
                   SoftwareFailure: type extends PlatformFailure;
                   LOW_Power: type;
                   Power_Issue: type;
             end types;

             error behavior ThreeState
                   events
                         Failure: error event;
                         Red_Failure: error event;
                         Critical_Failure: error event;
                         Error_Power_15V: error event;
                         Error_Power_6V: error event;
                         Error_Power_5V: error event;
                         Stop: error event;
                         Brake: error event;
                         ResetEvent: recover event;
                         Reset: recover event;
                   states
                         Operational: initial state;
                         Failstop: state;
                         Error_15V_power: state;
                         Error_15V_LOW_Power: state;
                         Error_6V_power: state;
                         Error_6V_LOW_Power: state;
                         Error_5V_power: state;
                         Error_5V_LOW_Power: state;
                         Gyro_error_power: state;
                         NonCriticalModeFailure: state;
                         NCritF_15V: state;
                         NCritF_6V: state;
                         NCritF_5V: state;
                         CritF_15V: state;
                         CritF_6V: state;
                         CritF_5V: state;
                         CriticalModeFailure: state;

             end behavior;
```

```
error behavior ThreeErrorStates
                    events
                          BadValueEvent: error event;
                          NoValueEvent: error event;
                          LateValueEvent: error event;
                    states
                          Operational: initial state;
                          BadValueState: state;
                          NoValueState: state;
                          LateValueState: state;
                    transitions
                    BadValueTransition: Operational -[BadValueEvent]->
BadValueState;
                    NoValueTransition: Operational -[NoValueEvent]-> NoValueState;
                    LateTransition: Operational -[LateValueEvent]->
LateValueState;

             end behavior;

             error behavior Simple
                    states
                          Operational: initial state;
                          Failstop: state;
                          Failed: state;
             end behavior;


             error behavior SimpleAndTransient
                    events
                          Failure: error event;
                          FailureTransient: error event;
                          Warning_Out_of_Range: error event;
                          Critical_Out_of_Range: error event;
                          Critical_Error: error event;
                          ResetEvent: recover event;
                    states
                          Operational: initial state;
                          Failed: state;
                          Warning_Error: state;
                          TransientFailure: state;
                    transitions
                      transerr: Operational -[Failure]-> Failed;

                      transerr2: Operational -[FailureTransient]->
                    TransientFailure;
                      treset: TransientFailure -[ResetEvent]-> Operational;
             end behavior;
      **};
end Error_annex;
```

APPENDICE 3: tipologie di componenti

COMPOSITE

SYSTEM

```
FAM: system IMU::Hardware::IMU_Hardware.impl;
ASE200: system IMU::ASE200_Board::IMU_ASE_Board.impl;
Sensing_Unit: system IMU::Hardware::Sensing_Elements.impl;
POWER_Board: system IMU::Power_Unit::Power_Board.impl;
[redacted] system IMU::SoC::SoC.impl;
Gyroscope: system Gyro.impl;
```

HARDWARE

PROCESSOR

```
cpu: processor SoC_CPU.impl;
fpga: processor SoC_fpga.impl;
cpu: processor IMU::SoC::SoC_CPU.impl;
```

DEVICE

```
EMI_FLT: device EMI_Filter.impl;
SURGE_STOP: device Surge_Stopper.impl;
DCDC_15V: device DC_DC_15V.impl;
DCDC_6V: device DC_DC_6V.impl;
DCDC_5V: device DC_DC_5V.impl;
channel_8_AD_1: device channel_8_AD.impl;
channel_8_AD_2: device channel_8_AD.impl;
FOG_AD_Converter: device FOG_AD.impl;
DIGITAL_PWR: device Digital_Power_Supply.impl;
accel_x: device accel.impl;
accel_y: device accel.impl;
accel_z: device accel.impl;
temp_accel_x: device Temperature_sensor.impl;
temp_accel_y: device Temperature_sensor.impl;
temp_accel_z: device Temperature_sensor.impl;
Coil_x_Temp_Sensor: device Temperature_Sensor.impl;
Coil_y_Temp_Sensor: device Temperature_Sensor.impl;
Coil_z_Temp_Sensor: device Temperature_Sensor.impl;
MIOC_x: device MIOC_XYZ.impl;
MIOC_y: device MIOC_XYZ.impl;
MIOC_z: device MIOC_XYZ.impl;
Fiber_Coil_x: device Fiber_Coil.impl;
Fiber_Coil_y: device Fiber_Coil.impl;
Optical_unit: device IMU::ASE200_Board::Optical_Circuit.impl;
Fiber_Coil_z: device Fiber_Coil.impl;
Pinfet: device Pinfet_Circuit.impl;
Filter_1: device IMU::Hardware::Filter.impl;
Fiber_Coil_z: device Fiber_Coil.impl;

…
```

…
…

MEMORY

```
SDRAM: memory Flash_SDRAM.impl;
…
```

BUS

```
AXI_BUS: bus AXI_bus.impl;
…
```

SOFTWARE

PROCESS

```
firmware: process IMU::Firmware::IMU_firmware.impl;
IMU_START_CSCI: process IMU::IMU_SW_START_CSCI::IMU_START_CSCI.impl;
IMU_MAINT_CSCI: process IMU::IMU_SW_MAINT_CSCI::IMU_MAINT_CSCI.impl;
IMU_NAV_CSCI: process IMU::IMU_SW_NAV_CSCI::IMU_NAV_CSCI.impl;
```

THREAD

```
BOOT: thread Boot.impl;
CSCI_Selection: thread CSCI_Selection.impl;
BIT_Stream_VAL_ACT: thread BIT_Stream_VAL_ACT.impl;
CSCI_VALIDATION: thread CSCI_Validation.impl;
CSCI_ACTIVATION: thread CSCI_Activation.impl;
```

SUBPROGRAM

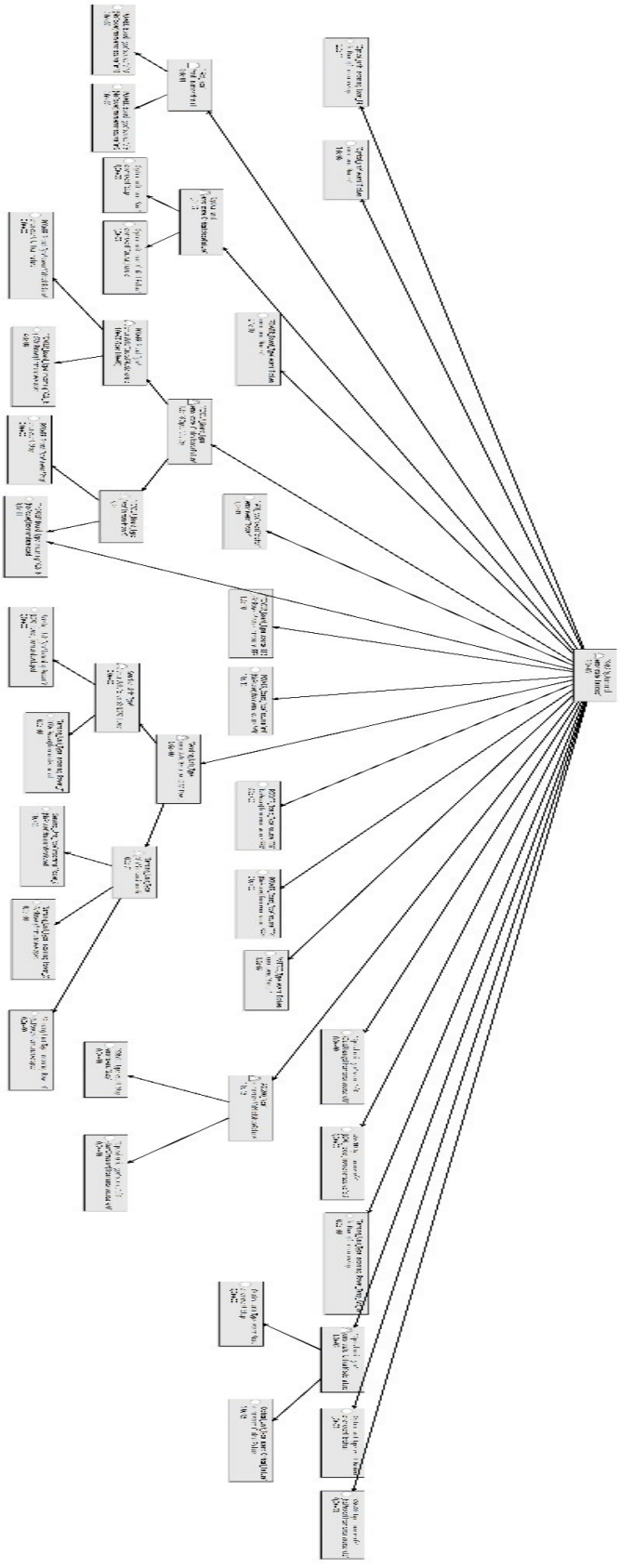
DATA

```
data UINT32_TYPE_AADL
  properties
    Type_Source_Name => "__po_hi_uint32_t";
    Source_Text => ("types");
    Data_Size => 4 Bytes;
  end UINT32_TYPE_AADL;

  data FLOAT32_TYPE_AADL
  properties
    Type_Source_Name => "__po_hi_float32_t";
    Source_Text => ("types");
    Data_Size => 4 Bytes;
  end FLOAT32_TYPE_AADL;
end IMU::Data_types;
```

…

APPENDICE 4: Fault Tree



APPENDICE 5: Macchine a stati dei vari elementi che contribuiscono allo stato di Failstop nel FTA

IMU_System

```
composite error behavior
    states

        [POWER_Board_Type.Failstop]-> Failstop;
        [Optical_unit.Failstop]-> Failstop;
        [FAM_Type.Failstop]-> Failstop;
        [ASE200_Type.Failstop]-> Failstop;
        [Sensing_Unit_Type.Failstop]-> Failstop;
        [Optical_unit_Type.Failstop]-> Failstop;

end composite;
```

FAM Type

```
component error behavior
    events
        Failure: error event;
        Critical_Failure: error event;
        Reset: recover event;
        Stop: error event;
    transitions
        t0: Operational -[Power_15V{NoPower}]-> Failstop;
        t1: Operational -[Power_6V{NoPower} and Power_5V{NoPower}]-> Failstop;
        t2: Operational -[Power_5V{NoPower}]-> CritF_5V;
        t3: Operational -[Power_15V{LOW_Power}]-> NCritF_15V;
        t4: Operational -[Power_6V{LOW_Power}]-> NCritF_6V;
        t5: Operational -[Power_5V{LOW_Power}]-> NCritF_5V;
        t01: Operational -[Power_Good_15V{NoPower}]-> Failstop;
        t12: Operational -[Power_Good_6V{NoPower}]-> Failstop;
        t23: Operational -[Power_Good_5V{NoPower}]-> Failstop;
        t34: Operational -[Power_Good_15V{LOW_Power}]-> NCritF_15V;
        t45: Operational -[Power_Good_6V{LOW_Power}]-> NCritF_6V;
        t56: Operational -[Power_Good_5V{LOW_Power}]-> NCritF_5V;
        t67: Operational -[Broken]-> Failstop;
        t6: CriticalModeFailure -[Reset]-> Operational;
        t7: Failstop -[Reset]-> Operational;

end component;
```

FAM Impl

```
composite error behavior
    states

        [redacted]impl.Operational]-> Operational;
        [redacted]impl.Failstop]-> Failstop;
        [redacted]impl.NonCriticalModeFailure]-> NonCriticalModeFailure;
        [redacted]impl.CriticalModeFailure and [redacted]impl.NonCriticalModeFailure]-> Failstop;
        [redacted]impl.CriticalModeFailure ]-> CriticalModeFailure;

end composite;
```

Sensing Unit Type

```
component error behavior
    events
        Reset: recover event;
    transitions
        t0: Operational -[Power_X{NoPower} or Power_Y{NoPower} or Power_Z{NoPower}]-> Error_5V_power;
        t11: Operational -[Power_Temp_Acc_5V{NoPower}]-> Failstop;
        t1: Operational -[Power_Temp_Acc_5V{LOW_Power}]-> NCritF_5V;
        t2: Operational -[Power_Y{LOW_Power}]-> Error_5V_LOW_Power;
        t3: Operational -[Power_Z{LOW_Power}]-> Error_5V_LOW_Power;
        t4: Operational -[Error_Power_5V]-> Error_5V_power;
        t5: Error_5V_LOW_Power -[ResetEvent]-> Operational;
        t6: Error_5V_LOW_Power -[Power_X{NoPower} or Power_Y{NoPower} or Power_Z{NoPower}]-> FailStop;
end component;
```

Sensing Unit Impl.

```
composite error behavior
    states
        [temp_accel_x.Failstop or temp_accel_y.Failstop or temp_accel_z.Failstop]-> Failstop;
        [temp_accel_x.CritF_5V or temp_accel_x.CriticalModeFailure]-> CriticalModeFailure;
        [temp_accel_x.NCritF_5V]-> NCritF_5V;

end composite;
```

Sensore di Temperatura Accel_X Type

```
component error behavior
    events
        Reset: recover event;
    transitions
        t0: Operational-[accel_failure]-> FailStop;
        t01: Operational-[Power_5V{LOW_Power}]-> CriticalModeFailure;
        t02: Operational-[Power_5V{NoPower}]-> Failstop;
        t03: NCritF_5V-[Reset]-> Operational;
        t04: CritF_5V-[Broken]-> FailStop;
        t05: CritF_5V-[Reset]-> Operational;
        t1: FailStop -[Reset]-> Operational;
        t2: Operational-[Critical_Failure]-> CriticalModeFailure;
        t3: CriticalModeFailure-[Stop]-> FailStop;
end component;
```

Optical Unit Type

```
component error behavior
    events
        Reset: recover event;
    transitions
        t0: Operational-[Broken]-> FailStop;
        t01: Operational-[Laser_IN{NoPower}]-> FailStop;
        t02: Operational-[Laser_IN{LOW_Power}]-> CritF_15V;
        t04: CritF_15V-[Broken]-> FailStop;
        t05: CritF_15V-[Reset]-> Operational;
        t1: FailStop -[Reset]-> Operational;
        t2: Operational-[Critical_Failure]-> CriticalModeFailure;
        t3: CriticalModeFailure-[Stop]-> FailStop;
end component;
```

ASE Board Type

```
component error behavior
    events
        Reset: recover event;
    transitions
        t0: Operational-[Broken]-> FailStop;
        t01: Operational-[Optical_x_OUT{OutOfRange}]-> FailStop;
        t02: Operational-[Optical_x_OUT{ItemOmission}]-> CriticalModeFailure;
        t3: CriticalModeFailure-[Stop]-> FailStop;
end component;
```

Processore SoC Type

```
  component error behavior
      events
          Failure: error event;
          Critical_Failure: error event;
          Reset: recover event;
          Stop: error event;
      transitions
          t0_1: Operational -[Power_15V{LOW_Power}]-> NonCriticalModeFailure;
          t0: Operational -[Acc_Temp_x{OutOfRange}]-> NonCriticalModeFailure;
          t1: Operational -[Power_15V{NoPower}]-> Failstop;
          t2: Operational -[Acc_Temp_x{ItemOmission}]-> CriticalModeFailure;
          t3: NonCriticalModeFailure -[ResetEvent]-> Operational;
          t4: NonCriticalModeFailure -[Power_15V{NoPower}]-> Failstop;
          t5: CriticalModeFailure -[Stop]-> Failstop;
          t6: CriticalModeFailure -[Reset]-> Operational;
          t7: Failstop -[Reset]-> Operational;

  end component;
```

Processore SoC Impl

```
composite error behavior
    states

        [PS_MODULE_impl.Operational]-> Operational;
        [PS_MODULE_impl.Failstop or PL_MODULE_impl.Failstop]-> Failstop;
        [PS_MODULE_impl.NonCriticalModeFailure]-> NonCriticalModeFailure;
        [PS_MODULE_impl.CriticalModeFailure]-> CriticalModeFailure;

end composite;
```

PS MODULE Type e Impl.

```
    component error behavior
        events
            Failure: error event;
            Critical_Failure: error event;
            Reset: recover event;
            Stop: error event;
        transitions
        --  t0: Operational -[Failure]-> Operational;
            t1: Operational -[Red_Failure]-> NonCriticalModeFailure;
            t2: Operational -[Critical_Failure]-> CriticalModeFailure;
            t3: NonCriticalModeFailure -[ResetEvent]-> Operational;
            t4: NonCriticalModeFailure -[Critical_Failure]-> CriticalModeFailure;
            t5: CriticalModeFailure -[Stop]-> Failstop;
            t6: CriticalModeFailure -[Reset]-> Operational;
            t7: Failstop -[Reset]-> Operational;

    end component;

    composite error behavior
        states

            [Software.Operational]-> Operational;
            [Software.Failstop]-> Failstop;
            [Software.NonCriticalModeFailure]-> NonCriticalModeFailure;
            [Software.CriticalModeFailure]-> CriticalModeFailure;

    end composite;
```

APPENDICE 6: Lista delle Immagini

Bibliografia

- Delange J. AADL IN PRACTICE: design and validate the architecture of critical systems. United State of America. Reblochon Development Company. 2017.

- Feiler P. SAE AADL V2: An Overview. Pittsburgh. Software Engineering Institute-Carnegie Mellon University

- Feiler P, Hudak J, Delange J, Gluch D. Architecture Fault Modeling and Analysis with the Error Model Annex, Version 2. Pittsburgh. Software Engineering Institute-Carnegie Mellon University. 2016.

- Feiler P. Hansson J. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Pittsburgh. Software Engineering Institute-Carnegie Mellon University. 2007.

- Feiler P. Hudak J. Developing AADL Models for Control Systems: A Practitioner's Guide. Pittsburgh. Software Engineering Institute-Carnegie Mellon University. 2007.

- Kordon F. Hugues J. Canals A. Dohet A. Part 4 AADL. Embedded Systems: Analysis and Modeling with SysML, UML and AADL. Wiley-ISTE. 2013.

- Larsen M. Modelling fiel robot software using AADL. Aarhus University. 2016.

- Adventium LABS. Integrated AADL Analysis. 2018

- Hansson J. Greenhouse A. Modeling and Validating Security and Confidentiality in System Architectures. Pittsburgh. Software Engineering Institute-Carnegie Mellon University. 2008.

- Bozzano M. Cimatti A. Katoen J.P. Noll T. Safety, Dependability and Performance Analysis of Extended AADL Models. The Computer Journal. 2011.

- Feiler P. Hudak J. The Architecture Analysis & Design Language (AADL): An Introduction. Software Engineering Institute-Carnegie Mellon University. 2006.

- Cicchetti S. CRANE-T Software Requirements Specification. Civitanavi Systems S.r.l. 2017

- Quatraro E. CRANE-T System Architecture Description. Civitanavi Systems S.r.l. 2018

- Camilletti M. CRANE-T Interface Control Document. Civitanavi Systems S.r.l. 2018