Politecnico di Torino

Corso di Laurea Magistrale
in Ingegneria Energetica e Nucleare

Tesi di Laurea Magistrale

Sviluppo di una strategia di interfaccia per i diversi moduli del codice 4C

Relatore
Prof. Antonio Froio
Co-relatore
Prof. Roberto Bonifetto

Candidato
Luca Marasco Belcastro

Dicembre 2021

Sommario

Capitolo 1

1 INTRODUZIONE

1.1 PROGETTI ITER E DEMO

EU DEMO, *European Union Demonstration Power Plant*, è un prototipo di centrale elettrica a fusione il cui scopo principale è di dimostrare la possibilità di produrre potenza elettrica sfruttando le reazioni di fusione nucleare tra gli isotopi di idrogeno, trizio e deuterio, in stato di plasma ad alta temperatura (milioni di gradi celsius). Il flusso termico che deriva da tali reazioni viene estratto da un fluido termovettore, elio, metalli liquidi o gas, e poi ceduto nuovamente in uno scambiatore di calore posto tra il circuito primario e il secondario, consentendo la produzione di gas ad alta pressione, che evolvendo in turbina e muovendo un alternatore, permette la produzione di energia elettrica. È pensato come successore di ITER (Fig. 1.1), *International Thermonuclear Experimental Reactor*, reattore a fusione in costruzione a Cadarache, nel sud della Francia, il cui "primo plasma" è stimato possa avvenire nel dicembre 2025 [1].

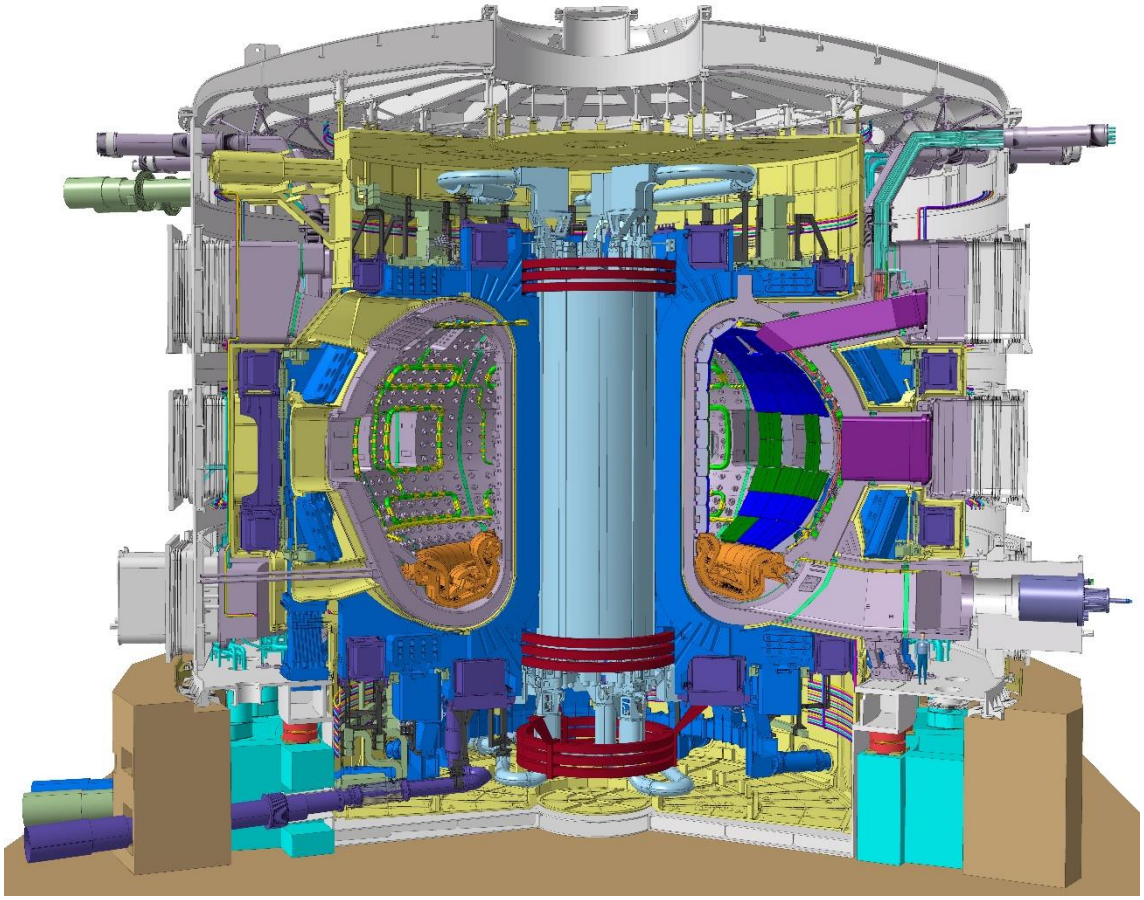


Fig. 1.1 - Sezione trasversale di ITER

In ITER il *fattore di guadagno Q*, definito come il rapporto tra la potenza termica generata dalla reazione di fusione ϕ_N e quella utilizzata per ionizzare la miscela DT e far avvenire la reazione ϕ_H sarà

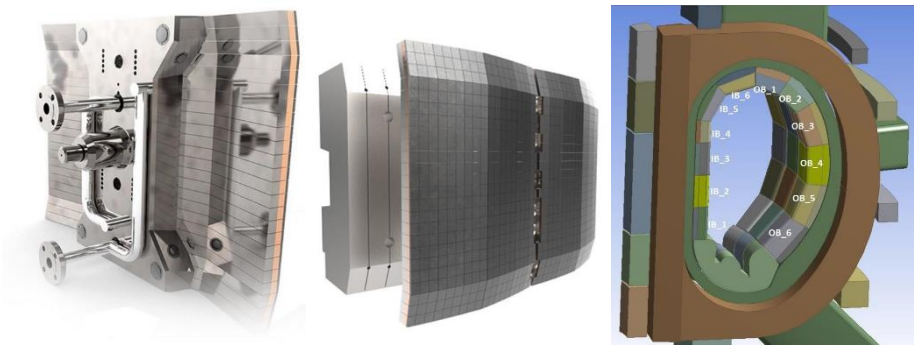
$$\mathrm{Q}_{breakeven} \triangleq 1 < \mathrm{Q}_{ITER} = \left(\frac{\Phi_\mathrm{N}}{\Phi_\mathrm{H}}\right)_{op} > 10$$

L'obiettivo è mantenere tale rapporto stabile per un intervallo di tempo di 400-600 s [2], mentre in DEMO si prevede un Q_{DEMO} pari a 25 sostenuto per un periodo di almeno 1000 s; in entrambi gli impianti il fattore di guadagno è ampiamente superiore all'unità, detta condizione di "breakeven" (o di pareggio). I fattori che ne condizionano il valore dipendono dai limiti ingegneristici delle tecnologie utilizzate per governare i fenomeni fisici.

Una volta innescata la fusione all'interno del tokamak saranno presenti un insieme di particelle cariche (particelle α, elettroni e^-, ioni di H) che, sottoposti al campo magnetico, ne saranno influenzati e rimarranno confinati all'interno del vessel, mentre le particelle non cariche, ovvero i neutroni liberati dalla reazione di fusione, attraverseranno il plasma e dissiperanno la loro

energia cinetica alle “prime pareti” del blanket (ovvero l’insieme dei moduli, collocati sulla parete interna del vessel, mostrato in Fig. 1.2).

In DEMO e nelle fasi finali di sperimentazione in ITER, saranno presenti dei moduli in grado di autogenerare trizio, Fig. 1.2c.



(a) (b) (c)

Fig. 1.2 – Pannello di prima parete (a) - Shield block (b) - Moduli autofertilizzanti (c)

Tale potenza termica in ITER verrà rimossa con acqua a 70 °C in pressione a 4 MPa, mentre in DEMO sarà asportata dal fluido termovettore (elio o PbLi liquido) per la generazione di potenza elettrica. Per compensare la perdita di masse e di energia dal plasma, il fattore di guadagno dev’essere almeno maggiore di 3. Un’altra perdita che fa aumentare il valore di Q è dovuta alle condizioni di funzionamento dei magneti che per essere utilizzati devono funzionare in regime di super-conduttività.

1.2 IL SISTEMA MAGNETI DI ITER

Il sistema di confinamento avviene mediante magneti composti al loro interno da fasce di fili di superconduttori, chiamati “cable-in-conduit conductors” (abbr. CICC, mostrato in Fig. 1.3) e saranno attraversati da una corrente elettrica che può raggiungere i 70 kA.

Fig. 1.3 - CICC

Gli alimentatori dei magneti (Fig. 1.4) convogliano i fluidi criogenici per raffreddare e controllare la temperatura dei magneti. In totale vi sono 31 alimentatori dei superconduttori che trasmettono energia elettrica. Le sbarre superconduttive sono progettate per assorbire grandi variazioni di temperatura durante il raffreddamento della macchina, da temperatura ambiente fino a circa 4 K (circa -269 °C).

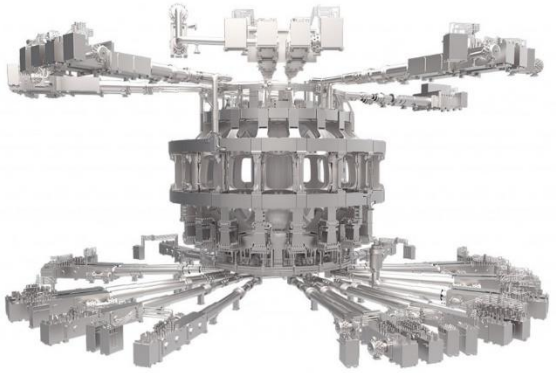


Fig. 1.4 – Alimentatore dei magneti

Il sistema magneti di ITER può essere suddiviso in 4 sottosistemi:

1) Il sistema a campo toroidale (Fig. 1.5b) è costituito, in ITER, da diciotto magneti a forma di "D"; la sua principale funzione è quella di confinare il plasma all'interno del vessel. Gli avvolgimenti di campo toroidali sono progettati per produrre un'energia magnetica totale di 41 GJ e un campo magnetico massimo di 11,8 T. I magneti sono avvolti in strati di conduttori a spirale incorporati in piastre radiali e racchiusi in grandi strutture di acciaio inossidabile.
2) Il solenoide centrale (Fig. 1.5a) consente di mantenere una potente corrente del plasma di 15 MA per una durata di 300-500 secondi. Il campo magnetico massimo, circa 13 T, sarà raggiunto al centro dei moduli rendendo il solenoide centrale il più potente sistema magnetico di ITER.

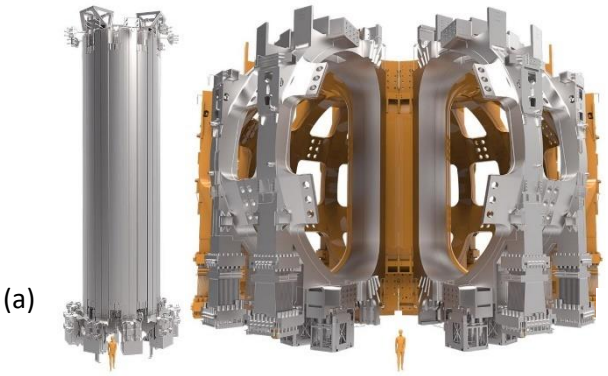


Fig. 1.5 – Solenoide centrale (a) e Sistema di campo toroidale (b)

3) Il sistema di campo poloidale (poloidal field system, Fig. 1.6b), costituito da 6 bobine a forma di anello è situato all'esterno della struttura del magnete del campo toroidale, gestisce la forma e la stabilità del plasma. La bobina più grande ha un diametro di 24 m e il più pesante ha una massa di 400 tonnellate. Le bobine di campo poloidale sono progettate per produrre un'energia magnetica totale di 4 GJ e un campo magnetico massimo di 6 T.
4) Il sistema magnetico di correzione (correction coils, Fig. 1.6a), è costituito da 18 bobine di correzione superconduttive collocate tra quelle di campo toroidale e quelle di campo poloidale e servono per compensare gli errori di campo causate da deviazioni geometriche dovute alle tolleranze di fabbrica o di assemblaggio durante i lavori. Le bobine saranno attraversate da una corrente più piccola rispetto a quella degli altri sistemi magnetici, circa 10 kA, e quindi anche più leggere.

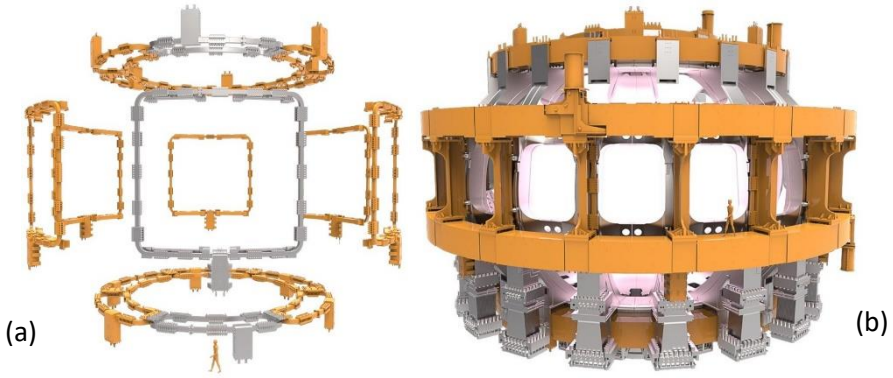


Fig. 1.6 – Bobine di correzione (a) e Sistema di campo poloidale (b).

1.3 IL SISTEMA CRIOGENICO DI ITER

Il sistema criogenico di ITER è adibito al raffreddamento graduale delle criopompe e dei magneti [3, 4, 5] per far fronte a un carico termico depositato nei magneti a causa delle variazioni di campo magnetico e dei neutroni prodotti dalla reazione di fusione tra deuterio e trizio. L'esposizione agli impulsi del plasma può avvenire in diversi scenari: 1) di breve durata, ma ad alta intensità (ordine di poche centinaia di secondi per una potenza di 700 MW), 2) di lunga durata, ma ad intensità ridotta (3000 secondi con una potenza di fusione di 365 MW).

Consiste in due sottosistemi: l'impianto criogenico (composto da refrigeratori ad elio e azoto combinati con loop di elio a 80 K) e di criodistribuzione fornito attraverso un complesso sistema di condotti criogenici (cryolines) che si sviluppano per una lunghezza di 3 km e 50 box di criodistribuzione. Il sistema di raffreddamento dei magneti e delle criopompe è schematizzato nella Fig. 1.7 ed è chiamato ACB (Auxiliary Cold Box) e si interfaccia ai magneti tramite le CTB (Cold Termination Box).

Per i magneti TF vengono utilizzati 9 CTB, 6 per il CS e 11 per i PF e CC, Fig. 1.8.

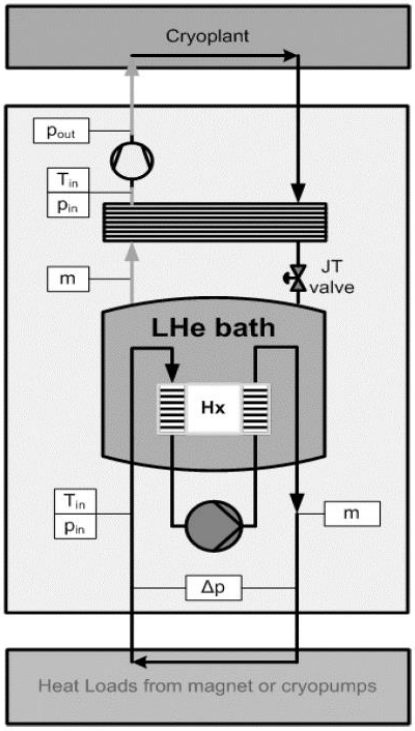


Fig. 1.7 – Schema di raffreddamento dei magneti e delle criopompe

Il circuito criogenico dei magneti di ITER contiene elio supercritico SHe che cede flusso termico attraversando vasche contenenti elio liquido in saturazione. Per smorzare infatti i picchi di potenza durante le operazioni all'interno del tokamak, le vasche di LHe in saturazione agiscono da buffer termici.

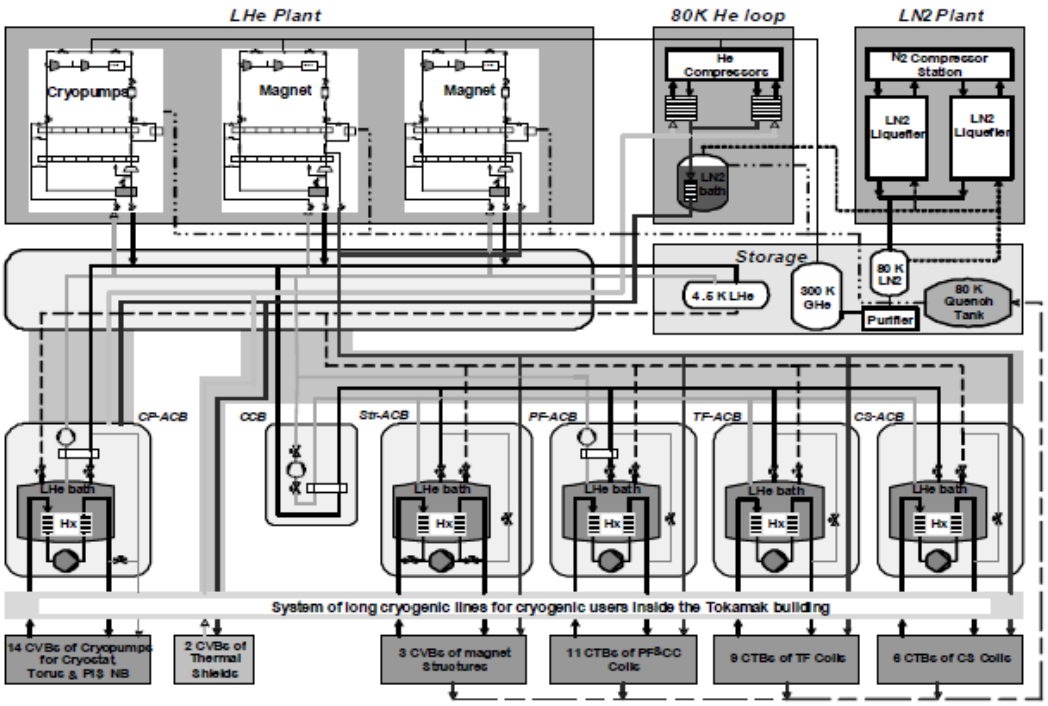


Fig. 1.8 - Architettura del Sistema criogenico di ITER.

1.4 IL CODICE 4C

Il codice Cryogenic Circuit Conductor and Coil (4C) è uno strumento sviluppato dal Dipartimento di Energia del Politecnico di Torino per la simulazione termoidraulica dell'intero sistema di magneti in ITER/DEMO [13]. Con riferimento al raffreddamento degli avvolgimenti di campo toroidale (TF Coils), come rappresentato dallo schema in Fig. 1.9, per studiare il raffreddamento del magnete si possono considerare due circuiti. Il primo è costituito dagli avvolgimenti e dai canali del *case* (1D), il secondo è quello costituito dall'insieme dei piatti radiali e del *case* degli avvolgimenti (2½ D), inoltre, è presente un modello 1D di fluido comprimibile che attraversa le linee criogeniche, scambiatore di calore e componenti a parametri concentrati che coinvolgono bilanci di massa ed energia, come valvole e pompe, realizzato in Dymola sfruttando la libreria ThermoPower [10, 11] e ExternalMedia [12].

Alcune applicazioni del codice 4C possono riguardare:

a) la sicurezza, verificando che il sistema di protezione del magnete possa gestire condizioni di guasto senza danneggiare i magneti, come in [6]
b) la progettazione di sistemi adeguati per la *quench protection*, descritto in [7]
c) le analisi termiche di magneti come JT-60SA TF coil, in [8]
d) lo studio di transitori diversi in riferimento a magneti di campo toroidale, ITER TF coil, come in [9]

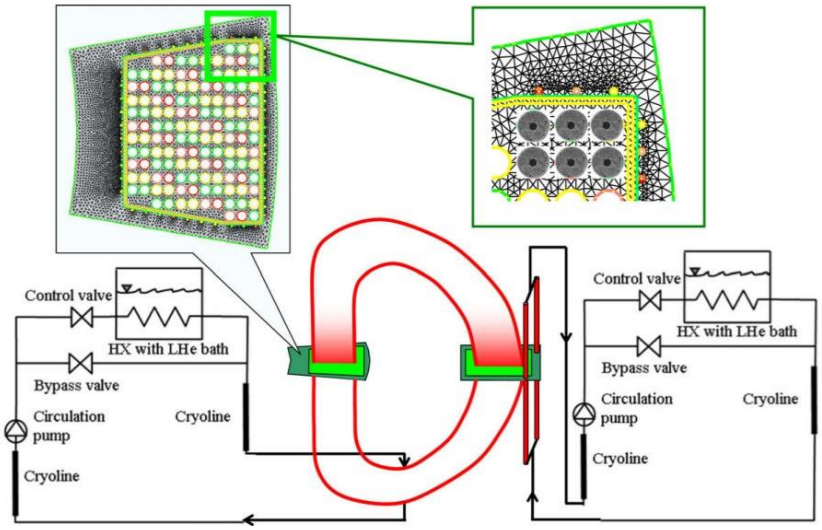


Fig. 1.9 - Schema dei componenti di 4C.

1.5 SCOPO DEL LAVORO

In 4C, strumento allo stato dell'arte, validato per lo studio dell'intero magnete di ITER, la comunicazione tra il blocco di simulazione termo-idraulica/conduzione attraverso le strutture (sviluppato Fortran/Freefem++) e il blocco di simulazione dei circuiti criogenici (sviluppato con Dymola) utilizza TISC, un ambiente di Co-Simulation a pagamento, realizzato da TLK-Thermo GmbH.

TISC, mostrato in Fig. 1.10, permette lo scambio dei dati e la sincronizzazione di simulazioni via standard TCP/IP, consentendo l'esecuzione in parallelo e su più computer connessi alla rete.

Nella presente tesi si mostra una possibile alternativa al TISC, sfruttando l'interoperabilità Fortran/C e interfaccia FMI.

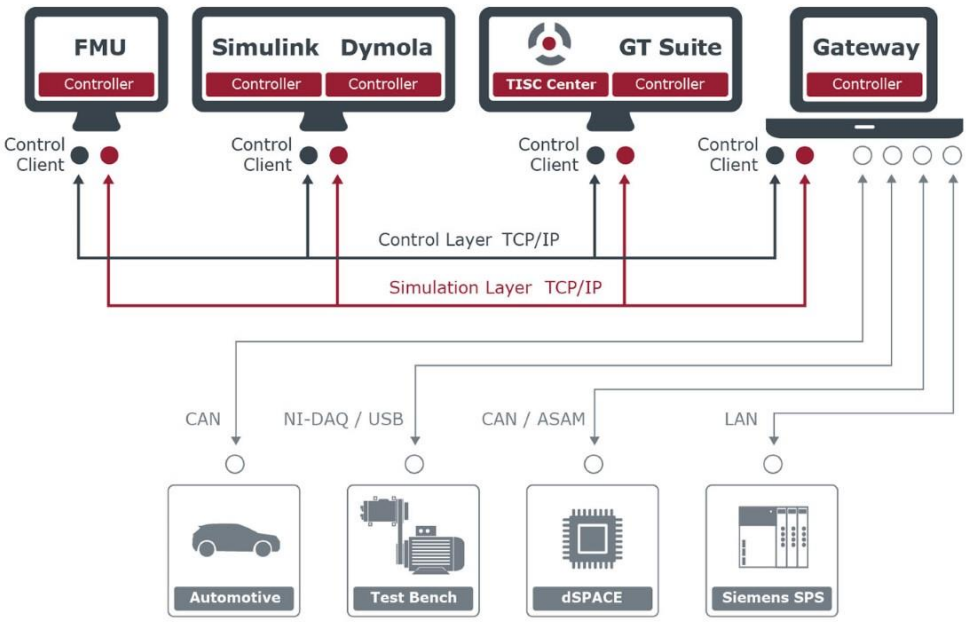


Fig. 1.10 - Comunicazione tramite TISC

Capitolo 2

2 AMBIENTE DI SVILUPPO

2.1 FORTRAN

FORTRAN è un linguaggio ad alto livello progettato da IBM nel 1957 e ampiamente utilizzato nel codice 4C.

FORTRAN è l'anagramma di FORmula TRANslation, le versioni disponibili sono Fortran 90/95, Fortran/2013 e Fortran/2018 ed è pensato per semplificare la scrittura di formule matematiche e fisiche in un linguaggio idoneo per il calcolatore.

Nella presente tesi viene utilizzato Fortran/90.

2.2 MODELICA E MODELICA STANDARD LIBRARY

Modelica è un linguaggio non proprietario, orientato agli oggetti per la simulazione di sistemi complessi dinamici.

Permette di sviluppare dei modelli fisici multi-dominio, meccanici, elettrici, idraulici, termici, controllo, ecc.

2.2.1 Modelica Standard Library

Il package "Modelica" è una libreria free, che può essere utilizzato sia per fini commerciali e non, il cui sviluppo è organizzato dalla Modelica Association Project – Libraries (MAP-LIB, Fig. 2.1).

Fornisce l'insieme di classi, tipi, connettori, partial model di componenti appartenenti a diverse aree applicative la cui divisione è realizzata tramite i Package.

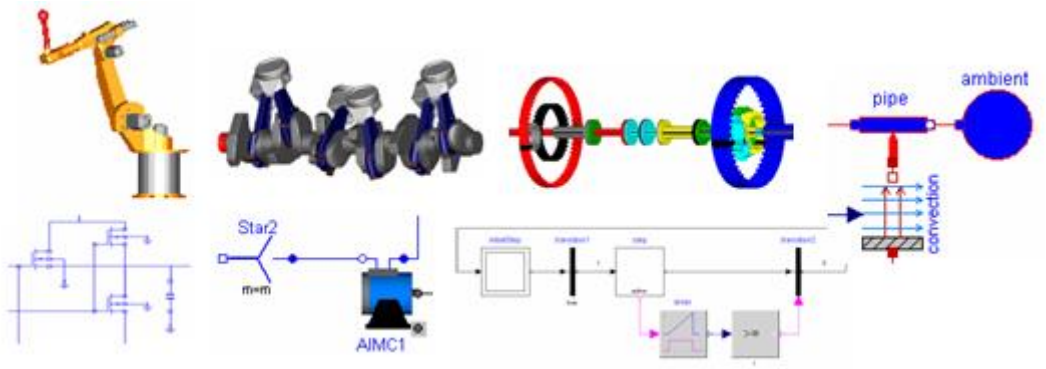


Fig. 2.1 – Modelica Standard Library, interfaccia grafica

2.2.2 ThermoPower

ThermoPower è una libreria Modelica open-source per la modellazione di sistemi dinamici con particolare focus sui sistemi energetici, centrali termiche e, più in generale, sistemi di conversione energetica [14].

Le equazioni del modello sono tipicamente basate su bilanci di massa energia e momento oppure correlazioni empiriche note.

Vi è un largo utilizzo dell'ereditarietà per la definizione dei vari componenti all'interno della libreria, pertanto, il comportamento di un modello non è specificato in una singola classe, ma suddivisa tra più componenti "base" in modo tale che possano essere riutilizzati e quindi estesi.

La connessione tra più componenti avviene tramite i connettori che devono essere sviluppati indipendentemente dai dettagli del modello, in particolare il componente della flangia (flange) non distingue tra componenti monofase, bifase oppure tra parametri concentrati o distribuiti.

È possibile, inoltre, specificare dei flag che escludono o includono porzioni di equazioni da utilizzare, per fare ciò è sufficiente mantenere la consistenza tra numero di equazioni e numero di variabili.

Importante è che i modelli siano indipendenti dalle equazioni costitutive del fluido utilizzato; infatti, tramite keyword definite da Modelica è possibile far riferimento alle caratteristiche del fluido all'interno di un modello in modalità astratta per poi assegnare in fase di simulazione un'implementazione concreta di tali proprietà. Ciò è concesso utilizzando le keyword inner e outer.

Una panoramica dei pacchetti presenti in ThermoPower è rappresentata nell'elenco qui di seguito:

- ThermoPower
    - System
    - Electrical
    - Icons
    - Functions
    - Examples
    - PowerPlants
    - Gas
    - Water
    - Thermal
    - Choices
    - Media
        - GenericGas
        - Air
        - NaturalGas
        - FlueGas
        - CombustionGas
    - Units
    - Test

2.3 FUNCTIONAL MOCK-UP INTERFACE

La Functional Mock-up Interface (FMI) è uno standard, realizzato da MPA (Modelica Association Project) le cui interfacce consentono:

1) lo scambio di dati tra più modelli dinamici, realizzati anche da software differenti;
2) l'incapsulamento dei modelli e dei loro motori di simulazione;
3) il controllo diretto della simulazione.

L'eseguibile che implementa tali interfacce, prende il nome di FMU (Functional Mock-up Unit) e consiste in un file ZIP, che contiene file xml (utilizzati per i metadata), file binari e risorse necessarie alla simulazione [15].

La FMI standard nasce dall'esigenza di poter utilizzare modelli multi-dominio che necessitano della definizione di un'interfaccia comune per poter effettuare l'accoppiamento. In questo modo si può evitare di dover scrivere del codice per poter adattare un modello rispetto ad un altro.

Può essere di "Co-Simulation" nel caso sia provvisto di solver oppure è detto di "Model Exchange" nel caso in cui è richiesto un ambiente di simulazione per effettuare l'integrazione numerica.

2.3.1 Distribuzione FMU

Ogni FMU è distribuito utilizzando un file zip, che contiene i file mostrati nella Fig. 2.2.

```
// Structure of zip file of an FMU
modelDescription.xml          // Description of FMU (required file)
model.png                     // Optional image file of FMU icon
documentation                 // Optional directory containing the FMU documentation
   index.html                 // Entry point of the documentation
   <other documentation files>
   licenses                     // Optional directory for licenses
        license.{txt,html}       // Entry point for license information
        <license files>          // For example BSD licenses
sources                       // Optional directory containing all C sources
   // all needed C sources and C header files to compile and link the FMU
   // with exception of: fmi2TypesPlatform.h , fmi2FunctionTypes.h and fmi2Functions.h
   // The files to be compiled (but not the files included from these files)
   // have to be reported in the xml-file under the structure
   // <ModelExchange><SourceFiles> … and <CoSimulation><SourceFiles>
binaries                      // Optional directory containing the binaries
   win32                      // Optional binaries for 32-bit Windows
      <modelIdentifier>.dll   // DLL of the FMI implementation
                              // (build with option "MT" to include run-time environment)
      <other DLLs>            // The DLL can include other DLLs
      // Optional object Libraries for a particular compiler
      VisualStudio8         // Binaries for 32-bit Windows generated with
                            // Microsoft Visual Studio 8 (2005)
         <modelIdentifier>.lib   // Binary libraries
      gcc3.1                // Binaries for gcc 3.1.
        ...
   win64    // Optional binaries for 64-bit Windows
      ...
   linux32  // Optional binaries for 32-bit Linux
      <modelIdentifier>.so  // Shared library of the FMI implementation
      ...
   linux64  // Optional binaries for 64-bit Linux
      ...
   < If an FMU is run through one of its binaries all items in that binary folder are
     recommended to be unpacked at the same location as the binary < modelIdentifier >.*
     is unpacked. If not it is likely that, if the FMU has dependencies on those items,
     it will not be able to find them. >
resources  // Optional resources needed by the FMU
   < data in FMU specific files which will be read during initialization;
     also more folders can be added under resources (tool/model specific).
     In order for the FMU to access these resource files, the resource directory
     must be available in unzipped form and the absolute path to this directory
     must be reported via argument "fmuResourceLocation" via fmi2Instantiate. >
```

Fig. 2.2 – Descrizione del pacchetto di rilascio FMU

Come si può vedere, la cartella “binaries” contiene i file binari in genere forniti come DLL/SharedObject. Sono suddivisi in cartelle, una per ogni combinazione di piattaforma di destinazione e di architettura.

Un’ulteriore suddivisione all’interno di ogni folder per macchina si può avere nel caso di combinato utilizzo di Model Exchange e di Co-Simulation come in Fig. 2.3.

```
[Example for different libraries:
  binaries
     win32
        MyModel_ModelExchange.dll    // ModelExchange.modelIdentifier =
                                     //    "MyModel_ModelExchange"
        MyModel_CoSimulation.dll     // CoSimulation.modelIdentifier =
                                     //    "MyModel_CoSimulation"
]
```

Fig. 2.3 – Tipologie di esportazione

2.3.2 Co-Simulation

L'esportazione della FMU può avvenire in due modalità: la prima chiamata Model Exchange, la seconda Co-Simulation.

La distinzione tra le due dipende dalla collocazione del risolutore (solver), infatti, nel primo caso è contenuto all'interno dello strumento di modellazione, mentre nel secondo è contenuta all'interno della FMU.

La FMI per la Co-Simulation è pensata per essere utilizzata per accoppiare differenti modelli sottosistema. Ciò è concesso sia per far girare i simulatori assieme ai suoi solver integrati o per agganciare la simulazione tramite tool, come mostrato nelle seguenti figure:

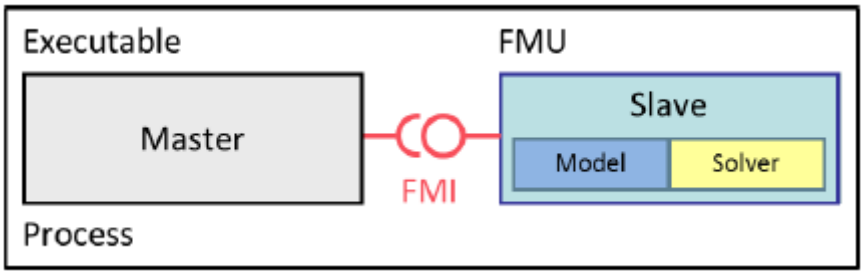


Fig. 2.4 -Co-Simulation con codice generato su un singolo computer (singolo processo)

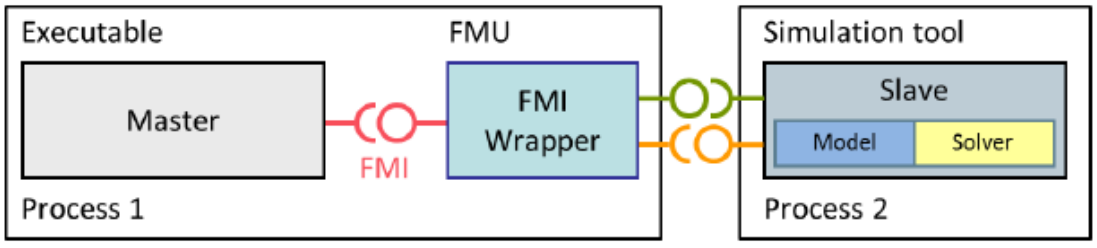


Fig. 2.5 - Co-Simulation con accoppiamento di un simulatore su un computer (multi processo)

Nella presente tesi verrà utilizzata la configurazione rappresentata in Fig. 2.4 in cui l'FMI Wrapper (classe che contiene le API per le chiamate FMI), è sviluppata in Fortran. Mentre viene utilizzata la libreria Python chiamata "PyFMI" per richiamare API della FMU. Infine, per consentire l'interoperabilità tra il Python e il C, viene utilizzata la libreria CFFI.

In particolare, PyFMI è un pacchetto per caricare e interagire con le FMU, consente di utilizzare FMU indipendentemente dalla piattaforma di simulazione come ad es. Dymola, JModelica.org, OpenModelica and SimulationX.

I requisiti per poterla utilizzare sono dettagliati nell'elenco seguente:

Requisiti:

- FMI Library (at least 2.0.1)
- Numpy (recommended 1.6.2)
- Scipy (recommended 0.10.1)
- Ixml (at least 2.3)
- Assimulo (at least 3.0)
- Cython (at least 0.18)
- Python-headers (usually included on Windows, python-dev on Ubuntu)

2.3.3 Descrizione della State Machine

Le chiamate definite dallo standard per effettuare la simulazione sono rappresentate nella Fig. 2.6.

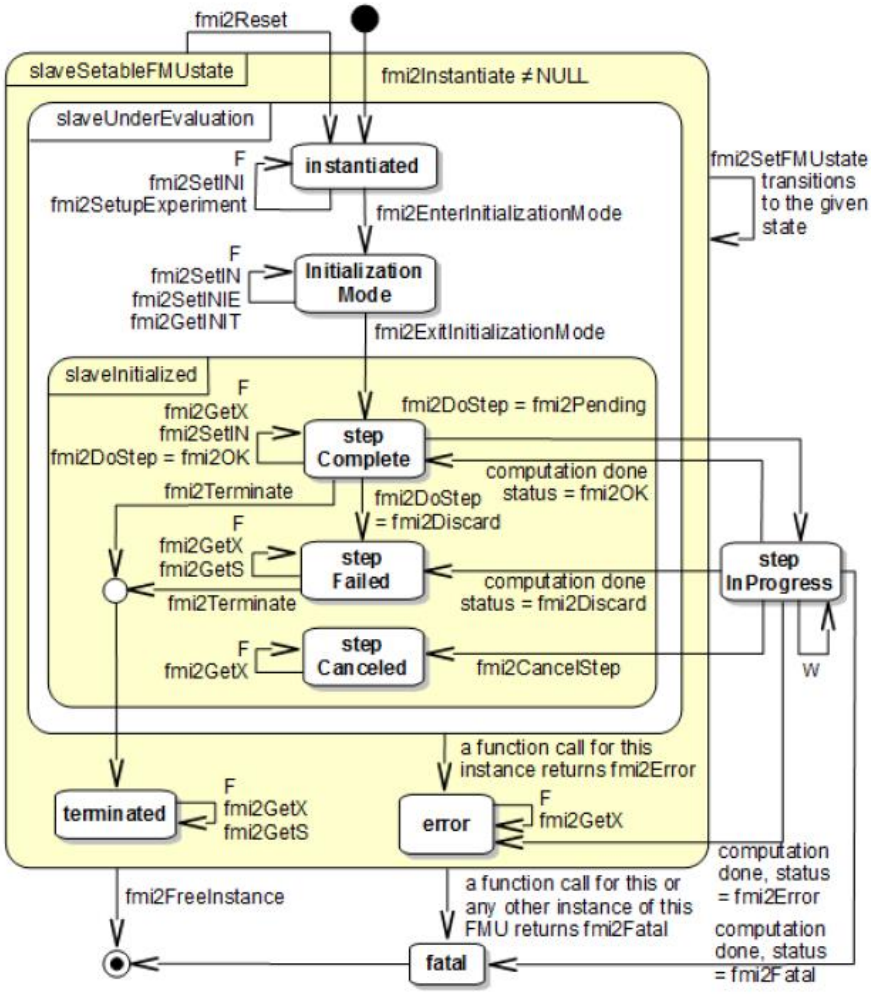


Fig. 2.6 – Descrizione della State Machine

Il metodo fmiInstantiate ritorna una nuova istanza della FMU (in assenza di eccezioni). Questa fase è necessaria prima di poter richiamare tutti gli altri metodi.

Successivamente deve essere chiamato subito il metodo fmiSetupExperiment, che ha come argomenti un valore booleano “toleranceDefined” e due numeri reali “tolerance” e “startTime”. Se la tipologia selezionata è quella di Co-Simulation e toleranceDefined uguale a true, l’intervallo di comunicazione è controllato dalla stima dell’errore. Tale flag potrebbe comunque essere ignorato dalla implementazione della FMU.

Successivamente bisogna richiamare i metodi fmiEnterInitializationMode e fmiExitInitializationMode che definiscono il blocco in cui poter settare tutte quelle variabili scalari che hanno come attributo initial uguale “exact” o “approx”.

Una volta finita la fase di setup si entra nel loop in cui vengono dapprima definiti gli input del modello con i metodi di fmuSetXXX. Poi viene richiamato il metodo fmiDoStep in cui vengono passati il tempo e l’intervallo di tempo di discretizzazione. Qui avviene l’integrazione. Lo “slave” deve integrare tra gli istanti di tempo tc_i e tc_{i+1}.

Una volta chiamata la subroutine fmiDoStep è possibile richiamare i metodi di get del tipo fmuGetXXX e ottenere i valori richiesti come output.

Infine, conclusa la simulazione per deallocare le risorse sarà sufficiente richiamare il metodo fmiTerminate.

È bene precisare che è possibile che vi sono differenze tra le funzioni definite nello standard e quelle realmente implementate nelle librerie contenute nella FMU.

Per esempio, OpenModelica non consente di settare lo stato “globale” in riferimento ad un determinato istante di tempo. Questo si può verificare controllando con il flag “canGetAndSetFMUState” all’interno del file modelDescription.xml.

In caso di crash fatale non sarà possibile, pertanto, partire da una situazione calcolata in precedenza, ma bisognerà invece far ripartire la simulazione dall’inizio; in quanto il fmiDoStep non è una funzione pura poiché è presente uno stato interno rappresentato da tutte quelle variabili che non fanno parte né degli input né degli output.

Le variabili interne vengono chiamate locali e una rappresentazione di tutti i parametri che gestisce la FMU sono raffigurati nella figura sottostante, Fig. 2.7.

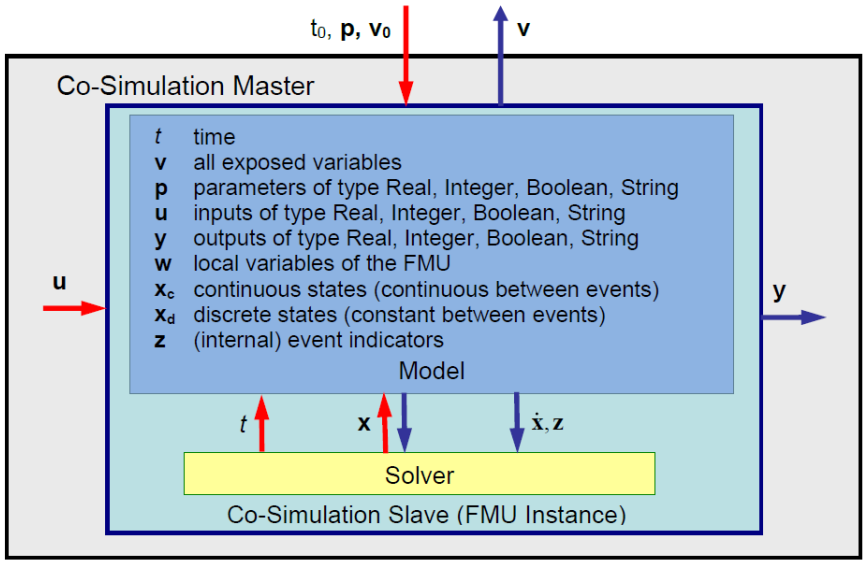


Fig. 2.7 – Punti di Comunicazione dei flussi di dati

2.4 CFFI - C Foreign Function Interface for Python

Per poter usufruire delle funzioni esterne al Python è possibile utilizzare la libreria “ctypes”, questo può essere un ostacolo poiché è necessario specificare tutte le operazioni e le definizioni dei tipi degli argomenti di ogni metodo. Per ovviare a ciò si può utilizzare la libreria CFFI che semplifica questo lavoro [16].

È sufficiente, infatti, specificare in un file Header le definizioni di funzioni e strutture. Sarà poi la libreria che al suo interno, utilizzando un parser, importerà il modulo e si assicurerà i tipi per le funzioni siano corretti.

Per effettuare l’embed di codice esterno all’interno del codice Fortran è sufficiente specificare nella funzione “embedding_init_code” il codice scritto in Python. In questa porzione di codice sarà poi possibile utilizzare dei metodi che CFFI mette a disposizione. La lista completa è rappresentata dalla figura seguente, Fig. 2.8.

Fig. 2.8 – CFFI metodi

In particolare il metodo ffi.buffer restituisce un oggetto buffer che fa riferimento ai dati in C. Un oggetto buffer rappresenta della memoria che può essere passata a varie funzioni integrate o di estensione che leggono e scrivono direttamente in memoria senza necessità di copie aggiuntive che rallenterebbero i tempi di esecuzione.

Capitolo 3

3 APPLICAZIONE

3.1 DESCRIZIONE MODELLO SEMPLIFICATO DEL SISTEMA DI RAFFREDDAMENTO

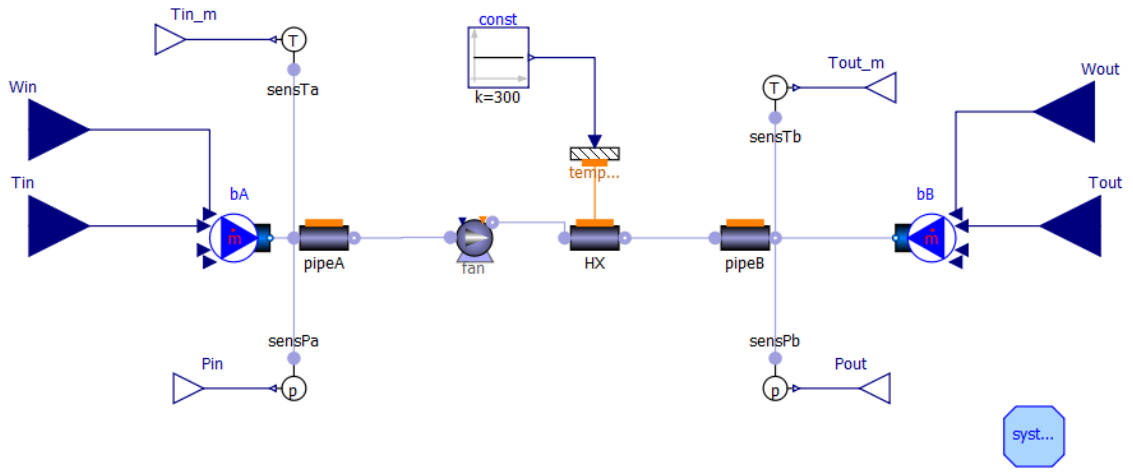


Fig. 3.1 -Schema del modello realizzato in OpenModelica

Il circuito utilizzato per produrre la FMU e testare l'interfaccia è stato realizzato in OpenModelica, il cui schema è rappresentato dalla Fig. 3.1. Rappresenta una semplificazione del circuito di raffreddamento costruito con i componenti ThermoPower in riferimento a [f].

Sono stati utilizzati sia componenti appartenenti alla libreria ThermoPower che dalla Modelica Standard Library. Per fissare le condizioni al contorno sono stati impiegati dei MassFlowSource che prendono in input le portate e le pressioni.

Per modellizzare le linee criogeniche (cryolines) sono stati utilizzati due Flow1DFV, mentre è stata sostituita la pompa con un ventilatore in quanto la temperatura dell'elio, fissata dal termostato (TempSource1DFV), è di 300 K.

Il codice del modello, scritto in OpenModelica, è il seguente:

```
import ThermoPower.*;
import ThermoPower.Gas.*;
import ThermoPower.Thermal.*;
import Modelica.Blocks.Sources.*;
```

```
  import Modelica.Fluid.Sources.*;
  import Modelica.SIunits.*;
  package HeMedium = Modelica.Media.IdealGases.SingleGases.He;

  parameter Real pMassFlowRate = 2e-3;
  parameter Integer pPipeNodes = 5;
  parameter Length pPipeLength = 1;
  parameter Diameter pPipeInternalDiameter = 0.05;
  parameter Volume pFanVolume = 0.1;
  parameter Real pCfNom = 0.005;
  parameter Density pDensityNom = 0.4807; // (3 bar, 300 K)
  parameter Enthalpy pEnthalpyNom = 1.565e6; // h(300 K)
  parameter AbsolutePressure pPressureNom = 3e5;
  parameter Real pCompressionRatio = 2.0;
  parameter Temp_K pBathTemperature = 300;

  parameter Radius pPipeInternalRadius = pPipeInternalDiameter / 2;
  parameter Length pPipeOmega = Modelica.Constants.pi *
pPipeInternalDiameter;
  parameter Area pPipeArea = Modelica.Constants.pi *
pPipeInternalRadius^2;

  Flow1DFV pipeA(
    redeclare package Medium = HeMedium,
    A = pPipeArea,
    Dhyd = pPipeInternalDiameter,
    DynamicMomentum = true,
    FFtype = ThermoPower.Choices.Flow1D.FFtypes.NoFriction,
    L = pPipeLength,
    N = pPipeNodes,
    dpnom(displayUnit = "Pa") = 10,
    omega = pPipeOmega,
    rhonom = HeMedium.density_pT(pPressureNom, pBathTemperature),
    wnom = pMassFlowRate,
pstart = pPressureNom
    )
```

```
  annotation (
    Placement(visible = true, transformation(origin = {-70, 0}, extent
= {{-10, -10}, {10, 10}}, rotation = 0)));

  ThermoPower.Gas.Fan fan(
    redeclare package Medium = HeMedium,
    redeclare function flowCharacteristic = testFan,
    dp0(displayUnit = "bar") = pPressureNom * (pCompressionRatio - 1),
    hstart = pEnthalpyNom,
    n0 = 1500,
    rho0 = pDensityNom,
    w0 = pMassFlowRate)
  annotation (
    Placement(visible = true, transformation(origin = {-20, -2},
extent = {{-10, -10}, {10, 10}}, rotation = 0)));

  Flow1DFV HX(
    redeclare package Medium = HeMedium,
    A = pPipeArea,
    Cfnom = pCfNom,
    Dhyd = pPipeInternalDiameter,
    FFtype = ThermoPower.Choices.Flow1D.FFtypes.OpPoint,
    L = pPipeLength,
    N = pPipeNodes,
    Nt = 2,
    dpnom(displayUnit = "Pa") = 10,
    omega = pPipeOmega,
    wnom = pMassFlowRate,
    rhonom = HeMedium.density_pT(pPressureNom, pBathTemperature),
    pstart = pPressureNom,
    DynamicMomentum = true,
    initOpt=ThermoPower.Choices.Init.Options.noInit)
   annotation (
    Placement(visible = true, transformation(origin = {20, 0}, extent
= {{-10, -10}, {10, 10}}, rotation = 0)));
```

```
  Flow1DFV pipeB(
    redeclare package Medium = HeMedium,
    A = pPipeArea,
    Cfnom = pCfNom,
    Dhyd = pPipeInternalDiameter,
DynamicMomentum = true,
    FFtype = ThermoPower.Choices.Flow1D.FFtypes.NoFriction,
    L = pPipeLength,
    N = pPipeNodes,
    Tstartbar = pBathTemperature,
    dpnom(displayUnit = "Pa") = 10,
    fixedMassFlowSimplified = true,
    omega = pPipeOmega,
    pstart = pPressureNom,
    rhonom = HeMedium.density_pT(pPressureNom, pBathTemperature),
    initOpt=ThermoPower.Choices.Init.Options.noInit,
    wnom = pMassFlowRate)
     annotation (
    Placement(visible = true, transformation(origin = {70, 0}, extent
= {{-10, -10}, {10, 10}}, rotation = 0)));

  MassFlowSource_T bA(
    redeclare package Medium = HeMedium,
    nPorts = 1,
    use_T_in = true,
    use_m_flow_in = true)
  annotation (
    Placement(visible = true, transformation(origin = {-98, 0}, extent
= {{-10, -10}, {10, 10}}, rotation = 0)));

  MassFlowSource_T bB(
  redeclare package Medium = HeMedium,
  nPorts = 1,
  use_T_in = true,
  use_m_flow_in = true)
  annotation (
```

```
    Placement(visible = true, transformation(origin = {130, 0}, extent
= {{10, -10}, {-10, 10}}, rotation = 0)));

  SensP sensPa(redeclare package Medium = HeMedium)
  annotation (
    Placement(visible = true, transformation(origin = {-80, -44},
extent = {{10, 10}, {-10, -10}}, rotation = 0)));

  SensP sensPb(redeclare package Medium = HeMedium)
  annotation (
    Placement(visible = true, transformation(origin = {80, -44},
extent = {{-10, 10}, {10, -10}}, rotation = 0)));

  SensT1 sensTa(redeclare package Medium = HeMedium)
  annotation (
    Placement(visible = true, transformation(origin = {-80, 44},
extent = {{10, -10}, {-10, 10}}, rotation = 0)));

  SensT1 sensTb(redeclare package Medium = HeMedium)
  annotation (
    Placement(visible = true, transformation(origin = {80, 44}, extent
= {{-10, -10}, {10, 10}}, rotation = 0)));

  TempSource1DFV tempSource1DFV(Nw = pPipeNodes - 1)  annotation (
    Placement(visible = true, transformation(origin = {20, 28}, extent
= {{-10, -10}, {10, 10}}, rotation = 0)));
  Constant const(k = pBathTemperature)  annotation (
    Placement(visible = true, transformation(origin = {-12, 60},
extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  Constant massflowInlet(k = pMassFlowRate) annotation (
    Placement(visible = true, transformation(origin = {-134, 40},
extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  Constant tempInlet(k = 300)    annotation (
    Placement(visible = true, transformation(origin = {-132, -4},
extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  Constant massflowOutlet(k = -pMassFlowRate) annotation (
```

```
    Placement(visible = true, transformation(origin = {124, 66},
extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  Constant tempOutlet(k = 300) annotation (
    Placement(visible = true, transformation(origin = {130, -52},
extent = {{-10, -10}, {10, 10}}, rotation = 0)));
  inner ThermoPower.System system annotation (
    Placement(visible = true, transformation(extent = {{124, -92},
{144, -72}}, rotation = 0)));

  function testFan = Functions.FanCharacteristics.linearFlow (
      q_nom = {0, 4.1606e-3},
      H_nom = {6.366e5, 624.1e3});

equation

  connect(bA.ports[1], pipeA.infl);
  connect(sensTa.flange, pipeA.infl);
  connect(sensPa.flange, pipeA.infl);
  connect(bB.ports[1], pipeB.outfl);
  connect(sensTb.flange, pipeB.outfl);
  connect(sensPb.flange, pipeB.outfl);
  connect(tempSource1DFV.wall, HX.wall);
  connect(const.y, tempSource1DFV.temperature);
  connect(pipeA.outfl, fan.infl);
  connect(massflowInlet.y, bA.m_flow_in);
  connect(tempInlet.y, bA.T_in);
  connect(bB.m_flow_in, massflowOutlet.y);
  connect(bB.T_in, tempOutlet.y);
  connect(HX.outfl, pipeB.infl);
  connect(fan.outfl, HX.infl);

annotation (
    uses(ThermoPower(version = "3.1"), Modelica(version="3.2.2")),
    version = "",
```

```
  __OpenModelica_commandLineOptions = "--matchingAlgorithm=PFPlusExt -
-indexReductionMethod=dynamicStateSelection -
d=initialization,NLSanalyticJacobian -d=bltdump",
  experiment(StartTime = 0, StopTime = 80, Tolerance = 1e-6, Interval
= 0.16));
```

3.2 IMPLEMENTAZIONE DEL CODICE

3.2.1 Sviluppo del connettore FMU

Una volta esportata la FMU tramite OpenModelica, sono stati implementati i metodi in Python in sostituzione delle chiamate al connettore TISC.

Connector FMU Implementation

```
from fmuconnector_plugin import ffi
from pyfmi.fmi import FMUModelCS2
import numpy as np

charEncoding = 'UTF-8'
# charEncoding = 'ascii'


fmu = None
fmu_time = 0.0
fmu_timeStep = 1.0
fmu_step = 0
fmu_variableReferenceInputDict = {}
fmu_variableReferenceOutputDict = {}


@ffi.def_extern()
def fmuPyInstantiate(fileName):
    global fmu
    pyFileName = ffi.string(fileName).decode(charEncoding)
    if fmu is None:
        print(f'fmuInstantiate: {str(pyFileName)}')
        fmu = FMUModelCS2(str(pyFileName))
    fmu.instantiate()
    fmu.initialize()
    # fmu.setup_experiment(start_time=fmu_time)
    fmu.enter_initialization_mode()
    fmu.exit_initialization_mode()
    return None


@ffi.def_extern()
def fmuPySetSyncRate(time_ms):
```

```
    global fmu_timeStep
    fmu_timeStep = time_ms[0] * 1e-3
    print(f'fmu_timeStep: {str(fmu_timeStep)} s')
    return None


@ffi.def_extern()
def fmuPyDefineInput(variableName):
    pyVariableName = ffi.string(variableName).decode(charEncoding)
    print(f'fmuDefineInput - VariableName: {pyVariableName}')
    pyVariableName = str(pyVariableName)
    variableReference = fmu.get_variable_valueref(pyVariableName)
    fmu_variableReferenceInputDict[pyVariableName] = variableReference
    return None


@ffi.def_extern()
def fmuPyDefineOutput(variableName):
    pyVariableName = ffi.string(variableName).decode(charEncoding)
    pyVariableName = str(pyVariableName)
    print(f'fmuDefineOutput - VariableName: {pyVariableName}')
    if pyVariableName == "timeFmu":
        return
    variableReference = fmu.get_variable_valueref(pyVariableName)
    fmu_variableReferenceOutputDict[pyVariableName] =
variableReference
    return None


@ffi.def_extern()
def fmuPySetInputDouble(variableName, value):
    pyVariableName = ffi.string(variableName).decode(charEncoding)
    pyVariableName = str(pyVariableName)
    print(f'fmuSetInputDouble - VariableName {pyVariableName}, Value
{value[0]}')
    if pyVariableName == "timeFmu":  # feature
        return
    variableReference = fmu_variableReferenceInputDict[pyVariableName]
    fmu.set_real([variableReference], [value[0]])
    return None


@ffi.def_extern()
def fmuPyExecuteStep():
    global fmu_step, fmu_time
    print(f'fmuExecuteStep - Time: {fmu_time}')
    fmu.do_step(fmu_time, fmu_timeStep)

    fmu_step += 1
    fmu_time += fmu_timeStep
    return None
```

```
@ffi.def_extern()
def fmuPyGetOutputDouble(variableName):
    pyVariableName = ffi.string(variableName).decode(charEncoding)
    variableReference =
fmu_variableReferenceOutputDict[pyVariableName]

    tmp = fmu.get_real([variableReference])
    print(f'fmuGetOutputDouble - VariableName {pyVariableName} - Value
{str(tmp)}')

    return tmp


@ffi.def_extern()
def fmuPyGetOutputDoubleMatrix(variableName, rows, columns,
outputMatrix):
    pyVariableName = ffi.string(variableName).decode(charEncoding)
    print(f'fmuPyGetOutputDoubleMatrix - VariableName
{pyVariableName}')
    np.frombuffer(
        ffi.buffer(outputMatrix, np.prod(rows, columns) *
ffi.sizeof('double')),
        np.dtype('f8')
    ).reshape((rows, columns))
```

Con il tool CFFI è stato sviluppato uno script per produrre il binding tra i metodi in Fortran e quelli in Python. Per effettuare questo passaggio è stato necessario scrivere anche gli header, in cui sono stati definiti i prototipi dei metodi richiamati dal Fortran, specificando il nome della subroutine o function, i tipi degli argomenti in input e il tipo del valore di output.

Connector Header

```
# ifndef CFFI_DLLEXPORT
# if defined(_MSC_VER)
# define CFFI_DLLEXPORT extern __declspec(dllimport)
# else
# define CFFI_DLLEXPORT extern
# endif
#endif

CFFI_DLLEXPORT void fmuPyInstantiate(char *);

CFFI_DLLEXPORT void fmuPySetSyncRate(int *);

CFFI_DLLEXPORT void fmuPyDefineInput(char *);

CFFI_DLLEXPORT void fmuPyDefineOutput(char *);
```

```
CFFI_DLLEXPORT void fmuPySetInputDouble(char *, double *);

CFFI_DLLEXPORT void fmuPySetInputDoubleMatrix(char *, int *, int *,
double *);

CFFI_DLLEXPORT void fmuPyExecuteStep();

CFFI_DLLEXPORT double fmuPyGetOutputDouble(char *);

CFFI_DLLEXPORT void fmuPyGetOutputDoubleMatrix(char *, int *, int *,
double *);
```

Client

```
module ClientFMU
contains
    subroutine InitializeFmu(syncRateMSec)
        use ConnectorFMU, only : fmuInstantiate, fmuSetSyncRate,
fmuDefineInput, fmuDefineOutput
        implicit none

        ! PARAMETERS
        integer, intent(in) :: syncRateMSec

        ! Instantiate FMU Connector
        call fmuInstantiate('Nome della FMU')

        ! Define TimeStep (ms)
        call fmuSetSyncRate(syncRateMSec)

        ! Define INPUTS
        call fmuDefineInput('Win')
        call fmuDefineInput('Wout')
        call fmuDefineInput('Tin')
        call fmuDefineInput('Tout')

        ! Define OUTPUTS
        call fmuDefineOutput('timeFmu')
        call fmuDefineOutput('Pin')
```

```
        call fmuDefineOutput('Pout')
        call fmuDefineOutput('Tin_m')
        call fmuDefineOutput('Tout_m')
    end subroutine InitializeFmu

    subroutine InputFromFmu(timeFmu, Tin, Tout, Pin, Pout)
        use ConnectorFMU, only : fmuExecuteStep, fmuGetOutputDouble
        implicit none

        ! PARAMETERS
        real(8), intent(out) :: timeFmu
        real(8), intent(out) :: Tin, Tout
        real(8), intent(out) :: Pout, Pin

        call fmuExecuteStep()

        call fmuGetOutputDouble('timeFmu', timeFmu)
        call fmuGetOutputDouble('Pin', Pin)
        call fmuGetOutputDouble('Pout', Pout)
        call fmuGetOutputDouble('Tin_m', Tin)
        call fmuGetOutputDouble('Tout_m', Tout)
    end subroutine InputFromFmu

    subroutine OutputToFmu(ToutAve, TinAve, dmdtOut, dmdtIn,
syncRateMSec)
        use ConnectorFMU, only : fmuSetSyncRate, fmuSetInputDouble
        implicit none

        ! PARAMETERS
        real(8), intent(in) :: ToutAve, TinAve
        real(8), intent(in) :: dmdtOut, dmdtIn
        integer, intent(in) :: syncRateMSec;

        call fmuSetSyncRate(syncRateMSec)
        call fmuSetInputDouble('Win', dmdtIn)
        call fmuSetInputDouble('Wout', dmdtOut)
```

```
        call fmuSetInputDouble('Tin', TinAve)
        call fmuSetInputDouble('Tout', ToutAve)
    end subroutine OutputToFmu
end module ClientFMU
```

Connector FMU Fortran

```
module ConnectorFMU
    use, intrinsic :: iso_c_binding, only : c_double, c_int, c_char
    implicit none

    interface
        subroutine fmuPyInstantiate(fmuFileName) bind (c, name =
"fmuPyInstantiate")
            use iso_c_binding, only : c_char

            character(c_char) :: fmuFileName
        end subroutine fmuPyInstantiate

        subroutine fmuPySetSyncRate(syncRateMSec) bind (c, name =
"fmuPySetSyncRate")
            use iso_c_binding, only : c_int

            integer(c_int) :: syncRateMSec
        end subroutine fmuPySetSyncRate

        subroutine fmuPyDefineInput(variableName) bind (c, name =
"fmuPyDefineInput")
            use iso_c_binding, only : c_char
            character(c_char) :: variableName
        end subroutine fmuPyDefineInput

        subroutine fmuPyDefineOutput(variableName) bind (c, name =
"fmuPyDefineOutput")
            use iso_c_binding, only : c_char
            character(c_char) :: variableName
```

```
        end subroutine fmuPyDefineOutput

        subroutine fmuPySetInputDouble(variableName, variable) bind
(c, name = "fmuPySetInputDouble")
            use iso_c_binding, only : c_char, c_double
            character(c_char) :: variableName
            real(c_double) :: variable
        end subroutine fmuPySetInputDouble

        subroutine fmuPySetInputDoubleMatrix(variableName, rows,
columns, variable) bind (c, name = "fmuPySetInputDoubleMatrix")
            use iso_c_binding, only : c_double, c_int, c_char
            character(c_char) :: variableName
            integer(c_int) :: rows, columns
            real(c_double) :: variable(rows, columns)
        end subroutine fmuPySetInputDoubleMatrix

        subroutine fmuPyExecuteStep() bind(C, name =
"fmuPyExecuteStep")
        end subroutine fmuPyExecuteStep

        function fmuPyGetOutputDouble(variableName) result(output)
bind (c, name = "fmuPyGetOutputDouble")
            use iso_c_binding, only : c_double, c_char
            character(c_char) :: variableName
            real(c_double) :: output
        end function fmuPyGetOutputDouble

        subroutine fmuPyGetOutputDoubleMatrix(variableName, rows,
columns, output) bind(c, name = "fmuPyGetOutputDoubleMatrix")
            use iso_c_binding, only : c_double, c_char, c_int

            character(c_char), value:: variableName
            integer(c_int), value :: rows, columns
            real(c_double) :: output(rows, columns)
        end subroutine fmuPyGetOutputDoubleMatrix
```

```
    end interface

contains
    subroutine fmuInstantiate(fmuFileName)
        character(len=*) fmuFileName
        character(len=256) pyFmuFileName

        pyFmuFileName = trim(fmuFileName) // char(0)
        call fmuPyInstantiate(pyFmuFileName)
    end subroutine fmuInstantiate

    subroutine fmuSetSyncRate(syncRateMSec)
        integer(c_int) :: syncRateMSec

        call fmuPySetSyncRate(syncRateMSec)
    end subroutine fmuSetSyncRate

    subroutine fmuDefineInput(variableName)
        character(len=*) :: variableName
        character(len=256) :: pyVariableName

        pyVariableName = trim(variableName) // char(0)
        call fmuPyDefineInput(pyVariableName)
    end subroutine fmuDefineInput

    subroutine fmuDefineOutput(variableName)
        character(len=*) :: variableName
        character(len=256) :: pyVariableName

        pyVariableName = trim(variableName) // char(0)
        call fmuPyDefineOutput(pyVariableName)
    end subroutine fmuDefineOutput

    subroutine fmuSetInputDouble(variableName, variable)
        character(len=*) :: variableName
        real(8) :: variable
```

```
        character(len=256) :: pyVariableName
        real(c_double) :: pyVariable

        pyVariableName = trim(variableName) // char(0)
        pyVariable = variable
        call fmuPySetInputDouble(pyVariableName, pyVariable)
    end subroutine fmuSetInputDouble

    subroutine fmuSetInputDoubleMatrix(variableName, variable, rows,
columns)
        character(len=*) :: variableName
        real(8) :: variable
        integer(c_int) :: rows, columns

        character(256) :: pyVariableName
        real(c_double) :: pyVariable(rows, columns)

        pyVariableName = variableName
        pyVariable = variable
        call fmuPySetInputDoubleMatrix(pyVariableName, rows, columns,
pyVariable)
    end subroutine fmuSetInputDoubleMatrix

    subroutine fmuExecuteStep()
        call fmuPyExecuteStep()
    end subroutine fmuExecuteStep

    subroutine fmuGetOutputDouble(variableName, output)
        character(len=*) :: variableName
        real(8) :: output

        character(len=256) :: pyVariableName
        real(c_double) :: pyOutput

        pyVariableName = trim(variableName) // char(0)
```

```
        pyOutput = fmuPyGetOutputDouble(pyVariableName)
        output = pyOutput
    end subroutine fmuGetOutputDouble

    subroutine fmuGetOutputDoubleMatrix(variableName, rows, columns,
output)
        character(len=*) :: variableName
        integer(c_int) :: rows, columns
        real(8) :: output(rows, columns)

        character(len=256) :: pyVariableName
        real(c_double) :: pyOutput(rows, columns)

        pyVariableName = trim(variableName) // char(0)
        call fmuPyGetOutputDoubleMatrix(pyVariableName, rows, columns,
pyOutput)
        output = pyOutput
    end subroutine fmuGetOutputDoubleMatrix

    function fmuIsActive() result(output)
        logical :: output
        output = .true.
    end function fmuIsActive
end module ConnectorFMU
```

Plugin builder

```
import cffi
ffiBuilder = cffi.FFI()
with open('connector.h') as f:
    data = ''.join([line for line in f if not line.startswith('#')])
    data = data.replace('CFFI_DLLEXPORT', '')
    ffiBuilder.embedding_api(data)

ffiBuilder.set_source("fmuconnector_plugin", r'''
#include "connector.h"
```

```
''')

with open('ConnectorFMUImplementation.py') as f:
    ffiBuilder.embedding_init_code(f.read())

ffiBuilder.compile(target="fmuconnector2.*", verbose=True)
```

Main

```
program Main
    use ClientFMU, only : InitializeFmu, OutputToFmu, InputFromFmu
    implicit none

    integer :: syncRateMSec

    do i = 1, N
        call InitializeFmu(syncRateMSec)

        call OutputToFmu(ToutAve, TinAve, dmdtOut, dmdtIn,
syncRateMSec)

        call InputFromFmu(timeFmu, Tin, Tout, Pin, Pout)
    end do

end program Main
```

Capitolo 4

4 CONCLUSIONI E PROSPETTIVE

È stata realizzata la comunicazione tra la FMU contenente un modello semplificato del circuito criogenico, mantenendo gli stessi input e output presenti in 4C, in forma scalare, realizzato in OpenModelica.

È stato implementato il connettore FMU in Python utilizzando la libreria PyFMI, conforme allo standard FMI 2.0, prendendo come riferimento le firme dei metodi richiamati in 4C.

Con la libreria CFFI, è stata eseguita una transcodifica Python/C per sfruttare l'interoperabilità Fortran/C, permettendo di richiamare funzionalità di Python dal codice Fortran presente in 4C. Tale procedura può essere utilizzata in forma più generale per consentire di eseguire Python o pacchetti Python dal Fortran.

Uno dei vantaggi è rappresentato dalla tipologia della licenza degli strumenti utilizzati. Infatti, i "wrapper" FMI, che implementano lo standard, risultano OpenSource, così come tutti i componenti utilizzati nella presente tesi, mentre TISC è un software commerciale.

La trasmissione dei dati, inoltre, non avviene tramite protocollo TCP-IP come in TISC, ma tramite passaggio di puntatori, con la prospettiva di velocizzare il processo di trasferimento tra i diversi componenti di 4C.

5 IMMAGINI

[Fig. 1.1] https://www.iter.org/doc/all/content/com/gallery/media/7%20-%20technical/in-cryostat%20overview%20130116.jpg

[Fig. 1.2] https://www.iter.org/mach/blanket

[Fig. 1.3] https://www.iter.org/mach/magnets

[Fig. 1.4] https://www.iter.org/mach/magnets

[Fig. 1.5] https://www.iter.org/mach/magnets

[Fig. 1.6] https://www.iter.org/mach/magnets

[Fig. 1.7] L. Serio, ITER Organization Domestic Agencies and Collaborators, "Challenges for Cryogenics at ITER", https://doi.org/10.1063/1.3422415, pag. 657, 23 aprile 2010

[Fig. 1.8] L. Serio, ITER Organization Domestic Agencies and Collaborators, "Challenges for Cryogenics at ITER", https://doi.org/10.1063/1.3422415, pag. 656, 23 aprile 2010

[Fig. 1.9] L. Savoldi Richard, F. Casella, B. Fiori, R. Zanino, "The 4C code for the cryogenic conductor and coil modeling in ITER", Cryogenics, vol. 50, pp. 167-176, 2010

[Fig. 1.10] https://tlk-thermo.com/index.php/en/tisc-suite

[Fig. 2.1] http://github.com/modelica/ModelicaStandardLibrary

[Fig. 2.2] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation", ver. 2.0.1, pag. 68, 2019

[Fig. 2.3] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation", ver. 2.0.1, pag. 69, 2019

[Fig. 2.4] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation", ver. 2.0.1, pag. 95, 2019

[Fig. 2.5] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation", ver. 2.0.1, pag. 95, 2019

[Fig. 2.6] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation", ver. 2.0.1, pag. 105, 2019

[Fig. 2.7] Modelica Association, “Functional Mock-up Interface for Model Exchange and Co-Simulation”, ver. 2.0.1, pag. 96, 2019

[Fig. 2.8] https://cffireadthedocs.io/en/latest/ref.html

6 RIFERIMENTI

[1] ITER Organization, “It’s now official: first plasma in December 2025”, https://www.iter.org/newsline/-/2482, 20 giugno 2016,

[2] ITER Organization, “What will ITER do?”, https://www.iter.org/sci/Goals

[3] L. Serio, ITER Organization Domestic Agencies and Collaborators, “Challenges for Cryogenics at ITER”, https://doi.org/10.1063/1.3422415, 23 aprile 2010

[4] V. Kalinin, E. Tada, F. Millet, N. Shatil, “ITER Cryogenic system”, Fusion Engineering and Design, vol. 81, pp. 2589-2595, 5 settembre 2006

[5] N. Peng, L.Y. Xiong, Y.C. Jian, J.C. Tang, L.Q. Liu, “Study on mitigation of pulsed heat load for ITER cryogenic system”, Cryogenics, vol. 66, pp. 1-5, 2015

[6] L. Savoldi , Member, IEEE, R. Bonifetto , N. Pedroni, and R. Zanino , Senior Member, IEEE, "Analysis of a Protected Loss of Flow Accident (LOFA) in the ITER TF Coil Cooling Circuit", IEEE Transactions on Applied Superconduttivity, vol. 28, no. 3, aprile 2018

[7] L. Savoldi, R. Bonifetto, A. Brighenti, V. Corato, L. Muzzi, S. Turtu’, R. Zanino & A. Zappatore, "Quench Propagation in a TF Coil of the EU DEMO", Fusion Science and Technology, pp. 1943-7641, 26 giugno 2017

[8] R. Bonifetto, P.K. Domalapally, G.M. Polli, L. Savoldi Richard, S. Turtu', R. Villari, R. Zanino, "Computation of JT-60SA TF coil temperature margin using the 4C code", Fusion Engineering and Design, vol. 86, pp. 1493-1496, 2011

[9] L. Savoldi Richard, F. Casella, B. Fiori, R. Zanino, “The 4C code for the cryogenic conductor and coil modeling in ITER”, Cryogenics, vol. 50, pp. 167-176, 2010

[10] F. Casella and A. Leva, "Modelica open library for power plant simulation:

design and experimental validation," in Proceedings of the 3rd International

Modelica Conference, Linköping, Sweden, 2003.

[11] F. Casella and A. Leva, “Modelling of Thermo-Hydraulic Power Generation

Processes Using Modelica,” Mathematical and Computer Modeling of

Dynamical Systems, vol. 12, no. 1, pp. 19-33, 2006.

[12] F. Casella and C. C. Richter, “ExternalMedia: a Library for Easy Re-Use of

External Fluid Property Code in Modelica,” in Proceedings of the 6th

International Modelica Conference, Bielefeld, Germany, 2008.

[13] R. Bonifetto, F. Buonora, L. Savoldi Richard, Member, IEEE and R. Zanino, member, IEEE, “4C Code Simulation and Benchmark of ITER TF Magnet Cool-Down From 300 K to 80 K”, IEEE Transaction on applied Superconductivity, vol. 22, n. 3, giugno 2012

[14] F. Casella, A. Leva, “Modelling of thermo-hydraulic power generation processes using Modelica”, Mathematical and Computer Modelling of Dynamical Systems, vol. 12, n. 1, pp. 19-33, febbraio 2006

[15] Modelica Association, “Functional Mock-up Interface for Model Exchange and Co-Simulation”, ver. 2.0.1, 2019

[16] A. Rigo, M. Fijalkowski, "CFFI Documentation", Release 1.15.0, Oct 13, 2021

7 RINGRAZIAMENTI

Ringrazio il relatore Prof. Antonio Froio e
co-relatore Prof. Roberto Bonifetto, per la loro disponibilità e pazienza.

Ringrazio la mia famiglia che mi ha sempre sostenuto.