Towards Efficient Co-Simulation of Cyber-Physical Systems

by

Talha Ibn Aziz

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

A cyber-physical system (CPS) is composed of interacting physical and computational components. CPS research addresses the combined sensing and actuating to monitor and control a CPS. Simulation techniques are used in advance of costly CPS deployments to provide sufficient understanding of the complex interactions within a CPS, but such simulations involve the simulation across multiple domains. High fidelity simulators for specific domains already exist, but are not built to interact with other simulators. To address this issue we adopt a co-simulation approach. Co-simulation requires a means of interaction and coordination among multiple simulators, guaranteeing that each simulator is informed of changes in the state of the other simulators that could influence its own state, and that events across multiple simulators do not violate causality. We adopt MOSAIK as the co-simulation framework and we illustrate how existing simulators are adapted to interface with the framework. We produce examples of co-simulation scenarios to capture a class of CPS problems, namely, smart-grid applications. The legacy simulators used, and partly modified, to interface with MOSAIK are OpenDSS, a power flow simulator for electrical grids, and ns-3, a data networking simulator. We address the problem of the joint configuration of the simulators using a combined ontology expressing the union of the simulated domains. We evaluate the performance of our co-simulation by simulating large scale smart-grid applications. The results demonstrate that with some additional programming effort, refinements in the handling of events within each individual simulator can improve the overall co-simulation efficiency.

To my wife - Tasneea Hossain,
For being there for me the whole time with patience and tremendous support.
I also dedicate this to my family and friends for all their help and support.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Professor Ioanis Nikolaidis for his help throughout my years of study, this would not be possible without him. I would also like to thank Professor Omid Ardakanian for his guidance and support, my committee chair Professor Herbert Yang, and committee member Professor Ken Wong for their valuable advice and feedback. I would like to express gratitude to my colleague Jihoon Og for helping me generate the ontology for my work.

The funding for this work has been provided by Future Energy Systems (FES) of the University of Alberta and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We define as cyber-physical system (CPS) any system in which an environment composed of physical components, is sensed and, often, actuated upon, by a computational and data communication infrastructure (the *cyber* part). Here physical components stand for anything that exhibits dynamics and behaviors that are understood not to be the result of mechanistic computation. Examples of physical components that exhibit dynamics are, e.g., the heat transfer phenomena in buildings, the human preferences in picking routes when driving, etc. In the heat transfer phenomena, the dynamics are governed by heat flow physics. In the case of route selection, the dynamics are driven by human behavior. In both examples, the dynamics typically happen outside a computational infrastructure, and hence are independent from whether a computational infrastructure exists or not. The computational aspect may include any type of processes, such as e.g., controllers in the sense of legacy control systems, and computation may involve interaction with users via user interfaces, or be completely distributed, autonomous, and with no humans in the loop.

1.1 CPS Simulation

Developing and deploying the sensing, actuating, computational, and communication elements of a CPS requires that we correctly capture the interaction between the physical and the cyber parts. The complexity of understanding those interactions is often handled by running, possibly long, simulations. The need to run many, efficient, and accurate simulations of a CPS has received

attention also because of the interest in machine learning techniques. When interacting with a physical environment, it is both economical, efficient, and often safer, if a learning system is interacting with a simulated physical environment.

Simulations are calculations imitating real-world operations performed using well-defined models for real-world objects [1]. Simulations were first carried out during World War II by John Von Neumann and Stanislaw Ulam [2], to avoid exorbitant costs of studying neutron behaviour with real-world experiments. The success of the experiments made simulations widespread in industry and business [3]. With the advent of computers, more detailed simulations became possible, which led to the emergence of putting together simulations using multiple simulators, namely *co-simulation*.

1.2 Co-Simulation Challenges

The challenge with simulating a CPS, is that each particular CPS may involve a different set of simulators, as its constituent parts depend on what kinds of physical systems the CPS includes. This is true of co-simulation in general. Ordinarily, the simulators available are developed by groups of researchers and practitioners with special interest in a particular domain, e.g., heat flow dynamics. While they devote a lot of detail and time for the specific domain, they have been traditionally uninterested in the integration of their simulator with other simulators. The focused development of a simulator for a specific domain has some notable benefits. Notably, community developed and supported simulators, have received the scrutiny and the continuous use by a user community that is very likely to have resolved most critical bugs and added a rich set of features.

Observation 1: It is preferable to develop co-simulations using as much as possible existing ("legacy") simulators which a large community of users and developers have debugged, evolved and used, rather than develop co-simulators "from scratch."

Since simulators are developed in isolation from other simulators, inte-

grating them in a co-simulation environment involves the task of identifying how their implementations can be modified to allow for such integration. In short, the simulator internal logic has to be "opened" to external control and to, equally, be able to *cause* external events which were not part of their original design. The need to modify the logic, assumes access to the source code of the simulators.

Observation 2: Open source simulators facilitate their integration into co-simulations because their software is readily available and modifiable.

Interfacing of the simulators with each other, even if facilitated by all of them being open source, is not necessarily a trivial matter. For example, each of the simulators may be authored in a different programming language. The integration of the various simulators may also be impossible to form a single executable process. Strategies for transforming the communication between simulators to language independent inter-process communication is needed. By virtue of the separation of each constituent simulator into a different process, co-simulation incorporates aspects of distributed simulation. As we will see the existence of co-simulation frameworks provides (some) of the answers to the task of integration. An ideal framework provides adequate flexibility to integrate disparate and drastically different simulators, providing the "glue" necessary among them. Yet, to interface and work as expected by a given framework, each constituent simulator needs to be modified.

Observation 3: Co-simulation frameworks offer, to various degree of flexibility, assistance in building co-simulators, but do not necessarily eliminate the effort of modifying existing simulators.

Regardless whether a framework is used, an additional issue is the joint configuration of the constituent simulators. Each simulator needs its own specific configuration. It is usual practice for simulators to read a configuration file when they start their execution. Configuration files are conceived around the definition of objects and their relationships that make sense for

the specific domain simulated by the simulator. For example, a simulator for heat transfer phenomena may need to know the thermal resistance and the geometry of walls, and it is this information that is included in the particular simulator configuration file. However, when viewed abstractly, the joint configuration of the co-simulator can result in the same object appearing in more than one constituent simulators, having a different role to accomplish in each of them. For example the simulation of an actuator may appear to a power grid simulation as a switch while in the data communication side it is a communication endpoint receiving packets encoding commands. To avoid duplication of information which can be causes of errors, the joint configuration should include a single description of each object, even if it is participating in more than one simulator. Additionally, there may be objects that have a correspondence only in one of the simulators.

To address the issue of a single joint configuration, there may be a need to invent a new configuration language that allows us to express relation using an ontology that spans all the domains simulated by the constituent simulators. From the joint configuration, individual simulator configuration should be produced. The existence of the joint configuration and its compilation to individual per-simulator configurations also assists in adopting naming schemes for the various objects across all simulators that is coherent and consistent across all simulators, thus helping with tasks such as debugging the simulation across the multiple simulators.

Observation 4: A useful element of a co-simulation environment is a joint configuration from which all constituent simulators should be configured in a consistent and coherent manner.

Note that the co-simulator framework choice and the joint configuration choice are, in principle, orthogonal issues. One can imagine that a co-simulator framework upon starting, invokes a tool to extract the individual configurations from the joint configuration and then, upon instantiating each constituent simulator, it passes to them the particular configuration each needs. That is, the framework's involvement can be as little as passing a specific (compiled) configuration to each simulator it starts.

Finally, there is scant information about the performance of co-simulators and co-simulation frameworks. There are understandable reasons for this phenomenon. First, the concrete performance depends on what is being simulated, e.g., the number of objects, how frequently they interact, etc. There are no standardized benchmarks across multiple domains that would allow a systematic comparison of how different co-simulators across the same combination of domains perform. To this end, the thesis approaches the performance evaluation from the point of view of the percentage of time and the kinds of interactions happening among the constituent simulators and the framework, as a relative view of performance. Because one of the simulation domains (the power grid) is often configured using certain standard power grid topologies, we adopt the same topologies as a first point of standardizing the performance evaluation of the co-simulation environment for future studies.

Observation 5: There is a lack of performance results for co-simulations and co-simulation frameworks involving combinations of simulators that capture specific simulated domains.

1.3 Application: Smart Grid Co-Simulation

The purpose of this thesis is to describe the integration of a co-simulation environment for a specific CPS application, namely for *smart grids*. The choice of smart grid simulation is due to the project needs of the University of Alberta Future Energy Systems (FES) which funded most of the research carried out in this thesis. Smart grids are an excellent testing ground for CPS co-simulation because they include at least three simulators: (a) a power grid simulator capturing the physics of electrical current flowing through the interconnected electrical components of the grid (i.e., the *physical* aspect of the CPS), (b) a simulator for the sensors installed at various points on the power grid, and (c) a simulator of the communication network and processing required to process the sensed data. Since seldom is the interest on smart grids restricted to just observing the power grid dynamics, a fourth simulator is also present, (d) a simulator for the actuation performed on the smart grid, e.g.,

changing the tap positions of transformers, opening and closing switches, etc. The smart grid examples (co-)simulated in the thesis involve two well-known simulators, OpenDSS and ns-3, corresponding to (a) and (b) respectively.

Observation 6: The co-simulation of smart grids is a good testing ground for CPS co-simulation because of the availability of good legacy simulators for power flow and data networking.

1.4 Thesis Structure and Contributions

We organize the remainder of this thesis as follows. The next chapter introduces the concepts related to co-simulation and details the current literature on co-simulation frameworks. We mention examples of notable frameworks and platforms, introduce our platform of choice and its literature, and the constituent simulators used in our research. In chapter 3 we define and detail the working principles of the constituent simulators, their internal components, and the connections formed between these components for the purpose of co-simulation. Furthermore, we describe the temporal event management of the co-simulation and the various refinement techniques we propose to improve said management. In chapter 4 we describe the various simulator configurations, define an ontology to manage the configurations and to facilitate a central configuration control. We then describe how to initiate the co-simulation and present the interfacing within the simulators to facilitate control by the co-simulation platform. We develop two CPS scenarios to evaluate the performance of our platform and we present the results of their simulations in chapter 5. Chapter 6 concludes the thesis while mentioning the notable limitations of the framework and its possible future enhancements.

In summary, the contributions of this thesis are:

- the development of a novel high-fidelity co-simulation platform utilizing standard open-source simulators and a flexible co-simulation framework to simulate smart-grid applications,
- the introduction of techniques to improve the efficiency of complex simulations of cyber-physical systems and co-simulations in general while

maintaining accuracy of causal order,

- the production of benchmark runs from the co-simulation of applications of smart-grid technology on large scale well-known power network topologies, and,
- the contribution towards an ontology to facilitate centralized, flexible, and consistent management of simulator configurations. The ontology part is joint work with Jihoon Og, at the University of Alberta.

Chapter 2

Related Work

Simulation is a vast topic that contributed in many research, technological, and commercial fields. The reviewed related work focuses on the topic of co-simulation as it has evolved over recent decades, and also to the systems that have been proposed or built as frameworks to enable co-simulation. We focus on one of the frameworks which is of a more recent vintage and has been claimed to properly address smart grid simulations. In the related work we also present the constituent simulators that were used in the developed co-simulation environment, as their structure and features influenced the way by which they were integrated into the co-simulation.

2.1 Introduction to Co-Simulation

Co-simulation is achieved by integrating multiple constituent simulators, each capable of simulating a restricted domain of the overall simulated system. Co-simulation, as a technical discipline, is interested in how the simulators communicate and how they progress together through simulated time, in a consistent manner, to produce correct results.

2.1.1 The Necessity for Co-simulation

A collection of discrete event simulators, each proceeding independently by generating and processing internally events along a timeline, may simulate the various facets of one phenomenon. For example, a simulator for the vehicular mobility and one for the computer communication network. A tempting, but naïve, solution is to use one simulator to produce the outputs that can be

ingested by the other simulator(s). That is, one simulator runs *in its entirety*, produces output and its output becomes input for the other simulator(s). For example, the mobility simulator may produce the locations of vehicles across time, and then their locations may be ingested by the network simulator to simulate, e.g., the communication network congestion caused by the different degrees of density of vehicles, depending on the vehicular traffic dynamics (period of traffic lights, speeds, etc.). While a number of studies are well served by the above synthesis of simulators, there are plenty of examples where the approach is insufficient.

Consider, for example, the case of a system where we wish to study a smart transportation network where a communication network delivers messages to/from vehicles about how they should re-calculate the paths they are driving on. A circular dependency exists between a vehicular traffic and the communication network simulators. Imagine for example the state of the vehicular traffic simulator where the simulated vehicles follow the least congested roads, and at some point during the simulation, congestion develops in certain road segments. Then, let's assume, a communication network simulator simulates the wireless transmissions and reception of messages among vehicles. A transmitted message is considered received by nearby receivers. The communication simulation may be introducing factors such as message delivery delay, degree of successful deliver of the message, etc. Given the received messages and what they contain as information, the vehicles in the vehicular simulator may recalculate the paths they are travelling on. This change of paths followed by the vehicles, will, in turn, change the adjacency of the vehicles and hence of which receivers can receive which transmissions. One simulator feeds back to the other(s), and the naïve interaction between simulators described in the previous paragraph is no longer expressive enough to capture this use case. A related concept to this point is that of *federations of simulators*, i.e., collaborating simulators, each of them simulating a different domain of the system under study. Generally we are not concerned with how the individual simulators operate. For example they can be executing sequentially or in parallel. The most important factor to address is how to synchronize the simulators, in that each one of them is no longer exclusively depending on

its own internal event generation but also events caused by other simulators. The global view of the progress of events being simulated across all simulators *must not violate causality*, i.e., the calculated *total system state* at any point in time is only dependent on the previous state in temporal order. Each simulator is holding part of the total system state, e.g., the locations of each simulated vehicle, how many packets are being transmitted on which links in a communication network simulator etc. The union of the state across all constituent simulators is the *total system state*.

2.1.2 Simulator Synchronization

Assume a common representation of the simulated time by the constituent simulators and that at simulated time t_0, the next event in temporal sequence to be processed by simulator A is at time $t^A_{next} > t_0$ and the next event to be processed by simulator B is at time $t^B_{next} > t_0$. To clarify: both simulators may also have other events to process in the future but later in time than their t^i_{next}. That is t^i_{next} denotes the *earliest* of their future events. Also note that simulator B does not know t^A_{next}, neither does simulator A know t^B_{next}.

Let us consider the case where t^A_{next} is later than t^B_{next}. Consistency requires that we first simulate t^B_{next} i.e., in temporal order of simulated time. However, the simulation of the event related to t^B_{next} by simulator B could cause other events in simulator B in the future (i.e., $> t^B_{next}$) and *also* could cause events (since the two simulators interact with each other) in simulator A. Those events caused by the event that executed at t^B_{next} in B could occur in A *earlier* than t^A_{next}. This is not just a case where the events that are caused by B to A are just "inserted" in the event list of simulator A. The nature of those events could drastically *alter* even events that are already in A's event list. For example the event that was supposed to execute at t^A_{next} in A could be a timeout. The timeout logic could be reset if an event, as the ones caused by B to A, happens earlier than t^A_{next}. That is, all events in the event list of interacting simulators could be tenuous at best. The one that certainly will be, and should be, simulated, earlier than all of them, across all simulators is the one with the earliest possible timestamp, i.e., the one at $\min_i t^i_{next}$ across all i simulators.

The above form of synchronization is called *conservative* [4] as it never allows the constituent simulators to execute events that could violate the causality order. Technically speaking, it is not the events that violate the causality order, but rather their execution that causes state changes which run contrary to causality. That is, we enforce an order that ensures that the state at time t is never derived from state at a future time $t' > t$[1]. A consequence of this approach is that only one of the simulators is executing at any point in time - the one handling the earliest, across all simulators' future events. Namely *"A conservative simulator executes an event only when it has a nontrivial lower bound on the timestamps on all event messages that will arrive in the future."* The nature of the "nontrivial" lower bound is explained in Section 2.1.3.

Other aspects of the integration are related to the simulated time across the various constituent simulators. For example, one simulator may track time as floating point and another one as integers. To the extent that the units of simulated time for the simulators are in a linear relation to each other, they can be assumed to be the same, when in reality what this means is that the timestamps of one simulator can be easily transformed to the timestamps of another simulator through multiplication by a constant factor (different for each pair of simulators), plus the addition of a constant, offset value (if the timestamps do not have the same *time zero* point of reference). Finally, we distinguish two kinds of time: simulated time, i.e., the time whose flow describes the simulated system, and wall-clock time, i.e., the time taken for *executing* the simulator. The two are not in any specific relation. The wall-clock time tells us how long the simulation is taking to execute. The simulation time tells us how long was the period of time of the system being simulated. To avoid repetition, in the rest of the paper "time" will refer to simulated time, and we will only qualify wall-clock time if we are referring specifically to the time it takes to execute simulations.

[1]The case of equality $t' = t$ is a special one and is discussed in Section 3.6.1.

2.1.3 Simulation Termination

The statement about nontriviality covers instances where $t^i_{next} = \infty$ for some simulator i, to capture the cases of simulators that, if they were executing alone, they would have to terminate, since there would no longer be events in their event list. However, because they are interacting with other simulators, they cannot terminate pending possible externally caused events from other simulators. Only when $t^i_{next} = \infty$ for all i can the simulation across all simulators be considered as terminated for lack of further events. Normally, the termination of the combined co-simulation occurs when all components have no events in their event list, i.e., $t^i_{next} = \infty$. There are however more varieties for *termination criterion* than just event list exhaustion across all simulators. One termination criterion can be that of reaching a specific simulation time, i.e., if the concerned simulator advances past the termination time, or it can be a logical condition, e.g., X number of things of a certain type have been processed. Assuming that one simulator of the federation of simulators "hits" the termination criterion before the other simulators, we have to:

(a) Terminate the simulation of the other simulators at exactly the same time point, i.e., not terminating them at different points of simulated time.

(b) Deal with the events pending at the other simulators so that the statistics we derive from them are "stopped" at the time point of termination.

For scenario (a), if the simulators are not terminated at the same time, they might keep producing events indefinitely on their own and keep executing them. In scenario (b), while we do not execute the events of the other simulators as that would again lead to indefinite event production (i.e., some events would produce other events), we would need a way to account for the partial state of the simulators, to the statistics we collect. For example, a simulator might be terminated externally in a "forced" manner because another simulator has terminated. This would require sorting of dependencies, such that only dependent simulators are "forced" terminated by independent simulators. A simulator is said to be dependent on another simulator, if the former requires input from the latter. Terminating dependent simulators before the

simulators they are dependent on - might be causally impossible as the latter generates events for the former.

2.2 Co-simulation Frameworks

Several frameworks have been developed in order to achieve co-simulation using simulators of different domains. However, our focus is on the field of cyber-physical systems (CPS) with smart-grids as one of its more well-known specializations [5]. Even before CPS, simulations requiring complex and co-operating integration of simulators due to their interdisciplinary nature, led to the emergence of *co-simulation frameworks*. One of the integral parts of these frameworks is a middle-ware that is responsible for data exchange among the constituent simulators and the temporal synchronization of the different simulation models. The de-facto standard which defines the specification of this middle-ware is the High Level Architecture (HLA) [6]–[8].

2.2.1 High Level Architecture (HLA)

The High Level Architecture (HLA) provides a specification for a common technical architecture for use across all classes of simulations developed by the US Department of Defence [9]. Although initially developed with the intention to have wide applicability across the full range of defense simulations, it is now an open project and is used for research purposes across all domains. The HLA is not a set of programs in any programming language but rather provides the basic rule-set for simulator interoperability. Therefore, as technological advancements became available, new and different implementations were possible within the framework of the HLA.

Functional Components

There are three major functional components of HLA that interact with each other as shown in Figure 2.1.

1. **Federates:** A federate is a comprehensive term for a simulator as it may include a computer simulation, a manned simulator, analytical data collectors or viewers, etc. Athough the definition of a federate is

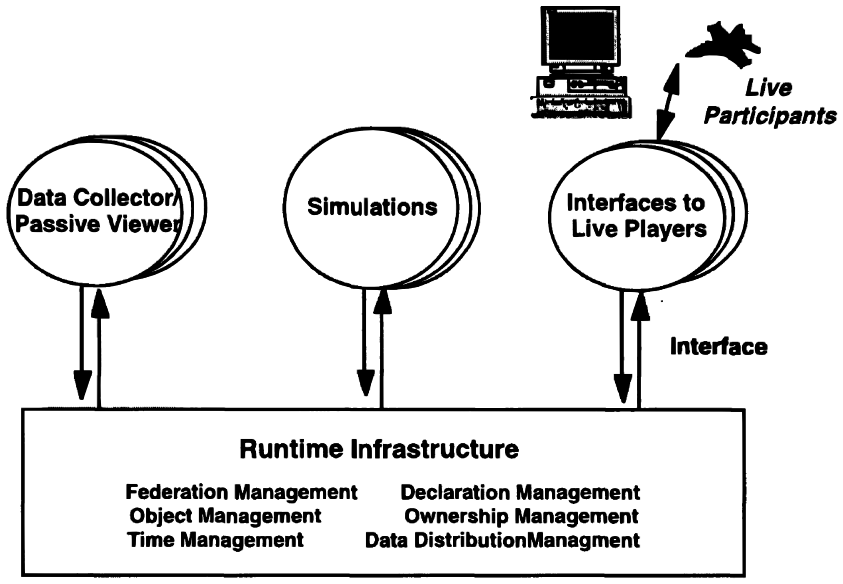

Figure 2.1: Functional Overview of HLA components as it appears in [9].

not restrictive, all federates must incorporate specified capabilities to allow objects in the simulation to interact with objects of other simulators through the Runtime Infrastructure (RTI).

2. **Runtime Infrastructure (RTI):** The RTI is a distributed operating system which manages the interactions between the simulators while maintaining the time synchronization required for chronological ordering of events.

3. **Runtime Interface:** The Runtime interface provides a standard interface that all federates need to follow when interacting with the RTI. This helps the RTI understand what to expect from the federates and how to command certain aspects of their simulation.

The RTI provides various services like creating and managing federate instances and object instances of each federate, time management of the federates, and efficient data collection and distribution among the federates. For our purposes, we describe the time management of HLA in greater detail.

Time Management

Time management of HLA is concerned with the mechanisms for controlling the advancement of each federate in simulation time. This is done in co-

ordination with object management of HLA to ensure timely delivery of data to federates i.e., synchronization to maintain causality. There are different kinds of advancements based on how simulation time paces with respect to wall-clock time. However, HLA uses *unpaced, coordination time advances*, which simulates the federates as fast as possible without any regard for wall-clock time, while maintaining time synchronization. The key components of the time management services of HLA are:

1. *Logical time:* This is the simulation time of the federates, which can also be referred to as *federate time*. Therefore, at any given time of execution, different federates may be at different logical times.

2. *Advancing logical time:* Each federate can only advance their time with the permission of the RTI time management service. Therefore, there must be mechanisms which allow the HLA to control the advancing of the logical time of the federates.

3. *Message synchronization:* Any federate can only advance its logical time when it has received all possible messages from other federates within the mentioned time. This ensures that the federates can proceed without any concern for incoming messages as there will be no additional ones, at least until the time to which it must proceed.

The time value during which a federate is guaranteed to not receive any messages from other simulators is called the *lookahead* value. This constraint allows the federates to advance their time efficiently without much inter-federate interaction. In order to calculate the lookahead value, the RTI must internally compute a lower bound on the time stamp (LBTS) of future messages that will be later received by the federate in question.

2.2.2 Co-simulation Platforms

Following the standardization of HLA in 2000 by IEEE for modeling and simulation [10], the HLA capabilities have expanded with many co-simulation platforms developed following the standard. One of the domains in recent years, which require such intricate co-simulation is the CPS simulation [5].

Real-time experimentation with large scale smart grids has proven to be challenging due to lab tests being too costly in terms of needed equipment and painstaking to set up. Furthermore, the complex nature of CPS simulations, of which smart grid simulation is an example, requires well-defined platforms and interfaces to allow for the gradual integration of more simulators that capture increasingly more facets of the cyber-physical environments. Therefore, platforms implemented based on the HLA framework have become a viable candidate for CPS research.

Early work in relevant co-simulation platforms focused primarily on the analysis of the interaction between power systems and communication networks [11]–[13]. Gradually, more generic co-simulation approaches called *co-simulation frameworks* became available.

Notable Frameworks

Besides the High-Level Architecture (HLA), other frameworks have been proposed throughout the years. A notable example is the Ptolemy project [14], which started as a framework focusing on *hierarchical heterogeneity*, as opposed to *amorphous heterogeneity*, to solve the problem of modeling interaction between heterogeneous sub-systems [15]. *Amorphous heterogeneity*, as they called it, is used by other frameworks (e.g., HLA) to provide an abstract and generalized definition for models to allow for fitting of ever-changing component simulators. The project evolved over time to address the design, study, and simulation of complex systems and the interaction between concurrent internal components. The framework became known as the *Ptolemy Classic*, with the focus of the project being shifted towards developing a Java-based platform following the proposed framework, known as PTOLEMY II [16]. The major challenge addressed by Ptolemy II is the mixture of heterogeneous models with hierarchy, to allow for specific modelling and interactions as well as reuse of components.

Another notable and more recent framework is the Functional Mock-Up Interface (FMI) [17], which provides a common interface by defining abstract functions to be implemented by every component simulator. FMI defines a standard container and interface to exchange dynamic models. The models

are a combination of XML files, binaries and C code, called the Functional Mock-Up Unit (FMU). The common application programming interface (API) is used by a simulation environment to generate one or more instances of an FMU and simulate them with other models. FMI allows flexibility to integrate simulation models in a distributed and parallel way [18], [19], thus providing speed-up over conventionally single-threaded simulation approaches.

Notable Co-simulation Approaches

Several co-simulation approaches have been developed in the last few decades (refer to Table 2.1). We focus on the approaches for smart-grids. A few of the notable smart-grid co-simulation approaches are as shown on Table 2.1.

	Power Flow Simulator	**Network Simulator**	**Simulation Framework**	**Time Management**
GECO [20], [21]	PSLF	NS-2	Ad-hoc (TCL linking)	Global event-driven
INSPIRE [12], [22]	DIgSILENT PowerFactory	OPNET Modeler	IEEE 1516-2010 (HLA evolved)	Dynamic time stepped
EPOCHS [23]	PSCAD/ EMTDC PSLF	NS-2	IEEE 1516-2010 (HLA evolved)	Fixed time stepped
ADEVS [24]	ADEVS	NS-2	Ad-hoc (ns-2 integration)	DEVS
VPNET [25]	VTB	OPNET Modeler	Ad-hoc (sockets)	Time stepped
GridSim [26]	Powertech TSAT	GridStat	Ad-hoc	Fixed time stepped
PowerNet [27]	Modelica	NS-2	Ad-hoc (named pipes)	Time stepped
[28]	NETOMAC	NS-2	Ad-hoc (JNI)	Time stepped
[29], [30]	OPAL-RT	OPNET SITL	Ad-hoc, emulated, sockets	Real-time
Greenbench [31]	PSCAD	OMNeT++	Ad-hoc (IPC)	Global event-driven

Table 2.1: State-of-the-art co-simulation approaches adapted from [32].

EPOCHS [33]: One of the first known simulators which combined power and communication systems is the Electrical Power and Communication Synchronization Simulator (EPOCHS). The concept of federated dynamic simulation utilized by EPOCHS integrates three component simulators namely, PSCAD/EMTDC, PSLF, and NS-2. PSLF performs large-scale power system stability simulations (RMS), NS-2 simulates the communication network, and

PSCAD/EMTDC simulate transient protection with short term time domain responses (EMT). A Run Time Interface (RTI) exchanges data periodically by interfacing and synchronizing the simulators. However, the synchronization points are fixed and pre-programmed. Increasing the number of synchronization points increases precision at the cost of efficiency.

ADEVS [24]: A Discrete EVent System simulator (ADEVS) is a C++ library for hybrid dynamic systems which uses the Zeigler's Discrete Event System Specification (DEVS) [25] as its base. The time management and interaction between the constituent simulators are handled following the specifications of DEVS. A Toolkit for Hybrid Modeling of Electrical power systems (THYME) within ADEVS is used to simulate the power systems, which is coupled with control algorithms and communication networks. Examples of co-simulation with communication networks are using NS-2 [24] and OMNeT++ [34]. The interaction between discrete event and continuous time sub-systems are modelled by encapsulating the continuous time dynamics within a discrete event model.

GECO [20], [21]: The Global Event-Driven Co-Simulation Framework (GECO) performs co-simulation using PSLF as the power system simulator, and NS-2 as the communication network simulator. As the name suggests, the co-simulation is done using a global event-driven framework. The global event scheduler maintains a global event queue, where the time-tagged discrete events of both simulators are combined.

INSPIRE [12], [22]: The Integrated co-Simulation of Power and ICT systems for Real-time Evaluation (INSPIRE) performs an HLA 2010 based simulation using the commercial simulators DIgSILENT PowerFactory and OPNET Modeler. An IEC 61850 based Object Model Template (OMT) is used to simulate object models, attributes, and interaction. Time synchronization is performed using a time-stepped synchronization which is based on the HLA time management services.

The remaining platforms have similar shortcomings - some are global event driven like GECO, some having fixed synchronization points like EPOCHS, while others have granular time steps executing every time unit instead of ones with events.

2.3 MOSAIK

The MOSAIK framework [35] targets CPS/smart grid research with a focus on large-scale system simulated system configurations, e.g., to study control strategies [36]. MOSAIK was developed capturing a concise set of functionalities aimed at CPS simulation studies [37]. The MOSAIK core is light-weight, translating to relatively small computational needs for the sake of the framework itself. Its core components are a simulator management module (*sim-manager*) and a *scheduler*, following the HLA standards. Accordingly, the simulator manager handles the federates (model creation, message passing, etc.) and the scheduler maintains the time synchronization of the simulation. The MOSAIK API handles the interfacing between the simulator manager and the component federates of the simulation.

2.3.1 Simulator Classification

HLA treats federates as either *time-stepped* or *discrete event simulators* [38]. Time-stepped simulators are not event-responsive and they update their variables based on the most recently received messages during synchronization. By contrast, a discrete event simulator is event-responsive and blocks during simulation awaiting messages intended for it. Furthermore, HLA allows the simulators to choose their simulator type at every time step. MOSAIK classifies simulators in a similar manner with minor differences in naming and operation, as described next.

Time-based Simulators

The simulators that have a notion of time are called *time-based* simulators. The states and events of such simulators can be mapped to certain points in time called *timestamps*, where time flows continuously. The execution of events always follow their chronological order of occurrence. Examples of such simulators are usually related to the physical world, e.g., the periodic generation of data from a simulated temperature sensor. In principle, such simulators can produce observational values for any arbitrary point in time.

Event-based Simulators

Event-based simulators "jump" from event to event in temporal order. Therefore, these simulators can be terminated due to lack of events. Events are related via timestamps with *specific* points in time. An example of such simulation is one simulating sending and receiving of messages in a communication simulator. Reception happens a fixed amount of time after the transmission event. Given the timestamp of the sending event, one can determine the timestamp of the corresponding receiving event.

Hybrid Simulators

Hybrid simulators are a combination of both the aforementioned simulators. They have a notion of time and can also jump from event to event. They can be used to represent any kind of combined system with both time-based and event-based components.

A co-simulation environment may contain one or more of different types of simulators. A simple example with three components would be a sensor-based voltage controller. The first component is the *sensor*, which simulates the voltage reading of a circuit periodically. The second component is a *communicator* that simulates message transfers from one place to another. The third component is the *controller*, which simulates the voltage regulation of the circuit based on the sensor reading received through the communicator. Each of the components can be classified based on dependency and paradigm as follows:

1. **Sensor:** The sensor is independent of any other component as it generates sensor readings regardless of what the other simulators are executing. As the readings need a timestamp indicating when the reading was taken, the sensor has a notion of time and can therefore be labeled as a time-based component.

2. **Communicator:** The communicator is dependent on the sensor for input and executes message transfer events *if* there is a message to deliver. Therefore, this component is an event-based component, where the events are triggered externally and periodically by the sensor.

3. **Controller:** The controller receives sensor data from the communicator (not directly from the sensor) and is therefore dependent on the communicator. This is also an event-based component.

Assume the termination criterion is to have the sensor stop sending simulated values at some point. If the sensor reaches a termination condition, i.e., a simulation end time, it will not generate further sensor data. The communicator being dependent on the sensor data, will not be triggered anymore. However, it will need to deliver the last sensor data, if the message is still being carried. Similarly, when the communicator delivers the last sensor data and no further events are scheduled, the communicator can terminate and let the controller know that it has terminated. The controller can then terminate after it has performed regulation based on the last sensor reading. This is one example termination condition that runs to exhaustion the complete processing of all that is dependent on sensor values. The last sensor value needs to be completely transferred, processed, and acted upon for the simulation to terminate.

Alternative designs also exist, e.g., if the controller performs periodic regulation, it becomes a time-based component as well. It is still dependent on the communicator as the sensor readings determine the nature of regulation. However, a different termination condition, e.g., a simulation end time of its own, may be necessary. Termination conditions can be described in arbitrary ways, and regardless of what they are, the co-simulator needs to gracefully handle it without impacting simulation correctness.

2.3.2 The MOSAIK API

The application programming interface (API) of MOSAIK is Python-based and handles the communication between MOSAIK and (1) Python-based simulators using the high-level API, (2) non-Python simulators using the low-level API. The primary difference between the API's is that the low-level API assumes the communication is done using JSON messages through a network socket and that the coupling simulator will follow a specific message format (refer to Figure 2.2). The high level API performs the same operation using a specified programming language, i.e., Python. MOSAIK treats each simulator

as a *SimPy* process and executes/blocks them as required to maintain synchronization. MOSAIK generates dependency graphs based on the user-supplied interconnections among the simulators and uses it to perform causally accurate co-simulation, lower bound computations (LBTS), etc.

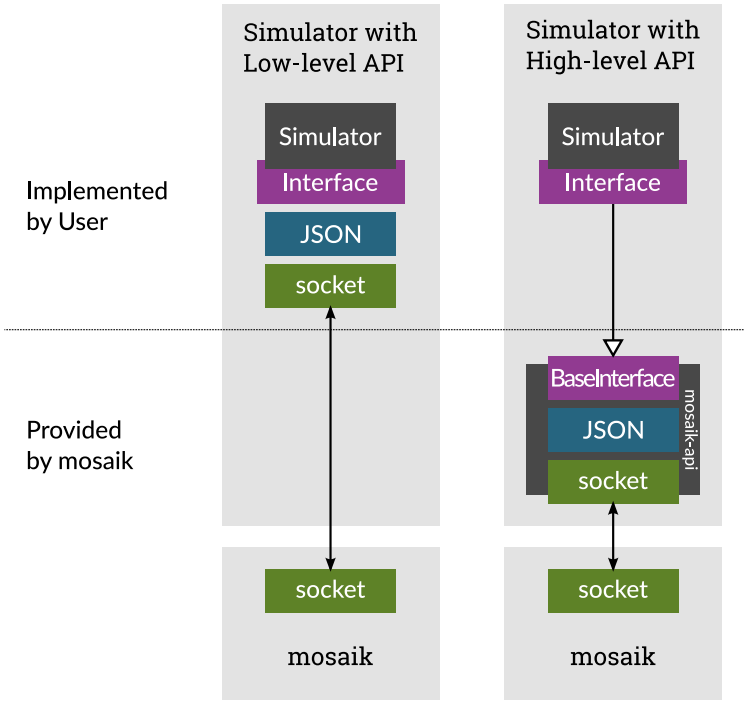

Figure 2.2: Low-level and high-level MOSAIK Application Programming Interface (API) variations as it appears in [39].

There are certain API functions specified by MOSAIK to allow management of the constituent simulators. The majority of these functions have their default implementation and need to be overloaded by the simulator for any further modification. Furthermore, MOSAIK allows addition of new functions as the simulator requires. There are two kinds of API functions - synchronous and asynchronous. Synchronous functions are used by MOSAIK to manage the simulators and they are called at specific stages of simulation. Asynchronous functions are used by the simulator to perform certain actions like add surplus data or change the data provided to MOSAIK before its scheduled expiry. However, they are not executed unless the user calls them during a synchronous function execution. The Table 2.2 lists the synchronous functions and Figure 2.3 shows the sequence of their calling during a co-simulation.

Function	Description	Output
`init`	initializes the simulator	metadata
`create`	creates simulator *models*	*model* dictionary
`setup_done`	called when simulation setup is done	no output
`step`	executes a simulator for a specified period	next event time
`get_data`	collect data for most recent step execution	simulator output
`configure`	allow simulator configuration with command line	no output
`finalize`	called after simulator terminates	no output

Table 2.2: Synchronous API functions of MOSAIK.

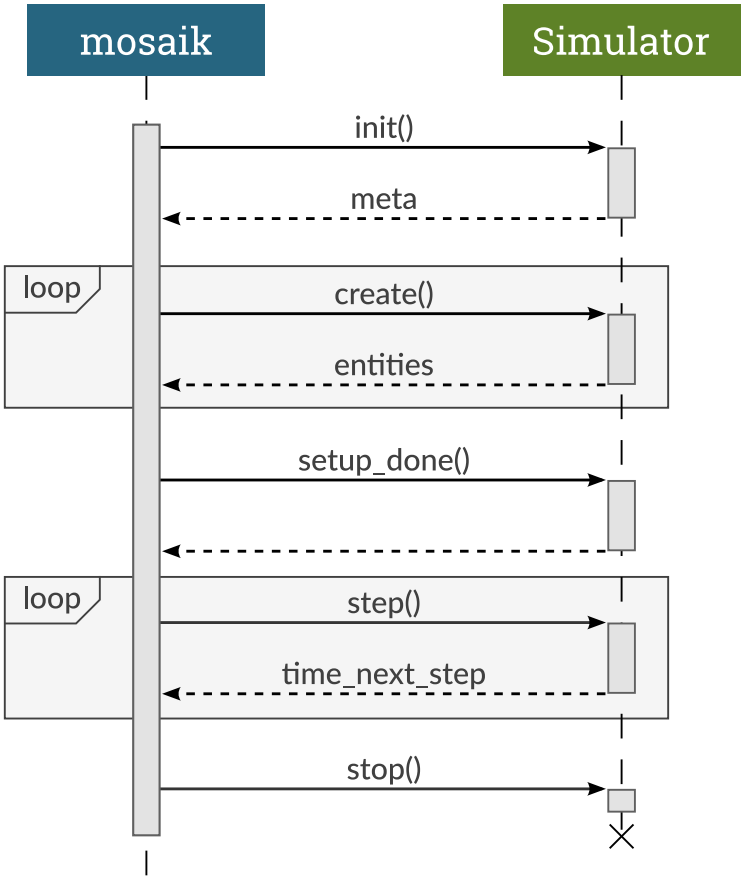

Figure 2.3: MOSAIK API synchronous functions call sequence (from [40]).

2.3.3 *Model* Instances

MOSAIK identifies each simulator with certain configurations - both mandatory and optional. Some of these configurations specify the type of simulator, their inputs and outputs, while others specify the connections and communication among them. The primary configurations for the latest version of MOSAIK (3.0.0 at the time of writing) are:

Simulator Configurations

A set of configurations need to be specified before initialization of the simulators. These include the name of the Python file and command for Python-based simulators. For other languages, the executable needs to be created and the command for invoking the executable has to be mapped to a user generated name for further reference. The command should also include the current location of the executable and other arguments if necessary. These are used to instantiate the simulators as *SimPy* processes along with some other MOSAIK-specific configurations like - the start and end time of simulation, the dependency graph activation flag, etc.

Metadata

The metadata of each simulator describes the type of simulator - time-based, event-based, or hybrid, the MOSAIK version, the names of simulator *models*, the number and names of attributes of those *models*, the type of attributes, etc. *Models* are components of the simulators which communicate with other *models*, instead of a direct simulator-simulator communication. These attributes are returned by the simulators after initialization and are used by MOSAIK for simulator specific handling. For instance, if the attributes are defined as *persistent*, MOSAIK assumes that the data provided by the simulator "persists" in time until the next set of data are provided. On the contrary, *non-persistent* data are not valid for the following time steps and need to be renewed if necessary. A *time step* is the unit of time by which MOSAIK advances each simulator. It can be 1 millisecond, or even several minutes, depending on the specified configurations and how each simulator perceives time. Similarly, an attribute defined as *trigger* is used to step a simulator when the listed

attributes are provided to the simulator as input (from other simulators). Attributes of event-based simulators are *trigger*'s by default as they are stepped using events. An event can also be the availability of input data from other simulators.

Connections

When a simulator can pass data to another simulator, or vice-versa, they are said to be connected to each other. As connections in the MOSAIK dependency graph are unidirectional, the direction of data needs to be specified when connections are formed. A simulator taking input from other simulators is called a *dependent* simulator (refer to Section 2.3.1). The nature of these inputs can be *triggering* or *non-triggering*, *persistent* in time or *non-persistent*. There can also be cyclic dependencies, however, one of the two connections need to be prioritized over the other. Consequently, if two simulators in a cyclic connection have a step *at the same point in time*, the simulator with prioritized *output* in the connection is executed first. If the connections are not specified, the simulators will not be able to communicate with each other. However, the co-simulation will still be valid and only warnings will be given about disconnected simulators.

Every simulator can have multiple instances of multiple *models* with various *attributes* and *parameters*. For example, a power grid simulator may have sensor *models*, actuator *models*, smart-meter *models*, etc. The *model* parameters are defined in the metadata when the simulator *models* are specified and they are initialized when the simulator instances are created. The attributes are used to pass data among *models* of different simulators. Although, MOSAIK can distinguish among multiple instances of the same *model*, the user cannot. Therefore, we introduce a unique instance identifier, or an *entity-id* (EID) to mark every *model* instance. This ensures that inputs received or outputs generated by an instance, are always tagged with, correspondingly, the generating instance EID and the receiving instance EID. This allows for easy multiplexing and de-multiplexing of messages passed from simulator instances to other simulator instances. The naming convention used for generating the unique IDs is discussed at length in chapter 4.

2.4 Simulator Selection

For the purposes of this work and because we simulate cyber-physical systems with the capability of sensing residential and grid power, estimating and regulating voltages, and experimenting with various communication infrastructures connecting the different devices in the circuit, we chose the following constituent simulators.

2.4.1 OpenDSS

We simulate the power elements of the system using the well-known electric power distribution system simulator (DSS) - OpenDSS [41]. OpenDSS is designed to support distributed energy resource (DER) grid integration and grid modernization. We use OpenDSS as it is a Python-based simulator and can therefore be integrated with MOSAIK easily. Furthermore, it is customizable, flexible, and easy to use even for large-scale electrical networks.

Background

The OpenDSS program initially began as "DSS" for Distributed System Simulator, after which it was acquired by EPRI Solutions [42]. Eventually, EPRI released the software under an open source license which allowed researchers to utilize and improve OpenDSS. Co-operating with other grid modernization efforts in the Smart Grid area, OpenDSS became able to express most electrical systems. Over the course of time, as object oriented programming became more popular, OpenDSS integrated the object oriented design into its core components. Most recent versions of OpenDSS allow programming in C++, MATLAB, Python, etc.

Motivation

OpenDSS has its roots in power system harmonics analysis and is therefore more powerful than a regular power flow program. However, the scripting concept of OpenDSS allows users to use specific modules and avoid the majority of the simulator tools that are not required. MOSAIK being a Python-based co-simulation platform, allows OpenDSS to be integrated with it and to

perform efficient power flow analysis. Furthermore, the interfacing requirements of MOSAIK can be facilitated by the scripting capabilities of OpenDSS. The static state system assumed by OpenDSS means the simulated circuit can be modified on-the-fly, instantly, and as needed, without any temporal constraints. Therefore, it can be used to handle the power flow analysis of large scale smart-grids at arbitrary time points and circuit configurations.

Configuration

Most of the sensors and electrical components of the circuit are modelled by OpenDSS Python scripts using the OpenDSS internal system as the circuit base. The scripts implement *models* which allow fetching of certain values and executing certain operations matching the device definition. For example, a smart-meter would be able to read the voltage and power of different phases at its assigned location. The circuit state variables are known using the OpenDSS system and then filtered using the scripted device *models*. These *models* are then used by the MOSAIK interface to generate simulations, change states according to input from other simulators, and provide output data when requested. Although OpenDSS does not have a notion of time, the interfacing generates timestamps and performs state changes, message passing, etc. at specified time steps.

2.4.2 Network Simulator 3 (ns-3)

Network Simulator 3 [43] or ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. NS-3 is free, open-source software, licensed under the GNU GPLv2 license, and maintained by a worldwide community.

Background and Motivation

The ns-3 project started in 2006 as an open-source project which develops and continues to improve on the ns-3 simulator. It provides models of data packet networks and a simulation engine for conducting simulation experiments. The models are designed based on real world network devices and protocols. and are developed using well-known standards and mathematical

models. While these simulations are not as accurate as real-world experiments, they can greatly reduce the costs of employing a physical network.

The object oriented design of ns-3 is well suited to the layered structure of the Internet protocols and for experimental tweaking of certain components of the layered devices. This reduces the overall complexity of simultaneous implementation of multiple protocols, frequent switching between protocols, expressing the addition/removal of infrastructure in large communication networks, etc. Furthermore, the customizable nature of the internal components allow user-developed modules required for smart-grid networks. Therefore, ns-3 is a good match for our co-simulation experiments.

Internal Components

The programming of ns-3 is done in two separate languages - C++ and Python, and the user can install and use either of the two. We use C++ as it is the language in which the simulation core and models were implemented initially, and is therefore allowing access to its internal logic. However, most of its API can be imported to Python programs as *modules*. The various components of the simulator can be divided into multiple tiers based on their dependencies as shown in Figure 2.4.

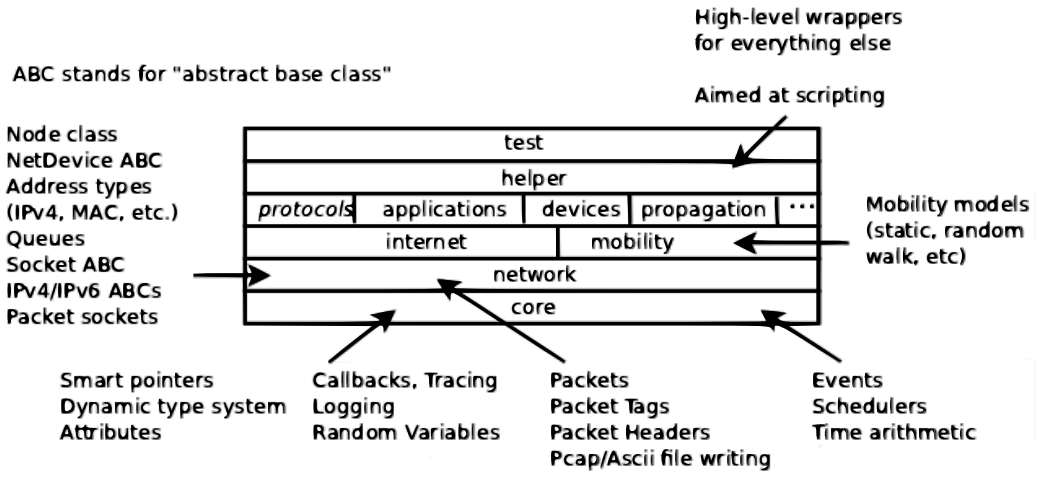

Figure 2.4: Internal software components of ns-3 (adapted from [44]).

1. The inner-most tier consists of the *core* which implements the time management i.e., the events, schedulers, lower bound calculations, etc.,

the memory management, random number generations, debugging and software specifics like callbacks, tracing, logging, etc. These modules are generally independent of one another and are used by all other components built on top of them.

2. The second tier consists of the *network* modules like devices, addresses, queues, sockets, etc. The different protocols which are required in any network are implemented in this tier. The implementations of this tier are all abstract base classes and are used by other models for specific implementation. These protocols are only dependent on the core components of ns-3.

3. The third tier contains the internet modules and the mobility modules. The internet modules implement the protocols required to connect and manage one or more networks. The mobility module handles the location information of the various devices during simulation like static location assignment, random movement, etc. This tier is dependent on the network tier which indicates that base models need to be created first before assigning them locations using the mobility modules or connecting them to other models using the internet modules.

4. The fourth tier consists of the specific implementations of protocols, devices, applications, propagation models, etc. The abstract classes of the *network* tier are inherited and specialized to generate specific usable model classes. The third tier models have implementations using the base classes and can therefore be fitted to the specific model implementations as well. To put in another way, the specific models use the internet and mobility models to define their attributes and characteristics like location data and communication link specifics.

5. The fifth tier houses the *helper* classes which are helpful wrappers for convenient and user-friendly implementation.

6. The sixth tier is the test tier which is basically test and example scripts using the helper classes.

2.4.3 Python-based Simulators

Besides OpenDSS and ns-3, other Python-based simulators are used in the current study to perform operations like controlling a certain simulator *model* or collecting simulation data for analysis.

Controller

The Controller simulator performs processing tasks like regulating the voltage of a certain portion of the circuit. This simulation is outside the domain of both OpenDSS and ns-3. The behavior associated with this simulator is in reference to particular locations in the circuit, and is assumed to receive data from several other devices. An example *model* is the `RangeControl` which provides tap control data based on the sensor device readings. Each Controller *model* may be associated with a certain operational function and may "control" one or more devices in the circuit. The Controller simulators are implemented as Python code obeying the MOSAIK interface and interaction requirements.

Estimator

The Estimator is similar to the Controller in that there is no underlying place to define it in the ns-3 or OpenDSS, and it is also implemented as separate Python code obeying the MOSAIK interface. An example *model* is the `DSESim`, which reads data from various devices in the circuit and estimate the current *state* of the power flow system. Any *state* of the system may include several system variables, the values of which may or may not be known. The known values are used by `DSESim` to generate the unknown values, determining the system state in the process. The values produced by the Estimator are then passed as input to other simulators. Note that in contrast to the Controller, the Estimator does not necessarily have a direct influence on the state of the power network. One can perform state estimation only for the sake of recording it.

Collector

The primary functionality of the Collector simulator, which is also writen in Python and obeying the MOSAIK interface, is to collect the data generated

by other simulators and store it in a data store. Currently, the only *model* of this simulator is `Monitor`, which is a single instance that connects to all *model* instances of the other simulators, and therefore “taps“ on all messages exchanged among simulators. The stored data is used for analysis after the simulation ends. However, in the future the Collector may be extended to allow interaction the user to interact with the simulators, i.e., by injecting messages that alter the behavior of the simulator instances.

Chapter 3

The Co-simulation Architecture

Our proposed co-simulation project allows the simulation of the networked monitoring and control of components in cyber-physical systems. The different simulators/federates are glued together using MOSAIK interfaces as shown in Figure 3.1. The power grid simulated using OpenDSS describes the placement of electrical components, circuit interconnections, and simulates any changes to the electrical system during simulation. The communication network simulated using ns-3 allows transfer of messages among the simulator *models*, as they would occur in a smart-grid. Additional, purpose-built, Python-based simulators simulate the control/regulation of certain electrical components, estimate the current system state of the smart-grid, and collect simulator statistics for further analysis. These five simulators, their interconnections, and the example applications they co-simulate, are detailed in the following sections.

3.1 Power Flow Simulator

A power grid is an amalgamation of multiple devices and electrical components. OpenDSS describes the most basic components of the circuit (e.g., capacitors, transformers, etc.), while other components (e.g., smart-meters, phasors, probers, etc.) are simulated outside of OpenDSS.

3.1.1 Circuit Structure

OpenDSS initializes the circuit and its parameters like - the name of the newly formed circuit, the default voltage and phases of the circuit, placement

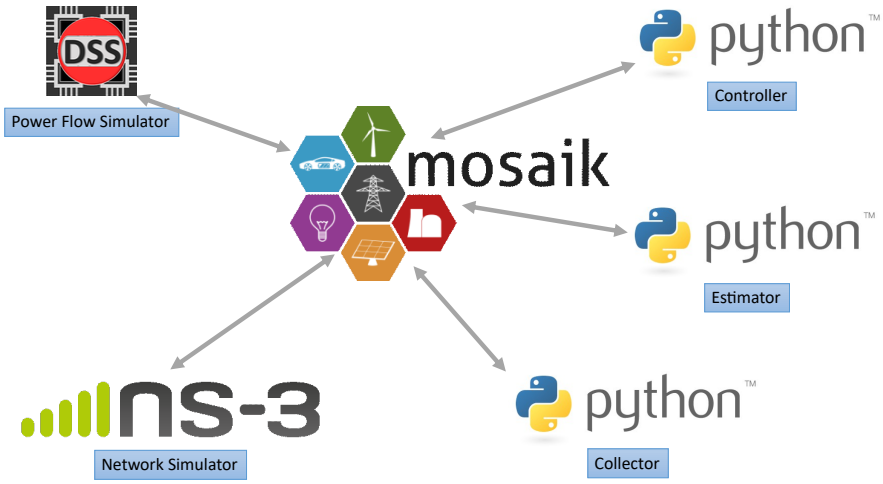

Figure 3.1: High-level view of a MOSAIK co-simulation instance.

of the voltage source, etc. Once the circuit is formed, basic electrical equipment are placed at the intended locations while specifying the parameters of the equipment. For example, placing a new transformer would require determining the buses where it would be attached, the number of phases it can handle, the number of windings it has, etc. If the transformer allows voltage regulation, further parameters like the number of taps or the max and min tap would be specified. These circuit specifics are fed to the simulator using certain configuration files, which will be discussed in Section 4.1.1. Although the loads of the equipment are initialized using these files, they are changed during the simulation using random load generation or by assigning set loads collected from empirical studies. The circuit may be divided into two parts based on the proximity of the circuit elements to the location of residential loads. The several parts of the network connected to the residential loads are called *residential* networks or *secondary* networks, while the central distribution grid connecting all the *residential* networks is called the *primary* network. The *primary* network carries high voltage electricity to minimizes loss of electrical power in the form of heat when when heavy current through electrical wires. Step-down transformers connect the *primary* network to the *secondary* networks and reduce the high-voltage to usable ranges.

3.1.2 Devices

Besides the basic components defined by OpenDSS, we simulate more complex electrical components which we will refer to as *devices*. These *devices* utilize the basic functionalities provided by OpenDSS to simulate certain actions like sensing, actuating, probing, etc. They are placed at various points in the circuit and can be of the following types:

`Sensor`

The `Sensor` is a generic *device* assumed to be able to generate readings of a certain circuit component at periodic intervals. It can be located at both the primary and/or secondary network. The simulation *model* for this *device* updates the readings periodically based of the current state of the power grid. `Sensor` readings need to be carried by the communication network to other locations. `Sensor` sub-classes provide specific forms of sensing, e.g., `Smartmeters` and `Phasors`. According to our *model* definitions, `Smartmeters` provide the voltage magnitudes and load readings of different phases and are located in the *secondary* network. One smart-meter typically reads data from 2-3 residences. The `Phasors` provide voltage and current readings (i.e., both magnitude and angle) and are situated in the *primary* network.

`Actuator`

The `Actuator` is a *device* that changes the state of the power network. As an example it can be used to change the voltage of the circuit on the fly based on received control data. Step-up/down actuators are located near variable transformers to allow remote voltage regulation. The voltage regulation is done in the form of tap ups or downs, which respectively increase or decrease the coil ratio in a transformer to change the voltage. The `Actuators` perform actuation based on external commands, and therefore the regulation decisions are taken by a separate but associated `RangeControl` instance.

`Probers`

The `Probers` provide any and all types of periodic sensor data collection. However, they do not need to transfer data through the communication network

and can be located at any point in the grid. They are used to log a certain *device* or circuit component for debugging or statistical purposes.

3.2 Network Simulator

The packet network simulated by ns-3 simulates transfer of messages from any required source (generation) location to the intended recipient location (destination). The topology of the generated communication network is a super-set of the electrical network. Hence the network simulator can simulate co-located as well as non co-located, to the grid, networking equipment. The network *model* associated with carrying messages i.e., the `Transporter` is instantiated for every source-destination pair. Although these *models* may have common internal architecture in ns-3, MOSAIK assumes each *model* instance to be different from the other. The communication network devices are divided into certain logical layers to allow division of functionality [45]. The protocols implemented in the different layers of the network devices and the procedure for sending and receiving data (sensory and control) are explained in the following sections.

3.2.1 The Underlying Network

The communication network at the physical level is an interconnected set of network devices with network layer routing capabilities. The devices with application layer capabilities form the data sources and sinks of the `Transporters`. They are simulated as being placed in a 3D Cartesian coordinate space using the ns-3 *mobility* module. Currently, we specify the coordinates as static i.e., we place the devices along the horizontal plane with very little variation along the vertical axis.

Implemented Protocols

The physical layer connections between the different network devices are determined using certain configuration files (refer to Section 4.1.3). The current architecture allows changing network-wide configuration settings during simulator initialization. For example, the user can specify the link transmission

delay, rate, and error, which will be applied to all existing links in the network. The network `mode` can be set to `P2P`, `P2Pv6`, `CSMA`, and `CSMAv6`. `P2P` and `P2Pv6` implement IPv4 and IPv6 over the network, where all device to device connection protocols are Point-to-Point. Similarly, the `CSMA` and `CSMAv6` implementations apply the corresponding Internet Protocols (IP) over connections with Carrier-Sense Multiple Access (CSMA). All connections in the `P2P` and `CSMA modes`, and the *primary* network connections in the `P2Pv6` and `CSMAv6 modes`, are wired and thus have very little delay and data loss during packet delivery, as set by the user. The `P2Pv6` and `CSMAv6 modes` apply IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) at the *secondary* networks. The data link layer of 6LoWPAN in ns-3 simulates the Low-Rate Wireless Personal Area Network (LR-WPAN) which has a completely different model for determining transmission loss and delays, and are therefore unaffected by the user-specified link parameters. The medium access sub-layer of LR-WPAN implements the collision avoidance version of CSMA (CSMA/CA). Future work may allow setting different protocols to every link in the network through the use of appropriate configuration files.

The Rationale for `6LoWPAN`

In practical scenarios, there is little change in the distribution topology of the *primary* network, and therefore the corresponding communication network consists of wired connections to allow loss-less long distance data delivery. On the contrary, *secondary* networks tend to have wireless technology to allow for flexible topology adjusting to the dynamics of the environment. To this end, the wireless network standard to prefer for simulations of wireless *secondary* networks was chosen to be `6LoWPAN`, as it is already included in the current ns-3 distribution.

3.2.2 Traffic Generation and Reception

When a sensor reading needs to be delivered to a controller or a control action needs to be conveyed to an actuator, the data is transferred through transport layer applications. The client-server applications of two well-known protocols - Transport Control Protocol (TCP) and User Datagram Protocol (UDP) have

been implemented for this purpose. For example - in a TCP scenario, when a sensor is placed in the network and the location of the receiver is known - a TCP client is installed at the sensor location and a TCP server is installed at the receiving sink. Following the establishment of source-destination connections, the routing tables of the network devices are populated using static routing. The routing algorithm used is Breath First Search (BFS) and the routing metric used is hop-count. The data generation and reception procedure is as follows:

1. The client generates data when the sensor does.
2. The data is simulated as being forwarded along the route determined through static routing.
3. When the data reaches the server, it is passed to the recipient *model* (of another simulator) through MOSAIK.

3.3 Controller

The Controller simulates control of the distribution grid. There are various controllers but we emphasize the example of controllers to regulate voltage. Other controllers deal with power storage, reactive power control, etc. We focus on on-load tap changers (OLTCs) with their appropriate control algorithms [46]. In our co-simulation evaluation, one set of examples demonstrate simulation of a grid with OLTC.

3.3.1 Constraints

Power systems with renewables in the distribution grids tend to have increased variability in the generated voltage. The resultant need to have frequent tap changes may accelerate wear-and-tear of OLTC devices. This is because, voltages need to be kept within the permitted range to avoid device damage and to ensure proper usage. Therefore, the OLTC devices need to maximize their lifetime by minimizing the total number of tap operations while ensuring a smooth voltage profile.

3.3.2 Basic Principles

The basic principle of OLTC voltage regulation is to compare the current voltage to a previously set voltage point V_{set}. If the comparison difference is within the specified bandwidth BW, no tap change is required. A tap change in a transformer increases/decreases the coil turn ratio, which in return changes the voltage level of the transformer output current. Instead of changing the coil turns manually every time a tap change is required, the number of coils remain the same, but the circuit taps in between coil contacts to effectively change the turn counts on one side of the transformer. This changes the turn ratio i.e., it changes the voltage and current in the resultant electricity flow. An intentional delay t_{delay}, is included in the algorithms to avoid frequent tap change operations. A tap change only occurs if the voltage level is outside the specified range for a duration of more than t_{delay}.

3.3.3 Control Algorithm

The control algorithm we simulate in our Controller uses the periodic sensor data readings to calculate the time delay and perform tap changes. The algorithm starts executing when a new sensor data is received as follows.

- Compare the received voltage value V_{meas} with the specified voltage set point V_{set} for this control *model*.
- If the difference $\Delta_v < \frac{BW}{2}$, do nothing and wait for the next sensor reading. Otherwise, proceed to the next step.
- If the current reading time is T_{meas}, compare it with the previously received reading. Let the difference be Δ_t. If $\Delta_t > t_{delay}$, it is assumed that the voltage reading is same for the last Δ_v time. Therefore, proceed to the next step. Otherwise, wait for the next sensor reading.
- If $\Delta_v < 0$, decrease tap i.e., increase the voltage as it is below the set point. If $\Delta_v > 0$, increase tap i.e., decrease the voltage as it is above the set point.

We name the *models* executing this control algorithm - `RangeControl`. Every `RangeControl` *model* instance has its own set voltage and delay values. However, all of them operate using the same control algorithm.

3.4 Estimator

The Estimator is a component that simulates distributed state estimation (DSE) using smart-meters and distribution level phasor measurement units (D-PMUs) [47]. D-PMUs are installed at a small number of buses in the primary network, and their measurements are available after a certain delay with added noise. The *model* of the Estimator which performs the estimation is called `DSESim`. In our co-simulation evaluation, one set of examples demonstrate simulation of `DSESim` on a power grid.

3.4.1 State Estimation Principles

State Estimation (SE) is the technique of estimating the system variables using redundant data from multiple sources. The state variables are usually the voltage phasors of the nodes, which includes the voltage and phase angles at the various nodes in the electrical system. The required data for the estimation are the distribution system network model, real-time measurements from the placed D-PMUs, and the load demands at the end-devices.

Network Model

The network model represents the network configuration and line parameters i.e., the impedance/admittance values. Although in practice, the entirety of the network model may not be known to the state estimator due to various reasons. In our experiments, we assume that the network model is known and available to the `DSESim`.

Real-time Measurements

Real-time voltage and phasor measurements can be obtained through devices located at various points in the circuit. In our simulations, the network of D-PMUs, each one of them being a sensor, provide synchronized voltage and current phasor readings of three phases.

Load Demands

Smart-meters are placed at end devices i.e., the households in the secondary network which provide sample data periodically. Due to the slow sampling rate of smart-meters, and thus the unavailability of online load data for state estimation, pseudo-measurements are used. These measurements are predicted load values based on historical data. `DSESim` instances may make use of such load value estimates.

3.4.2 State Estimation Methodology

The state estimation technique used in our simulations is the Weighted Least Squares (WLS) method [48]. The stages of state estimation performed at every step of the simulator can be described briefly as follows:

- Collect the required data: (a) nodal admittance matrix (b) D-PMU measurements, and (c) pseudo-measurements of (predicted) real and reactive power consumption. Here, the nodal admittance matrix is collected only once during the creation of the `DSESim` *model* instances. However, the D-PMU and smart-meter readings are collected at every step of the simulation.
- The error function is used to introduce measurement errors into the readings - measurements with less error are given higher weights. The WLS method is then used to reduce the difference between the collected measurements and the estimated state variables.

3.5 Collector

The Collector is a simulator containing a single *model* called `Monitor`. The `Monitor` is stepped every time any other simulator steps with outputs. The output data generated by the *models* of the respective simulators are taken as input by the `Monitor`, which locally stores the collected data. At the end of the simulation, the stored data is written to a Hierarchical Data Format (HDF) Store using the *pandas* module of Python. The stored data is then used by

a separate program to perform analysis and for graphical rendering of the results.

3.6 MOSAIK Interconnections

The simulator *models* communicate with each other through predefined MOSAIK connections. These connections are unidirectional and time-synchronized. The *sim-manager* of MOSAIK controls the passing of data through these connections and the duration of validity of the passed data. The *scheduler* advances the simulators based on their dependencies to maintain chronological order of messages and thus, the order of event execution. Cyclic dependencies are permitted only if one of the two formed connections is marked as *weak* or *time-shifted*.

3.6.1 Connection Parameters

The connection parameters define the sender and recipient of the data, the content of the passed message, and the type of connection in cyclic dependencies i.e., *weak* or *time-shifted*.

Time-shifted

The *time-shifted* parameter is used to resolve cyclic dependencies between time-based simulators. The dependent simulators of time-shifted connections are stepped first i.e., a forceful stepping priority is established to avoid a deadlock formed by the simulators associated in the cyclic dependency. For example, let us assume that simulator A and B are connected to each other with some parameter $x \in A \rightarrow B$ and some parameter $y \in B \rightarrow A$. Let us resolve the deadlock by marking the latter connection as *time-shifted*. Therefore, when both simulators have a step at the *same* timestamp, A is stepped first and x is generated. Consequently, B steps using the input from A and generates the output y. However, this output is only passed to A in its next time step. The question arises - what will be the input of A for its first time step? i.e., when B has not generated yet the first set of values for y. This is resolved by the user providing a separate connection parameter called the *initial value* which is used only for the first time step.

Weak

Event-based simulators use a different parameter to resolve cyclic deadlocks - namely, *weak*. Similar to the time-based counterpart, a non-weak connection is prioritized over a weak connection. However, simulators may be stepped at the same timestamp multiple times to establish *stability*. For better understanding - let us utilize the previous example, except the connection $y \in B \rightarrow A$ is now *weak*. When A and B both have a step at the same timestamp, A generates x first and then B steps and generates y. A is stepped again *at the same timestamp* to receive the updated y parameter from B. If A generates a novel x value for B, B is also stepped again. This carries on until both A and B have stepped using the latest input parameters, i.e., they have 'stabilized'. The loops formed in this manner are called *same-time (algebraic) loops* and are permitted in MOSAIK as long as there are finite number of such loops. The initial data i.e., the input data for the first step needs to specified by the user as done for time-shifted connections.

3.6.2 Inter-model Connections

We define the connections between the different simulator *models* using configuration files generated by our ontology generators (refer to chapter 4). The connections between the *models* (Figure 3.2) are determined based on the type of the simulators to which the *models* belong, and the priority of the message being carried by a connection. We define the Power Flow simulator, Controller, and Estimator, as *hybrid*, and the remaining simulators as *event-based*. The cyclic dependencies are therefore resolved using *weak* connections.

1. The `Sensors`, `Smartmeters`, and `Phasors`, of the Power Flow simulator are connected to the `Transporter` *model* of the Network Simulator. The data generated by these devices are carried to their intended destinations by the `Transporter` *model*. Once a sensor reading is generated, it becomes invalid for the next time steps, so as not to trigger duplicate input data for `Transporters`. Accordingly, the simulator is defined as *hybrid* so that the outputs of the `Sensor` *models* can be marked as *non-persistent* (refer to Section 3.7.2).

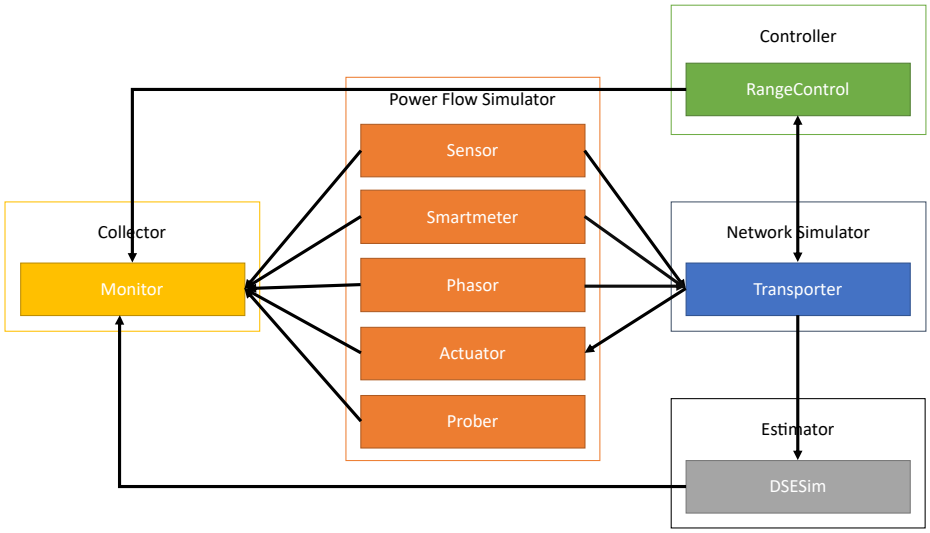

Figure 3.2: MOSAIK inter-connections between the different simulator *models*

2. The `Transporters` are connected to the respective `RangeControl` *model* instances of the sensor devices. When the sensor readings reach their destination through the `Transporters`, ns-3 passes the forwarded data to MOSAIK, which passes it on to the `RangeControls` for processing. For state estimation examples, the `Transporters` are connected to a single `DSESim` *model*. The data carry loop ends here for state estimation applications unless the estimation result is used for control operations.

3. The `RangeControl` *model* instances are again connected to their dedicated `Transporters` which carry the control data to their place of execution. For example, a tap control application will have `RangeControls` sending control data to `Actuators` for voltage regulation. This connection is *weak* as the `RangeControls` need sensor data before taking any control decision.

4. The `Transporters` are connected to the Power Flow `Actuators` as they carry the data to the `Actuator` locations and forward the control data to them through MOSAIK. This completes the control loop for certain applications like the tap control application which have a sense-process-act cycle. The connection is specified as *weak* as the Power Flow simulator needs to run in order to generate (new) sensor data before actuation is

possible.

5. Finally, all *models* are directly connected to the `Monitor` *model* of the Collector. Every set of data generated by any *model* is passed to the `Monitor` which writes the data to a common HDF data store.

3.7 Time Synchronization and Stepping

The MOSAIK *scheduler* maintains the chronological order of simulator events given that the simulators -

- maintain the order of their internal events,
- generate messages only for future timestamps,
- step when requested and up to the required time,
- provide a time when the next step might be performed and,
- avoid executing beyond the provided *max advance* time.

The max advance time is the maximum any simulator can execute without disrupting the chronological order of co-simulation events and is analogous to the *LookAhead* of HLA. The performance of any co-simulation depends greatly on the refinement of the time-steps and fine-tuning the generation of the *max advance* time. We provide several scenarios in the following sections which represent our various stages of performance improvements while providing a gradual understanding of the different time synchronization techniques.

3.7.1 Stage 1: Granular Time Steps in MOSAIK 2

The previous version of MOSAIK (MOSAIK 2) did not have the ability to differentiate between time-based and event-based simulators. The constituent simulators were similar in all aspects and were unable to access the *LookAhead* for future events. The initial approach, for the sake of simplicity, was to perform co-simulation with granular time steps. The characteristics of the simulators were as follows:

- All simulators start from time 0 and advance in time steps of variable units.
- The simulators *must* return their next time step, which is used to calculate the *LookAhead* internally. The simulators were not aware of the next step time except their own. Therefore, we specify the next time step as 1 unit.
- The time unit of all simulators are uniform and can be represented as any real-time unit. We assume each time unit to be 1 millisecond.
- As the simulators are stepped at every time unit, any data generated by a simulator is forwarded to MOSAIK immediately i.e., at the current time step.
- All simulators keep advancing their time with the permission of MOSAIK until the designated simulation *END TIME* set by MOSAIK.

The passing of data immediately after generation helps avoid triggering events that might break causality. However, the granularity of the time steps *requires all simulators to be stepped at every time step*. Consequently, the co-simulation is inefficient as a significant portion of the time steps are uneventful and the overall execution time is greatly increased.

3.7.2 Stage 2: MOSAIK 3 with max advance

MOSAIK versions 3.0.0 and later have classified the simulators into time-based, event-based, and hybrid, with the simulators having access to *max advance* time during every time step. The availability of this look ahead allows simulators to process internal events without concern for break of causality. In order to calculate the *max advance* time, MOSAIK needs to know the next time step of all the simulators participating in co-simulation. The simulator management changes can be summarized as follows:

- The output of time-based and hybrid simulator *models* are *persistent* by default. The input of event-based simulators are *trigger*'s by default. Hybrid simulators can have non-persistent outputs and triggering inputs as well.

- Time-based simulators *must* return their next step time at every step. Event-based and hybrid simulators may or may not provide their next time step as they can be *triggered* by external events.
- Time-based simulators and hybrid simulators without triggering inputs have their *max advance* time set to the simulation *END TIME*. This is because they can only be stepped by themselves and are therefore unaffected by external events.

Power Flow Simulator

We define the Power Flow simulator as a hybrid simulator. The sensor devices generate readings periodically and the `Actuator` operates based on external inputs. The simulator steps every time a sensor data is generated and when there is an `Actuator` input data to be processed. Following practical scenarios, the rate of actuation is typically less than the rate of sensor data generation. The data generated is *non-persistent* in time i.e., the data is only valid for the timestamp of the step. This is because if the same data is persistent for more than one timestamp, ns-3 will generate duplicate sensor data for as long as the data is *persistent*. Any time a sensor data is generated, the next generation timestamp is inserted into a priority queue and all events with earlier timestamps are removed from the queue at every time step. The next step which is the lower bound of all generated events, is calculated as the event with minimum timestamp in the queue.

Network Simulator

The network simulator is an event-based simulator as the timestamps of events in a packet network is irrelevant except when calculating delays i.e., difference between two timestamps is more important than the timestamp of a specific event. However, ns-3 has its own time management mechanism which inserts events into its internal queue and processes them as time progresses. For the second stage of refinement, we calculate the next step of ns-3 by checking the next earliest event in its own internal event queue. As we do not have a way of understanding which events are of interest to MOSAIK, ns-3 usually has the most number of steps in any co-simulation test run.

Controller

The Controller *models* may have their own operation interval, given that the interval is greater than the data generation interval of their sensor counterparts. However, they may be stepped when sensor readings are available, only to process the input without generating output data. If control data is generated as soon as they are processed, the provided lower bound computation may be inaccurate. The Controller simulator is therefore classified as a hybrid simulator, where the simulator steps based on the control operation rates of the `RangeControl` *models* and the data delivered by ns-3. The lower bound computation of its internal events is done similar to the Power Flow simulator, where the next control step of a *model* is inserted into a priority queue. The next time step is the control event with the minimum timestamp in the queue.

Estimator

The Estimator *models* collect data from all sensors to generate an estimated system state. As there are no current recipients of the estimated data, except the `Monitor`, the state estimation simulator is defined as a hybrid simulator as it generates estimation data periodically for recording purposes, even if adequate event-driven data is not received. Besides stepping periodically, it is also stepped whenever there is sensor data. A lower bound computation is not necessary as its only recipient does not generate external events for other simulators.

Collector

The Collector simulator is an event-based simulator and is always stepped when any other simulator *model* generates data. The input parameters of the simulator *model* - `Monitor`, are not explicitly defined so that it can accept any and all types of data.

3.7.3 Stage 3: Relevance Filtering in ns-3

Most of the internal events generated by ns-3, such as those describing the arrival and departures of packets as they are routed from one endpoint to an-

other, are irrelevant to the rest of the simulators, The internal ns-3 events that are relevant to other simulators are those describing when specific data are received by specific nodes in the network. Hence, the lower bound computation of ns-3 is very inefficient as it often reflects the next upcoming, but very likely irrelevant event. Efficient co-simulation greatly depends on the accurate calculation of lower bounds of the internal events in the constituent simulators. However, access to the internal event queue of ns-3 is restricted and the relevance of an event cannot usually be determined until it has already been executed. In order to ascertain the relevance of an internal event, let us study the process of extracting relevant event data.

Extraction of Relevant Data

The generation and reception of co-simulation data that require passing through the communication network, are done in the form of client-server application traffic. Clients are installed at data generation locations, which schedule data packets when there is new data. The communication network passes the data through the simulated route, which generates several irrelevant events. Once the data reaches the intended destination, the server receives the data and calls a callback function to process the received data from the application side. We modify the server to call a custom callback function that stores the received data in a queue to be forwarded to MOSAIK when required. Although, this allows extraction of the received data at the exact timestamp, the irrelevant events generated throughout the process demands frequent stepping of ns-3.

Filtering of Relevant Data

The availability of *max advance* through the new version of MOSAIK provides a safe look ahead for processing the internal events of ns-3. Utilizing this look ahead, every time there is a step for ns-3, instead of only processing the events up to the designated step time, we advance ns-3 until there is a relevant event or until the *max advance* time is reached. A relevant event in this case would be the reception of data by a destination node in the network. If a relevant event is processed, the received data is inserted into the queue,

and the timestamp of this event is returned to MOSAIK as the next time step. On the contrary, if the *max advance* time is reached, no data is returned to MOSAIK and the procedure repeats itself. As ns-3 cannot roll back time to process past events, if any external events are generated with a timestamp earlier than the processed relevant event - the simulation would become invalid. However, if the *max advance* time is accurate, no such external events will be generated and thus ns-3 can jump to either the relevant event or the *max advance* time without stopping to process every irrelevant event.

Chapter 4

Simulation Configuration

Generation of simulator *models* and formation of connections among the generated *models* become tedious for large-scale simulations. Moreover, the unique EID generation of the *models* require prior knowledge of all generated *models*. We automate this process using an ontology definition over all the simulator *models*, connections, and their properties.

4.1 Configuration Requirements

There are several configuration data required by MOSAIK and the constituent simulators to perform organized co-simulation. Some determine the connections between the different nodes in the communication network while others specify the placement of devices in the electrical network. Each configuration file has its own format which allows systematic parsing and editing by users or the separate simulator programs.

4.1.1 OpenDSS Circuit Generator

The electrical system part of the smart-grid co-simulation is generated by OpenDSS using certain configuration files. The configuration files describe the formation of the circuit and the placement of devices in the circuit are text files with “.dss" extensions. Multiple such files might be used depending on the system scale. The contents of the circuit generator file are the location and properties of electrical lines, loads, transformers, etc. Part of an example OpenDSS configuration file is presented in Figure 4.1, showing the configuration of a new circuit and of the transformers used in a tap control simulation.

```
...
!-- Generated Circuit/generator
new circuit.IEEE13Nodeckt
  basekv=115 pu=1.0001 phases=3 bus1=SourceBus
  Angle=30
  MVAsc3=20000 MVAsc1=21000
!-- Generated Transformer
New Transformer.Sub Phases=3 Windings=2 XHL=(8 1000 )
  wdg=1 bus=SourceBus conn=Delta kv=115 kva=5000 %r=(.5 1000 )
  wdg=2 bus=650 conn=Wye kv=4.16 kva=5000 %r=(.5 1000 )
New Transformer.XFM1 Phases=3 Windings=2 XHL=2
  wdg=1 bus=633 conn=Wye kv=4.16 kva=500 %r=0.55 XHT=1
  wdg=2 bus=634 conn=Wye kv=0.480 kva=500 %r=0.55 XLT=1
...
```

Figure 4.1: Example snippet from an OpenDSS configuration file.

4.1.2 Device List

The device list is used to place our devices of interest like sensors, smart-meters, phasors, and actuators, which are listed in a separate configuration file in a comma-separated values (CSV) format. This is the configuration for the remaining constituent simulators outside of OpenDSS and ns-3. The most significant table headers of the device list are:

1. **type** - This is the type of device - sensors and actuator types.

2. **src** - This is the location of the device, i.e., the source location from where the device data will generate for sensors, or from where the control data will generate for actuators.

3. **dst** - This variable stores the intended destination for the device data. The destination is usually controllers or estimators for sensor devices, and actuators for actuator devices.

4. **cidx** - The devices are usually part of a control loop, the ID of which is specified by the control index or *cidx* parameter.

5. **nidx** - The *nidx* is used to separate between two or more devices with the same source, destination, and control loop ID.

6. **period** - The period specifies the step size or rate of data generation of the device.

7. **error** - The error rate of the generated data.

8. **cktElement** - The circuit element to which the device is mapped. This is used to refer to the OpenDSS Circuit Generator when required.

9. **cktTerminal** - The location of attachment of the device, i.e., the bus or line in the electrical circuit to which the device is connected.

The concatenated **type**, **src**, **dst**, **cidx**, and **nidx** values are used as the EIDs of the generated *model* instances of the various simulators. Part of an example of the Device List configuration file as given in Figure 4.2. The EID of the *devices* in the given example would be `Sensor_611-632.0.0`, `Prober_611.2.0`, and `Prober_650.3.2`, respectively.

```
type, src, dst, cidx, nidx, period, error, cktElement, ...
Sensor, 611, 632, 0, 0, 100, 0.0001667, Line.684611, ...
...
...
Prober, 611, None, 2, 0, 100, 0, Load.611.3, ...
Prober, 650, None, 3, 2, 100, 0, 650, ...
```

Figure 4.2: Sample snippet from a Device List CSV configuration file.

4.1.3 Network Topology

As OpenDSS requires a configuration file to generate the circuit of the electrical system, so does ns-3 to generate the network of communication devices. One of the most well-known representations of connectivity in a communication network is the *adjacency matrix* of the generated graph. Although adjacency matrices are easy to process, they consume more memory than required to store connectivity information. The joint configuration file developed following the defined ontology is formatted in JSON. The JSON file is passed to ns-3 during initialization, which is then used to generate the necessary configuration data for ns-3. Specifically, the list of node names to create the network nodes and map their indices to their appropriate names, the location

data used to place the nodes in the 3D space using ns-3 *mobility* modules, and the *adjacency matrix* used to form network devices and interfaces, and connect the nodes through these interfaces. The connections in the JSON file, however, are maintained in the form of *adjacency lists* to reduce the necessary file size. The generation process can handle any topology as long as it is reflected accurately through the JSON file. Although, the current implementation uses ns-3 protocol helpers to install similar protocols throughout the network, future developments may include custom protocol assignment to specific connections based on the modified JSON file. A sample snippet from a generated JSON file is presented in Figure 4.3. The snippet lists the properties of the the *nodes* `650`, `RG60`, and `611`. The properties include their names, co-ordinates, and a list of their connections to other *nodes*.

4.1.4 Load Data

In order to configure realistic electric load scenarios for the power flow simulator we also produce load configurations. The loads of the various households in the secondary network, or the overall load of the primary nodes, may be generated in two ways - (a) random load generated using standard distributions and realistic load ranges (b) real-time data collected from households in an electrical distribution system. Our applications have access to both types of loads, the latter being stored as load data. These load values are stored as time series data in MAT files, which are processed by the Power Flow simulator to get non-randomized load data.

4.1.5 Nodal Admittance

The nodal admittance matrix is required for the distributed state estimation component and specifically used in our examples by the `DSESim` *model*. The matrix is stored as a *NumPy* binary file and is passed directly to the `DSESim` *model* instances during *model* creation.

```
{
  "nodes": {
    "650": {
      "x": 200,
      "y": 350,
      "connections": [
         "SourceBus",
         "RG60"
      ]
    },
    "RG60": {
      "x": 200,
      "y": 300,
      "connections": [
         "650",
         "632"
      ]
    },
    "611": {
      "x": 0,
      "y": 100,
      "connections": [
         "684"
      ]
    },
    ...
  ...
  }
}
```

Figure 4.3: Generated JSON snippet for network topology configuration.

4.2 Ontology

The Ontology definition establishes the objects of the ontology with their appropriate properties, which are then used to form connections between the different objects. There are four core classes that define the different objects of the ontology, and then there are several properties associated with each object which also contains the connection attributes.

4.2.1 Equipment Class

The entities which have a physical presence in the smart-grid world are defined as the *Equipment* class. There are two different kinds of equipment depending on whether they are in the communication network, or in the electrical system.

Electrical Equipment

The *Electrical Equipment* class has various sub-classes like *Bus*, *Capacitor*, *Transformer*, etc., relevant to the power grid network and each with their own set of constraints to allow proper and realistic functioning of the simulator. For example, *Generator* classes and *Load* classes can only *attach* themselves to one *Bus* object, while the remaining classes like *Line*, *Switch*, *Capacitor*, etc., can attach themselves to two *Bus* objects.

Network Devices

The *Network Devices* class has two sub-classes expressing data communication and processing, respectively. The first sub-class is called *Communication Node*, while the latter is called *Processor*. *Processor* nodes may represent actuators, controllers, estimators, or sensors, in the communication network.

4.2.2 Characteristics Class

The *Characteristics* class defines the various properties of the *Equipment* class objects.

Circuit

The properties of the primary circuit used in the power grid power flow simulator is specified in this sub-class. There is usually only one circuit active at a time as OpenDSS cannot simulate more than one active circuits simultaneously.

LineCode

Objects of *Line*, which is a sub-class of *Equipment*, have their own set of properties. These properties specify the impedance values that are used to generate the nodal admittance matrix for state estimation. These properties are known as *LineCodes*.

Transporter

The *Transporter* sub-class defines the specifications of the communication technologies associated with *Equipment* objects of the *Network Devices* class.

4.2.3 Location and Measurable Class

The *Location* class contains the geographical location of the object it is *attached* to. The location description can be co-ordinates, or coarser location identifiers, like cities, neighborhoods, etc. The *Measurable* class allows measurement of the attached object and specifies the properties of measurement, like the phenomenon being measured, the units of measurement, the equipment doing the measuring, etc.

4.2.4 Object Properties

Besides specifying the attributes of objects, the *Object Properties* also determine the relation between classes. Table 4.1 summarizes the different object properties, their domains, and ranges. The domains and range of the object properties are also objects. For example, the objects of the class *Sensor* may have the property *measures* to calculate or take the measurements of a *Measurable* object. Whether the measurement is of a voltage level of the power grid or the power consumed by a certain household, is determined by the user-defined object of the *Measurable* class. Certain properties may have

an inverse property e.g., the *isMeasuredBy* is the inverse of the *measures* property and can be used to get the sensor which is doing the measuring, through the *Measurable* object. Similarly, the *Sensor* object can get its sensor readings using its *measures* property to fetch the *Measurable* object.

Object Property	Inverse	Domain	Range
measures	isMeasuredBy	Sensor	Measurable
isMeasuredBy	measures	Measurable	Sensor
monitor	isMonitorBy	Sensor	Equipment
isMonitorBy	monitor	Equipment	Sensor
locatedAt	isLocationOf	Equipment	Location
isLocationOf	hasLocation	Location	Equipment
controls	isControlledBy	Actuator	Equipment
isControlledBy	controls	Equipment	Actuator
feeds	isFedBy	Equipment	Equipment
isFedBy	feeds	Equipment	Equipment
connectsTo	connectsTo	Network/Control Equipment	Network/Control Equipment
attachesTo		Electrical Equipment	Bus
primaryAttachesTo		Electrical Equipment/Circuit	Bus

Table 4.1: Object properties.

4.2.5 Example Scenario

Figure 4.4 shows the ontology of the smart-grid tap control application. The dashed lines indicate inheritance and the filled lines indicate object relations through their properties. The Generator, Load, Bus, etc. objects and the *Network Devices* sub-class are inherited from the *Equipment* class. Therefore, dashed lines connect them to the *Equipment* class. Similarly, most electrical equipment like the *Generator*, *RegControl*, *Load*, *Line*, etc. are connected to the Bus equipment. The *Bus* class forms the connection between the other electrical equipment classes and the *Circuit* class. The filled line connecting the classes like *RegControl* and *Line* to the Bus is the property *(primary)attachesTo*, which indicates that there is both an *attachesTo* property

and a *primaryAttachesTo* property associated with it. The *locatedAt* property connects the *Equipment* class objects to the *Location* class objects, while the *feeds* property connects to itself, as *Equipment* class objects feed their data to the other objects of the same class. The *connectsTo* property is more direct as it does not require any medium object in between the connected objects, compared to the *feeds* property which usually transfers the data through the communication network. The *Sensor* objects monitor other *Equipment* objects and *measures* the *Measurable* object related to the monitored object. The *Controller* objects feed the *Actuator* objects, and the *Actuator* objects *control* other *Equipment* objects, e.g., a *RegControl* object attached to a *Transformer*.

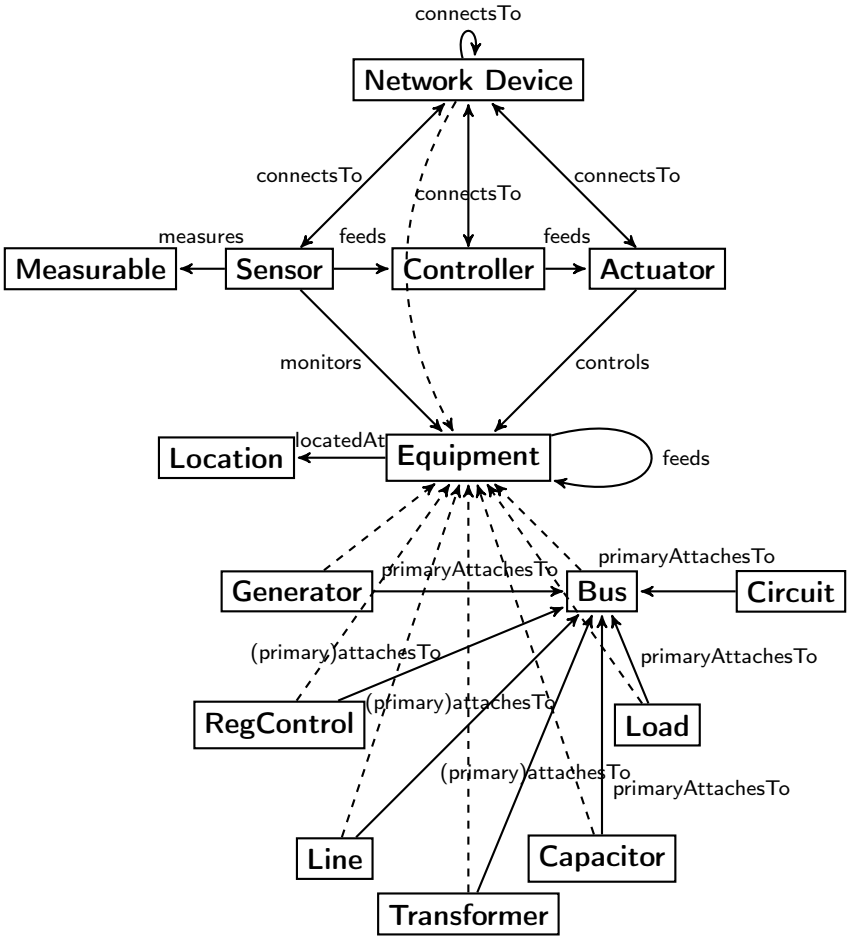

Figure 4.4: Class relations in the Smart grid ontology.

4.3 Co-simulation Execution

The co-simulation is started using the Linux shell script `smartgrid_run.sh`, which is responsible for the set up of the environment for execution, importing required Python libraries and executing a master script and an analysis script. As the application domain is smart grid simulation, the OpenDSS libraries are typically imported. The simulator calling configurations are initialized in the variable `SIM_CONFIG` in the MOSAIK master script (as presented in Figure 4.5). The simulator classes, such as, `PFlowSim`, `PktNetSim`, `ControlSim`, etc., are defined in their respective interface codes. We use a convention of appending to the interface the version number of MOSAIK used. For example, the MOSAIK 2 interfacing codes for the Power Flow simulator are in file `simulator_pflow_2.py`, while for MOSAIK 3, interfacing is done in `simulator_pflow_3.py`. For non-Python simulators like ns-3, execution files are referenced with appropriate libraries instead of the Python interface scripts. The execution file is generated by compiling multiple C++ class files before executing the master MOSAIK script.

4.3.1 Simulator Initialization

MOSAIK initializes the simulators after configuration, using the `init` API function. The simulators implement their version of the interface functions which are called from the master script for initialization. The Controller and Estimator simulators take the name of the simulators, *model* prefixes, and the verbose level of information printed during execution, as input during initialization (Figures 4.6 and 4.7). The Collector simulator has added parameters for storing the collected data into a data store (Python *HDF5 dataset*). The Power Flow simulator takes the topology file (Section 4.1.1), which specifies the different electrical equipment in the circuit, the initial load data of the nodes (Section 4.1.4), the rate of load data generation throughout the simulation, and the verbose level of printing information. Notice that the MOSAIK 2 version also takes the step size as input as it was stepped periodically only. The Network Simulator takes two configuration files as input - the JSON file containing the number, location, connections, properties, etc., of nodes in the

```
SIM_CONFIG = {
    'Collector': {
        'python': 'simulator_collector_3:Collector',
    },
    'ControlSim': {
        'python': 'simulator_controltap_3:ControlSim',
    },
    'PFlowSim': {
        'python': 'simulator_pflow_3:PFlowSim',
    },
    'PktNetSim': {
        'cmd': NS3_EXE_PATH + '/NS3MosaikSim %(addr)s',
        'cwd': Path( os.path.abspath(os.path.dirname(NS3_EXE_PATH))),
        'env': {
                'LD_LIBRARY_PATH': NS3_LIB_PATH,
                'NS_LOG': "SmartgridNs3Main=all",
        }
    },
    'Estimator': {
        'python': 'simulator_dse:Estimator',
    },
}
```

Figure 4.5: Example simulator configuration in the MOSAIK master script.

network (Figure 4.3), and the devices file containing the location and properties of *devices* (Section 4.1.2). The JSON file is parsed within ns-3 to generate the node adjacency matrix and the node co-ordinates (Section 4.1.3). The notable of the remaining parameters (as exhibited in Figurea 4.6 and 4.7) specify the link parameters, simulation start and end times, transport control protocol used, network and data link layer protocol specifications, and the user specified name of the circuit to be used for simulation. Other parameters are the verbose level which works similar to other simulator parameters, and the random seed value used to generate random values within ns-3. The parameters passed through the MOSAIK API functions are received by the simulator-specific implementations of these functions. We detail a Python-based implementation and a non-Python based implementation of the `init` function in the following sections.

Python-based: Power Flow

The Power Flow simulator interface contain the implementation of the API functions required by MOSAIK to control the simulator. The mandatory functions to be implemented by a simulator are the `step`, and `get_data`. There are five more API functions which have default implementations and are not required to be implemented. Furthermore, extra API functions can be added with prior notification to MOSAIK. Therefore, the `init` function needs to be overwritten if there are user-defined simulation parameters to be passed to the simulator during initialization as shown in Figure 4.8. The default parameters are the simulation identification or `sid`, and the time resolution for specifying the relation between MOSAIK time and simulator-specific time (only present in MOSAIK 3). Besides these two, the remaining simulator parameters require an overwritten implementation of the `init` function.

The `init` function implementation stores the input parameters and initializes other modules, except the simulator *models*, which are generated by the MOSAIK API function `create`. However, the `init` function needs to notify MOSAIK of the simulator *models* and extra API functions beforehand in the form of a metadata returned to MOSAIK as in the example of Figure 4.9. The metadata contains the version of MOSAIK in use, the type of simulator, the

```
if Mosaik_2:
    pflowsim    = world.start('PFlowSim',
                          topofile = DSS_EXE_PATH + TOPO_RPATH_FILE,
                          nwlfile  = DSS_EXE_PATH + NWL_RPATH_FILE,
                          ilpqfile = DSS_EXE_PATH + ILPQ_RPATH_FILE,
                          step_size = 1,
                          loadgen_interval = 80,
                          verbose = 0)
else:
    pflowsim    = world.start('PFlowSim',
                          topofile = DSS_EXE_PATH + TOPO_RPATH_FILE,
                          nwlfile  = DSS_EXE_PATH + NWL_RPATH_FILE,
                          ilpqfile = DSS_EXE_PATH + ILPQ_RPATH_FILE,
                          loadgen_interval = 80, # IEEE13
                        #   loadgen_interval = 1000, # IEEE33
                          verbose = 0)

if Scenario == 1:
    controlsim  = world.start('ControlSim', verbose = 0)

    pktnetsim = world.start( 'PktNetSim',
        model_name      = 'TransporterModel',
        json_file       = JSON_RPATH_FILE,
        devs_file       = DEVS_RPATH_FILE,
        linkRate        = "512Kbps",
        linkDelay       = "15ms",
        linkErrorRate   = "0.0001",
        start_time      = 0,
        stop_time       = END_TIME,
        random_seed     = args.random_seed,
        verbose         = 0,
        tcpOrUdp        = "tcp",
        network         = "P2P",
        topology        = "IEEE13"  # For now only IEEE13 and IEEE33
    )
else:
    estimator  = world.start('Estimator',
            eid_prefix = 'DSESim_',
            verbose = 0)
```

Figure 4.6: Simulator Initialization: Part 1

```
else:
    estimator  = world.start('Estimator',
            eid_prefix = 'DSESim_',
            verbose = 0)

    pktnetsim = world.start( 'PktNetSim',
        model_name      = 'TransporterModel',
        json_file       = JSON_RPATH_FILE,
        devs_file       = DEVS_RPATH_FILE,
        linkRate        = "1024Kbps",
        linkDelay       = "1ms",
        linkErrorRate   = "0.0001",
        start_time      = 0,
        stop_time       = END_TIME,
        random_seed     = args.random_seed,
        verbose         = 0,
        tcpOrUdp        = "tcp",
        network         = "P2Pv6",
        topology        = "IEEE33"
    )

collector   = world.start('Collector',
                eid_prefix = 'Collector_',
                verbose = 0,
                out_list = False,
                h5_save = True,
                h5_panelname = 'Collector',
                h5_storename ='CollectorStore.hd5')
```

Figure 4.7: Simulator Initialization: Part 2

```
def init(self, sid, time_resolution, topofile, nwlfile,\
        loadgen_interval, ilpqfile="", verbose=0):
    self.sid = sid
    self.verbose = verbose
    self.loadgen_interval = loadgen_interval

    self.swpos = 0
    self.swcycle = 35
    self.total_exec_time = 0.0
    self.step_count = 0

    if (self.verbose > 0): print('simulator_pflow::init', self.sid)
    if (self.verbose > 1): print('simulator_pflow::init', topofile,\
        nwlfile, ilpqfile, verbose)

    #--- start opendss
    self.dssObj = SimDSS(topofile, nwlfile, ilpqfile)
    if (self.verbose > 2):
        self.dssObj.showLoads()
        self.dssObj.showVNodes()
        self.dssObj.showIinout()
        self.dssObj.showVMagAnglePu()
        dss.run_command("Show Buses")
        dss.run_command("Show Voltages LN nodes")
        dss.run_command("Show Taps")

    #--- create instance of LoadGenerator
    #--- IEEE13
    self.objLoadGen = LoadGenerator(nwlfile,
                                    PFLimInf   =  0.95,
                                    PFLimSup   =  0.99,
                                    LoadLimInf = -1.65,
                                    LoadLimSup =  0.70,
                                    AmpGain    =  0.30,
                                    # Freq       =  1./8640,
                                    Freq       =  1./100,
                                    PhaseShift = math.pi)

    sys.stdout.flush()
    return self.meta
```

Figure 4.8: Power Flow implementation of MOSAIK API function `init`

model names, parameters, attributes and attribute properties, and the name of the extra API functions, formatted as a Python *dictionary*. This *dictionary* is returned to the caller after execution of the function. The metadata of the Power Flow simulator currently lists five simulator *models* with similar parameters and attributes. The parameters include the unique ID of the *model* instances, i.e., the Entity IDentification or `eid`, the periodic interval of performing sensing or actuating or `step_size`, the rate of error when performing said action, the verbose level of information printing, and the circuit properties required to map them to the electrical circuit generated by OpenDSS. The mapping is done during *model* instance creation through the `create` API function, using the EIDs generated from the device list configuration files as explained in section 4.1.2. These *models* and their parameters are set to be publicly accessible by MOSAIK to allow access of their `eid`'s during formation of interconnections between the different *model* instances. The `init` function initializes the encapsulating interface to the Python OpenDSS libraries using the module `SimDSS`, and the load generator module `LoadGenerator` which generates new load data when called upon in the `step` function.

Non-Python: Packet Network

The network simulator simulates the packet network while communicating with MOSAIK through a TCP socket. We use two classes - `MosaikSim` and `NS3Netsim` to simulate the interface for MOSAIK and the packet network, respectively. Another class `ns3-helper` is used to aid in performing various tasks by the other two classes. The `MosaikSim` class takes the address for the TCP socket and an object of the `NS3Netsim` class as input. There is a separate driver C++ file containing the `main` function, which calls the constructor of the `MosaikSim` class (Figure 4.10). These four files are compiled as a group to generate the execution file used by MOSAIK to simulate the packet network. The constructor opens the TCP socket using the passed address, maps the MOSAIK API function names to functions of `MosaikSim`, initializes the data structure to store input parameters of the `init` function, and starts the *Main Loop* of the interface. The *Main Loop*, as shown in Figure 4.11, executes throughout the duration of the simulation to receive JSON messages from MOSAIK con-

```
META = {
    'api-version': '3.0',
    'type': 'hybrid',
    'models': {
        'Sensor': {
            'public': True,
            'params': ['eid', 'cktTerminal', 'cktPhase', 'cktProperty',\
                'step_size', 'cktElement', 'error', 'verbose'],
            'attrs': ['v', 't'],
            'non-persistent': ['v', 't'],
        },
        'Actuator': {
            'public': True,
            'params': ['eid', 'cktTerminal', 'cktPhase', 'cktProperty',\
                'step_size', 'cktElement', 'error', 'verbose'],
            'attrs': ['v', 't'],
            'trigger': ['v', 't'],
            'non-persistent': ['v', 't'],
        },
        'Prober': {
            'public': True,
            'params': ['eid', 'cktTerminal', 'cktPhase', 'cktProperty',\
                'step_size', 'cktElement', 'error', 'verbose'],
            'attrs': ['v', 't'],
            'non-persistent': ['v', 't'],
        },
        'Phasor': {
            'public': True,
            'params': ['eid', 'cktTerminal', 'cktPhase', 'cktProperty',\
                'step_size', 'cktElement', 'error', 'verbose'],
            'attrs': ['v', 't'],
            'non-persistent': ['v', 't'],
        },
        'Smartmeter': {
            'public': True,
            'params': ['eid', 'cktTerminal', 'cktPhase', 'cktProperty',\
                'step_size', 'cktElement', 'error', 'verbose'],
            'attrs': ['v', 't'],
            'non-persistent': ['v', 't'],
        },
    },
    'extra_methods': [
        'set_next'
    ],
}
```

Figure 4.9: Power Flow simulator metadata returned at the end of `init`.

taining the API function calls. The messages are parsed and the API functions are executed using the mapping generated beforehand. The results generated by the functions are formatted as JSON messages and sent back through the socket. Consequently, the Python-based API is imitated using back-and-forth JSON messages and appropriate API function executions.

```
MosaikSim::MosaikSim(std::string varargin, NS3Netsim *obj)
{
  std::cout << "Starting MosaikSim class with varargin: ";
  std::cout << varargin << std::endl;

  //--- Gets server from mosaik and verify if it has two parts
  assert(!varargin.empty() and varargin.find(':'));

  //--- get NS3 object
  objNetsim = obj;

  //--- initialize Mosaik commands map
  initMosaikCommands();

  //--- initialize NS-3 Properties map
  initNetsimProps();

  //--- split host and port
  AddrPort srvAP = parseAddress(varargin);
  host = srvAP.host;
  port = srvAP.port;

  //--- Initial verbose setting (0 = no message)
  verbose = 0;

  //--- Initialize step counter
  step_count = 0;

  //--- create socket
  openSocket();

  //--- start the mainLoop
  stopServer = false;
  startMainLoop();
}
```

Figure 4.10: Constructor of `MosaikSim` class.

The `init` function stores the input arguments received from MOSAIK,

```
void MosaikSim::mainLoop(void)
{
  if (verbose > 1)
    std::cout << "MosaikSim::mainLoop" << std::endl;

  std::string result;
  int currentMsgId;
  std::string messages;
  Json::Value jsonMessage;
  total_exec_time = 0.0;

  while (!stopServer)
  {
    try
    {
      mosaikLastMsgOp = SUCCESS;
      messages = readSocket();
      auto start = std::chrono::system_clock::now();
      jsonMessage = deserialize(messages, currentMsgId);
      result = simSocketReceivedRequest(jsonMessage);
      result = serialize(result, mosaikLastMsgOp, currentMsgId);
      send(sock, result.c_str(), result.size(), 0);

      if (verbose > 1)
        std::cout << "MosaikSim::mainLoop ***** MSG SENT !! *****"
            << std::endl;

      if (jsonMessage[0].asString() == "step" ||
        jsonMessage[0].asString() == "get_data")
      {
        auto end = std::chrono::system_clock::now();
        std::chrono::duration<double> diff = end - start;
        total_exec_time += diff.count();
      }
    }
    catch (std::exception &e)
    {
      std::cout << e.what() << std::endl;
    }
  }
}
```

Figure 4.11: Main Loop function of the `MosaikSim` class

```
std::string
MosaikSim::init(Json::Value args, Json::Value kwargs)
{

  std::string result;
  std::string param;
  std::string value;

  mosaikSid = args[0].asString();

  //--- process each parameter
  for (Json::Value::const_iterator item = kwargs.begin();
    item != kwargs.end(); item++)
  {
    param = (item.key()).asString();
    value = (*item).asString();

    if (netsimProp.count(param))  netsimProp[param] = value;
    else
    {
      std::cout << "Unknown init parameter"
        << item.key() << std::endl;
      mosaikLastMsgOp = FAILURE;
    }
  }

  vecNetSimConn = readDevicesFile(netsimProp["devs_file"]);
  verbose = stoi(netsimProp["verbose"]);

  //--- Initialize NS3 class
  objNetsim->init(netsimProp["json_file"],
                  netsimProp["devs_file"],
                  netsimProp["linkRate"],
                  netsimProp["linkDelay"],
                  netsimProp["linkErrorRate"],
                  netsimProp["start_time"],
                  netsimProp["stop_time"],
                  netsimProp["verbose"],
                  netsimProp["tcpOrUdp"],
                  netsimProp["network"],
                  netsimProp["topology"]);
  ...
}
```

Figure 4.12: Partial snippet of `init` function of the `MosaikSim` class

calls the `init` function of `NS3Netsim` to initialize the ns-3 network configurations, and returns the metadata similar to the Power Flow simulator. The `ns3-helper` class function `ReadAdjListJson` is used to generate the required node and link properties by parsing the JSON file (Figure 4.13), which are then used to create network devices or *nodes*, place the *nodes* at the specified co-ordinates, install protocol stacks and attach the required network device interfaces at the *nodes*, and connect the interfaces according to the parsed adjacency data. *Nodes* located at the gateways connecting multiple networks may have multiple, possibly different, types of network devices. When `Transporter` *models* are created by MOSAIK through the API function `create`, ns-3 places source and sink applications on required *nodes*. The route from the source to the sink is then determined through static routing (Figure 4.14), where the routing metric is hop count and the routing algorithm used is Breadth First Search (BFS). Note that the static routes are generated *only* for a source-destination pair “connected" through a `Transporter`.

4.3.2 Simulator Stepping and Message Passing

Following the initialization of the simulators, their timestamped execution is managed by MOSAIK using the API function `step`. While the `step` function only returns the next time step if any, the results of the execution are fetched using a different API function `get_data`. Therefore, multiple fetch commands may be executed for a single step, depending on the input requirements of the participating simulators. The *system state* (refer to Section 2.1.1) representation of all simulators except ns-3, is static which allows stepping and storing data to be returned without temporal and causal reordering. For example, the simulation of the Power Flow is performed as one-shot executions of OpenDSS executables. Any change done to the *system state* of the Power Flow requires re-executing OpenDSS and therefore there is no saved step retained over steps. The required system variables are stored and re-used until the system needs to be changed. Consequently, the Power Flow simulator generates load for the appropriate time stamp and updates the *state* of the circuit i.e., perform actuation and get new sensor data generated from the applied load and actuation, during the said step. The *state* remains constant

```
vector<vector<bool»
ReadNodeAdjListJson(string jsonFileName)
{
  NS_LOG_INFO("ReadNodeAdjListJson");
  memset(parent, -1, sizeof(parent));
  ifstream adjListFile;
  adjListFile.open(jsonFileName.c_str(), std::ios::in);
  if (adjListFile.fail())
    NS_FATAL_ERROR("File " « jsonFileName.c_str()
      « " not found");
  json config;
  adjListFile » config;
  unsigned int num_nodes = config["nodes"].size();
  vector<vector<bool» array;
  unordered_map<string, unsigned int> node_to_idx;
  array.resize(num_nodes, vector<bool>(num_nodes));
// This will map the node to a numerical index
  unsigned idx = 0;
  for (auto &node :  config["nodes"].items())
  {
    node_to_idx.emplace(node.key(), idx);
    ++idx;
  }
// This will create the adjacency matrix
  for (auto &node :  config["nodes"].items())
  {
    unsigned int curr_idx = node_to_idx[node.key()];
    for (auto &neighbour :  node.value()["connections"])
      array[curr_idx][node_to_idx[neighbour]] = true;
  }
  adjListFile.close();
  return array;
}
```

Figure 4.13: `ns3-helper` class function `ReadAdjListJson`, which parses JSON files to generate the adjacency matrix.

```
...
while(nextHop != srv)
{
  nextHop = FindNextHop(clt, srv, nodeAdjMatrix);
  Ptr<Node> nextHopNode = Names::Find<Node>(nextHop);
  Ptr<Node> cltNode = Names::Find<Node>(clt);
  Ptr<Ipv4> nextHopIpv4 = nextHopNode->GetObject<Ipv4> ();
  Ptr<Ipv4> cltIpv4 = cltNode->GetObject<Ipv4> ();
  uint32_t hostIfIndex, hopIfIndex;
  // The interfaces are in reverse order
  if(DeviceMap.find(make_pair(clt, nextHop)) == DeviceMap.end())
  {
    NetDeviceContainer link_dev =
      DeviceMap[make_pair(nextHop, clt)];
    hostIfIndex = link_dev.Get(1)->GetIfIndex() + 1;
    hopIfIndex = link_dev.Get(0)->GetIfIndex() + 1;
  }
  else // The interfaces are in correct order
  {
    NetDeviceContainer link_dev =
      DeviceMap[make_pair(clt, nextHop)];
    hostIfIndex = link_dev.Get(0)->GetIfIndex() + 1;
    hopIfIndex = lin_dev.Get(1)->GetIfIndex() + 1;
  }
  Ipv4Address nextHopAddress =
    nextHopIpv4->GetAddress(hopIfIndex, 0).GetLocal();
  staticRouting = ipv4StaticRouter.GetStaticRouting (cltIpv4);
  staticRouting->AddHostRouteTo(
    destAddress, nextHopAddress, hostIfIndex);
  Ipv4Address cltAddress =
    cltIpv4->GetAddress(hostIfIndex, 0).GetLocal();
  staticRouting =
    ipv4StaticRouter.GetStaticRouting (nextHopIpv4);
  staticRouting->AddHostRouteTo(
    srcAddress, cltAddress, hopIfIndex);
  clt = nextHop;
}
...
```

Figure 4.14: Code snippet of class `NS3Netsim` that performs static routing in IPv4 networks to form a connection between the specified source-destination pair. The best path is calculated by the function `FindNextHop` which uses BFS as the search algorithm and hop count as the metric.

regardless of further steps if the applied load is unchanged and no actuation is performed.

Network simulators typically have several events in the system queue at a time, some of which could be unrelated to the other simulators e.g., simulated extra traffic to create artificially high loads, while others require data to be returned to MOSAIK, such the messages that arrived at end points that will operate actuators. Therefore, when the `step` function of ns-3 is called, we perform the following actions while maintaining temporal event order,

1. Parse the JSON message from MOSAIK and map it to the appropriate function of `MosaikSim` class. In this case, the function is the implementation of the API function `step`.

2. Parse the input arguments from the JSON message to generate the data collected from participating simulators.

3. If there is data generated by a sensor or controller *model*, schedule an event for generating traffic from the appropriate application source. Note that the event to be scheduled **must** have a timestamp that is greater than the current time in ns-3 (refer to Section 3.7).

4. Once all events are scheduled, execute the events in the queue until the required time provided by MOSAIK. To optimize this process, we may execute future events while maintaining causal and temporal consistency as explained in Section 3.7.3.

5. Collect the data received by any application sink during the executed time step and store it for further processing (when `get_data` is called).

The events in the ns-3 system queue are executed in multiple ways using the function `runUntil` to compare between the three stages of simulation refinement (refer to Section 3.7) as shown in Figures 4.15 and 4.16. The relevance filtering flag determines whether all events will be treated as relevant, or only the events which cause application sinks to receive data through the packet network. Furthermore, the *max advance* time is only available when MOSAIK 3 is being used.

```
std::string
NS3Netsim::runUntil(uint64_t time, string nextStop)
{
  if (verbose > 1)
  {
    std::cout << "NS3Netsim::runUntil(time=" << time
        << " + 1)" << std::endl;
    #if PERFORMANCE_TEST < 2
      std::cout << "NS3Netsim::Max Advance time = "
        << nextStop << std::endl;
    #endif
  }
  sim = DynamicCast<SmartgridDefaultSimulatorImpl>(
    Simulator::GetImplementation());
  #if PERFORMANCE_TEST < 2
      uint64_t max_advance = stoul(nextStop);
  #endif
  currentTime = time;
  uint64_t next_step;
  bool relevance = false;
  uint64_t runUntil_time = time;
  while(!relevance)
  {
    #ifdef PERFORMANCE_TEST
      if (PERFORMANCE_TEST == 1 || PERFORMANCE_TEST == 3)
        relevance = true;
    #endif
    if (runUntil_time < (uint64_t)stopTime-1)
      sim->RunUntil(MilliSeconds(runUntil_time + 1));
    else
      sim->RunUntil(MilliSeconds((uint64_t)stopTime-1));

    if (verbose > 3)
    {
      DataXCHG dataSnt;
      for (auto it = 0; it != dataXchgOutput.size(); ++it)
      {
        dataSnt = dataXchgOutput.front();
        cout << "NS3Netsim::runUntil NS3 OUTPUT Buffer Src: "
            << dataSnt.src
            << " Dst: " << dataSnt.dst
            << " Val: " << dataSnt.val
            << " Time: " << dataSnt.time
            << endl;
        dataXchgOutput.pop();
```

Figure 4.15: `runUntil` function of the `NS3Netsim` class: Part 1

```
        dataXchgOutput.pop();
        dataXchgOutput.push(dataSnt);
      }
    }

    //--- Get the next new event
    next_step = (uint64_t)sim->Next().GetMilliSeconds();
    #if PERFORMANCE_TEST < 2
      //--- If next step exceeds max advance time
      if (next_step > max_advance || next_step >= (stopTime-1))
        relevance = true;
      //--- OR If there is a message received by a server,
      //--- a relevant event has been processed
      else if (!dataXchgOutput.empty())
        relevance = true;
    #endif
    runUntil_time = next_step;

    if (verbose > 1)
    {
      std::cout << "NS3Netsim::runUntil After_run NS3 time: "
        << Simulator::Now().GetMilliSeconds() << std::endl;
      std::cout << "NS3Netsim::runUntil next event: "
        << next_step << std::endl;
    }
  }
  //--- Return a step time so that Mosaik has to give "stop" command
  if (next_step == (uint64_t)stopTime-1)
    return std::to_string(next_step+1);
  else if (next_step > (uint64_t)stopTime-1)
  {
    if (time == (uint64_t)stopTime-1)
      return std::to_string((uint64_t)stopTime);
    else
      return std::to_string((uint64_t)stopTime-1);
  }
  //--- If there is data in buffer, a relevent event has been processed
  //--- return the current time of NS3 as the next event time for Mosaik
  if (!dataXchgOutput.empty() && time < Simulator::Now().GetMilliSeconds())
    return std::to_string(Simulator::Now().GetMilliSeconds());
  return std::to_string(next_step);
}
```

Figure 4.16: `runUntil` function of the `NS3Netsim` class: Part 2

```
void sendMessageToUpperLayer(string message, Ptr<Node> sourceNode,
    Ptr<Node> destinationNode)
{
  std::size_t current;
  //--- get val and val_time
  current = message.find("&");
  string id = message.substr(0, current);
  message = message.substr(current+1);
  current = message.find("&");
  string val = message.substr(0, current);
  string val_time = message.substr(current+1);
  //--- insert data on dataXchgOutput / give to upper layer
  DataXCHG dataRcv = {id,
                      Names::FindName(sourceNode),
                      Names::FindName(destinationNode),
                      val,
                      stoll(val_time)};
  dataXchgOutput.push(dataRcv);
  dataXchgTime.push((uint64_t)Simulator::Now().GetMilliSeconds());
}

void ExtractInformationFromPacketAndSendToUpperLayer(Ptr<Socket> socket)
{
  Address from;
  Ptr<Packet> packet = socket->RecvFrom(from);
  uint32_t srcNodeId;
  if (v4)
  {
    Ipv4Address srcIpv4Address =
    InetSocketAddress::ConvertFrom(from).GetIpv4();
    srcNodeId = mapIpv4NodeId[srcIpv4Address];
  }
  else
  {
    Ipv6Address srcIpv6Address =
    Inet6SocketAddress::ConvertFrom(from).GetIpv6();
    srcNodeId = mapIpv6NodeId[srcIpv6Address];
  }
  Ptr<Node> srcNode = NodeList::GetNode(srcNodeId);

  packet->RemoveAllPacketTags();
```

Figure 4.17: Code snippet of the relevant data reception and processing functions of the `NS3Netsim` class: Part 1

```
packet->RemoveAllPacketTags();
packet->RemoveAllByteTags();

uint32_t packetSize = packet->GetSize();
uint8_t *buffer = new uint8_t[packetSize];
packet->CopyData(buffer, packetSize);
string recMessage = string((char *)buffer);
recMessage = recMessage.substr(0, packetSize);

PacketMetadata::ItemIterator i = packet->BeginItem();
//A packet can contain fragments, complete payloads or
//a combination of both.
while (i.HasNext())
{
  PacketMetadata::Item item = i.Next();
  if (item.isFragment)
  {
    if (item.type == PacketMetadata::Item::PAYLOAD)
    {
      //We check if the sender node has an entry in the fragment
      //buffers hash table
      if (fragmentBuffers.find(srcNodeId) == fragmentBuffers.end())
      {
        //If there is no entry, insert an entry with an empty string
        fragmentBuffers[srcNodeId] = "";
      }
      //This packet is a fragment in its entirety, it can
      //correspond to one of the middle or end fragments of
      //a fragmented package
      if (item.currentSize == packetSize)
      {
        fragmentBuffers[srcNodeId] =
        fragmentBuffers[srcNodeId] + recMessage;
      }
      else if (item.currentSize < packetSize)
      {
        if (item.currentTrimedFromStart == 0)
        {
          string fragmentMessage =
          recMessage.substr(recMessage.size() -
          item.currentSize, item.currentSize);
          fragmentBuffers[srcNodeId] =
          fragmentBuffers[srcNodeId] + fragmentMessage;
```

Figure 4.18: Code snippet of the relevant data reception and processing functions of the `NS3Netsim` class: Part 2

```
            fragmentBuffers[srcNodeId] =
            fragmentBuffers[srcNodeId] + fragmentMessage;
            //Remove the fragment from the received message string
            //so that we can process that fragment
            recMessage = recMessage.substr(0, recMessage.size()
                - item.currentSize);
          }
          else if (item.currentTrimedFromStart > 0)
          {
            string fragmentMessage = recMessage.substr(0, item.currentSize);
            fragmentBuffers[srcNodeId] =
                fragmentBuffers[srcNodeId] + fragmentMessage;
            //Remove the fragment from the received message string
            recMessage = recMessage.substr(item.currentSize);
          }
        }
      }
    }
    else
    {
      unordered_map<uint32_t, std::string>::iterator
        fragmentBufferIterator = fragmentBuffers.find(srcNodeId);
      if (fragmentBufferIterator != fragmentBuffers.end())
      {
        string assembledFragments =
        fragmentBufferIterator->second;
        sendMessageToUpperLayer(
            assembledFragments, srcNode, socket->GetNode());
        fragmentBuffers.erase(fragmentBufferIterator);
      }

      //send the message received in the unfragmented payload
      string partMessage = recMessage.substr(0, item.currentSize);
      sendMessageToUpperLayer(partMessage, srcNode, socket->GetNode());
      recMessage = recMessage.substr(item.currentSize);
    }
  }
}
```

Figure 4.19: Code snippet of the relevant data reception and processing functions of the `NS3Netsim` class: Part 3

The application layer of ns-3 performs its own processing of the data received at any *node*. This is done in the form a *callback* function that is called when a new packet is received an application sink. We replace the default application layer of ns-3 with our custom implementation of the TCP and UDP source and sink. The custom sink calls a different function to extract the required information from the received data (Figures 4.17, 4.18, and 4.19). The extraction may require parsing the message or forming a complete message from packets fragmented by the sender. Once the extraction is complete, the collected message is reformatted and inserted into a local queue for fetching by the `get_data` function as shown in Figure 4.17. The `get_data` function returns the contents of the queue to MOSAIK upon request, and removes them from the queue to avoid duplication.

Chapter 5

Experimental Results

In this chapter, we generate two scenarios for co-simulation, perform accurate simulation of the scenarios with changing parameters, and evaluate the performance of our platform in terms of efficiency and scalability.

5.1 Simulation Examples

For our experiments, we develop two example simulation scenarios which demonstrate the application of our simulation platform in the CPS domain. The first scenario simulates a tap control application in a small-scale smart-grid. The second scenario simulates distributed state estimation in a large-scale smart-grid environment with residential loads attached to *secondary* networks.

5.1.1 Tap Control

The tap control application environment is the well-known IEEE 13 node test feeder circuit [49]. This topology has a generator attached to the end of the circuit, a single transformer, and a voltage regulator to allow implementation of tap control algorithms (Figure 5.1). The black dots represent the places where electrical equipment may be attached, and we refer to them as *nodes*. For example, the *node* to which the generator is attached, is called the *SRCBUS* (source bus).

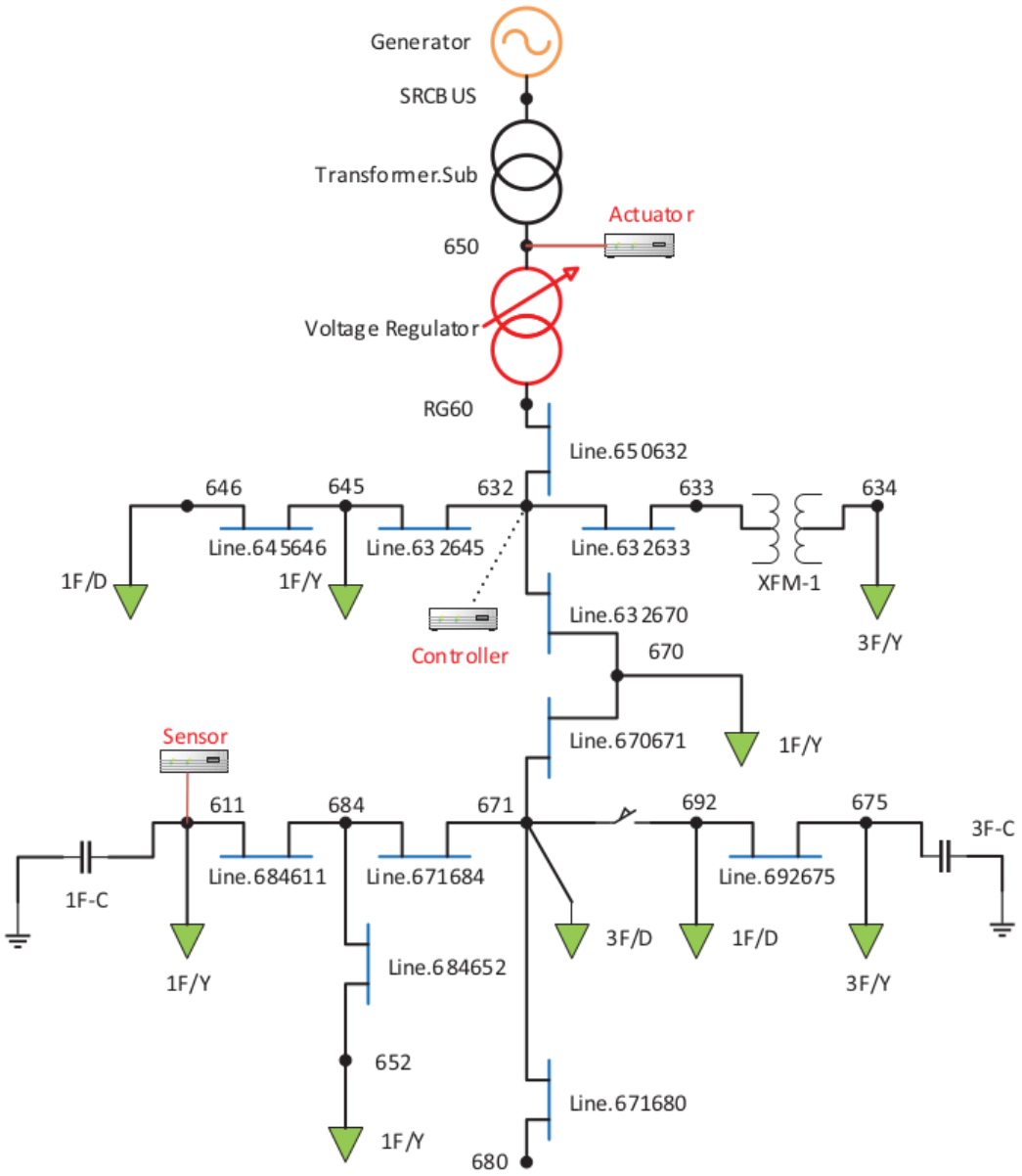

Figure 5.1: IEEE 13 node test feeder [50].

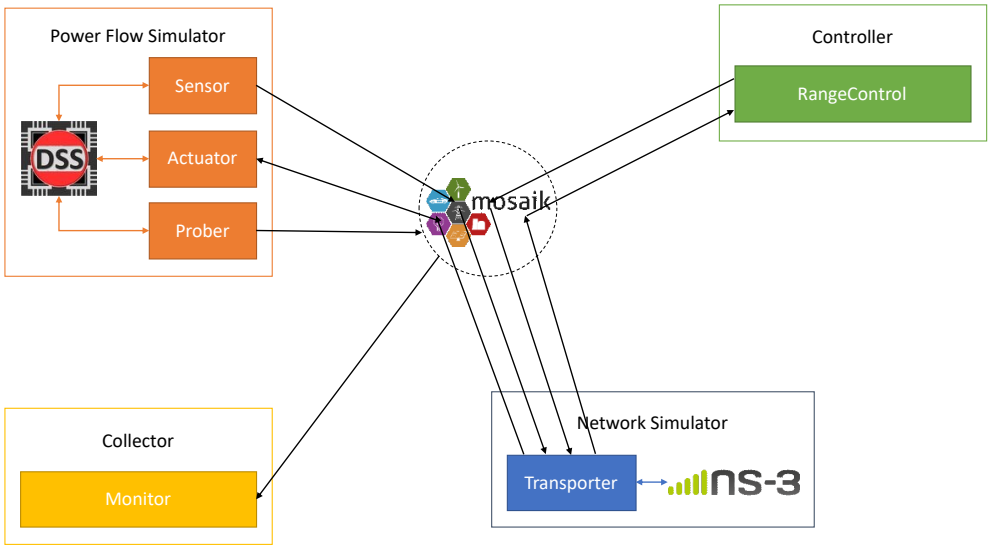

Figure 5.2: Tap Control data flow among simulation *models*.

Power Flow

For our simulation, we place a `Sensor` at *node 611* and an `Actuator` at *node 650*. The data generation period of the `Sensor` is 100 ms and the rate of actuation depends on the generation of control data by the Controller simulator. Generated sensory data contains the voltage value at the placed location and the timestamp of the generated data. The intended recipient of all sensory data is the Controller instance located at *node 632*. The placement of the `Sensor` is done at an edge of the circuit to allow for maximum data travel, while the location of the `Actuator` requires a nearby variable transformer on which the actuation is executed. The loads of the nodes are generated with uniform random distribution at an interval of 80 ms. The base voltage of the circuit is set to 115 kV with three phases. The base frequency is 60 Hz and the voltage angle is 30 degrees.

Communication Network

The communication network places network devices on all 16 *node* locations of the electrical circuit. The co-ordinates of the network devices are calculated approximately following their locations in Figure 5.1, with *node 646* and *node 611* as the left-most co-ordinates ($x = 0$) and *node 680* as the bottom-most

co-ordinate ($y = 0$). These co-ordinates were calculated manually and then included in the ontology using the *locatedAt* property. The links between the *nodes* follow the power system topology, each having data rates of 512 $Kbps$, delays of 15 ms, and error rates of 0.0001. The error rate may be set to bit, byte, or packet error rates, as per user requirement and is dependent on the underlying protocol of the link. However, the current ns-3 implementation can only simulate packet error rates and thus the default type of error rate (i.e., packet) is used. The links are wired and may follow IPv4 or IPv6, coupled with P2P or CSMA (i.e., Ethernet-like) protocols.

Controller Settings

The Controller simulator generates tap control values with their associated timestamp of generation. The set of control values $C = \{-1, 0, 1\}$ represent tap up, no action, and tap down respectively. The set voltage point is $V_{set} = 2178\ V$, the specified bandwidth is $BW = 13.6125$, the intentional delay is $t_{delay} = 60\ ms$, and the interval between control data generation is 200 ms.

5.1.2 Distributed State Estimation

For the state estimation example, we utilize a modified version of the IEEE 33 node test feeder circuit [51] as the *primary* network and we attach to each of the 32 primary *nodes* (except the first one) the IEEE European low voltage test feeder [52]. The first *node* of the primary network is the source of electricity and has a generator attached to it. The remaining *primary nodes* are connected to their respective *secondary* networks through a step-down transformer as shown in Figure 5.3. Therefore, the total number of *nodes* in the network is $33 + 32 \times 55 = 1793$.

Power Flow

State estimation requires data to be collected from many nodes in a large network. Therefore, for testing the performance limits of our simulation platform, we place `Phasors` at every *primary node*, and we place `Smartmeters` at every *secondary node*. The data generation of Supervisory control and data acquisition (SCADA) Phasor Measurement Units (PMUs) is typically 2 to 4

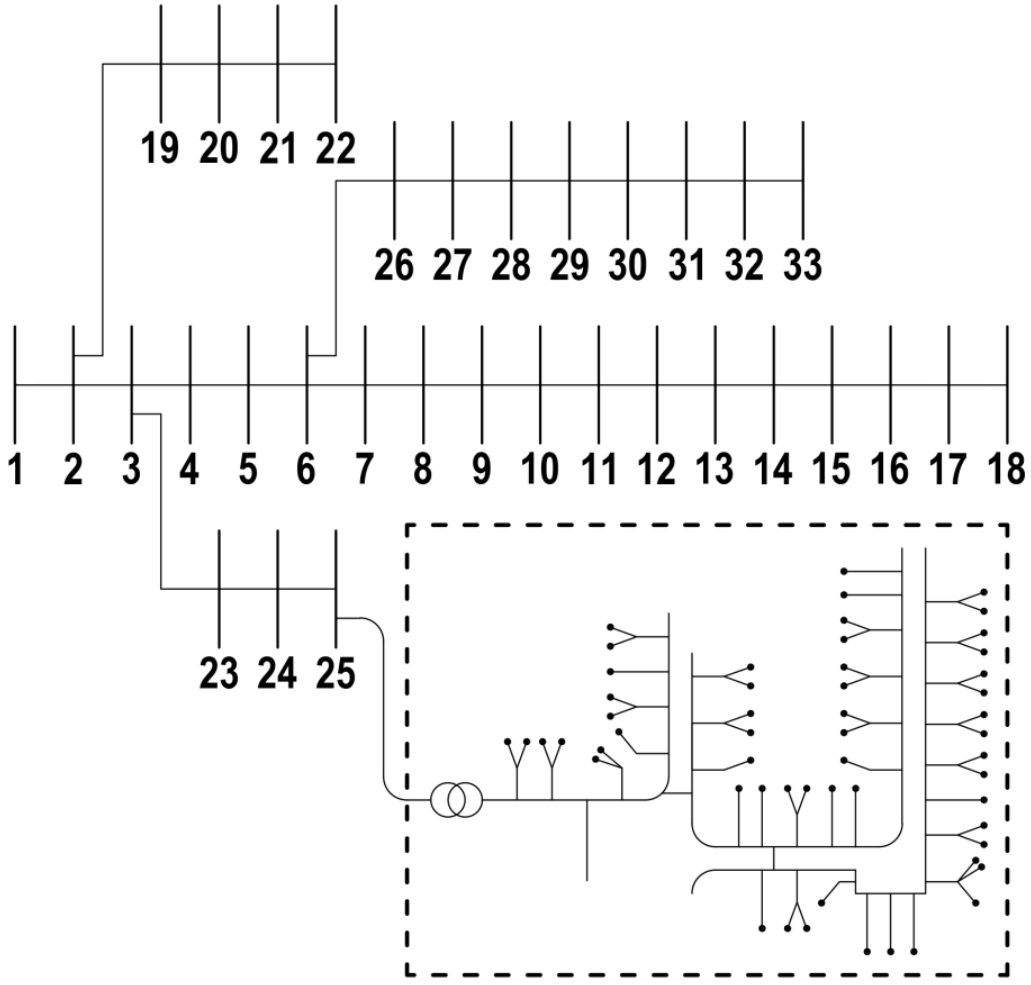

Figure 5.3: IEEE 33 node test feeder modified with added European test feeders [47]. Only one of the secondary networks is shown in the figure.

.

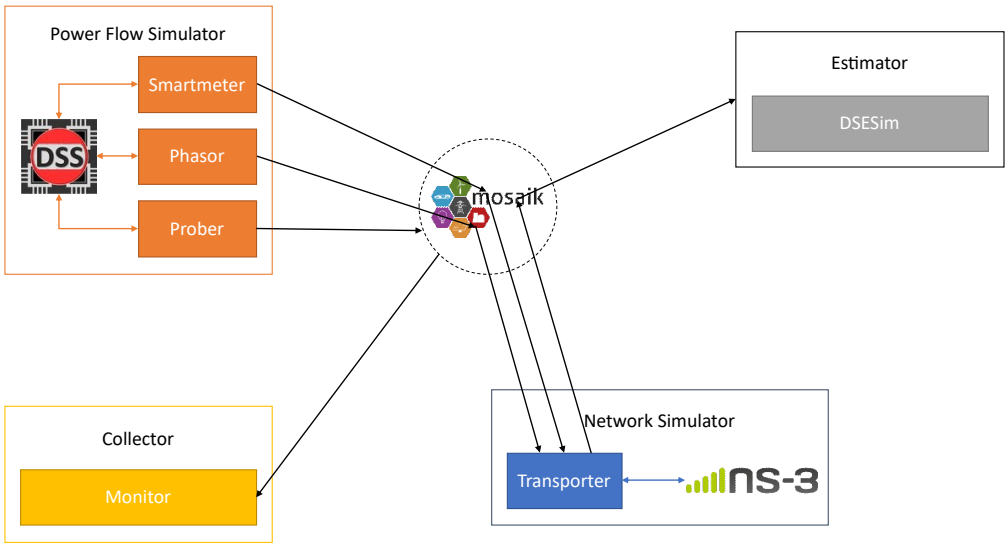

Figure 5.4: State Estimation data flow among simulation *models*.

seconds [53]. Therefore, we set the `Phasor` measurement rates as 2 $seconds$. Real-world smart-meter sampling rates are typically equal to, or longer than, 15 $minutes$ [47]. However, we set the data generation rate of `Smartmeters` to 15 $seconds$ for stress testing of our platform. A total of 8030 sets of sensory data are generated per minute of simulation time, all of which are intended for the Estimator. The base circuit voltage is 12.66 kV and the default frequency is 60 Hz. The loads set for the *secondary* network are generated by linearly interpolating data-sets of residential loads and adding noise to the data as done in [47]. The *primary* network loads are set according to the power consumption data provided in [51]. The residential loads set at the *secondary* network have been added randomly i.e., the houses have been connected to the *secondary nodes*, until the sum of the loads match the data given in the 33-bus system data. The loads are fetched from load sets and assigned to the $devices$ at an interval of 1000 ms.

Communication Network

The number of network devices is 1793, each placed at the location of the respective *nodes* in the electrical network. The co-ordinates are specified in a similar way to the tap control example, following the placement of devices in the electrical network as shown in Figure 5.3. The communication links have

a delay of 1 ms, transmission rates of 1024 $Kbps$, and error rates of 0.0001, when the links are wired and follow the same protocol combination as the tap control application. When the protocol applied is $6LoWPAN$, the underlying links become wireless and follow $LR-WPAN$ protocols [54] instead of the parameters mentioned above. The delay is simulated using a constant speed propagation delay model and the loss of data is simulated using a log distance propagation loss model. The error rates depend on the loss model and the collisions simulated by the implemented Medium Access Control (MAC) protocol - CSMA/CA. The current ns-3 LR-WPAN physical layer is implemented following various IEEE standards and the ATMEL's AT86RF233 device, and thus the transmission rate depends on the power spectral density of the simulated medium and the transmission capabilities of the modelled transceiver antenna [55]. The default data rate of the corresponding standard i.e., IEEE 802.15.4-2006, is 250 $Kbps$ [56].

There are different stages of refinement that are triggered using the `#define` variable `PERFORMANCE_TEST` to perform conditional compilation, where the value 0 indicates the highest level of refinement or stage 3, the value 1 indicates stage 2, and the value 3 indicates stage 1 (refer to Figures 4.15 and 4.16). The relevance filtering can be turned on for MOSAIK 2 with the value 2, however this does not affect the simulation as the steps are exhaustive with no room for refinement.

Estimator

The Estimator maintains two periods - the accounting period and the estimation period. The accounting period is the interval between message recordings to indicate how many messages were received by the estimator during the interval. The estimation period is the interval between each state estimation, which is typically longer than the accounting interval. We set the accounting period to 200 ms, and the estimation period to 1000 ms.

5.2 Simulation Results

We evaluate the performance of the different stages of refinement for both scenarios. For all our experiments, we utilize the same Ubuntu host (`Ubuntu`

20.04.4 LTS, Linux Kernel: 5.13.0-51-generic) running on an Intel™ Pentium™ G3220 (clocked at 3 GHz). G3220 is a 2-core Central Processing Unit (CPU). The physical memory of the host is 4 GB DDR3, which, as we observed, was adequate to accommodate the working set for the co-simulation processes. Note that the co-simulation is single-threaded with physical memory usage below 700 MB for all test runs. Apart from the simulations, the host was executing no other resource–intensive task, but it was running background maintenance tasks typically found in Ubuntu 20.04 installations. As there were two cores, and the background tasks were light, the impact of the background tasks on the (single-threaded) execution of the co-simulator was insignificant. The G3220 processor is an older vintage processor, and hence the results presented here capture a worse case scenario compared to the performance one expects from current CPUs.

Initially, we compare the execution time of the different refinement stages of the scenarios with varying sensor devices. We run 40 *second* simulations of the tap control application with 4 Probers, 1 Actuator with its corresponding RangeControl instance, and varying Sensors located throughout the network. All sensors have data generation rates of one per 100 ms, and control actions are generated once every 200 ms. We perform 10 test runs for every test case and present the average of all runs with maximum and minimum execution times represented as error bars in Figure 5.5. For all stages of refinement, the average execution time increases with the number of Sensors due to the increase in relevant event generation. If the overall execution time of the simulation is less than 40 *seconds*, then the simulation is faster than the duration simulated. For the lowest stage of refinement, the execution time exceeds the simulated time in certain cases (with 3 and 4 Sensors). Stage 2 refinement drastically improves the execution time due to significant decrease in stepping of simulators compared to the exhaustive lock steps of stage 1. Relevance filtering of ns-3 events (stage 3) improves the execution time further. For all stages, the increase of events generated due to the increasing number of Sensors, increases execution time linearly.

Applying the same simulation time for state estimation, we generate two test cases for execution time comparison of two stages of refinement. Due to

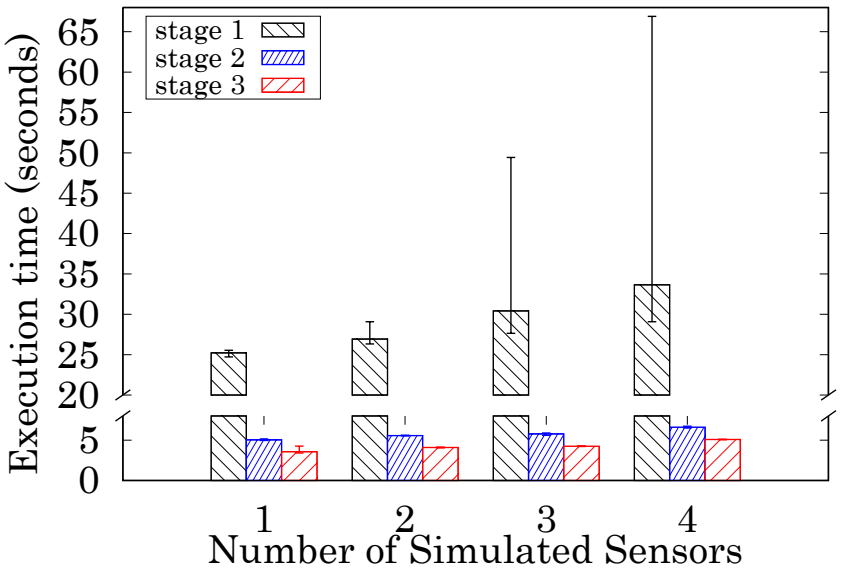

Figure 5.5: Execution time of the overall simulation of the tap control scenario for different stages of refinement.

the larger scale of the distribution network, we do not simulate stage 1 for state estimation, as the execution time of the latter stages already exceed significantly the simulated time. For the first test case we place 4 `Phasors` at the *primary network*, namely at *nodes* 1, 31, 32, and 33. Note that the estimator is located at *node* 1, and therefore the `Phasor` at that location does not require passing messages through the communication network. We set the data generation rate of the `Phasors` to 1 per 100 ms. We perform 10 test runs and present the average execution times along with the maximum and minimum times in Figure 5.6. The improvement of relevance filtering is negligible in this case as the number of events generated in ns-3 are small in number. However, for the second test case, we place `Phasors` at all *primary nodes* and `Smartmeters` at all *secondary nodes*. We set the data generation rates of `Smartmeters` and `Phasors` to one per 15 seconds and 2 seconds, respectively. The `DSESim` *model* performs accounting every 200 ms and the state of the system is estimated once every 1000 ms. Consequently, the execution time increases significantly due to the large number of events generated at the power flow and communication network - 33 `Phasor` readings per 2 seconds and 1760 `Smartmeter` readings per 15 seconds. Therefore, the relevance filtering of stage 3 reduces the execution time by approximately 59.4%. We further

study the simulation specific execution times to understand the remarkable improvement due to the fine-tuning of a single simulator.

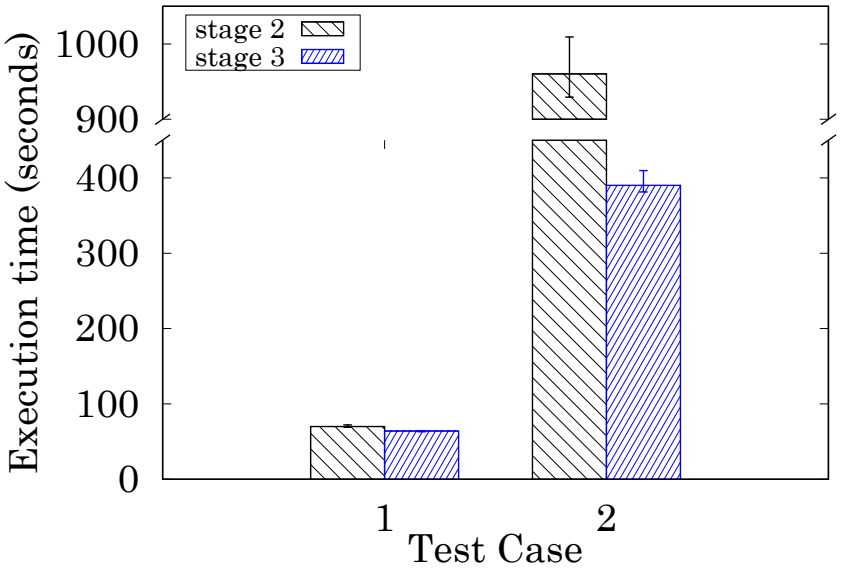

Figure 5.6: Execution time of the overall simulation of the state estimation scenario for two stages of refinement.

We calculate the execution time of the network simulator as the time duration between the reception of a JSON message from `mosaik` to the returning of the result generated by the appropriate API function. MOSAIK does not consider the initialization of simulators and their respective *models* when calculating simulation execution time. Therefore, we only include the `step` and `get_data` API functions in our calculation, however, the parsing and forming of JSON messages before and after applying the API function is included as well. For other simulators, the execution time is calculated as the time duration of the *steps* and fetching of data after every step. MOSAIK may fetch data multiple times from a simulator after a step is executed. The execution time of MOSAIK is calculated by subtracting the times of the participating simulators from the total execution time. The average simulator execution times represented in Figure 5.7 are for the first of the two test cases of the State Estimation scenario mentioned previously. The execution times are averaged, and the maximum and minimum values are collected, from the same 10 test runs as in Figure 5.6. The results indicate that the Power Flow simulator and Estimator consume the majority of the computational time due to the pres-

ence of a large number of *devices* in the circuit. The network simulator has comparatively faster execution due to fewer messages being generated from the *devices*. Being responsible for only the collection of data and with little processing, the Collector is faster than all other simulators. MOSAIK execution time is average compared to the other simulators as its execution is dependent on all the simulators. The refinement of ns-3 in stage 3 leads to improvement of execution times of MOSAIK and ns-3. The remaining simulators have similar execution times indicating a lack of any effect. This is because the fetching of data from a simulator is relatively less heavy on resources, and thus reduction of the number of data fetching actions (due to reduction of stepping of the Network Simulator) does not affect the Power Simulator. Furthermore, the generation of relevant data by ns-3 is unchanged as well. This results in the same number of external events being generated for the Estimator, which explains its unchanged execution time. The Collector does not collect data from ns-3 and so its performance is also unchanged. We present the average percentage of the overall simulation time taken by the constituent simulators in Figure 5.8. Power Flow contributes the most in terms of execution times - 41.1% and 45% for stages 2 and 3 respectively. The Estimator follows in second with 39% and 42.7, respectively. Therefore, for case 1 of state estimation, less than 20% of the overall simulation time is attributed to the remaining simulators.

For the second test case, we are able to better verify the effect of refinement on ns-3 due to the increase in the number of messages generated in the packet network. We collected the execution times as represented in Figure 5.9, from the same 10 test runs as Figure 5.6. The average execution times of the Power Flow simulator has increased compared to the first test case as more data is generated and fetched by ns-3. The Estimator execution time is slightly reduced due to the large size of the measurement vector when calculating the state variables using the weighted least squares method (WLS). The large size of the measurement vector is a result of more *devices* providing readings to the Estimator. Consequently, fewer values are required to be estimated. The Collector is the fastest among the simulators similar to the previous test case, with a slight increase in execution times. The majority

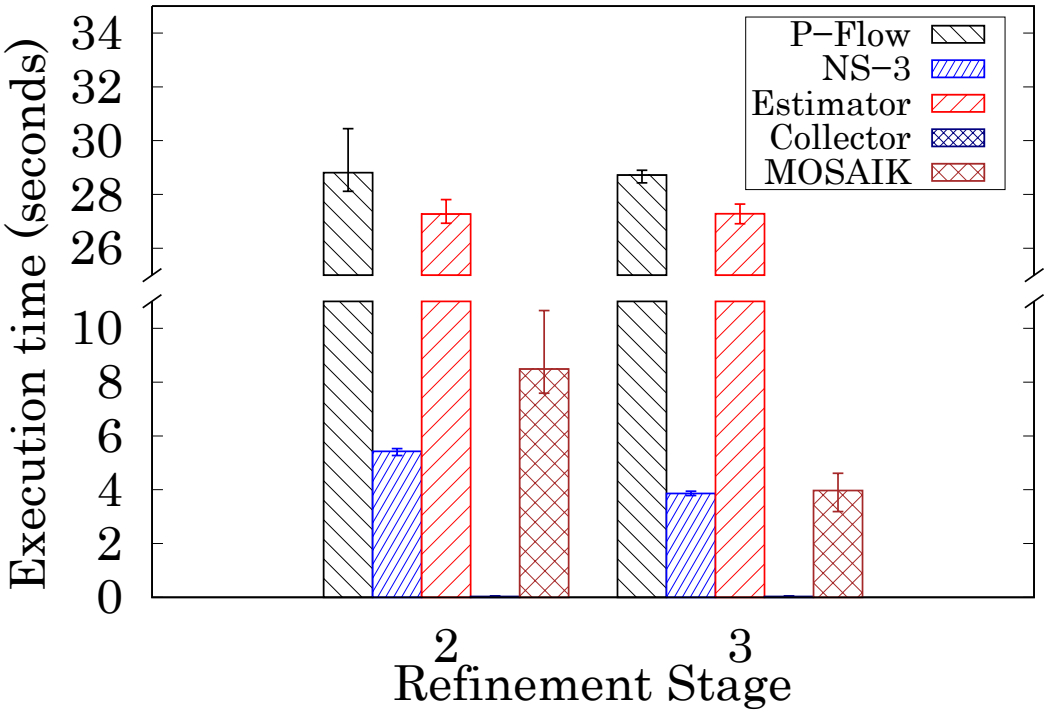

Figure 5.7: Execution time of participating simulators and co-simulation platform for test case 1 of state estimation. The total execution time for stage 2 refinement is 70 seconds and for stage 3 refinement is 63.9 seconds.

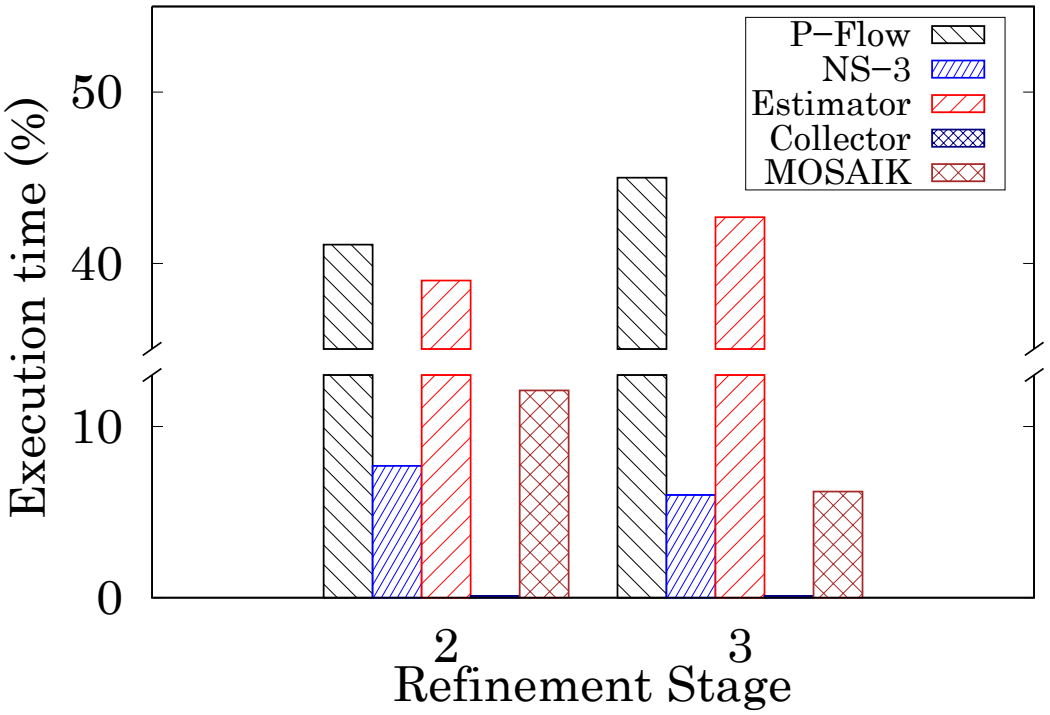

Figure 5.8: Execution time in percentage of participating simulators and co-simulation platform for test case 1 of state estimation.

of the overall execution time is attributed to ns-3 and MOSAIK. For stages 2 and 3, ns-3 execution times are approximately 61.8% and 56.3% of the overall execution time, respectively, being the slowest of the participating simulators (refer to Figure 5.10). The Collector execution time is negligible (almost 0.0%) compared to other simulators in case 2 (0.1% in case 1). `mosaik` contributes by 31.9% and 28.4% having the second greatest execution time. This is because - higher number of steps for ns-3 means more processing for MOSAIK. Therefore, the relevance filtering of ns-3 events reduces its execution time by a remarkable 63%, and that of MOSAIK by 63.8%, resulting in an overall reduction of 59.4%.

Test Case	Refinement	P-Flow	NS-3	Controller	Collector
1	stage 1	40000	40000	40000	40000
	stage 2	592	6262	579	592
	stage 3	592	1682	579	592
2	stage 1	40000	40000	40000	40000
	stage 2	599	6651	963	599
	stage 3	599	2337	963	599

Table 5.1: Steps taken by the simulators in the tap control example.

Test Case	Refinement	P-Flow	NS-3	Estimator	Collector
1	stage 2	400	16417	1600	400
	stage 3	400	2818	1600	400
2	stage 2	21	31034	6021	21
	stage 3	21	11133	6021	21

Table 5.2: Steps taken by the simulators in the state estimation example.

We study the change in the total number of steps executed by the simulators in the Tables 5.1 and 5.2. For the tap control scenario, we compare the step counts for all stages of refinement of case 1 and 2 (Table 5.1). Note that the number of steps for all simulators in stage 1 is 40000, which is the total number of time units for a 40 *second* simulation. The step counts are significantly reduced for all simulators in stage 2. For stage 3, the step count of ns-3 is reduced while the remaining simulators have the same number of steps. Higher number of steps taken does not necessarily mean greater execution times. For example, let us consider the step counts of cases 2 and 3 of state

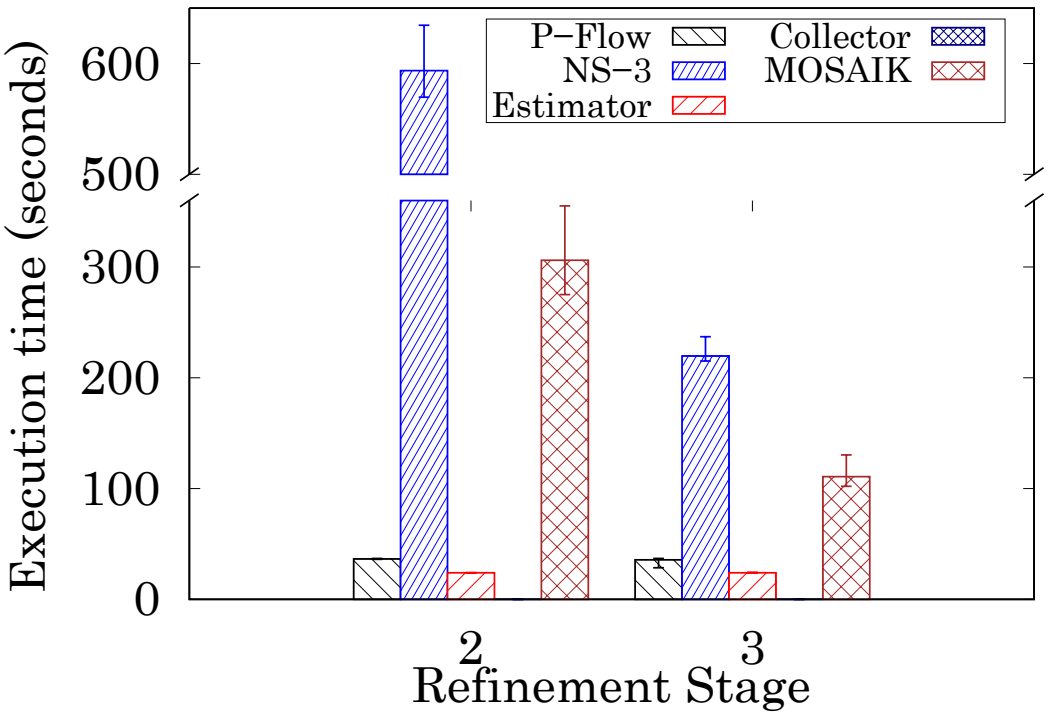

Figure 5.9: Execution time of participating simulators and co-simulation platform for test case 2 of state estimation. The total execution time for stage 2 refinement is 960.2 seconds and for stage 3 refinement is 390 seconds.

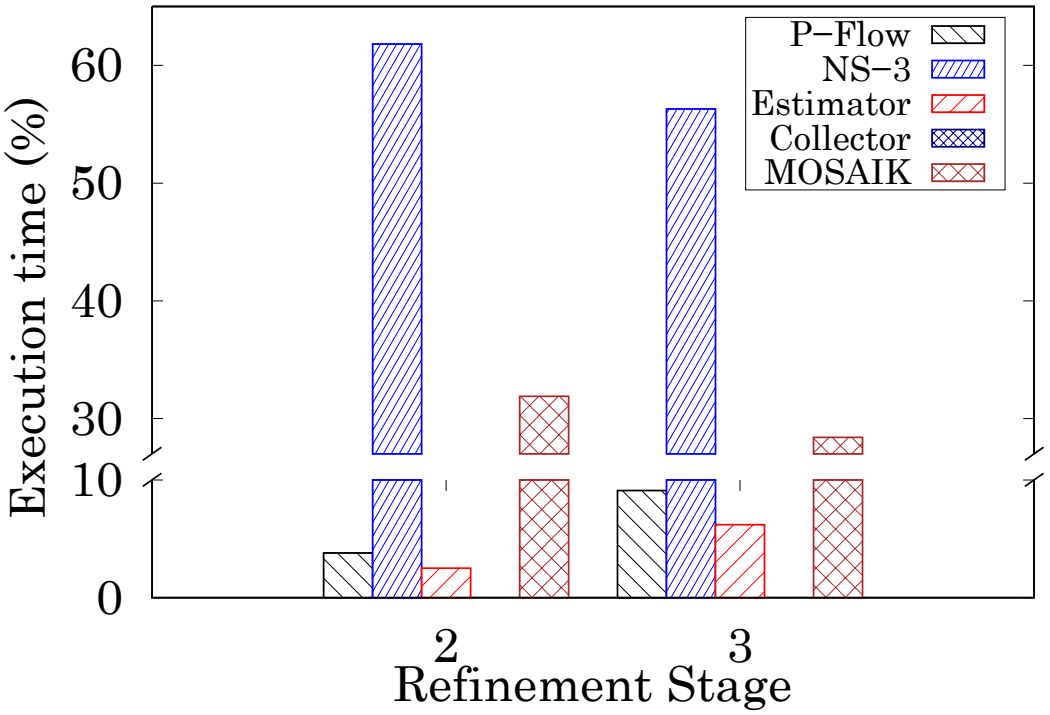

Figure 5.10: Execution time in percentage of participating simulators and co-simulation platform for test case 2 of state estimation.

estimation. The Power Flow simulator execution time is greater in test case 2 of state estimation (as shown in Figure 5.9) compared to test case 1 (as shown in Figure 5.7) due to the larger number of *devices* generating data. However, the frequent rate of data generation in test case 1 yields a higher number of steps. The step count of the Estimator is affected by its own periodic accounting and estimation cycles, and by the reception of data from ns-3. Therefore, for a test case with large amount of generated data, the Estimator requires more frequent stepping compared to the one with fewer data. The step count of ns-3 is reduced greatly as irrelevant data are processed without notifying MOSAIK, thus reducing the execution times of both MOSAIK and ns-3.

Chapter 6

Conclusion

Our co-simulation platform facilitates high fidelity simulation - improving execution without compromising the accuracy of the simulation. The logical time lower bound computation within the simulators allow for an efficient approach to processing internal events without breaking the causal order. The filtering of irrelevant events in the packet network simulator further improves execution by reducing redundant time management and execution. The simulation of large networks with several electrical equipment, generating messages in the communication network, benefit greatly from the mentioned refinement. However, the execution time or wall-clock time of co-simulation exceeds the simulated time for larger networks. Therefore, the processing of the overall simulation is demanding in terms of processing.

Communication networks associated with city-wide electrical networks of smart-grids having innumerable electrical devices generate frequent and large number of packets. In such cases, our refinement techniques can greatly reduce the simulation times of these networks. This allows the efficient use of more complex algorithms, machine learning techniques, enhanced cyber-security applications, and rigorous testing before deployment of a smart-grid network.

6.1 Limitations

There are certain limitations associated with our co-simulation platform. The first limitation is related to the distribution of events within the duration of simulation. The second limitation is the requirement of transparency among

participating simulators.

6.1.1 Limitation 1: Event Count

Although, the lower logical time bound computation (LBTS) reduces the number of times a simulator needs to be interrupted during the simulation, this reduction depends on the number of events and their timestamps. For example, consider the following two cases where we consider as “step" of a simulator the continuous execution between two successive points of blocking the corresponding simulator process:

1. There are two simulators A and B, where both simulators provide input to each other, and A has 10 events at timestamp 1, for a 10 time unit simulation. In this case, A can execute all the events at the beginning and provide the processed data to B. Considering that A has a higher priority and can execute first, B can then execute after receiving data from A. If B has no events output for A after processing all inputs, B can execute 10 time units of simulation without interrupting A. Similarly A can execute the remaining 9 time units of simulation without interrupting B. Therefore, simulator A requires two steps and B requires one step, to complete this simulation.

2. In the second case, consider that A has the same number of total events, i.e., 10. However, each event has a different timestamp, starting from 1 and ending at 10. In this case, A needs to simulate the first event with timestamp 1 and provide data to B. B executes time step 1 with the collected input, however, it cannot proceed further as A has another event at time 2. A simulates the event at time 2 with the input from B (if any) and repeats the process. This continues with both simulators being interrupted at every time unit to execute one event. The total number of steps for each simulator is 10.

Typical simulations do not have all events at any particular time unit, rather events are scattered throughout the duration of the simulation. However, co-simulation with simulators containing several *model* instances generating multiple events at different timestamps may have events at many times-

tamps. In such cases, lower bound computation or relevant event filtering may not be enough to significantly reduce execution time or it reduces it only modestly. In the worst case, if events are present at every single timestamp, refinements 2 and 3, may result in a greater execution time compared to an exhaustive simulation as done in stage 1.

6.1.2 Limitation 2: Inter-simulator Transparency

For refinement stages 2 and 3, the next event timestamp of the simulators need to be known beforehand. The *max_advance* time required by MOSAIK is used by simulators and MOSAIK to execute simulator events while maintaining causal order. However, this transparency may not be present in certain simulators, or may be inaccurate in others. For example, in the Tap Control scenario, if the Controllers *models* take immediate control decisions instead of periodic ones, the next time step is not known to the Controller, and thus unknown to the other simulators and MOSAIK. A new event may be generated as soon as a `RangeControl` model instance receives data from a `Transporter` model instance. This may be solved if all future events of the packet network is known and provided to the Controller to generate its next event timestamp. However, this level of transparency is typically absent in simulators. In fact, for this reason, stage 3 executes future events to determine the timestamp of the next relevant event as the internal event queue of ns-3 needs to be accessed to do the same without executing irrelevant events.

6.2 Future Work

Future enhancements of the our co-simulation platform may include converting the single-threaded simulation into a multi-threaded one. This may be done by assigning separate threads to participating simulators or by parallel execution of the simulators themselves. The first approach requires multi-threaded capabilities provided by the co-simulation platform. However, if the overall execution time of the co-simulation is dominated by a single simulator as we saw in a number of instances, assigning a single thread to the mentioned simulator may hardly reduce overall execution time. The sec-

ond approach may be more appropriate in such cases and would require the simulators to have parallel computing capabilities as well. Certain simulators like ns-3 already have experimental modules which allow multi-threaded simulation. Incorporating that into the co-simulation platform and other computation-intensive simulators is a future endeavour.

With the advent of electric vehicles (EVs) and unmanned aerial vehicles (UAVs), network devices and consumers of electricity have attained high mobility. Simulating a modern smart-grid requires taking into consideration the unpredictable nature of their movements and the load put on the power system to charge these vehicles. In future work, mobility of the devices in the communication network through the mobility module of ns-3 could be incorporated. The vehicles would also consume energy at charging stations, thus requiring data to be collected from these vehicles to better prepare the stations with necessary power beforehand. Storing energy in large amounts at such stations may alleviate the unpredictable load requirements, however, the limited generation and distribution of power coupled with its low storage efficiency makes this an unfeasible and expensive solution. Therefore, planning and prediction of load requirements is a more feasible solution, one that requires a communication network spanning the entire network of mobile and static *nodes*. The mobility simulation of these *nodes* may be achieved through addition of well-known vehicle simulators into the list of simulators participating in co-simulation.

References

[1] J. Banks, J. Carson, B. Nelson, and D. Nicol, *Discrete-Event System Simulation*, 5th ed. Prentice Hall, 2010, ISBN: 0136062121.

[2] R. Eckhardt, "Stan Ulam, John von Neumann, and the Monte Carlo method," *Los Alamos Science*, vol. 15, no. 131-136, p. 30, 1987.

[3] J. F. Robeson, *Logistics handbook*. Simon and Schuster, 1994, ISBN: 9781451665697.

[4] D. R. Jefferson and P. D. Barnes, "Virtual time III: Unification of conservative and optimistic synchronization in parallel discrete event simulation," in *2017 Winter Simulation Conference (WSC)*, IEEE, 2017, pp. 786–797.

[5] P. Palensky, A. A. Van Der Meer, C. D. Lopez, A. Joseph, and K. Pan, "Cosimulation of intelligent power systems: Fundamentals, software architecture, numerics, and coupling," *IEEE Industrial Electronics Magazine*, vol. 11, no. 1, pp. 34–50, 2017.

[6] IEEE, "IEEE standard for modeling and simulation (M&S) high level architecture (HLA) – framework and rules - redline," *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000) - Redline*, pp. 1–38, 2010. DOI: 10.1109/IEEESTD.2010.5953411.

[7] IEEE, "IEEE standard for modeling and simulation (M&S) high level architecture (HLA)– federate interface specification," *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, pp. 1–378, 2010. DOI: 10.1109/IEEESTD.2010.5557728.

[8] IEEE, "IEEE standard for modeling and simulation (M&S) high level architecture (HLA)– object model template (OMT) specification," *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)*, pp. 1–110, 2010. DOI: 10.1109/IEEESTD.2010.5557731.

[9] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, "The Department of Defense high level architecture," in *Proceedings of the 29th conference on Winter simulation*, 1997, pp. 142–149.

[10] IEEE, "IEEE standard for modeling and simulation (M&S) high level architecture (HLA)-framework and rules," *1516-2010*, 2010.

[11] H. Lin, S. Sambamoorthy, S. Shukla, J. Thorp, and L. Mili, *IEEE PES innovative smart grid technologies (ISGT)*, 2011.

[12] H. Georg, S. C. Müller, N. Dorsch, C. Rehtanz, and C. Wietfeld, "INSPIRE: Integrated co-simulation of power and ict systems for real-time evaluation," in *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, IEEE, 2013, pp. 576–581.

[13] K. Mets, T. Verschueren, C. Develder, T. L. Vandoorn, and L. Vandevelde, "Integrated simulation of power and communication networks for smart grid applications," in *2011 IEEE 16th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, IEEE, 2011, pp. 61–65.

[14] *The Ptolemy Project*, [accessed 3-Aug-2022]. [Online]. Available: `https://ptolemy.berkeley.edu/index.htm`.

[15] J. Eker, J. W. Janneck, E. A. Lee, *et al.*, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[16] C. Brooks and E. Lee, "Ptolemy II: An open-source platform for experimenting with actor-oriented design," in *Berkeley EECS Annual Research Symposium (BEARS)*, vol. 3, 2016.

[17] *Functional Mock-Up Interface for Model Exchange and Co-Simulation 3.0*, [accessed 11-Aug-2022]. [Online]. Available: `https://fmi-standard.org/`.

[18] M. U. Awais, W. Gawlik, G. De-Cillia, and P. Palensky, "Hybrid simulation using SAHISim framework," *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems*, vol. 3, no. 9, e2–e2, 2016.

[19] V. Galtier, S. Vialle, C. Dad, J.-P. Tavella, J.-P. Lam-Yee-Mui, and G. Plessis, "FMI-based distributed multi-simulation with DACCOSIM," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, 2015, pp. 39–46.

[20] H. Lin, S. S. Veda, S. S. Shukla, L. Mili, and J. Thorp, "GECO: Global event-driven co-simulation framework for interconnected power system and communication network," *IEEE Transactions on Smart Grid*, vol. 3, no. 3, pp. 1444–1456, 2012.

[21] H. Lin, Y. Deng, S. Shukla, J. Thorp, and L. Mili, "Cyber security impacts on all-PMU state estimator-a case study on co-simulation platform GECO," in *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*, IEEE, 2012, pp. 587–592.

[22] H. Georg, S. C. Müller, C. Rehtanz, and C. Wietfeld, "Analyzing cyber-physical energy systems: The inspire cosimulation of power and ict systems using hla," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2364–2373, 2014.

[23] K. Hopkinson, X. Wang, R. Giovanini, J. Thorp, K. Birman, and D. Coury, "Epochs: A platform for agent-based electric power and communication simulation built from commercial off-the-shelf components," *IEEE Transactions on Power Systems*, vol. 21, no. 2, pp. 548–558, 2006.

[24] J. Nutaro, P. T. Kuruganti, L. Miller, S. Mullen, and M. Shankar, "Integrated hybrid-simulation of electric power and communications systems," in *2007 IEEE Power Engineering Society General Meeting*, IEEE, 2007, pp. 1–8.

[25] W. Li, A. Monti, M. Luo, and R. A. Dougal, "VPNET: A co-simulation framework for analyzing communication channel effects on power systems," in *2011 IEEE Electric Ship Technologies Symposium*, IEEE, 2011, pp. 143–149.

[26] D. Anderson, C. Zhao, C. Hauser, V. Venkatasubramanian, D. Bakken, and A. Bose, "Real-time simulation for smart grid control and communications design," *IEEE Power Energy Mag*, vol. 10, no. 1, pp. 49–57, 2012.

[27] V. Liberatore and A. Al-Hammouri, "Smart grid communication and cosimulation," *IEEE 2011 EnergyTech*, pp. 1–5, 2011.

[28] J. Bergmann, C. Glomb, J. Götz, J. Heuer, R. Kuntschke, and M. Winter, "Scalability of smart grid protocols: Protocols and their simulative evaluation for massively distributed DERs," in *2010 First IEEE International Conference on Smart Grid Communications*, IEEE, 2010, pp. 131–136.

[29] D. Babazadeh, M. Chenine, K. Zhu, L. Nordström, and A. Al-Hammouri, "A platform for wide area monitoring and control system ICT analysis and development," in *2013 IEEE Grenoble Conference*, IEEE, 2013, pp. 1–7.

[30] D. Babazadeh and L. Nordström, "Agent-based control of VSC-HVDC transmission grid-a cyber physical system perspective," in *2014 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, IEEE, 2014, pp. 1–6.

[31] M. Wei and W. Wang, "Greenbench: A benchmark for observing power grid vulnerability under data-centric threats," in *IEEE INFOCOM 2014-IEEE conference on computer communications*, IEEE, 2014, pp. 2625–2633.

[32] S. C. Müller, H. Georg, J. J. Nutaro, *et al.*, "Interfacing power system and ICT simulators: Challenges, state-of-the-art, and case studies," *IEEE Transactions on Smart Grid*, vol. 9, no. 1, pp. 14–24, 2016.

[33] K. Hopkinson, R. Giovanini, X. Wang, K. Birman, J. Thorp, and D. Coury, *The electric power and communication synchronizing simulator (EPOCHS)*, [accessed 11-Aug-2022]. [Online]. Available: `http://www.cs.cornell.edu/hopkik/epochs.htm`.

[34] J. J. Nutaro, *Building software for simulation: theory and algorithms, with applications in C++*. John Wiley & Sons, 2011.

[35] S. Schütte, S. Scherfke, and M. Tröschel, "Mosaik: A framework for modular simulation of active components in smart grids," in *2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS)*, IEEE, 2011, pp. 55–60.

[36] S. Rohjans, S. Lehnhoff, S. Schütte, S. Scherfke, and S. Hussain, "Mosaik-a modular platform for the evaluation of agent-based smart grid control," in *IEEE PES ISGT Europe 2013*, IEEE, 2013, pp. 1–5.

[37] C. Steinbrink, A. A. van der Meer, M. Cvetkovic, *et al.*, "Smart grid co-simulation with MOSAIK and HLA: A comparison study," *Computer Science-Research and Development*, vol. 33, no. 1, pp. 135–143, 2018.

[38] R. M. Fujimoto, "Time management in the high level architecture," *Simulation*, vol. 71, no. 6, pp. 388–400, 1998.

[39] MOSAIK *API documentation*, [accessed 28-Aug-2022]. [Online]. Available: `https://mosaik.readthedocs.io/en/latest/mosaik-api/index.html`.

[40] MOSAIK *communication with simulators using API*, [accessed 09-Sept-2022]. [Online]. Available: `https://mosaik.readthedocs.io/en/latest/mosaik-api/overview.html#how-mosaik-communicates-with-a-simulator`.

[41] R. C. Dugan, "Reference guide: The open distribution system simulator (opendss)," *Electric Power Research Institute, Inc*, vol. 7, p. 29, 2012.

[42] C. Starr, "The electric power research institute," *Science*, vol. 219, no. 4589, pp. 1190–1194, 1983. DOI: `10.1126/science.219.4589.1190`. [Online]. Available: `https://www.science.org/doi/abs/10.1126/science.219.4589.1190`.

[43] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*, Springer, 2010, pp. 15–34.

[44] *Network simulator 3.33 manual*, [accessed 31-Aug-2022]. [Online]. Available: `https://www.nsnam.org/docs/release/3.33/manual/singlehtml/index.html`.

[45] B. A. Forouzan, *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[46] M. Hartung, E.-M. Baerthlein, and A. Panosyan, "Comparative study of tap changer control algorithms for distribution networks with high penetration of renewables," in *CIRED Workshop*, 2014, pp. 1–5.

[47] M. M. Haji and O. Ardakanian, "Practical considerations in the design of distribution state estimation techniques," in *2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, IEEE, 2019, pp. 1–6.

[48] A. Abur and A. G. Exposito, *Power system state estimation: theory and implementation*. CRC press, 2004.

[49] G. Sybille, G. Shirek, and W. Kersting, *IEEE 13 node test feeder*, [accessed 28-Sep-2022]. [Online]. Available: https://www.mathworks.com/help/physmod/sps/examples/ieee-13-node-test-feeder.html.

[50] E. De Souza, O. Ardakanian, and I. Nikolaidis, "A co-simulation platform for evaluating cyber security and control applications in the smart grid," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, IEEE, 2020, pp. 1–7.

[51] M. E. Baran and F. F. Wu, "Network reconfiguration in distribution systems for loss reduction and load balancing," *IEEE Power Engineering Review*, vol. 9, no. 4, pp. 101–102, 1989.

[52] *IEEE PES distribution systems analysis subcommittee's radial test feeders*, [accessed 14-Aug-2022]. [Online]. Available: https://cmte.ieee.org/pes-testfeeders/resources/.

[53] *Phasor measurement unit*, [accessed 15-Aug-2022]. [Online]. Available: https://en.wikipedia.org/wiki/Phasor_measurement_unit.

[54] *Network simulator 3 (ns-3) low-rate wireless personal area network (LR-WPAN)*, [accessed 16-Aug-2022]. [Online]. Available: https://www.nsnam.org/docs/models/html/lr-wpan.html.

[55] *Network simulator 3 (ns-3) low-rate wireless personal area network (LR-WPAN) physical layer model (PHY)*, [accessed 16-Aug-2022]. [Online]. Available: https://www.nsnam.org/docs/models/html/lr-wpan.html#phy.

[56] IEEE, "IEEE standard for information technology– local and metropolitan area networks– specific requirements– part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low rate wireless personal area networks (WPANs)," *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pp. 1–320, 2006. DOI: 10.1109/IEEESTD.2006.232110.