Masaryk University
Faculty of Informatics

Smart Grid Testing Management Platform

Bachelor's Thesis

Martin Schvarcbacher

Brno, Fall 2017

Masaryk University
Faculty of Informatics

Smart Grid Testing Management Platform

Bachelor's Thesis

Martin Schvarcbacher

Brno, Fall 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Martin Schvarcbacher

Advisor: Bruno Rossi, Ph.D.

Acknowledgement

I would like to thank my advisor Bruno Rossi, Ph.D. for his help in making this thesis a possibility and his help during our consultations. I would also like to thank Mgr. Katarína Hrabovská for her help and her master thesis on Smart Grids [1], on which this thesis is based. I would also like to thank the members of Lasaris for providing valuable feedback during my writing. I want to express my gratitude to my family for their support during my studies and writing this thesis.

Abstract

The Smart Grid domain is highly complex with an infrastructure composed of multiple layers, devices and components that have to communicate, integrate and cooperate together as a unified system. Testing of Smart Grid components and simulations within the Smart Grid ecosystem can assist in uncovering potential problems in security, functionality and reliability. In this thesis, we implement and showcase a new platform for executing real-time hardware-in-the-loop Smart Grid tests and experiments along with examples of the platform's usage. Furthermore, a summary of the requirements for testing and simulations of Smart Grids with hardware-in-the-loop testing is presented and the Mosaik co-simulation framework is covered along with its applicability for Smart Grids testing.

Keywords

Smart Grid, hardware in the loop testing, Smart Meters, real-time testing, co-simulation framework

Contents

1 Introduction **1**
- 1.1 *Problem Statement* . . . 2
- 1.2 *Thesis Contribution* . . . 2
- 1.3 *Thesis Structure* . . . 3

2 Smart Grids and their Testing Requirements **4**
- 2.1 *Smart Grid Ecosystem* . . . 4
- 2.2 *Smart Grid Software Testing Requirements* . . . 6
- 2.3 *Hardware-in-the-Loop Testing* . . . 12

3 Mosaik Framework **14**
- 3.1 *Overview* . . . 15
  - 3.1.1 Simulator API . . . 17
  - 3.1.2 Scenario API . . . 20
  - 3.1.3 Simulator Manager . . . 21
  - 3.1.4 Scheduler . . . 21
- 3.2 *Mosaik Limitations* . . . 22
  - 3.2.1 Data flows and circular data flow handling . . . 22
  - 3.2.2 Real-Time Simulation Capabilities . . . 24

4 Smart Grid Testing Management Platform – Design **27**
- 4.1 *Use Cases* . . . 27
- 4.2 *Domain Model* . . . 28
- 4.3 *Platform Deployment* . . . 33

5 Smart Grid Testing Management Platform – Implementation **35**
- 5.1 *Mosaik Interface* . . . 35
- 5.2 *Configuration Generation* . . . 36
  - 5.2.1 Boundary Value Testing . . . 36
  - 5.2.2 Enhanced Simulator Connection Support . . . 37
- 5.3 *Test Execution and Result Processing* . . . 39
  - 5.3.1 Test Executor . . . 40
  - 5.3.2 Test Run Scheduler . . . 40
  - 5.3.3 Local Test Evaluator . . . 41
  - 5.3.4 Global Test Evaluator . . . 41
  - 5.3.5 Test Result Processing . . . 42

5.3.6 Logging 43
5.4 *Examples of Usage* 44

6 Conclusion **46**
6.1 *Future Work* 47

Bibliography **48**

A Appendix **52**
A.1 *Testing Management Dashboards* 52
A.2 *Attachments* 53

List of Tables

2.1 Summary of SG testing requirements *11*
3.1 Mosaik example simulators *23*

List of Figures

2.1 Smart Grid Architecture Model *6*

3.1 Architectural view of the Mosaik Framework *16*
3.2 Mosaik simulation topology *17*
3.3 Mosaik API levels *18*
3.4 Mosaik API calls *20*
3.5 Mosaik circular data flow handling *24*

4.1 Use Case Diagram *28*
4.2 Test and its dynamic TestRun view *29*
4.3 Requirements necessary for each Test to execute *30*
4.4 Measure representation *31*
4.5 Compound inequality object diagram *31*
4.6 Expected and observed measures during a test run *32*
4.7 Simulation Topology *33*
4.8 Deployment diagram *34*

5.1 Test templating representation *37*
5.2 Conceptual model of simulator connections *38*
5.3 Framework generated connections between simulators *39*
5.4 State diagram of a Test Run execution *41*
5.5 Global Test Evaluator activity diagram *42*
5.6 Activity diagram of a Test Run inside the framework *43*

A.1 Test definitions overview *52*
A.2 Overview of Test Runs for Test *52*
A.3 New test definition creation *53*

1 Introduction

Today's world is increasingly becoming dependent on electricity, which needs to be supplied reliably and efficiently exactly where and when it is required. Additionally, the number of electricity producers has been increased rapidly in the past few years when households or small enterprises built their own renewable energy power generators and stared to add back the excess electricity into an energy distribution grid. Power delivery requirements can change at a moment's notice, due to changing demands, power grid distribution failures or renewable sources of energy becoming unavailable. Even a momentary loss of power can lead to widespread blackouts and subsequent damage of devices and economical losses. To accommodate these changes and new requirements, several technologies are being actively developed, where one of the key components is a Smart Grid [2].

In a traditional power distribution grid, electrical power is transmitted in only one direction: from power plants to transmission lines and finally, to the energy consumers. This makes it slow to adapt to any change in the network, since there are only a few sensors at the distribution points. This can lead to the worst-case failure scenarios which result in total power loss (a blackout) in the entire grid. Grid operators have only a limited means to change how the power is distributed [3].

In contrast, a Smart Grid is a new enhanced approach that tries to solve all of the above problems. The Smart Grid is defined as an interconnected system of many devices in a power distribution grid. One of the key components are smart meters deployed at each distribution endpoint, which allows continuous power usage monitoring and reporting data back to the grid. Smart meters also allow consumers to monitor their usage and consequently modify their consumption based on the current price and conditions. Together all of these devices allow bidirectional flow of both information and electricity in the energy grid [3]. In addition, a Smart Grid can be segmented into multiple micro-grids, where each segment is capable of producing enough energy for a sustained operation and only taking energy from the surrounding grids when the demand exceeds the power production capacity. This allows moving power production closer to the consumers and promotes renewable energy production [2].

1.1 Problem Statement

One of the key challenges of establishing a new Smart Grid is being able to evaluate newly proposed system deployments or changes without having to rely on already deployed nodes in the world for evaluation. There is also a need to have a testbed for finding regressions in security, functionality or performance in all of the grid components. This testbed can be additionally used to continuously model and evaluate changing Smart Grid needs. The Smart Grid is composed of many devices, which need to be able to communicate between themselves; therefore, interoperability testing before these devices are purchased and deployed in large scale can reveal potential incompatibilities or it can serve as a validation tool to ensure they will not compromise the overall system's stability or security [1]. This brings us to the need to have a Smart Grid testbed platform. Having a reproducible testing environment can bring many benefits and for these reasons, this thesis puts forwards a platform that can be used to accomplish all of the above outlined goals.

1.2 Thesis Contribution

The contribution of this thesis is the following:

- Provide an overview of the requirements for software testing of Smart Grids and hardware-in-the-loop testing
- Describe the Mosaik co-simulation framework and its applicability for testing Smart Grid systems
- Propose a platform for running user specified tests of both hardware and software components in a Smart Grid and review their test results.
- Implement a prototype of this platform, which can serve either as a production environment or as a base for a future system.

1.3 Thesis Structure

This thesis is split into multiple parts as follows. Chapter 1 provides a high-level overview of the challenges and requirements of Smart Grid testing and introduces the problem domain. Chapter 2 introduces the overall concept of a Smart Grid along with the requirements for software testing of a Smart Grid environment. An outline of hardware-in-the-loop testing is introduced. Chapter 3 provides an overview of the Mosaik co-simulation framework. Chapter 4 introduces an architectural proposal of a system for Smart Grid device testing. Chapter 5 describes an implementation of the proposed system and shows examples of usage. Chapter 6 concludes this thesis with the summary of the requirements and how they were applied into the prototype system implementation.

2 Smart Grids and their Testing Requirements

This chapter provides an overview of the Smart Grid (SG) ecosystem and what is required to test SG systems. The requirements for SG simulation and testing are discussed with a focus on software requirements in section 2.2. The concept of hardware-in-the-loop (HIL) testing is introduced in section 2.3. HIL testing is an important feature of the platform described in chapters 4 and 5.

2.1 Smart Grid Ecosystem

A SG is a new innovative approach for power distribution across a given region. It builds upon a traditional power distribution grid, and thus it has all of its components with several important additions. At its core, it allows communication between SG components, which is accomplished by connecting all of the components and sensors into a grid network [3]. A SG has self-healing capabilities; therefore, in case of a failure of a power supplier or distributor, it can automatically reroute power across a different network or send an alert to other power generators to increase their power output [2]. In a traditional grid, this typically leads to blackouts for the entire distribution region. Power rerouting is accomplished by using controllable power routers in the power distribution grid. They allow mixing of different power sources or switching to a single dedicated power source for their controlled region to balance power output and consumption.

Another benefit of a SG is that individual power producers may decide to add electricity to the grid, mainly from renewable resources (e.g., photo-voltaic panels, wind turbines) [2]. By allowing even small scale producers to sell energy into the grid, the overall environmental impact of power production from non-renewable sources is reduced. This can also allow better load balancing during times of a peak power usage. Non-renewable power plants need time on the order of minutes to hours to change their power output, which is the reason why there must be a certain power production safety margin along with a deployment of smaller generators, which can be switched on or off quickly to meet the changing demands and to prevent catastrophic grid-wide blackouts [4].

One challenge with renewable energy sources is their intermittent availability, which can depend on the weather, time of day or other factors. For these reasons, it is currently not possible to fully rely on renewable energy sources until more efficient long-term energy storage solutions are developed. Nonetheless, incorporating renewable sources of energy into a grid can reduce the overall carbon footprint of power plants, if their predicted power output can be modeled reliably to reduce non-renewable power plant resource consumption while still keeping the energy grid reliable [4]. Combination of these features allows the SG to be more resilient to power failures and rapidly changing power needs due to its self-healing capabilities, which can enable a certain section of the grid to become independent ("island mode") if enough energy is supplied from small local producers in an emergency situation [3].

The core component for monitoring and source of consumer data for predictions in a SG is a Smart Meter (SM). SM are located at each power distribution endpoint (residential or industrial). They are responsible for monitoring the power consumption like traditional energy meters; in addition, they also send usage statistics back to the grid, which allows instant collection of energy consumption for a given region. Together with other data from the grid, an instant power production and power consumption profile can be generated. Over time, a consumption model can be developed, which will serve as a way to predict power usage over time. This can be used to intelligently change the power production of non-renewable energy power plants to meet the current and near-future energy demand, which will help to save resources and at the same time prevent potential blackouts [4]. With the help of SM deployments, the time required to detect these abnormalities is significantly reduced when compared to a traditional power distribution grid [5].

Together, all of these systems make up the SG, as shown in the Figure 2.1, which provides a high-level overview of all of the required systems. Throughout this thesis, the term "Smart Grid" will refer to the entire infrastructure as described in this section.

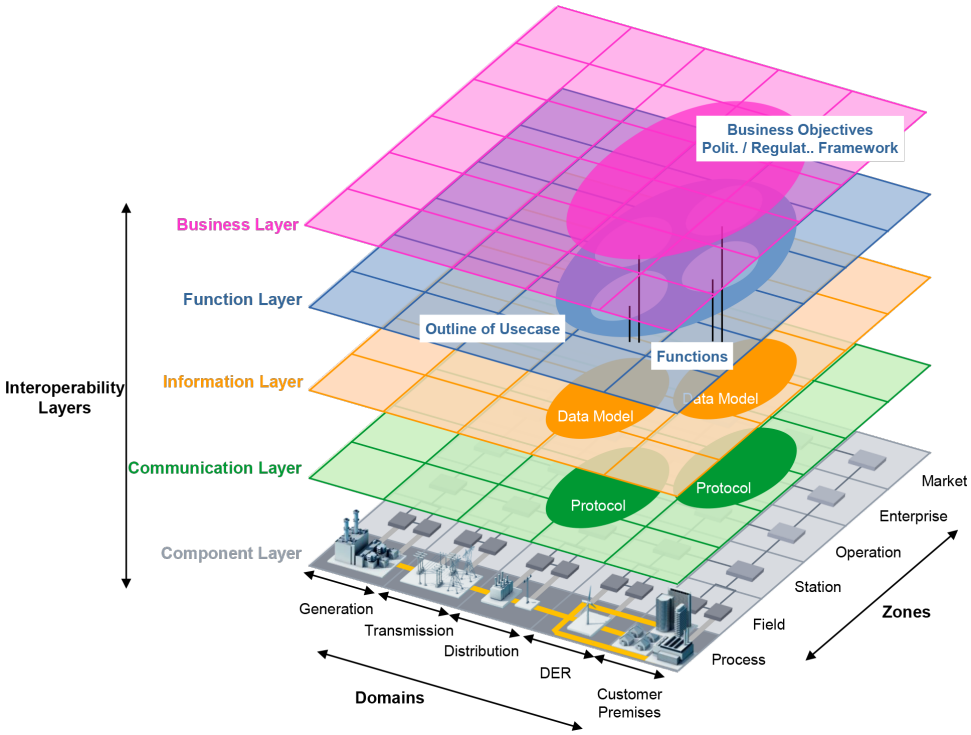

Figure 2.1: Smart Grid Architecture Model (Source: [6])

2.2 Smart Grid Software Testing Requirements

This section discusses the requirements for creating a Smart Grid (SG) testing environment from the software perspective. The software environment can take care of simulation orchestration, test result processing and providing the user meaningful feedback from each test. The SG domain encompasses many areas as shown in Figure 2.1; each of these areas must be tested in isolation and as a whole system before a deployment, when evaluating system changes or as a continuous testbed for replication of errors reported from already deployed infrastructure. The SG Testing Management Platform described in chapters 4 and 5 is designed to adhere to the requirements described in this section to the full extent allowed by the underlying technologies. A summary of the requirements described in this section is in Table 2.1.

As shown in [7], the core of each SG simulation is being able to simulate both the power grid (electricity flow) and data flows between

entities related to the power grid. Power flow in each simulation can be either real or simulated. For real power flows the energy is created by actual hardware (generators powered by turbines, photo-voltaic panels, etc.) and this power is transmitted to other parts of the grid. Other components can pass along this energy or consume it. By using a real power flow, it is possible to bring the simulation closer to the real deployment, as all aspects of power production, transmission and consumption can be tested. When using real power, it is also possible to exactly match field deployed hardware or use a scaled model; for example, if it would be infeasible to simulate high-level voltages inside a laboratory. Another possibility is to use simulated power flow, where all or some of the power flow is simulated in software. If all of the tested energy distribution devices are virtualized or it is possible to give hardware devices only the simulated readings, this alternative can work and give bigger flexibility in the amount of simulated devices and has lower requirements for the SG testing laboratory. If certain tested devices require power from the grid to function, it is possible to take a hybrid approach, in which the power flow is still simulated; however, the devices requiring power are attached to a generator, whose power output is controlled by the simulation environment. This frees the laboratory from having to deal with the setup of potentially large power generators and having to deal with energy distribution, while still retaining the ability to use actual hardware devices with grid supplied (and thus controlled) energy. Certain parts of the generation and distribution grid can be fully or partially virtualized with no impact on the remaining sections of the grid under testing.

From the information exchange side, according to [7], the simulation must handle the following three device (simulator) types: devices exchanging data not attached directly to the power grid (type A), devices directly attached to the power grid not exchanging data (type B; e.i., not "smart" devices), devices both connected to the power grid and exchanging data (type C). Each of these device types presents a different challenge for the testing platform, since they must all be handled differently. Type B devices are primarily power lines and transformers without any network reporting capabilities. They are primarily required for knowing about the entire SG topology and come into play in system-wide simulations, where all SG components must be tested. Information about them can be used in network load simulations and

failure simulations when parts of them are disconnected (made inoperative). Type A devices are part of the SG and are thus sending data to other parts of the grid, receiving data from other parts of the grid or external systems. These devices do not have to be software only, they can be for example weather stations used for predicting the power output of photo-voltaic panels. By sending commands to other components, they can influence the state of the SG. In the SG domain, these types of devices are primarily control systems (e.g., Supervisory control and data acquisition (SCADA) control systems). Type C devices are a combination of Type A and Type B devices; therefore, the simulation environment must be capable of interacting and simulating parts of both the power grid and the information exchange network. The simulation environment must be able to facilitate data and energy exchange between all of these components and observe the interactions between them.

An extension of the previous device classification is proposed in [8]. Devices are classified as previously described with the addition of intruders (both physical and virtual). This view focuses primarily on the network (data exchange) side, as it is mainly concerned with simulation and study of cyber-physical attacks on the SG. For the simulation of an attack requiring physical access to a device, the hardware device must present an interaction port to the simulation through which the attack can take place. This enables the initial stage of network infiltration through direct physical access. For attacks which happen without physical access, the simulation must expose the virtual device addresses to the intruders so they can run their attacks on the network. By being able to test cyber-physical attacks, it is possible to evaluate SG safety and identify potential attack vectors. Once an attack is taking place in the simulation, it can be further used for automatic anomaly and intrusion detection on the network. All of the network data should be traceable and logged for future analysis. Network analysis can be either real-time and part of the simulation, or it can be run after the simulation has finished if it requires significant amount of processing. Additionally, the simulation environment should support features such as controllable packet loss, congestion simulation and variable network bandwidth.

The review of [7, 9] shows the requirements for the evaluation of a SG simulation. These requirements include extensive interaction

capture between all of the devices under testing and any outside interactions with the system. No interactions should be done outside of the simulation environment, since these interactions can not be captured, reproduced and analyzed, leading to unreproducible tests. In an ideal situation, if all of the data exchange and state data is captured, it should be possible to "play back" the entire simulation or observe the state of any device at a specific point in simulation's time. For long-running simulations, this is a critical requirements, as it can be infeasible to run the simulation again to recreate the exact circumstances of the failure. These requirements leads to the next one: handling large amounts of data for processing and storage. If the state of the simulation should be recorded at each specific time, it means that there must be a logging system in place to handle processing of all of this data and subsequent storage for future retrieval. For real-time systems, it means that the logging system must also process this data continuously for an extended periods of time (days to months) without failure. Extensive logging can also aid in analyzing component or whole system crashes and aid in determining the root cause of a device failure (e.g., destruction, firmware inoperability) where the device is no longer operational and data can not be retrieved from it directly. Another requirement is being able to view the simulation state and state of the devices during the simulation to immediately know the state of all devices. This can be used by SG technicians to detect any possible faults early on in the simulation, thus saving time.

A SG can be deployed to many locations and it can also span several cities, regions or even countries. It is therefore important to simulate a SG not only for one specific topology, but for multiple possible topologies [9, 10]. The topology of the SG to be deployed can not always be known in advance during the evaluation phase, or it can be desirable to know if the created SG model will generalize to multiple possible topologies. This can be accomplished by generating multiple SG topologies, either automatically as in [10] or based on real-world data and requirements. The topology may also vary based on the given region, so finding a general solution is desirable. The simulation must handle multiple possible SG topologies, which are ideally changeable without the simulators being aware of this change or knowing anything about the surrounding topology in advance. This is generally not possible when simulating a full SG with real

hardware connected to the simulated power grid, as any change in topology could require physical changes (e.g., cable routing). From the simulation perspective, the SG topology can be fixed, changeable before the start of the simulation or changeable at runtime multiple times. By using a simulation capable of changeable topology, it is possible to simulate failures of certain parts of the grid and how the grid will re-route itself after a failure.

From a simulator integration perspective, the testing platform must support several types of simulators made by several companies and with no guarantees of compatibility [9]. The simulators can be single unit systems, communicating through a defined interface (both software and hardware) or the simulator can be composed of several units, creating a Multi Agent System (MAS). A single MAS can be reduced through abstraction to a single unit system, so the internals of the MAS can be considered a black box for the simulation. An alternative is to decompose the MAS into its individual components and connect them together by the simulation framework. The advantage of this approach is that parts of the MAS can be virtualized/exchanged more easily, while keeping other hardware parts for hardware-in-the-loop (HIL, see section 2.3) testing. Each simulator can be purely hardware (e.g., power generator), software (e.g., weather station) or a combination of the two (e.g., Smart Meter). The simulation environment and its API should allow for easy integration of current and future simulation entities and systems. If the simulator implements a certain industry standard for data and power exchange, it could allow seamless integration into the simulation environment by simply plugging it in and configuring it. For other devices where no standards exist, the simulation environment should allow the creation of an adapter between the simulation and simulator. This requires having a well-defined API for the simulation environment, which simulator integrators can use to create the adapter. Another advantage of this approach is that the simulation platform can support even previously unknown or new devices after an adapter is created.

Reference	Criteria	Support Requirements
[7]	Power flow	Real (1:1, scaled); simulated
[7]	Data flows	Power grid only; information grid only; combined
[7, 9]	Interaction capture	RT capture&monitoring; large data volume; simulation playback
[9, 10]	Topological changes	Before test; at simulation start; during runtime, multiple changes
[9]	Multi-agent systems	One entity; breakdown into components
[9]	Simulator integration	Well defined API; extensibility
[9, 8]	Entity classification	Power producer/consumer/transporter; state reporter; network intruder; SCADA
[8]	Network requirements	Network analysis; packet injection; expose to simulated intruders
[10]	Topology generation	Automatic; determine if model generalizes; model future SG deployments
[10]	Simulation platform	Support different SG topologies

Table 2.1: Summary of SG testing requirements

2.3 Hardware-in-the-Loop Testing

The principles for testing of purely software systems can not be directly applied to the SG domain, due to the importance of hardware testing [11], which is why Hardware-in-the-Loop (HIL) testing is one of the techniques used. HIL testing is a method of testing in which real hardware is incorporated into a simulation. This simulation can contain other hardware and software components, which can interact together as if they were physically connected. This can lead to an increase of simulation accuracy, as less hardware components have to be emulated via software [12]. Not all hardware can be reliably and accurately simulated, in which case HIL is the only option for its testing as part of a larger system. HIL can be used for the development and simulation of complex real-time systems, such as controllers used in SG systems [13]. HIL testing helps to reduce simulation costs while retaining the usage of real hardware and does not require developing software simulation models for all of the hardware which would be virtualized. Certain parts of the environment which can be difficult to recreate inside a laboratory (e.g., high voltage systems, see section 2.2) can be virtualized, while keeping other hardware in place for the simulation. HIL can also be used for algorithm verification on hardware and verifying that all components are capable of real-time interaction and event processing. This thesis primarily focuses on HIL testing, as this kind of testing is the most relevant for overall SG testing.

Portions of a system under test which can be emulated (e.g., sensors, probes) and can be subsequently controlled by the simulation platform. These emulated components can serve as an entry and measurement point for the HIL system under testing [13]. They must serve as a bridge between the hardware and the software simulation environment. Together the hardware, emulated components and software simulation makes up a HIL simulation platform. An example of HIL testing is shown in section 5.4, where a Raspberry Pi, Arduino, LED array and a photo-voltaic panel is used for the evaluation of photovoltaic panel deployment testing.

One of the downsides of HIL simulations is that hardware simulators can not be easily cloned like software simulators; therefore, the limit on how large a simulation can be is the lab's budget. Additionally, depending on the hardware, it can be hard to separate the tested hard-

ware from other external systems, which can increase the complexity of incorporating it into the simulation. Due to its physical nature, an experiment can not be temporarily suspended and neither is it possible to go back in simulation time. For this reason, all HIL testing requires that the underlying testing platform is capable of real-time event processing and sending information (simulation data) to all of the simulated components [12]. Another major consideration point is the simulation speed: HIL generally requires following the real wall clock (i.e., one simulation second is one real world second).

The next chapter provides an overview of the Mosaik Framework, which can be used for SG co-simulation and evaluates its HIL testing capabilities. This framework is the foundation of the Smart grid Testing Management Platform implementation described in chapter 5.

3 Mosaik Framework

Mosaik is an open-source co-simulation framework written primarily in Python [14, 15]. It aims to allow connection of multiple independent Smart Grid simulation entities into one centralized simulation grid. In this simulation grid, Mosaik takes care of all of the data exchange between the simulation entities and ensures simulator time-frame synchronization.

Time-frame synchronization is a crucial feature for all hardware-in-the-loop (HIL) testing frameworks, as all simulation entities can run with variable update rate ranging from once every few milliseconds to minutes or hours; by ensuring time-frame synchronization, all simulation entities can send and receive data from each other without having to worry about the liveness of the received data. This behavior of Mosaik is classified as discrete-event simulation, since each simulator runs in its own process and receives events signaling arrival of new data or a request to send data elsewhere.

In order to accomplish the goals of this thesis, several simulation frameworks were initially considered besides Mosaik. These frameworks included: OMNeT++ and Simulink. OMNeT++[1] is an open-source framework for performing network simulations [16]. It works as a discrete event based simulation environment, which poses problems for simulating continuously running systems in parallel, as it does not provide any methods for time synchronization across all of the simulated devices. Network communication is only a part of Smart Grid testing; other parts can include device reliability and interoperability. Fully simulating and evaluating the latter parts in OMNeT++ was considered problematic and thus OMNeT++ was not selected as the primary simulation framework. Simulink[2] is a commercial (closed-source) framework and multi-domain simulation environment, which can be used for the development of embedded system prototypes and their simulation or testing [17]. It can simulate software models and interface with hardware to run simulations which require communication between individual simulators. In this regard, it is similar to Mosaik and also provides the benefit of having the possibility to

1. `https://omnetpp.org`
2. `https://www.mathworks.com/products/simulink.html`

integrate into the MATLAB environment. Simulink was not selected primarily for licensing reasons. For the purposes of this thesis, Mosaik was chosen as it could best fulfill all of the criteria described in section 2.2 and being open-source, thus providing the possibility of extending the system as shown in chapters 4 and 5.

3.1 Overview

Mosaik provides a way to connect several independent simulators into one simulation and it does this by providing a single API for the user supplied simulators. Mosaik classifies all simulation entities as follows: Mosaik server and simulators. Figure 3.1 provides an architectural view. Mosaik server is transparent to the simulators. A simulator handles input/output sharing via the low/high-level API. Each simulator can have multiple models and each model can have multiple instances. An example of a simulator can be: a program that manages the runtime of several models or a program that interfaces with one or several hardware components. An example of a model can be software simulation of a photo-voltaic panel given certain input parameters or the individual hardware components. For certain simulation models allowing multiple model creations is not possible; e.g., if there is only one hardware component available and concurrent access is forbidden. For purely software based simulators, care must be taken to ensure the given simulator can handle all of the parallel model processing; subsection 3.2.2 shows further requirements.

One of the key advantages of Mosaik is its ability to integrate almost any simulator into the simulation by implementing the Mosaik API for each simulator. If the user implements the API for their simulator, the simulator could be almost anything, even a gateway to a completely different simulation framework such as Simulink. Thus in practice, it is possible to combine several simulation frameworks using Mosaik as the primary data exchange framework. In [19] the possibility of having self-contained Simulink simulators of photo-voltaic panels is discussed along with methods for integration of this Simulink simulation model into Mosaik, where power grid entities can be simulated, either by native Mosaik simulators or with user defined simulators.

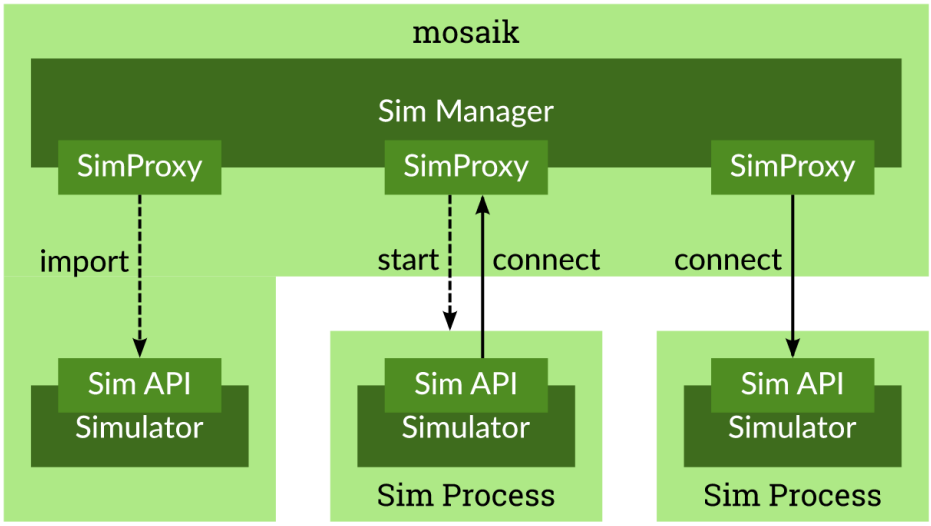

Figure 3.1: Architectural view of the Mosaik Framework (Source: [18])

Due to the fact that Mosaik requires reliable and real-time network communications between its individual simulators [16], it is not suitable for handling simulations requiring network responses over an unreliable network. If at any point in the simulation timeframe a request or a response from the simulator does not reach the simulation server, the simulation is terminated and classified as failed. This mechanism precludes the possibility of directly simulating unreliable networks with controlled features such as packet loss or variable bandwidth. The user must simulate sending these inputs out-of-band via another device and evaluate whether the out-of-band information was received in time or at all. While this imposes certain additional overhead, it also ensures that if a simulation succeeds, Mosaik was able to exchange all control data between the entities as required. One solution to incorporate simulation of unreliable networks is presented in [16], where the possibility of integrating OMNeT++ with Mosaik is discussed. Integrating OMNeT++ into Mosaik would remove the need for out-of-band management and enable creation of Mosaik-native unreliable data connections between individual simulators.

The Mosaik frameworks provides a conceptual abstraction of connecting different simulators together, which allows the user to view all

of the simulators as directly connected, as shown in Figure 3.2 labeled as logical data flow. However, due to the need to ensure data and individual simulation time-frames are synchronized, this data must be sent to the central Mosaik server, which takes care of distributing it to the designated model at the appropriate time. Thus the real model inside the framework looks like in Figure 3.2 labeled as actual data flow. This approach allows user transparent synchronization of both the time-frames and data to be exchanged.

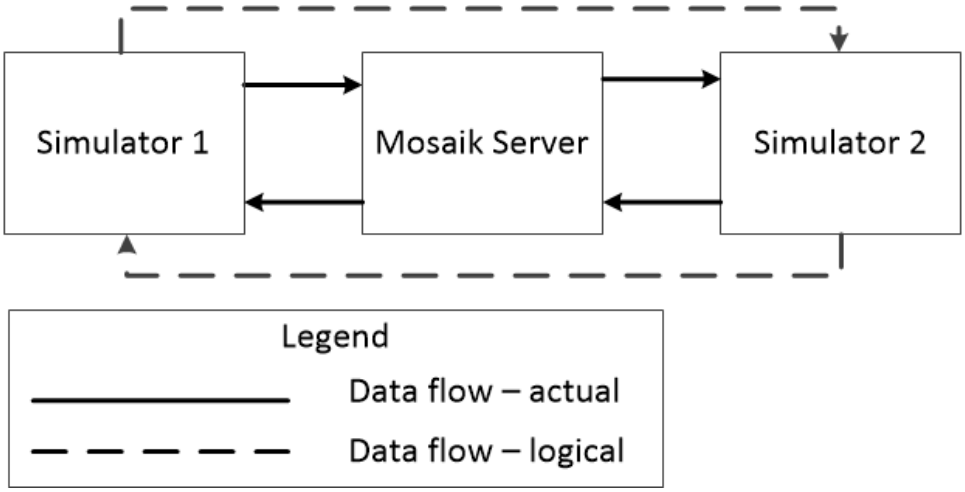

Figure 3.2: Mosaik simulation topology from a logical and implementation point of view

3.1.1 Simulator API

Mosaik provides an API for building user-defined simulators. There are two kinds of this API provided: high-level and low-level. The low level API provided is language agnostic, as it only describes how data communication between Mosaik and simulators works. The method of communication is web sockets which transmit JSON data. The user of the low-level API must handle network communications with the Mosaik server, create and process JSON and then interface with the simulator. Mosaik server will initiate a socket connection once the simulation is started. The high-level API is provided for Python, Java and C#. This version of the API takes care of all of the network

communications between the Mosaik server and the user's simulator, which frees the user from the need to handle network requests and process JSON into language specific objects. In addition, it provides language specific classes which can be extended to suit the specific simulator's needs. Figure 3.3 illustrates the differences in the API levels.

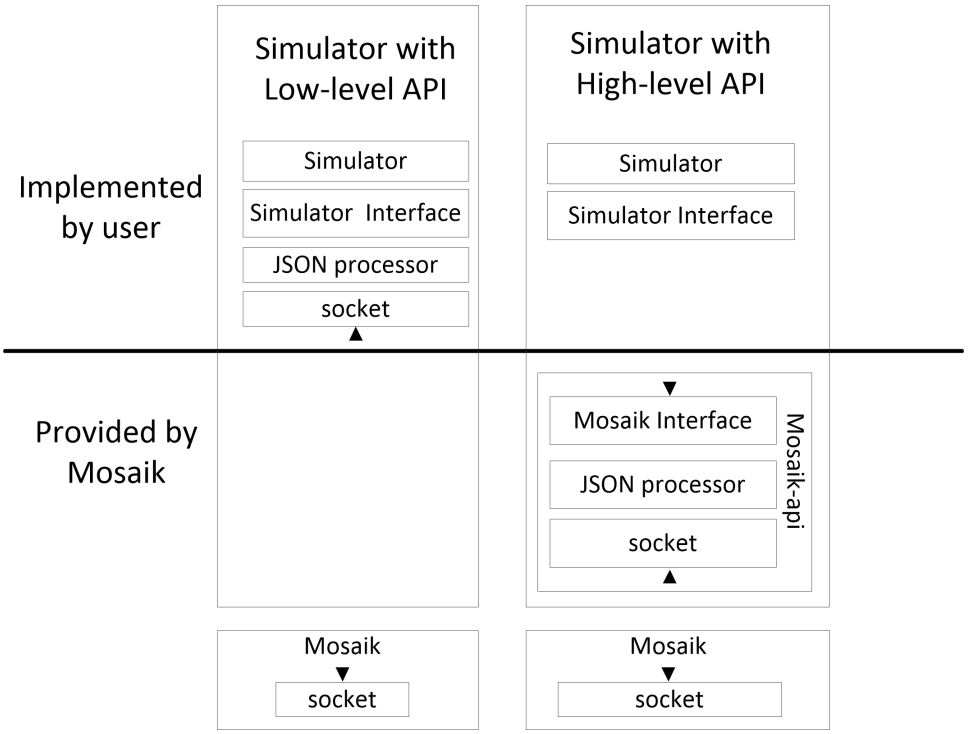

Figure 3.3: Mosaik API levels (Adapted from: [20])

Notwithstanding the API level selection, the user needs to handle at least the following data exchange requests to have a working simulator:

Simulation initialization. In this step, the user must create the required simulators, whether they are purely software based ones, hardware based ones or use HIL. For example, here the user can initialize connection to a hardware device via a hardware interface (e.g., RJ-45, RS-232, USB), send test initialization data, or reset the device and prepare it for testing. The API provides the user with simulation parameters, which can for example contains the desired step size of the simulator or simulator name prefix. At the end, the

user must return a meta-model of the simulator, which contains: API version, each model's attributes (inputs/outputs), parameters and model name.

Model creation. Since several models of different objects can exist in one simulator, this method specifically creates one or more instances of a model. The API provides the user with model parameters, which can contain model specific data. An example can be initial input/output values, network topology and other model specific data.

Step function. This function is called either each simulation step or in positive integer simulation step increments. In this function, the simulator must process received inputs from other simulators and pass these inputs to the corresponding models. Start of the execution of the step function is the only time input data can change for a simulator, which means that outside a simulation step the input data will not change further for any other reason. The simulator is responsible for calling the individual step functions of each model, to notify it of newly received inputs (state change). Finally, the user must return the time when the step function for this simulator should be called again. In case of asynchronous data sending, it is in this function that the corresponding asynchronous messages must be created and sent; the details are discussed in subsection 3.2.1.

Get data function. Called independently of the step function by Mosaik server, here the simulator must respond to any data requests for a specific model's output values (attributes). This data is then sent back to the Mosaik server, which processes it and distributes the corresponding outputs to destination model's inputs. This data is then sent in the `step` function.

Further optional methods are:

Setup done. Called after all of the simulators have been initialized but no `step` methods have been performed yet.

Stop. Called after all of the simulators have finished their simulations and no more `step` methods will be called.

The Figure 3.4 shows the sequence of the Mosaik API calls during simulation setup, simulator and model initialization and how the simulation is terminated. It also shows what calls and data is exchanged during each simulation step.

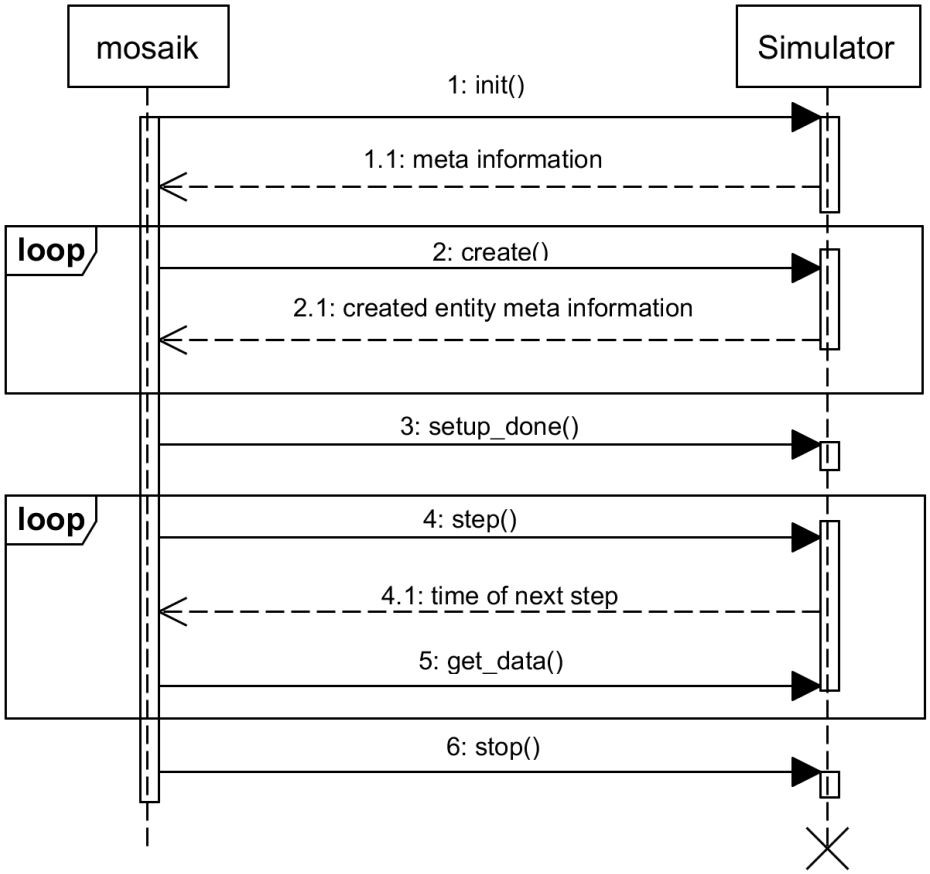

Figure 3.4: Mosaik API calls between Mosaik server and simulators (Adapted from [21])

3.1.2 Scenario API

The scenario API is used for creating simulation layouts (topology). The topology contains information about all of the simulators, namely how to reach them and their names. Each simulator can be reached in one of the following ways: starting an embedded Python class, starting a native OS process or connecting to an already running simulator defined by its IP:PORT address. This information is kept in `Mosaik.World`, which additionally keeps track of simulator time frames. From within `Mosaik.World` requests for individual simulator models are given. Once all of the simulator's models are instantiated,

they are connected (wired). Wiring works by specifying the name of the model and its output name along with the destination model and its input name. These connections must be designated as receiving asynchronous requests to enable two-way simulator communication. Without this flag, asynchronous requests are ignored and simulators time-frames can become desynchronized. These wire connections create the simulator data flow topology. Furthermore, the direct flow topology creates a directed acyclic labeled multi-graph. The limitations of using this approach are discussed in subsection 3.2.1. The final step is to initialize the simulation by specifying how many steps should be performed and whether the simulation should be real-time, in which case the "real-time scale factor" (how long should one real-world second take in the simulation) must be specified, along with whether the simulation should fail if real-time performance can not be achieved during the simulation. This allows flexibility in simulation time, as all components can be artificially sped up or slowed down, provided that they are capable of such performance.

3.1.3 Simulator Manager

The simulation manager is user transparent and is responsible for handling the socket communications between remote simulators, JSON data exchange and processing. Simulation manager translates all of the Scenario API requests into network requests, including simulator and model creation, input/output exchange and final simulation shutdown.

3.1.4 Scheduler

The scheduler uses the SimPy framework[3] for discrete-event simulation and controls when data should be exchanged between the simulators [15]. The advantage of this library is that it can allow variable step sizes of individual models and that these models can all run in parallel. One important fact to note is that all Mosaik time units are integer seconds, thus if the user wants to simulate real-time step size in units bigger than one second, appropriate conversion to seconds must be done by the user. For non-real-time simulation, the distinction

3. `https://pypi.python.org/pypi/simpy`

between seconds and any other unit is arbitrary, it is only important to choose the smallest possible base unit; the simulation will still attempt to run as fast as possible regardless of units chosen.

3.2 Mosaik Limitations

The following two subsections discuss certain implementation details and inner workings of the Mosaik framework, which are relevant for the design of the Smart Grid Testing Platform discussed in chapter 5. Being able to handle bi-directional communications between simulation entities was one of the key features that a Smart Grid simulation framework required to be able to simulate all possible Smart Grid topologies. Mosaik does support this feature, albeit with certain drawbacks which are discussed in subsection 3.2.1. By requiring HIL testing capabilities from the Testing Platform, it was critical to know whether Mosaik can handle real-time communications, which is a requirement for HIL simulation. After evaluating Mosaik's performance, it was determined that it is also suitable for this task if certain conditions are met, which are discussed in subsection 3.2.2.

3.2.1 Data flows and circular data flow handling

As a part of its core functionality, Mosaik must facilitate data transfer between individual simulators and ensure their time-frame synchronization, so that no simulator can go too forward in time. In the Scenario API, the user must define how the data will flow between individual simulation models. This is accomplished by specifying two tuples: (source model, destination model) and (source output name, destination input name). One important limitation of this approach is that there can not be direct circular data flows (both direct and transitive) between entities[4] [22]. However, in most non-trivial simulation environments, the need for entity A to communicate (share data) with entity B and vice-versa arises, creating a circular data flow between them. The steps to resolve this situation are described below. The following example simulation scenario illustrates how cyclic data

4. A discussion of these reasons from the Mosaik authors is in [22]

flows can be encountered in even very basic simulations. Table 3.1 shows two example Mosaik simulators: Entity and Controller.

Table 3.1: Mosaik example simulators

	Entity (E)	**Controller (C)**
Inputs:	delta	value
Outputs:	value	delta

There is a simulation entity (E); each simulation step, the E.value will be changed by current value of E.delta, thus `E.value += E.delta`. A controller entity (C) is tasked with keeping the E.value within a specified range, for example (-3, 3). A naive solution would be to directly connect E.value ⇒ C.value and C.delta ⇒ E.delta; however this is not currently supported in Mosaik, since this creates a data flow cycle between E and C. The only solution is to designate one entity as primary (action) and one entity as secondary (reaction). Once this is done, only connections going from primary to secondary are left in the Simulation API definitions. Handling data flows in the opposite directions is accomplished within the secondary model's `step` method using asynchronous data sending.

Figure 3.5 illustrates how data is exchanged between the two models. At the start of the simulation, the time is 0. Both E and C first receive their initial input values and run their respective `step` methods concurrently. After E is done simulating at $t = 1$, it changes its output values and Mosaik Simulator takes a snapshot of them. During the first step of C, it reacts to the initial input, makes computations and sends asynchronous request to change the input value of E.delta. This request will reach E at $t = 1$ and E will be able to react to it in its `step` method. As the chart illustrates, both E and C always run in parallel and their respective data is only exchanged at the start of the step boundary, never during. That is, no matter how many times the output value of E or C changes during a simulation step, only the value at the end of the step will be used as snapshot and transmitted to entities which requested to receive this data via their wire connection. Using this method, it is possible to simulate complex action-reaction multi agent systems within the Mosaik framework. However, the requirement to plan ahead which models will send data directly and

which asynchronously imposes several challenges on the end user implementing the API. It permanently ties the simulator to a certain model topology. For example in one test case it may be desired to have entity A as primary and entity B as secondary, and in another test case the opposite. With the current situation which requires the use of `send_async` inside `step` it would be required to create a new simulator for each topology. Further details on how data exchange works can be found in [23], a method for overcoming these limitations is in subsection 5.2.2.

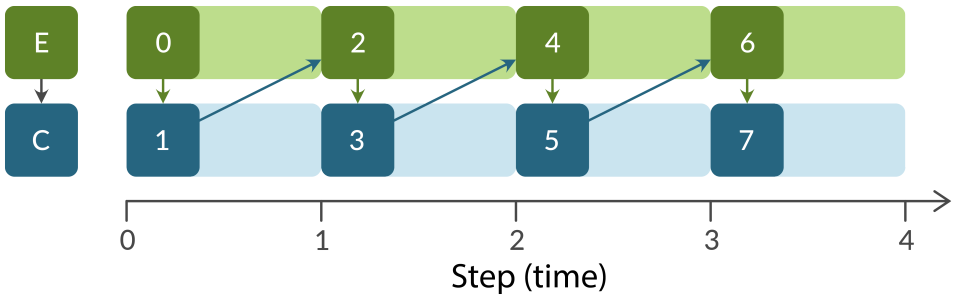

Figure 3.5: Mosaik circular data flow handling (Source: [23])

3.2.2 Real-Time Simulation Capabilities

For the purposes of testing Smart Grids, it is important that at least some simulations can be performed in real-time and with actual hardware components, which work in real-time. The Mosaik framework has support for both soft and hard real-time simulations, with some limitations discussed below. Each Mosaik simulation can designate whether it wants to be real-time or to run through the simulation as fast as possible. In case of real-time simulations, the specific time step can be specified with accuracy to fractions of a second. In each of these time steps, Mosaik will ensure data synchronization between simulation entities and if the simulator should be stepped into, it will send it new inputs. Additionally, the simulation can be specified to fail if some simulator fails to respond (send back data) within the specified time step. If no failure is desired, the amount of time drift from target time is logged for user's evaluation.

In case of purely hardware simulators with a software Mosaik API wrapper, the API only needs to read the current hardware values, send new data along with control commands and read back the current state to be sent to other entities. The order of these actions is important. For example, there can be the following possible sequence of events inside a `step` function:

1. Read received inputs from other simulators
2. Read hardware device state via a hardware interface (e.g., USB, RJ-45, RS-232)
3. Send inputs to hardware device via hardware interface
4. (If applicable) Send asynchronous data to other models

Depending on the implementation, there can pass a (significant) non-zero time between steps 2 and 3 on the device side due to the time it takes to communicate over the hardware interface and the hardware simulator still doing its own simulation in the background. If the order of steps 2 and 3 is swapped it would allow the device to react to the new input, thus possibly skewing the results. If asynchronous connections are used, then step 4 sends a snapshot of the data from step 2. This will capture the state of the hardware device before it had a chance to process the newly received inputs.

For software simulators, care must be taken to ensure that the simulation can run either continuously in its own thread and allow data exchange with the main simulator thread, or it must be able to simulate the required functionality within the time allocated to a step function.

The following paragraph discusses some of the limitations of the Mosaik framework in regards to how fast a simulation can exchange data while retaining all of the required real-time guarantees. Let the maximum simulation resolution be defined as the lowest possible step length in real-time second. For hardware simulators it must be considered how fast they can respond to the simulator's API calls which either send commands or request state data. From testing of devices using RS-232, the maximum simulation resolution approaches 500 milliseconds (ms), see section 5.4. In case of 500 ms resolution, this

would mean that the simulator could both send, process and receive data each 500 ms; communication failure or other delays would result in the simulation not running in real-time and thus being forcibly aborted by Mosaik. Attempt to request data at a faster rate could either result in simulation failure or the simulator not processing the data between simulation steps, due to the communication link saturation (e.g., exceeding the receiver buffer). Thus Mosaik should not be used for hardware simulations which require sub-second updates. However, should one second resolution be sufficient, tests in [24] have shown that it is adequate for even large scale real-time simulations if the mentioned constraints are followed. For purely software simulators, the constraint becomes network overhead for communication and the processing capability of the system along with maximum possible processes which can be handled in parallel.

This chapter provided an overview of the Mosaik framework, outlines some of its functionality and applicable use cases along with its technological capabilities. The next two chapters will showcase a Smart Grid Testing Management Platform which uses Mosaik for the Smart Grid simulation functionality.

4 Smart Grid Testing Management Platform – Design

This chapter describes the design of the proposed Smart Grid Testing Management Platform (SGTMP). An architectural overview containing use case diagrams to explain the basic platform functionality is presented in section 4.1, followed by a description of the domain model and an analysis of its individual parts in section 4.2. Finally, the method of deployment is discussed in section 4.3. The domain model is based on the work of Hrabovská [1] and is influenced by the ISO/IEC/IEEE 29119 Software Testing Standard. All of the UML models in this section are based on the work of Hrabovská [1] with modifications. In particular, each test in this domain model has a test run, it is not sub-divided further. The system for measurements was changed to incorporate their data source and measurements are now a direct part of the expected simulation results. Expected results can be defined per each step, not just for the end result, which allows more flexibility in how to define the acceptable ranges during a test run.

4.1 Use Cases

The functionality of this system is given in the form of a use case diagram in Figure 4.1, which shows an overview of the main Smart Grid Testing Management Platform functionality that is user accessible. A Smart Grid operator is the primary user of this system, responsible for adding new test scenarios to be executed into the system. Each test scenario is defined by a Smart Grid topology and test pass requirements. Currently the operator must also schedule tests for execution. Once the tests have been completed, the operator can view the results of a single test or view historical data for a given test to determine if there are any possible reliability or functionality problems. The SG Testing Platform System schedules prepared test runs for execution by checking if all of their test requirements are satisfied. If they are satisfied, then the test run can be started (executed). Once a test run is finished, the system saves all of the relevant log files, test results and if an incident occurred during the test run it stores it for future analysis.

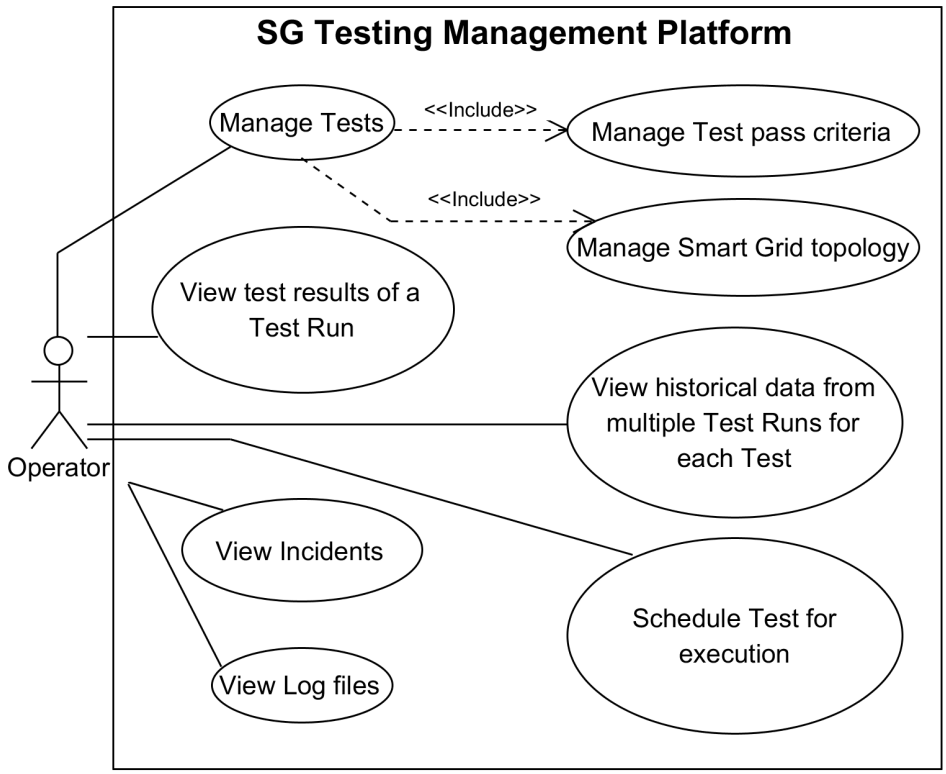

Figure 4.1: Use Case Diagram

4.2 Domain Model

This section details how the domain model of this Smart Grid Testing Platform looks like. The relationships between individual domain model entities are shown. A dynamic view showing the interactions between them is shown in chapter 5.

The most important class of this testing framework is the Test class and all of its immediately related classes shown in Figure 4.2. It defines what kind of a test to run, how it will run, what is its goal and how the result should look like. Each time a test is scheduled for an execution, a new test run is created. Multiple test runs can be scheduled for the execution from a single test, for example due to multiple combinations of initial variable values; a test can be thought of as a template from which multiple test runs are created. Each test run holds information

pertaining to a single test execution, with all test parameters. Test run is updated throughout its life-cycle, which is defined by TestRunStatus; the dynamic view is detailed in subsection 5.3.2. Every finished test run must be in one of the states defined by the TestRunEvaluation. If all of the criteria for a test success were met, it will be marked as successful and as failed otherwise. In case an incident happens, it will be marked in the special out of range state. Incidents are events that occur due to user error, framework error or hardware errors that can not be recovered from automatically. Each incident should be carefully investigated by the laboratory technician to determine the exact reason for its cause. An incident can for example hint at hardware failure and the need to replace the faulty hardware.

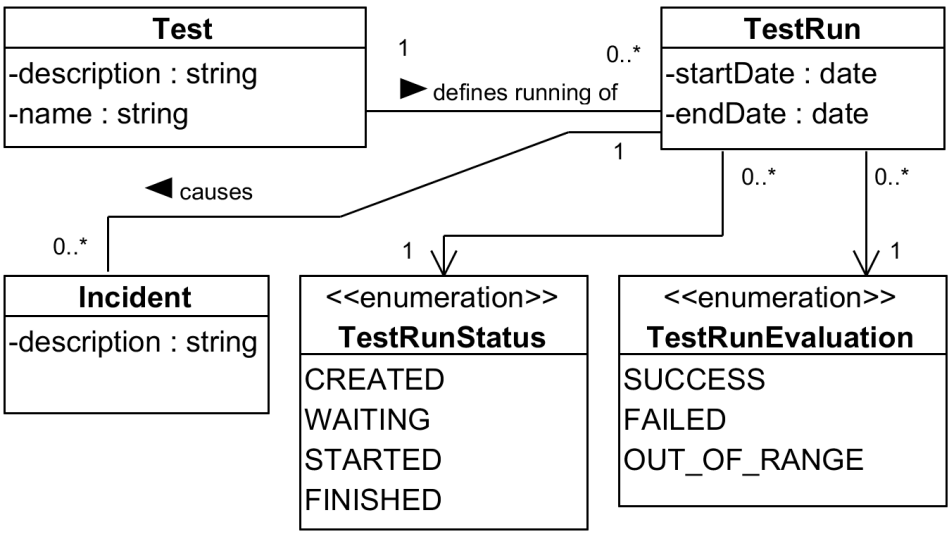

Figure 4.2: Test and its dynamic TestRun view

Each test can have several requirements that must be satisfied before the test can be executed. These requirements might be environmental, data requirements and simulator requirements. Environmental requirements can include a temperature inside the test room, a specific Smart Grid topology (e.g., minimum number of nodes) [1]. Data requirements define initial data that must be made available to the testing platform before testing can begin. This can be remote data that must be fetched and stored prior to each new test, local data

or a set of changing test template values (details in subsection 5.2.1). Simulator requirements show which simulators have to be made available for the duration of each test run, so that no two test runs attempt to use the same simulator and cause unexpected behavior. Certain software-only simulators can be marked as virtual and they are then treated as always available and with the possibility to have several of their instances running at once without any risks of interference.

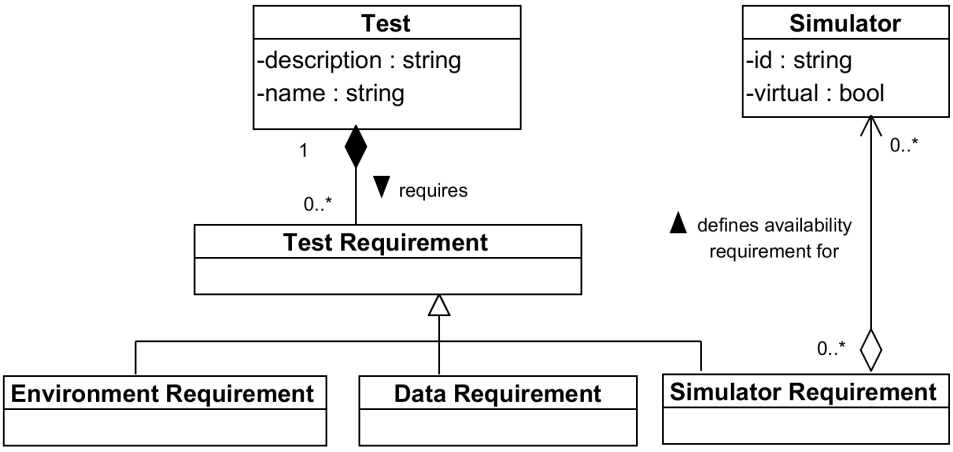

Figure 4.3: Requirements necessary for each Test to execute

To allow the Smart Grid operator to define complex test pass criteria, the following model of measurements presented in Figure 4.4 was created. It allows setting several attributes to be measured (e.g., voltage, current, pressure, internal state) during a test run. These measures can be recorded and evaluated whether they are in a specified range for each step of the test run. Each test step can define different measures and their acceptable values. As an example, at the start and within ten simulation steps, the observed measure of a device might be set with a larger tolerance than afterwards. If a measure falls outside the specified range set by expected result, the test will be marked as failed (but allowed to finish the entire test run). The SG operator can then investigate this failure and using the observed measure values, determine the root problem.

There are two basic ways to define an expected result for each test step. The most basic one is via a Numeric operand, which supports

three comparison modes: $x = y, x < y, x > y$, where x is a user supplied constant and y is the runtime compared measure value. The second method of setting expected criteria is by using the CompositeOperand. By using the composite AND and OR operands, more complex range criteria can be specified such as compound inequalities. An example showing a compound inequality is in Figure 4.5.

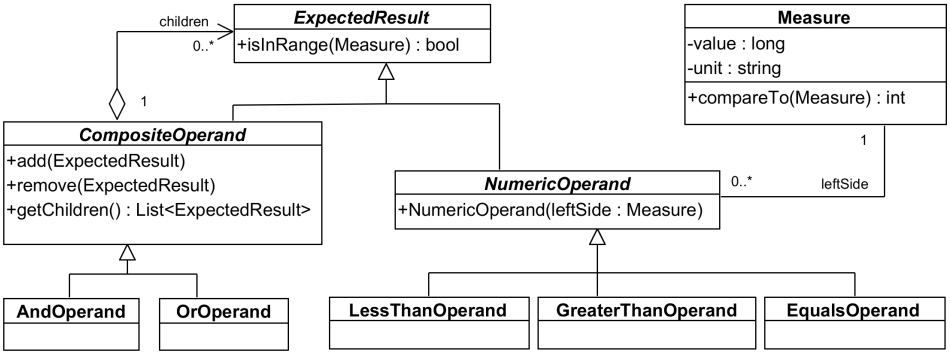

Figure 4.4: Measure representation

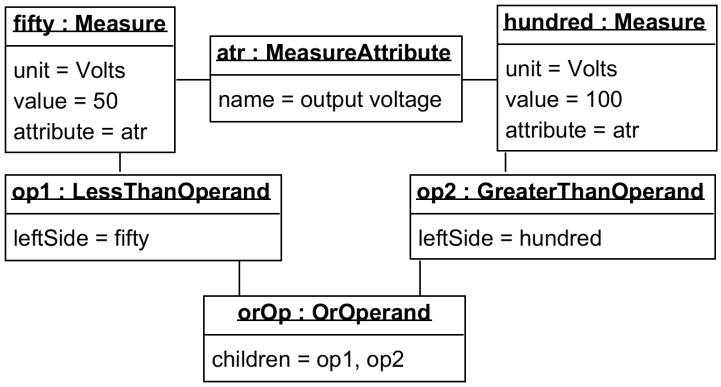

Figure 4.5: An object diagram representing the compound inequality: $x < 50 \bigvee x > 100$

To allow detailed state logging of each simulation model and allow comparison of actual and expected values per each step, the following technique presented in Figure 4.6 is used. Each Test can have several TestSteps, each defining what kind of measures are expected. Once a

TestRun is executed, simulators send their measures back and they are recorded for future use. This can be beneficial to determine what state a given simulator was in at a specific time and can aid in determining the cause of a test failure or other potential problems (this is the playback functionality requirement from section 2.2).

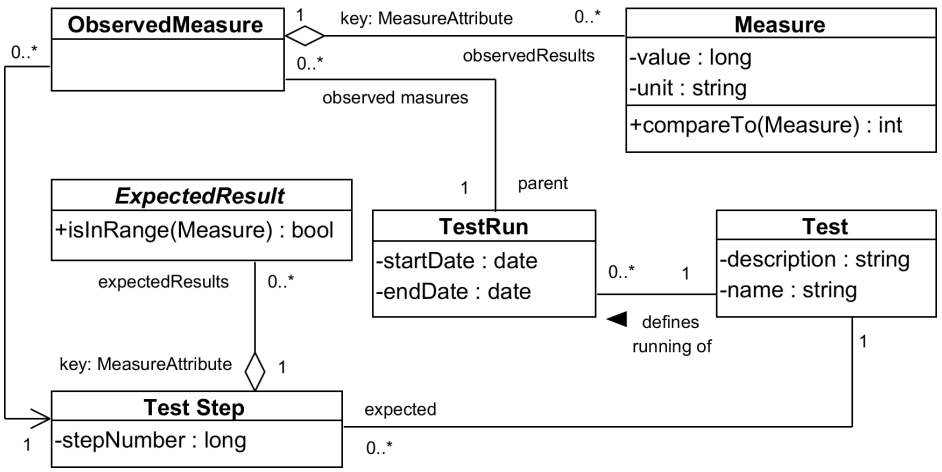

Figure 4.6: Expected and observed measures during a test run

To represent the simulated Smart Grid topology, the following model represented in Figure 4.7 is used. A simulator can be considered a black-box device that must be able to interface with the SGTMP. Some simulators may expose additional options that make their management easier (e.g., auto-restart, automatic log upload). Each simulator defines models, which are running inside of it. The model represents either a software or hardware models and their state inside the simulator. The simulator is responsible for exchanging data between the outside world and the models. If two internal models should exchange data between each other, it is possible to connect these models together and let Mosaik handle the data exchange, or the data exchange can be handled internally by the simulator. To represent all forms of data flows between models, the WireConnectionEdge class is used. From the relationships represented by this class, the full Smart Grid topology can be created and processed further by the SGTMP.

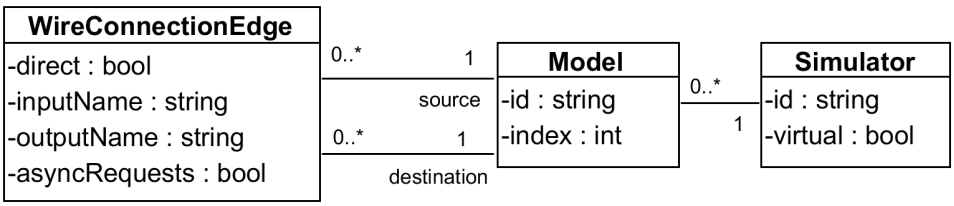

Figure 4.7: Simulation topology representing all of the data flows

4.3 Platform Deployment

The deployment diagram in Figure 4.8 showcases how the SGTMP is deployed. The primary component is the Testing Management Platform itself, running on the Java Virtual Machine. The functionality is exposed via an API and a HTTP server implementing a subset of the functionality as discussed in the Appendix A.1. On the same server, the Mosaik Interface must be running to support a connection to the Mosaik framework via inter-process communication (IPC). This Mosaik Interface along with Mosaik is responsible for communication with other simulators. Each simulator must implement the Mosaik API to be able to communicate with Mosaik. The implementation of the Mosaik API takes care of communication and data exchange with both software and hardware simulators. Software simulators can be running on the same machine or they can be accessed remotely and the Mosaik API implementation serves only as a bridge to them. Hardware simulators can be connected to the Simulator via a hardware interface (e.g., RS-232, RJ-45, USB) and the API implementation is responsible for managing this connection and other administrative tasks. The database server is a data store for storing all of the related test data, which are generated before, during and after each test run. The connection between SGTMP and the database is handled by Java Database Connectivity (JDBC).

This chapter introduced the design of the Smart Grid Testing Platform by showing certain parts of the design. A domain model for the testing environment was analyzed by its parts. The next chapter details the implementation based on the presented design and presents dynamic views of this platform.

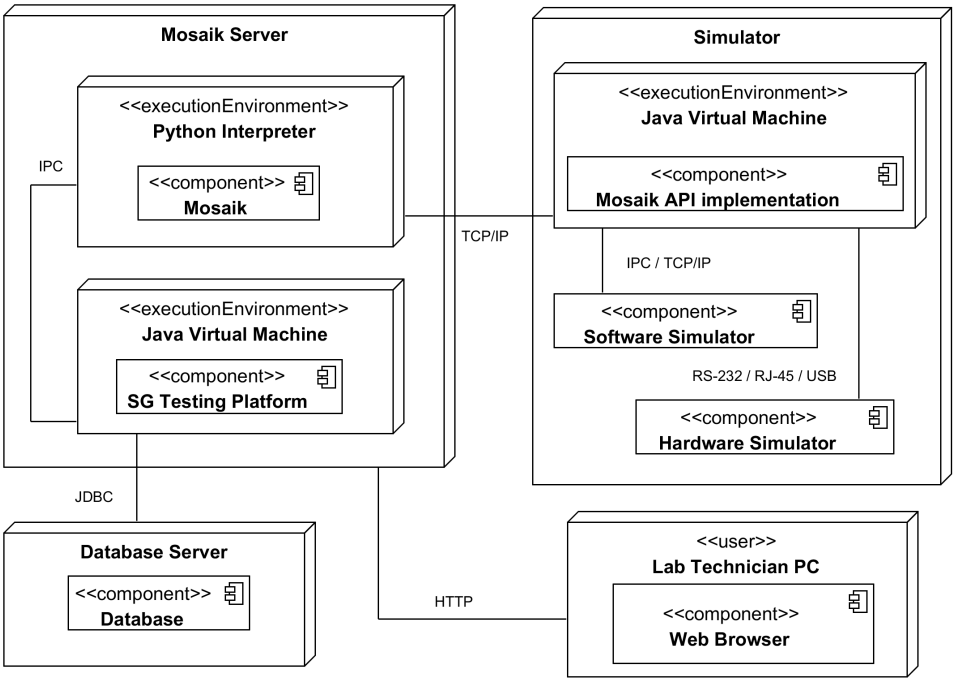

Figure 4.8: Deployment diagram of the Smart Grid Testing Framework

5 Smart Grid Testing Management Platform – Implementation

Based on the requirements discussed in chapter 2 and design discussed in chapter 4, a prototype implementation of a Smart Grid Testing Platform was developed. This platform has several parts, namely the Mosaik Interface written in Python and the Java based Testing Platform. The Testing Platform encompasses test management and test execution, viewing test or simulation results and a web interface. For a laboratory technician, this is the primary entry point to the Smart Grid Testing Management Platform (SGTMP). For developers, there is an extension of the Mosaik high-level Java API for integration of existing simulators into this system. Existing Mosaik simulators can be integrated with little changes required. Finally a demonstration of this platform's capability is presented on a sample scenario for photo-voltaic panel deployment in a Smart Grid.

5.1 Mosaik Interface

To interface with Mosaik from the SGTMP, a Python program is used. It is launched by the SGTMP as a separate process once all of the required configuration files have been generated. As an input parameter, it receives all of the configuration files required to start an individual simulation: simulation configuration, list of simulators, list of simulator's models and how the models are connected to each other. First it tries to establish a connection to all of the specified simulators if they are an external process or tries to start a new simulator running locally based on user entered parameters. Because several of these configuration files contain additional information that can be only known at runtime, they must be passed to the simulators and models during their creation in this part of the framework. In this phase all of the asynchronous connection data for each simulator is collected and then during their creation sent as part of additional information payload. Each simulator then receives data describing what models to create. In the next phase, all of the direct connections are connected in the Mosaik framework. If any connection contains asynchronous

data, the information is extracted from the configuration files and the connection is labeled as accepting asynchronous requests. In the last preparation phase, the simulation length and real-time requirements are entered and afterwards, the simulation is started. Once the simulation is finished, either successfully or failed due to some runtime error, the process will terminate. All of the Mosaik process output is logged including the return code; this information is then processed in the SGTMP.

5.2 Configuration Generation

To support various setups of Smart Grid laboratories and their equipment, the SGTMP gives the users several configuration options which can work best for their requirements. This is accomplished by letting the user define global configuration for the SGTMP containing system information and specific setup information for each test. This gives the users flexibility and thus the SGTMP can be deployed on different devices after configuration. Internally, the SGTMP converts and generates multiple configuration files for each test, as described in the subsections below. The first step is running a templating engine, which allows replacing placeholder values and substituting their real values for each test run. Another important feature is defining the SG topology for each test. Every test can have several simulators and these simulators can have multiple models; the user is responsible for specifying how these simulators and models will be created and how they will communicate with each other. Once this is specified, the SGTMP takes care of automatic connection between these entities inside of the Mosaik framework. One key addition of the SGTMP is that it allows treating both direct and asynchronous connection requests equally and abstracting away these differences also in the provided Simulation API, which gives the user better simulator re-usability for multiple scenarios and SG topologies.

5.2.1 Boundary Value Testing

Following the ISO/IEC/IEEE 29119-4 Test Techniques for testing, it was desirable to add a way for the user to specify test boundary values

and then run the same set of tests while only changing one specific value, while keeping the others constant [1]. In this framework, this is made possible by the use of `TestTemplateParameters`, where the user can specify either the boundary values (their minimum and maximum) or string literals which will be used as the test parameters. The model representation is shown in Figure 5.1.

Since each test can have multiple `TestTemplateParameters` and each `TestTemplateParameter` might yield several possibilities, all of their permutations must be generated for each test; therefore, each new test execution may produce several test runs with differing parameter values from the value domain of each template value.

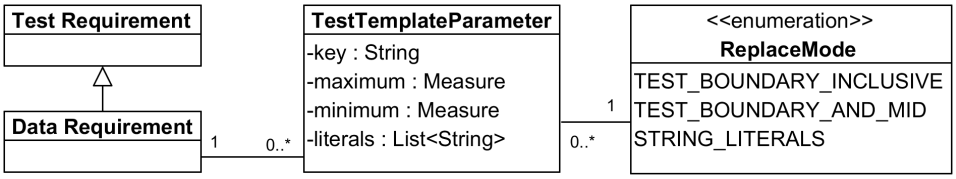

Figure 5.1: Test templating representation

5.2.2 Enhanced Simulator Connection Support

The Mosaik Scenario API supports wiring two entities by specifying the following two tuples: (source model, destination model) and (output name, input name), the details are discussed in chapter 3. One major limitation is that no data flow cycles can be introduced into the data flow multi-graph. Mosaik uses this data flow multi-graph to keep track of in what order simulators should execute their `step` function and in what order data should flow between the simulators. The user's models must make asynchronous requests in code, which brings significant inflexibility should the simulation topology change. The following subsection discusses how this test framework works around this limitation to support simulator transparent cyclic data flows.

After specifying the simulators and their models in the relevant XML configuration files, the user must specify how to handle data flows between the entities. This framework allows specifying all connections (both direct and asynchronous) using one unified format and

only specifying whether the connection is direct or indirect. The implemented simulators do not need to distinguish between receiving and sending asynchronous data as this is now handled automatically inside the `AbstractSimulator`. This is advantageous primarily because all of the data flows are explicitly specified in one file and are no longer tied to a specific simulation topology, which tightly couples two or more simulators together. Now all simulators are independent of the simulation topology and can be re-used in multiple tests. A conceptual model of how the user has to input topology data is in figure 5.2, which shows that the user has to distinguish between direct and asynchronous wire connections. However, the user does not have to distinguish between these two modes inside their own `AbstractSimulator` implementation.

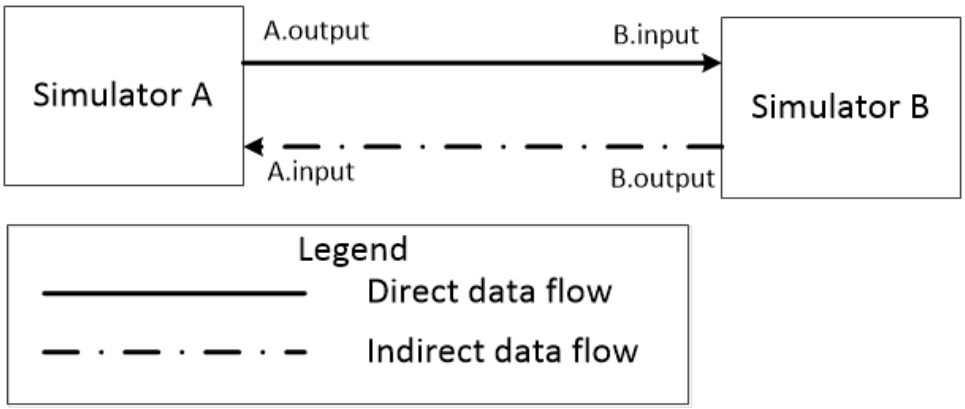

Figure 5.2: Conceptual model of how the user specifies connections between two simulators

Figure 5.3 shows what this testing framework generates automatically for the user. A new input is added to both simulators (name generated from their unique id and input names) along with a direct wire connection between them. This is done to facilitate reverse data flow between the two entities without having to use asynchronous values for the initial data exchange. After this setup, all asynchronous data will flow through the appropriate asynchronous connection automatically.

The user is currently responsible for designating connections as either direct or indirect; however, unlike when using just Mosaik alone,

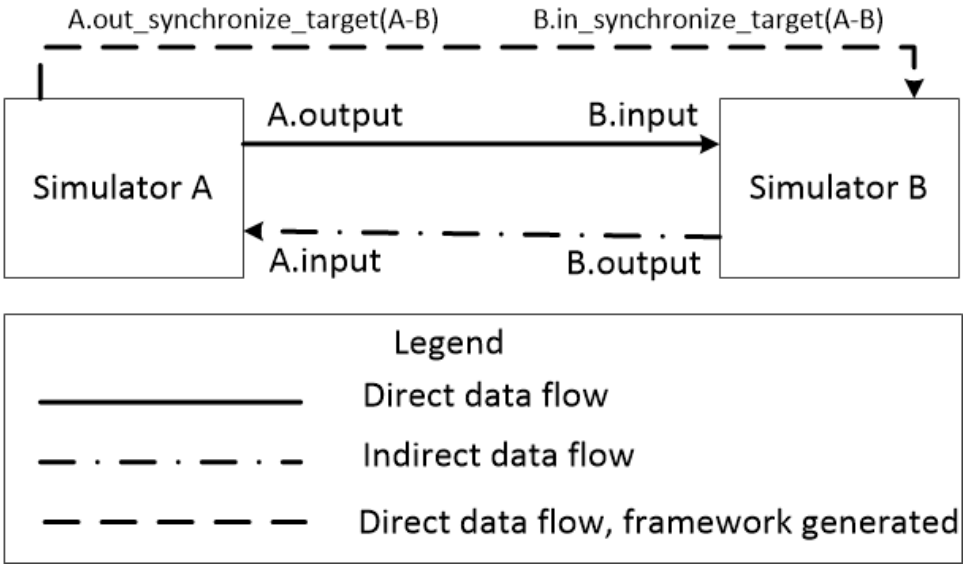

Figure 5.3: Model of the framework generated simulator connection configuration between two simulators

this platform frees the user from having to handle any of the further problems and challenges as all of the automatically generated configuration files are transparent to the user. In the new simulation API, the difference between direct and indirect connections was removed and their handling from the user's perspective is uniform. As an addition, changing the decision to make a simulator primary or secondary is only a configuration file switch and does not necessitate simulator code rewriting.

5.3 Test Execution and Result Processing

A test represents only a static view of how a test should be run, without actually executing a test no data can be gained or a device tested; a test is thus only a template from which test runs are made. It is desirable that each test can be executed under repeatable conditions, if all of the data and topology remains unchanged. Once either the laboratory technician or another part of this platform schedules a test for execution, one or more test runs are created. This section describes

what happens to each test run and how the results are evaluated both during and after each test run and simulation.

5.3.1 Test Executor

The test executor is used to initiate the creation of each individual test run from a single test. All test runs are initially in the `CREATED` state and are awaiting to be executed. The user makes a request to execute a new test and this request is then forwarded to the test executor. After creating all of the test runs, using a strategy discussed in subsection 5.3.2, the test executor then selects test run(s) which can be immediately executed. For each test run, a new Mosaik process along with any other user specified local processes are launched. Once it is confirmed that all of the processes are successfully started or have finished (in case the process merely initiates another process) the simulation can begin. All output of the processes are recorded and stored along with the test run results for later review by a laboratory technician in case an incident occurred. The test executor is also capable of generating incidents if any of the setup tasks fail, which signify to the lab technician that the test must be examined for possible configuration errors.

5.3.2 Test Run Scheduler

Once a test run is created, its status is set as `CREATED`. Given that a single test might create several test runs, depending on the amount of configuration value permutations it is not sufficient to simply start each test run as soon as it is created. In addition, each test and by extension test run has several requirements. From the scheduling perspective, the most important are the simulator requirements, which describe the simulators that must be made available for the duration of the test run. Given that it is desirable to have as many running tests as possible at the same time, test runs must be selected so that they will not attempt to use each other's resources, which could result in a deadlock or unpredictable simulation results. The Figure 5.4 shows the life-cycle of an individual test run. The scheduler is responsible for transitioning test runs into the `STARTED` stage, ideally with the best efficiency as specified by the business or laboratory metrics.

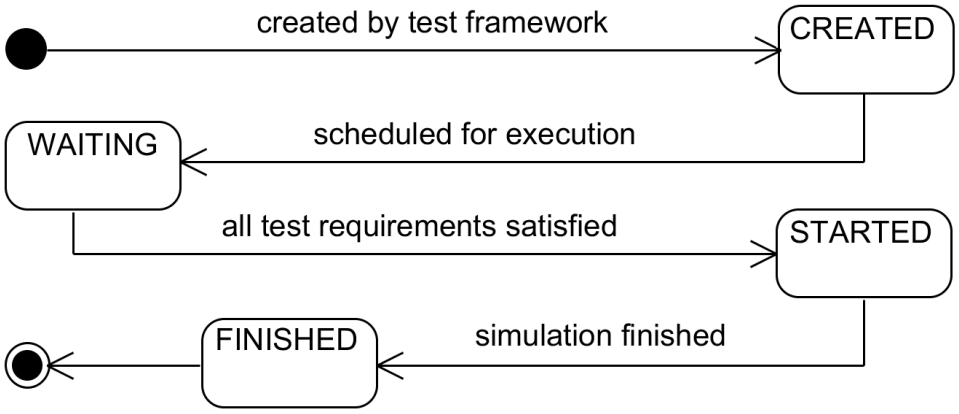

Figure 5.4: State diagram of a Test Run execution

5.3.3 Local Test Evaluator

The local test evaluator is either a separate simulator with one specific task or another model attached to an existing simulator. In both cases, the Local Test Evaluator is responsible for evaluating the state of its attached model(s) and reporting the step-by-step status as pass/fail to the Global Test Evaluator via standard wire connections. In addition, it can send measures to the Global Test Evaluator if it is desirable to have a finer reporting resolution of what happened during each test step. It is advantageous to make Local Test Evaluators part of a simulator to reduce the communication overhead and possibly access certain internal data that might otherwise have to be transmitted across simulators.

5.3.4 Global Test Evaluator

The Global Test Evaluator is a simulation entity which is responsible for the gathering of all of the inputs from local test evaluators. These inputs include their test status and current measures (if supported by given Local Test Evaluator). In each step, the state of all of the local test evaluators is observed and if all of them indicate a test pass, the global simulator will also indicate a test pass. If there are also set expected measures for a given step, they will be evaluated against the received measures from local test evaluators. Together the combina-

tion of measures being in range and test status indicates whether a simulation completed successfully or not. At the end of the simulation, the simulation results are stored so that they could be read back by the laboratory technician. Figure 5.5 shows an overview of how the Global Test Evaluator functions.

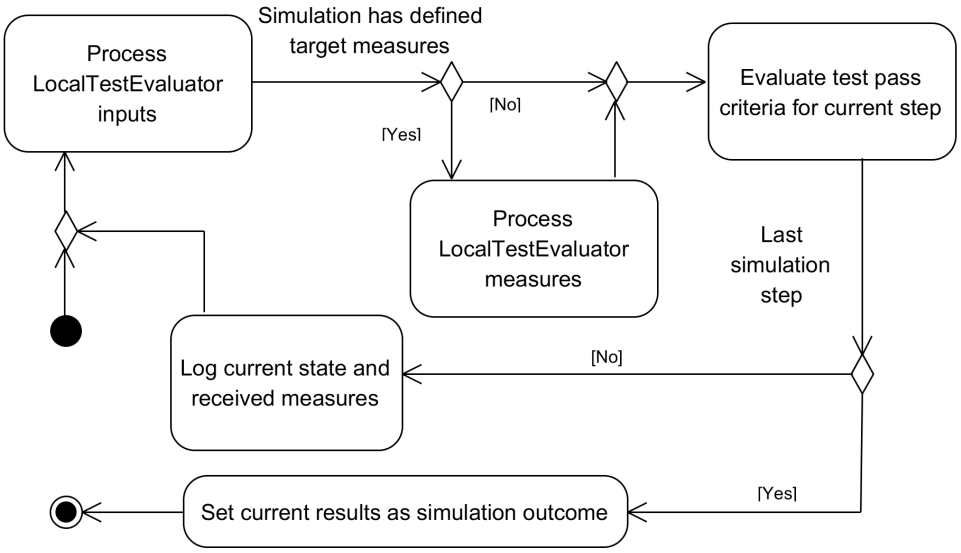

Figure 5.5: Activity diagram of a Global Test Evaluator

5.3.5 Test Result Processing

Test result processing consists of determining if any incidents occurred during the test run and reading back the Global Test Evaluator results. Incidents in this stage are created for the following reasons: an exception during test execution occurred; no result files are found, indicating that the Global Test Evaluator crashed; and/or the return code of any launched process is nonzero. Should an incident occur, the test run result is marked as OUT OF RANGE to indicate that the test did not finish as expected. If there were no incidents, the Global Test Evaluator's final results are processed and the test run state is set as FINISHED. The user can then view the results at a later time. Figure 5.6 shows the test run life-cycle in detail.

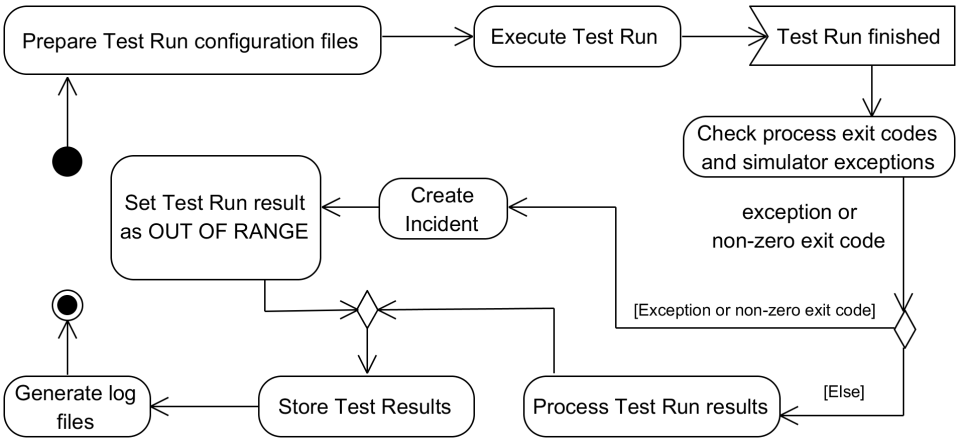

Figure 5.6: Activity diagram of a Test Run inside the framework

5.3.6 Logging

During a test run that can be running for hours or days, it is a requirement to have more information about a test run than only a simple pass/fail end result. Certain conditions may be hard to reliably recreate (combination of several factors), too time consuming to run again (e.g., fault manifests only after running a simulation for multiple days) or the equipment got destroyed in the process and the technician must determine the exact cause of this catastrophic failure from only data which survived. For these purposes, a logging framework was added to the SGTMP. There are three parts of the logging framework: simulator local logging, measure logging and Mosaik logging.

Each simulator is responsible for logging additional information that is not measure related to standard output. If the simulator was launched by the SGTMP, the log file will be directly accessible to the user. For simulators started and managed outside of the SGTMP, they must manage their log file themselves.

Logging of measurements is done by the Global Test Evaluator, which receives all of the tracked measures each step and is responsible for evaluating whether they are within the specified range for each step, as specified by the relationship between test step and expected result. In addition to evaluating measures, it also logs them. These measures are then saved to the database for future analysis, which

can be done for each simulation step. By allowing the technician to view all of the measures during a test run, it is possible to determine the state of each simulator for any given step in time.

The final logging component is from the Mosaik framework itself. If a simulation unexpectedly crashes or otherwise does not finish, this should be the first log examined by the laboratory technician. It should contain information regarding which simulators crashed, whether there were any configuration errors or whether Mosaik process itself crashed. This log is available to the lab technician immediately after the Mosaik process quits.

5.4 Examples of Usage

This section will present examples of usage to show how the Smart Grid Testing Management Platform implemented in the thesis can be used in real-world scenarios for evaluation of various Smart Grid scenarios and topologies. This example serves as a way to validate the functionality of the created platform.

The following example demonstrates a hardware-in-the-loop (HIL) and real-time (RT) SG testing scenario, in which the deployment of photo-voltaic (PV) panels can be evaluated. This example uses a Raspberry Pi, an Arduino and several hardware components to simulate a power grid. There is an active and reactive element to show how bi-directional data flows work. The aim of this scenario is to simulate a SG supplied purely by energy provided by PV panels and demonstrate how it can be used to predict power demand and power production response. The details of this architecture are discussed in [4]. The SGTMP is responsible for connecting all of the components together to create a SG topology, test orchestration, test result processing and evaluation. By using the SGTMP it is possible to create a prototype of this system faster and with more functionality out of the box compared to designing a new system or using Mosaik directly.

Each node in this scenario consists of one Raspberry Pi, Arduino, PV panel, voltmeter and a LED array. The Arduino controls the LED array and reads the voltage of the PV panel. The Raspberry Pi is running the Mosaik API implementation and is connected to the Arduino via RS-232, which facilitates data and command exchange between

these two devices. The LED array is created in such a way that several light intensity levels can be generated; this can be accomplished by combining several LED power levels. Each LED is controlled by a solid state relay (SSR), thus it is possible to switch all of the LEDs independently. These LEDs then shine on the PV panel, which generates electricity. The voltage generated by the PV panel is then read by the Arduino and transmitted to the Raspberry Pi. Every node can generate power based on the current light conditions and supply the generated power outside into the simulated power grid. By changing the LED intensities, a full day-night cycle with different weather state can be simulated. This allows modeling how these conditions will impact renewable energy power generation based on changing environmental conditions. The following describes how light profiles for each node can be generated:

> "Before the start of the simulation, an individual light-level profile is generated for each LED array. One method for generating the sunlight profiles is to use past weather data and estimate sunlight levels using heuristics, or use real world data from existing PV power stations cross-referenced with local weather at given time of day to predict future behavior. Once a model is developed for a given location, then this model allows predicting sunlight levels based on various conditions: from time of the day to changing local weather patterns. This allows simulating PV power stations spread across the country to model a nation-wide energy grid.
>
> The customized light profile is generated and transmitted to each node. Each individual node can then represent a different PV power station in the grid. Inside the node, based on the current simulation time, the [Mosaik API implementation] running on the Raspberry Pi issues appropriate light level commands to the Arduino LED controller. In turn, this modifies the output light intensity and changes the generated PV voltage, which is then measured by the Arduino." [4]

The next final chapter concludes this thesis and provides an overall overview of the results accomplished in this thesis.

6 Conclusion

The first focus of this thesis was on analyzing the software requirements for a Smart Grid testing platform. This was accomplished by doing a review of literature and summarizing their requirements. From the complexity of the Smart Grid domain, it was clear that software based testing will not be enough, and specialized processes must be used. One of these processes is hardware-in-the-loop testing, which uses real hardware parts or whole systems inside a simulation. This brings the benefits of not having to create virtualized models and real hardware can be incorporated into the simulation, thus saving development time and ensuring greater simulation accuracy.

Based on these requirements, the Mosaik co-simulation framework was selected as the simulation provider for a SG testing platform. Mosaik can incorporate a wide range of devices (simulators) into its simulation environment, thus providing great flexibility in the type of situations which can be simulated. One of the key advantages was that it can natively incorporate HIL simulators and thus enable the integration of a wide range of SG devices which can be tested as individual parts or as part of a larger system. A domain model of a platform for testing of Smart Grids was defined based on the previous requirements. This platform enables SG technicians to define the SG topology of devices to test, test parameters (e.g., length of simulation runtime, simulation speed) and test pass criteria. It also enables the operator to playback the entire simulation based on detailed logging data saved during the simulation.

The implementation of the Smart Grid Testing Management Platform (SGTMP) is detailed along with some of the technical challenges faced. An extension of the Mosaik API for integration into the SGTMP system is described along with the benefits it brings to the system integrator. The inner workings of the SGTMP are described, namely how configuration files are generated, interaction with Mosaik and test lifecycle. Finally, a complex example of usage is presented, detailing all of the SGTMP's functionality in a real-world scenario using both hardware and software simulators.

6.1 Future Work

There are several options for future work based on this thesis and the Mosaik system for the purposes of Smart Grid testing.

Currently all simulators must be managed (updated, started, restarted) outside the simulation framework if the SGTMP connects to them remotely. To solve this problem, a helper environment could be deployed on each simulator, which would take care of managing the simulator for each test run. These tasks could include automatic configuration file synchronization, log file upload after each test and unified platform related software updates.

Following the idea proposed by Nägele and Hooman in [25], the current XML based configuration of the simulation environment could be replaced by implementing a Domain Specific Language (DSL) for their configuration. The DSL could be used to generate the desired XML for the user. The advantage would be an easier to create and understand model, resulting in faster simulation development.

Currently the simulation API is only compatible with the Mosaik Java high-level API and as part of a future work, more of the supported languages could be added.

Certain concepts from the current Smart Grid Testing Management Platform could be applied onto other Smart Grid simulation frameworks; a future work could include creating support for multiple simulation frameworks to create an unified environment, in which the user could select the framework which is the most suitable for their current needs.

Bibliography

1. HRABOVSKÁ, Katarína. *Supporting a Smart Grids Laboratory: Testing Management for Cyber-Physical Systems*. 2017. Available also from: https://is.muni.cz/th/396210/fi_m/Thesis_Hrabovska.pdf. Master's Thesis. Masaryk University.
2. HEBNER, Robert. *Nanogrids, Microgrids, and Big Data: The Future of the Power Grid* [online]. 2017 [visited on 2017-04-28]. Available from: http://spectrum.ieee.org/energy/renewables/nanogrids-microgrids-and-big-data-the-future-of-the-power-grid.
3. FANG, Xi; MISRA, Satyajayant; XUE, Guoliang; YANG, Dejun. Smart grid — The new and improved power grid: A survey. *Communications Surveys & Tutorials, IEEE*. 2012, vol. 14, no. 4, pp. 944–980. Available from DOI: 10.1109/SURV.2011.101911.00087.
4. SCHVARCBACHER, Martin; ROSSI, Bruno. Smart Grids Co-Simulations with Low-Cost Hardware. In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2017, pp. 252–255. Available from DOI: 10.1109/SEAA.2017.43.
5. CHREN, Stanislav; ROSSI, Bruno; PITNER, Tomáš. Smart grids deployments within EU projects: The role of smart meters. In: *Smart Cities Symposium Prague (SCSP), 2016*. 2016, pp. 1–5. Available from DOI: 10.1109/SCSP.2016.7501033.
6. BRUINENBERG, Jan et al. Smart Grid Reference Architecture. *CEN, CENELEC, ETSI, Tech. Rep*. 2012. Available also from: ftp://ftp.cencenelec.eu/EN/EuropeanStandardization/HotTopics/SmartGrids/Reference_Architecture_final.pdf.
7. KOK, Koen; KARNOUSKOS, Stamatis; RINGELSTEIN, Jan; DIMEAS, Aris; WEIDLICH, Anke; WARMER, Cor; DRENKARD, Stefan; HATZIARGYRIOU, Nikos; LIOLIOU, Valy. Field-testing smart houses for a smart grid. In: *21st International Conference and Exhibition on Electricity Distribution (CIRED 2011)*. 2011. Available also from: http://www.cired.net/publications/cired2011/part1/papers/CIRED2011_1291_final.pdf.

8. HAHN, Adam; ASHOK, Aditya; SRIDHAR, Siddharth; GOVINDARASU, Manimaran. Cyber-Physical Security Testbeds: Architecture, Application, and Evaluation for Smart Grid. *IEEE Transactions on Smart Grid*. 2013, vol. 4, no. 2. ISSN 1949-3053. Available from DOI: 10.1109/TSG.2012.2226919.

9. KARNOUSKOS, Stamatis; HOLANDA, Thiago Nass de. Simulation of a Smart Grid City with Software Agents. In: *2009 Third UKSim European Symposium on Computer Modeling and Simulation*. 2009, pp. 424–429. Available from DOI: 10.1109/EMS.2009.53.

10. WANG, Zhifang; SCAGLIONE, Anna; THOMAS, Robert J. Generating Statistically Correct Random Topologies for Testing Smart Grid Communication and Control Networks. *IEEE Transactions on Smart Grid*. 2010, vol. 1, no. 1, pp. 28–39. ISSN 1949-3053. Available from DOI: 10.1109/TSG.2010.2044814.

11. DUFOUR, Christian; BÉLANGER, Jean. On the Use of Real-Time Simulation Technology in Smart Grid Research and Development. *IEEE Transactions on Industry Applications*. 2014, vol. 50, no. 6, pp. 3963–3970. ISSN 0093-9994. Available from DOI: 10.1109/TIA.2014.2315507.

12. LU, Bin; WU, Xin; FIGUEROA, Herman; MONTI, Antonello. A Low-Cost Real-Time Hardware-in-the-Loop Testing Approach of Power Electronics Controls. *IEEE Transactions on Industrial Electronics*. 2007, vol. 54, pp. 919–931. ISSN 0278-0046. Available from DOI: 10.1109/TIE.2007.892253.

13. JEON, Jin-Hong; KIM, Jong-Yu; KIM, Hak-Man; KIM, Seul-Ki; CHO, Changhee; KIM, Jang-Mok; AHN, Jong-Bo; NAM, Kee-Young. Development of Hardware In-the-Loop Simulation System for Testing Operation and Control Functions of Microgrid. *IEEE Transactions on Power Electronics*. 2010, vol. 25, no. 12, pp. 2919–2929. ISSN 0885-8993. Available from DOI: 10.1109/TPEL.2010.2078518.

14. BÜSCHER, Martin et al. Integrated Smart Grid simulations for generic automation architectures with RT-LAB and mosaik. In: *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 2014, pp. 194–199. Available from DOI: 10.1109/SmartGridComm.2014.7007645.

15. SCHLOEGL, Florian; ROHJANS, Sebastian; LEHNHOFF, Sebastian; VELASQUEZ, Jorge; STEINBRINK, Cornelius; PALENSKY, Peter. Towards a classification scheme for co-simulation approaches in energy systems. In: *2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST)*. 2015, pp. 516–521. Available from DOI: 10.1109/SEDST.2015.7315262.

16. DEDE, Jens; KULADINITHI, Koojana; FÖRSTER, Anna; NANNEN, Okko; LEHNHOFF, Sebastian. OMNeT++ and mosaik: Enabling Simulation of Smart Grid Communications. In: *2nd OMNeT++ Community Summit, IBM Research*. 2015. Available also from: http://arxiv.org/abs/1509.03067. arXiv: 1509.03067.

17. BOUISSOU, Olivier; CHAPOUTOT, Alexandre. An Operational Semantics for Simulink's Simulation Engine. *SIGPLAN Not.* 2012, vol. 47, no. 5, pp. 129–138. ISSN 0362-1340. Available from DOI: 10.1145/2345141.2248437.

18. SCHERFKE, Stefan; NANNEN, Okko; EL-AMA, André. *The simulator manager — mosaik 2.3.1 documentation* [online] [visited on 2017-10-04]. Available from: http://mosaik.readthedocs.io/en/latest/simmanager.html.

19. METS, Kevin; OJEA, Juan Aparicio; DEVELDER, Chris. Combining Power and Communication Network Simulation for Cost-Effective Smart Grid Analysis. *IEEE Communications Surveys Tutorials*. 2014, vol. 16, no. 3, pp. 1771–1796. ISSN 1553-877X. Available from DOI: 10.1109/SURV.2014.021414.00116.

20. SCHERFKE, Stefan. *The mosaik API — mosaik 2.3.1 documentation* [online] [visited on 2017-09-16]. Available from: http://mosaik.readthedocs.io/en/latest/mosaik-api/.

21. SCHERFKE, Stefan; NANNEN, Okko; EL-AMA, André. *How mosaik communicates with a simulator — mosaik 2.3.1 documentation* [online] [visited on 2017-10-04]. Available from: http://mosaik.readthedocs.io/en/latest/mosaik-api/overview.html.

22. SCHERFKE, Stefan. *Discussion of design decisions — mosaik 2.3.1 documentation* [online]. 2017 [visited on 2017-09-16]. Available from: http://mosaik.readthedocs.io/en/latest/dev/design-decisions.html. Accessed: 2017-09-16.

23. SCHERFKE, Stefan. *Scheduling and simulation execution — mosaik 2.3.1 documentation* [online] [visited on 2017-09-16]. Available from: `http://mosaik.readthedocs.io/en/latest/scheduler.html`.

24. BÜSCHER, Martin et al. Integrated Smart Grid simulations for generic automation architectures with RT-LAB and mosaik. In: *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 2014, pp. 194–199. Available from DOI: `10.1109/SmartGridComm.2014.7007645`.

25. NÄGELE, Thomas; HOOMAN, Jozef. Rapid Construction of Co-Simulations of Cyber-Physical Systems in HLA Using a DSL. In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2017, pp. 247–251. Available from DOI: `10.1109/SEAA.2017.29`.

A Appendix

A.1 Testing Management Dashboards

The current primary entry point to the SGTMP is via its API; however, for user friendliness a web interface is also implemented for some of the functionality. This web interface allows a laboratory technician to upload test definitions, which contain the Smart Grid topology and test pass criteria. They can then schedule tests for execution based on the current requirements. Once the tests are completed, the technician can inspect their results and try to determine the cause of any incidents. Screenshots of this web interface are below.

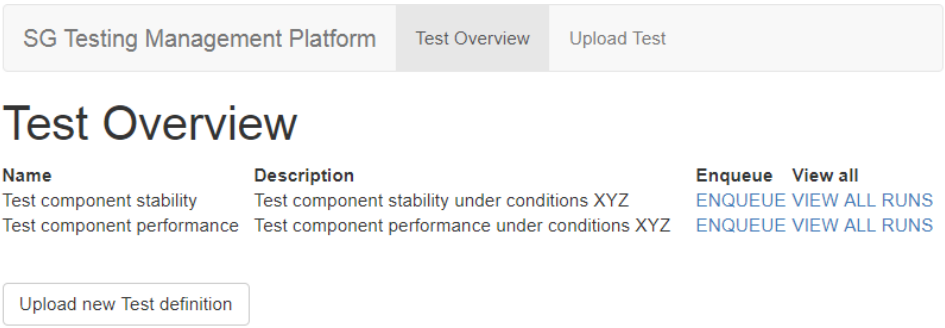

Figure A.1: Test definitions overview

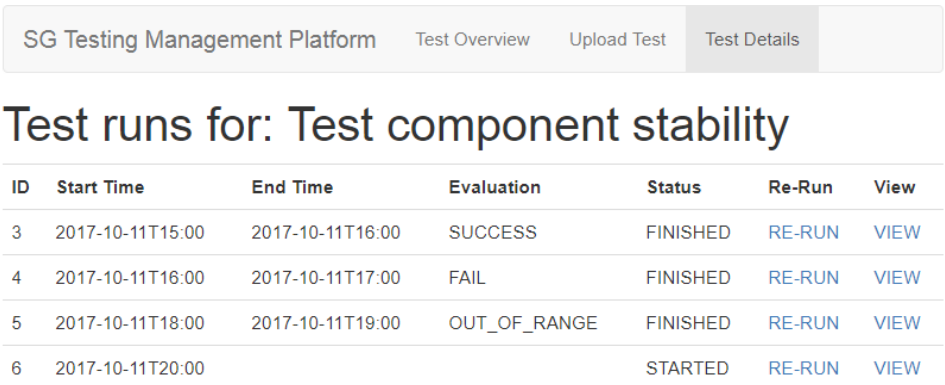

Figure A.2: Overview of Test Runs for Test

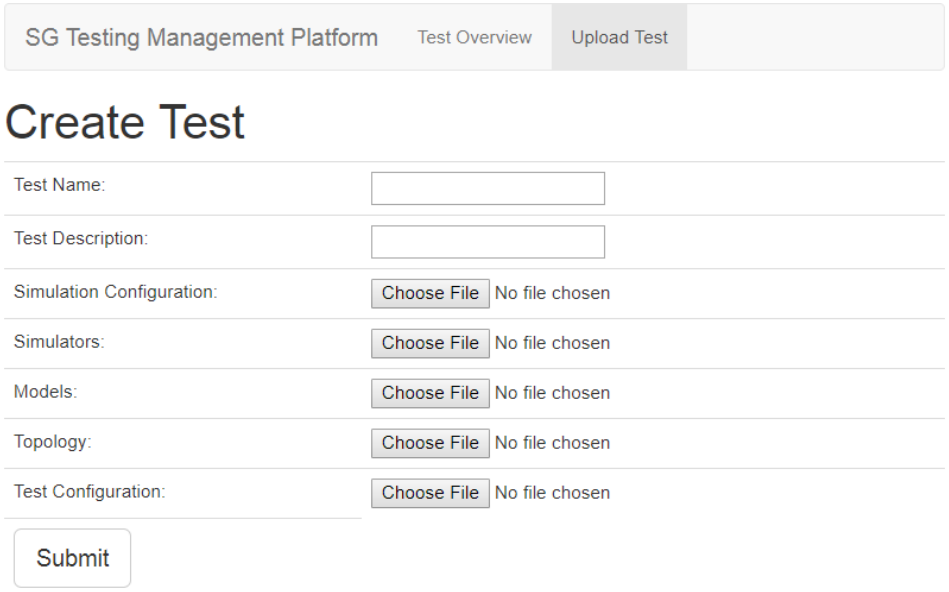

Figure A.3: New test definition creation

A.2 Attachments

The archive of this thesis contains the complete source code of the SGTMP and the full domain model diagram from section 4.2. There is a Dockerfile[1] which can be used to setup a working demonstration environment.

1. `https://www.docker.com/`