Institute for High Voltage Technology

RWTH AACHEN UNIVERSITY

This work was submitted to the Institute of High Voltage Technology

Master Thesis

by

Herrn Markus Stroot

Co-Simulation of distributed flexibility coordination schemes

Co-Simulation dezentraler Flexibilitätskoordinationsmechanismen

Examiner:	Dr.-Ing. Ralf Puffer
Supervisor:	Thomas Offergeld, M. Sc. Immanuel Hacker, M.Sc.
Date of Submission:	22. May 2020

Declaration

Name, Vorname | Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Co-Simulation dezentraler Flexibilitätskoordinationsmechanismen

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum | Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum | Unterschrift

Abstract

For many years now, there has been an unmistakable trend in the energy sector towards Renewable Energy Sources (RES). This has been encouraged both by political measures as well as general public awareness of environmental issues tied to the usage of fossil or nuclear energy sources. However, this leads to an increased volatility of the overall residual demand.

In that context, the flexibility provided by participants on all levels of the grid takes an increasingly important role. Especially small, private sources of flexibility are usually inaccessible for congestion reduction measures. In order to leverage that flexibility potential new control strategies must be designed and implemented. These strategies need to encourage local energy consumption and peak reduction.

The development of such a control system is complex task involving multiple engineering domains. In order to provide a platform on which such a collaborative development can take place a co-simulation environment is implemented and tested in this thesis. The environment uses a co-simulation framework, which enables the connection and combined execution of multiple simulators. To realize individual runtime environments, each simulator is packaged into a separate software container. The container deployment is automated and incorporated into an extension of the framework API.

To test the environment, an exemplary flexibility coordination scheme is designed and simulated within it. Several simulators are implemented in order to realize the individual grid and control components. These include a smart home simulation consisting of load, PV, and storage elements which are augmented by a Model Predictive Control (MPC) based flexibility optimization system. This scenario is scaled up to 20 households with varying flexibility to test a superordinate controller which facilitates inter-household flexibility coordination.

The exemplary results show that the environment works as expected and can handle a variety of simulation scenarios. Concerning the flexibility coordination simulation, the results show how smart flexibility control instead of uncoordinated consumption optimization can decrease the danger of grid congestion. The implemented MPC approach not only lowers the absolute energy demand, but also the maximum power drawn from the grid. Additionally, it shows how the coordination of flexible and static households in conjunction can lead to a grid wide flexibility optimization which single households could not achieve without outside information.

Kurzfassung

Seit vielen Jahren gibt es einen erkennbaren Trend im Energiesektor zugunsten erneuerbarer Energien. Dies wird sowohl durch politische Maßnahmen als auch durch die öffentliche Meinung gegenüber fossiler Brennstroffe und Atomkraft bestärkt. Jedoch führt dies zu einer gesteigerten Variablilität der Residuallast im Energienetz.

Dadurch gewinnt Flexibilität auf allen Netzebenen immer mehr an Bedeutung. Besonders kleine und private Flexibilitätsquellen stehen oft nicht für die Vermeidung von Netzüberlastung zur Verfügung. Um dieses Flexibilitätspotential nutzen zu können, müssen neue Regelungsstrategien entwickelt werden. Diese Startegien sollten lokale Energienutzung und die Vermeidung von Leistungsspitzen begünstigen.

Die Entwicklung eines solchen Regelungssystems ist eine komplexe Aufgabe, welche eine Vielzahl von Ingenieursbereichen vereint. Im Rahmen dieser Abschlussarbeit wird eine Co-Simulationsumgebung implementiert und getestet, welche eine Plattform für gemeinsame Entwicklungsarbeit zur Verfügung stellt. Die Umgebung nutzt ein Co-Simulationsframework, um verschiedene Simulatoren zu verknüpfen und auszuführen. Jeder Simulator erhält eine isolierte Ausführungsumgebung durch den Einsatz von Softwarecontainern. Die Cointainerbereitstellung ist automatisiert und in das Framework-API integriert.

Zum Test der Co-Simulationsumgebung wird ein exemplarischer Flexibilitätsregelungsmechanismus entwickelt und simuliert. Dazu werden unterschiedliche Simulatoren implementiert, die verschiedene Netzelemente darstellen. Zu diesen gehören Haushaltskomponenten, wie Last-, PV- und Speicherelemente, welche durch ein Model Predictive Control (MPC)-basiertes Regelungssystem ergänzt werden. Dieses Szenario wird schließlich auf 20 Haushalte mit variierender Flexibilität hochskaliert, um einen weiteren, übergeordneten Regelungsmechanismus zu testen.

Die exemplarischen Ergebnisse zeigen, dass die Simulationsumgebung funktioniert und unterschiedliche Szenarien verarbeiten kann. Im Zuge der Flexibilitätssimulation kann festgestellt werden, dass intelligente Regelungsstrategien im Gegensatz zu unkoordinierter Eigenverbrauchsoptimierung die Gefahr von Netzüberlastung verringern können. Der implementierte MPC Regelungsansatz optimiert nicht nur den Eigenverbrauch, sondern reduziert auch Leistungsspitzen. Des Weiteren zeigen die Ergebnisse, dass die Regelung von flexiblen Haushalten mit Rücksicht auf statische Teilnehmer zu einer haushaltsübergreifenden Optimierung der Flexibilitätsnutzung führen kann.

Contents

List of Figures

List of Tables

List of Abbreviations

DER	Distributed Energy Resource
RES	Renewable Enegry Source
ESS	Energy Storage System
SH	Smart Home
HEMS	Home Energy Management System
EMS	Energy Management System
MPC	Model Predictive Control
PV	Photovoltaic
SoC	State of Charge
OS	Operating System
VM	Virtual Machine
IP	Intellectual Property
IC	Integrated Circuit
LP	Linear Programming
MILP	Mixed-Integer Linear Programming
ICT	Information and Communication Technology
HLA	High Level Architecture
CPES	Cyber-Physical Energy System
API	Application Programming Interface
JSON	JavaScript Object Notation
PaaS	Platform as a Service
CLI	Command Line Interface
PFE	Power Flow Equation
CSV	Comma Separated Value

RNG Random Number Generator

SNR Signal-to-Noise Ratio

1 Motivation and Goals

1.1 Motivation

For many years now, there has been an unmistakable trend in the energy sector towards Renewable Energy Sources (RES) [EDE14]. This has been encouraged both by political measures as well as general public awareness of environmental issues tied to the usage of fossil or nuclear energy sources. Therefore, energy providers and private households have contributed to the rising number of distributed generation units and Energy Storage Systems (ESS). The coordination and efficient use of these new technologies forces the distribution grid to continuously evolve and become a *smart grid* [FAR10; INT15]. An interesting part of this new grid is the coordination of flexibility provided by adjustable demand and generation units and ESSs. An increasing amount of these systems, however, are deployed "behind the meter", meaning they are currently not directly observable or controllable by the grid operator. Such grid entities, which are more involved in grid operation than simple consumption can be called *prosumers*. Currently, prosumers' ESSs and flexible generation and demand units are mostly used to minimize their own energy expenditures by lowering the demand from the grid. Such uncoordinated behavior, however, can lead to unexpected consequences, like grid congestion due to rapidly changing demand curves. With proper coordination, on the other hand, the flexibility might additionally be used to benefit the overall grid state. Especially hierarchical distributed control schemes are a point of interest to coordinate the decentralized generation and storage infrastructure. Therin, prosumers and traditional entities are grouped into *microgrids*, which use their local flexibility to become more independent from the overall grid.

To realize these goals, new communication and coordination schemes must be developed and tested. That work is usually not done in-field because of the cost of setting up a secured large-scale testbed [STR17]. Therefore, we turn to simulations to analyze problems and test new solutions. However, the simulation of a complex system like this involves models from different domains, which a suitable tool should be able to combine. Naturally, this includes the behavior of the electrical grid and all of its components. Additionally, it should consider communication and control strategies, as well as auxiliary processes, such as environmental effects and human behavior. It becomes clear that no single simulation tool provides this kind of environment. On the other hand, the areas of research involved already provide individual tools and expert knowledge. Therefore,

it would be useful for research and industry alike to be able to combine these domain specific tools in one framework. This thesis addresses this by using the co-simulation approach to provide a customizable multi-domain simulation environment. In this environment, individual tools should work in conjunction with each other, whether they were especially designed for it or not. This will allow different groups of researchers to work individually on improving and expanding the simulation environment, each in their area of expertise. In turn, they will be able to use high-quality simulation models from other domains in their own research without detailed knowledge of that domain. In the end, this will yield a more complete and realistic analysis of everyone's models and ideas.

1.2 Goals

The goal of this thesis is the development and implementation of an environment for the co-simulation of smart grid entities. Furthermore, the concept of electrical grid flexibility will be analyzed and an exemplary flexibility coordination scheme will be integrated into the previously developed simulation environment.

Functionally, the environment needs to facilitate time-synchronicity between individual simulations and coherent data flow, a task called scheduling. Next, it should retain a persistent, shared data center which forms the ground truth for and collects relevant data from the individual simulators. Lastly, a shared monitoring unit should provide an overview over all simulation processes and the simulation's meta information.

A special focus is laid on horizontal and vertical scalability, as well as usability of the environment. Horizontal scalability means that the environment should be extensible to accommodate and interact with a variety of different simulation tools. These might include grid entity simulation for loads, DERs, and ESSs, alongside with control and communication system simulators. Vertical scalability signifies the ability to create multiple instances of these simulators in order to extend the scenario to more realistic sizes in the context of grid wide coordination tasks. Usability includes clean and simple interfaces and deployment automation, which enables even inexperienced users to integrate their subsystems into more complex simulation scenarios and execute them.

The exemplary simulation task for this environment is a hierarchical approach to low-level flexibility coordination in private prosumer households. This includes local EMSs which control prosumer behavior locally and form a layer of abstraction and aggregation on top of the actual controllable assets. These abstracted systems can then be provided with external control signals from an superordinate, multiple-household controller. The main goal of these controllers is the reduction of injection peaks caused by renewable energy sources (RES), such as PV systems.

2 Theoretical Background

In this chapter, the theoretical background of this thesis will be discussed. Section 2.1 discusses the practical use case of this thesis: the concept and usage of flexibility in the context of energy systems. While subsection 2.1.1 explores different definitions of the concept itself, subsection 2.1.2 concentrates on the implementation and usage of flexibility in a private end-user scenario. Additionally, subsection 2.1.3 explains the MPC scheme and subsection 2.1.4 the involved optimization approach, which will be used for flexibility coordination.

In section 2.2, the processes of modeling and simulation is introduced. It includes basic simulation theory, highlighting different kinds of simulation models and solving procedures (subsection 2.2.1). Furthermore, the co-simulation approach is motivated as a strategy for multi-domain system design simulation and design (subsection 2.2.2).

Lastly, the software engineering basics for software virtualization are presented in section 2.3. Here, the idea of virtualization itself is explained and different levels of virtualization are described.

2.1 Flexibility

In the context of smart grid technology, flexibility is a central concept. The power grid is shifting from its traditionally passive and hierarchical structure to a new, more distributed structure. In accordance with public opinion and political incentives, energy providers are starting to add more and more RESs into their energy mix. Estimates expect that even in the mid-term, RESs will make up 50 % of the produced energy in Germany. Conventional energy sources are increasingly forced to adapt to the *residual demand*, the energy demand after the consideration of RES injection. The volatile nature of this injection leads to rapid changes in residual demand. Given a high enough penetration of RESs, the conventional energy sources might not be able to follow that demand anymore. [BUN17]

And not only large-scale projects like wind parks are getting more abundant, but also private energy generation solutions like PV systems. These systems introduce additional challenges for grid operators because they distort the standard demand behavior. Normally, private consumers are served using standard load profiles. These profiles get less and less accurate and the consumers energy demand gets more and more volatile, the

more RESs are deployed behind the meter. Furthermore, if a region can regularly produce more energy than it is consuming, that energy needs to be transported to other regions, increasing the potential for grid congestions. [BUN17]

All of this leads to the conclusion that the passive grid infrastructure must change towards a smarter grid, which allows the intelligent and automated control of distributed and volatile energy sources. A central feature of this new grid infrastructure is its *observability*. Instead of relying only on statistical load profiles, grid operators should have access to real-time data provided by smart power meters from all grid entities, including consumers, generators, and distribution hardware. This enables a more realistic overview over the current grid state. Together with environmental factors, even behavioral projections into the future can be generated. This information can be used to take necessary control decisions in order to avoid congestion and instability in the grid before they occur. [MOM09]

This goal can only be reached if together with observability, the system's *controllability* is increased. Storage systems and smart appliances are one way to introduce more controllable assets into the grid. However, the amount of active grid participants is currently limited. Private consumers have virtually no possibility to interact with any global or local energy market. The lack of incentives leads these consumers to optimize their own demand instead of using potential flexibility in their behavior to benefit grid balancing and congestion reduction. After all, transmission grid resources are limited. Being able to control these prosumers down to the private household level using markets or explicit control signals can help avoid imbalances where they occur and alleviate grid congestion problems. [MOM09]

2.1.1 Definition

In literature, there are several different definitions of flexibility in the domain of energy systems. They often depend on the use case or scope of the article in question. In [NEU15] the given definition is:

> The possibility to influence the operation mode of energy producers or consumers by shifting production or consumption under given constraints is called flexibility[.]

This captures the main aspect of being able to change production and consumption patterns to achieve some goal. However, it is not very precise. It is not stated whether the shift is meant to be temporally, in amplitude or both. Moreover, it is not clear what is included in the "given constraints".

A more broad definition can be found in [DAL17]:

> Flexibility can generally be seen as a system's ability to provide secure and economical supply-demand balance across spatial and temporal scales by leveraging and seamlessly coordinating various controllable assets.

This definition mentions spatial and temporal aspects of flexibility and references its source as the "controllable assets" of the system. Nevertheless, it already implies a use case to be supply-demand balancing, which might not necessarily be true. In that light, any potential for change in a system that is not available for balancing purposes would not be considered flexibility. Additionally, any system that purposefully needs to be kept in an imbalance would also not be taken into account.

The most technical definition is probably proposed in [MAU17]:

> The flexibility of an energy system is the collection of valid combinations of system inputs and their state dependent outputs in terms of all energy carriers, i.e., all combinations that provide all mandatory energy services in manner of ensuring system stability.

This definition neither limits the existence of flexibility to any specific use case, nor to any energy type. The only constraints are those within the capability and stability of the given system itself. Nevertheless, this definition is so very focused on general applicability that it loses most explicitness in the context of smart grid development.

The probably most useful definition, and basis for this thesis, is therefore taken from [CEN14]:

> The flexibility in demand and supply in the context of Smart Grids [...] covers the changes in consumption/injection of electrical power from/to the power system from their current/normal patterns in response to certain signals, either voluntarily or mandatory.

This definition gives one of the best idea of what is means for a power system to be flexible. One might consider generalizing it away from only "electrical power", since smart grids might incorporate other energy carriers. Moreover, flexibility itself is technically only the possibility of change, while the actual change as stated in the definition would be the use of said flexibility.

In the end, flexibility is an abstract concept whose definition is either very broad or context dependent. Overall, the definitions mentioned above give an outline of what it entails, as seen from different perspectives.

2.1.2 Energy Management Systems

Flexibility in energy systems can be further categorized into two different types. They are called *physical* and *structural flexibility*. Physical flexibility denotes the actual capability of a system to provide flexible demand/generation behavior. Structural flexibility is the ability of the system to use the physical flexibility that is present. This includes coordination infrastructure and operation or market mechanisms. [AKR19] In a simple household scenario, installing an ESS or having controllable appliances are sources of physical flexibility. They enable the household system to change its consumption or injection patterns away from the normal, static behavior. The structural component that enables the use of this flexibility is usually called a Home Energy Management System (HEMS). This system, in its most basic form, tries to lower the household's energy cost by reducing its demand from the grid. However, physical flexibility within private homes is also potentially interesting to the grid operators and energy providers. Given the necessary structural flexibility, it can be used to optimize, and therefore stabilize the overall grid behavior. Different approaches exist on how a more overarching Energy Management System (EMS) and HEMSs might interact in order to satisfy customer and grid operator goals alike. In [MAU17], approaches are summed up into four different categories.

Physical Demand Response Here, the HEMS gives direct load and generation control of the household's internal entities to the overlying controller. It propagates device states and capabilities and tunnels back operation commands and set points.

Direct Market Demand Response In this scheme, local devices are abstracted into models and then aggregated into a combined flexibility model for the household. The global EMS gets the explicit flexibility information and has to decide on suitable flexibility settings for all participants.

Indirect Market Demand Response For this approach, the household flexibility is not explicitly determined or modeled. Rather, the overall EMS tries to exploit implicit flexibility by giving incentives to the households (usually changing tariffs). This allows for iterative algorithms between HEMS' and the global controller, in order to find the optimal flexibility usage.

Decentralized Market Demand Response This category is the only one not relying on a central entity for coordination. Decentralized schemes use peer-to-peer household networks to perform the optimization. Distributed heuristics determine the overall coordination process. This is comparable to other distributed communication protocols as known from decentralized sensor networks or distributed robotics. Since most approaches in literature employ at least one central entity for communication or coordination, truly decentralized schemes are rare.

These categories are, of course, not necessarily mutually exclusive. One might imagine a more complex control scheme, where direct market demand response for explicitly known flexibility is augmented by indirect market demand response to harness flexibility which cannot be explicitly determined. Alternatively, a part of the grid (e.g. a neighborhood or a cell) might employ a decentralized scheme while still being tied into a larger grid structure. The overall controller could use indirect market response to incentivise the optimal use of the distributed cells' flexibility within the larger context.

Nevertheless, all schemes have their advantages and disadvantages. Physical demand response is very deterministic and humanly understandable. However, the underlying systems are not abstracted at all and almost fully controlled by the global coordinator. This is not a realistic solution for large scale systems and raises other concerns, such as privacy protection. Indirect market demand response, on the other hand, promises to leverage all available flexibility with comparably little computational effort. However, deterministic planning and human interaction with the control system become less predictable. This thesis, however, will use *direct market demand response* as an exemplary use case for flexibility coordination. This approach provides a compromise between system abstraction and determinism, which helps to showcase different system interactions. Individual smart grid components are abstracted by fitting models and operation decisions of both HEMS and global EMS are determined by a so called Model Predictive Control (MPC) scheme.

2.1.3 Model Predictive Control (MPC)

The concept of MPC has been used in industrial process control for many years. In recent years it has also found its way into the power engineering sector as an efficient way to coordinate flexibility resources for load balancing. [ARN11] MPC is a collection of feedback control schemes that all use a similar procedure to determine their control signal. The basic principles involve the explicit use of a model in order to predict the future behavior of the system to be controlled. Furthermore, the control signal is determined by minimizing or maximizing a given objective function. Lastly, all MPC methods involve a receding horizon strategy. This means that in each execution of the control loop, the objective function is optimized up to a certain time horizon. On consecutive execution, the horizon is displaced into the future accordingly. Therefore, MPC algorithms are also called Receding Horizon Predictive Control (RHPC). [CAM04]

The different MPC algorithms can be distinguished by the implementation of the basic principles. The system model is arbitrary; one of the classical examples from control theory is an impulse response model. But also empirical models without an analytical representation and nowadays even neural networks are valid. The only requirement is

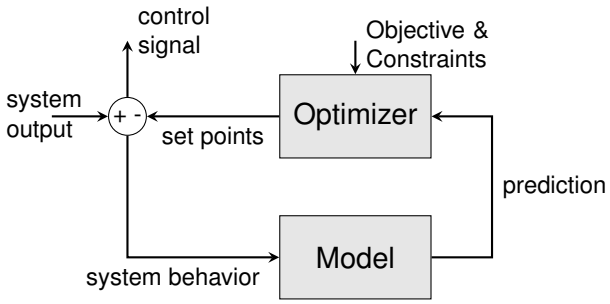


Figure 2.1: Basic MPC structure

that they produce a prediction of the controlled systems behavior. The more accurate the model, the more precise the control will usually be.

The objective function is usually some kind of optimization problem (e.g. making the system's output follow some reference signal). The system's controllable values are the variables over which the problem will be optimized, while system constants, inputs, and predictions are seen as fixed parameters. This generates a reference trajectory for the control horizon, thus giving the controller a priori knowledge about the system's evolution. Therefore, the control system can eliminate the effects of processing delays by starting to act before the actual change has occurred.

Additionally, any practical system is subject to constraints. Those might be physical constraints, like an actuator's field of action or a sensor's limited scope. Alternatively, they might be constructive, safety-related, environmental or operational constraints, such as temperature, voltage, power or energy constraints. Many objective functions can be adequately realized by (Mixed Integer) Linear Programming (MILP). The following of a target trajectory is usually implemented as a Least Mean Squares problem which can be solved by Quadratic Programming. However, other non-linear or stochastic optimization procedures are also feasible.

Lastly, the control law has to be obtained from the objective function. This is done by solving the given optimization problem. Rarely, this can be done analytically (e.g. for an unconstrained linear problem), but normally an iterative solver is applied. The choice of the solver also depends on the type of optimization problem. If all system constraints have been implemented correctly, the optimized solution's variables can be used directly as set points for the system's controllable entities.

2.1.4 Mixed-Integer Linear Programming (MILP)

In the context of this thesis, the MPC scheme applied in flexibility coordination uses a Linear Programming (LP) based approach for optimization. Linear Programming is a kind

of optimization problem, where the goal is to minimize or maximize a linear *objective function* of the form:

$$\zeta = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = \mathbf{c}^T \mathbf{x}. \tag{2.1}$$

Where **x** denotes a vector of *decision variables*, which are unknown and need to be chosen in an optimal manner. The coefficient vector **c** can be seen as a kind of cost parameter. The higher the absolute value of an individual coefficient is, the more influence it has on the value of the objective function. The goals of minimization or maximization can be converted into each other by negating the objective function $(\max \zeta = \min -\zeta)$. Finding the solution of such a problem can be done analytically, since it only means finding an extreme point in a linear function.

In most real LP problems, there are additional constraints. The simplest example would be that a variable cannot be negative. Generally, constraints are represented by:

$$\mathbf{a}^T \mathbf{x} = a_1 x_2 + a_2 x_2 + \cdots + a_n x_n \begin{Bmatrix} \leq \\ = \\ \geq \end{Bmatrix} b. \tag{2.2}$$

The kind of equation or inequality does not matter here because conversion between all of them is possible. There can be any number of constraints m, such that

$$\mathbf{A} = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}. \tag{2.3}$$

Meaning the final problem can always be expressed in the standard form:

$$\max \quad \mathbf{c}^T \mathbf{x} \tag{2.4}$$

$$s.t. \quad \mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{2.5}$$

This kind of problem is commonly found in real applications, such as profit maximization in the presence of market and resource constraints in economics.

Since LP is a subclass of convex optimization, it can be efficiently solved. A commonly described way is called the simplex method. This is an iterative algorithm, which starts with an arbitrary point inside the solution space. It then improves the solution in each step as much as possible and then substitutes one of the decision variables using one of the constraints. If no more improvement is possible, the optimal point has been found. Over the years, even more efficient algorithms have been developed. [VAN14]

If at least one of the decision variables has to be an integer, the problem is called Mixed-Integer Linear Programming (MILP). This kind of constraint can occur easily in real examples such as optimal worker deployment or flight scheduling. It is naturally impossible to deploy half a worker or fly 2.6 planes somewhere.

Even though the change from LP to MILP seems small, the latter is a superordinate category of optimization problems, which are not necessarily convex anymore. Finding a general solution for an MILP problem has been proven to be NP-hard. However, there are special cases, which are efficiently solvable and in many cases a solution can be found within a reasonable time frame.

The most commonly known algorithm to solve these kinds of problems is called *branch-and-bound*. This algorithm starts by relaxing the MILP problem into a normal LP problem. This problem is then solved as described above. The solution can generally not be expected to be feasible in terms of the integer constraints, but it provides an upper bound for the optimal solution. Afterwards, the problem is split into two sub-problems with an additional constraint. These constraints split the solution space along one of the relaxed integer variable's optimal value x_i^{opt} and force that variable to fulfil either (2.6) or (2.7) respectively.

$$x_i \leq \lfloor x_i^{opt} \rfloor \tag{2.6}$$

$$x_i \geq \lceil x_i^{opt} \rceil \tag{2.7}$$

This procedure is applied recursively to both sub-problems (*branches*) until either no integer solution is feasible anymore, the found solution fits the integer constraints or the objective function performance is below another known integer solution (*bounds*). When all branches have terminated, the best performing integer solution is the optimal solution to the MILP problem. [VAN14]

As described, this strategy involves the repeated solution of LP problems. It is therefore in general significantly slower than solving a single LP problem. Furthermore, the underlying LP-solver performance becomes more significant because even a slight speedup there will accumulate over the repeated executions.

2.2 Simulation

The topic of simulation is important in almost any scientific area of study. It enables researchers to gain insight into processes which are very hard or even impossible to observe in reality. Physicists for example oftentimes analyze everything, from the evolution of the universe to quantum interactions through various means of simulation before conducting physical experiments. In engineering, simulation enables the analysis of system

designs under a variety of circumstances in a controlled environment. In this section, the theory behind simulation systems in general and co-simulation in particular will be discussed.

2.2.1 Basic Modeling and Simulation

Firstly, the basic structure and workflow of a typical simulation system will be discussed. This will help to gain an understanding of how co-simulation differs from traditional simulation. Furthermore, it lays the groundwork for the design decisions taken during the development of the co-simulation environment.

Models and Simulators

The term *model* is used frequently throughout the thesis. An equivalent expression is *dynamical system*, which is a theoretical model of a *physical system*, hence the term *model*. At any point a model is described by its state, a set of values the dynamical system can assume. The set of possible states comprises the state space of the system. The so called *evolution rules* describe how the state changes over an independent variable (e.g. time). An example in the context of this thesis is a real ESS. For the sake of clarity in this section, energy transfer losses as well as border cases will be ignored. Therefore, the only state variable is the ESS's State of Charge (SoC) and its evolution rule is the simple integral (2.8).

$$soc(t) = SoC_0 + \int_0^t p_{in}(T)dT \tag{2.8}$$

The SoC changes continuously as an integral of the input power over time. Being continuous, this state can take infinitely many values, although it might be bounded. On the other hand, it is also possible for a state to only have a finite number of values (e.g. [charge, discharge, idle]). The set of curves described by the model's state is called the *behavior trace* or simply *behavior*. The ESS's behavior, for example, would follow the linear curve (2.9) in case of constant input power P_{in}.

$$soc(t) = SoC_0 + P_{in}t \tag{2.9}$$

Lastly, the *validity* of a model expresses the difference between the model's and the real system's behavior. The individual requirements on validity depend on the use-case. Usually, it is sufficient for a model to be valid within a limited part of the state space that highlights the features of interest. [GOM17]

A *simulator* is an algorithm that computes the behavior of a model. For that purpose it uses a solver, e.g. numeric or sequential solvers, depending on the model in question. Since these solvers are executed on digital computers, the generated behavior is usually an approximation, especially for continuous behavior of the model. The *accuracy* is a measure of the closeness of generated behavior to theoretical analytic behavior. It is important to notice that it is possible to have an accurate behavior for an invalid model and vice versa. Therefore, the choice of the solver and proper modeling are equally important for generating useful results. Lastly, most simulators depend on external inputs. A system consisting of a model and solver, emulating a real system, is therefore called a *simulation unit*. A *simulation* is the behavior obtained from a simulation unit by supplying its inputs. [GOM17]

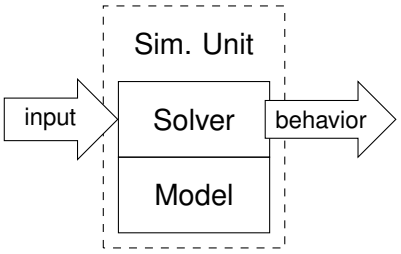


Figure 2.2: A simulation consisting of the simulation unit that generates a behavior trace from an input trajectory.

Time

Another important topic for modeling and simulation is time. In physical systems, time naturally progresses. An experiment or procedure will take some amount of *physical time* to complete. Inside the simulator, this time is abstracted by an ordered set of values where each value represents an instant in time. This is called the *simulated time*. Thirdly, the time passing for the simulation user while it is processing is called *wall-clock time*. In the ESS example, researchers might be interested in its performance during one day of physical time. The simulated time t might be a float value counting the seconds passed. The model might require 3 minutes to complete simulation which is this simulation's wall-clock time. [GOM17; FUJ99]

Many simulations aim for as-fast-as-possible simulation. Hence, they try to use as little wall-clock time to execute the desired simulation time span. In that case, the progression of simulation time is not directly tied to wall-clock time. The alternative is called scaled real-time simulation. Here, the simulated time progresses linearly to the wall-clock time. A special case is real-time simulation, where simulated and wall-clock time move similarly.

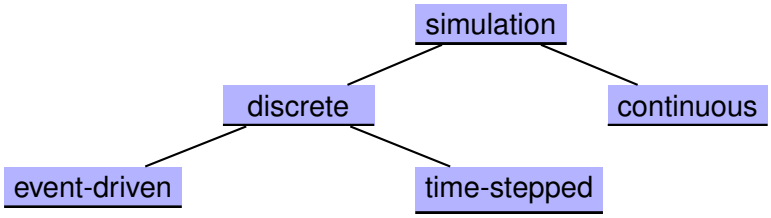


Figure 2.3: Classification of time flow mechanisms for computer based simulation execution

This is interesting for several applications, such as Hardware-in-the-Loop simulations and virtual environments (e.g. video games). [GOM17; FUJ99]

Time Flow Mechanisms

Having clarified the notion of time within the simulation, the models can be divided into categories based on their temporal behavior, as shown in Figure 2.3. Broadly speaking, they can be either continuous or discrete. A *continuous model* produces behavior that is continuously changing over time. This is often described by differential equations. The ESS model is actually a continuous model. It could also be described by the differential equation (2.10).

$$\frac{dsoc}{dt} = p_{in}(t) \tag{2.10}$$

Examples for such models can be found in many areas, such as thermodynamics, motion equations, energy transfer, etc. In *discrete models*, the state changes only at specific points in time, otherwise it is considered constant. Therefore, the computation of a new state is only necessary at those discrete points. In practice, continuous models are oftentimes handled similarly. Even though their behaviour is considered continuous, their state is only recomputed at certain time steps, as decided by the simulation program. In our example, if the SoC $soc(t)$ of the ESS is only needed at certain time steps, it is sufficient to use the integral equation to solve those specific points in time. [FUJ99]

In case of continuous simulation, the notion of time flow is usually unnecessary since models are solved analytically. Hence, they yield an analytical function describing their behaviour during the whole simulated time. In discrete simulations on the other hand, simulation time has to be explicitly advanced. There are generally two mechanisms that are used in discrete execution, called *time-stepped* and *event-driven* simulation. In time-stepped simulation, time is advanced by equal-sized time steps. All models' states are resolved for each step. The most basic time-stepped simulation mechanisms also assume that simultaneous actions are independent and can therefore be computed in parallel. Causal relationships have to be realized by executing actions in concurrent time steps. Therefore, step size has a big impact on the simulation's accuracy. Time-stepped

execution is one straightforward way to approximate the behaviour of continuous models, especially if they cannot be solved analytically. In the ESS example, this might be the case, if the input power has no feasible analytic representation. The discrete approximation would assume the input power to be stepwise analytical (e.g. constant) and solve the system for each time step consecutively. [FUJ99]

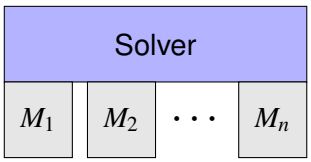
In contrast, event-driven simulation has no fixed time step width. Recalculating all state variables in a fixed manner might be inefficient in some cases, yet not be precise enough in others. Therefore, event-driven simulation will only execute a simulation step whenever an *event* occurs. This event is an abstraction of an instantaneous action in a real system. Each event has a time stamp, indicating when it will occur. The simulation in event-driven time flow then advances from one event to the next, recalculating states according to the event's time stamps. The ESS simulation example could be realized by creating an event whenever the input power changes or the change is larger than some threshold. Then the SoC will be calculated over a long period of time during which the input stays constant. When it changes quickly however, the state will be recalculated more frequently, depending on the rate of change. This will give a high temporal resolution when the system behaves dynamically and saves computation time when it behaves more statically. However, this needs a more elaborate simulation execution. In time-stepped execution, a simple loop that iterates simulation time will suffice. In event-driven execution, a *scheduler* is needed. This is an algorithm, which handles the list of upcoming events. It keeps them chronologically sorted, continuously triggers the execution of the next event and advances the simulation time accordingly. This called the *event-processing loop*. During the execution, the simulation unit changes its state and can schedule new events. Those events are only allowed to have time stamps greater or equal to the current one, otherwise the simulation would break basic causality. [FUJ99]

Additionally, it is worthwhile to note that the two aforementioned time flow mechanisms can be emulated within each other. Time-stepped execution can be realized in event-driven simulation by triggering events in regular time intervals. Implementing event-driven behavior in a time-stepped simulation is less common. It can be done by defining the time step size to be the greatest common divisor of all event time stamps. This can lead to rather inefficient execution, though, since most time steps may not need any computations. Lastly, both mechanisms can be used to perform real-time simulations by forcing the simulation to idle until the wall-clock time reaches the next time stamp to be simulated. This naturally assumes the simulation to advance faster than the corresponding physical system. [FUJ99]

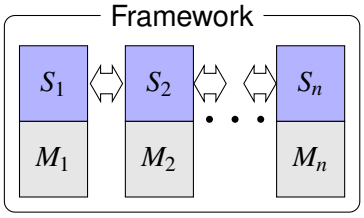
2.2.2 Co-Simulation

A simulation as described in subsection 2.2.1 is called a monolithic simulation because a unit is only comprised of a single model/solver stack and can only be executed as a single entity. It can be used to obtain the behavior of a specific system. However, as motivated in section 1.1, it is useful to obtain a correct simulation for complex multi-domain systems as well. A first approach to do this is *co-modeling*, a special kind of simulation. Here, the different parts of the system are modeled individually. The simulation unit then consists of multiple models coupled by a common solver (see Figure 2.4a). This can help in modeling more complex systems. However, it limits the simulation unit to one solver and one design strategy, which might not provide sufficient accuracy and validity for all models. [GOM17]

In *co-simulation*, another kind of simulation, the sub-systems are not only represented by individual models, but by individual simulation units. These units are coupled via their inputs and outputs to produce a *co-simulation unit* (see Figure 2.4b). Additionally, a co-simulation unit also contains an *orchestrator* or *co-simulation framework* which is needed to coordinate execution and data exchange of the simulation units. Hence, the behavior of the co-simulation unit is called a *co-simulation*. The information needed to generate a correct co-simulation unit, such as the data flow between simulators and external inputs, is called the *co-simulation scenario*.



(a) A Simulation unit with co-modeling. Different problems are modeled individually and then solved by a common solver. Coupling of models is realized through the solver.



(b) A co-simulation unit consisting of individual simulation units and a framework. Data exchange and coordinated execution are realized by the framework.

Figure 2.4: Co-Modeling and Co-Simulation Comparison

This approach has multiple advantages. Firstly, it improves the development of a coupled system by different teams and experts. To each, the other sub-systems are black boxes, which adhere to a clean interface. This decouples individual simulation requirements. Each sub-system simulation unit can be realized by the languages and tools of the expert's choice. The black box nature of the units also simplifies the integration of closed-source tools and protected IP. Furthermore, the analysis of system wide problems and design choices can be realized cheaply and early into the project. Returning to

the ESS example, one researcher might work on the battery model, while another works on designing a control unit. If both work within the same co-simulation environment the controller can continuously be verified against an increasingly realistic battery simulation. Otherwise, building an ESS test bed might be time consuming and expensive. Moreover, in case of a design error it might even break. Additionally, since the simulation units represent physical systems, the whole simulation setup becomes more humanly understandable. It also simplifies integration later on because most system interactions are already known and tested. Lastly, with proper orchestration, the simulation units can also be executed in a distributed manner.

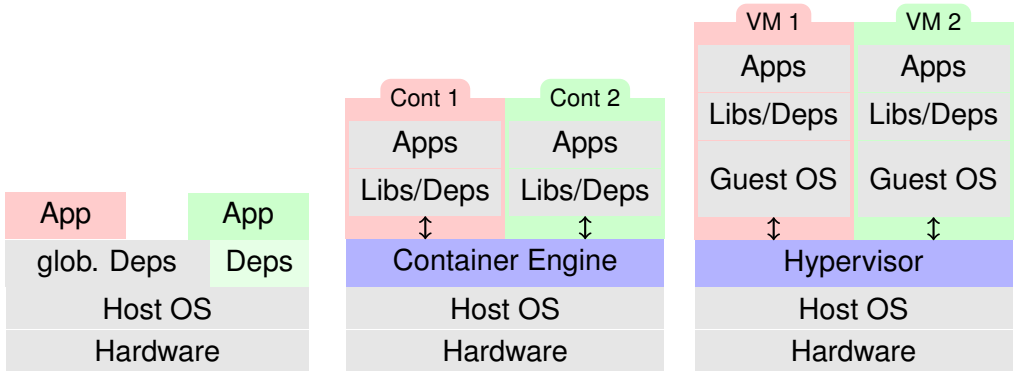
On the other hand, co-simulation can introduce some uncertainties. It often cannot be said whether and how the properties of individual simulation units translate into a composite simulation. Furthermore, the distinction of highly coupled sub-system into weakly coupled simulation units can affect solver accuracy. For instance a problem that requires iterative solving can span across multiple simulation units. The framework has to allow this interaction which might break the clean interface of the simulation. Lastly, several examples in literature [GOD10; BIA15; TRO16] use co-simulation only for limited scenarios such as the combination of two specific simulators, e.g. communication and power systems. This shows its usability in multi-domain simulation, yet also how much potential there still is with a more flexible co-simulation environment.

2.3 Virtualization

As already discussed in subsection 2.2.2, monolithic design for simulators is not always desirable. This argument can be made for the software layer as well, especially with the goal of a co-simulation environment in mind. Preparing such a tool as a monolithic unit of software comes with many problems and difficulties. Firstly, maintaining such a software project becomes very difficult. All available simulators need to be integrated with the actual environment software which leads to system updates becoming intertwined with external tool upgrades. Functional updates to any single component can potentially influence the whole system. Moreover, the different tools might cause dependency conflicts if they depend on different, incompatible versions of the same library. A solution to such problems in modern software development is called *microservice architecture*. This approach divides a complex application into smaller, independent tasks, the microservices. Each microservice should have a single, well-defined responsibility. This helps to keep the code base cohesive, meaning all related code stays together. A cohesive codebase is useful because it helps to reduce redundancy and makes code optimization more straight-forward. To achieve a more complex goal, the services then interact through a well-defined interface, oftentimes Representational State Transfer based APIs

(e.g. RESTful-API) or comparable protocols. It must be noted that the network communication used by microservices is always slower than the in-memory data sharing in a monolithic application. However, the added flexibility and separation of concerns is often worth this trade-off, especially if the interface speed is not the limiting factor for execution speed. [DRA16; THÖ15]

To realize such an architecture, each microservice has to be developed with its own runtime environment. Otherwise, the afore mentioned dependency issues would still exist between the services. Additionally, microservices should be deployable to a variety of hosts, including distributed systems. Such an abstraction of different software units from the host system is called *virtualization*. The most shallow form is called application virtualization. This structure involves the (partial) sandboxing of an application's file system access. In a virtualised application, its file system access is (partially) rerouted into a directory, created especially for the application. This is comparable to the virtual environments used by most Python applications. This mechanism redirects installation and import calls of third-party Python packages into the virtual environments subdirectories. This enables users to develop and execute different programs with potentially conflicting dependencies in parallel by using different virtual environments. While such a strategy can alleviate the dependency problems, it does not solve all problems. Firstly, the application is still dependent to the host's operating system and so is the virtual environment. Additionally, there is no supervision over the usage of hardware resources. [WIN07]



(a) On the left: A traditional software stack, where the application is on the global libraries and uses the local OS. On the right: a virtualised application, for which certain dependency calls are rerouted to its own environment.

(b) Schematic visualization of OS-level virtualization. Containers package their application and dependencies and are executed isolated from the host system, but can use the underlying kernel.

(c) Schematic visualization of virtualization using virtual machines. VMs package applications with dependencies and OS, completely isolating it from the underlying operating system.

Figure 2.5: Different forms of virtualization

The other extreme of virtualization are virtual machines, as provided by solutions like VirtualBox or VMware. They facilitate nearly complete hardware and software abstraction

from the underlying host system. Here, a hypervisor software runs as a program on the current host operating system. This hypervisor creates virtual hardware representations as if it were an actual computer, upon which any operating system can be installed and applications can be executed. A virtual machine therefore provides the most flexibility and security. The application can be deployed on any host system, as long as this system supports the hypervisor program. Scaling of the service can be realized by creating new VM instances and all allowed interaction between such systems are equal to the interactions of physical computer systems. However, it also comes with much overhead because every software unit is packaged with its own OS and all system calls are performed through the hypervisor. [SMI05]

A midway between these two solutions is called OS-level virtualization. This approach describes the introduction of multiple isolated user space instances on top of a single OS kernel. Additional user space instances for specific applications have different names, such as containers, zones, virtual private servers, virtual kernels, jails, etc. In this thesis, the term *container* will be used to refer to an instance of an OS-level virtualised software unit. Containers are isolated from each other and the host system by different mechanisms. Usually, they exist in their own kernel namespace, meaning their processes are not directly visible to the other systems and vice versa. Additionally, most OS-level virtualization mechanisms provide ways of resource control over the containers through the kernel. More precisely, containers can be restricted in their hardware access, e.g. a container might not be able to use more than two processor cores, or 1 GB of memory etc. Furthermore, the container's file system has its own root directory, making it impossible to access other file systems without explicit connections (e.g. mounts). A *Container Engine* that runs on the host system is used to keep track of all containers and provides ways for the user to interact with them. Containers are not as versatile as VMs because they are still tied to the kernel functionality. That makes it impossible to run applications designed for another kernel. Programs reliant on Windows specific kernel features for example cannot be containerized for Unix-based systems. However, the deployment of a service to all systems based on the same kernel (e.g. different flavors of unix-based systems) is usually no problem because the container already packages all dependencies. Furthermore, containers considerably lessen the overhead compared to VMs, both in terms of memory space and hardware interaction. Most notably, processes running inside a container directly use the same kernel functions as normal processes on the host system do, without the major drawback of going through translation in a hypervisor. [YU07]

3 Modeling

This chapter describes the concrete design of the exemplary flexibility simulation on the basis of the previously discussed concepts and ideas. Firstly, the simulation environment is presented and its intended workflow is outlined. Afterwards, a description is given of how the environment is used to implement and execute the actual simulation.

3.1 Simulation Environment

As discussed in subsection 2.2.2, the simulation environment should enable different simulation tools to work together. In literature, several solutions for application independent co-simulation have been presented. Most notably are High Level Architecture (HLA) and Mosaik. HLA is a standard that defines interfaces between simulators and the overall workflow in a co-simulation setup. Mosaik is a co-simulation framework developed by OFFIS, the institute for information technology at Carl von Ossietzky University in Oldenburg. A comparison study can be found in [STE18]. Mosaik was chosen to be the core part of the new co-simulation environment. From the comparison study and further evaluation it became clear, that this tool provides the necessary customizability, while still being easy to use. The latter aspect cannot be ignored, since the environment aims to provide an easy solution for collaborative simulation development. This cannot be achieved, though, if the integration of new simulation tools becomes to complicated.

3.1.1 Mosaik

Development of Mosaik started in 2011 and is still ongoing [OFF11]. It was designed especially with CPES/smart grid simulation in mind. Its goal is to provide an easy-to-use framework for the development of large-scale system simulation, which coincides with the goals of this thesis. The development language is Python and the system is based on the ‘simpy’ simulation module.

The basic architecture of Mosaik can be seen in Figure 3.1. The core application consists of a simulation manager module (*sim-manager*) and a *scheduler*. The sim-manager handles the data flow between simulators. It connects to each simulator via a TCP network

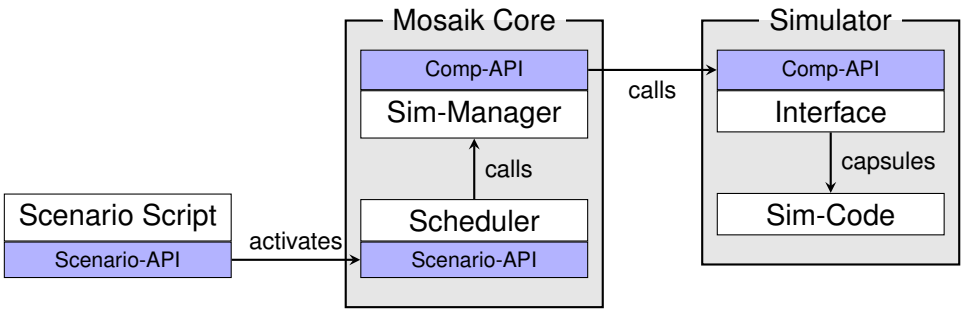


Figure 3.1: Basic Mosaik Architecture. The core application uses the sim-manager to control individual simulator and the scheduler to coordinate the execution according to the scenario definition. Component- and Scenario-API provide the necessary interfaces.

connection and sends or receives the necessary data. The scheduler coordinates this data exchange and the simulator execution. It also maintains the global simulation clock. The time flow scheme for the advancement of this global simulation time can be described as a hybrid between event-driven and time-stepped execution. This is achieved by the underlying simpy-module, which is an event-driven simulation tool. Each simulator can tell the Mosaik core at which time step it should be executed next. The sim-manager then schedules an execution event for that time step. Since each simulator can only request events that trigger their own execution, this yields a time-stepped execution with the possibility of variable step width. This scheduling process is illustrated in Figure 3.2. Proper event-driven execution, on the other hand, would entail the ability of any simulator to generate events, which could trigger the execution of other simulators. Whenever multiple simulators in the given scheme are scheduled for execution on the same time step, the scheduler analyzes the data dependencies. Simulators with no dependencies between them can be executed in parallel, while those who need data from others are executed sequentially according to the dependencies. An exemplary execution flow can be seen in Figure 3.3b. Circular dependencies are not allowed and have to be resolved beforehand. One way to achieve this is to explicitly define a connection within a circular dependency to be time shifted, meaning the receiving simulator will get the data generated during the last execution, not the current one.

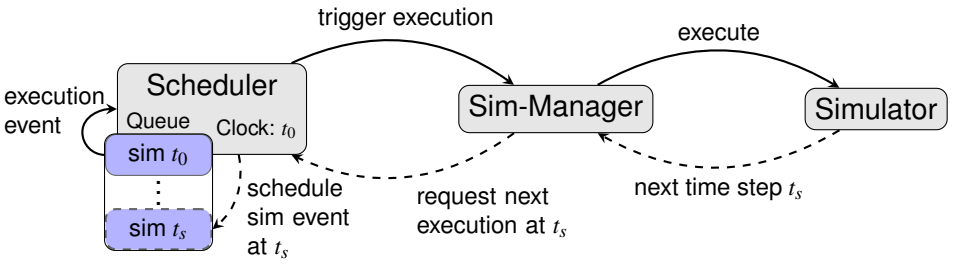
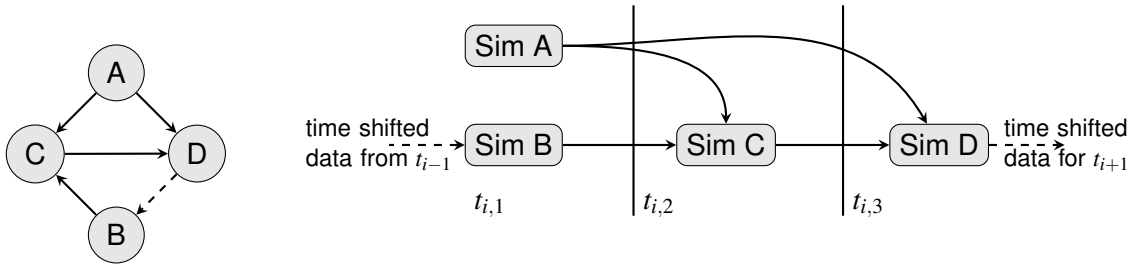


Figure 3.2: Mosaik's variable step size execution scheme. On each execution, the simulators get to choose when they need to be executed next. This is realized by an underlying event queue in the scheduler.



(a) Exemplary data dependency graph. Since simulators B, C and D would produce a circular data flow, the connection between D and B is realized time shifted.

(b) The simulation schedule for time step t_i. Simulators A and B have no inter-dependencies and can therefore be executed in parallel. Simulators C and D are executed sequentially, because of their dependencies. The time-shifted connection between D and B is realized in consecutive time steps.

Figure 3.3: Mosaik intra time step scheduling

Further shown in Figure 3.1 are two APIs, which provide the interfaces for interaction with the Mosaik core. Firstly, there is the *Component-API*. This needs to be implemented by each simulator. It defines the interactions between the sim-manager and the simulators. A high-level API is provided for Python and some other languages (e.g. Matlab, Java, etc.). However, those are high-level abstractions of the underlying low-level API. This uses the aforementioned TCP network connection to exchange commands and data in the form of JSON objects with the sim-manager. This interface can be implemented in most common programming languages, making it possible to attach simulators independent of the language used for development. Furthermore, this design makes it possible for the simulators to be run locally on the same machine or on different computers within the same network. There are two different kinds of interface calls in the component-API. The regular or synchronous calls are triggered by the sim-manager for a simulator. They include simulator initialization, model creation, simulation step execution and data retrieval. Additionally, there are asynchronous calls, which are issued by the simulators towards the sim-manager during its step execution. They include getting and setting data asynchronously, retrieving simulation progress and dependency information.

The second API is called the *Scenario-API*. This is implemented in Python and provides an interface to the scheduler. In the scenario script, this API is used to trigger the sim-manager to initiate all necessary simulators, create simulation models and define data flow connections between those models. Upon their start, each simulator returns a meta description of their models, including their name, parameters and attributes. When a model is created, this description is used to create a local proxy model, representing the actual model to the sim-manager. During data flow definition, the proxy models ensure some basic validity of the scenario, e.g. whether the connected parameters actually exist in the models. The sim-manager also uses the proxy models to execute the component-API calls during the simulation execution. Each proxy in turn holds the information on how to reach its respective model. This structure decouples the network interface from

the actual simulation execution. It also makes it easy to include auxiliary, non-simulator services into the simulation infrastructure. A data logger for example can be realized by implementing it with the component-API, as if it were an actual simulator. The model parameters of interest are then connected to this pseudo-simulator and it can simply save the provided data e.g. to a database server running in the background.

This decoupled design, both in terms of component-API language and execution location makes Mosaik a versatile tool, that can integrate most simulation tools and additional services. Furthermore, the simplicity of the interfaces make it very straightforward to add existing or new simulators in any language, but especially Python. A disadvantage is that the chosen time flow mechanism only allows for variable time steps, but never for true event-driven behavior. That is the cost Mosaik users pay for the simple interfaces and straightforward scenario definition. Lastly, the sim-manager is, by design, the central entity for data exchange. This simplifies Mosaik's implementation, but it might lead to congestion and slower execution for large-scale simulations compared to peer-to-peer data exchange. Nevertheless, Mosaik's versatility and ease of use made it the tool of choice for this co-simulation environment, despite those minor drawbacks.

3.1.2 Docker

Using Mosaik as a basis for simulator execution, the next challenge for a versatile simulation environment is to provide a runtime environment for different simulation tools. As discussed in section 2.3, a co-simulation framework lends itself to being designed as a microservice architecture. For this purpose, the individual simulators, as well as the Mosaik core, were implemented as independent, isolated microservices, using Mosaik's component API as a communication interface. Since it is implemented as a RESTful-API, this fits with the general picture of microservices as seen in most literature. As further discussed in section 2.3, a virtualization of the services is sensible, since it avoids dependency issues and limits the unwanted influence of different parts of the software on each other. Moreover, it increases the services deployment and scaling options. Therefore, OS-level virtualization was chosen for the implementation of the services because it provides the mentioned advantages without introducing too much overhead, compared to VMs. In detail, the containerization tool chosen for implementation is called *Docker* [DOC20a].

Docker is an open source Platform as a Service (PaaS) containerization suite, which was first released in 2013 and is currently developed by Docker Inc. PaaS products are services which provide a platform for users to develop, run and manage applications on. As a containerization tool, Docker does exactly this. It provides tools, which are used to bundle applications into containers, deploy them to a system and manage them over their lifetime. The heart of the service is called the *Docker Engine*. This engine employs

a client-server architecture. The server is a long-running daemon process, which creates and manages all Docker objects, such as images, containers, networks, etc. The clients use different APIs, both for direct communication with the daemon on the local machine or over a network. Additionally, a Command Line Interface (CLI) is offered for direct interaction with the Docker daemon through command line instructions.

Docker Workflow

To create a Docker container, an image is used. This image acts like a blueprint from which the daemon knows how to construct the container. Any number of containers can be created from the same image. The images can also be generated and managed by the Docker Engine. It uses a configuration file, called *Dockerfile*, to generate the image. A Dockerfile defines different instructions to be executed in order to prepare an image. Each instruction creates a new image layer. To be more resource efficient, Docker saves intermediate layers. That way an image can be rebuild more quickly if there are only changes in the upper layers, because the lower layers do not need to be recreated. Additionally, different images, which have similar base layers can also use this cache. The mechanism is called *layer caching*. It also allows for Dockerfiles to use other images as a basis and apply new layers on top, which is useful for sharing base images and adjusting them to the current use case. This general workflow is shown in Figure 3.4.

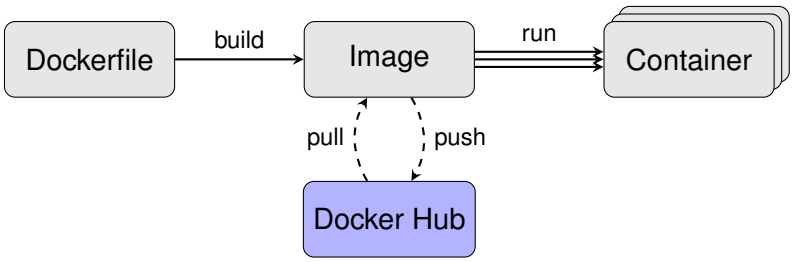


Figure 3.4: A simple overview of the Docker workflow, including the possibility of pulling/pushing images from/to the online Docker repository.

Once an image is built, Docker can run a container from it. However, Docker provides several more convenience feature for container management and interaction, like volumes and networking. Networks are probably the most important additional feature of Docker. Each container includes a network socket, through which it can be interacted with from the host. The networking service can create additional virtual networks, to which containers can attach. All containers in the same network are then able to communicate through their network interfaces. Moreover, Docker networks provide host name resolution for all attached containers, meaning the containers can address each other using their host names instead of potentially changing IP addresses. Additionally, this network is abstracted from the actual host system's network topology. It is therefore inconsequential

whether the containers are deployed on a single machine or in a distributed system. As long as they are connected to the same Docker network, they are able to communicate using the host name resolution.

Volumes are a service provided by the Docker Engine to persistently store container data. One disadvantage of containers can be that all their data is deleted when they are destroyed because their whole file system is deleted. For most containers, this is no problem because one principle of microservice design is that services should almost always be stateless and thus hold no data to be persisted. However, this is not always possible, e.g. if one containerizes a database for data storage. Naturally, this data should be kept beyond the container's lifespan. Volumes are directories on the host system, which are managed by the Docker Engine and can be mounted into the container. This can be used to save data and also to provide data to a container upon its start, without the need to copy this data into or out of the container explicitly.

Docker Community

Currently, Docker is one of the most widely used OS-level virtualization tools and its success paired with its open-source nature has spawned a big and active community. This provides many advantages for developers using Docker, including the Docker Hub. This is Docker's online image repository. Whenever the Docker daemon is instructed to run an image which is not available in its local image repository, it checks the online Docker Hub for the image. If it is found, the image is downloaded and available for deployment. This enables developers to share and reuse useful images. Many software companies also provide official Docker images for their products. This is another reason for using Docker in co-simulation in particular, because for many simulators, a well designed Docker image might already exist. Those can be used as a base, on top of which the co-simulation interface can be implemented. In the context of this thesis, Docker Hub has been used to provide the official Python image, which prepackages the Python interpreter. This image forms the base for all container images, which encapsulate individual simulators and the Mosaik Core. Moreover, the officially provided InfluxDB and Grafana images are used to create containerized database and visualization servers.

Furthermore, Docker Inc. has not only made efforts to connect end users, but also other companies in the field of container-based virtualization. Docker for example supports container deployment to container orchestration platforms such as Kubernetes. This makes it possible to easily deploy a containerized Docker application to a cloud computing service. That is essential for scaling up microservice applications to sizes/complexities that a single computer would not be able to handle efficiently. This is an important advantage, which enables also relatively inexperienced users to move large-scale multi-domain co-simulations to the cloud or other computing clusters.

Synergy with Mosaik

On the topic of disadvantages, Docker naturally comes with all drawbacks that are inherent to containerized application design. Those include the increased memory overhead by isolating each container with its libraries, instead of sharing them across all services. Furthermore, the increased communication overhead caused by the use of network communication instead of in-memory data sharing is not negligible. However, compared to virtual machines, the memory overhead of containers is still much smaller. Additionally, the chosen co-simulation framework, Mosaik, uses network communication to connect to the different simulators in any case, making this potential drawback more of an advantage. Moreover, by running the Mosaik Core inside a container itself, it becomes much easier to deploy the simulation to a variety of systems. In general, if a system provides a Docker daemon instance, the simulation should be deployable there. For large scale simulations, it might also be desirable to distribute the simulators over multiple computers or deploy it to a cloud computing platform. In a traditional setup, this would change Mosaik's simulation configuration every time the simulation network topology changes. Within the Docker network though, each simulator is always reachable by its host name, independent of its container's actual location. This further increases usability by enabling different orchestration techniques, such as load balancing.

3.1.3 Functional Co-Simulation Environment Design

Bringing Mosaik and Docker together yields a flexible and versatile co-simulation setup. This subsection describes three different kinds of containers, which are used by the simulation environment. An exemplary container setup is visualized in Figure 3.6.

Main Container

The Mosaik Core runs inside a main container and uses the Docker network to communicate with the simulators, which are running inside their own containers (see Figure 3.6). The main image is built from its individual Dockerfile. The configuration is based on the official Python 3.7 image provided on the Docker Hub. On this basis, the necessary third-party libraries are installed, most importantly the Mosaik and Docker Python packages. In order for the Docker-API to function, it needs to be able to connect to a Docker daemon. Therefore, the file system interface of the host system's daemon is mounted into the main container. This enables the container to access the host's Docker daemon and create/manage sibling containers. The scenario script and other relevant code is copied into the image and the default operation upon container start is set to execute the simulation scenario. The information, which additional simulation containers need

to be started, is already implicit in the simulation setup inside the scenario description. Therefore, Mosaik's Scenario-API was partially abstracted by an additional container-API class inside the main container. This class provides the ability to start Docker containers using Docker's Python-API [DOC20b]. Because the provided daemon is actually running on the host system, the created containers are running as siblings to the main container on top of the host system's kernel. When a Docker image name and a number of container instances is provided upon simulator initialization, the container-API starts the given number of container instances. The containers are then added to Mosaik's simulation metadata and initialized via the component-API. It is the users responsibility to assure that the given image is available to the Docker daemon. The container-API class keeps track of all initialized simulators and their containers. It also abstracts model creation because it realizes a simple form of load balancing. If multiple instances of a simulator are created, the model creation method divides them evenly between all simulator instances. Having multiple container instances for a large number of simulation models increases parallelization options and flexibility during distributed deployment. Finally, the model creation methods returns a list of proxy models, similar to the normal Mosaik Scenario-API. Those can then be used to create data connections between models using Mosaik's plain Scenario-API. All these interactions are visualized in Figure 3.5. When the container-API class is deleted, i.e. when the simulation is finished, it cleans up all running simulation containers it created.

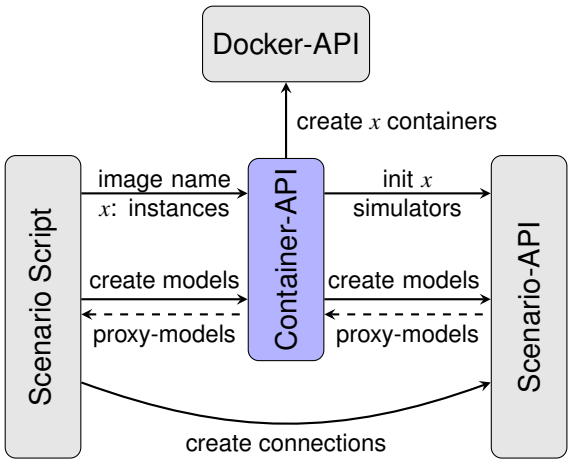


Figure 3.5: Co-simulation container-API functionality. It uses both the Docker-API and the original Mosaik Scenario-API to create container instances for the simulators. Models are created on all simulator instances in a balanced manner.

Simulation Containers

A simulation container encapsulates a single simulator, which can be used in conjunction with the main container through the use of Mosaik's component-API. They are build from

their own Dockerfiles. A standard simulation container image is based on an adequate base image, e.g. the official Python image for a Python-based simulator. On top of the base image, all third-party dependencies, such as system libraries, Python packages, etc. are installed. Lastly, the actual simulator code, including the Mosaik interface, is copied into the image. The order of those steps is relevant because this way the Docker layer caching accelerates the build process during simulator development. Lastly, the default command on container start is set to execute the main simulator with the Mosaik component-API implementation. In case of the deployment of an existing image from a local or online repository, the build step can be skipped. The image can be used to create any number of container instances, but it usually needs the containers' IPs or host names and the intended Mosaik port as additional arguments. These are forwarded to the simulator instance, so the Mosaik API knows where to to listen for incoming communication from the sim-manager. The simulator is not initialized at this point. It is waiting for the manager to establish a connection and trigger initialization. Usually, a simulator is set to terminate after a certain time of waiting for a connection, but this setting can be adjusted as needed during development of the simulator's Mosaik interface.

Auxiliary Containers

In addition to the Mosaik-related containers, two other containers are included for the simulation convenience. One is a database container, which runs a simple database server for simulation result storage. The tool used for this purpose is called InfluxDB. This is a simple time-series database. Such a database is optimized for the storage and retrieval of timestamped data points, which fits the data structure generated by a simulation. Here, a Docker volume can be used to save the database contents beyond the DB-container's lifespan for later evaluation.

Secondly, a Grafana container is deployed. Grafana is a web-server based data visualization tool. It can use several database systems, including InfluxDB, as a data source. Users can connect to the server using a web-browser. The visualization is organized in dashboards. Each dashboard consists of a collection of panels, which hold the actual visualized data. Upon first start of a new Grafana server previously designed default dashboards can be provided via the provisioning system. This default setup can then be individualized by the users by adding and editing dashboards and panels. This provides a highly customizable visualization environment, which can be adjusted by every user to the project and task at hand. All changes to the setup are local to the current browser session, but they can be saved to the server's internal visualization database. As long as this database is not dropped, the saved setup will always be recovered when the server is restarted. For the use with Docker, Grafana provides an official Docker image on Docker Hub. That makes it straightforward to deploy Grafana within the microservice architecture.

The container is started with the above mentioned database container as its default data source and a default dashboard is provided. A Docker volume saves the visualization database, such that the saved layout is preserved between container deployments.

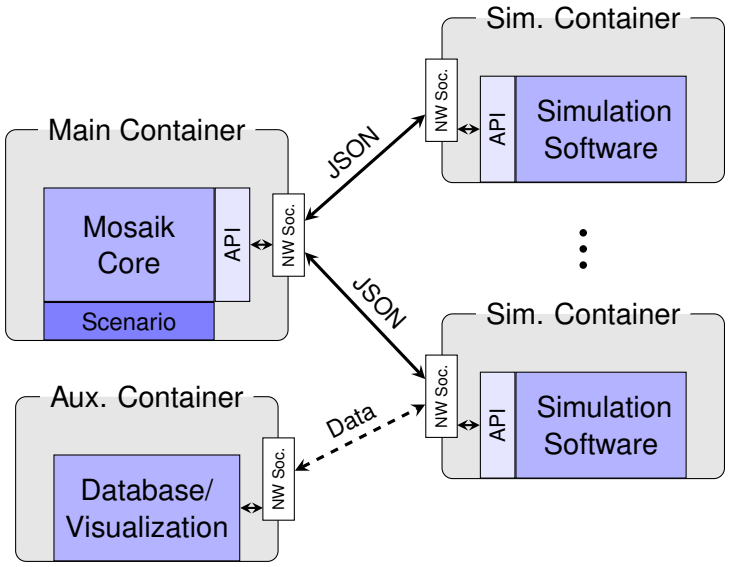


Figure 3.6: The main simulaton environment structure. A main container holds the Mosaik Core software and the Scenario Script. Each Simulator is packaged in its own container. Connection between the core and the smulators are realized using Mosaik's Component-API and the Docker network. Auxilary Containers provide additional functionality.

3.2 Flexibility Co-Simulation

The previously described simulation is now used to design and execute a simulation scenario with the goal of analyzing a flexibility coordination scheme. Firstly, the simulators that were implemented for this purpose are described (subsection 3.2.1). Afterwards, subsection 3.2.2 describes which instances of these simulators were created and how they are connected in order to create the final simulation setup.

3.2.1 Simulators

All simulators for the flexibility coordination analysis are implemented in Python. The external Python packages chosen during the development of the simulators provide suitable capabilities for the individual tasks. All simulators need to implement four functions in order to be able to interface properly with the Mosaik component-API. This structure gives the main pattern after which the simulators are designed.

Firstly, the *init*-function is called, when the simulators are first started. Here, global simulator properties can be set. For these simulators, the standard properties are the simulation's step size and an eid-prefix. Because there is no other coordination of consistent simulator execution, each simulator is designed to request their next execution step to take place one step size from the current time. This parameter should be set to the same value for all simulators, since it does not make any sense in this simulation setup to execute any of the simulator faster or slower than the other ones. The eid-prefix is a string that will be prepended to the entity-IDs used by Mosaik to identify the simulation model, for user convenience.

Secondly, the *create*-function is called to create a certain number of simulation models with a given set on initial parameters. The model's logic could be implemented directly into the simulator class. However, all simulation models in this project are implemented as their own model classes. This allows more flexibility, in case the simulation model needs to be refactored or exchanged for a different version. Additionally, it allows the simulator to save only a list of model instances and execute certain interface methods to interact with them. This greatly improves readability of the simulator class, especially if the simulator provides multiple model types.

Next, the *step*-function is executed on every simulation step. Here every simulator has a custom logic on how to handle different input parameters. However, each simulator at some point loops through their list of simulation models and executes their respective simulation logic for the current simulation time. Each model saves its state internally.

Finally, the *get_data*-function is used to retrieve the models' state variables. Here each simulator loops through the requested state variables, finds the respective model in its internal model list, retrieves its state and returns it to the Mosaik Core. This data is then propagated to other simulators by Mosaik according to the data connections defined in the scenario script.

Power Flow Simulation

This simulator provides the basis of any electrical power system simulation: the power flow simulation. It simulates the physical grid that connects the individual components and models their interaction. The basis for the simulation in this implementation is the Power Flow Equation (PFE) analysis tool *pandapower*. This utility is a combination of the data analysis library *pandas* and the PFE solver *pypower*. Pypower itself is a Python adaptation of the Matlab module *matpower*.

This simulator provides a variety of models, which represent the different elements within a power grid. These models include loads, generators, transformers, etc. with all their respective parameters and attributes. However, the only model that can be created through

the Mosaik API is a general *grid model*. This grid model is initialized by a configuration dictionary. This dictionary is parsed by a custom configuration parser. The parser supports a multitude of configuration settings. It can build a network directly from explicitly defined pandapower objects or read a similar configuration from a yaml-file. Alternatively, it can use pandapower's load function to initialize a previously saved network from a JSON-file. However the pandapower network is created, the simulator additionally creates a Mosaik-model representation for all grid elements. These models are then returned as children of the originally requested grid model. They can be used to interact with the power flow grid from other Mosaik simulator. For example, to set the power value of a load element inside the pandapower simulation from an external load simulator, the respective load child model is connected to the chosen load simulator model via Mosaik's scenario-API.

The simulation step function is what actually converts pandapower, which is only a solver, into a simulator. It parses all given inputs for the current simulation step and sets the entity parameters inside the simulated grids accordingly. Afterwards the resulting PFEs are solved.

Time Series Simulation

A time series simulator is a general purpose simulation tool. It is especially useful in case a system needs to be simulated whose inner proceedings are unknown or too complex to be modeled explicitly. In that case, statistical behavior data can be collected, for example in the form of an average daily output profile. This data is saved as a series of output values with attached time stamps: a time series. Such a time series is then interpolated by the simulation model to generate its output values and hence imitate the original system.

The only model implemented for this simulator is called *CSVTimeseries*. It's initialization data is outlined in Table 3.1. It reads the necessary time series data from a CSV-file. The file can be as simple as one time column and one data column, but it may hold multiple time and data columns. Which columns represent the time reference and data values is defined by additional parameters. Moreover, a constant time offset can be defined, in case the time reference does not correspond to the simulation time. Lastly, additive Gaussian noise with the given standard deviation can be applied to the output data. For that purpose a seed value can be provided. This ensures that the same random noise values are applied during every simulation execution, making the results reproducible.

When a simulation step is executed, the input values shown in Figure 3.7 are parsed. If a new scale or data column are given, they are applied. Afterwards, each model generates

model	parameter	description
CSVTimeseries	filename	CSV file that the time series data used to produce the simulator output is read from
	time column	Column within the data, which holds the data points' time reference
	data column	Column within the data, which holds the data values
	interpolation	The scheme, which is used for data interpolation: linear or nearest-neighbor
	offset	Time offset, which is applied to the time series data
	scale	Scaling factor applied to all data values
	seed	Seed value for the noise generating RNG
	std. dev.	Standard deviation for the noise generating RNG.

Table 3.1: Time series simulator initialization parameters

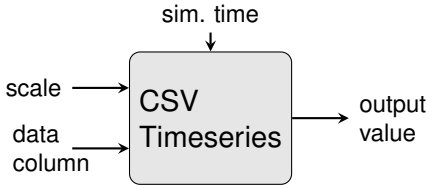


Figure 3.7: CSVTimeseries input/output block diagram

its new output state according to the simulation time, data, interpolation scheme and other settings.

ESS Simulation

This simulator emulates the behavior of Energy Storage System (ESS)s. The implemented model is called *storage* and represents a simple storage system connected to a power bus by an adequate transformation unit and some internal protection systems. On model creation, the parameters listed in Table 3.2 must be provided. E_{max} describes the maximum amount of energy that can be stored inside the system. E_{SoC} is the initial amount of energy inside the system st the simulations start. The value of ε is the system's efficiency, which represents the energy loss during energy transactions. A value of 0 implies all energy is lost and a value of 1 means perfect efficiency. Finally, P_{max} is a tuple of two values, which represent the system's maximum charging and discharging power respectively.

During the simulation step, the new SoC for the current simulation step is calculated in each model. Since a storage system would have no reason to change its state by itself, an external set point can be provided. The storage system then adopts this set point, if it is within the accepted power boundaries, and updates its SoC accordingly. The theoretical

model	parameter	description
storage	E_{max}	System maximum energy capacity
	E_{SoC}	Initial system SoC
	ε	System efficiency
	P_{max}	Tuple of maximum charge and discharge power values

Table 3.2: ESS simulator initialization parameters

behavior follows the integral:

$$soc(t+t_{step}) = soc(t) + \int_{t}^{t+t_{step}} \varepsilon \max(P_{set}(\tau),0) + (2-\varepsilon)\min(P_{set}(\tau),0)d\tau \tag{3.1}$$

The value for $soc(t)$ is already known because it is the result of the previous execution or the initial SoC in case it is the first execution step. The integral also takes into account the fact that an efficiency value of $\varepsilon < 1$ causes a decrease in charged energy, but an increase in retrieved energy compared to an ideal storage system.

However, the set point is expected to be a constant value for the given time step, so the integral can be solved to:

$$soc(t+t_{step}) = soc(t) + \quad \varepsilon \; P_{set} t_{step}, \quad if P_{set} > 0 \tag{3.2}$$

$$soc(t+t_{step}) = soc(t) + (2-\varepsilon) P_{set} t_{step}, \quad if P_{set} < 0 \tag{3.3}$$

After calculating this theoretical Energy demand the energy system model verifies that this operation will neither lower the SoC below 0 nor increase it beyond the maximum energy level. In such a case, it will adjust the set point to stay within these bounds. The system outputs are its actually applied power value during the current time step and the SoC at the end of that time step. An overview is given in Figure 3.8.

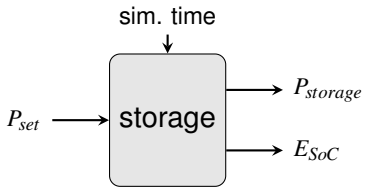


Figure 3.8: Storage model input/output block diagram

HEMS Coordinator Simulation

The HEMS coordinator simulator was designed to simulate the central decision making entities for flexibility coordination inside of a single household energy system. Its main job is to provide the set point for the ESS simulation.

Two models are implemented for this simulator, one is called *SetPointHEMS* and the other *MinMaxHEMS*. Both types of model are initialized with the target ESS model's initial SoC E_{SoC}, maximum power rating P_{max} and maximum energy level E_{max}. Considering the implementation of the ESS simulation model mentioned above, these parameters are not necessarily needed because the storage system will protect itself from overcharging. However, in real physical systems that is more of an security mechanism. It is always better for the controlling system to know the state and boundaries of the controlled system. After all, the ESS model might change to one that relies on the HEMS model for protection.

model	parameter	description
SetPointHEMS & MinMaxHEMS	E_{max}	Controlled ESS's maximum energy capacity
	E_{SoC}	Controlled ESS's initial SoC
	P_{max}	Controlled ESS's maximum power ratings

Table 3.3: HEMS coordinator simulator initialization parameters

When the execution step is called, each EMS model needs its respective household's current power consumption P_{load} and generation P_{gen} as input. Those values are usually produced by other simulators, which simulate the households load behavior and any DER inside the household energy bus. Additionally, it needs to know the storage system's current SoC, to produce a valid decision. Then the two models diverge.

The SetPointHEMS model only needs one additional attribute. This is a set point for the combined household power demand P_{goal}. The system then aims for an ESS set point

$$P_{storage,set} = P_{goal} - (P_{load} - P_{gen}).$$

If $P_{storage,set}$ is a valid set point, the overall set point P_{goal} is usually perfectly reached.

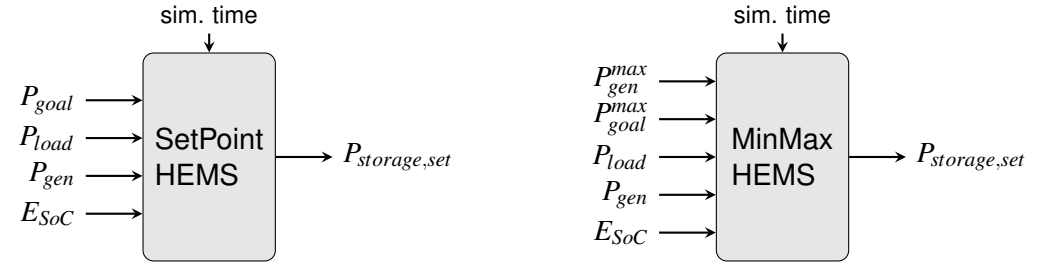
The MinMaxHEMS model needs two additional attributes, a maximum load value P_{load}^{max} and a maximum generation value P_{gen}^{max}. These two points are interpreted as upper and lower bounds of the overall household demand. The set point $P_{storage,set}$ for the ESS is therefore chosen, so that (3.4) holds.

$$P_{gen}^{max} \leq P_{load} - P_{gen} + P_{storage,set} \leq P_{load}^{max} \tag{3.4}$$

In case the difference $P_{load} - Pgen$ already satisfies this condition, the set point is set to $P_{storage,set} = 0$.

In both cases the set point is verified against the ESS's physical restrictions. In case they are violated the desired value is adjusted to the closest valid one. Another interesting observation is, that the latter model can emulate the former by setting $P_{load}^{max} = P_{gen}^{max} = P_{goal}$.

However, the readability and intent of the simulation scenario is improved by adding the additional interface.



(a) SetPointHEMS input/output block diagram **(b)** MinMaxHEMS input/output block diagram

Figure 3.9: HEMS coordinator input/output block diagrams

A complete physical HEMS usually includes additional components, like a predictor and an optimizer. Those systems, however, can be seen as individual subsystems. Therefore, it makes sense to externalize them into their own simulators. That way different system architectures can be exchanged more easily and the interfaces are more clearly defined.

Predictor Simulation

As mentioned in section 3.2.1, the prediction mechanism is usually part of a physical EMS, but is externalized into its own simulator. It describes a subsystem, which provides behavioral predictions for another system's value/output. The complexity can reach from simple averaging of the system's past behavior, over more sophisticated statistical approaches, like a Kalman filter, to a neural network which is trained to deduce the systems future behavior. As introduced in subsection 2.1.3, a MPC scheme will be deployed to control the prosumer flexibility. This simulator will provide the 'model' entities, as shown in Figure 2.1.

The only model implemented in this setup uses a simple averaging and interpolation logic. It resembles largely the time series simulation model and implements all of its features, as described in section 3.2.1. It is called the *CSVPredictor* and the initialization are exactly identical to the CSVTimeseries as listed in Table 3.1. When a model is created it can also be provided with data from a CSV-file. This data will form the basis for the prediction. For any project which does not specifically focus on analyzing prediction algorithms, this is useful because the predictor can be initialized fully trained. Therefore, no simulation time has to be spent to train the predictor, before any other mechanisms can be analyzed. Additional parameters include the prediction window and step size.

During the simulation step execution the predictor model will generate predictions for a timespan the length of the prediction window Δt and starting at the current simulation time. The resolution of the predictions is set by the input t_{res}. If none is provided, it is set to the predictor simulators step size. For each prediction point, the prediction data is interpolated to generate a fitting value. Additionally, the predictor can receive the actual value of the system it is predicting at the current simulation time v_{curr}. If such a value is provided, it is set as the first prediction value instead of the possibly inaccurate prediction. This is done only for convenience because this value could simply be left out and provided to any subsequent simulator directly. However, by looping it through the predictor, only one data connection is needed to get the system's current and future behavior for the whole prediction window. The output is a list of the predicted values. An overview is given in Figure 3.10

Lastly, the input values can be used to train the predictor and make it adjust to changing system behavior. If activated, the underlying prediction data is constantly updated, such that

$$v_{pred,new}(t_{curr}) = \frac{v_{pred,old}(t_{curr}) + v_{curr}}{2}.$$

Where $v_{pred,new}(t_{curr})$ is the new, updated prediction value at the current simulation time, $v_{pred,old}(t_{curr})$ is the previous prediction value for the current simulation time and v_{curr} is the current value of the system that is to be predicted.

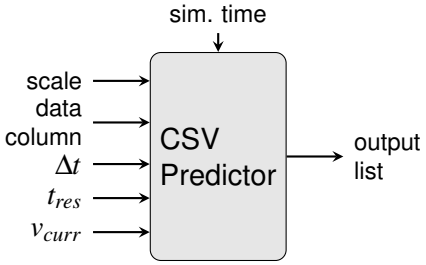


Figure 3.10: CSVPredictor input/output block diagram

Local Flexibility Controller

This simulator also provides models which are part of a physical HEMS system. The goal is to gather the household's predicted consumption and generation behavior in order to decide how to optimally use the given flexibility in from of an ESS. As introduced in subsection 2.1.3, a MPC scheme will be deployed to control the prosumer flexibility. This simulator will provide the 'optimizer' entities, as shown in Figure 2.1.

The only model type available for this simulator is called *MILPOptimizer* and provides a peak-shaving optimization. It uses the *pyomo* Python package to model the optimization

problem and the COIN Branch and Cut (CBC) solver to find the optimal solution. Upon creation, the model is initialized with the values listed in Table 3.4. They include the maximum energy level E_{max}, maximum power ratings P_{max} and efficiency ε of the storage system, which is available as a flexibility resource.

model	parameter	description
MILPOptimizer	E_{max}	Optimized ESS's maximum energy capacity
	P_{max}	Optimized ESS's maximum power ratings
	ε	Optimized ESS's efficiency

Table 3.4: Local flexibility controller initialization parameters

When the simulation step is executed, the model is provided with prediction curves for the household's total power generation P_{gen}^{pred} and consumption P_{load}^{pred}. It also receives the household's ESS's current SoC $E_{SoC,init}$. These parameters are used to construct a MILP optimization problem with the objective function of:

$$\min P_{gen}^{max} + P_{load}^{max} \tag{3.5}$$

where

$$P_{tot,i} = P_{load,i} - P_{gen,i} + P_{storageCh,i} - P_{storageDis,i}, \quad \forall i \tag{3.6}$$

$$P_{load}^{max} \geq P_{tot,i}, \quad \forall i \tag{3.7}$$

$$P_{gen}^{max} \geq -P_{tot,i}, \quad \forall i \tag{3.8}$$

with $P_{load,i} \in P_{load}^{pred}$ being the predicted household consumption and $P_{gen,i} \in P_{gen}^{pred}$ being the predicted household generation values for each time step i into the future. The values of $P_{storageCH,i}$ and $P_{storageDis,i}$ are the values representing the ESS's charging and discharging power for any given time step. The sum in (3.6) is modeling the expected overall household demand, which is to be optimized. The value of P_{load}^{max} is the upper bound of the overall household energy consumption and P_{gen}^{max} is the maximum of the household's power injection into the grid. Alternatively $-P_{gen}^{max}$ can be interpreted as the lower bound of the overall demand curve. By doing this min-of-max optimization, the system will automatically express a peak-shaving behavior. When the injection is high it will charge the storage system to lower P_{gen}^{max} and when the consumption is high it will discharge in order to lower P_{load}^{max}. The process is illustrated in Figure 3.11.

Additionally, the mathematical model of the ESS has to reproduce the physical bound-

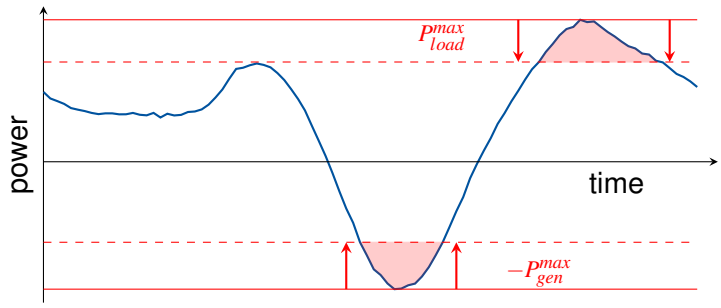


Figure 3.11: Illustration of the min-of-max optimization algorithm.

aries of the system:

$$0 \leq P_{storagech,i} \leq P_{storageCh,max} \cdot dir_i, \quad \forall i \tag{3.9}$$

$$0 \leq P_{storageDis,i} \leq P_{storageDis,max} \cdot (1 - dir_i), \quad \forall i \tag{3.10}$$

These constraints limit the charge and discharge power to their respective intervals. They cannot be negative, and they cannot be higher than the maximum power rating of the storage system. Here, the rating for charging and discharging are two distinct values in case the system does not behave symmetrically. Note the additional variable dir_i is a binary value, meaning it can take up the value of 0 or 1. It ensures that the storage system cannot charge and discharge at the same time in the mathematical model.

Next, it is necessary to model the SoC:

$$E_{SoC,0} = E_{SoC,init} + (\varepsilon P_{storageCh,0} - [2-\varepsilon] P_{storageDis,0}) t_{step} \tag{3.11}$$

$$E_{SoC,i} = E_{SoC,i-1} + (\varepsilon P_{storageCh,i} - [2-\varepsilon] P_{storageDis,i}) t_{step}, \quad \forall i \geq 1 \tag{3.12}$$

$$0 \leq E_{SoC,i} \leq E_{max}, \quad \forall i \tag{3.13}$$

The constraint in (3.12) describes how the SoC changes from one time step to the next, by calculating the (dis)charged energy using the respective power values and the given time step size. This works for all steps except the first one. For the first step, (3.11) uses the ESS's current SoC as the initial energy level. Both constraints also model internal losses in the storage system. The parameter ε represents the system's efficiency, which is set during model creation. That makes the system slightly asymmetrical, because discharging with a certain power uses more energy than charging with the same power deposits. The efficiency is only valid in the interval $[0,1]$. An efficiency of 0 would mean all energy is lost during charging and an efficiency of 1 represents a lossless system. Finally, (3.13) ensures that the SoC can never drop below 0 (undercharging) or increase beyond the initialized energy maximum E_{max} (overcharging).

Finally, there is one functional constraint:

$$P_{storageCh,i} \leq P_{gen,i}, \quad \forall i \tag{3.14}$$

This constraint is not taken from a physical system boundary, but from the fact that it is never beneficial for the prosumer to charge its storage system from the grid. This results from the ESS's internal losses. Assuming a constant energy price, it is always advantageous to draw necessary energy from the grid when needed, instead of saving it beforehand. Therefore, the ESS is only allowed to charge from the generation unit. Furthermore, this ensures that any stored power is purely produced by RESs. This is especially relevant for statistical and financial reasons, if power is fed back into the grid from the ESS.

After solving this problem, the model has found the ideal charge and discharge pattern for its flexibility source in order to cut off load and generation peaks locally. However, this flexibility usage is not necessarily optimal from a global view point. In order to provide an overlying controller the opportunity to coordinate multiple independent prosumers, the flexibility margins are calculated. In other words, it needs to determine how much the prosumer can diverge from its optimized demand curve at any given point during the prediction. The minimal boundaries are given by the physical system limitations. Therefore, with

$$P_{storage,i} = P_{storageCh,i} - P_{storageDis,i}, \quad \forall i \tag{3.15}$$

$$P_{pred,i} = P_{load,i} - P_{gen,i} + P_{storage,i}, \quad \forall i \tag{3.16}$$

as the predicted household storage and demand curves after optimization, the flexibility bounds will be:

$$P_{flexMin,i} = \min\left(P_{storageDis,max}, \frac{SoC_i}{t_{step}}\right) - P_{storage,i}, \quad \forall i \tag{3.17}$$

$$P_{flexMax,i} = \min\left(P_{storageCh,max}, \frac{E_{max} - SoC_i}{t_{step}}, P_{gen}\right) - P_{storage,i}, \quad \forall i \tag{3.18}$$

Finally, the model takes the first time step in each curve and uses it to determine the control value for the current simulation step. For that purpose it checks if a flexibility request of an overlying controller has arrived via the input $P_{flexReq}$. In such a case, the request is verified against the newly determined flexibility margins and used as the set point, otherwise the value from the local optimization is taken. This result can be used as the desired set point (P_{goal}) in the HEMS simulation described in section 3.2.1. All other time steps in the optimized curves are seen as prediction curves for future time steps. They are available and can be connected to a superordinate flexibility coordinator

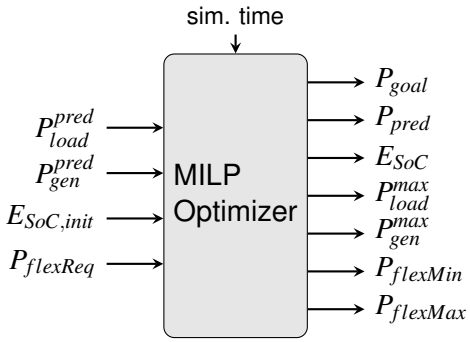


Figure 3.12: MILPOptimizer input/output block diagram

as described in section 3.2.1. An overview over all values is provided in Figure 3.12.

Coordinating Flexibility Controller

This simulator handles models which are part of a grid level EMS. These models take the role of optimizers in an overarching second-layer MPC scheme, compared to the local MPC described in section 3.2.1. They aim to coordinate a collection, cell or neighborhood of flexible prosumers and normal consumers. To take an informed decision, the controller needs predictions of the all participants' behavior and potential flexibility. Those predictions can either originate from the prosumers' HEMS or be generated by other prediction mechanisms inside the coordinating EMS.

The only implemented model is called *LPOptimizer* and provides another version of the peak-shaving algorithm. The model has no initialization parameters because it is designed to handle participants dynamically and no other information is necessary.

During the execution step, the model analyzes the received prediction data. Only entities for which prediction data is provided considered for optimization and are therefore able to receive flexibility set points. Entities which provide no flexibility margin are considered to not be able to act as flexibility resources and are only considered for optimization as a static component. Traditional customers usually fall into this category. Active prosumers on the other hand provide a forecast of their intended system behavior, based on their local optimization. This behavior consists of the predicted demand curve $P_{pred,i,e}$, the flexibility margins $P_{flexMin,i,e}, P_{flexMax,i,e}$ and a projection of their SoC $E_{SoC,i,e}$. This SoC information is necessary, because the controller would not have any indication on how the change in demand at one point in the curve affects the flexibility margins at a later point due to the maximum capacity $E_{max,e}$ of the storage systems. The parameter e represents the different entities/participants of the optimization and i indexes the consecutive time steps. All the provided predictions are accumulated to find the overall neighborhood

behavior:

$$P_{pred,i} = \sum_e P_{pred,i,e}, \quad \forall i \tag{3.19}$$

$$P_{flexMin,i} = \sum_e P_{flexMin,i,e}, \quad \forall i \tag{3.20}$$

$$P_{flexMax,i} = \sum_e P_{flexMax,i,e}, \quad \forall i \tag{3.21}$$

$$E_{max,cell} = \sum_e E_{max,e}, \quad \forall i \tag{3.22}$$

$$E_{SoC,i} = \sum_e E_{SoC,i,e}, \quad \forall i \tag{3.23}$$

The optimization is constructed as a Linear Programming (LP) problem with the objective function:

$$\min P_{max} - P_{min} \tag{3.24}$$

where

$$P_{min} \leq P_{flex,i} \leq P_{max} \quad \forall i \tag{3.25}$$

Similarly to the local problem in section 3.2.1, this objective function tries to minimize the difference between P_{max} and P_{min}, the upper and lower bounds of the optimized demand curve. The variable $P_{flex,i}$ is the to-be-requested flexibility value in each time step, which the optimization will determine. This approach, again, will shave off the peaks both in power injection and consumption, as illustrated in Figure 3.11.

Firstly, the flexibility request is constrained by the accumulated flexibility margins:

$$P_{flexMin,i} \leq P_{flex,i} \leq P_{flexMax,i}, \quad \forall i \tag{3.26}$$

Additionally, the collective SoC of the neighborhood will be modeled, in order to depict the influence that a flexibility request has on the future of the flexibility margins.

$$E_{SoCflex,i} = E_{SoC,i} + t_{step} \sum_{j=0}^{i} P_{flex,j}, \quad \forall i \tag{3.27}$$

$$0 \leq E_{SoCflex,i} \leq E_{max,cell}, \quad \forall i \tag{3.28}$$

Constraint (3.27) takes the neighborhood's accumulated predicted SoC $E_{SoC,i}$ and adjusts it according to the changes caused by the flexibility requests. That is done per time step by summing up all flexibility requests until the respective time step and multiplying it by the step duration, to get the overall change in energy demand on that step. These changes are then added to the respective SoC. Finally, (3.28) ensures that the adjusted SoC can

never exceed the accumulated maximum stored energy.

In the end, the model demultiplexes the global flexibility request into individual requests to the different prosumers. This is done by requesting flexibility adjustment from each participant relative to their offered flexibility. Equations (3.29) and (3.29) show, how the individual requests are calculated for each participant e.

if $P_{flex,i} < 0$:

$$P_{flex,i,e} = P_{flex,i} \frac{P_{flexMin,i,e}}{P_{flexMin,i}}, \quad \forall i,e \tag{3.29}$$

if $P_{flex,i} > 0$:

$$P_{flex,i,e} = P_{flex,i} \frac{P_{flexMax,i,e}}{P_{flexMax,i}}, \quad \forall i,e \tag{3.30}$$

These individual flexibility requests are written back as an input to the local flexibility controllers using Mosaik's asynchronous *set_data*-call. This allows the global flexibility coordination to write the requests into the individual input buffers of each receiving participant, instead of broadcasting them to all connected simulators via an output attribute. A schematic overview of the models input/output behavior is provided in Figure 3.13

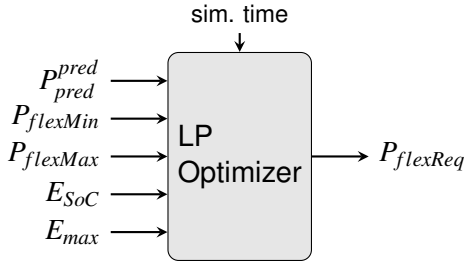


Figure 3.13: LPOptimizer input/output block diagram

Data Collector

As described in subsection 3.1.1, the Mosaik structure allows services to be included into the simulation infrastructure that are not actual simulators, as long as they adhere to the interface. The data collector is such a service. It is designed with the same methods as the other simulators, however, it plays no active role in the actual simulation procedure.

The simulator only provides one model, which is the *DatabaseCollector*. This model connects to an external database server, in this case an InfluxDB server (see section 3.1.3). The only initialization parameter is the collectors name, which also serves as the database name. For each simulator that delivers data to the collector model, a

separate database tables is created. In that table, the index column holds the given simulation time stamp. The other columns represent the simulation data parameters. So during each simulation step, a new row is added with the corresponding time step and the current simulation data values.

As shown in Figure 3.14, the model has the special feature, that it can receive any input parameter. Normal simulators restrict their inputs to their respective interfaces. This enables the user to monitor any simulation data by simply connecting the model output(s) of interest to the data collector model. The collector model will then save the produced data in the respective table. The model currently has no output parameters, because it is intended to be a simple data sink. Additionally, the simulator restricts the model creation to one model instance. This ensures all data is saved to the same location and no data fragmentation or conflicting database operations occur.

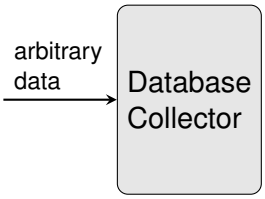


Figure 3.14: DatabaseCollector model input/output block diagram

3.2.2 Simulation Data Flow

For the purpose of simulating flexibility coordination inside the newly developed co-simulation environment, a simulation scenario was designed. This scenario includes information about which simulators and simulation models are used and how the data flows between them.

To visualize the simulation scenario, the graph structure introduced in Figure 3.15 will be used. The different square blocks represent the different simulator instances, with their name written at the top. The simulator type, written in **bold**, is one of the simulators described in subsection 3.2.1. The name of the chosen model within the simulator is given in <triangle brackets>. This information is especially relevant, when the simulator provides multiple models. Afterwards, in *italic* font, additional information on the specific simulator is given. Such information includes for example initialization data.

The final simulation setup can be seen in Figure 3.16. This overview shows the different simulators representing the individual components. Additionally, it shows which parts of the simulation represent a certain physical system together.

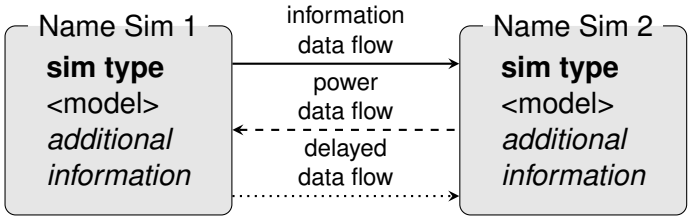


Figure 3.15: Data flow chart example

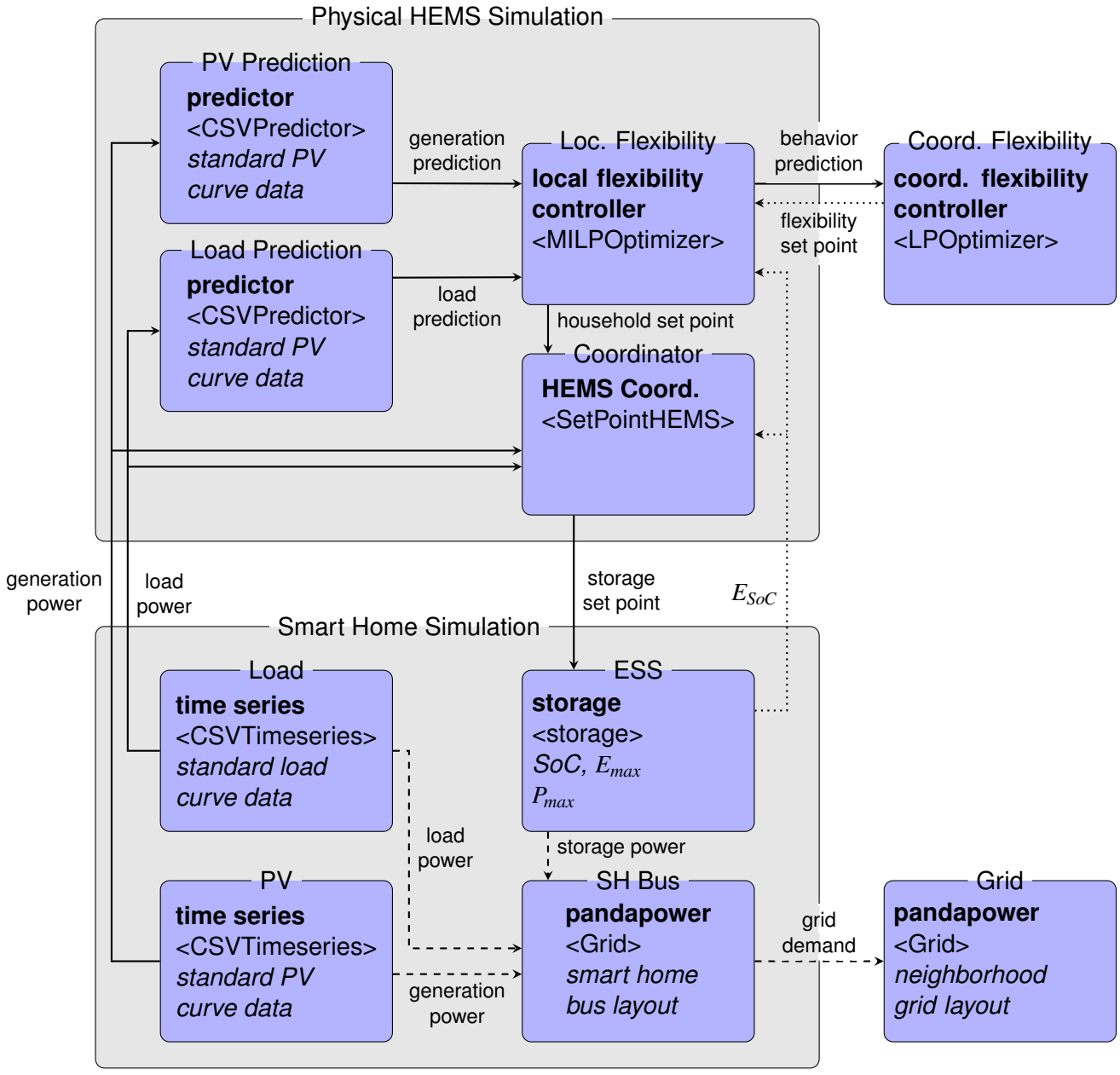


Figure 3.16: Complete simulation data flow

The basis of the scenario is the household power grid simulation. This system includes all simulators that represent the electrical grid and its components inside a single (smart) home. Here, two time series simulators are used to represent the household's load and generation power. The data for the load simulation is a typical household demand curve [BDE17]. The source CSV-file provides profiles for different times of the year and different weekdays. The individual data source can be chosen through the data column attribute according to the simulated date, but in order to keep the results more comparable the same curve will be used throughout all simulations in this case. Using the scale parameter, the data can be adjusted to the desired household size.

The household's power generation is simulated as a PV unit. The source data is a simple Gaussian bell curve with its maximum at 1 p.m., which can be used as an adequate approximation of the PV system's behavior over the course of a day. Since the curve has a maximum value of 1, the scale attribute should be set to the peak power of the system, which might depend on the system dimensions as well as weather conditions. Both the generation and consumption curve are distorted by uncorrelated Gaussian noise. This does not properly model the typical variation of the systems from their average behavior, but it provides some variance, which makes the behavior more individual.

The third element in this grid simulation is the storage element. It is simulated by the storage simulator, using the basic storage system model. As described above, the model implements a system with a fixed maximum capacity, fixed maximum power ratings and internal losses. These limitations are absolute and, if necessary, will be enforced by internal protection mechanisms.

All three simulators deliver their output to the actual power grid simulator. This simulator is a pandapower-based power flow simulator. The power grid is set up according to the top half of Figure 3.17, with the three grid elements connected to the households power bus. The simulators for each element provide the values for each element respectively. Line losses are not considered inside the household bus, therefore the power flow simulation simply accumulates the individual power contributions of the elements. The result is provided through a fourth element, called the *external grid*, which is the point where the household is connected to an external distribution grid.

The second simulated system is the prosumer's HEMS. Its purpose is to provide structural flexibility to the household's power grid. The physical flexibility is given by the storage system, which needs an external set point. This set point is generated in the HEMS.

The basis for the generation of the set point is the prediction of the household behavior. In a real system, a predictor would be trained over time to be able to infer the future behavior from past data and additional information, like time and date. For the purpose of this co-simulation setup, two simulators are used, one for the load behavior and one

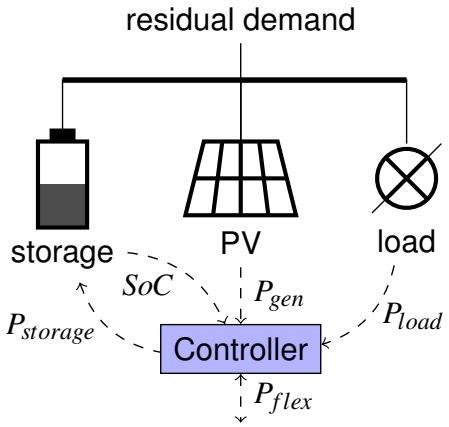


Figure 3.17: Smart Home layer. Each prosumer network consists of three basic elements. Household load, a PV generation unit and an energy storage system.

for the generation behavior. Each model is initialized with the same CSV-files as the respective system it predicts. That way, the predictor resembles an already well-trained prediction algorithm. As long as the load and PV simulators itself apply noise to their outputs, the predictions are imperfect, but resemble a daily average of the behavior. The simulators additionally take the current value of their to-be-predicted systems as an input, in order to update their initial prediction.

The predicted curves are then handed to the next simulator, which is the local flexibility controller. It solves the MILP optimization problem, as described in section 3.2.1 and generates the overall household demand set point.

That set point is used by the coordinator simulation to generate the actual ESS set point. In principle, this is done by simply calculating the difference

$$P_{storage} = P_{setpoint} - (P_{load} - P_{gen}).$$

For this purpose, it also receives the currently measured load and generation power. However, the coordination also takes into account the storage systems physical limitations and adjusts the set point to stay within these constraints in case the flexibility controller did not do so.

Finally, the local flexibility controller and the coordinator both require to know the ESS's SoC at the start of the current time step. This value is provided in a time-delayed manner, meaning the ESS simulator is executed after these two simulators and so the provided value always originates from the previous simulation step. This is no problem though, because the storage system's output SoC represents its state at the end of the current time step, which corresponds the beginning of the next step. As there is no previous

step to the first execution step, the connection between the ESS and the optimizer and coordinator is initialized with the same initial SoC that the storage system model uses.

Lastly, multiple household grid simulations are combined to simulate a whole neighborhood of coordinated prosumers. Both the simulation of the power grid and of the control infrastructure are therefore connected to their global counterparts. The external grid elements of each household's power flow simulation is connected to a single global power flow simulation, which represents the whole neighborhood's distribution grid, as shown in Figure 3.18. This simulator is another pandapower-based power flow simulator, which is provided with the corresponding neighborhood's grid layout. Since the goal of this simulation is the analysis of general flexibility coordination, the individual participants are considered to be close enough together, such that line losses are still considered negligible. This also makes the resulting power flow simulation a simple accumulation of all individual grid elements' demands. The resulting neighborhood behavior is again available at this grid's external grid node, which represents the point where the neighborhood grid is connected to the higher voltage levels.

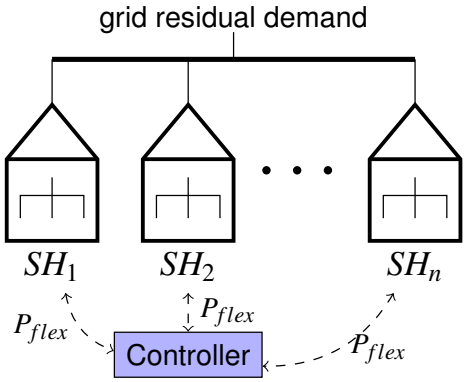


Figure 3.18: Neighborhood grid layer.

The corresponding global element for the control system is the coordinating flexibility controller. As described in subsection 2.1.2, the direct market demand response approach is used. This means that each prosumer provides a modeled prediction of their future behavior and an explicitly expressed flexibility potential. The global flexibility coordinator then solves the LP optimization problem described in section 3.2.1 and returns an explicit flexibility allocation to each participant. This flexibility request is also realized in a time-delayed manner. Thus, the returned flexibility allocation is only relevant from the upcoming step onwards and a circular data dependency is avoided.

4 Results and Discussion

In this chapter, the energy system as well as the simulation environment, which have been previously modeled and implemented, are demonstrated. A systematic overview of the final flexibility simulation scenario is displayed in Figure 4.1. In subsection 4.1.1, the basic functionality of the environment will be tested. In addition, this provides an opportunity to observe the consumption and generation data, which form the base for all further simulations. In subsection 4.1.2, a single household simulation is assembled and analyzed throughout different scenarios of increasing complexity. This isolated analysis emphasizes the changes that occur once multiple households are combined in subsection 4.1.3. There, 20 households are combined and their collective behavior is analyzed. Afterwards, the established control system will be tested under high noise and time shifted prediction conditions. Known problems and performance boundaries will be discussed (section 4.2). Lastly, an additional environment feature, the data visualization service is showcased (section 4.3).

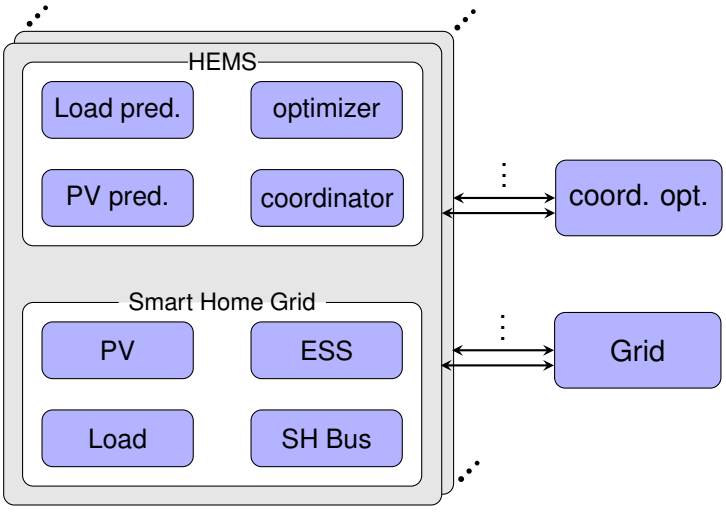


Figure 4.1: Simulation scenario overview

4.1 Exemplary Flexibility Coordination Simulation

In this section, the general behavior and functionality of the designed flexibility coordination scheme will be analyzed in different scenarios. Therefore, the simulators are

initialized with realistic, yet low-noise scenario. The scenario parameters can be found in Table 4.1. Unless otherwise defined, these values are used to initialize all necessary simulators. The amount of consumed and produced energy in conjunction with the storage capacity are chosen, such that the storage system is not able to simply store all the produced energy.

simulator	parameter	value
Load	peak P_{load}	11.3 kW
	noise σ	100 W
PV	peak P_{gen}	15 kW
	noise σ	100 W
Storage	P_{max}	30 kW
	E_{max}	10 kWh
	E_{SoC}	0 kWh
	ε	0.95

Table 4.1: Initial values for the basic smart home simulators.

4.1.1 Environment Test

Firstly, the general environment needs to be tested. For that reason, a simple test scenario was designed, where one time series simulation connects to the data collector, as described in section 3.2.1. This setup verifies the correct execution of the simulators, the Mosaik interface and the database system used for data storage. The scenario was executed twice, once with the standard load curve and one with the standard PV generation curve.

The results can be seen in Figure 4.2 for the load curve and Figure 4.3 for the PV curve. The dashed plots follow the general standard load and PV curves given as the input data exactly. The solid plots show, how the curves diverge from the original, when noise with a standard deviation of 700 W is applied. This first scenario shows, that the environment works as expected, running the respective containers with their simulators and collecting the data for later analysis. It additionally showcases the standard curves used as a basis for all simulations and how the time series simulation randomizes the behavior.

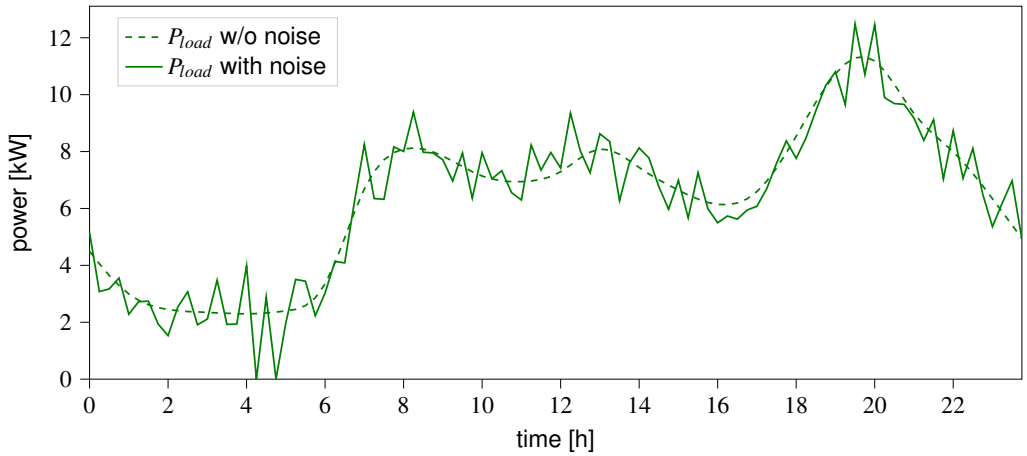


Figure 4.2: Standard Household Load Consumption Curve

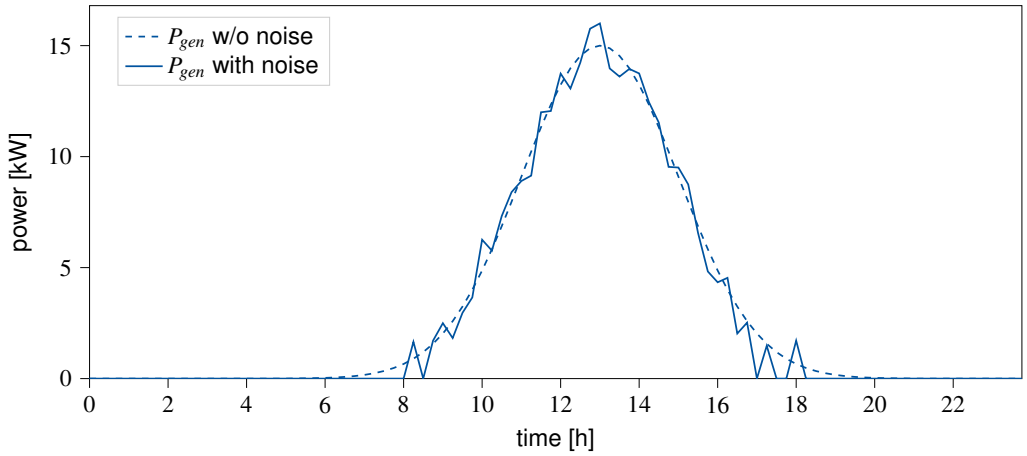


Figure 4.3: Standard Household PV Generation Curve

4.1.2 Single Household

After verifying the general environment behavior, the co-simulation scenario was iteratively increased in complexity. The first steps involved designing a single smart home setup, a prosumer, which could later take part in a superordinate form of flexibility coordination. The basic layout of the electrical grid of such a smart home can be seen in Figure 4.4 along with the parameters, which will be observed throughout this chapter. This simulation data will be used to analyze and evaluate the simulation scenario. The load and PV elements only provide their static consumption/generation power values P_{load} and P_{gen}. In the case of the storage system, the applied power value $P_{storage}$ can be observed along with its SoC (in %). One of the most relevant values is the residual household demand, which is the accumulated value of all three home grid elements. This values is important, because it represents the behavior of the household as seen from an outside perspective, e.g. by the energy distribution grid. Oftentimes, the household demand is plotted together with a curve which represents the accumulated behavior of

only the load and PV entities, without the storage system. That curve shows the behavior of the smart home in case it had no storage capabilities, therefore emphasizing how an ESS influences the household's overall behavior.

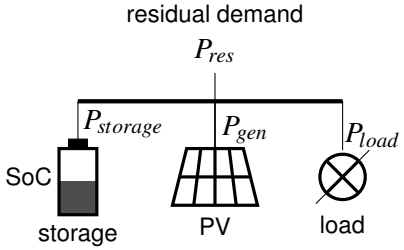


Figure 4.4: Single household grid layout, including observed parameters

No Storage Scenario

Initially, the load and PV simulations were combined with a simple power flow simulation to create a household with a DER, but no flexibility. The resulting household demand is displayed in Figure 4.5. For reference the original demand curve without the PV system has been included in the plot. It can be seen that the resulting curve looks like the standard PV curve subtracted from the standard load curve, which is exactly what was expected of this scenario. This result again highlights the relevance of grid flexibility. A traditional customer with a standard load profile has some variability, but it never injects energy back into the grid. When a significant amount of private households install PV systems, their demand curves start being more variable. Transformers need to face the challenge of meeting vastly different demand levels during the course of a single day and even handle excess injected energy. This problem occurs because the energy generation is correlated between the household, since all of them will produce most of their energy when the sunlight level is highest. This is augmented by the fact that in a typical household the peak in the consumption curve happens in the evening (as shown in Figure 4.2), which does not coincide with the peak in the PV generation curve around noon. Therefore, the demand curve changes over the course of the afternoon from high injection to high consumption behavior in all households with PV systems similarly. This pattern, which can be observed in the plot, is also sometimes called a *duck curve* [DEN15].

Simple Storage Scenario

A solution to the challenges faced by the grid operators mentioned in the last scenario would be to store excess energy locally and use it at a later point, when the demand is

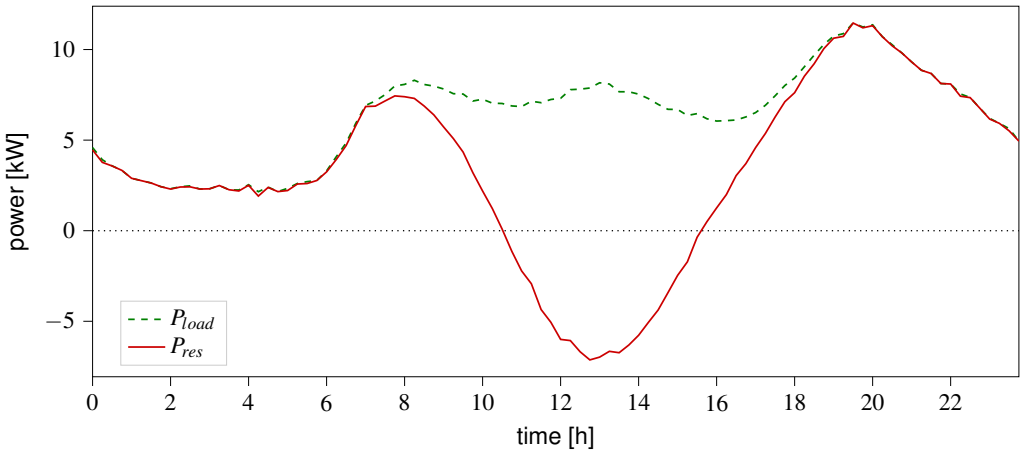


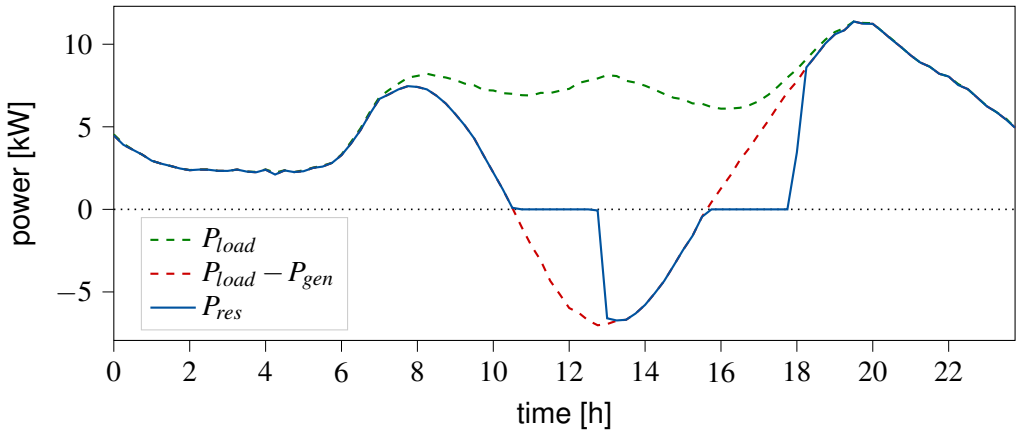
Figure 4.5: Single household, no storage scenario: residual demand curve and standard load profile for reference

higher. This is usually economically more efficient than injecting power back into the grid. An ESS can be used to do exactly this and thus enables a household to change their demand behavior. In other words, it provides physical flexibility.

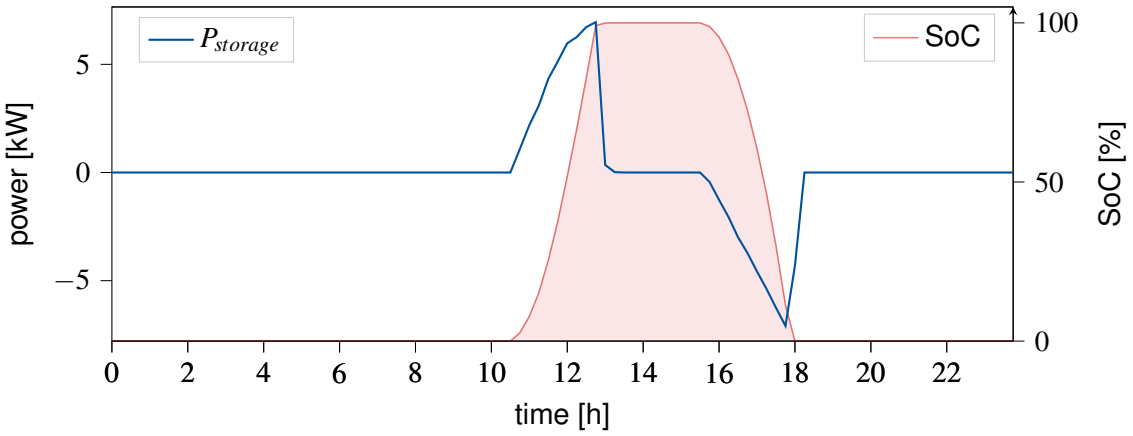
A simple example of the usage of this flexibility can be seen in the next simulation plot (see Figure 4.6a), which includes a storage system simulator and a HEMS coordinator in addition to the previous setup. The set point for the coordinator in this case is always 0, meaning it will store excess energy inside the storage system whenever possible. Furthermore, it will draw on said energy, whenever the household consumption is higher than the energy created by the PV system. It ensures an optimal use of the storage system from the customer's point of view, since the storage system is always charged as much as possible and therefore as much energy as possible is used locally.

However, the injection and demand peaks are still present. In this case, the reduction of the injection power is about 4 %. The consumption peak is not reduced at all. Additionally, this scenario represents the worst possible case because the system expresses a very high power gradient. This is caused by the ESS being fully charged during the injection peak. It becomes clear, that this simple behavior is not helpful from the grid operator's perspective.

The only way to avoid this with such a coordination strategy would be to dimension the storage system big enough, such that it can save all the produced energy. However, that is usually not economically viable. It would imply that the system is not charged to its full capacity during most of the year, except for some very sunny days. Thus, such a configuration imposes the additional expense of a larger storage system on the customer, with little to no benefit from their perspective. It is therefore to be expected that many private storage systems will not be able to store all of the expected injection energy,



(a) Single household residual demand with uncoordinated storage capability



(b) Single household storage behavior, including storage power and state of charge. Maximum (dis)charge power: 30 kW; Maximum Energy Capacity: 10 kWh

Figure 4.6: Single household with uncoordinated storage scenario: simulation output

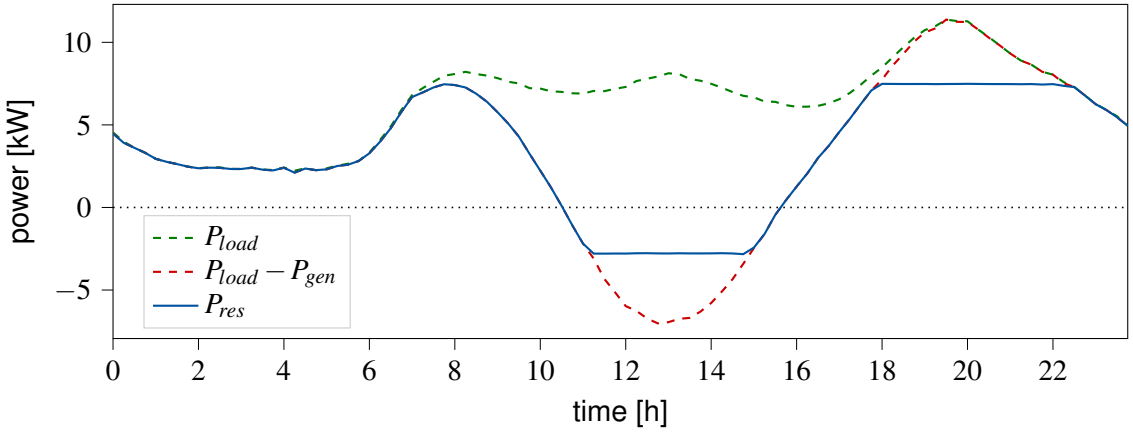
especially on very productive days.

Peak Shaving Scenario

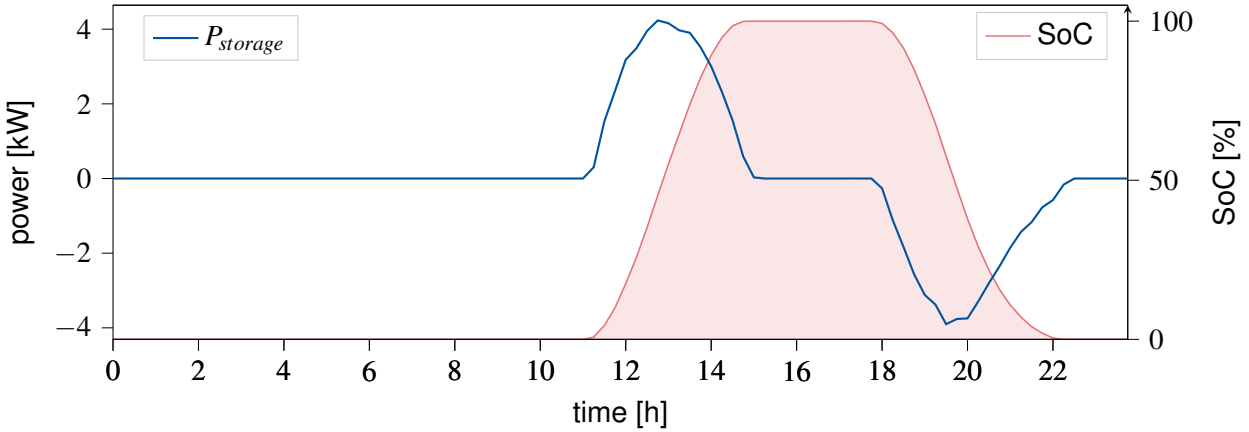
The previous scenario shows the importance of structural flexibility. That is the infrastructure and ability to efficiently coordinate physical flexibility, e.g. an ESS. Therefore, the scenario was expanded by additional simulators, which provide an MPC-based optimization for the flexibility usage. The layout is now equivalent to one complete instance of the smart home grid and HEMS simulation scenario as displayed on the left side of Figure 3.16. The predictor simulators generate the expected system behavior and the optimizer decides which household set points will ensure an optimal usage of the flexibility.

In Figure 4.7, the result of this simulation scenario can be observed. As can be seen from the SoC in Figure 4.7b, the storage system is still used to its full potential. At the same time, Figure 4.7a shows how the optimization adjusts the timing and amount of discharged energy such that the consumption and injection peaks are flattened. Overall, the consumption behavior is lowered by 38 %. The absolute value of the injection peak is smaller to begin with. Therefore, the relative reduction using the same amount of energy is even higher. In this csase, it was reduced by roughly 60 %.

Interesting to look at is also the ESS's power profiles in Figure 4.6b and Figure 4.7b. It is visible how the system follows the injection curve as long as possible in the uncoordinated case. During the optimized charging though, the storage system stays idle until a certain point. Then the power profile follows a cut-off version of the injection peak in order to produce the bounded behavior of the grid demand seen in Figure 4.7a. Similar behavior can be observed at the consumption peak.



(a) Single household residual demand with peak-shaving storage coordination



(b) Single household storage behavior, including storage power and state of charge. Maximum (dis)charge power: 30 kW; Maximum Energy Capacity: 10 kWh

Figure 4.7: Single household with coordinated storage scenario: simulation output

4.1.3 Neighborhood

After setting up a single smart home co-simulation, the next step is the analysis of multiple households in conjunction. For that reason, all the simulation models used in the single household scenarios are instantiated 20 times. Each instance is then connected according to the data flow scenario in Figure 3.16, with the respective instances inside the other simulators. This creates 20 individual processing chains, which each simulate one individual household. Their only connection is realized via the distribution grid and global flexibility control models. The grid is schematically shown in Figure 4.8. The relevant parameter for the analysis of the neighborhood's behavior is the grid residual demand P_{res}^{grid}. For the individual households, all previously discussed simulation values are still available.

For all scenarios that include storage systems, the goal is to analyze inter-household flexibility coordination. Therefore, 10 households were designed with no storage capabilities $(E_{max} = 0\,\text{kWh})$. These are called the inflexible participants. The remaining 10 households are the flexible participants. They were given double of the in Table 4.1 stated maximum capacity, so $E_{max} = 20\,\text{kWh}$. This way, the overall storage capacity of the neighborhood stays proportional to the scaling of the neighborhood size $(20 \times 10\,\text{kWh} = 10 \times 20\,\text{kWh})$. However, the heterogeneous nature of the individual prosumers forces the system to rely on grid-scale coordination for optimal flexibility usage, instead of only local control.

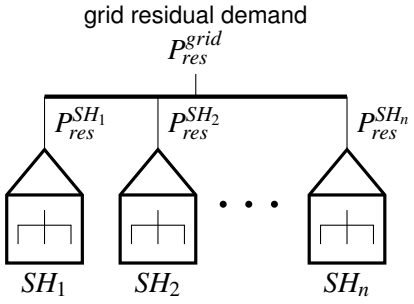


Figure 4.8: Neighborhood grid layout, including observed system parameters

No Flexibility Scenario

For reference, the neighborhood behavior for all 20 households without any physical flexibility is plotted in Figure 4.9. This simulation result looks mostly like its single household counter part shown in Figure 4.5 scaled up by a factor of 20. As expected though, the neighborhood's profile looks smoother and more like the average load curve shown in Figure 4.2. This results from the uncorrelated Gaussian noise, which is added to the individual household simulations during the time series simulation. Equation (4.2) shows

how the standard deviation of the accumulated noise behaves. Opposed to the mean value, which is multiplied by the number of accumulated values, the standard deviation only increases with the square root of that factor. For 20 households this leads to an increase in signal-to-noise ratio by a factor of $\sqrt{20} \approx 4.47$ (see (4.3)). The more households are accumulated, the better the SNR will get. However, this approach assumes an underlying average across the different participants and noise with a zero mean value. Potentially different average system behaviors are blended together and non-zero mean noise is also accumulated.

$$P_{res} = \bar{P}_{res} + N(0, \sigma^2) \tag{4.1}$$

$$\begin{aligned} P_{res}^{glob} &= 20 P_{res} \\ &= 20\bar{P}_{res} + N(0, [\sqrt{20}\sigma]^2) \\ &\approx 20\bar{P}_{res} + N(0, [4.47\sigma]^2) \end{aligned} \tag{4.2}$$

$$\begin{aligned} \varepsilon_{SNR} &= \frac{SNR_{neighborhood}}{SNR_{single}} \\ &= \frac{20\bar{P}_{res}}{\sqrt{20}\sigma} \times \frac{\sigma}{\bar{P}_{res}} \end{aligned}$$

$$\varepsilon_{SNR} = \sqrt{20} \approx 4.47 \tag{4.3}$$

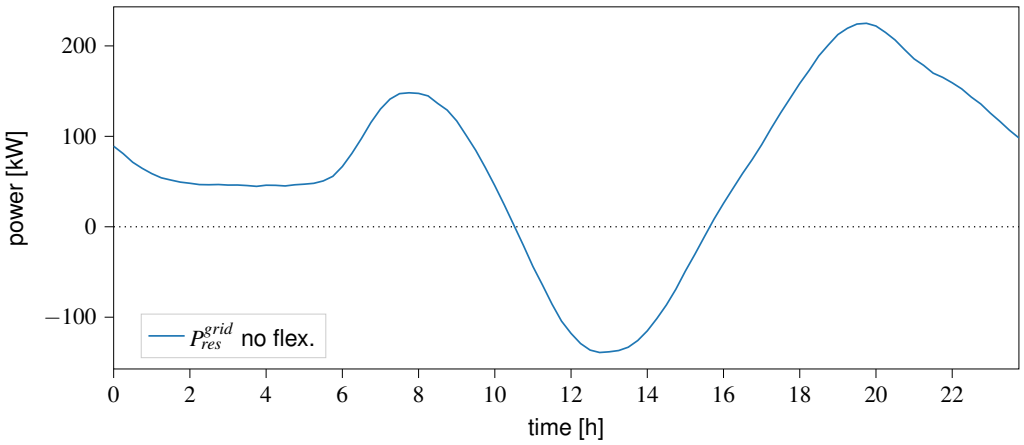


Figure 4.9: Neighborhood residual demand curve in a no flexibility scenario

Local Control Only

Another point of reference is the neighborhood behavior without any global flexibility coordination. Each flexible prosumer in that case would express the behavior of the local peak shaving scenario as described in section 4.1.2 and visualized in Figure 4.7a. The inflexible participants produce the results shown in Figure 4.2.

The overall neighborhood behavior can be seen in Figure 4.10. It resembles, as expected, a curve similar to the neighborhood scenario without storage. However, the consumption and injection peaks are flattened somewhat because half of the households are performing the local peak shaving optimization while the other half has no possibility to do so. This behavior reduces the injection peak by 47 % and the consumption peak by 23 % compared to the no storage scenario.

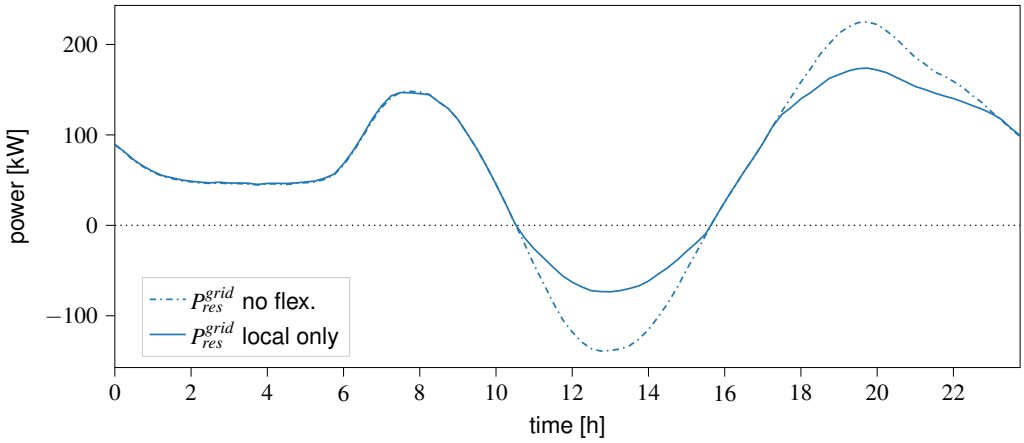
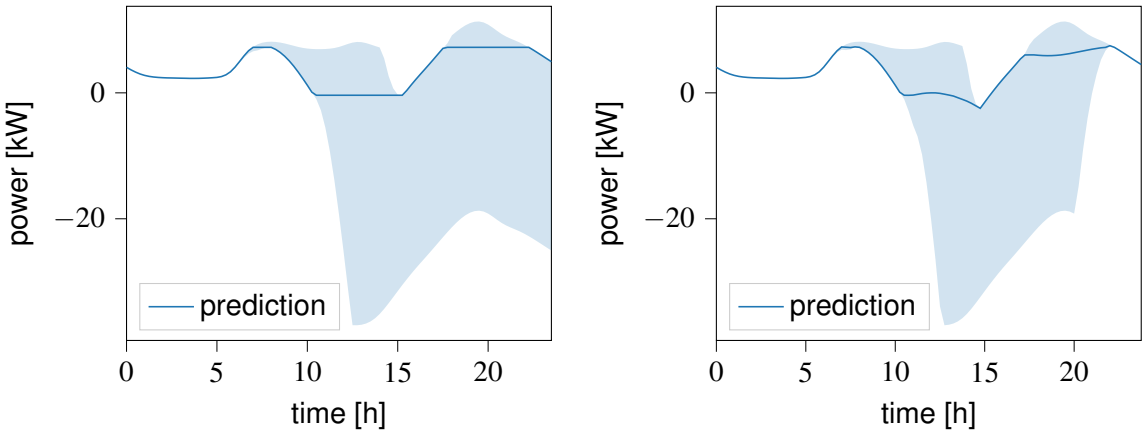


Figure 4.10: Neighborhood residual demand curves in a no flexibility and local control scenario

Coordinated Control

In order to achieve an optimal flexibility coordination on the neighborhood's distribution grid level, the global flexibility coordination model has to be included into the simulation. This model acts as an optimizer in a neighborhood-wide MPC flexibility coordination scheme. Similar to the flexibility optimization in the single household scenario, this algorithm performs global consumption and injection peak shaving. The modeled future behavior of the to-be-optimized cell is generated by the individual households themselves. Their individual predictions are accumulated to find the overall expected system behavior.

An exemplary prediction at the beginning of the simulation can be seen in Figure 4.11a. It includes the planned system behavior and the possible flexibility margins for the next 24 h. However, this prediction is newly generated during every simulation step and therefore changes its shape when the system behavior is influenced by the global control system. This effect is shown in Figure 4.11b. These plots look quite similar, but the latter displays the prediction that was made on each time step for the upcoming step. Therefore, it shows how the local control system is disturbed by the flexibility set points from the global controller.



(a) Initial household prediction ($t = 0$)

(b) Final household predictions ($t = 24\,\text{h}$)

Figure 4.11: Household prediction curves including flexibility margins

Also visualized in Figure 4.11 are the prosumer's flexibility constraints. Between 0 h and 7 h there is no flexibility because the ESS is empty and cannot provide negative flexibility. The positive flexibility is limited by the amount of generated power, of which there is none at night. For the same reason, the absolute maximum of energy demand during the whole day is always the household load curve, which can be seen in the profile of the positive flexibility margin.

Another constraint comes into play around 15 h, when the storage system reaches its full capacity. Then the system cannot offer positive flexibility anymore. In terms of negative flexibility, the SoC is the only constraining factor. As soon as the system starts charging the negative flexibility increases to the maximum value given by the storage system's maximum power ratings. It then follows the same profile as seen in a static household demand because a constant flexibility margin is applied in addition to the original household behavior.

The result of the global optimization can be seen in Figure 4.12. It achieves a global peak shaving by flattening the remaining peak using the available flexibility. Instead of finding the local optimum, the flexible prosumers are controlled so that they will compensate the peaks of the inflexible participants. This achieves an injection peak reduction of 56 % which is an improvement of 9 % over the local only scenario. The consumption peak is reduced by 32 % which is an improvement of 9 % as well.

In Figure 4.13a, the coordinated behavior of a single flexible household is displayed (P_{res}). For comparison, the previously discussed locally optimized behavior is also displayed $(P_{res,local})$, along with the no flexibility case $(P_{load} - P_{gen})$. These references show how the global control restricts the storage system from charging before and after the peak, but

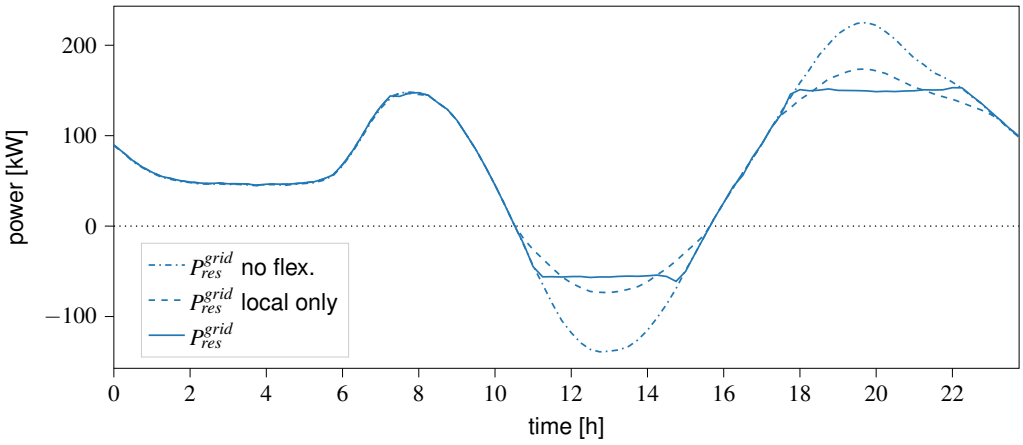


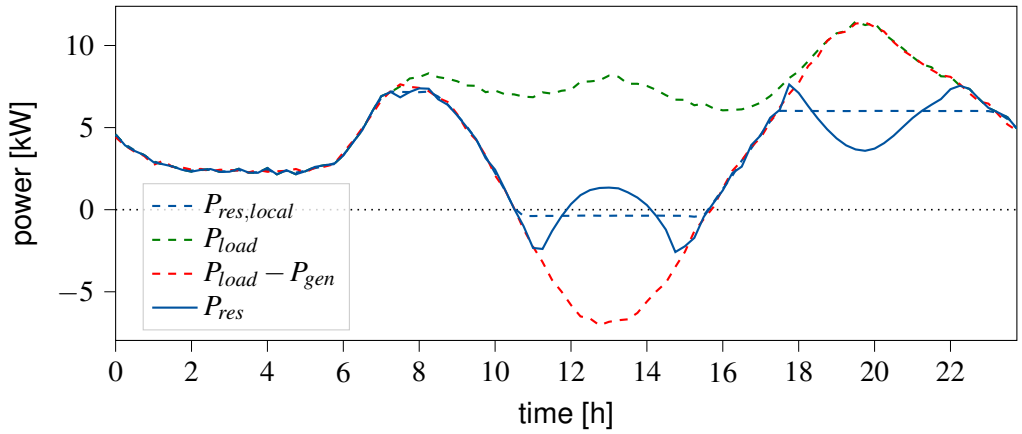
Figure 4.12: Neighborhood residual demand curves in a coordinated control scenario

makes up that deficit by increasing the charging power in between. That creates an inverse peak in all controllable prosumers, which negates the peaks of static consumers.

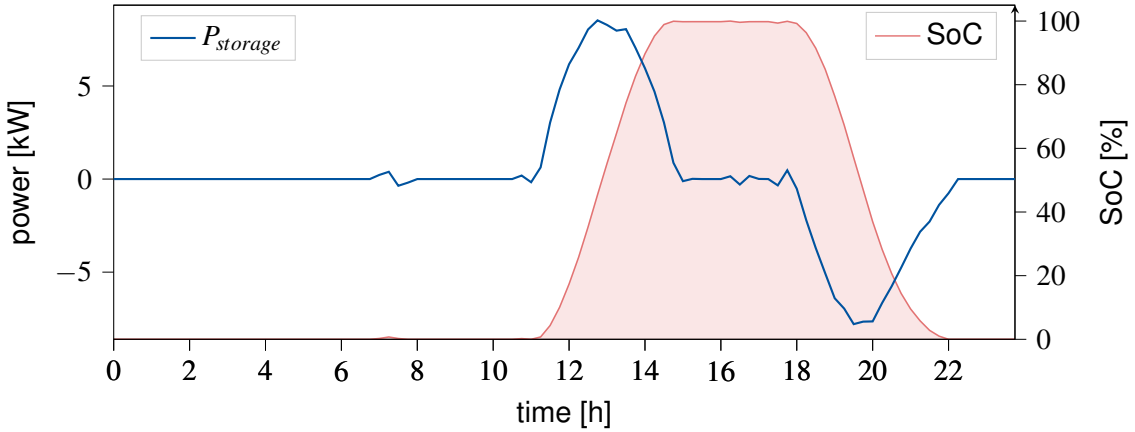
The storage system behavior in Figure 4.13b confirms that the ESS is still used to its full potential, even with the global control system in place. This stems from the fact that the goals peak shaving and local energy usage are complementary. The more energy is stored, the lower the injection peak will be and the more energy will be available to reduce the demand peak. It therefore makes sense for this optimization scheme to use all of the storage system's capacity. The important benefit of the control system is its ability to mediate between prosumer capabilities and traditional consumers.

4.2 Robustness and Problems

All scenarios discussed in section 4.1 are designed to be realistic to a certain degree by using established standard household profiles etc. Nevertheless, for the purpose of demonstrating the basic functionality of the given control scheme, they represent an idealized version of reality. There are several problems in realistic deployment, which have not been adressed. The most significant issue is the fact that the MPC scheme relies on accurate predictions. For generation processes like PV systems this is oftentimes feasible, because the installed system stays fixed and the environmental influences can be predicted via weather forecasts. The load prediction is a bigger problem, since it is highly reliant on individual human behavior, which can oftentimes be hard to predict or change spontaneously. In this section, an attempt will be made to show how the control system performs under less ideal circumstances.



(a) Single household residual demand with global peak-shaving storage coordination



(b) Single household storage behavior, including storage power and state of charge. Maximum (dis)charge power: 30 kW; Maximum Energy Capacity: 20 kWh

Figure 4.13: Single household, global coordination scenario: simulation output

4.2.1 High Noise Level

The first analysis is in case of high noise conditions. This scenario models the case that the average prediction is accurate, but the individual behavior varies strongly from the average scenario for each time step. For consumption, this might represent residents using certain appliances sporadically. A resident might, for example, use the oven to cook their dinner for 20 minutes in the evening. This contributes to the average consumption peak around that time, but it also creates a spike in that household's consumption curve on that specific occasion. For a PV system the noise might model a loose cloud cover, which causes the system fluctuate between direct sunlight and cloud shadow. Both consumption and generation noise is herein approximated by simple uncorrelated Gaussian noise with a standard deviation of $\sigma = 1\,\mathrm{kW}$, so one order of magnitude higher as in the previous examples.

The results of the noisy simulation for a single household can be seen in Figure 4.14. It is clearly visible that during the times of no flexibility, e.g. at night the noise significantly influences the residual demand. During the peak times, however, the controller manages to keep the demand relatively flat. This shows that the control scheme performs reasonably well under noisy conditions, as long as the predictions match the average trend of the system. The reduction of the injection peak here lies at 59 % and in consumption peak at 40 %. These values are almost identical to the less noisy scenario.

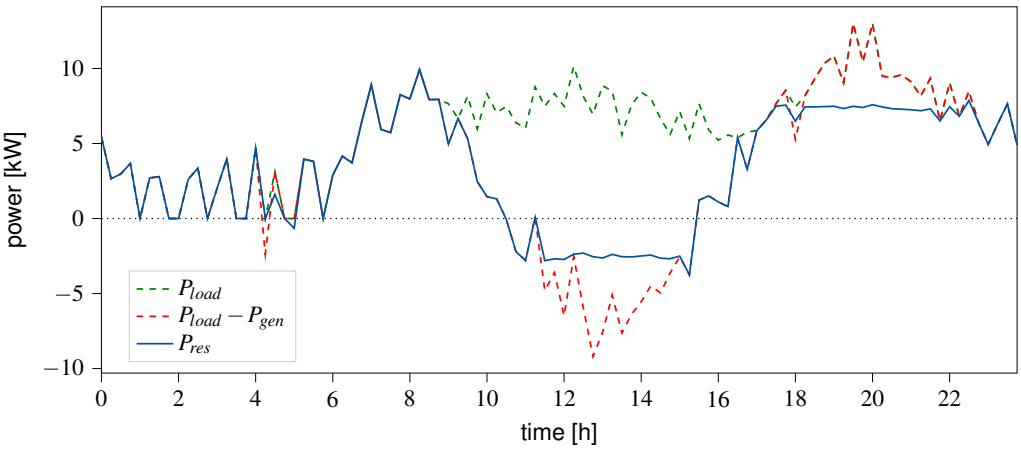


Figure 4.14: Single household peak shaving in high-noise scenario

The coordinated neighborhood simulation is shown in Figure 4.15. Here, an even smoother behavior can be observed. This underlines the averaging effect of summing multiple households as discussed in (4.2) and (4.3). The noise also increases by a factor of 10 for the neighborhood, however, its signal-to-noise ratio is about a factor of 4.47 better than that of the single household to begin with. Therefore, the accumulated neighborhood behavior is closer to the averaged prediction curve, making the prediction more robust than in the single household scenario. The peak reduction here amounts to 51 % and 27 % for injection and consumption respectively. This is a performance decrease of 5 % in both cases due the higher noise level.

4.2.2 Time Shift

Another problem might occur if the prediction is time shifted relative to the actual system behavior. Reasons for such a shift could be as simple as the prediction ignoring daylight savings time. Customers might also shift their behavior or habits, e.g. they sometimes prepare dinner at 7 p.m. and other times at 8 p.m. Alternatively, the PV system might be under cloud cover during the predicted peak, shifting the actual peak to another time with

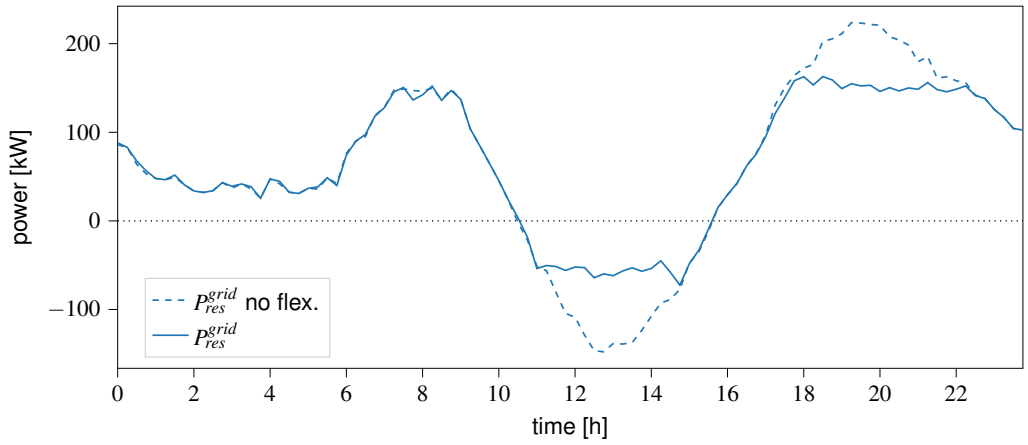


Figure 4.15: Multiple household peak shaving in high-noise scenario

direct sunlight. The prediction error was modeled by a time offset of 1 h of the household consumption and generation.

The results of both the single household case (Figure 4.16) and the coordinated neighborhood case (Figure 4.17) show much more significant peaks in injection and consumption than the idealized cases. The peaks are only reduced by about 10 % and 17 % respectively which is the worst performance out of all the peak shaving scenarios.

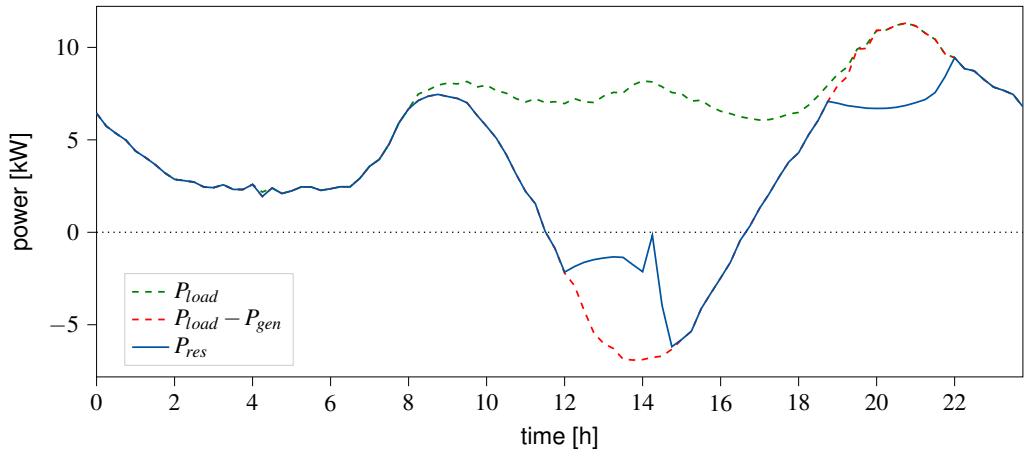


Figure 4.16: Single household peak shaving in time-shifted scenario

This reveals a weakness of this control scheme: its dependence on accurate predictions. In case of noise it can cope because the underlying average is still correct. When the prediction is shifted however, the average values for each time step are incorrect. Furthermore, if the shift is correlated between the households, e.g. all are shifted in the same direction, the average over those households will also be shifted. For design reasons, the global flexibility controller uses pure predictions and no current values for the flexibility coordination, which leads to a bigger discrepancy in the neighborhood coordination (see

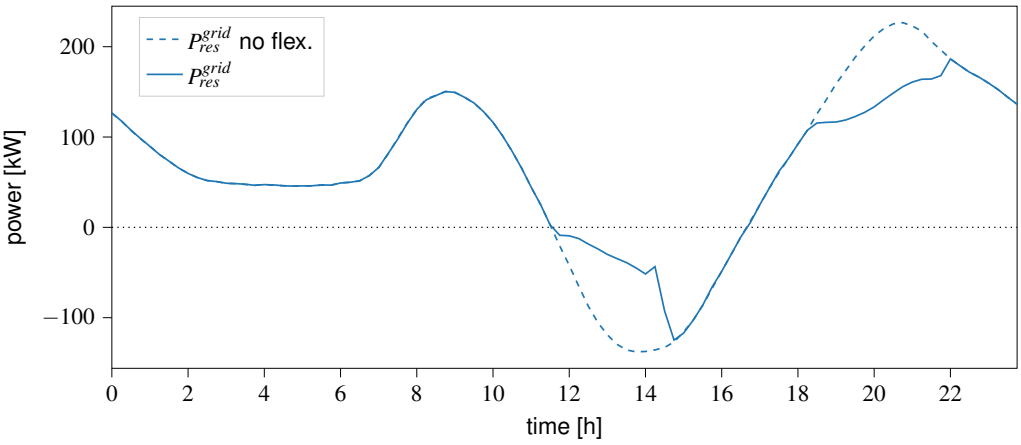


Figure 4.17: Multiple household peak shaving in time-shifted scenario

subsection 4.2.3).

4.2.3 Other Known Challenges

The biggest known problem in the current simulation setup is the issue of circular dependencies and simulation step size. Certain simulators, namely the local and global flexibility optimizer, depend on each other's results. The global optimizer needs to know the predicted behavior and the flexibility margins of the local one, while the latter needs to know the flexibility request of the global optimizer in order to incorporate it into the set point for the smart home coordinator. Since none of the simulators is executed twice, the dependency in the current setup is resolved by time shifting one of the data connections. However, no matter which simulator is executed first, the result is always that the global controller can never influence the same time step that its prediction data stems from. This means the global optimizer has no access to the actual output behavior for the time step it is optimizing and only predicted data is available. Naturally, a delay between local and superordinate control is actually a realistic scenario, but that should always be a choice within the simulation scenario and not forced by the environment.

Under normal circumstances, the circular connection could simply be split into two sequential connections, but in this case it is not as easy. Since multiple households provide their data to a single global entity, all these connections need to be split and the global controller needs information on which input entity corresponds to which output entity. That usually breaks the clean interfaces because it introduces the task of data exchange and part of the simulation topology into the simulators instead of letting Mosaik handle the data exchange.

Similar problems arise any time handshake procedures of certain communication protocols or iterative algorithms across multiple models are considered. Any time data is

exchanged back and forth between entities in order to find an actual physical outcome, there are currently two choices: Find a workaround that does not have circular dependencies or use a small execution step size to approximate the continuous or iterative behavior. The latter alternative might not be possible without significant increase in computation time and the former might not lead to accurate results. An important part of future work in this project will be to solve this problem for the benefit of more complex simulation setups.

4.3 Grafana Interface

As mentioned in section 3.1.3, a graphical front-end was introduced into the simulation infrastructure, in order to enable users to observe the simulation outputs during execution. This is a convenient feature especially for debugging the simulated models. During the development of a multi-domain system such as this flexibility control scenario, many individual parts need to be implemented and design decision need to be taken. It is therefore helpful to have a quick way to look how changes and additions influence the simulation behavior. Moreover, being able to view the output at runtime saves development time, because developers can see significant bugs or other problems immediately, stop the execution and fix the problem. Otherwise they might have to wait for a long and complex simulation to finish, only to find out that the results are unusable.

The default dashboard can be seen in Figure 4.18. At the top of the dashboard, a progress bar shows the current simulation progress, which is provided by Mosaik as additional meta information. The two bottom panels are used to observe a single household (left) and the whole neighborhood (right). For the single household the residual demand is plotted in green while the underlying load reference is plotted in blue. The orange plot is the households residual reactive power which is always at 0 in this scenario. The neighborhood plot shows the residual grid demand and reactive power accordingly. In the center there are two panels which show the outputs of the flexibility optimization simulators. On the left, the local controller is shown with its optimized prediction and flexibility margins. These values are forwarded by all households to the coordinating flexibility controller which is shown on the right. Here, the combined predictions are visualized alongside the flexibility requests and the resulting optimized neighborhood behavior. Finally, the predicted local and global SoC is shown as an outline in the respective panels.

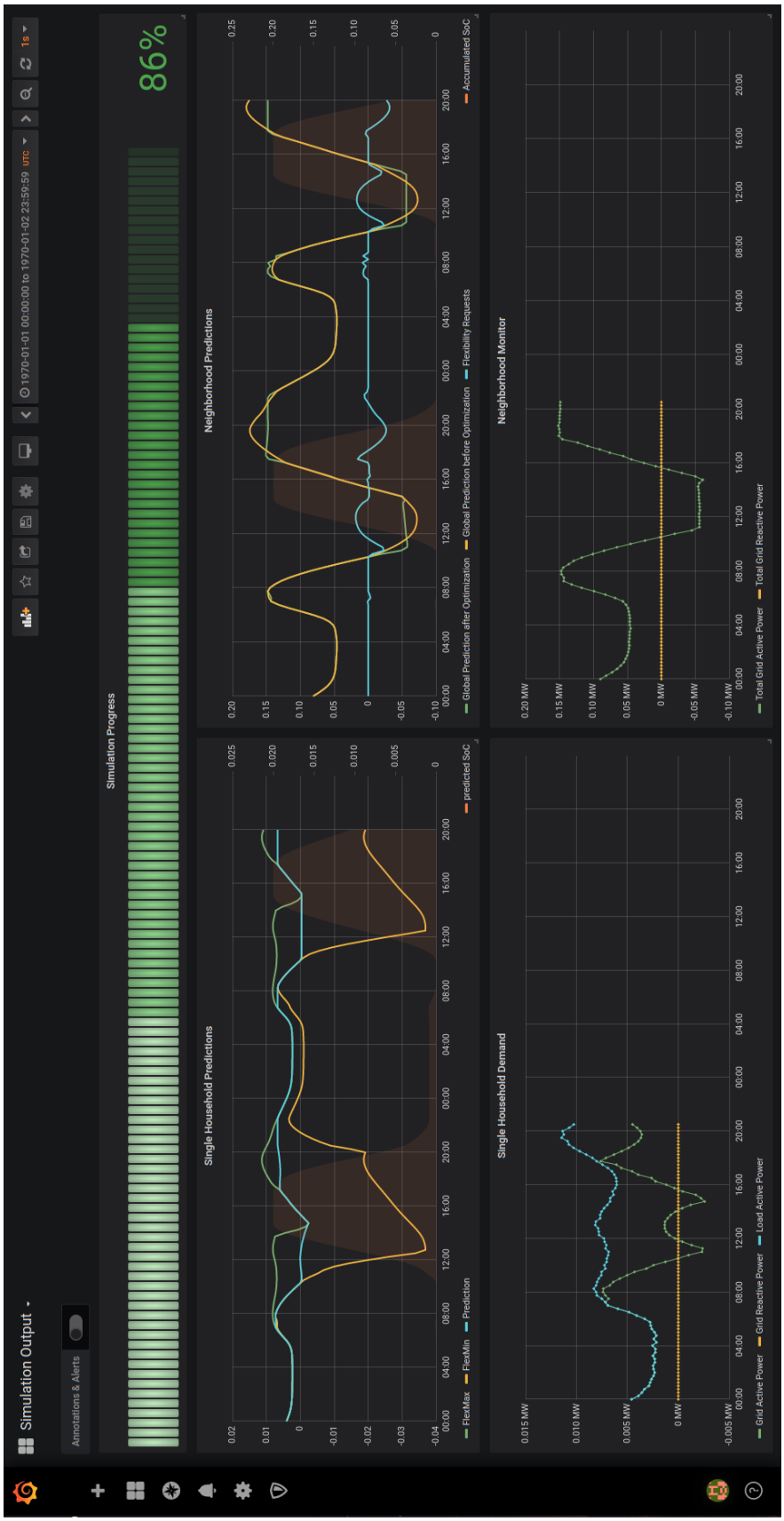


Figure 4.18: Default Grafana dashboard layout

5 Conclusion and Outlook on Future Work

In this chapter, the overall results of the thesis are summarized and analyzed in reference to the original goals. Afterwards a quick outlook is given on potential future developments for the simulation environment and the flexibility control scheme.

5.1 Conclusion

The goal of this thesis is the development of a scalable, easily usable co-simulation environment in the context of smart grid simulation and flexibility coordination. Furthermore, it should involve the testing and analysis of a distributed hierarchical coordination scheme for privately deployed flexibility sources.

The development of the final simulation was split into two parts, the development of the environment and the design of the simulation scenario within it. The environment uses the Mosaik co-simulation framework, which provides well-defined interfaces between the simulators as well as simulator scheduling capabilities. To provide individual runtime environments and deployment variability for each simulator the Docker containerization software was chosen. This tool packages individual simulators and the Mosaik Core application into separate containers, which can be freely instantiated and deployed across a variety of systems without the risk of dependency issues. The simulator container deployment was automated and incorporated into an extension of the Mosaik Scenario-API.

For the flexibility simulation scenario, several simulators needed to be implemented, to realize the individual grid and control components. Firstly, a smart home simulation with load, PV generation and ESS elements was designed. This was augmented by an MPC-based flexibility optimization HEMS, which uses the given flexibility to reduce injection and consumption power peaks. Finally, the simulation was scaled up to 20 households with varying flexibility. A superordinate controller was applied over all these grid participants in order to achieve inter-household flexibility coordination and realize cumulative peak reduction.

The exemplary results show on the one hand, that the environment works as expected and can handle a variety of simulation scenarios. Since all simulators implement the

common interface and are deployed automatically, the generation of the resulting observations for different setups and conditions is straightforward. The scenario script serves as the central simulation definition, where all necessary parameters can be adjusted. The database is the central repository from which the result data can be retrieved afterwards. Even runtime simulation monitoring is possible via the provided visualization server.

Concerning the exemplary flexibility coordination simulation, the results show how smart flexibility control instead of uncoordinated consumption optimization can decrease the danger of grid congestion. The implemented MPC approach not only lowers the absolute energy demand, but also the maximum power drawn from the grid. Additionally, it shows how the coordination of flexible and static households in conjunction can lead to a grid wide flexibility optimization, which single households could not achieve without outside information.

5.2 Outlook

Firstly, the developed co-simulation environment still offers many possibilities for improvement and expansion. In the current setup for example, the given scenario script still involves the user calling similar API initialization and connection functions for all the involved simulators and models. The next step in simulation automation would be to create an automated scenario implementation, which reads in the scenario from a configuration file, object or database. In such a case the user effort would be reduced to providing only the basic scenario data with as little redundancy as possible.

Another important feature is the distributed deployment of simulators. Through the design of the simulation as a containerized microservice infrastructure, the groundwork has been laid for the environment to interact with clusters and cloud computing services. However, a proper support still needs to be implemented and tested, before this kind of deployment is possible.

Additionally, it might be worth investigating if the Mosaik scheduling service could be altered to bring forth the underlying event-driven simpy scheduling as an additional option. In many cases, the currently provided variable time step execution might be sufficient, but the system loses the flexibility of employing a true event-driven scheme, if necessary. Even other hybrid schemes might be thinkable, where the normal schedule follows the known strategy, but certain events can be triggered that cause the execution of other simulators on the fly.

Lastly, for simple user comfort, the environment could be extended by a simple web-service, which can react to callbacks from the Grafana visualization interface. This would

enable users to actually interact with the simulation through this graphical interface, instead of passively watching the simulation progess. Basic examples are buttons to start, stop and pause the simulation or a file dialogue, which allows the users to load a new simulation scenario into the environment.

On the topic of flexibility coordination, the simulations have show that an hierarchical MPC scheme can be useful. A more elaborate optimization can increase the robustness of this scheme towards unexpected disturbances such as the prediction time shift.

In order to give even more reliable results, the simulation should also be increased in complexity at several points. A more dynamic load curve and a PV simulation dependent on underlying environmental factors would create more realistic and heterogeneous household behavior. Additionally, the incorporation of grid elements like transmission lines and transformers would increase the accuracy of the grid simulation.

Finally, there are also other flexibility coordination strategies in literature, like price signal based approaches, which incentivise prosumers to adjust their demand depending on a form of local energy market. These and other strategies need to be tested under similar simulation conditions in order get comparable results.

Bibliography

[AKR19] Alireza Akrami, Meysam Doostizadeh, and Farrokh Aminifar. “Power system flexibility: an overview of emergence to evolution”. In: *Journal of Modern Power Systems and Clean Energy* 7.5 (May 2019), pp. 987–1007.

[ARN11] Michele Arnold and Göran Andersson. “Model predictive control of energy storage including uncertain forecasts”. In: *Power Systems Computation Conference (PSCC), Stockholm, Sweden*. Vol. 23. Citeseer. 2011, pp. 24–29.

[BDE17] BDEW. *Standardlastprofile*. `https://www.bdew.de/energie/standardlastprofile-strom/`. Jan. 2017.

[BIA15] D. Bian et al. “Real-time co-simulation platform using OPAL-RT and OPNET for analyzing smart grid performance”. In: *2015 IEEE Power & Energy Society General Meeting*. IEEE, July 2015.

[BUN17] Bundesnetzagentur. *Flexibilität im Stromversorgungssystem. Bestandsaufnahme, Hemmnisse und Ansätze zur verbesserten Erschließung von Flexibilität. Diskussionspapier*. Apr. 2017.

[CAM04] Eduardo F. Camacho and Carlos Bordons. *Model Predictive Control*. Springer-Verlag GmbH, June 1, 2004. ISBN: 1852336943.

[CEN14] CEN-CENELEC-ETSI Smart Grid CoordinationGroup. *SG-CG/M490/L Flexibility Management – Overview of the main concepts of flexibility management – Version 3.0*. Nov. 2014.

[DAL17] Emiliano Dall’Anese, Pierluigi Mancarella, and Antonello Monti. “Unlocking Flexibility: Integrated Optimization and Control of Multienergy Systems”. In: *IEEE Power and Energy Magazine* 15.1 (Jan. 2017), pp. 43–52.

[DEN15] Paul Denholm et al. *Overgeneration from Solar Energy in California: A Field Guide to the Duck Chart*. Tech. rep. National Renewable Energy Laboratory, Nov. 2015.

[DOC20a] Docker Inc. *Docker*. `https://www.docker.com/`. 2020.

[DOC20b] Docker Inc. *Docker SDK for Python*. `https://docker-py.readthedocs.io/en/4.2.0/`. 2020.

[DRA16] Nicola Dragoni et al. “Microservices: yesterday, today, and tomorrow”. In: (June 13, 2016).

[EDE14] Ottmar Edenhofer. *Climate change 2014 : mitigation of climate change : Working Group III contribution to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*. New York, NY: Cambridge University Press, 2014. ISBN: 978-1-107-05821-7.

[FAR10] H. Farhangi. “The path of the smart grid”. In: *IEEE Power and Energy Magazine* 8.1 (Jan. 2010), pp. 18–28.

[FUJ99] Fujimoto. *Parallel and Distributed Simulation*. John Wiley & Sons, Dec. 20, 1999. 324 pp. ISBN: 0471183830.

[GOD10] Tim Godfrey et al. “Modeling Smart Grid Applications with Co-Simulation”. In: *2010 First IEEE International Conference on Smart Grid Communications*. IEEE, Oct. 2010, pp. 291–296.

[GOM17] Cláudio Gomes et al. “Co-simulation: State of the art”. In: *Computing Research Repository* (Feb. 2017).

[INT15] International Energy Agency (IEA). *Technology Roadmap How2Guide for Smart Grids in Distribution Networks: Roadmap Development and Implementation*. OECD Publishing, 2015.

[MAU17] I. Mauser et al. “Definition, Modeling, and Communication of Flexibility in Smart Buildings and Smart Grid”. In: *International ETG Congress 2017*. 2017, pp. 1–6.

[MOM09] James A. Momoh. “Smart grid design for efficient and flexible power networks operation and control”. In: *2009 IEEE/PES Power Systems Conference and Exposition*. IEEE, Mar. 2009.

[NEU15] Judith Neugebauer, Oliver Kramer, and Michael Sonnenschein. “Classification Cascades of Overlapping Feature Ensembles for Energy Time Series Data”. In: *Data Analytics for Renewable Energy Integration*. Springer International Publishing, 2015, pp. 76–93.

[OFF11] OFFIS Oldenburg. *Mosaik*. `https://mosaik.offis.de/`. Oldenburg, 2011.

[SMI05] Jim Smith. *Virtual Machines*. Elsevier LTD, Oxford, June 1, 2005.

[STE18] C. Steinbrink et al. “Smart grid co-simulation with MOSAIK and HLA: a comparison study”. In: *Computer Science - Research and Development* 33.1-2 (Sept. 2018), pp. 135–143.

[STR17] Thomas Strasser et al. “Towards holistic power distribution system validation and testing—an overview and discussion of different possibilities”. In: *e & i Elektrotechnik und Informationstechnik* 134.1 (Dec. 2017), pp. 71–77.

[THÖ15] J. Thönes. “Microservices”. In: *IEEE Software* 32.1 (2015), pp. 116–116.

[TRO16] Gustavo O. Troiano et al. “Co-simulator of power and communication networks using OpenDSS and OMNeT++”. In: *2016 IEEE Innovative Smart Grid Technologies - Asia (ISGT-Asia)*. IEEE, Nov. 2016.

[VAN14] Robert Vanderbei. *Linear programming : foundations and extensions*. New York: Springer, 2014.

[WIN07] Philip Winslow et al. *Desktop Virtualization Comes Of Age*. Tech. rep. CreditSuisse, Nov. 2007.

[YU07] Yang Yu. “OS-level Virtualization and Its Applications”. PhD thesis. Stony Brook University, Dec. 2007.