FLORIDA STATE UNIVERSITY

FAMU-FSU COLLEGE OF ENGINEERING

AN ADAPTABLE SIGNAL INTERFACE TO ENABLE A FAULT MANAGEMENT DEVELOPMENT PROCESS FOR A MEDIUM VOLTAGE DC MICROGRID

By

CARLOS A. WONG

A Thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

2020

Carlos A. Wong defended this thesis on November 17, 2020.
The members of the supervisory committee were:

Michael Steurer

Professor Directing Thesis

Sastry Pamidi

Committee Member

Omar Faruque

Committee Member

Karl Schoder

Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

To my parents and brothers who believed in me. To my loving wife, Valeria, who patiently helped me get through this process. To my dear friends and colleagues that have helped get to this point. Thank you.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my advisor and mentor, Dr. Mischa Steurer, for the opportunity to work with and to learn from these last 2 years. His knowledge, vision, and attention to detail have helped structure the way I approach problems. I would like to extend my gratitude to Dr. Mark Stanovich, Dr. Karl Schoder, and Dr. James Langston, whose insight and feedback has aided me throughout the entire thesis. A thank you, to the professors serving on my committee, Dr. Sastry Pamidi and Dr. Omar Faruque.

Furthermore, I would like to extend my gratitude to the student members of the Power Systems Group: Colin Ogilvie, Thiago Szymanski, Behshad Mohebeli, Isabel Barnola, Mohammed Namidi, Sihun Song, Matthias Musil, Robin Ramin, Cristoph Diendorfer, Caleb Gove, and Angelina Lanh. A deep thank you to the staff: Matthew Bosworth, Mike Sloderbeck, Dionne Soto, Harsha Ravindra, Jodie Bell, and Nick Hoeft.

I would also like to extend a thank you to my friends and colleagues at the FAMU-FSU College of Engineering and the Center for Advanced Power Systems for their support: Satish Vedula, Yu Zheng, Taiwo Ojo, Gabriel Omoniyi, Sina Ameli, Skrikar Telikapalli, Aniket Jambhale, Rami Yehia, Ioannis Zografopoulos, Tanvir Toshon, Aakash Nagarajan, Dr. Sam Yang, Dr. Mehrzad Bijaieh, and Dr. Jose Ospina. I would like to extend another thank you to Dr. Olugbenga Moses Anubi for his continued support and wisdom throughout my academic career.

I would like to express my gratitude to the administrative staff at the college and CAPS, thank you Melissa Jackson, Nancy Reiney, and General Gaskin for all the help provided to me.

Finally, I would like to express my sincere gratitude to my family and my loving wife for providing me with the strength to continue forward with my endeavours and for helping me become a better person every day. This work was sponsored by the US Office of Naval Research under grant number N00014-16-1-2956.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES	All-Electric Ship
APDP	Adaptive Percentage Differential Protection
CAPS	The Center for Advanced Power Systems
CEF	Control Evaluation Framework
CHIL	Control Hardware-in-the-Loop
CIM	Control Interface Manager
CRTT	Controller Round-Trip Time
CSRT	Controller to Simulation Response Time
CTRL	Controller
DDS	Data Distribution Service
ESRDC	Electric Ship Research Development Consortium
FM	Fault Management
FMI	Functional Mockup Interface
FMU	Functional Mockup Unit
FPGA	Field Programmable Logic Array
HFS	High-Fidelity Simulation
ICD	Interface Control Document
IDL	Interface Description Language
IEEE	Institute of Electrical and Electronics Engineers
IFAC	International Federation of Automatic Control
IPES	Integrated Power and Energy System
IRD	Interface Requirement Document
IP	Internet Protocol
JSON	JavaScript Object Notation
LFS	Low-Fidelity Simulation
MVAC	Medium Voltage Alternating Current
MVDC	Medium Voltage Direct Current
OMG	Object Management Group
OSI	Open Systems Interconnection
PCIe	Peripheral Component Interconnect Express
PCM	Power Conversion Module
PGM	Power Generation Module
PMM	Propulsion Motor Module
PTP	Precision Time Protocol
QoS	Quality of Service
QSI	Quasi-Static Interface
QSPS	Quasi-Static Power System
RTDS	Real-time Digital Simulator
RTS	Real-time Simulator
SCRT	Simulation to Controller Response Time
SIM	Simulation Interface Manager
SPS	Shipboard Power System
SRTT	Simulation Round-Trip Time
UDP	User Datagram Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

ABSTRACT

A medium voltage DC (MVDC) shipboard power system (SPS) architecture is a notional candidate design for future "all-electric" ships, as it is expected to increase in power density and distribution efficiency. However, the lack of experience in proven MVDC technology has inherent risks of substantial delays during the development process of controllers. Model based system engineering methods, specifically controller hardware in the loop simulations, provide powerful means to lower the risk of a control development cycle. The ability to incrementally refine the fidelity of the model during the development and integration of a controller towards a more accurate representation of the power system is expected to reduce the frequency and severity of potential problems.

In the early stages of the control development, a control developer's main concern is to verify that the algorithm implementation meets the functional requirements to satisfy the needs of the shipboard power system. Performance of the implementation at this stage is of less concern. There may also be limited knowledge about the specific power system that in the future will communicate with the controller. Hence, initially a reduced fidelity model can suffice. As the development process progresses, the model will evolve to ever increasing fidelity and accuracy. Eventually, the model will have to include all the relevant details and execute in real-time to test the controller's real-time performance before field deployment.

In order to facilitate such progressive refinements in fidelity and real-time capability, an adaptable signal interface between high-level supervisory controller and models has been developed and tested. In particular, this thesis presents design criteria and requirements for such an interface focusing on the development of a fault management approach for a breakerless MVDC system. The signal interface considers different operational platforms, changing communication behaviors, signal descriptions, and simulator and controller endpoints, to connect different simulation environments to the controller making it adaptable.

Experimental results validate the interface design and control development process. The process begins by evaluating the functional requirements of the fault management system in a low-fidelity discrete event simulation. Once, there is sufficient confidence that the algorithm can perform its core functionalities, by virtue of the proposed interface, the development process continues on a high-fidelity real-time simulation. At this point, the process proceeds to evaluate the functional

and performance requirements with respect to time, arriving at a more rigorous evaluation of the control. The thesis concludes that the proposed approach is a feasible method to ease the control development process, by incrementally integrating models of higher fidelity. While acknowledging the additional work required to continually improve the process for a more generic use in the future.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Future generations of all-electric ships (AES) are envisioned to be equipped with an integrated power and energy system (IPES). Said system is tasked with the automated allocation of resources on a shipboard power system to meet the mission requirements. This automated process supports efficient use of resources, as well as aiding in the fault handling capabilities of the power system. The Electric Ship Research and Development Consortium (ESRDC) and the Center of Advanced Power Systems (CAPS) are researching heavily into the electrification of the next generation shipboard power systems.

The hardware infrastructure at CAPS lends itself to real-time simulation of electrical power systems and communication network to evaluate system-level control performance. It consists of dedicated real-time simulators (RTS), embedded controls, and servers to enable Controller Hardware-in-the-Loop (CHIL). There exists multiple variants of notional Shipboard Power System (SPS) models implemented [1, 2] in the Real Time Digital Simulator (RTDS) [3] environment. A medium voltage dc (MVDC) shipboard power system is a good candidate for the future generation of AES, because of its increased power distribution efficiency and power density [4]. These notional SPS models are then used to explore potential challenges in a MVDC shipboard power systems.

The benefits of MVDC systems are impeded by the vast experience and technology readiness of traditional Medium Voltage Alternating Current (MVAC) systems. Switching over from MVAC to MVDC comes along with different hardware and control designs, creating new avenues of research. To ease the transition from a MVAC system to a MVDC system, CAPS leverages the control evaluation framework (CEF) to scrutinize and further study the performance and functionalities of control algorithms [5, 6, 7, 8, 9, 10, 11]. The CEF is an automated test framework to evaluate system-level control implementations in a relevant environment. It coordinates simulation hardware, control hardware, and automated evaluation of test results with the use of metrics. It consists of a set of modules that enable the probing, characterization, stress-testing of said system-level controls.

Specifically, it aims to ease the development process from early stage development to the technology demonstration stage, thus aiding the validation and verification of system requirements.

1.2 Thesis Statement

The CEF is a complex system with a potentially lengthy integration process. So how can one facilitate the integration process of a system-level controller to the Controls Evaluation Framework? The focus is put on the interface because they are the crux of system integration and if not handled correctly can lead to undesirable outcomes. Therefore, the following research question is posed, how should an interface be designed to allow for control developers in the early development stages to validate their algorithms on low-fidelity models while ensuring the capability to transition to more relevant environments? A tentative answer consists of the following statement,

> *Control developers can validate their algorithms in increasing levels of environment relevance when the interface maintains a common communication point and the interface specifications are modifiable.*

The following questions are answered herein:

- How does such an interface look like?
- What are some existing modifiable interface implementations?
- How will the interface have modifiable specifications?
- How will the transition between models of varying fidelity occur?

1.3 Scope of Thesis

The work in this thesis presents the requirements and design criteria for an interface that facilitates the progressive increase in model fidelity between a controller and a simulation environment. In particular, this work addresses the development process for a fault management approach for a breakerless MVDC shipboard power system. The two simulation environments simulate a notional MVDC SPS model, in a low-fidelity quasi-static discrete event simulation and a high-fidelity real-time simulation. During each increase in model fidelity the communication behavior between said systems may change, requiring that this interface be adaptable to varying forms of signal exchange mechanisms. At each stage of the control development process, a subset of the overall control

requirements must be met. The very ability for the interface to meet the varying Experimental results validate that the development process maintains requirement traceability, thus ensuring that at each subsequent step the prior set of requirements is verified. It is by the virtue of the proposed adaptable signal interface design that such incremental steps are possible. The thesis concludes that the proposed interface design is feasible while acknowledging the additional work required to continually refine the process for a more generic use in the future.

1.4 Contributions

The contributions of this thesis the demonstrate the effectiveness of an adaptable signal interface, with the following efforts:

- A notional four-zone MVDC SPS in a quasi-static power system (QSPS) simulation environment.
- A QSPS environment equipped to enable controller hardware-in-the-loop testing.
- The requirements and design criteria for an adaptable signal interface.
- A method to incrementally integrate system-level controllers with model of varying fidelity to ease the development process.
- Partitioning of fault management requirements into verifiable subsets for each level of model fidelity enabling requirement evolution.
- A case study with respect to a fault management approach to show how to transition from one model to another.
- A process for de-risking the development cycle of system-level controllers.

The work outlined in this thesis results in a conformance test kit for control developers. This conformance test kit provides a manner for which a new control developer can set up their algorithm to interface with the Control Evaluation Framework. Once, the interface is established the control's performance can be further characterized and its requirements can be validated and verified in a more relevant environment.

1.5 Thesis Outline

The remainder of this thesis is structured as follows, Chapter 2 provides four sections that give an overview of the necessary concepts to outline the work involved in this thesis. The first section summarizes the different System Engineering methodologies employed to characterize interfaces and capture their requirements, as well as introduce existing interface implementations. The second section provides an in-depth explanation into the quasi-static power system modeling environment, relaying the modeling of the power system components and its software implementations. The third section gives an overview of the notional MVDC shipboard power system model, and its implementation in the quasi-static power system environment and in the Real Time Digital Simulator (RTDS) environment. Finally, the fourth section entails a brief explanation of the different modules in a fault management algorithm namely, fault detection and localization, fault isolation, and reconfiguration. In Chapter 3, the design of the interface is explained and its timing performance characterized. In Chapter 4, the interface is used to validate and verify a subset of the requirements needed for a fault management algorithm in a low-fidelity environment and then the same interface is used to validate and verify a subset of requirements in a more relevant environment with a real-time high-fidelity simulation environment. Finally, in Chapter 5, the future works and concluding remarks are given.

CHAPTER 2

RELEVANT LITERATURE

2.1 System Interface

In this modern-day world, most of the system we build are complex e.g. an airplane, rocket, or power system grid. A lot of effort is focused on the designing the individual components that make up the whole ensuring its meets functionalities, tolerances, and specifications. Often interfaces are neglected, thus resulting in potential "weak points" for a system. The saying "a chain is only as strong as its weakest link" cannot be truer, all the work developing a subsystem can be squandered by a faulty interface. As an example, the Mars Climate Orbiter was launched by NASA on December 11, 1998, to study the Martian climate. However, 286 days into its mission all communication was lost with the spacecraft, because the onboard computer received a non-SI unit for pound-force seconds instead of Newton-seconds [12]. As we can see, interfaces are at the crux of system integration, and, therefore, careful consideration must be taken when designing them.

Before continuing let the discussion begin with a definition, Fosse and Delp define an interface as "The system boundary that is presented by a system for interaction with other systems." [13] For every interface there must be a corresponding interface specification defined as "The details that describe the nature of the boundary presented by a system or component in terms of properties and functionality."

2.1.1 Canonical Types of Interfaces

In Systems Engineering there are four canonical types of interfaces [14],

- Physical Connection
- Mass Flow
- Information Flow
- Energy Flow

Two parts are in physical connection if they directly touch each, are permanently connect to each other, or they have a reversible connection between them. For example, welds, USB plugs, and a finger touching a touchscreen. An energy flow exists if there is an exchange of work between two

components. This can be in the form of electrical power, thermal power, radio frequency power, and mechanical power. Energy typically flows in a directed fashion from a source to a sink. Whenever there is an exchange of mass between two components there exists a flow of mass, this can be in the form of gases, fluids, and solids. Like the flow of energy, mass flows from a source to a sink.

Finally, but not the least, there is information flow which is defined as "the movement of information between people and systems" [15]. When there is an information flow from a system to a user or operator there is need for a human machine interface (HMI) or a graphical user interface (GUI), both of which are interfaces. When there is an information flow between two systems, interface specifications are needed to describe the nature of the system boundaries, for example, in analog communication ADC/DAC are required, in digital communication DIO are necessary, and in wireless communication a standard such as the IEEE 802.11.

2.1.2 Interfaces Layers

One way to ease the design process of an interface is to apply structure to it. This structure provides a systems engineer with the ability to focus concerns on specific portions at a time. In the late 1970s, in the efforts characterize and standardized the communication functionalities of emerging computing systems the Open System Interconnection (OSI) model was born. The OSI model partitions the flow of data in a communication system into seven abstraction layers, from the low-level physical implementation to the high-level representation of a distributed application. Figure 2.1 provides an intuitive representation of each of the OSI layers and how they interact with one another. A brief description of each layer is given in the following list [16]:

1. Physical Layer: Responsible for the physical connection between devices.
2. Data Link Layer: Responsible for the node to node delivery of a message.
3. Network Layer: Responsible for the transmission of data from one host to another located on a different network.
4. Transport Layer: Responsible for the End to End delivery of a complete message through quality of service.
5. Session Layer: Responsible for establishment of connection, maintenance of sessions, authentication, and security.
6. Presentation Layer: Responsible for the translation of the data into the required format.

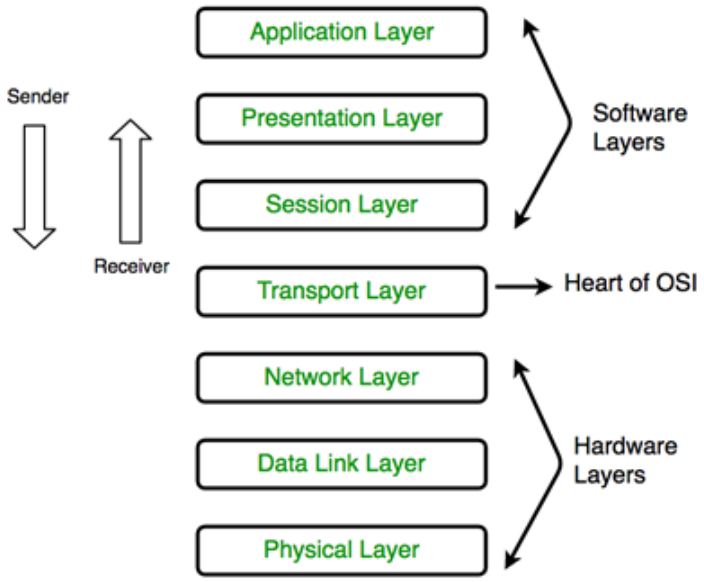

Figure 2.1: Partitioning of the OSI layers [16].

7. Application Layer: Responsible for creating the data, serving as an interface to access the network.

Each one of the layers can only interact with the layer directly beneath or above, essentially making each layer an interface to the other layers. Phillips proposes a simplified three-layer model [17] to aid in the design stages of an interface:

- Semantic
- Descriptive
- Physical

Providing yet another perspective for which a Systems Engineer can approach interfaces. The physical layer is like the OSI physical layer, referring to the parts that can be held physically in one's hand. The descriptive layer describes the information or material that passes through the physical layer, e.g., with respect to electricity in a home "125 Volts" and "15 Amperes" would suffice. The semantic layer relates to the messages that are passing through the interface and their interpretation.

2.1.3 Interface Documentation

Through the years NASA has compiled countless lessons into what is known as the NASA Systems Engineering Handbook [18]. The specific section of interest for the work pertaining to this thesis is Interface Management, which is a process to assist in controlling product development when efforts are divided among parties and/or to define and maintain compliance among the products that should interoperate. Figure 2.2 describes the interface management process.

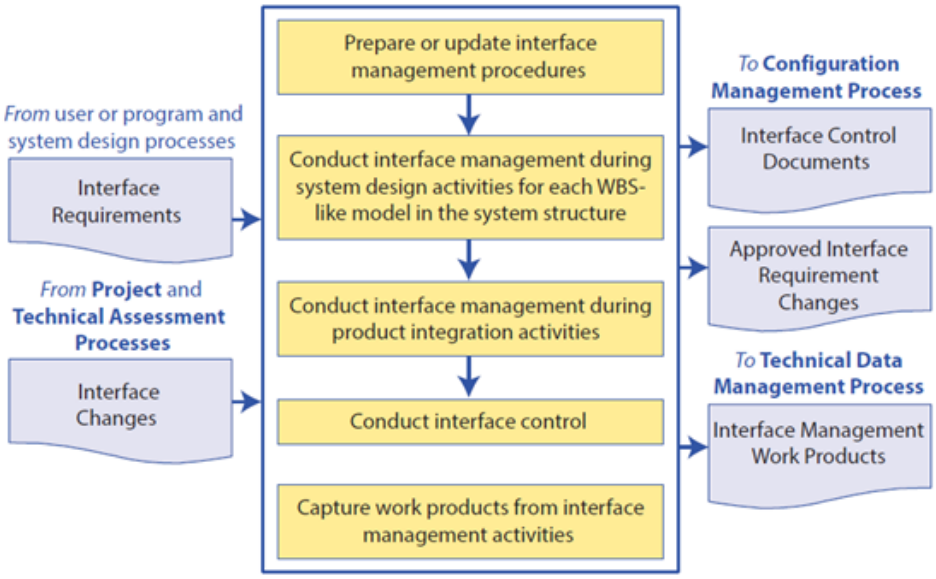

Figure 2.2: Interface Management process as defined by the NASA SE Handbook [18] .

Here we can see that the inputs to the process are the interface requirements and any proposed changes to said interface. The process begins at preparing the interface management procedures, such as establishing responsibilities, how changes will be handled, and what type of process will guide the maintenance and control of the interfaces. The next step consists of establishing the origin, destination, stimuli, and special characteristics of the interfaces that need to be documented and maintained. During the product integration phase, activities support the system integration process and ensures the interface are compatible with the specifications. Finally, an Interface Working Group (IWG) is responsible for the planning, scheduling, and execution of all interface activities. During each phase, the work products are captures ranging from rationales, assumptions, lessons learned, etc. On the output end of the process includes the interface control documents, approved interface

changes, and the interface management work products. The following section describe the interface requirement and interface control documents in more detail.

Interface Requirements Document (IRD). As defined in [19], an interface requirement document defines "the functional, performance, environmental, human, and physical requirements and constraints that exists at a common boundary between two or more systems." This type of document is normally with "shall" statements, for example, "The motor shall provide mechanical power to the electrical generator through a common shaft." A corresponding interface control document (ICD), then discusses the rotational speed, torque, geometry, etc.

Interface Control Document (ICD). Another form of interface documentation are interface control documents. Figure 2.3 shows an architecture that clarifies key interfaces, that further details can describe the nature of that boundary. Those details are encapsulated in an ICD document, one ICD can contain multiple interface descriptions. All the inputs to and outputs from a system must

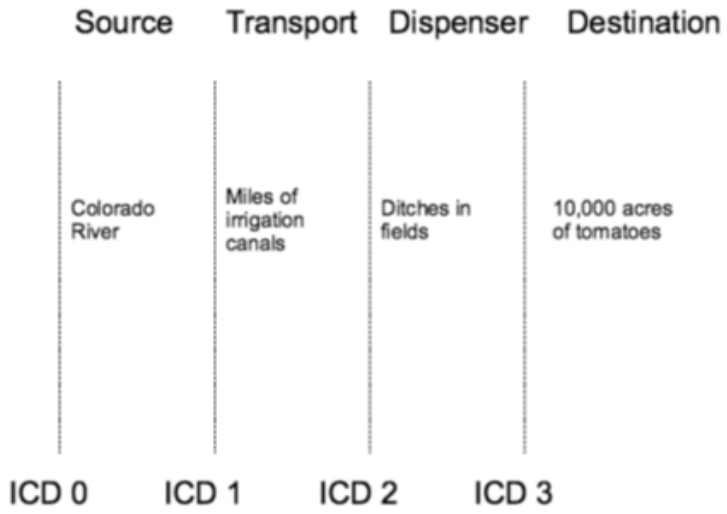

Figure 2.3: System boundary diagram for a sample Irrigation System [17]

be described in either low-level or high-level details. Specifically Figure 2.3 describes the interfaces for an irrigation system fed by the Colorado river. It has the following four interfaces:

- ICD 0: Separates the source of water for the Colorado River from the river itself.
- ICD 1: Separates the river from the irrigation canals.

- ICD 2: Separates the irrigation canals from the small ditches in the fields.
- ICD 3: Separates the water-carrying ditches from the acres of produce.

Each ICD is structured with either the OSI model or the Phillips model, both providing a certain insight to the interfaces.

2.1.4 State of the Art: Interfaces

This chapter discusses the approaches found in literature and gives an overview two specific interface designs that are relevant to theme of the thesis. The two frameworks are called Functional Mockup Interface (FMI) and Data Distribution Service (DDS).

Functional Mockup Interface. Functional Mockup Interface [20] rose from the need to support the exchange of simulation models between suppliers and original equipment manufacturer regardless of the model's implementation. FMI is a tool independent standard for the exchange of dynamic models and for co-simulation. Specifically, of interest is the FMI for Co-Simulation, where multiple simulation can be coupled together. A master algorithm controls the data exchange between subsystems and the synchronization of all slave simulation solvers. Figure 2.4 shows an example FMI for automobile implementation, each component connected to the FMI, is known as a Function Mockup Unit (FMU). Each FMU contains the following:

- XML-file: The XML-file contains the definition of all the variables of the FMU that are available to the environment, as well as other pertinent model information.
- C-functions: All the necessary model equations, communication initialization, and data exchange capabilities are provided as C-functions.
- Auxiliary files: Includes any file such as documentation, maps, and/or object libraries.

In summary, FMI eases the model exchange and co-simulation between different tools by providing a standard so that different developers can interoperate their systems.

Data Distribution Service. The second framework is Data Distribution Service [22] a high-performance, scalable, secure, and data-centric publish-subscribe data distribution system. It is based on the Open International Data-Centric Connectivity Standard from the Object Management Group (OMG). A data-centric architecture is based on a data model that is appropriately

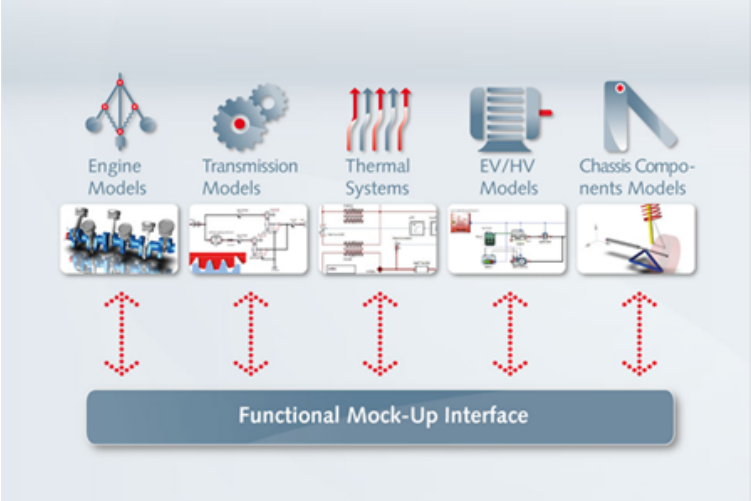

Figure 2.4: Example FMI setup [21].

documented i.e., human understandable, formally defined i.e., machine understandable, and discoverable i.e., can be found during execution. By describing DDS with the Unified Modeling Language (UML), it provides a platform independent model. Figure 2.5 provides a look into the structure of a DDS implementation. The center orange strip corresponds to the data bus, where different applications can access the information. Some of the blue boxes below the orange data bus are Quality-of-Service (QoS) characteristics, that provide control over almost every aspect of the data distribution such as reliability, logging, or message persistence. These QoS provide participants with different manner to interact with the data on the bus. DDS leverages the OMG Interface Description Language (IDL), that provides another standardized form to characterize the interaction between two system boundaries.

A central theme for both frameworks consists of a standardized language to characterize the interface. For the Functional Mockup Interface, it is the XML-file and for Data Distribution Service it is based on UML and IDL.

2.2 Quasi-Static Power Flow

Future all-electric navy ships will be equipped with emerging electric mission loads and sensory system that could exceed power generation capacity for short periods, thus motivating the investigation into the sizing and placement of energy storage [24]. There are many design considerations with

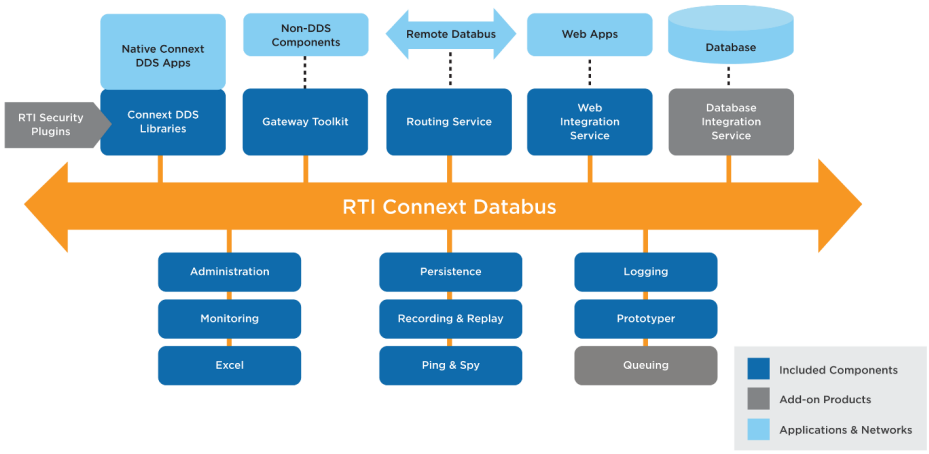

Figure 2.5: An implementation of the OMG DDS by RTI [23].

respect to the placement of energy storage units ranging from fault current contributions, dynamic power transfer capabilities of the power system, system stability, etc. Considering all aspects may prove challenging due to the large parameter space and the computational burden for the dynamic simulations needed to evaluate the different designs. A framework that considers a subset of those considerations may be useful in the early design stage to provide a better direction for comprehensive design analyses in later stages. The Quasi-Static Power Flow toolbox does exactly that, it provides an early design tool for analysis of systems employing distributed energy storage systems, using a quasi-static analysis of power transfer through the system. The framework considers the following aspects

- The topology of the power system networks, including power limits on generations units and power delivery equipment.
- The effects of load buffering, leveling, and shedding are included as they can play a major role in the capacity of the system to deliver energy to mission critical loads.
- The incorporation of uncertainty in the early design states, as to avoid making assumptions and making wrong conclusions.
- The size, weight, and cost of a system are included in the analysis to balance physical constraints and the energy constraints.

The very fact that this toolbox captures the connectivity of the power system, power capacities, and load weight (criticality) lends itself useful to the early development of distribution network reconfiguration algorithms. The main goal of distribution network reconfiguration algorithms is to modify the connectivity of the power system to attain a certain goal. These goals range from minimizing system losses to recovering critical loads, the scope of this work specifically address the recovery of critical loads.

In the following section a brief overview of the Quasi-Static Power System toolbox is given in Section 2.2.1, Section 2.2.2 explains the concept behind a graph-based power flow solution. Then, Section 2.2.3 explains the modeling of each of the power system components of interest. For details pertaining to the Quasi-Static Power system toolbox please refer to Appendix A.2 for an explanation of the implementation details.

2.2.1 Overview

A graph-based power flow is used to assess the ability of the power system sources to meet load demands in a single point in time, based on the connectivity of the system and component power capacities. This module is then used by a quasi-static time-domain module to account for the change in power over time. This quasi-static time-domain module can be either a time-series power flow or an event-driven power flow simulation. Both the power flow module and the quasi-static time-domain modules are explained in further detail in Sections 2.2.2 and Appendix A.2, respectively. Casting the system topology as a directed graph allows the power flow problem to become, without dynamic behavior, a maximum flow problem. The graph-based power flow problem can then be solved with linear programming techniques, allowing for a relatively computationally inexpensive solution.

Figure 2.6, presents a flow diagram of the process to simulate a Quasi-Static Power System. The process begins with the creation of a power flow topology that requires a corresponding table file that embodies the power system connectivity, power system rating and component weights. Depending on the type of simulation one wishes to execute either a time-series or an event-driven discrete solver is used. Afterwards, the corresponding power system components are added to simulation, as denoted in Figure 2.6. Finally, the quasi-static power system is simulated. For a more in depth explanation of a quasi-static model approximation and how it pertains power system models, please refer to Appendix A.1.

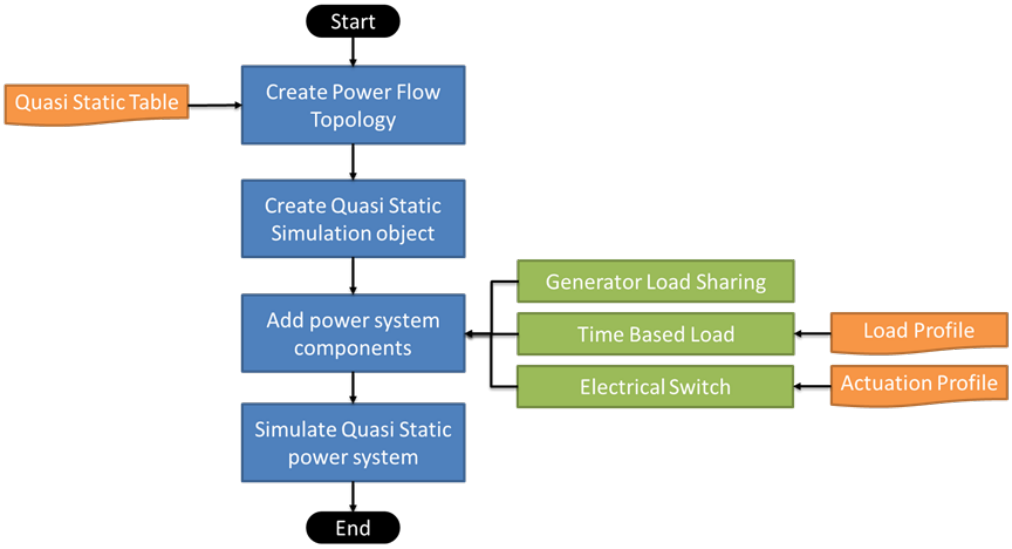

Figure 2.6: Flow diagram of quasi-static power system solver.

Modeling Limitations. Due to the time-scales used during the modeling of the quasi-static power system simulation environment, it will not be able to represent the fault transient information needed for protection systems to properly detect, locate, and isolate an electrical fault. It can, however, emulate the response of a power system to a fault, given a fault location a priori, by injecting an extra power flow to resemble a potential fault current, to trip the protection system in a scripted manner.

For example, say that at time t_{k-1} there is no fault, at time t_k a fault occurs in the system, and then at time t_{k+1} there is an extra amount of power flow in a cable that would either trip an overcurrent or differential based protection scheme. Utilizing the fault profile available in the quasi-static environment, the simulation injects a power flow at a given time instance, thus effectively emulating a fault in the quasi-static power system.

2.2.2 Power Flow Solution

The power flow module determines the power flow of each directed edge, thus calculating the power that will be delivered to each load at a fix point in time, given load demand and criticality, source generation capacity and priorities and the power capacity of the transfer components. The quasi-static power flow [25] problem is casted as, (2.1), a maximum weighted flow problem of the

following form:

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & w^T x \\ \text{subject to} \quad & A_{eq} x = 0 \\ & x_{lb} \leq x \leq x_{ub} \end{aligned} \tag{2.1}$$

Here, x is the optimization variable namely the flow in each edge, w is the weights of each edges, A_{eq} defines the balance of the flows as in (2.1), 0 is a vector of zeros, and x_{lb} and x_{ub}, define the lower and upper bounds, respectively, of each edge. The use of a graph-based power flow solution greatly minimizes the computational expenses and required information. In Figure 2.7, we can see an example power system with a ring bus configuration represented as directed graph. With

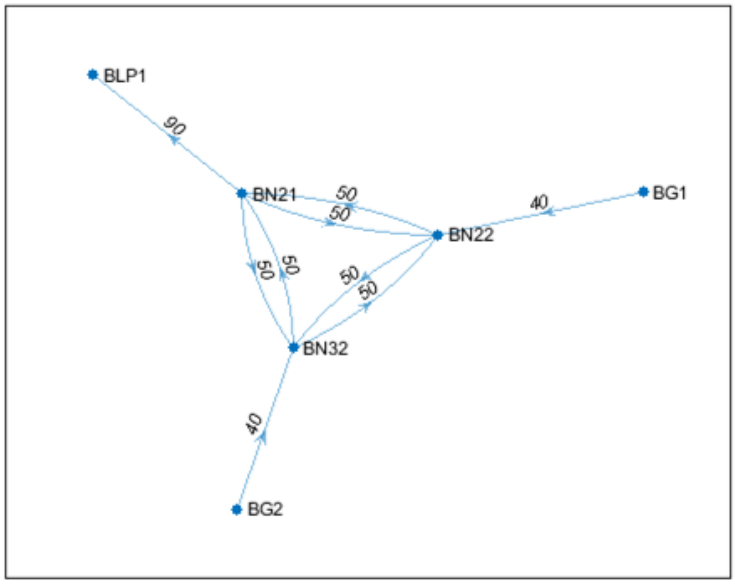

Figure 2.7: Example power system network with a ring bus configuration.

this formulation, generators are represented as sources (BG1 and BG2), loads are represented as sinks (BLP1), buses are represented as nodes (BN21, BN22, and BN32) and finally power transfer components (cables, transformers, converters, etc.) are represented as edges. It is important to note that this power flow approach does not consider voltage nor currents, thus eliminating the need to specify impedance information as in traditional load flow analysis.

As aforementioned, the graph-based quasi-static power flow problem is casted as maximum weighted flow problem, this type of problem can be solved with linear programming techniques.

Should the case arise where load demand cannot be met, the power flow solution will then reflect the curtailing or shedding of loads with respect to the system constraints. The addition of generator and load criticality values as weights allows the solution to present the optimal power flow given the constraints, such that the solution will automatically load shed based on those criticality values. Therefore, the power flow module provides the capability of calculating the power flow and power levels given the system constraints and at fixed point in time.

2.2.3 Component Models

The power flow module calculates the flow in terms of graph-based concepts such as sources, sinks, and edges. Therefore, power system functionalities must be included by another higher-level function. To that end power system components are implemented as MATLAB classes, each component class specifies the capacity and criticality that is used by the power flow module. At every time step, each component keeps track of any needed state variable and then operates its specified logic for the next power flow solution. Three basic types of components are needed to adequately represent a power system: a load component, a generator component, and a power transfer component. The following sections explain the components as wells as any of the derived components.

Load Component. A load component models an electrical load, which supports a time-based demand profile and priority based on integrated power demand deficit, where said demand profile is defined as a piecewise function [25]. The concept of integrated power demand deficit is expressed in (2.2),

$$E_d(t) = \int_{t_d}^{t} [P_{demand}(\tau) - P_{delivered}(\tau)]\, d\tau \tag{2.2}$$

Where,

- t_d is the instance in time when there is a deficit of power generation.
- $E_d(t)$ is the time-varying integrated power demand deficit.
- $P_{demand}(t)$ is the power demand profile.
- $P_{delivered}(t)$ is the power delivered.

The use of load priorities is useful when designating loads as vital, semi-vital, and non-vital. These load designations are necessary during the recovery period of a power system, to determine which

loads require power that can influence the success of a mission. The behavior of a load component is specified by the following parameter:

- $P_{demand}(t)$ is the power demand profile.
- I is the load priority.

Generator Component. A generator component acts a source in the power flow formulation, where a power capacity and source priority define its capabilities. The generator component can be configured with a time-based capacity profile and a static profile. The behavior of the generator component is specified by the following inputs:

- $P_{capacity}(t)$ is the power capacity of the generation unit.
- I is the priority for the generation unit.

Power Transfer Component. Power transfer components model equipment that is used to deliver power from sources to the loads, such as, cables, power converters, and transformers. These types of components correspond to edges in the power system, and, therefore, provide power capacity information to the power flow module. A power transfer component is configured with a time-based capacity profile, e.g. this profile can model capacity degradation loss of a component for a given scenario. A power transfer component is characterized by the following inputs:

- $P_{capacity}^{forward}(t)$ is the power capacity in the forward direction.
- $P_{capacity}^{reverse}(t)$ is the power capacity in the reverse direction.

Electrical Switch Component. The electrical switch component is a modification of the power transfer component modeling the actuation of an electrical switch. Like the power transfer component an electrical switch component corresponds to an edge and, therefore, provides a power capacity to the power flow module. The key difference between both components is that an electrical switch component is configured with a time-based actuation profile. An electrical switch component is characterized by the following parameters:

- $P_{capacity}^{maximum}$ is the maximum rated power capacity of the cable and/switching device.
- $S_{state}(t)$ is the actuation profile of that state of the switch, which can be either opened or closed.

The electrical switch is furthered extended as an event-driven component, where the actuation of the electrical switch can also be controlled externally by a controller interfacing with the quasi-static power system simulation.

Emulated Fault Component. An emulated fault component provides the scripting functionality to emulate a fault scenario on a specific edge. This method injects a value to the measurement of the power flow solution of the specified edge, before sending the system measurements to the controller. In essence it spoofs the power flow measurement to either trip an overcurrent limit or to trip a form of differential protection. An emulated fault component takes the following inputs:

- $F_{injection}(t)$: Injects a specified amount of power to a cable edge at a time instance specified by a fault profile.

2.3 Shipboard Power System

A shipboard power system's responsibility is to supply energy to a variety of loads, such as propulsion, communication, and mission loads. Considering the nature of the power system an SPS is comparable to an islanded micro-grid if it is not connected to the main grid when docked. The very fact that the SPS is isolated from the main grid implies very strict requirements with respect to providing power to vital loads. Given the space and weight constraints, the following is a list of some of the key system needs:

- high survivability, reliability, and robustness.
- no significant excess generation when the system is fully loaded.
- reconfiguration must occur fast enough to mitigate damage to ship and crew.

The list of needs are specific to the scope of this thesis, which pertain to recovery of power system after a disruptive event, such as an electrical fault. In order to meet those requirements, the SPS needs a system to enable those functionalities, covered in the following section. The following sections describe a medium voltage dc distribution system in the context of a shipboard power system, its implementation in a real-time high-fidelity environment and in a quasi-static low-fidelity environment.

2.3.1 MVDC Shipboard Power System

According to Doerry and Amy [26], there is a strong push and need to head in the direction of MVDC SPS, requiring mathematical models that capture sufficient details to represent the system. The following sections detail the modeling efforts of a four-zone notional MVDC shipboard power system in the Quasi-Static Power System (QSPS) environment and the Real Time Digital Simulator (RTDS) environment.

Notional Real-Time Zonal MVDC Shipboard Power System. The scope of this work studies how a fault management system interacts with a four-zone model [27]. Figure 2.8 shows a single line diagram of a notional four-zone shipboard power system architecture rated for a medium voltage of 12 kVdc and a power rating of 100 MW. The ship has four switchboards (SWBD) one per zone for both the port and starboard side of the ship. It has five Power Generation Modules (PGM), where three are main PGMs and two are auxiliary PGMs. Each PGM has dual wound generators, which are connected to two Modular Multilevel Converters (MMC), respectively, to provide dc power. It contains four Power Conversion Modules (PCM-1A) that provide power to low-voltage sections, there are two Propulsion Motor Modules (PMM) one for the port and starboard side of the ship, as well as system loads (SL). The red and black boxes represent the switches in the system, where a red box means a closed switch and a black box means an opened switch.

The figure shows the notional shipboard power system in a split plant alignment, but it can also be configured into a parallel and ring plant alignment given mission. This four-zone notional model implements a breaker-less approach, that is disconnect switches are used instead of circuit breakers. In the situation of a dc fault, the power electronics in the PGMs limits the fault current and can de-energize the bus in 10 ms. The disconnect switches (DS) only actuate once the current and the potential difference from the terminals are less than certain thresholds, for example 20 A for the current and 50 V for the voltage difference. These values can vary depending on characteristics of the disconnect switch under consideration. This system is implemented in the Real Time Digital Simulator (RTDS) environment, for a more detailed description of the Real-Time Implementation of the four-zone MVDC shipboard power system and its components please refer to its corresponding Model Description Document [27].

Notional Quasi-Static Zonal MVDC Shipboard Power System. The quasi-static four-zone MVDC shipboard power system is based on the same system architecture as in the previous

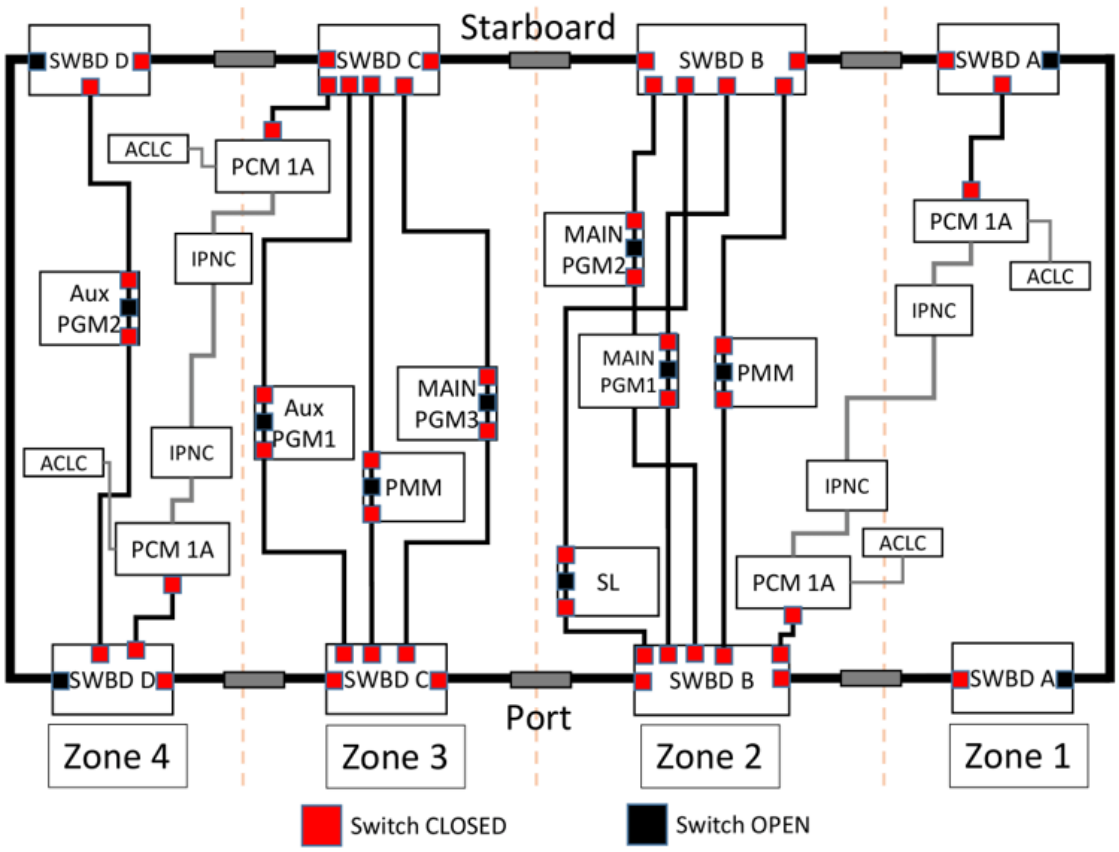

Figure 2.8: Single line diagram of a notional four-zone MVDC SPS architecture [27].

section, but it is implemented in the Quasi-Static Power System environment described in the earlier section. The Python graph-based representation helps generate the quasi-static network topology table for the notional MVDC shipboard power system. The table contains each edge, generator, and load in the power system, as well as their respective power ratings. The QSPS representation of the four-zone MVDC shipboard power system has the same connectivity as in Figure 2.8. It is important to note that the specific representation used in this thesis does not consider the low-voltage distribution network, therefore, PCMs are modelled as loads and not power converters. The power system connectivity, generator, and load power ratings are derived from the four-zone MVDC model description document [27]. The quasi-static model requires maximum rating for the cables provided in the Notional Ship Data document [28].

2.4 Fault Management

The International Federation of Automatic Control (IFAC) defines a fault as an un-permitted deviation of at least one characteristic property or parameter of the system from the acceptable condition [29]. Therefore, the purpose of a fault management system is to have a method to systematically correct the un-permitted deviations of the system. A *fault management* system consists of the following stages [30, 31]:

- Fault Detection and Localization.
- Fault Isolation.
- System Reconfiguration.
- Recovery.

2.4.1 MVDC Fault Management

This section provides an explanation of the implementation of the Fault Management algorithm in the context of a MVDC shipboard power system. The connectivity of the power system is stored in a graph data structure, where vertices can be generators, rectifiers, switches, loads, or current sensors, and the edges can be bus bars or cable sections. Using this graph-based fault management approach greatly simplifies the fault protection aspect.

Figure 2.9 shows the sequence of events that must occur to recover a MVDC SPS after a fault. Stage 1 consists of detecting that a fault has occurred and determining its location, during this stage the power converters in the PGMs should limit the current to reduce the severity of the fault. In stage 2, considering the breaker-less design of the power system implementation, the PGMs must de-energize the system before actuating any disconnect switches because they cannot break current flow. It is important to note that with a different protection device that can break current flow, the de-energization step can be disregarded. Stage 3 consists of the opening of the correct switches to isolate any current from flowing to the fault. During stage 4, the best sequence of switches must be actuated to restore the flow of power to the most vital loads to maximize operability. Finally, during stage 5, the PGMs re-energize the power system to continue system operations. The following sections provide more detail into the implementation of each steps.

Fault Detection and Localization. The primary protection system utilizes current differential scheme to the detection of faults that in conjunction with a graph-based system representation facilitates the localization [32].

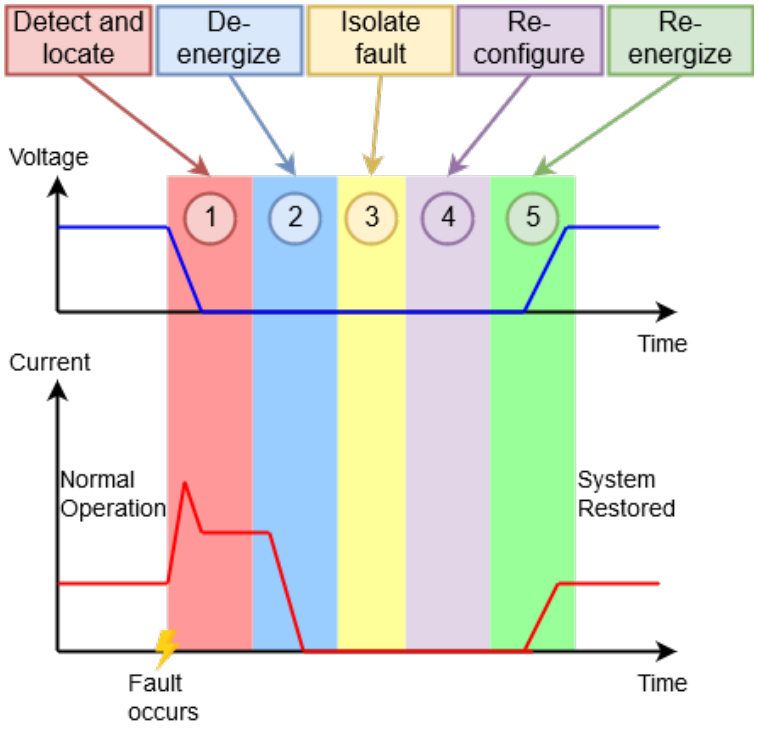

Figure 2.9: Sequence of event during an MVDC fault.

Adaptive Percentage Differential Protection (APDP). Defines the operating current I_{OP}, the restraint current I_{res}, the minimum operating current I_{min}, and a slope threshold K. APDP is based on Kirchhoff's Law, where the sum of the currents entering a component must equal the amount exiting. Where I_{OP} is defined as follows,

$$I_{OP} = \left| \sum_{i=1}^{n} I_i \right| \tag{2.3}$$

And I_{res} is defined as,

$$I_{res} = \sum_{i=1}^{n} |I_i| \tag{2.4}$$

Henceforth the ratio between operating and restraint current will be referred to as the operating point. The following equation describes APDP,

$$fault = \begin{cases} true, & \text{if } : I_{res} > I_{min} \ \ and \ \ \frac{I_{OP}}{I_{res}} > K \\ false, & otherwise \end{cases} \tag{2.5}$$

Figure 2.10 shows two regions in which the APDP operating point may find itself in, namely the restraint region where no protection action occurs and the operation region which signals to the protection system to trip.

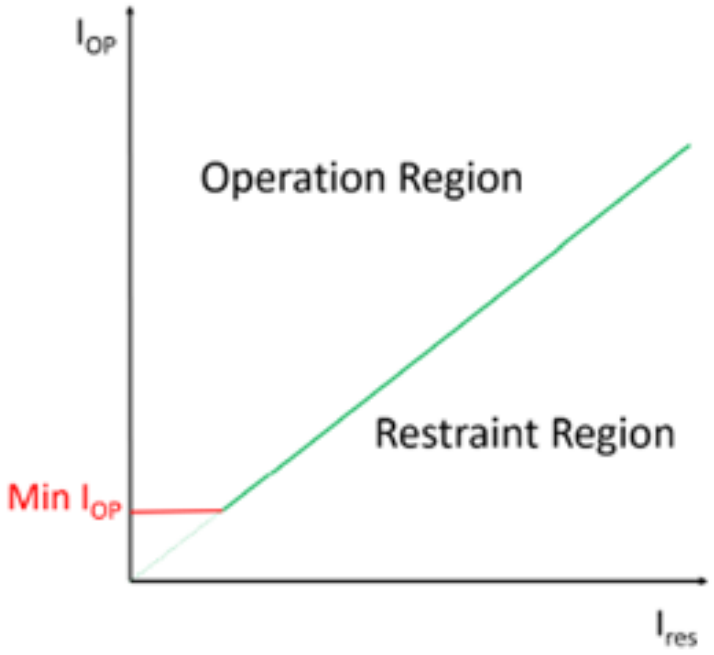

Figure 2.10: Adaptive Percentage Differential Protection [32]

Figure 2.11 shows how the operating point looks like with respect to time, where the blue line is the slope threshold, and the orange line is the measured operating point.

Graph-based Fault Detection and Localization. To facilitate the detection and localization of fault in a system the notional four-zone MVDC power system is partitioned into minimal isolation sections (MIS). A minimal isolation section is the smallest set of components that can be isolated from the system, thus requiring a device that is able to isolate current to enter or exit a minimal isolation section. Each MIS is stored in graph data structure to maintain the information of the connected components. Then, for each minimal isolation section the calculations for ADPD are performed and then after a fault is detected 3 times a signal is sent to trip the protection system [31, 32].

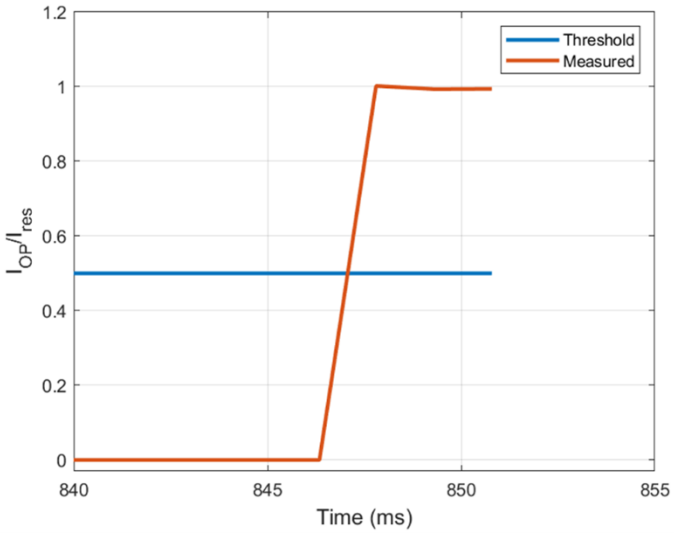

Figure 2.11: Ratio between operating and restraint current plotted against time

Fault Isolation. With the graph representation and the fault location, the graph can be traversed to identify connected components that can both conduct and isolate current. But, as aforementioned before isolation occurs the system must be de-energized, for which the graph is then traversed to locate the connected sources and de-energizes the units [31].

Graph-based Approach for Reconfiguration. The tree-based algorithm finds paths from sections with insufficient generation capacity to sections with excess generation capacity. It makes use of three different types of sections namely, a minimal section, a connected section, and a section, which are seen in Figure 2.12. The description of a minimal section was covered previously, a connected section consists of two minimal section connected by a switch, and finally a section can consist of multiple connected sections.

After isolating a fault, the algorithm proceeds to adaptively aggregate all the connected minimal sections. Figure 2.13 shows a connected minimal section highlighted in blue, where C denotes the section's generation capacity and F denotes a fitness value, a function of the load priorities in the section [5, 33]. From the algorithm's perspective, there is now only a new connected minimal section instead of three minimal sections, with a total capacity of 30 MW. From here, the algorithm grows trees from all the negative sections until reaching a section with excess generation capacity. The final solution is the tree with the minimum fitness value. Once algorithm calculates the final

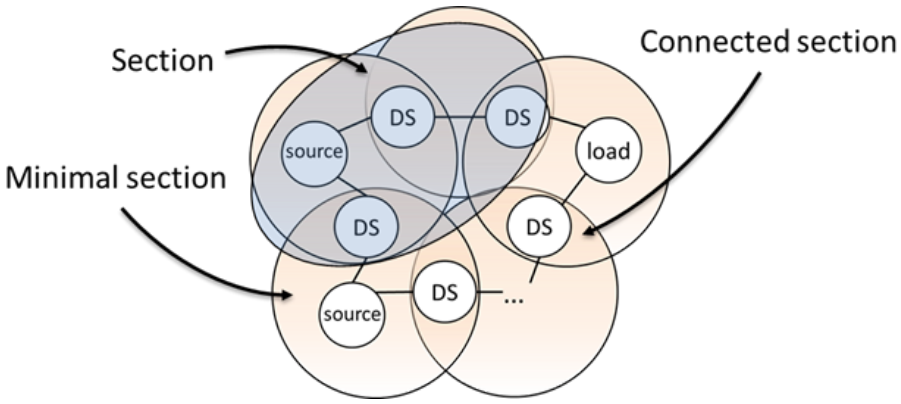

Figure 2.12: Example section graph of a power system

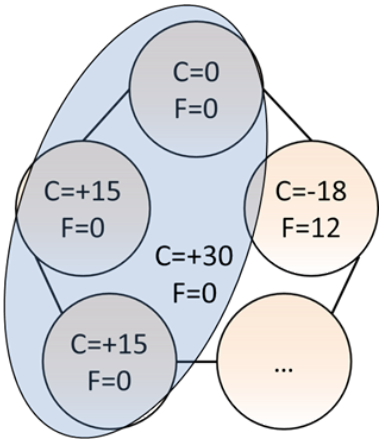

Figure 2.13: An example connected minimal sections with section generation capacity and its corresponding fitness value

solution, it sends the commands to module in charge of actuating switches. If the solution path contains switches that are currently energized it sends the necessary commands to de-energize their corresponding buses and then proceeds to send the command to actuate the switches.

2.5 Conclusion

This section encapsulates the interfaces from system engineering perspective covering the theory behind them, the necessary documentation needed to characterize an interface, and some of the state of the art interface implementations. Specifically, this thesis leans on the format of the documentation and the underlying commonalities of the some of the most well known interface

implementations. Then, the quasi-static power system simulation environment explanation provides an in-depth understanding into the modeling and implementation aspect of the toolbox to provide the necessary background information to transition from a low-fidelity model to one of high-fidelity. Following that is an explanation of the notional MVDC shipboard power system architecture, from which both a QSPS and an RTDS model are instantiated from to be used in the transition of models. Finally, what follows as a brief explanation of the control algorithm used to test the effectiveness of the interface design and development process.

CHAPTER 3

INTERFACE MANAGER DESIGN & IMPLEMENTATION

This chapter documents the necessary requirements and design criteria of an adaptable signal interface, as well as the acceptable forms of communication behaviors. The purpose is to clearly describe the exchange of signal information from a simulation environment to at least one controller. In essence, this chapter is a combination of an Interface Requirement Document (IRD) and an Interface Control Document (ICD), thus encapsulating salient details of this interface manager. The pertinent systems discussed in this chapter are as follows:

- A low-fidelity simulation environment which corresponds to a quasi-static simulation environment, specifically the Quasi-Static Power System (QSPS) environment.
- A high-fidelity simulation environment which corresponds to a real-time simulation environment, specifically the Real-Time Digital Simulator (RTDS) environment.
- An interface manager that consists of at least one simulation interface manager (SIM) and at least one control interface manager (CIM).
- A controller which corresponds to a power system control algorithm.

The chapter begins with a requirement analysis of the interface, it proceeds to present the adaptable signal interface design, and defines each interface boundary with respect to the OSI model. The following section provides further insight into how the interface manager exchanges signal information from a low-fidelity simulation and a high-fidelity simulation. Finally, a set of metrics characterize the performance of the interface. For the entire implementation of the interface manager code, please refer to Appendix C.

3.1 Interface Requirements Analysis

To adequately propose a solution to the given problem it is important to define the requirements. This section closely follows the outlined sections of an IRD as defined by [18]. It consists of describing the functional, performance, and interface requirements, as well as the constraints.

3.1.1 Functional Requirements

During the design phase of any given system, it is imperative to first characterize the required functionalities. The following list describes the required functionalities for the interface manager,

- The interface manager shall exchange signal information from a simulation environment to at least one controller.
- The interface manager shall exchange signal data with a low-fidelity simulation environment, such as the quasi-static power system model.
- The interface manager shall exchange signal data with high-fidelity simulation environment, such as the real-time digital simulation environment.
- The interface manager shall define the source of simulation signal data i.e., it can transition from low-fidelity simulation environment to a high-fidelity simulation environment.
- The interface manager shall exchange signal data with one or more controllers.
- The interface manager shall ensure messages are communicated to the controller in the correct order with respect to the time it was sent from the simulation environment.

The first requirement explicitly defines that this interface manager will mediate the exchange of power system signals between one simulation and many controllers. The second and third requirements state that this exchange can occur with either a low-fidelity or high-fidelity model. The fourth requires that the interface manager re-route the flow of signal information to a controller from one model to another. The fifth requirement states that multiple controllers must be able to exchange signal data with the interface manager. Finally, the last requirement states that there cannot be messages that are delivered out of order, if the interface manager can appropriately handle the messages sent, then the interface manager can maintain traceability of requirement when transitioning from one degree of model fidelity to another. Requirements traceability is defined as "the ability to describe and follow the life of a requirement in both a forwards and backwards direction." Transitioning from a low-fidelity model to a high-fidelity one provides a step in the forward direction. Note that this is not a limitation, if message re-ordering is necessary then a dedicated simulation of communication aspects can be included as in [11].

Derived Functional Requirements. From the core functional requirements, a set of derived requirements arises with further specifications the following outlines the derived functional requirements:

- Shall define a communication behavior for each controller.
  - Event-driven communication specifying who initializes the conversation.
  - Periodic communication with a specified update rate.
- Shall define a communication behavior for the simulation environment.
  - Event-driven communication specifying who initializes the conversation.
  - Periodic communication with a specified update rate.

These set requirements arise because stating that the interface manager shall communicate with either a controller or simulation is just not enough. One must specify how will the interface behave with respect to the exchange of signal information. The first and second derived functional requirements specific the communication behavior used to exchange signal data.

3.1.2 Performance Requirements

There exists a minimum set of performance requirements for the interface manager to uphold. The following describes the performance requirements:

- The interface manager shall be capable of periodic update rates of in the order of 1 millisecond.
- The interface manager shall ensure that during periodic communication, between a controller and a simulation environment, a minimum of 97% of the messages during an experiment must be sent before the end of the specified period.
- The interface manager shall ensure that at maximum 1% of messages can be delivered out of order.

The first requirement states that at the very least the interface manager must handle the exchange of signal data at a rate of 1 millisecond per signal exchange between simulation and controller. The second requirement states that at a minimum 97% of the total message exchanged must be sent in less than or equal to the specified periodic update rate. The value of 97% is chosen because it is a conservative estimate of the possible signal exchanges that may not meet the deadline, due to the activity on the operational hardware of the interface manager. Finally, it is required that

the signal data arrive in the correct order as it is sent, therefore there must be no possibility of the information arriving out of order.

3.1.3 Interface Requirements

There are a couple of underlying requirements for the physical interface of such an interface manager. The interface manager shall have a common physical connection that allows for information flow, e.g. Ethernet cables and network switches, between every component i.e., simulation operational platform, interface manager host machine and controller operational platform.

3.1.4 Constraints

One of the constraints, on the interface manager is imposed by the data type of the RTDS simulation environment. The interface manager shall handle 32-bit floating point values and integers with a little-endian byte ordering.

3.2 Interface Design Process

From the requirements analysis, one can extract that the interface manager (IM) will be responsible for the communication behavior with a simulation environment and a set of controllers. Changing which simulation environment to communicate with. As a result, the interface manager will need:

- A **common physical interface** that maintains the same communication endpoints throughout the development stages.
- **Modifiable specifications** to meet the requirements of each stage.

Several methods exists to ensure that the interface manager can carry out its responsibilities ranging from hardware to software approaches. The hardware approach ensures a deterministic response with respect to its timing and functionality, but the drawback is that designing it to meet varying forms of requirements may make it impractical to implement. The software approach can take considerably less time to implement and it is much more versatile so that requirements can be easily modified and simulation environments swapped.

Figure 3.1 shows the proposed adaptable signal interface design. Beginning on the left-most side of the figure, are the two simulation environments on their operation hardware, the simulation endpoints, and their respective simulation interface managers. Then on the right-most side of the figure

are the controllers on their operational hardware, the controller endpoints, and the corresponding control interface managers. In the center of the figure, denoted in a green box, one can see that an interface manager is composed of a simulation interface manager (SIM) and at least one control interface manager (CIM). Specifically, there are two types of SIMs a low-fidelity simulation (LFS) interface manager and a high-fidelity simulation (HFS) interface manager. Directly in the center of the interface manger exists a common physical interface which allows for the flows of information routed to a controller to transition from a low-fidelity simulation to a high-fidelity simulation environment. That is only possible with modifiable interface specifications that, denoted in the orange rectangle with a wavy base, representing a document. This document encapsulates the interface behavior and description. It provides answers to the following questions:

- How will the interface communicate?
- What will the interface communicate?
- Who will the interface communicate with?

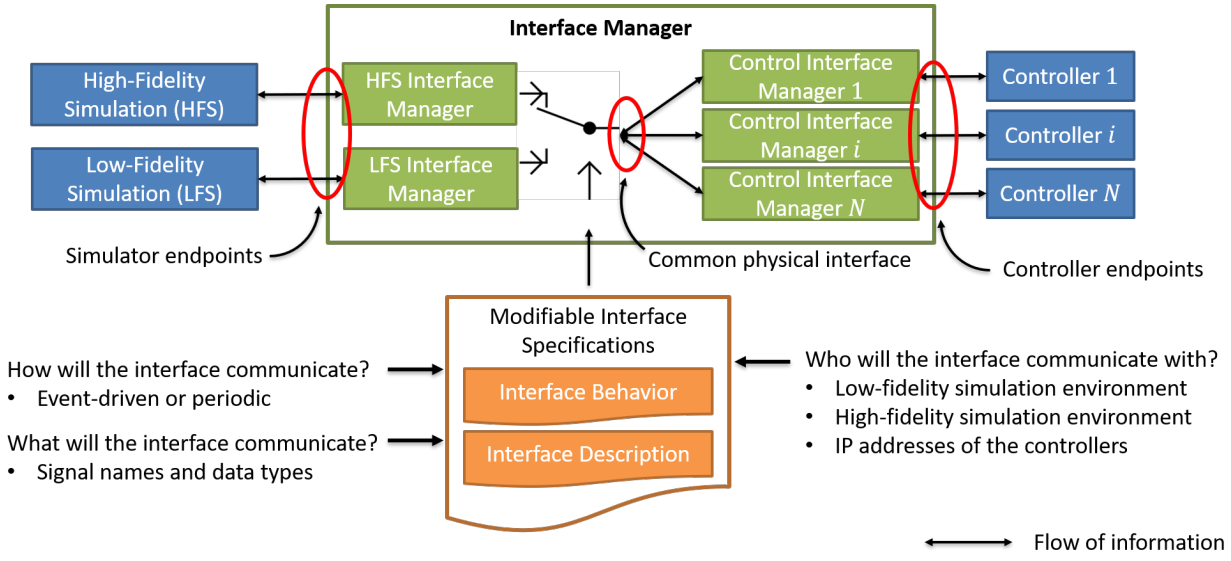

Figure 3.1: Proposed adaptable signal interface design

Functionally, the interface manager can now mediate the signal exchange between a simulation and at least one controller. The two different types of SIMs allow for the interaction with different simulation environments. The structure of the CIM allows for multiple controllers to interact with

a single simulation environment. Finally, the interface specifications permit for the re-routing of information flow between simulators.

After studying the requirements needed for this interface manager, the process begins to break down the design by applying the concepts from Section 2.1 to add some structure to the interface. At first glance there are seemingly two key interfaces, the one between the simulation environment and interface manager and the interface manager and controller. But after careful analysis, the interface manager can be effectively split into two systems resulting in the boundary diagram as seen in Figure 3.2. The interface manager is composed of a simulation interface manager (SIM) that manages the communication with its corresponding simulation environment represented as ICD-0, where the simulation environment will either be a low-fidelity or a high-fidelity model. And a control interface manager (CIM) that handles the communication with its associated controller, represented as ICD-2. Finally, the interface that allows for both SIM and CIM to communicate with each other is denoted as ICD-1. The interface denoted by ICD-0, can be further categorized with respect to

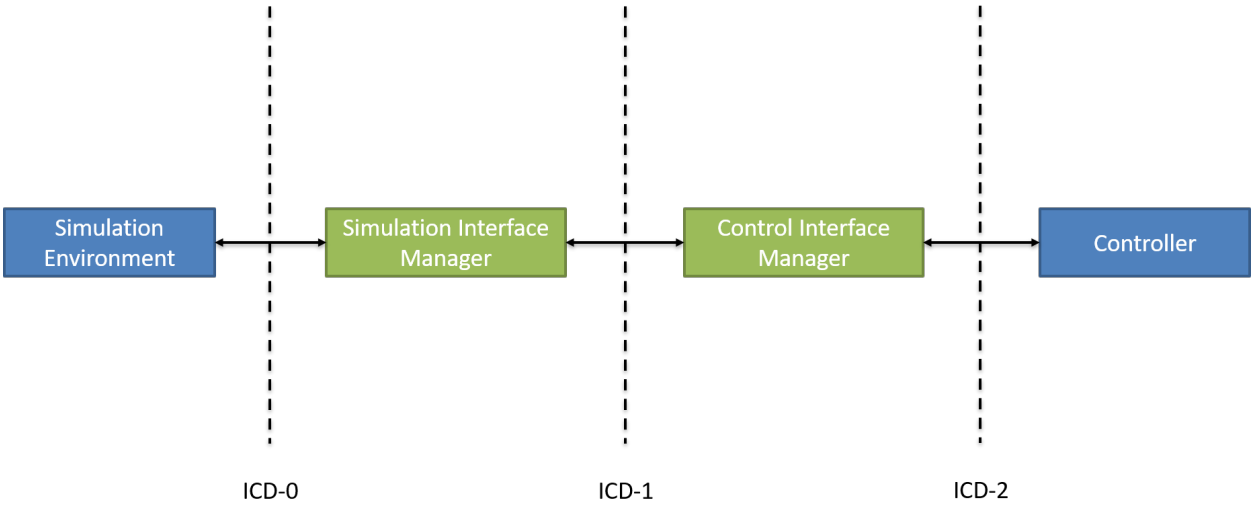

Figure 3.2: Interface Manager Boundary Diagram

its associated simulation environment into the following:

- ICD-0L: Describes the interface between a low-fidelity simulation environment.
- ICD-0H: Describes the interface between a high-fidelity simulation environment.

This distinction will help explain what is needed to transition from a low-fidelity environment to a high-fidelity environment in the later sections.

3.3 Interface Implementation

This section explains the implementation of the interface manager, so as to enable an adaptable signal interface. It begins by first describing the modifiable interface specifications and then proceeds to detail each interface as seen in Figure 3.2.

3.3.1 Modifiable Interface Specifications

A key design of the interface manager is its ability to be easily modified for varying specifications for communication interactions. That is accomplished with a file that encapsulates both the interface description and the interface behavior. The interface specifications are detailed in a YAML [34] file structure as seen in Listing 3.1. The following sections cover, the interface specifications in more detail, as noted in Figure 3.1, consisting of the interface description and interface behavior.

Interface Description. The interface description defines what information will be exchanged i.e., the name of power system signals and their data types as aforementioned which can either be a 4-byte float or integer value. On a semantic level, `to_control_signal_names`, is the power system time-varying monitored signals that capture aspects of a power system for example, power flow, switch states, generator power output and frequency, and rectifier output voltage. And for the controller, `from_control_signal_names` consists of the controller output that actuate power system components such as electrical switches and generator voltage set points. It also defines which simulation will be the main source of data with `simulator` and the IP addresses of the controllers, given by `network`, to which it will interact with.

Interface Behavior. The interface behavior characterizes how the interface will behave. There are currently two supported messaging patterns, given by `behavior`, namely request-reply, specified by `event` and periodic communication, specified by `periodic`. With the request-reply, which is event-driven, messaging paradigm the controller sets the update rate. The periodic update rate can be at a minimum 1 millisecond, the behavior of the interface manager is not defined if the update rate is faster.

```
name : TestController # control name
network : # control IP address and port
    - ip_address : '146.201.63.248'
    - port : 42000
to_control_signal_names:
    # signals directed to controller
    - C_21_22pYI : float
    - C_22_32pYI : float
    - C_32_21pYI : float
    - LP1YP : float
    - G1YP : float
    - G2YP : float
    - G1YVdc : float
    - G2YVdc : float
    - Dswt : int
from_control_signal_names:
     #signals directed to simulation
    - Dswtext : int
simulator : qsps

behavior : event
direction : sim_first
blocking : True
log_enable : False
send_period : 60000000 # nanoseconds
```

Listing 3.1: Example Interface Specification file

Listing 3.1, presents an example of what an interface specification file looks like for the example system shown in Figure 2.7 simulated in the QSPS environment. Table 3.1 describes of keywords in an interface specification file and their corresponding descriptions. It is important to note that use of both the `direction` and `send_period` keywords is not supported and only one must be used, it is presented in Listing 3.1 for the sake of completeness. Table 3.2 defines the acceptable forms of communication behavior supported by the interface manager. Option 1 defines an event driven messaging scheme that expects the QSPS environment to send a message first. Option 2 defines an event driven messaging scheme that expects the controller to send the first message. Option 3 defines a periodic messaging scheme with the QSPS environment, where the interface manager periodically sends a message to a controller at the rate specified by `send_period`. Option 4 defines an event driven messaging scheme that expects the RTDS environment to send a message, consisting

Table 3.1: Description of keywords for interface specification

Keyword	Description
`name`	The name of the control with which the interface manager will interact with.
`network`	Set of key-value pairs to describing the `ip_address` and `port`.
`to_control_signal_names`	Set of key-value pairs used to describe the name of the signals and its corresponding data type that will be sent to the control unit.
`from_control_signal_names`	Set of key-value pairs used to describe the name of the signals and its corresponding data type that will be received from the control unit.
`direction`	Specifies who will communicate first in a request-reply communication pattern, either `ctrl_first` or `sim_first`.
`blocking`	Specifies whether to enable blocking mode during message exchange.
`simulator`	Defines the simulation environment the interface manager will interact with either `qsps` or `rtds`.
`behavior`	Specifies the nature of the simulation environment, this can either be `event` or `periodic`.
`log_enable`	Specifies whether to enable the timestamping of message across the interface manager either `True` or `False`.
`send_period`	Specifies the periodic update rate in nanoseconds with which the IM will send new system measurements and expect a response from the control unit.

of system measurements, first. Option 5 defines an event driven messaging scheme that expects the controller to send the first message. Option 6 defines a periodic messaging scheme with the RTDS environment, where the interface manager periodically sends a message to a controller at the rate specified by `send_period`.

3.3.2 Detailed Interface Descriptions

The following sections describe the message format, communications methods, and a detailed description of the interface with respect to the OSI model for each ICD mentioned in the previous section.

Description for ICD-0L. This interface addresses the communication interaction between a Quasi-Static Power System (QSPS) simulation and a low-fidelity simulation interface manager.

Table 3.2: Acceptable forms of communication behaviors

Option	`sim`	`behavior`	`direction`	`blocking`	`send_period`
1	`qsps`	`event`	`sim_first`	True/False	NA
2	`qsps`	`event`	`ctrl_first`	True/False	NA
3	`qsps`	`periodic`	NA	True/False	Update rate
4	`rtds`	`event`	`sim_first`	True/False	NA
5	`rtds`	`event`	`ctrl_first`	True/False	NA
6	`rtds`	`periodic`	NA	True/False	Update rate

Figure 3.3 shows two systems, the one on the left is the simulation environment on its operation platform crossing ICD-0L to interact with a Low-Fidelity Simulation Interface Manager (SIM) on its operation platform.

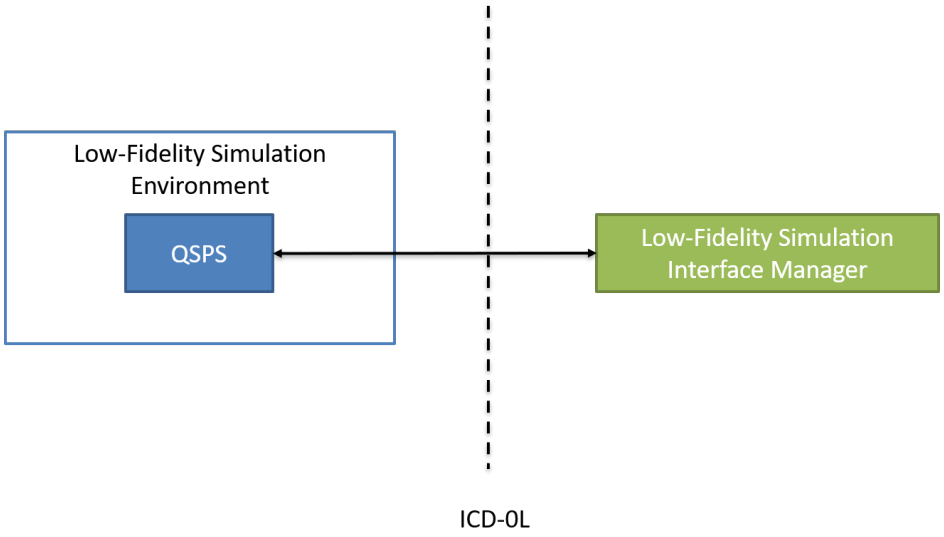

Figure 3.3: Boundary Diagram for ICD-0L

Message Format. The data assembly process abides by the JSON standard [35] to data package for message exchanges between QSPS and its corresponding SIM. For an example, please refer to Appendix B.1. The signal definitions consist of the signal name and data type for each signal in the message. It is specified as a key-value pair under the keywords `to_control_signal_names` for system measurements and `from_control_signal_names` for control inputs.

Communication Methods. The communication interaction between a QSPS and the SIM occurs with User Datagram Protocol (UDP). It begins with a handshake, where the interface manager sends a message to the QSPS that it is ready, then the low-fidelity simulation environment replies with number of timesteps in the simulation to the SIM, more details on the information flow is provided in the next sections. Finally, the interface manger then responds that the simulation may begin. After the handshake, during each simulation iteration the SIM and the QSPS exchange messages as defined by the `behavior` keyword. Table 3.3 provides further detail with respect to the OSI model. Note that the description for each layer is bi-directional, except the application layers for each communication direction. This specific interface uses the User Datagram Protocol to

Table 3.3: Description of each OSI layer for ICD-0L

Layer	**Description**
Physical	Via copper Ethernet cable.
Data Link	Via Ethernet protocol.
Network	Via Internet Protocol (IP) network.
Transport	Via User Datagram Protocol (UDP).
Session	Not applicable.
Presentation	ASCII character encoding.
Application on SIM	See Application Layer - SIM.
Application on QSPS	See Application Layer - QSPS.

transfer messages, the backbone of the infrastructure uses the Internet Protocol. Once the machine hosting the interface manager receives a message, the presentation layer either encode or decodes the message into the ASCII representation. The one distinction is on the Application layer the following two sections provide more detail into each.

Application Layer - SIM. The application layer on the SIM consists of the low-fidelity simulation interface manager providing the functionality to retrieve and set signal data. It defines four methods implemented in the Python programming language:

- `__getitem__`: Takes as input a signal name, used to index the signal's value from a Python dict class. If the dictionary is empty, then it waits to receive a message as a JSON object. If the message is from an expected IP address the JSON object is loaded as a Python dict. Finally, it returns the value corresponding to the signal name provided.
- `__setitem__`: Takes as input a signal name and a value, the new value is loaded as a key-value pair to a Python dictionary. Once all the outbound signals are updated, then the dictionary

is packaged as a JSON object and sent to the interface manager. Both the dictionaries used in the `__getitem__` and `__setitem__` methods are cleared for the next exchange of signal data.

- `check_to_sim`: Takes as input a signal name and data type, this function verifies whether the signal is available by cross-referencing the low-fidelity power system model document.
- `check_from_sim`: Takes as input a signal name and data type, this function verifies whether the signal is available for retrieval by cross-referencing the low-fidelity power system model document.

Application Layer - QSPS. The application layer on the QSPS side provides the functionality to receive and send messages JSON objects. The functions are as follows:

- `recv_json`: Takes no input but waits to receive a message by the specified IP address. After receiving a message the sender's IP is verified, and then proceeds to decode the message into a MATLAB `struct` for use in the QSPS model.
- `send_json`: Takes as input a MATLAB `struct` that is then encoded into the JSON standard with the `jsonencode` function provided by MATLAB.

Description for ICD-0H. This interface specifically describes the interaction between an FPGA, that interfaces with an RTDS rack, and the machine that hosts the interface manager software, as seen Figure 3.4. Note that there exists another interface between the RTDS rack the FPGA, but the description of that interface is not vital for the discussion and it is assumed to function without errors.

Message Format. Messages on this interface will be packed into a binary array, where the data type will be specified by a format string. The byte order of the data is little-endian specified by '<', and the format string can be either a float or an integer of 4-bytes, denoted by 'f' and 'i', respectively. Please refer to Appendix B.2 for an example. The signal definitions consists of the signal name and data type for each signal in the message. It is specified as a key-value pair under the keywords `to_control_signal_names` for system measurements and `from_control_signal_names` for control inputs.

Communication Methods. On the simulation side of ICD-0H, is an RTDS rack that communicate with an FPGA over Aurora protocol. The FPGA then processes the data and communicate with a general-purpose machine via a PCI-Express bus. It is important to note that the location of

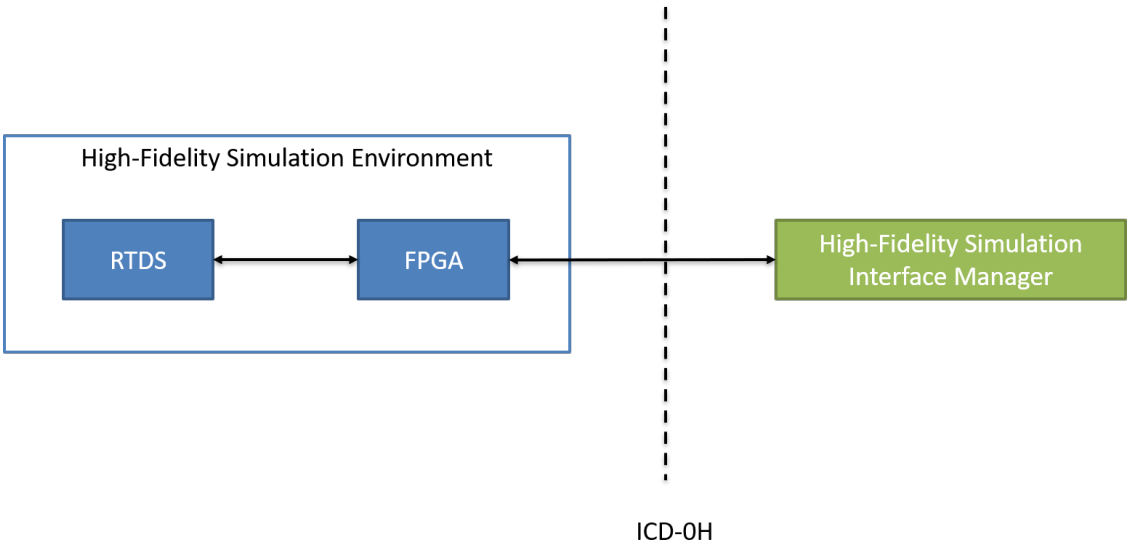

Figure 3.4: Boundary Diagram for ICD-0H

ICD-0H is not the PCIe-bus but the mapped memory space. From here, the binary data is written to the memory address associated with the PCI-Express bus. On the simulation interface manager (SIM) side, exists a software interface that reads from and writes to the memory bus associated with the FPGA that then communicates the information back to the RTDS rack. The RTDS sends the signal data through Aurora protocol, from which the FPGA then processes the data and sends it via PCI-Express to the host machine. As already noted, the specifics of this interface are not key for the discussion and characterization of the interface manager.

When the SIM wants access to the values of a certain signal it reads from its memory address to extract the float or integer value. The SIM on the host machine can also write to the memory address, so that the FPGA can read those values and send them to the RTDS rack. Table 3.4 provides further detail with respect to the OSI model, note that the description for each layer is bi-directional. There is a limitation of 64 signals that the FPGA can be send and receive, imposed by the RTDS simulation environment. It is worth noting that this describes the interface between a single FPGA and a host computer, but that is only limited by the number of PCIe slots available. Multiple FPGAs can interface the with the host computer and for each FPGA they have their own corresponding physical and data link layer. On the application layer, the simulation interface

Table 3.4: Description of each OSI layer for ICD-0H

Layer	Description
Physical	Via PCI-Express.
Data Link	Via PCIe bus standard.
Network	Not applicable.
Transport	Not applicable.
Session	Not applicable.
Presentation	Defined in Message Format
Application	Described below.

manager provides the functionality to retrieve and set generalized signal data. It is an abstraction of multiple FPGAs with four specific methods implemented in the Python programming language:

- `__getitem__`: Takes as input a signal name which maps the name to its corresponding FPGA and to its positional index used to read the allocated memory space on the host computer with respect to the signals' data size. With the signal's corresponding data type, the 4-byte binary value is unpacked and returned.
- `__setitem__`: Takes as input a signal name and a value, similarly to the `__getitem__` function, the signal name corresponds to a FPGA and positional index. The index and signal data type are used to write the new value to the allocated memory space.
- `check_to_sim`: Takes as input a signal name and data type, this function verifies whether the signal is available for updating by cross-referencing the high-fidelity power system model document.
- `check_from_sim`: Takes as input a signal name and data type, this function verifies whether the signal is available for retrieval by cross-referencing the high-fidelity power system model document.

These four methods allows for the interface manager to communicate with an RTDS rack via an FPGA.

Description for ICD-1. This interface describes the interaction between a Simulation Interface Manager and a Control Interface Manager, as seen in Figure 3.5. It is the key interface that maintains the same physical communication point enabling the transition from a low-fidelity environment to a high-fidelity environment.

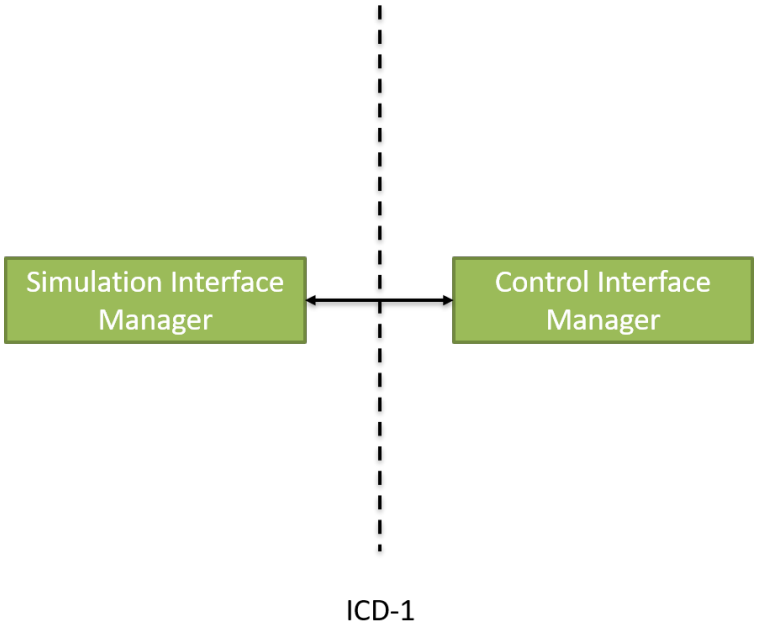

Figure 3.5: Boundary Diagram for ICH-1

Message Format. The message format depends on the active simulation environment, please refer to the Message Format from Section 3.3.2 or 3.3.2. The signal definitions consists of the signal name and data type for each signal in the message. It is specified as a key-value pair under the keywords `to_control_signal_names` for system measurements and `from_control_signal_names` for control inputs.

Communication Methods. Table 3.5 provides further detail with respect to the OSI model, note that the description for each layer is bi-directional. This interface resides entirely in the logical layer

Table 3.5: Description of each OSI layer for ICD-1

Layer	**Description**
Physical	On the same machine.
Data Link	Not applicable.
Network	Not applicable.
Transport	Not applicable.
Session	Not applicable.
Presentation	Not applicable.
Application	Described below.

as a software implemented in Python. The control interface manager (CIM) make use of the four methods available to both high and low fidelity simulation interface managers (SIMs). As described in Section 3.3.2 and Section 3.3.2. During the initialization of the CIM, it utilizes `check_to_sim` and `check_from_sim` to verify that all the pertinent signals are available for communication exchange. Moreover, there are two methods used to exchange data with simulation and controller:

- `send_to_control` This method iterates each signal name directed to the controller and uses the `__getitem__` function from the SIMs to retrieve the most updated value for that signal name. The CIM packages all the signal values with their corresponding data type, for transfer to the controller.
- `send_to_simulation`: This method takes as input a binary array consisting of all the signals directed to the simulation environment, which is unpacked according to the signals data type. The CIM uses the `__setitem__` method from the SIM, to update the simulation with the controller's most recent control inputs.

Description for ICD-2. This section describes the system boundary between a CIM and a single controller as seen in Figure 3.6, although multiple controllers can be instantiated with their respective CIM.

Message Format. This interface abides the same formatting specified in the Message Format of Section 3.3.2. The signal definitions consists of the signal name and data type for each signal in the message. It is specified as a key-value pair under the keywords `to_control_signal_names` for system measurements and `from_control_signal_names` for control inputs.

Communication Methods. The communication between the CIM and the controller occurs with User Datagram Protocol (UDP). The communication behavior can either be event-driven or periodic, when the update rate is specified in the interface specifications file, and must abide by the acceptable communication behaviors as outlined in Table 3.2. Table 3.6 provides further detail with respect to the OSI model. Note that the description for each layer is bi-directional, except the application layers for each communication direction.

Application Layer - CIM. This side of application layer pertains to the CIM functionalities, it has two pertinent methods defined in Section 3.3.2.

- `send_to_control`: After packaging the signal values the data is sent over a UDP socket.
- `send_to_simulation`: After receiving a message from the controller, continues the discussion of `send_to_simulation` in ICD-1.

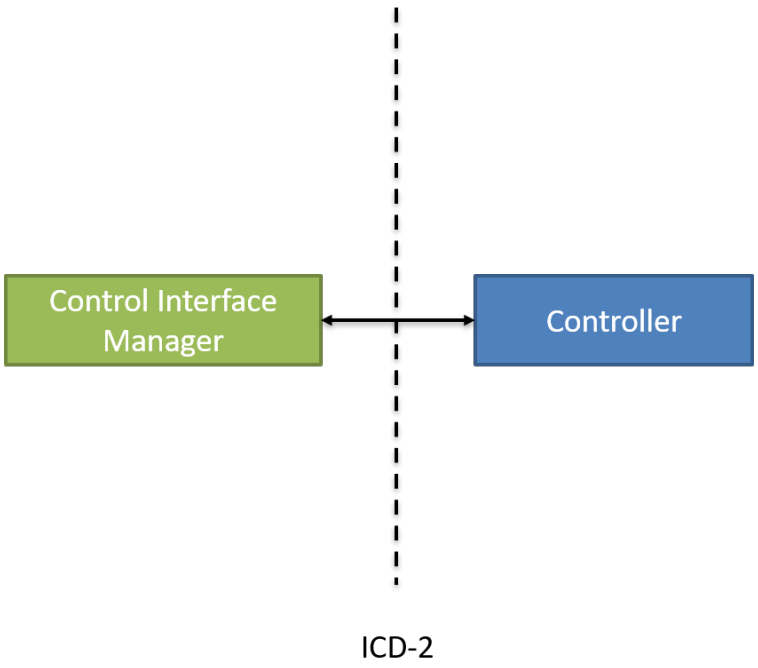

Figure 3.6: Boundary Diagram for ICH-2

Application Layer - CTRL. The application layer on the controller consists of a UDP module that manages the receive and send functionality. The following functions describes the UDP module on the controller side:

- `__getitem__`: Takes as input a key corresponding to a signal name as defined in the interface specification file. It then retrieves the value assign the key via a Python `dict` that store the data for a signal exchange.
- `__setitem__`: Takes as input a key and a value, to update the dictionary storing the signal data with the new value.
- `update_values`: This function performs the signal exchange functionality. Internally, it send and receives new signal data or it waits receive and then send a new update, depending on the communication behavior.
- `clear_cache`: This function clears the dictionary storing the signal data to prepare for the next signal exchange.

Table 3.6: Description of each OSI layer for ICD-2

Layer	Description
Physical	Via copper Ethernet cable.
Data Link	Via Ethernet protocol.
Network	Via Internet Protocol (IP) network.
Transport	Via User Datagram Protocol (UDP).
Session	Not applicable.
Presentation	Defined in Message Format.
Application on CIM	See Application Layer - CIM.
Application on CTRL	See Application Layer - CTRL.

3.4 Interface Setup and Model Transition

After defining each key interface and detailing the interface specifications, one can begin to take a further look at how the entire system works together and explain how to transition from a low-fidelity environment to high-fidelity environment with this interface manager. The following section explains the interface manager flow diagram and the interface setup with respect to both simulation environments and one controller.

3.4.1 Flow Diagram

Figure 3.7 illustrates the flow diagram of how the interface manager sets up either a periodic or event-driven communication. The process begins at the left most side where either a RTDS IM or QSPS IM take as input a RSCAD draft file or Quasi-Static table, respectively. Here RSCAD [3] is the graphical user interface used to interact with the RTDS simulation environment. Then, the interface manager selects the simulation IM corresponding to the `simulator` keyword. This simulation IM, provided to the control IM so as to provide it with the necessary functionalities to communicate with the simulation environment. From here, the communication behavior defined in the interface specification is selected with their corresponding parameters.

3.4.2 Low-Fidelity Setup

Figure 3.8 shows the system setup when interfaced with a low-fidelity simulation environment, specifically the QSPS environment. The QSPS communicate to the interface manager via its UDP module. On the interface manager side, the Control Interface Manager interacts with the QSPS Interface Manager, the SIM specific to the QSPS environment, to exchange the measurement signals

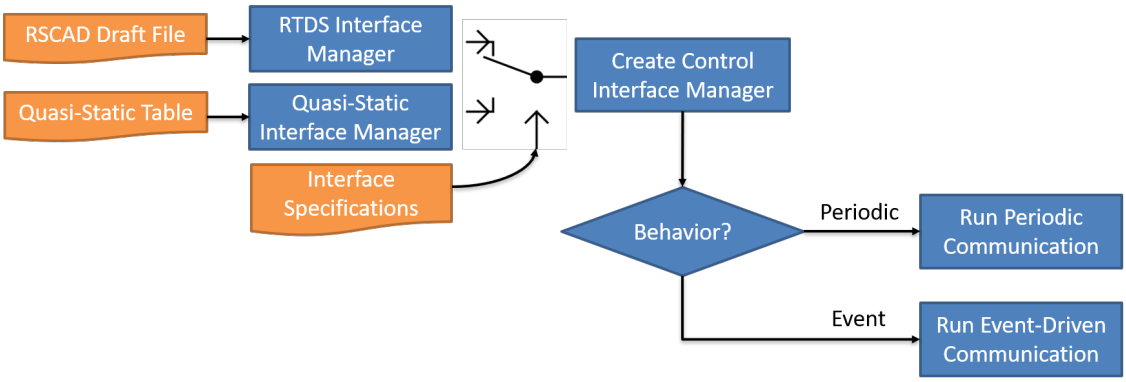

Figure 3.7: Interface Manager flow diagram

with the appropriate controller. From the controller side, it utilizes a UDP module to interact with the interface manager. Here the power system model file corresponds to the Quasi-Static Table described in Section A.2.

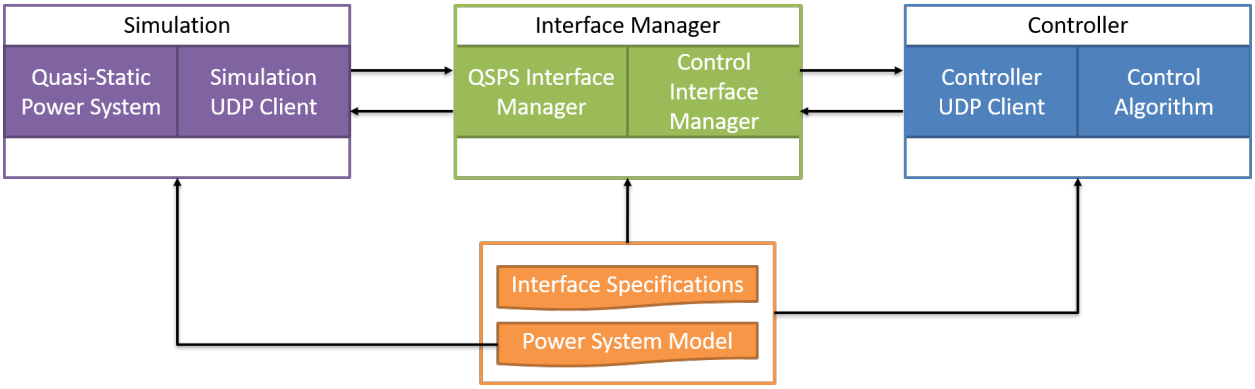

Figure 3.8: Setup for a low-fidelity simulation

Figure 3.9 describes the flow of information between a controller and the Quasi-Static Power System simulation environment. During the initialization, both controller and simulation wait to receive a message from the interface manager specifying that it is ready to mediate the signal exchange, denoted by `Interface Ready`. The simulator then packages any local information pertinent to the controller and/or interface manager into its reply and sends it to the interface manger denoted as `Simulator Ready`. Parallelly, the controller packages any local information that may be pertinent to the simulation and/or interface manager and sends it out as well, denoted as `Control Ready`.

Once, the interface manager receives both ready messages it sends out another message denoted as `Start` to both simulation and controller, so that they begin simulating and controlling, respectively. The handshake between the interface manager and simulation is necessary because of the non-real-time nature of the QSPS, if not setup properly the simulator could start without the controller being ready. After both systems handshake and initialize, the main signal exchange loop begins. The event driven QSPS simulation, first calculates the power flow, then sends the signal data to the interface manager, which sends the signals to the controller. At this point, the controller performs its necessary calculations and then replies with the control actuations. Finally, after receiving the control inputs the QSPS proceeds to update its power system components for the next timestep. This process occurs for the number of timesteps specified by the simulation environment.

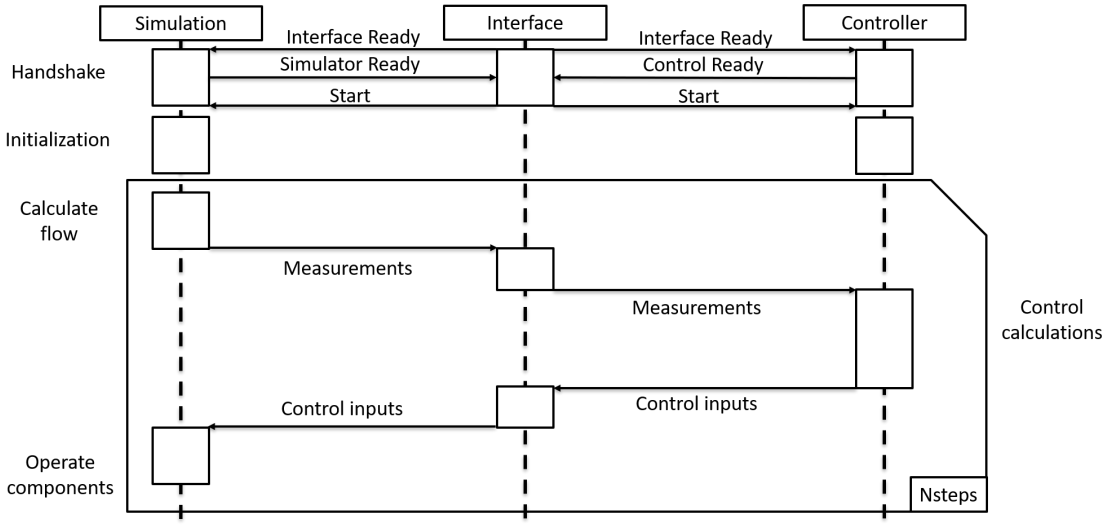

Figure 3.9: Sequence diagram of the information flow from a quasi-static simulation and a controller

3.4.3 Interface Transition

By simply changing the values of the interface specification file, the interface can change the flow of information from a QSPS environment to a RTDS environment. This is accomplished by setting `simulator` from `qsps` to `rtds`. The communication behavior must abide by the supported options defined in Table 3.2. Figure 3.7 illustrates how the adaptable signal interface transitions models from one to another.

3.4.4 High-Fidelity Setup

Figure 3.10 shows how the system looks when it is connected to a high-fidelity environment such as RTDS. On the left hand side is the RTDS operation hardware, the FPGA used to interact with the simulation hardware and the interface manager. On the right hand side is the controller and its UDP module on its operational hardware. In the center is a RTDS Interface Manager, the specific

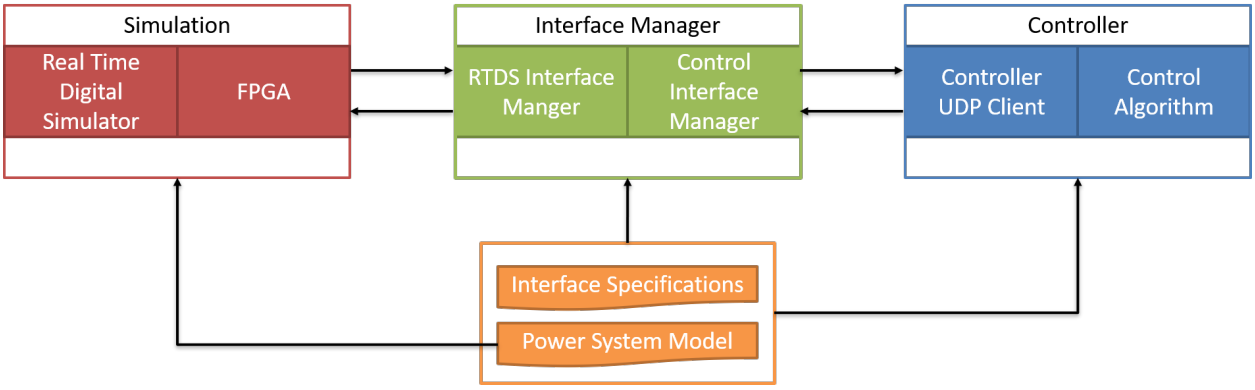

Figure 3.10: Setup for a high-fidelity simulation

SIM for a RTDS environment, that communicate with a controller via its CIM. In this case, the power system model pertains to an RSCAD draft file that defines the entire power system model.

Figure 3.11 shows a sequence diagram of controller communicating via the proposed interface manager. One difference with Figure 3.9 is that this is a control initiated event-driven communication. Another difference is that the life lines depicted as the boxes with a black border extends for the entire duration, unlike in Figure 3.9, this to exemplify that both the simulation and controller now operate in real-time. This message scheme requires that the controller first send a message to request a reply from the interface. After the interface manager software is running on the host machine it begins waits until it receives a message from a controller. Once the first message is received the communication exchange begins, and the interface writes the control inputs to the mapped memory space as described in Section 3.3.2. Each time the interface manager receives a message, it reads the most recent values in the mapped memory space and then sends it to the controller. This happens until the controller no longer requests for new system measurements.

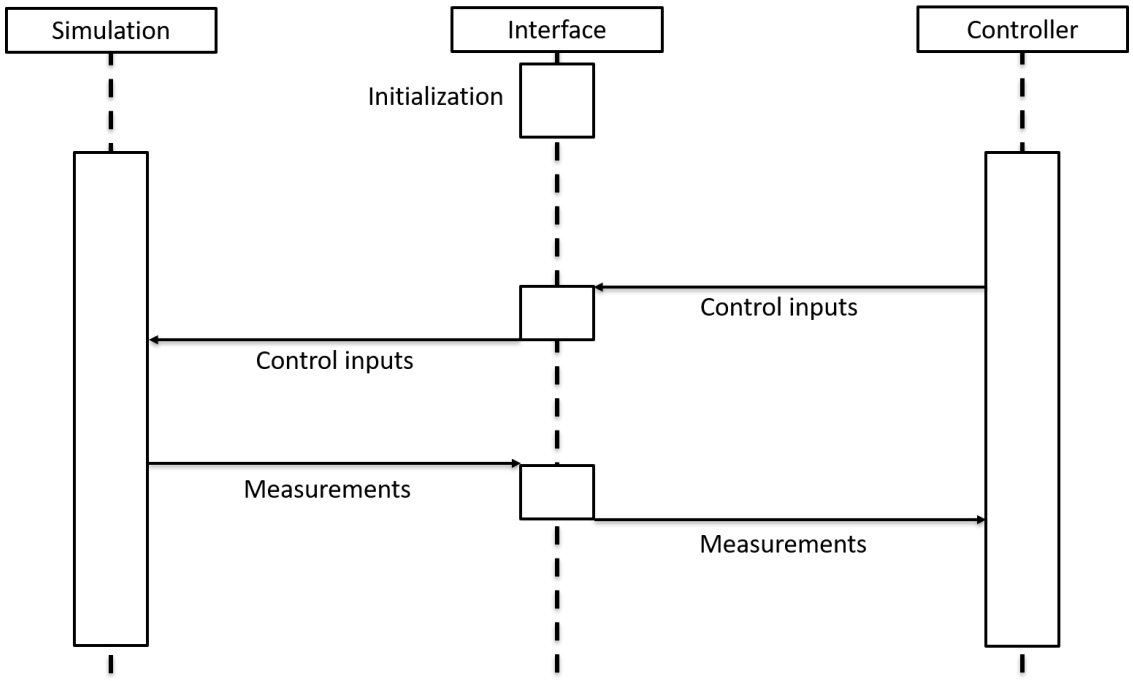

Figure 3.11: Sequence diagram of the information flow from a real-time simulation and a controller

3.5 Interface Performance

The next step consists of characterizing the performance of the interface design with respect to time. The following section consists of an explanation of the selected communication point to timestamp the flow of signal data. Then, a breakdown of the metrics used to assess the elapsed times of characteristics events of the proposed interface is provided. Finally, experimental results validate the performance of the interface implementation.

3.5.1 Message Time Stamping

Figure 3.12 denotes all possible locations where a message may be timestamped. It presents the ideal scenario of gathering data from multiple distributed resources. It consists of one log file for each controller, interface, and simulation environment. On the left hand side, are the two simulation environments each with their own logging capabilities to log the signal traffic. At the center, is the interface manager with logging functionalities on both the simulation and controller sides. Finally, on the right hand side, are the controllers each with their own respective timestamping modules.

Where each log entry will contain (a) Time stamp, (b) Sender address, (c) Receiver address and (d) Message. The timestamp adheres to the ISO 8601 standard for the representation of time with the following format `YYYY-MM-DD HH:MM:SS.fffffffff`. Where the 9 `f`'s denote the sub second portion of time. It is important to note that in order to timestamp messages the keyword `log_enable` must be set to `True`.

To a certain extent having multiple timestamped log files provides a better way to probe if the interface implementation is functioning correctly. Now, if the host machines for the simulation, interface manager, and controller are distributed, it becomes a little more difficult. Distributed resources may not always have the same time as another machine, thus un-synchronized clocks make it difficult to reason when each message was truly sent and received. Synchronization of

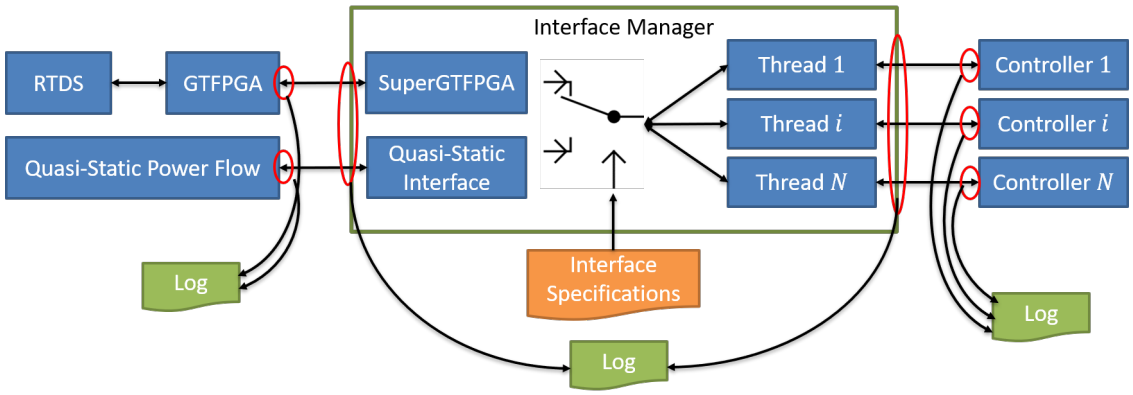

Figure 3.12: Interface manager denoting possible locations of time stamping.

the distributed resources is possible with the IEEE 1588 Precision Time Protocol (PTP). But, the synchronization of distributed resources is out of the scope of this thesis and can be left for future works. Because the analysis is with respect to the correctness of the interface manager, it is sufficient to only timestamp the messages at the simulation to interface manager boundary (ICD-0) and the interface manager to controller boundary (ICD-2).

Consider Figure 3.13, where the specific instances in time are defined with respect to ICD-0 and ICD-2. Where,

- $t_{out}^{sim}[k]$: the k^{th} instance of time when a message is created by simulation
- $t_{in}^{sim}[k]$: : the k^{th} instance of time when a message is received by simulation

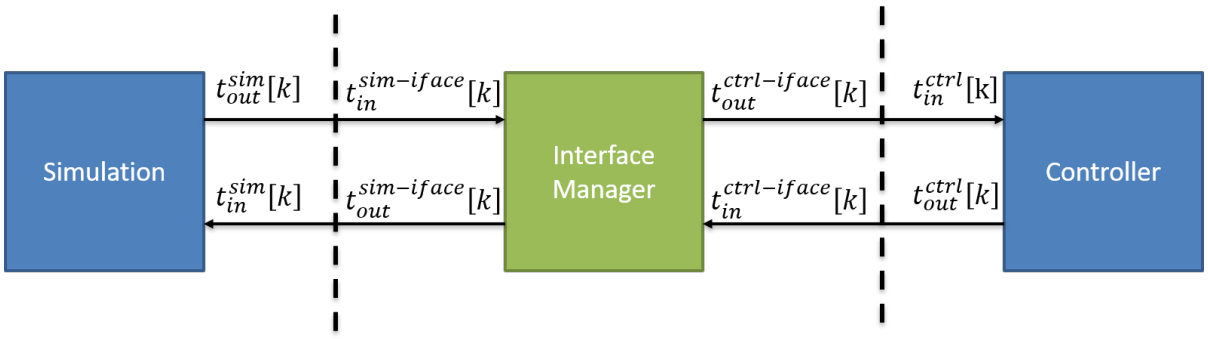

Figure 3.13: Interface manager defining the specific time instances for timestamping functionalities.

- $t_{in}^{sim-iface}[k]$: the k^{th} instance of time when a message from simulation reaches interface
- $t_{out}^{sim-iface}[k]$: the k^{th} instance of time when a message is sent from interface to the simulation
- $t_{in}^{ctrl-iface}[k]$: the k^{th} instance of time when a message is sent from interface to controller
- $t_{out}^{ctrl-iface}[k]$: the k^{th} instance of time when a message from controller reaches interface
- $t_{in}^{ctrl}[k]$: the k^{th} instance of time when a message is received by controller
- $t_{out}^{ctrl}[k]$: the k^{th} instance of time when a message is created by controller

These time instances enable the formulation of metrics to evaluate the interface implementation with respect to time.

3.5.2 Interface Performance Metrics

During the assessment of the interface manager, three types of characteristic events are of interest. They are as follows:

- Response Time
- Round-Trip Time
- Periodic Update Rate

There are two separate metrics for the response time, one for the information flow from the simulation to controller and controller to simulation. Here in (3.1), $\Delta t^{SCRT}[k]$ denotes, simulation to controller response time (SCRT), the elapsed time from when a message enters the interface

manager from the simulation side and when the message is sent to the controller, for the k^{th} signal exchange. And in (3.2), $\Delta t^{CSRT}[k]$ denotes, controller to simulation response time (CSRT), the elapsed time from when a message enters the interface manager from the controller side and when the message is sent to the simulation, for the k^{th} signal exchange.

$$\Delta t^{SCRT}[k] = t_{out}^{ctrl-iface}[k] - t_{in}^{sim-iface}[k] \tag{3.1}$$

$$\Delta t^{CSRT}[k] = t_{out}^{sim-iface}[k] - t_{in}^{ctrl-iface}[k] \tag{3.2}$$

The round trip time can be further broken down into two different elapsed times, namely, Simulation Round-Trip Time (SRTT) and Controller Round-Trip Time (CRTT). In (3.3), $\Delta t^{SRRT}[k]$ is the elapsed time from when the interface manager receives a message from simulation and when the interface manager sends the control inputs, after the controller processes the received data, for the k^{th} signal exchange. In (3.4), $\Delta t^{CRRT}[k]$ is the elapsed time from when the interface manager receives a message from the controller and when the interface manager sends a message back to the controller, after simulating the next timestep, for the k^{th} signal exchange.

$$\Delta t^{SRRT}[k] = t_{in}^{sim-iface}[k] - t_{out}^{sim-iface}[k] \tag{3.3}$$

$$\Delta t^{CRRT}[k] = t_{out}^{ctrl-iface}[k] - t_{in}^{ctrl-iface}[k-1] \tag{3.4}$$

Finally, in (3.5), $\Delta t^{RTPU}[k]$ denotes the elapsed time from the previous instance when the interface manager sent signal data to the controller, effectively timing how often the interface manager sends signal data to the controller.

$$\Delta t^{RTPU}[k] = t_{out}^{ctrl-iface}[k] - t_{out}^{ctrl-iface}[k-1] \tag{3.5}$$

The following section analyzes the experimental results of the five metrics.

3.5.3 Interface Validation

This section consists of experiments to characterize the performance of the interface with respect to the round trip time, response time, and update rate of the interface manager as defined by the metrics. The interface performance is evaluated with a quasi-static simulation of the notional MVDC SPS, with the communication behavior as specified by Option 1 from Table 3.2.

Response Time. Figure 3.14 shows the distribution of the response time of the interface manager when exchanging a message in both directions. Essentially capturing the elapsed time for the interface manager to receive, process, and send to a simulation or controller. The response time when sending a message to a controller is on average 750 μs and as for the response time when sending to the simulation it is on average 325 μs. There is clear difference of approximately 425 μs, likely due to the design choice of utilizing JSON encoding as opposed to the formatted binary array, specified in Sections 3.3.2 and 3.3.2, respectively.

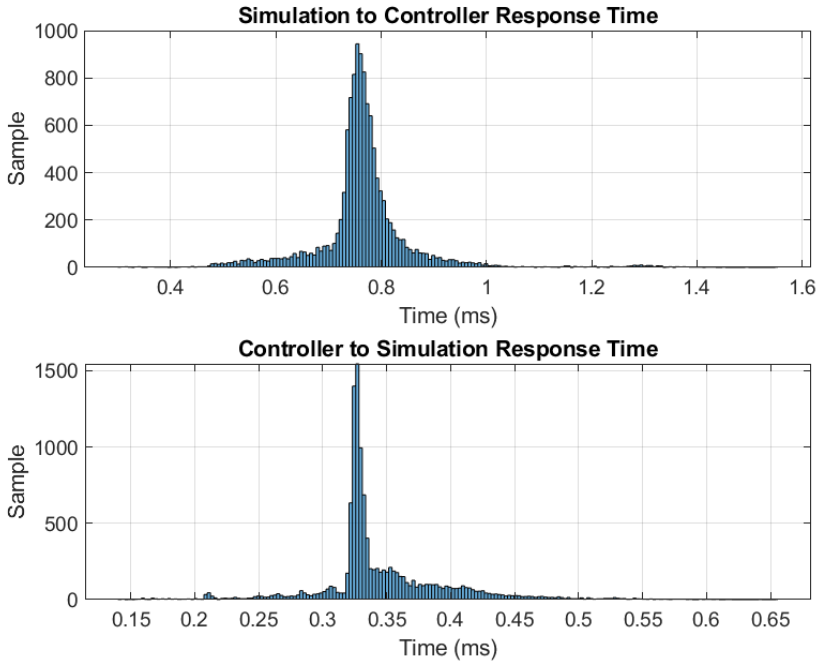

Figure 3.14: Response time for both communication directions.

Round-Trip Time. Figure 3.15 presents the distribution of experimental results for the round trip time of both simulator and controller. One can clearly see that there is a stark difference in round trip time between the two systems. On average, the elapsed time from when the interface manager receives a message from the simulation to when it returns a reply from the controller is around 1.45 ms. The controller round trip time is considerably longer with an average of around 260 ms, as it includes the solution time of the QSPS model and both response times.

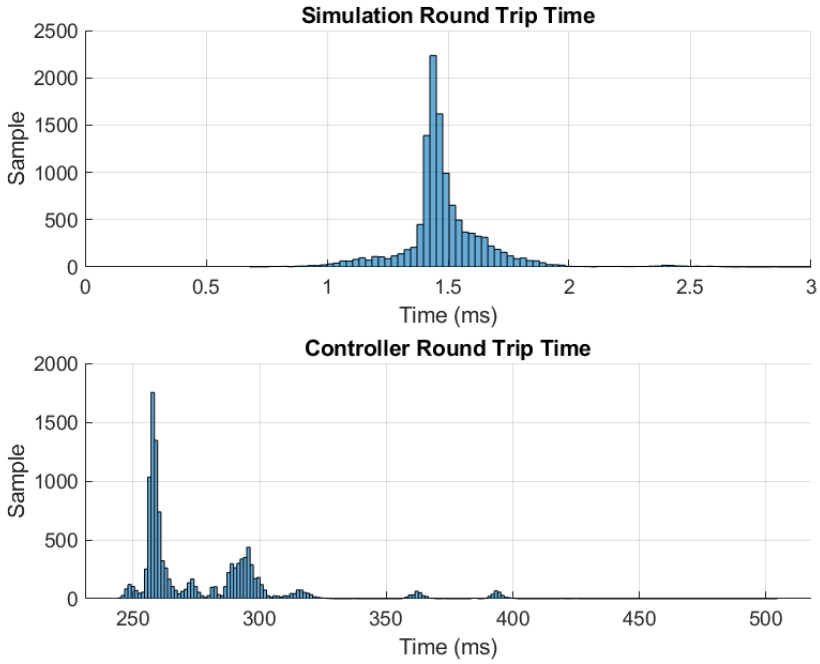

Figure 3.15: Round trip time as experienced by the simulation and controller.

Update Rate. Figure 3.16 shows the distribution of the update rate as seen from the controller, when interfaced with a QSPS simulating the notional four-zone MVDC SPS. In other words, how often it takes for the interface to send new system measurements. There is a large degree of variability due to the operational platform used to simulate the QSPS model. This however can be reduced by utilizing a high performance operational platform and a more efficient programming language, such as C++. It is apparent that the simulation, on average, took 260 ms to perform its calculations and for the interface manager to send the data to the controller. For example, if a controller specified a periodic update rate of 300 ms this setup would be unacceptable.

3.6 Conclusion

In conclusion, the adaptable signal interface implementation allows for the transition of power system models of varying fidelity. The interface specification file encapsulates the necessary details for the interface manager to correctly mediate the signal traffic. Both interface setups are presented one corresponding to a QSPS model and another for a RTDS model. The performance of the

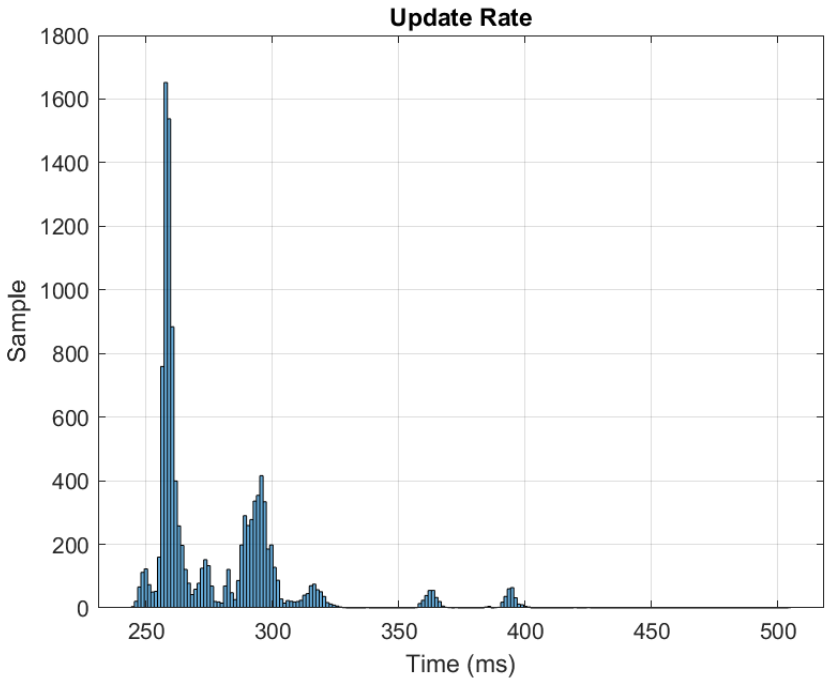

Figure 3.16: Update rate when interfaced with a QSPS simulating the notional four-zone MVDC SPS.

interface manager is assessed in terms of its response time, the simulation and controller round trip time, and finally its update rate. The most important detail of this interface is that is has very low overhead as shown in Figure 3.14, which is necessary for real-time communication.

CHAPTER 4

FAULT MANAGEMENT DEVELOPMENT PROCESS

During the early stages of development of a new fault management algorithm, the main objective is to characterize the algorithm's implementation in a relevant environment to validate and verify the system requirements. Attempting to integrate a new algorithm in a high-fidelity environment in a single integration process may seem like the most straightforward manner of approaching the problem. But that is not entirely the case, Tahan and Ben-Asher model and analyze a single integration process and an incremental integration process in [36]. Here they show that as the complexity of a system increases the probability of meeting a deadline decreases with a single integration process. On the other hand, for complex systems incremental integration offers a higher likelihood of meeting deadlines.

During the rapid prototyping process, to increase the likelihood of testing a fault management system in a timely fashion, the thesis leans on the concepts of requirement evolution [37]. Figure 4.1 captures the essence of requirement evolution; the process begins at the user needs which are converted into a tightly coupled set of initial requirements R_1, denoted by the solid line. Domain D_1, consisting of an environment and controller, is loosely coupled to the user needs, denoted by the dashed line, because regardless of the domain a user desires needs meet. Now that is not the case for the requirements and domains, there is tight coupling between these two components because the requirements must be tested, verified, and validated with respect to the domain in question to reflect the user's needs. As the process develops so does the knowledge of the environment, to the point that the domain is then refined, denoted by the arrows, to D_2 to capture more details to better approximate a more relevant environment. Domain refinements must be monotonic in nature, so as to ensure that the updated version still contains key characteristics apparent in the previous model. Requirements R_1 are tested against this new domain from which the requirements are furthered refined to R_2. These iterative steps can occur multiple times throughout the development process. Finally, the system specifications are the set of requirements that will satisfy the user's needs.

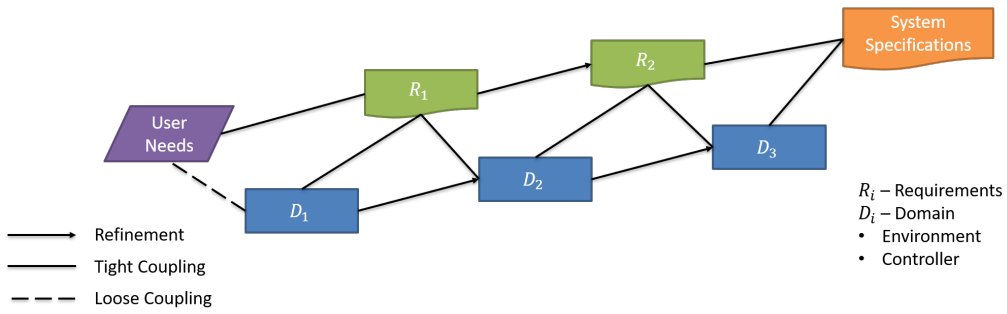

Figure 4.1: Requirement Evolution Process.

The following sections present a requirement analysis for a fault management system, an explanation of the requirement evolution process with respect to a fault management implementation, and notional MVDC shipboard power system. Then, a description into the process of measuring domain equivalence to ensure that the key characteristics are present in both domains. Finally, experimental results show the effectiveness of the interface design and development process.

4.1 Requirements Analysis: Fault Management

The main objective of a FM is to monitor the current operating status of the electrical power systems, identify a fault at an early stage, and take appropriate actions to minimize the effects of the fault to enable reliable, safe, and robust restoration [30]. The specific Fault Management implementation developed at CAPS is explained in the following papers [5, 31]. The FM was integrated with a medium-voltage dc shipboard power system (SPS) that uses disconnect switches for electrical isolation. A key caveat of disconnect switches is that before they can be opened, they must first be de-energized as to mitigate electrical arcing from occurring. The following section outlines the functional, performance, and interface requirements, as well as the constraints for such a system.

4.1.1 Functional Requirements

The fault management system at its core must provide the following functionalities to mitigate the propagation of electrical faults.

- The fault management system shall detect and locate a rail-to-rail short circuit fault.

- The fault management system shall isolate fault from rest of power system after detection.
- The fault management system shall reconfigure the power system to route power from operational generator to critical loads.
- The fault management system shall recover power to the shipboard power system after reconfiguration.

It must provide the means by which it can detect and locate a fault, isolate a fault, reconfigure, and then recover the power system. Those are the high-level requirements, but further specification is necessary to accurately define what those requirement are with respect to a breakerless MVDC SPS.

Derived Functional Requirements. The derived functional requirements breakdown the high-level requirements into more tangible ones that can be measured. The following list details the derived functional requirements for a fault management system:

- The fault management system shall receive current flows to determine where and when a fault occurs.
- The fault management system shall de-energize the sections of the shipboard power system needed for isolation.
- The fault management system shall open the correct electrical switches to isolate fault.
- The fault management system shall actuate electrical switch to establish a power flow to critical loads.
- The fault management system shall re-energize the SPS after reconfiguration to restore power.

By breaking down the high-level requirements into low-level ones, it becomes easier to reason about how to measure the characteristic events in order to validate and verify these functional requirements. For the detection to work properly it requires that the FM receive the proper flow variables to determine if a fault exists and where it's located. During the isolate sequence, with respect to a breakerless MVDC SPS, it is first necessary to de-enerize the power system. Then, one must specify that in order to isolate the correct switches must be opened. Following the isolation of a fault, FM must then determine the best set of switches to actuate to restore the flow of power to critical loads. Finally, to recover the power system, it must re-energize the buses so that power can flow again to the loads.

4.1.2 Performance Requirements

The control evaluation does not stop at its functional requirements. In order to provide these function in a timely fashion to mitigate the propagation of fault currents to the rest of the system, it must perform at a minimum the following performance requirements.

- The fault management system shall recover the power system in less than 8 ms.
- The fault management system shall have an upper bound on the time to compute the switches to isolate.
- The fault management system shall have an upper-bound on the run time to detect and locate a fault.
- The fault management system shall have an upper-bound on the run time to isolate a fault.
- The fault management system shall have an upper-bound on the run time to reconfigure. and recover the shipboard power system.
- The fault management system shall have an upper-bound on the run time to recover the shipboard power system.

Part of the evaluation process is measuring the worst-case performance to determine what other forms of mitigation is necessary until the system is fully recovered. Thus, it is required that there exists an upper bound on each of the characteristic events. The 8 ms is chosen to be comparable to an AC protection system that can isolate a fault at the zero crossing point.

4.1.3 Interface Requirements

As for the interface requirements, this section further clarifies the nature of the interface. This section outlines the semantics of the information that is transferred by the interface.

- The fault management system shall acquire a sequence of system measurements consisting of current flow, power levels for generators and loads, and state of electrical switches.
- The fault management system shall output switch states during isolation and reconfiguration.
- The fault management system shall output a signal to de-energize or re-energize a rectifier unit.
- The fault management system shall handle 32-bit floating point values and integers with little-endian byte ordering.

4.2 Fault Management Requirement Evolution

A key portion of the thesis consists of the verification of requirements for a fault management system in environments of monotonically increasing relevance. The idea is to incrementally integrate components instead of performing a single-stage integration, which is known to increase the time spent on the development process [14]. Zowghi and Gervasi [37] explain their view on evolutionary incorporation of requirements during the development process in their paper "The Three Cs of Requirements: Consistency, Completeness, and Correctness." Figure 4.2 encapsulates the concept with respect to the fault management development process. There are two set of models, namely the

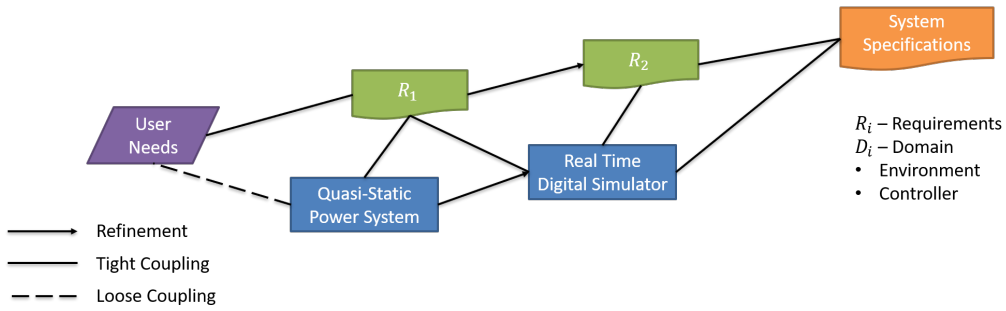

Figure 4.2: Requirement Evolution for Fault Management.

low-fidelity and high-fidelity models, with their corresponding requirements. Both models represent the same power system characteristics, such as power flow, switch state, and power generation, the key differences lies in the granularity of the data. The low-fidelity model is a quasi-static power system simulation, and the high-fidelity model is a real-time power system simulation. Due to the quasi-static nature of the low-fidelity power system certain functionalities cannot be verified in that domain level, such as the fault detection and localization module of the fault management algorithm because of the lack of transient information needed during power system fault analysis.

The arrows express a monotonic relationship i.e., having the property of never decreasing. With respect to the power system models, the characteristics that are present in the low-fidelity model must also be present in the high-fidelity model but with a greater level of detail. With respect to the requirements, the high-fidelity requirements must contain at least the low-fidelity requirements. It is important to note that middle line between low-fidelity requirements and the high-fidelity model, expresses that there is also a tight coupling between them, and thus the low-fidelity requirements

must be verified in the high-fidelity model. Implying that the low-fidelity requirements are carried over throughout the evolutionary process to the high-fidelity requirements. Finally, the process arrives at the system specifications, which are the final documented set of requirements to be satisfied in the most relevant environment. The Fault Management requirements are defined in Section 4.1. The following sections provide a brief description of what the domain consists of and what requirements can be verified at the given domain level.

4.2.1 Low-Fidelity Model and Requirements

This section describes the low-fidelity model and the requirements verifiable at this level of fidelity. The low-fidelity model consists of the four-zone MVDC notional shipboard power system derived in the quasi-static power system environment described in Section 2.3.1. It is simulated as a discrete event simulation by specifying Option 1 in Table 3.2. It is assumed that the power system is in a split plant alignment, where both the starboard and port distribution networks are separated from each other. The low-fidelity requirements are as follows:

- The fault management system shall de-energize the sections of the shipboard power system needed for isolation.
- The fault management system shall open the correct electrical switches to isolate a fault.
- The fault management system shall actuate electrical switch to establish a power flow to critical loads.
- The fault management system shall re-energize the SPS after reconfiguration.
- The fault management system shall acquire a sequence of signal measurements consisting of current flow, power levels for generators and loads, and state of electrical switches.
- The fault management system shall output switch states during isolation and reconfiguration.
- The fault management system shall output a toggle signal to de-energize or re-energize a rectifier unit.

The outlined requirements are verifiable in a QSPS model of the four-zone MVDC SPS, because it consists of functional and interface requirements. Wall-clock time is not considered in this work, therefore it is not a good model to use to take into account timed events.

4.2.2 High-Fidelity Model and Requirements

The high-fidelity model consists of the four-zone MVDC notional shipboard power system derived in the real-time digital simulation environment as described in Section 2.3.1. It is simulated as a real-time simulation with an event-driven message pattern, where the controller initiates the communication, as specified in Option 4 from Table 3.2. Similar to the low-fidelity model, it is assumed that the power system is in a split plant alignment, where both the starboard and port distribution networks are separated from each other. The requirements consist of the low-fidelity requirements and the following requirements:

- The fault management system shall satisfy the low-fidelity requirements.
- The fault management system shall detect and determine the location of a rail-to-rail short circuit fault.
- The fault management system shall recover the power system in less than 8 milliseconds.
- The fault management system shall have an upper-bound on the runtime to detect and locate a fault.
- The fault management system shall have an upper-bound on the runtime to isolate a fault.
- The fault management system shall have an upper-bound on the runtime to reconfigure. and recover the shipboard power system.
- The fault management system shall have an upper-bound on the runtime to recover the shipboard power system.

The first requirement is implied by the concept of Requirement Evolution where each new step in the evolution process has its previous step as a subset. Therefore, it must be able to satisfy the low-fidelity requirements. The second requirement needs high-order dynamic to accurately capture a fault event, which is avialable in a RTDS simulation. Finally, the last requirement considers time in terms of the algorithm implementation, where it is required to restore the power system in less than 8 ms.

4.3 Monotonic Domain Refinements

Each model contains a specific set of power system characteristics, that must be carried over when transitioning from one domain to another. The key power system characteristics are as follows:

- Power flow.
- Power generation.
- Load demand.
- Switch states.
- Load criticality.
- Connectivity.

To compare equivalence of each power system characteristic for each model, a set of metrics are developed. They consist of comparing each component, for each power system characteristic, in the low-fidelity model to its corresponding component in the high-fidelity model. Let k denote the timestep of the simulation, then the following conditions must hold for each timestep:

$$|f_i^{qsps}[k] - f_i^{rtds}[k]| < \delta, \forall_i \in \mathcal{E} \tag{4.1}$$

$$|s_i^{qsps}[k] - s_i^{rtds}[k]| < \delta, \forall i \in \mathcal{S} \tag{4.2}$$

$$|G_j^{qsps}[k] - G_j^{rtds}[k]| < \delta, \forall j \in \mathcal{G} \tag{4.3}$$

$$|L_j^{qsps}[k] - L_j^{rtds}[k]| < \delta, \forall_j \in \mathcal{L} \tag{4.4}$$

Where, f_i is the power flow for the i^{th} edge, s_i is the state for the i^{th} switch, G_j is the power generated for the j^{th} generator, L_j is the power delivered for the j^{th} load, $\mathcal{L}$ is the set of all the switches, $\mathcal{E}$ is the set of all the edges, $\mathcal{G}$ is the set off all the generators, $\mathcal{L}$ is the set of all the loads, and finally, *qsps* and *rtds* denote that the signal belongs to a QSPS and RTDS simulation, respectively.

That is to say that for each timestep the error between the system measurements, the power flow of each branch (4.1), the state of each switch (4.2), the power generated from each generator (4.3) and the power demanded from each load (4.4), of both a QSPS and an RTDS simulation must be bounded by some value. Considering that both models are running on entirely different operational platforms and have different methods of calculating the flows in the network, it is expected that the error will not be zero. But, if the error between the values is too large, then going from a QSPS simulation to a RTDS simulation is not monotonic with respect to the level of detail. Thus, it does not make sense to develop on this platform at an early stage. The comparison process first begins by assessing the connectivity i.e., the network topology; this comparison is rather straightforward when the connectivity is stored in a graph. From a graph structure a matrix representation namely the adjacency matrix is generated from both models. Figure 4.3 presents the sparsity pattern of the adjacency matrix for the four-zone MVDC SPS. Assuming that both adjacency matrices have

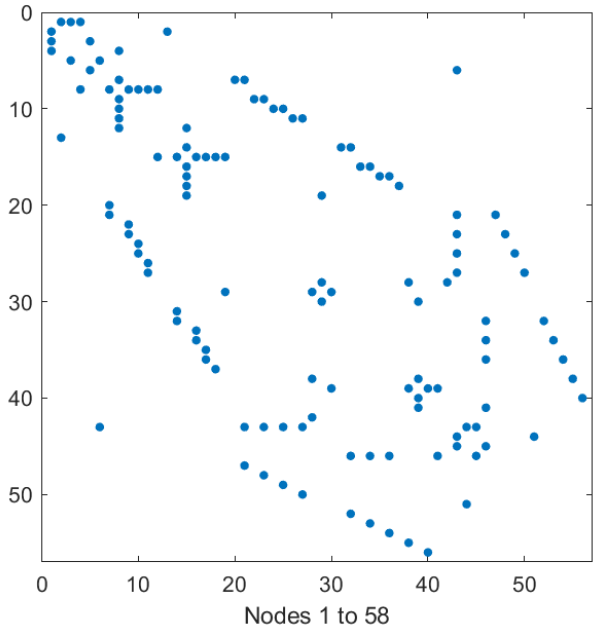

Figure 4.3: Adjacency Matrix for four-zone MVDC SPS.

the same order with respect to vertices and the graph is undirected, then the following equation expresses the equivalence in terms of connectivity.

$$||A^{qsps} - A^{rtds}||_{1,1} = \sum_{i=1}^{N}\sum_{j=1}^{N}|a_{ij}^{qsps} - a_{ij}^{rtds}| \leq \delta \tag{4.5}$$

Where N is the number of buses in the power system. Both the QSPS and RTDS have identical sparsity patterns, therefore, with respect to (4.5), δ is equal to zero.

After assessing the network connectivity, one can begin to determine the equivalence of the remaining power system characteristics. Because said characteristics depend heavily on the load demand at each time instance, the same load profile is fed to both models. A relatively light load profile is selected to avoid the quasi-static power system simulation from inherently curtailing load delivery. To then corroborate that the power system truthfully represents the switch states, after 12 iterations into the simulation, the plant switches its alignment from split plant to parallel plant. With respect to Figure 2.8, the middle switch, normally open, denoted by the black box, is closed for the parallel lines, not including the bow and stern connections of the SPS.

At this point there are two data sets one for the quasi-static power system simulation and one for the real-time simulation environment. But due to rich signal granularity available in the real-time simulation environment the signal data will have to be made comparable to the quasi-static simulation. This process consists of applying a low-pass filter at a frequency of 0.1% of the sampling frequency of the real-time environment. Once the high order dynamics is filtered out, the signal data is downsampled to match the same number of samples as in the quasi-static simulation. After

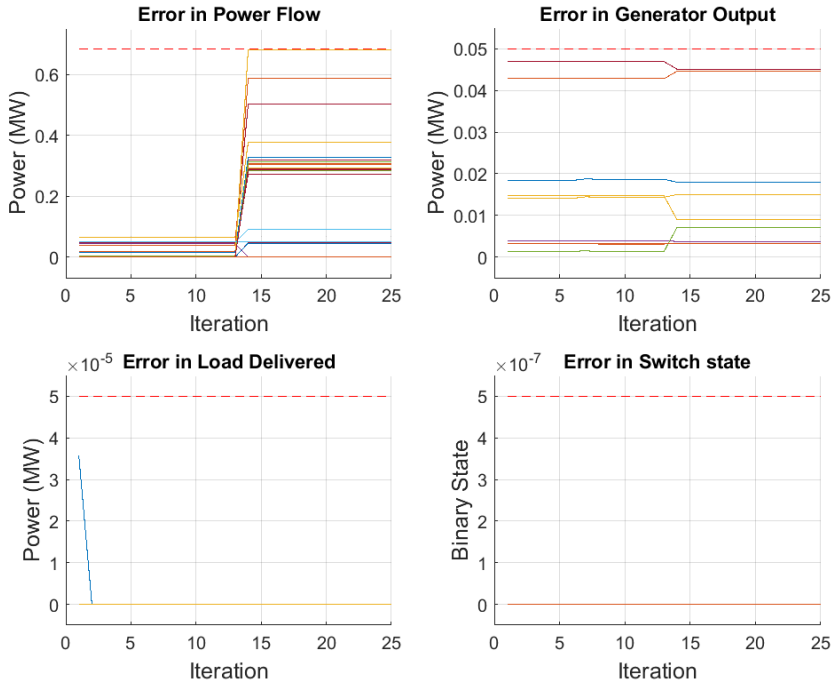

Figure 4.4: Absolute error between each time-varying power system characteristic.

processing the data, one can now apply the metrics to ascertain the equivalence of the power system characteristics between both models. Figure 4.4 shows the absolute error between both models for each time-varying power system characteristics i.e., power flow, power generation, power delivery, and switch state. The red dashed line represents the value of the δ parameter used. For the power flow, $\delta_{flow} = 0.684$ MW, for generator output it is $\delta_{gen} = 0.05$ MW, for the load delivered is $\delta_{load} = 5 \times 10^{-5}$ MW, and finally for the binary switch state it is $\delta_{switch} = 1 \times 10^{-6}$. It is clear that both the load demand and switch state comparison are equivalent to a high extent. But the power flow and power generation have a higher margin of error, that is because each simulation has a

different type of solver. Not to mention each have a separate model representation, in the case of the QSPS it does not consider the system impedance which plays a role in the power flow calculations. Regardless of the error margin, there is a sufficient degree of information that is retained between both models so that a controller can initially begin with a quasi-static representation of the four-zone MVDC SPS and then transition to a real-time representation.

4.4 Fault Management Metrics

To properly validate and verify the fault management requirements it is necessary to develop another set of metrics. There are five key characteristic events involved in the evaluation of the fault management algorithm. They are the (1) time to compute the isolation actions, (2) the time to apply the isolation sequence, (3) the time to compute the reconfiguration solution, (4) the time to apply the reconfiguration, and finally (5) the time to recover the power system. Of those five events, the time to apply the isolation (2), the time to reconfigure (4), and the time to recover the power system (5), need metrics to assess the performance of the implementation. For each simulation environment there is a slight variation in the metrics implementation, due to their respective notion of time.

4.4.1 Isolation Metric

The intent of the isolation metric is to capture the elapsed time from when a fault occurs in the power system, to the time instance when all the necessary switches are opened to isolate the fault. This is accomplished by means of a pre-determined table that contains the names of all the switches needed to isolate for each fault location in the shipboard power system. Using the table the state of each switch is checked, until they are actuate to their necessary position. For a discrete event simulation, the following mathematical statement encapsulates the metric

$$k_i^{isolation} = \max\{k : s_{ij}(k) \ \forall \ j \text{ is open}\} \tag{4.6}$$

$$\Delta k_i^{isolate} = k_i^{isolation} - k_i^{fault} \tag{4.7}$$

Where k represents the k^{th} signal exchange, i is the i^{th} fault location and j is the j^{th} nearest electrical switch connected to the fault location. Here, k_i^{fault} is the k^{th} signal exchange when the i^{th} fault occurs, $s_{ij}(k)$ is the state of the j^{th} electrical switch at fault location i, and $k_i^{isolation}$ is

number of iteration corresponding to the last electrical switch to open. Thus, $\Delta k_i^{isolate}$ defines the number of iterations to isolate a fault. In the case of a real-time simulation, the metric measures elapsed wall-clock time instead of signal exchanges, as defined in the following expression.

$$t_i^{isolation} = \max\{t : s_{ij}(t) \ \forall \ j \text{ is open}\} \tag{4.8}$$

$$\Delta t_i^{isolate} = t_i^{isolation} - t_i^{fault} \tag{4.9}$$

The real-time metrics are similar to those in the previous expression, where t is the wall-clock time.

4.4.2 Reconfiguration Metric

This metric captures the elapsed time from after the isolation of a fault to when the system is reconfigured. Specifically, in the scope of this thesis, a power system is considered reconfigured when the fault management algorithm sends the command re-energize the network. Therefore, the metric checks when the algorithm sends the command to re-energize and assigns that as the reconfiguration instance. With respect to a discrete event simulation, the following expression defines the reconfiguration metric,

$$k_i^{reconfiguration} = \max\{k : G_{ij}(k) \ \forall \ j \text{ is commanded to re-energize}\} \tag{4.10}$$

$$\Delta k_i^{reconfigure} = k_i^{reconfiguration} - k_i^{isolation} \tag{4.11}$$

Here, G_{ij} is the j^{th} generator connected to the i^{th} fault location at iteration k when the generator is commanded to re-energize, $k_i^{reconfiguration}$ is number of iteration for the last generator to be commanded to re-energize, and $\Delta k_i^{reconfigure}$ is elapsed number of iteration to reconfigure the system. As for the real-time simulation metrics, the following measures the elapsed time to reconfigure a power system.

$$t_i^{reconfiguration} = \max\{t : G_{ij}(k) \ \forall \ j \text{ is commanded to re-energize}\} \tag{4.12}$$

$$\Delta t_i^{reconfigure} = t_i^{reconfiguration} - t_i^{isolation} \tag{4.13}$$

Similarly, one must consider the metrics where t is the wall-clock time.

4.4.3 Recovery Metric

This metric captures the elapsed time after sending the re-energization command to the instance when the bus voltage reaches acceptable thresholds. This is done by checking the voltage value until

it passes 97% of its nominal voltage. As expressed in the following equations,

$$k_i^{recovery} = \max\{k : V_{ij}(k) \ \forall \ j \text{ is } \geq 97\% \text{ of nominal voltage}\} \tag{4.14}$$

$$\Delta k_i^{recovery} = k_i^{recovery} - k_i^{reconfiguration} \tag{4.15}$$

Where, $k_i^{recovery}$ expresses the k^{th} iteration when the last bus voltage goes above 97% of the nominal voltage rating corresponding to the i^{th} fault location and $V_{ij}(k)$ defines voltage level at iteration k for the j^{th} bus voltage. Here, $\Delta k_i^{recovery}$ expresses the number of iterations to recover the power system. During a quasi-static discrete event simulation, this is not possible due to the fact that the low-fidelity model does not consider voltage levels, but instead power levels. Moreover, the quasi-static power system does not contain dynamics, therefore, as soon as the command to re-energize is sent, it is applied. Therefore, $\Delta k_i^{recovery}$ for a quasi-static discrete event simulation is zero. The following expression defines the recovery metric, in the case for real-time simulation,

$$t_i^{recovery} = \max\{t : V_{ij}(t) \ \forall \ j \text{ is } \geq 97\% \text{ of nominal voltage}\} \tag{4.16}$$

$$\Delta t_i^{recovery} = t_i^{recovery} - t_i^{reconfiguration} \tag{4.17}$$

Similarly, one must replace k with t and consider the elapsed time and time instances, instead of number of iterations.

4.5 Experimental Results

4.5.1 Fault Management with a Quasi-Static MVDC SPS

This section covers the validation and verification of the low-fidelity requirements outlined previously. The subset of requirements mainly consists of functional requirements that are measurable with the metrics defined in the previous section. Figure 4.5 shows 1041 samples of the distribution of the number of signal exchanges necessary to properly isolate a fault, reconfigure the power system, and then recover the loads. THe CEF is leveraged to automate the generation of the experimental samples used in these results. During the isolation sequence, the concentration of samples is centered at 5 signal exchanges. This signal exchange consists of a full communication interaction as specified in Figure 3.9. Three of the signal exchanges consists of the protection logic in the FM, as specified in the Section 2.4.1. Of the two remaining signal exchanges, one is for FM to send the command to isolate and the other is for the control input to be applied to the power system. From this one can

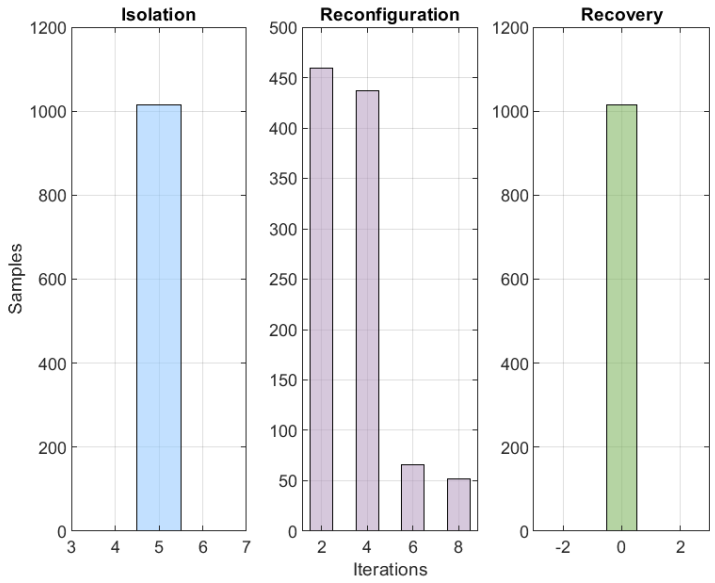

Figure 4.5: Distribution of the number of signal exchanges to recover the power system after a fault.

conclude that low-fidelity functional requirements consisting of the de-energization capabilities of FM, opening of the correct switches to isolate a fault, the actuation of switches to establish a power flow to critical loads, and re-energization functionality are all validated and verified experimentally. One can also conclude that the interface requirements are satisfied as well consisting of the signal inputs to FM, the desired switch states, and toggle signal to control the set point of the rectifier unit.

4.5.2 Model Transition

By virtue of the adaptable signal interface, changing the interface specification file allows FM to be evaluated in the power system model of high-fidelity. The following list describes the modification made to the keywords:

- `simulator`: Changed to its default value of `rtds` denoting the Real Time Digital Simulator environment.
- `behavior`: Kept as `event` to specify an event-driven messaging scheme.
- `blocking`: Disabled to reduce potential latency for real-time communication.

- `direction`: Changed to a control initiated communication, as defined by `ctrl_first`.

These setting consists of Option 4 from Table 3.2. Please refer to Appendix C to see the interface specifications used.

4.5.3 Fault Management with a Real-Time MVDC SPS

This section covers the validation and verification of the high-fidelity requirements as defined in Section 4.2.2. This set of requirements consists of the low-fidelity requirements, as well as the performance requirements. Figure 4.6 shows 391 samples each characteristic events in a fault recovery sequence as previously outlined in Section 4.4. The samples are of the FM response to each fault location is automated by leveraging the Control Evaluation Framework (CEF), more information can be found [6, 7, 8, 9, 10, 11]. As opposed to the previous section these set of results consider wall-clock time, therefore it includes the time for the algorithm to calculate its isolate and reconfiguration sequence. One can clearly see that it takes less than 13 ms to compute the desired switch states to isolate, thereby validating the second high-fidelity requirement. The time to isolate the fault takes less than 52 ms because of the de-energization configurations of the rectifier units in the real-time MVDC SPS model. From here one can see that the time to calculate the switch states to reconfigure the system takes less than 15 ms. Depending on the specific fault location it is sometimes required for the opposite side of the SPS to de-energize in order to properly reconfigure it, thus one can see a upper-bound on the runtime of 40 ms. Finally, during the recovery of the power system it takes less than 27 ms to re-energize the bus voltage so that power flow is restored to the loads. Hence, one can conclude that the low-fidelity requirements are validated and verified in RTDS environment. The upper bound performance requirements are also verified, resulting in a worst-case run time of around 147 ms to fully restore power to critical loads. Therefore, demonstrating that the current implementation cannot meet the 8 ms requirement to recover the power system. The breakdown of the characteristic events gives some insight into where improvements can be made for the algorithm and model. For example, after detecting a fault, in parallel, one module can handle the isolation while another handles the reconfiguration. Thus, thereby reducing the time to reconfigure because the de-energization command can be sent sooner, in case that both the starboard and port distribution network need to be de-energized. On the other hand, these results show that in order to meet the 8 ms requirement either a different protection device that can both

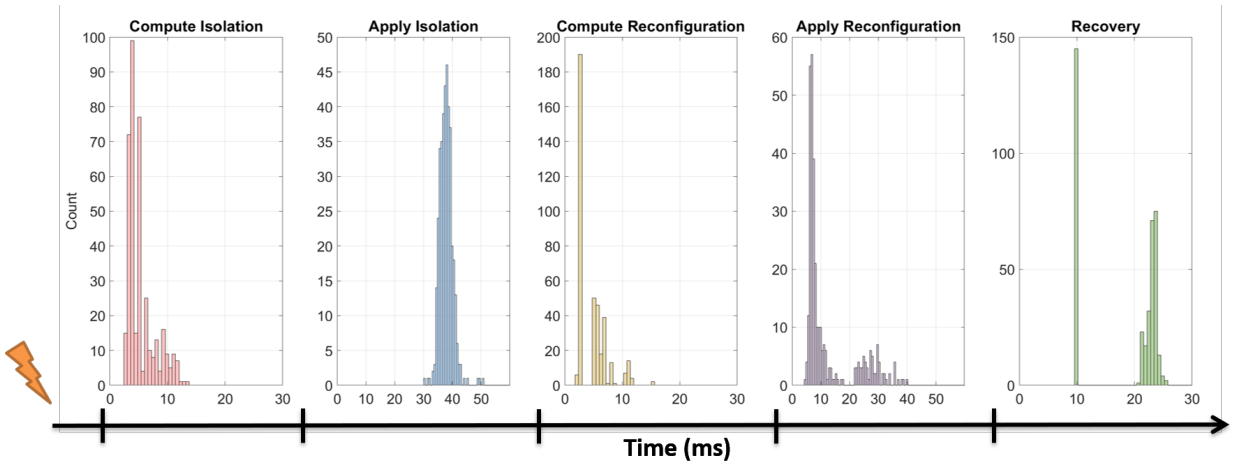

Figure 4.6: Distribution of the elapsed times for each of the characteristic events in a fault management recovery sequence with the interface manager.

interrupt and isolate current must be used. Or the configuration of the rectifiers must be modified to allow for faster response time so that the disconnect switches may be actuated quicker. These results show that the interface manager is capable of transitioning from a low-fidelity model to one of high-fidelity. To ensure that the performance of this setup the current results are compared to an implementation of the fault management setup without the interface manager.

4.5.4 Interface Validation

This section compares the experimental results of the FM case study with and without an interface manager. Figure 4.7 presents 7890 experiments consisting of the characteristic times for a fault management recovery sequence. The upper-bound on the runtimes for each characteristic event are as follows:

- Time to compute isolation: 12 ms
- Time to apply isolation: 25 ms
- Time to compute reconfiguration: 5 ms
- Time to apply reconfiguration: 20 ms
- Time to recover: 25 ms

The resulting elapsed times equate to a worst-case run time of around 87 ms to fully restore power to critical loads. One can clearly see that there is a stark difference between the 87 ms upper-bound without the interface and the 147 ms upper-bound with the interface. This stresses the fact that there is still more work required to achieve the desired performance of the interface manager. Taking

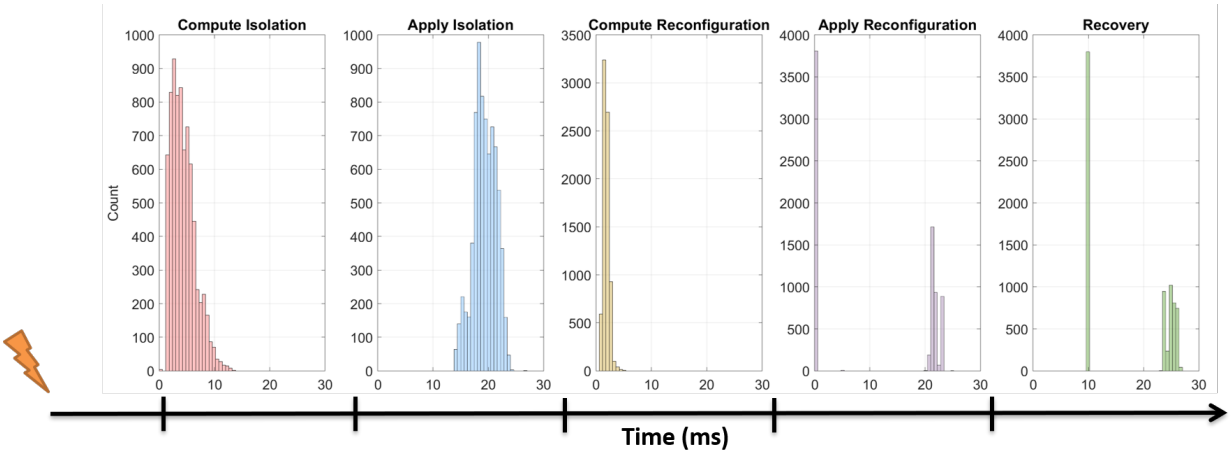

Figure 4.7: Distribution of the elapsed times for each of the characteristic events in a fault management recovery sequence with the interface manager.

a closer look at the comparison of each event, one can see that for the isolation calculation both distribution are similar other than a slight delay seen at the beginning of the compute isolation event in Figure 4.5. There is clear discrepancy between the apply isolation event for both figures, as a 20 ms delay is illustrated in Figure 4.5. The compute reconfiguration event in Figure 4.7 is equivalent to that of the experiments with the interface manager. Although, the apply reconfiguration event is shifted by around 10 ms, as instead in a peaks at 1 ms and 20 ms, Figure 4.5 presents peaks in 10 ms and 30 ms. Finally, both figures emphasize that the recovery event is equivalent in both experimental setups.

4.6 Conclusion

This chapter shows the effectiveness of utilizing an adaptable signal interface to ease the development process of a fault management implementation on a breakerless four-zone MVDC shipboard power system. It begins by outlining the functional, performance, and interface requirements for a fault management approach. The development process with respect to the requirement evolution

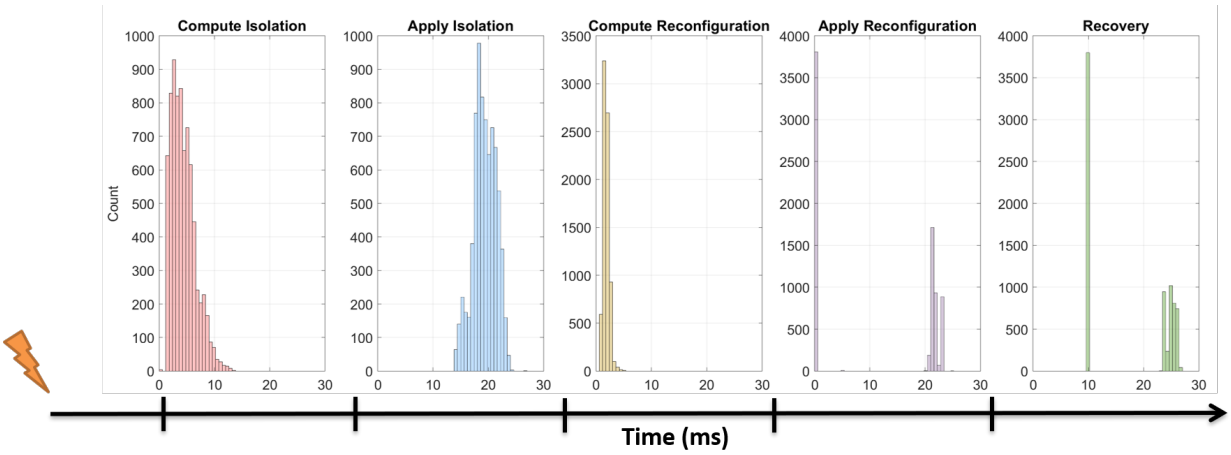

Figure 4.8: Distribution of the elapsed times for each of the characteristic events in a fault management recovery sequence without the interface manager.

concept is explained, and the requirement verifiable at each model fidelity is defined. The power system characteristics of both models are assess to determine their equivalence, before transitioning to the high-fidelity model to ensure that it contains a sufficient degree information. The metrics needed for the quantification the functional requirements are stated. And finally, experimental results are shown where a FM implementation interfaces with a MVDC SPS model in QSPS environment to verify its low-fidelity requirements. Then, with the adaptable signal interface the FM implementation can interface with a MVDC SPS model in a RTDS environment to verifiy the rest of its requirements. The results of the FM case study are compared to with an implementation with the proposed interface manager to determine its effectiveness.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Conclusion

In conclusion, it is apparent that an adaptable signal interface proves beneficial during the development process of a fault management algorithm for a notional four-zone MVDC shipboard power system. Starting the development process in a quasi-static discrete event simulation of the MVDC SPS provides insight into the functional abilities of the system-level control implementation. After a developer is sufficiently confident in the abilities of their control implementation, the model fidelity is increased by virtue of the proposed interface because of its modifiable interface specifications and common communication point. Finalizing the development process in a high-fidelity real-time environment allows for further insight into the timing performance of the controller.

As a direct result of the work presented in this thesis, a control developer can validate algorithms in increasing levels of environment relevance without changing the interface on the controller. Thus, enabling a process to de-risk the control development cycle by integrating system-level controls with models of incrementally refined fidelity. Such a de-risking process reduces the man-hours needed to develop a system-level control and can reduce the overall cost of the endeavour. Finally, this process allows the control implementation to go up in technology readiness levels, while reducing the time to field deployment.

5.2 Future Works

This thesis presented the requirements and design criteria for an adaptable signal interface for use with a quasi-static discrete event power flow simulation and a high-fidelity real-time power system simulation. However, there are a few improvements that can be made to the development process, interface, algorithm, and modeling implementations. Beginning with the interface improvements, this work is specifically focused on a fault management controller, but it can be extended to function with different controllers, thereby, generalizing the development process. Testing the interface with distributed controller is another avenue to explore the capabilities of the interface manager. A key

feature of the interface is its ability to perform handshakes, this allows for information to be sent to all communicating entities i.e., simulator, controller, and interface manager. In this case, the number of interactions and fault location are sent to the controller during its handshake with the QSPS simulation. But it is not limited to that, the graph connectivity can also be sent to the algorithm so as to use it in its calculation instead of necessitating previous knowledge of the power system connectivity. The QSPS model can be improved by modifying it for real-time implementation. This can be achieved by running the QSPS model on a high performance computer and setting an internal timer to wait until the next event to continue execution. To aid in the development process, other models of higher fidelity can also be created to further aid in its incremental steps.

APPENDIX A

QUASI-STATIC POWER SYSTEM

A.1 Quasi-Static Approximation

The low-fidelity model is a quasi-static approximation of the power flows in a power system. A quasi-static approximation is commonly understood as equations that keep a static form, i.e. do not involve time derivatives, although some values are permitted to change slowly with respect to time.

Typically, a power flow study consists of a numerical analysis of the following key characteristics of an AC power system namely voltages, voltage angles, real power, and reactive power. The power system is assumed to be in steady-state operation, and it depends highly on the admittance values of power system components such as cables, generators, and loads. The real and active power balance equations for each bus are given by, (A.1) & (A.2).

$$0 = -P_i + \sum_{k=1}^{N} |V_i||V_k|(G_{ik}cos(\theta_{ik}) + B_{ik}sin(\theta_{ik})) \tag{A.1}$$

$$0 = -Q_i + \sum_{k=1}^{N} |V_i||V_k|(G_{ik}sin(\theta_{ik}) - B_{ik}cos(\theta_{ik})) \tag{A.2}$$

Where,

- N is the number of buses in the system.
- P_i & Q_i is the net active power and the net reactive power, respectively, injected at bus i.
- G_{ik} is the real part of the element in the bus admittance matrix Y_BUS corresponding to the i^{th} row and k^{th} column.
- B_{ik} is the imaginary part of the element in the Y_{BUS} corresponding to the i^{th} row and k^{th} column.
- θ_{ik} is the difference in voltage angle between the i^{th} and k^{th} buses.

When transient information is needed to characterize a power system the generator swing equation is introduced. This equation expresses the relationship between time-varying angular position of the electrical machine's rotor, mechanical torque, and the electrical torque, given by

$$J\frac{d^2\theta}{dt^2} = T_m - T_e = T_{net} \tag{A.3}$$

Where the J is the moment of inertia of the rotor mass in $kg - m^2$, θ_m is the angular position of the rotor with respect to a stationary frame in radians, T_m is the mechanical torque supplied by the prime mover in $N - m$, T_e is the electrical torque output in $N - m$, and T_{net} is the net torque in $N - m$. Given the angular velocity of the rotor, ω, and the electrical torque, T_e, the active power injection at bus i, ignoring stator losses, is given by

$$P_i = \omega\ T_e \tag{A.4}$$

This section briefly scratches the surface of the level detail that can be incorporated into a power system simulation. But, for the early development of distribution network reconfiguration algorithms, this level of detail is not entirely necessary. Of course, once key functionalities of the control algorithm are validated and verified it is imperative to assess the performance of said functionalities in an environment with a high level of detail.

As previously discussed, a quasi-static formulation of the power flow problem relaxes the number of parameters that are needed to perform a power system simulation. The mathematical representation of the power flow problem is a simplification of the typical power flow equations, expressed as:

$$0 = -P_i + \sum_{k=1}^{N} P_{ik}^f \tag{A.5}$$

Here, N is the number of buses in the system, P_i is the active power injection to the i^{th} bus, and P_{ik}^f is the flow of power from the i^{th} bus to the k^{th} bus. Due to the simplified model, voltages and voltage angles are not studied and thus reactive power cannot be properly represented in this quasi-static formulation.

As a direct result of a quasi-static power flow approximation, there are restriction on the type of events that can be accurately represented. Figure A.1 shows the course classification of different type of power system events with respect to their associated time scales.

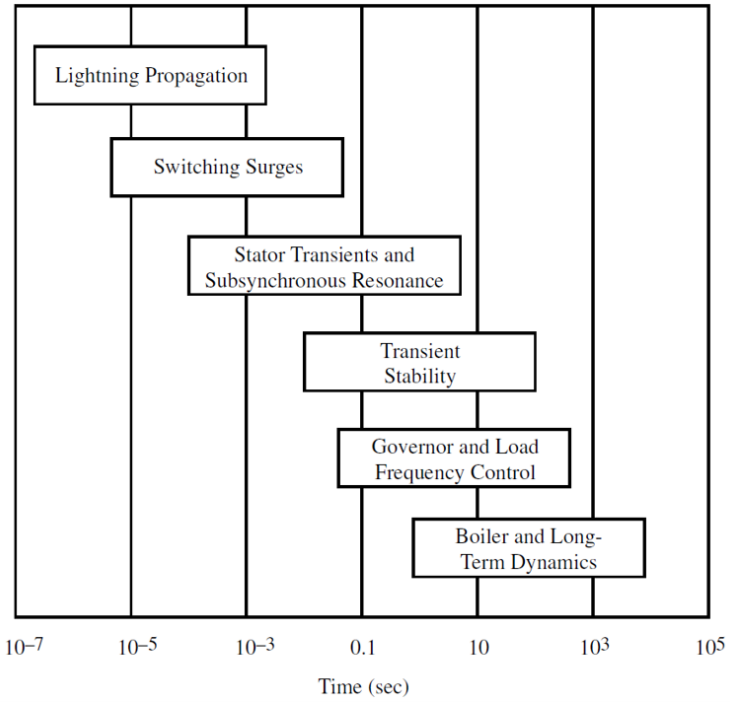

Figure A.1: Time scale for power system events [38]

The dynamics can range from microseconds to hours and as notion time changes for each phenomenon, so does the mathematical modeling needed to accurately represent it. The level of detail and information needed depends entirely on the phenomena, say we are interested in modeling how the unit commitment problem will evolve for the following day including generator transients would far exceed the level of detail needed to understand the phenomena of interest.

Specifically, if there is an interest to analyze faults in the system the quasi-static power flow model is not sufficient as it does not capture the necessary information to accurately represent that phenomena. However, this model does capture the connectivity of a power system network and the flow of power, which in the early stages of designing a distribution network reconfiguration algorithm is exactly what is needed.

A.2 Quasi-Static Power System Software Tools

The groundwork for this early design stage tool is defined in [25], to serve the purpose as an early design tool for reconfiguration algorithms called for modifications. This section describes the

key tools and modification implemented in the MATLAB 2020a computing environment, which are as follows:

- A class to perform an event-driven power flow simulation by a set of power flow solutions.
- A communication module to interface externally with different controllers.
- Classes for both an event-based and time-based electrical switch.

Power Flow Solution with Linear Programming. As aforementioned in Figure 2.6, the process begins by generating a PowerFlowTopology class. This class provides a means of representing a power system as a directed graph and solving for the power flow given a demand profile and generation limits. Again, in this formulation sources represent generation units, sinks represent loads, and edges represent power transfer components, such as cables, switches, and power converters. The constructor for this class takes as an input specifying the topology of the power system. The topology can either be a table of edge information or a string specifying the path of the file containing the following information for each edge

- **id**: Provides a unique string identifier for each edge.
- **description**: Provides a brief description of each edge.
- **type**: Specifies the edge type, which one of the following,
  - **source** denotes a power source.
  - **sink** denotes a power sink.
  - **transfer** denotes a unidirectional power transfer component e.g. power converter.
  - **transfer2** denotes a bi-directional power transfer component.
- **rating**: Specifies the power rating of each edge.
- **node1**: Specifies a string identifier for the node from which an edge emanates.
- **node2**: Specifies a string identifier for the node from which an edge terminates.
- **weight**: Specifies a numerical value that represent the priority of each edge.

In addition to base information provided in the original implementation of the quasi-static power flow [25], the following information provides a simple manner to capture the information necessary to interface with an external controller.

- **monitor**: Denotes a JSON object that specifies a list of string identifiers of information that will be sent from the simulation to an external controller.
- **control**: Denotes a JSON object that specifies a list of string identifiers of information that will be sent from external controller to the simulation.
- **parameters**: Denotes a JSON object that specifies a set of key-value pairs that may be used as initial conditions for the power system component.

Table A.1 shows an example for a simple system. It is important to note that the Quasi-Static power system simulation environment interprets signals with a "#" character in either to monitor or control column, as binary signals. For example, looking at the first row with id "C_21_22", the monitored signal with a "#" is "Dswt#2" that means that the value for that signal is the binary value at position 2, assuming that the first bit begins at position 0. The information in Table A.1,

Table A.1: Topology table for an example system

id	description	type	rating	node1	node2	weight	monitor	control	parameters
C_21_22	Cable from BN21 to BN2	transfer2	50	BN21	BN22	1	["C_21_22pYI", "Dswt#2"]	["Dswtext#2"]	{"switch_init": 1}
C_22_32	Cable from BN22 to BN32	transfer2	50	BN22	BN32	1	["C_22_32pYI", "Dswt#1"]	["Dswtext#1"]	{"switch_init": 1}
C_32_21	Cable from BN21 to BN32	transfer2	50	BN32	BN21	1	["C_32_21pYI", "Dswt#0"]	["Dswtext#0"]	{"switch_init": 1}
G1	Generator 1	source	40	BG1	BN22	1	["G1YP", "G1YVdc"]	[]	{}
G2	Generator 2	source	40	BG2	BN32	1	["G2YP", "G2YVdc"]	[]	{}
LP1	Load P1	sink	90	BN21	BLP1	1	["LP1YP"]	[]	{}

may be implemented in a spreadsheet format, the path of the spreadsheet file is then fed to the constructor of the PowerFlowTopology class. The class constructor generates a table denoted as E, which contains the same information as the topology table, but it adds a corresponding edge for edges of type 'transfer2' to allow the bi-directional nature of the flow. Said class contains the following methods:

- **maxFlow()**: This method solves for the maximum weighted flow of each edge given a certain loading conditions, power generation and transfer capacities. For bi-directional edges, the net flow must be calculated before returning the result of the power flows. This method uses the function *grMaxFlowsMultiW()* a modified version of *grMaxFlows()* provided in the graph toolbox [39], which uses *linprog()* from the MATLAB optimization toolbox.
- **setComponentValues(components,values,key)**: This method updates values specified by the key parameter for a set of *components*.

Time-Based Quasi-Static Power Flow Simulation. The *TimeSeriesPowerFlow* class is used to perform a time-based quasi-static simulation using a sequence of power flow solutions provided in the *PowerFlowTopology* class. Together with the updating functionality of the set component value function this allows for time-varying loads and changed to occur in the system. The *TimeSeriesPowerFlow* class has the following parameters:

- **m**: A *PowerFlowTopology* instance with the power system specifications to be simulated.
- **tstep**: A numerical value specifying the time interval at which the power flow will be evaluated at.
- **Tstop**: A numerical value specifying the final time of the simulation, assuming that the state time is zero.
- **components**: A set of component objects in a cell array with an *operate()* method for each that is called during each time interval before a power low solution.
- **monitor**: A cell array of strings specifying the id's that will be log during the simulation run.

The class the following methods:

- **addComponent()**:Method to add a component to the list of faults to be called.
- **addFault()**: Method to add a fault to the list of faults to be called.
- **simulate()**: This method executes a sequence of power flow solutions for the number of tsteps between zero and **Tstop**. At each interval the following actions take place: **update_components()**, **maxFlow()**, and then **update_output_structs()**.
- **update_components()**: Sets the new configurations for each power system components. *update_output_structs()*: Updates the log of flow, weights, rating, and switch state.

An example of a TimeSeriesPowerFlow object can be seen in [25].

Event-Driven Quasi-Static Power Flow Simulation. The *EventDrivenPowerFlow* class is used to perform an event-driven quasi-static power flow simulation. It is an extension of the *TimeSeriesPowerFlow* class, equipped with a communication module to interface with an external controller. The *EventDrivenPowerFlow* class, in addition to the parameters specified for the *TimeSeriesPowerFlow* class it has the following parameters:

- **control**: A cell array of strings specifying the id's for controllable components.

- **iface_ip**: An IP address to the interface manager.
- **iface_port**: A port that will be used with the interface manager.
- **iface_socket**: A socket that provides communication capabilities between the simulation and interface manager.

In addition to the methods for the *TimeSeriesPowerFlow* class, this class has the following methods:

- **open_socket()**: This method creates and opens a UDP socket with the specified **iface_ip** and **iface_port**.
- **interface_handshake()**: This method performs a handshake with the interface manager by notifying the interface manager that the simulation environment is ready for communication and the number of timesteps for said simulation.
- **update_controller ()**: This method packages the most recent system measurements into a struct and sends to the interface manager via **send_json()**.
- **update_power_system()**: This method receives the most recent signal data containing the controller input via **recv_json()**, unpacks the data, and then calls **update_components()** to update the component values.
- **recv_json()**: This method receives a JSON object and then returns a variable of type struct.
- **send_json()**: This method packages a variable of type struct and generates a JSON object which is then sent though the UDP socket.
- **delete()**: This method is called when the class is cleared, it closes and deletes the socket so that a new connection can be established.

An example that shows the usage of an *EventDrivenPowerFlow* object can be seen in Appendix A.3.

Electrical Switch Component. The *TimeBasedSwitch* and *EventBasedSwitch* are both electrical switch components, that can open or close an electrical switch by means of an actuation profile or by an external input to the simulation. Both switch classes take the following inputs:

- **id**: A unique string identifier for the electrical switch.
- **node1**: The emanating node from an edge.
- **node2**: The terminating node from an edge.

- **tv**: A sequence of numerical values that represent an instance in time used with parameter Sv to describe the actuation profile.
- **Sv**: A sequence of numerical values that capture the state of an electrical switch either 1 for closed or 0 for open.
- **rating**: A numerical values the express the maximum power rating of the electrical switch.

Both classes have a single method:

- operate(): This method describes how the electrical switch operates. The *TimeBasedSwitch* strictly operates the component with the values from the actuation profile. The *Event-BasedSwitch* strictly operates the component with the values it receives from the external controller.

A.3 Sample Code for Quasi-Static Simulation

```
%% Setup
tstep=1;  % Time-step size.
Tstop=1.5*60;  % Duration of simulation.
topology = 'Topology_C3.xlsx';

%% Interface Manager Connection
remote_ip = '146.201.63.222';
remote_port = 45000;

%% Building Power System
m=PowerFlowTopology(topology);

%% Power Flow Simulation
sim=EventDrivenPowerFlow(m, remote_ip, remote_port,...
    'tstep',tstep,'Tstop',Tstop);

%% Time Varying Load Component
il_LP1 = rgrep('LP1',m.E.id);
sim.addComponent(TimeBasedLoad(...
    m.E.id{il_LP1}, ...
    m.E.node1{il_LP1}, ...
    m.E.node2{il_LP1}, ...
    [0 40 45 65 66], ... % tv
    10*[2 8 8 1 1])) % Pv

%% Generator Load Sharing Component
```

```
Egen=m.E;
i1=rgrep('^G\d+',Egen.id);
Egen=Egen(i1,:);
sim.addComponent(GenLoadShare('GenLoadShare1',Egen));

%% Electrical Switch initially closed, opens@t=30s -> closes@t=60s
cid = 'C_21_22';
il_c = find(contains(m.E.id,cid),1,'first');
sim.addComponent(EventBasedSwitch(cid,...
    m.E.node1{il_c}, m.E.node2{il_c},...
    [0 30 60],[1 0 1], m.E.rating(il_c)));

%% Perform Simulation
[f,w,r,s,d]=sim.simulate();

%% Results
figure();
subplot(2,1,1)
plot(f.t, f.LP1, '-', f.t,f.G1,'-',f.t,f.G2,'r--'); grid on;
legend('L1','G1','G2','Location','Best')
title('Power Levels'); ylabel('Power (MW)'); xlabel('Time (s)')

subplot(2,1,2)
plot(f.t, f.C_21_22, '-',...
    f.t, f.C_22_32, 'r--',...
    f.t, f.C_32_21,'b-.')
grid on;
legend('C_21_22','C_22_32','C_32_21',...
    'interpreter','none','Location','Best')
title('Cable Flows'); ylabel('Power (MW)'); xlabel('Time (s)')

clear sim
```

APPENDIX B

MESSAGE FORMATTING

B.1 JSON Encoding Example

For example, let us look at the following Python `dict` and use the JSON library, to export the dictionary into a human-readable format.

```
A = dict("signal1": 34, "signal2": 456, "signal3": 567)

import json

A_json = json.dumps(A)
```

Now, we consider a MATLAB `struct` with the same data and use the JSON function to prepare the data

```
A = struct("signal1": 34, "signal2": 456, "signal3", 567);

A_json = jsonencode(A);
```

Both results in the following string:

```
A_json="{"signal1": 34, "signal2": 456, "signal3": 567}"
```

One can then reconstruct a Python `dict` with

```
import json

A = json.loads(A_json)
```

Or a MATLAB struct with

```
A = jsondecode(A_json);
```

JSON is very similar to a YAML file, where YAML is known to be a superset of JSON. JSON was selected because it is easy to use and requires minimal installation. To use YAML in MATLAB another program is needed to process YAML files.

B.2 Binary Message Packing

For example, packing the following values [1, 4.67, 0] with the following format string '<ifi' into a array would result in `b'\x01\x00\x00\x00\xa\4p\x95@\x00\x00\x00\x00'`. If the order of the format string stays is known with respect to the position of value on the byte array, then it can be unpacked when the values are needed.

```
import struct
data = [1, 4.67, 0]       # data to pack
frmt = "<ifi"             # data type formatting string
# Data packing
r = struct.pack(frmt,data)
# Data unpacking
data_unpacked = struct.unpack(frmt,r)
```

APPENDIX C

INTERFACE MANAGER IMPLEMENTATION

C.1 Fault Management Interface Specifications

```
name: fm
network:
    - ip_address : '146.201.63.248'
    - port : 42000

direction: sim_first
simulator: qsps

# direction: ctrl_first
# simulator: rtds

# log_enable: True

to_control_signal_names:
    - ZPDswt : int
    - ZSDswt : int
    - PGMDswt : int
    - SLDswt : int
    - ZPaDApYI : float
    - ZPaDBpYI : float
    - ZPbDApYI : float
    - ZPbDBpYI : float
    - ZPcDApYI : float
    - ZPcDBpYI : float
    - ZPdDApYI : float
    - ZPdDBpYI : float
    - ZSaDApYI : float
    - ZSaDBpYI : float
    - ZSbDApYI : float
    - ZSbDBpYI : float
    - ZScDApYI : float
    - ZScDBpYI : float
    - ZSdDApYI : float
    - ZSdDBpYI : float
    - GaDApYI : float
    - GaDBpYI : float
    - GaDCpYI : float
    - GbDApYI : float
    - GbDBpYI : float
    - GbDCpYI : float
```

- GcDApYI : float
- GcDBpYI : float
- GcDCpYI : float
- GdDApYI : float
- GdDBpYI : float
- GdDCpYI : float
- GeDApYI : float
- GeDBpYI : float
- GeDCpYI : float
- LEDApYI : float
- LEDBpYI : float
- LEDCpYI : float
- LFDApYI : float
- LFDBpYI : float
- LFDCpYI : float
- RGDApYI : float
- RGDBpYI : float
- RGDCpYI : float
- LABDApYI : float
- LBBDApYI : float
- LCBDApYI : float
- LDBDApYI : float
- LABYP : float
- LBBYP : float
- LCBYP : float
- LDBYP : float
- LEAYP : float
- LEBYP : float
- LFAYP : float
- LFBYP : float
- RGaYP : float
- RGbYP : float
- GaRBaYP : float
- GaRBbYP : float
- GbRBaYP : float
- GbRBbYP : float
- GcRBaYP : float
- GcRBbYP : float
- GdRBaYP : float
- GdRBbYP : float
- GeRBaYP : float
- GeRBbYP : float
- GaRBaYVdc : float
- GaRBbYVdc : float
- GbRBaYVdc : float
- GbRBbYVdc : float
- GcRBaYVdc : float
- GcRBbYVdc : float
- GdRBaYVdc : float
- GdRBbYVdc : float
- GeRBaYVdc : float

```
    - GeRBbYVdc : float

from_control_signal_names:
    - ZPDswtext : int
    - ZSDswtext : int
    - PGMDswtext : int
    - SLDswtext : int
    - GaRBa2UVrefext : float
    - GaRBb2UVrefext : float
    - GbRBa2UVrefext : float
    - GbRBb2UVrefext : float
    - GcRBa2UVrefext : float
    - GcRBb2UVrefext : float
    - GdRBa2UVrefext : float
    - GdRBb2UVrefext : float
    - GeRBa2UVrefext : float
    - GeRBb2UVrefext : float
```

C.2 QSPS Interface Manager

```
class QuasiStaticInterface:
    def __init__(self, yaml_file, qs_dft_file,
                 sim_ip = '127.0.0.1', sim_port = 45000):
        self.sim_ip = sim_ip
        self.sim_port = sim_port
        self.sim_addr = (self.sim_ip,self.sim_port)

        self.qs_dft_file = qs_dft_file
        self._load_qs_dft(self.qs_dft_file)

        self.yaml_file = yaml_file
        self._parse_yaml(self.yaml_file)

        self.fr_sim_store = {}
        self.to_sim_store = {}

        self.counter = 0

        # create blocking udp socket and set to be reusable
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.settimeout(None) # blocking
        self.sock.setblocking(True)

        if sim_ip is not '127.0.0.1':
            addr = ('0.0.0.0', sim_port)
            self.sock.bind( addr )
            print(f'iface_sim: Binding simulation socket to {addr}')

        if not self.remote: self.simulator_handshake()
        # else: if remote ControlThread will manage handshake
```

```
        print('---')

    def enable_log(self):
        logfiledate = datetime.now().strftime('%Y_%m_%d_%H_%M_%S')
        logging.basicConfig(level=logging.INFO,
            filename=f'InterfaceManagerLogFiles/{logfiledate}_IM.log',
            format='%(asctime)s,%(msecs)06f; %(name)s; %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S')
        self.info = logging.getLogger('QSI').info

    def simulator_handshake(self, control_handshake = None):
        print('iface_sim: Performing simulator handshake.')
        self.sock.sendto('iface_ready'.encode(), self.sim_addr)
        data, sim_addr = self.sock.recvfrom(BUFFER_SIZE)
        assert sim_addr[0] == self.sim_ip, \
            f'''Unexpected ip address: {sim_addr[0]};
            Expecting: {self.sim_ip}'''
        ready, params = data.decode().strip('\n').split(',',1)
        self.params = json.loads(params)
        assert ready == 'simulator_ready', 'Simulator not ready.'
        if control_handshake is not None:
            control_handshake(params)
        self.sock.sendto('start'.encode(), self.sim_addr)
        print('iface: Handshake complete.')

    def _load_qs_dft(self, qs_dft_file):
        self.qs_dft = dict()
        self.qs_monitor = set()
        self.qs_control = set()
        with open(qs_dft_file) as csvfile:
            reader = csv.DictReader(csvfile)
            for row in reader:
                key = row.pop('id')
                self.qs_monitor.update([sig.split('#')[0]
                # '#' in int signal specifies binary signal
                    if '#' in sig else sig
                    for sig in json.loads(row['monitor'])])
                self.qs_control.update([sig.split('#')[0]
                # '#' in int signal specifies binary signal
                    if '#' in sig else sig
                    for sig in json.loads(row['control'])])
                self.qs_dft[key] = row

    def _parse_yaml(self, yaml_file):
        with open(yaml_file, 'rt') as fp: yaml_str = fp.read()

        self.all_fr_sim_to_iface_keys = set()
        self.all_to_sim_fr_iface_keys = set()
        for yaml_doc in yaml.load_all(yaml_str):
            if yaml_doc is not None:
                self.all_fr_sim_to_iface_keys.update(
```

```
                self.extract_keys(
                    yaml_doc['to_control_signal_names']))
            self.all_to_sim_fr_iface_keys.update(
                self.extract_keys(
                    yaml_doc['from_control_signal_names']))
            self.log_enable = yaml_doc['log_enable'] \
                if 'log_enable' in yaml_doc else False
            if self.log_enable: self.enable_log()
            self.remote = yaml_doc['remote'] \
                if 'remote' in yaml_doc else False
            self.simulator = yaml_doc['simulator'] \
                if 'simulator' in yaml_doc else 'rtds'

    @staticmethod
    def extract_keys(list_of_dicts):
        '''
        @return a set of keys from a list of dictionaries.
        '''
        return set([list(dct.keys())[0] for dct in list_of_dicts])

    def __getitem__(self, key):
        '''
        If fr_sim is an empty dict, then wait to receive message.
        Else, pop value with corresponding key, so when all keys
        are popped the process repeats.
        Assumes that message will be JSON object.
        '''
        if not self.fr_sim_store:
            data, srvr_addr = self.sock.recvfrom(BUFFER_SIZE)
            if self.log_enable:
                self.info(f"sim; iface_{self.counter}; {data}")

            assert srvr_addr[0] == self.sim_ip, \
                f'''Unexpected ip address: {srvr_addr[0]};
                Expecting: {self.sim_ip}'''

            try:
                self.fr_sim_store = json.loads(data)
            except ValueError:
                raise ValueError('Message not a JSON object.')

            diff=self.fr_sim_store.keys()-self.all_fr_sim_to_iface_keys
            assert self.all_fr_sim_to_iface_keys == \
                self.fr_sim_store.keys(), \
                f'''Unexpected keys: {diff};
                Expecting: {self.all_fr_sim_to_iface_keys}'''

        return self.fr_sim_store[key]

    def __setitem__(self, key, val):
        '''
```

```
        Each set item call will fill a key of the dictionary.
        Once all the keys that are defined in the IDL are in
        the dict, send control inputs to simulation environment.
        '''
        self.to_sim_store[key] = val
        if self.all_to_sim_fr_iface_keys == self.to_sim_store.keys():
            data = json.dumps(self.to_sim_store).encode()
            self.sock.sendto(data,self.sim_addr)
            self.counter += 1
            if self.log_enable:
                self.info(f"iface_{self.counter};␣sim;␣{data}")
            if self.counter % display_every_n == 0:
                print(f'qsi:␣Exchanges:␣{self.counter}\n——')
                print(f'qsi_{self.counter}:␣'+
                      f'sent:␣{data}␣to␣{self.sim_addr}')
            self.to_sim_store = {} # clear store
            self.fr_sim_store = {} # clear store
            if self.counter == self.params['timestep']:
                raise Exception('End␣of␣simulation␣timesteps.')

    def check_to_sim(self, signame, type):
        '''
        Cross-references signame to the
        list of control signals in the qs_dft
        '''
        return signame in self.qs_control

    def check_from_sim(self, signame, type):
        '''
        Cross-references signame to the list
        of monitor signals in the qs_dft
        '''
        return signame in self.qs_monitor
```

C.3 RTDS Interface Manager

```
class SuperGtfpga:
    def __init__(self, gtfpgas):
        self.gtfpgas       = gtfpgas
        self.key_to_gtfpga = {}

    def __getitem__(self, key): return self.key_to_gtfpga[key][key]
    def __setitem__(self, key, val):   self.key_to_gtfpga[key][key] = val

    def check_to_sim(self, signame, type):
        for g in self.gtfpgas:
            if g.check_to_sim(signame, type):
                self.key_to_gtfpga[signame] = g
                return True
        return False
```

```
    def check_from_sim(self, signame, type):
        for g in self.gtfpgas:
            if g.check_from_sim(signame, type):
                self.key_to_gtfpga[signame] = g
                return True
        return False
```

C.4 Control Interface Manager

```
class ControlThread(threading.Thread):
    def __init__ (self, name,
        ip_address, port,
        from_control_to_iface, to_control_from_iface,
        simulator,
        send_period=None,
        direction = None,
        log_enable = False,
    ):
        super().__init__()
        self.name = name
        self.from_control_to_iface = tuple(from_control_to_iface)
        self.to_control_from_iface = tuple(to_control_from_iface)

        self.ip_address = ip_address
        self.port       = port
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.simulator = simulator
        self.counter = 0
        self.data = None
        self.client_addr = None
        self.send_period = send_period
        self.direction = direction

        print(f'log_enable : {log_enable}')
        self.log_enable = log_enable
        if self.log_enable: self.enable_log()

        self.sock.bind( ('0.0.0.0', self.port) )
        if self.simulator.remote:
            self.simulator.simulator_handshake(
                control_handshake=self.control_handshake)

    def enable_log(self):
        logfiledate = datetime.now().strftime('%Y_%m_%d_%H_%M_%S')
        logging.basicConfig(level=logging.INFO,
            filename=f'InterfaceManagerLogFiles/{logfiledate}_IM.log',
            format='%(asctime)s,%(msecs)06f; %(name)s; %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S')
        self.info = logging.getLogger('ControlThread').info
```

```
def control_handshake(self, params):
    print('Performing control handshake.')
    addr = (self.ip_address, self.port)
    self.sock.sendto(f'ctrl_iface_ready, {params}'.encode(), addr)
    data, ctrl_addr = self.sock.recvfrom(BUFFER_SIZE)
    assert ctrl_addr[0] == self.ip_address, \
        f"Expected ip address: {self.ip_address}"
    ready = data.decode()
    assert ready == 'ctrl_ready', 'Controller not ready.'
    self.sock.sendto('start'.encode(), addr)

def send_to_control(self):
    fmt_str = '<'
    packed_data = bytes()
    for to_val in self.to_control_from_iface:
        signal_name, fmt = list(to_val.items())[0]
        sim_value = self.simulator[signal_name]
        packed_data += struct.pack(f'<{fmt[0]}', sim_value)
        # fmt (e.g., float, int)
        fmt_str += fmt[0]

    self.client_addr = (self.client_addr[0], self.port)
    self.sock.sendto(packed_data, self.client_addr)
    if self.log_enable:
        self.info(f"iface; {self.name}; {packed_data}")
    v = struct.unpack(fmt_str, packed_data)
    fmt_vals = [f'{val:.2f}' for val in v]
    self.counter += 1
    if self.counter % display_every_n == 0:
        print(f'sent to {self.name} ({self.client_addr}): {fmt_vals}')

def send_to_rtds(self, data):
    """ returns True if no RTDS variable
        found otherwise returns False
    """
    if self.log_enable:
        self.info(f"{self.name}; iface; {data}")
    signals = []
    for item in self.from_control_to_iface:
        # from_control_to_iface = [ {signame, type}, ... ]
        signame, sigtype = list(item.items())[0]
        signals.append((signame, sigtype[:1].lower()))

    fmt = '<' + ''.join([fmt_chr for _,fmt_chr in signals])
    try:
        values_from_control = struct.unpack(fmt, data)
    except Exception as exc:
        print(f'Exception({exc}): {self.name}({self.client_addr})')
        print(''.join([sig_name for sig_name,_ in signals]) + '------')
        raise
```

```
        for i,signal in enumerate(signals):
            signame, _ = signal
            val_to_rtds = values_from_control[i]
            if not math.isnan(val_to_rtds):
                self.simulator[signame] = val_to_rtds
            else:
                print('WARNING: Nan recieved ' +
                    f'({self.name} {self.client_addr})')
        self.counter += 1
        if self.counter % display_every_n == 0:
            print(f"received from {self.name}: {values_from_control}")

    def run(self):
    print(f'Starting {self.name}, at port, {self.port}')
    try:
        if self.send_period is None:
            if self.direction == 'sim_first':
                self.sim_initiated_comm()
            elif self.direction == 'ctrl_first':
                self.control_initiated_comm()
            else:
                raise Exception('Communication direction not defined.')
        else:
            self.periodic_comm(self.send_period)
    except Exception as exc:
        print(f'WARNING: run(): {exc}')

    def control_initiated_comm(self):
        print('Control initiated communication.')
        while True:
            try:
                self.data, self.client_addr = self.sock.recvfrom(2048)
                if self.client_addr[0] != self.ip_address:
                    print('received from unexpected ip address: '+
                        f'{self.client_addr[0]}; '+
                        f'expecting {self.ip_address}')
                    continue

                self.send_to_rtds(self.data)
                self.send_to_control()
                self.counter += 1
                if self.counter % display_every_n == 0:
                    print("Number of data processed in "+
                        f"{self.name}: {self.counter}")
            except Exception as exc:
                print(f'WARNING: {exc}')
        sys.exit('ERROR: abnormal termination, exiting.')

    def sim_initiated_comm(self):
        print('Simulation initiated communication.')
        self.client_addr = (self.ip_address, self.port)
```

```
        while True:
            try:
                self.send_to_control()

                self.data, self.client_addr = self.sock.recvfrom(2048)
                if self.client_addr[0] != self.ip_address:
                    print('received from unexpected ip address: '+
                        f'{self.client_addr[0]}; '+
                        f'expecting {self.ip_address}')
                    continue
                self.send_to_rtds(self.data)

                self.counter += 1
                if self.counter % display_every_n == 0:
                    print("Number of data processed in "+
                        f"{self.name}: {self.counter}")
            except Exception as exc:
                print(f'WARNING: {exc}')
                if 'timesteps' in exc.__str__(): break
        sys.exit('ERROR: abnormal termination, exiting.')

    def periodic_comm(self, send_period):
        print('Periodic communication.')
        self.client_addr = (self.ip_address, self.port)
        TIMEOUT_SEC = send_period/1e9
        self.sock.settimeout(TIMEOUT_SEC)
        while True:
            self.send_to_control()
            try:
                self.data, self.client_addr = self.sock.recvfrom(2048)
                self.send_to_rtds(self.data)
            except socket.timeout:
                pass
```

C.5 Control Interface

```
class Signal:
    def __init__(self, name, type):
        self.name, self.type = name, type
        self.fmt = f'<{self.type[0].lower()}'
        self.val = 0.0
        if self.fmt[-1] == 'i':
            self.val = int(self.val)

    def bytes(self):
        if self.val is None:
            print(f'WARNING: {self.name} None -> 0')
            self.val = 0

        if self.fmt[-1] == 'i':
            self.val = int(self.val)
```

```
        return struct.pack(self.fmt, self.val)

    def update(self, data):
        nr_bytes = struct.calcsize(self.fmt)
        _bytes = data[:nr_bytes]
        self.val = struct.unpack(self.fmt, _bytes)[0]
        return data[nr_bytes:]

class SignalConnection:
    def __init__(self, yaml_entry, gtfpga_address='10.146.64.21'):
        self.name = yaml_entry['name']
        self.gtfpga_address = gtfpga_address
        def get_signals(yaml_key):
            signals = [ Signal(*list(dct.items())[0])
                for dct in yaml_entry[yaml_key] ]
            print([s.name for s in signals])
            return collections.OrderedDict([
                (s.name,s) for s in signals
            ])
        print('to_control_signal_names:')
        self.drts_to_ctrl_signals = \
            get_signals('to_control_signal_names')
        print('from_control_signal_names:')
        self.ctrl_to_drts_signals = \
            get_signals('from_control_signal_names')
        net_dct = {}
        for dct in yaml_entry['network']:
            k,v = list(dct.items())[0]
            net_dct[k] = v
        #self.ip_address = net_dct['ip_address']
        self.port       = net_dct['port']

        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind( ('0.0.0.0', self.port) )

        self.remote = yaml_entry['remote'] \
            if 'remote' in yaml_entry else False
        self.send_period = yaml_entry['send_period'] \
            if 'send_period' in yaml_entry else None
        self.simulator = yaml_entry['simulator'] \
            if 'simulator' in yaml_entry else 'rtds'
        self.params = {'simulator': self.simulator}
        if self.simulator == 'qsps':
            self.interface_handshake()
            self.set_socket_blocking(True)
        elif self.simulator == 'rtds':
            self.set_socket_blocking(False)

    def set_socket_blocking(self,blocking):
        if blocking:    block_timeout = (True,None)
```

```
        else:               block_timeout = (0,0)
        self.sock.setblocking(block_timeout[0])
        self.sock.settimeout(block_timeout[1])

    def interface_handshake(self):
        print('Performing interface handshake.')
        addr = ( self.gtfpga_address, self.port)
        print('Waiting on interface response.')
        data, iface_addr = self.sock.recvfrom(BUFFER_SIZE)
        assert iface_addr[0] == self.gtfpga_address, \
            f"Expected ip address: {self.gtfpga_address}"
        ready, params = data.decode().split(',',1)
        self.params.update(json.loads(params))
        assert ready == 'ctrl_iface_ready', 'Interface not ready.'
        print('Interface ready.')
        self.sock.sendto('ctrl_ready'.encode(), addr)
        data, _ = self.sock.recvfrom(BUFFER_SIZE)
        start_msg = data.decode()
        assert start_msg == 'start', 'Cannot start controller.'
        print('ctrl: Handshake complete.')

    def exchange(self):
        packed_data = b''.join(
            [s.bytes() for _,s in self.ctrl_to_drts_signals.items()])
        self.sock.sendto(packed_data, (self.gtfpga_address, self.port))
        data = None
        while True:
            try: data, _ = self.sock.recvfrom(BUFFER_SIZE)
            except socket.error:
                break
        if data is not None:
            for _,sig in self.drts_to_ctrl_signals.items():
                data = sig.update(data)
        else:
            print('WARNING: No update')
            print([s.val for _,s in self.drts_to_ctrl_signals.items()])

    def exchange_with_blocking(self):
        data = None
        try: data, _ = self.sock.recvfrom(BUFFER_SIZE)
        except socket.error as e:
            raise(f'ERROR: {e}')
        if data is not None:
            for _,sig in self.drts_to_ctrl_signals.items():
                data = sig.update(data)
        else:
            print('WARNING: No update')
            print([s.val for _,s in self.drts_to_ctrl_signals.items()])
        packed_data = b''.join(
            [s.bytes() for _,s in self.ctrl_to_drts_signals.items()])
        self.sock.sendto(packed_data, (self.gtfpga_address, self.port))
```

```
    def update(self, dct):
        ''' update dct with drts-ctrl values
            update local with ctrl-drts values
        '''
        for sig_name,sig in self.ctrl_to_drts_signals.items():
            sig.val = dct[sig_name]
        for sig_name,sig in self.drts_to_ctrl_signals.items():
            dct[sig_name] = sig.val
        return dct

    def get_signal_to_self_mapping(self):
        return { s.name:self
            for _,s in {**self.ctrl_to_drts_signals,
                        **self.drts_to_ctrl_signals}.items()
        }

class GtfpgaUDP:
    def __init__(self, yaml_file, iface_ip, blocking = False):
        self.yaml_file = yaml_file
        self.iface_ip = iface_ip
        self.block = blocking
        with open(yaml_file, 'r') as stream:
            signal_connections = [
                SignalConnection(entry,gtfpga_address = self.iface_ip)
                    for entry in yaml.load_all(stream)]

        self.signal_connections = signal_connections
        self.signame_to_conn = {}
        self.params = {'timestep': float('inf')}
        for sc in signal_connections:
            for k,v in sc.get_signal_to_self_mapping().items():
                self.signame_to_conn[k] = v
            self.params.update(sc.params)

        if self.params['simulator'] == 'rtds':
            self.block = False
        self.clear_cache(store_history=False)
        self.nbr_exchanges = 0

        self.t_previous_update = time.time()

        self.all_fr_sim_to_iface_keys = set()
        self.all_to_sim_fr_iface_keys = set()

        for sc in self.signal_connections:
            self.all_fr_sim_to_iface_keys.update(sc.drts_to_ctrl_signals)
            self.all_to_sim_fr_iface_keys.update(sc.ctrl_to_drts_signals)

    def get_bit_signal(self, bits_signal, bit_nr):
        def bit_is_set(bit_idx, bits_signal):
```

```
            bit_status = self[bits_signal] & (1 << bit_idx)
            return bit_status != 0

        def set_bit(bit_idx, bits_signal, val=1):
            if val == 1:
                update_val = self[bits_signal] |  (val << bit_idx)
            else:
                update_val = self[bits_signal] & ~(1   << bit_idx)
            self[bits_signal] = update_val
            return update_val

        class Bit:
            def is_set(self):
                return bit_is_set(bit_nr, bits_signal)
            def set(self, value=1):
                return set_bit(bit_nr, bits_signal, value)

        return Bit()

    def set_bit(self,bit_idx, bits_signal, val = 1):
        return self.get_bit_signal(bits_signal=bits_signal,
            bit_nr=bit_idx).set(value=val)
    def bit_is_set(self, bit_idx, bits_signal):
        return self.get_bit_signal(bits_signal=bits_signal,
            bit_nr=bit_idx).is_set()
    def set_simulation_off(self):
        pass
    def is_online(self):
        return True

    def store_cache_history(self):
        self.store_history = self.store.copy()

    def clear_cache(self, store_history = True):
        if store_history:           self.store_cache_history()
        self.store = {name:None
            for name,_ in self.signame_to_conn.items()}
        if not store_history:       self.store_cache_history()

    def update_values(self,verbose=False):
        self.nbr_exchanges += 1
        if self.nbr_exchanges == self.params['timestep']+1:
            raise Exception('End of simulation timesteps.')
        print(f'---\nExchange Number: {self.nbr_exchanges}')
        t_current_update = time.time()
        elapsed_time = (t_current_update-self.t_previous_update)*1e3
        print(f'Elapsed time from last update: {elapsed_time} ms.')
        self.t_previous_update = t_current_update
        print('---')

        for _,s in self.signame_to_conn.items():
```

```
            s.update(self.store)
        try:
            for sc in self.signal_connections:
                if self.block:      sc.exchange_with_blocking()
                else:               sc.exchange()

            res = 'vvv\n<- From ctrl to sim:\n'
            for sig in self.all_to_sim_fr_iface_keys:
                res += f'<- {sig} : {self[sig]}\n'
            res += '^^^'
            if verbose: print(res)
        except Exception as exc:
            sys.exit(f'WARNING: {exc}')

    def update_sequence(self,verbose=False):
        self.update_values(verbose=verbose)
        self.clear_cache()

    def echo_update_sequence(self,delayed=True,verbose=False):
        self.echo_values(delayed=delayed,verbose=False)
        self.update_sequence(verbose=verbose)

    def echo_values(self,delayed=True,verbose=False):
        '''
        HACK: Used when the FM is running remotely with
        to ensure that the correct signal values are sent back
        during the first signal exchange.
        delay       : Maps the output signal with the previous value sent.

        Mapping:
            - G[abcde]RB[ab]YVdc    -> G[abcde]RB[ab]2UVrefext
            - ZPDswt                -> ZPDswtext
            - ZSDswt                -> ZSDswtext
            - PGMDswt               -> PGMDswtext
            - SLDswt                -> SLDswtext
        '''
        for sig in self.all_to_sim_fr_iface_keys:
            if self[sig] is None:   # if already update do nothing
                if 'G' in sig[0]:   # generator set points
                    self[sig] = self[f'{sig[:5]}YVdc']
                else:
                    self[sig] = self[sig[:-3]] # binary values
                # echo delayed values
                if delayed:
                    self[sig] = self.store_history[sig]
                res1 += f'{sig} : {self[sig]}\n'
            else:
                res2 += f'{sig} : {self[sig]}\n'

    def __getitem__(self, key):
        self.signame_to_conn[key].update(self.store)
```

```
        return self.store[key]

    def __setitem__(self, key, value):
        self.store[key] = value
```

BIBLIOGRAPHY

[1] H. Ravindra, M. Stanovich, and M. Steurer, "Documentation for a notional two zone medium voltage dc shipboard power system model implemented on the rtdstm," *ESRDC*, 2016.

[2] ESRDC, "Model description document notional four zone mvdc shipboard power system model," *ESRDC*, 2019.

[3] "Real-time digital simulator technologies," 2020. [Online]. Available: https://www.rtds.com/

[4] N. Doerry and K. McCoy, "Next generation integrated power system: Ngips technology development roadmap," Naval Sea Systems Command Washington DC, Tech. Rep., 2007.

[5] C. Neuwirt, J. D. H. Haslwanter, M. Stanovich, K. Schoder, H. Ravindra, M. Steurer, and C. Wong, "Tree-based reconfiguration algorithm for zonal shipboard power systems," in *2019 IEEE Electric Ship Technologies Symposium (ESTS)*. IEEE, 2019, pp. 249–256.

[6] K. Schoder, M. Stanovich, T. Vu, H. Vahedi, C. Edrington, M. Steurer, H. Ginn, A. Benigni, C. Nwankpa, K. Miu *et al.*, "Evaluation framework for power and energy management shipboard distribution controls," in *2017 IEEE Electric Ship Technologies Symposium (ESTS)*. IEEE, 2017, pp. 388–393.

[7] K. Schoder, M. Stanovich, V. Vu, C. Edrington, and M. Steurer, "Extended heterogeneous controller hardware-in-the-loop testbed for evaluating distributed controls," in *Proceedings of the International Ship Control Systems Symposium (iSCSS)*, 2018.

[8] C. Edrington, T. Vu, H. Vahedi, D. Gonsoulin, D. Perkins, B. Papari, G. Ozkan, K. Schoder, H. Ravindra, M. Stanovich *et al.*, "Demonstration and evaluation of large-scale distributed control for integrated power and energy mvdc systems on ships," in *Advanced Machinery Technology Symposium*, 2018.

[9] K. Schoder, H. Ravindra, M. Stanovich, J. Langston, I. Leonard, and M. Steurer, "Shipboard power system baseline modeling and evaluation," in *2019 IEEE Electric Ship Technologies Symposium (ESTS)*. IEEE, 2019, pp. 536–541.

[10] B. Mohebali, K. Schoder, M. Stanovich, M. Steurer, and G. Erlebacher, "Role of sensitivity analysis in stress testing power system controllers," *Florida State University: Department of Scientific Computing*, 2020, accessed: 2020-11-19. [Online]. Available: https://www.sc.fsu.edu/xpos/2020/1484-mohebali-behshad

[11] C. Ogilvie, J. Ospina, C. Konstantinou, T. Vu, M. Stanovich, K. Schoder, and M. Steurer, "Modeling communication networks in a real-time simulation environment for evaluating controls of shipboard power systems," *arXiv preprint arXiv:2008.10441*, 2020.

[12] J. Oberg, "Why the mars probe went off course [accident investigation]," *IEEE Spectrum*, vol. 36, no. 12, p. 34–39, Dec 1999. [Online]. Available: https://spectrum.ieee.org/aerospace/robotic-exploration/why-the-mars-probe-went-off-course

[13] E. Fosse and C. L. Delp, "Systems engineering interfaces: A model based approach," *2013 IEEE Aerospace Conference*, Jan 2013.

[14] O. L. de Weck, "Fundamentals of systems engineering," 2015. [Online]. Available: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-842-fundamentals-of-systems-engineering-fall-2015/

[15] J. Spacey, "Types of information flow," Oct 2017. [Online]. Available: https://simplicable.com/new/information-flow

[16] "Layers of osi model," accessed: 2020-11-02. [Online]. Available: https://www.geeksforgeeks.org/layers-of-osi-model/

[17] D. Phillips, *Just Enough Systems Engineering*. Dwayne Phillips, 2010. [Online]. Available: http://dwaynephillips.net/systemsengineering/JustEnoughSystemsEngineering.pdf

[18] NASA, *Systems Engineering Handbook*, N. Aeronautics and S. Administration, Eds. National Aeronautics and Space Administration, 2016, nASA SP-2016-6105 Rev2.

[19] S. R. Hirshorn, "Expanded guidance for nasa systems engineering. volume 1: Systems engineering practices," *National Aeronautics and Space Administration*, 2016.

[20] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold *et al.*, "The functional mockup interface for tool independent exchange of simulation models," in *Proceedings of the 8th International Modelica Conference*. Linköping University Press, 2011, pp. 105–114.

[21] "Functional mockup interface target for simulink coder," accessed: 2020-11-02. [Online]. Available: https://www.mathworks.com/products/connections/product_detail/iti-fmi-target.html

[22] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[23] "Connext data distribution service professional," accessed: 2020-11-02. [Online]. Available: https://www.rti.com/products/connext-dds-professional

[24] J. Langston, M. Steurer, M. Bosworth, D. Soto, D. Longo, M. Uva, and J. Carlton, "Power hardware-in-the-loop simulation testing of a 200 mj battery-based energy magazine for shipboard applications," in *2019 IEEE Electric Ship Technologies Symposium (ESTS)*. IEEE, 2019, pp. 39–44.

[25] J. Langston, K. Schoder, M. Stanovich, M. Andrus, and M. Steurer, "Framework for analysis of distributed energy storage version 2.0," Electric Ship Research and Development Consortium (ESRDC), Tech. Rep., January 2016.

[26] N. Doerry and J. Amy, "The road to mvdc," in *ASNE Intelligent Ships Symposium*, vol. 5, 2015.

[27] E. Team, "Model description document notional four zone mvdc shipboard power system model," *ESRDC Website*, 2017. [Online]. Available: https://www.esrdc.com/library/notional-four-zone-mvdc-shipboard-power-system-model/

[28] ESRDC, "Draft esrdc initial notional ship data," *ESRDC Website*, 2017. [Online]. Available: https://www.esrdc.com/library/draft-esrdc-initial-notional-ship-data/

[29] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2006.

[30] M. Babaei, J. Shi, and S. Abdelwahed, "A survey on fault detection, isolation, and reconfiguration methods in electric ship power systems," *IEEE Access*, vol. 6, pp. 9430–9441, 2018.

[31] C. Diendorfer, J. D. H. Haslwanter, M. Stanovich, K. Schoder, M. Sloderbeck, H. Ravindra, and M. Steurer, "Graph traversal-based automation of fault detection, location, and recovery on mvdc shipboard power systems," in *2017 IEEE Second International Conference on DC Microgrids (ICDCM)*. IEEE, 2017, pp. 119–124.

[32] C. Diendorfer, "Graph traversal based fault management for medium voltage dc shipboard power systems," Master's thesis, University of Applied Sciences Upper Austria, 2017.

[33] C. Neuwirt, "A tree-based reconguration algorithm for zonal shipboard power systems," Master's thesis, University of Applied Sciences Upper Austria, 2019.

[34] "The official yaml web site," https://yaml.org/, accessed: 2020-11-18.

[35] "Introducting json," https://www.json.org/json-en.html, accessed: 2020-11-18.

[36] M. Tahan and J. Z. Ben-Asher, "Modeling and analysis of integration processes for engineering systems," *Systems engineering*, vol. 8, no. 1, pp. 62–77, 2005.

[37] D. Zowghi and V. Gervasi, "The three cs of requirements: consistency, completeness, and correctness," in *International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage*, 2002, pp. 155–164.

[38] P. W. Sauer and M. A. Pai, *Power system dynamics and stability*. Wiley Online Library, 1998, vol. 101.

[39] "grtheory - graph theory toolbox," https://www.mathworks.com/matlabcentral/fileexchange/4266-grtheory-graph-theory-toolbox, accessed: 2020-11-08.

BIOGRAPHICAL SKETCH

EDUCATION

- **M.S. in Electrical Engineering.** Florida State University, Fall 2020 - GPA 4.0/4.0.
- **B.S. in Computer Engineering.** Florida State University, Spring 2018 - GPA 3.6/4.0.

EXPERIENCE

- **Graduate Research Assistant** - *Center for Advanced Power Systems at Florida State University* (May 2018 - Dec 2020)
  - Developed an adaptable signal interface to ease the control development process of a fault management approach on a notion four-zone MVDC shipboard power system.
  - Leveraging the concept of root cause analysis to design an electrical protection system for a low voltage DC network of a shipboard power system utilizing solid state circuit breakers on a digital real-time simulation platform.
  - Implemented a fault management system, that identifies, isolates, reconfigures the grid network for short circuit faults of a notional shipboard power system to ensure that critical loads are always satisfied on a digital real-time simulator.
  - Designed custom amplifier PCBs, created MATLAB script to auto generate Factory Acceptance Tests & created a Modbus API to communicate between proprietary hardware and the real time digital simulator, while working in an expert controlled environment.
  - Successfully implemented C code on a DSP to read an indicator off an embedded computer for isolation of fault.
- **FREEDM Undergraduate Researcher** - *FAMU-FSU College of Engineering at Florida State University* (Aug 2017 - May 2018)
  - Implemented a system of ODEs in conjunction with the Newton Raphson method in C++ to model & simulate the charge and discharge characteristics of Li-S batteries, with the goal to reduce the transport shuttling effect.
  - Producing 3D visualizations in Blender of the internal dynamics of lithium sulfur species interacting with the cathode interface, to provide a clear picture of the research being produced.
- **Control Systems Internship** - *Utilities and Engineering Services at Florida State University* (Mar 2015 - May 2018)

- Organizing and managing HMI programs with Siemen's APOGEE Building Automation Software to be accessible by the Field Technicians, Engineers and the Florida State University management.
- Programming of enhanced alarms for fault detection purposes that require temperature sensitive research environments, producing a safe contingency plan for the researcher's work.

TOOLS AND SOFTWARES

- **Programming Languages**: Python, MATLAB, C++.
- **Softwares**: MS Office, MS Excel, MS PowerPoint, MS Access, AutoCAD, Blender, RSCAD, KiCAD, LabView, Latex

AFFILIATIONS

- IEEE Power and Energy Society (PES) student member.
- IEEE Control Systems Society (CSS) student memeber.