CARL VON OSSIETZKY universität OLDENBURG

Faculty II – Computing Science, Business Administration, Economics and Law
Department of Computing Science

Simulation Model Composition for the Large-Scale Analysis of Smart Grid Control Mechanisms

Thesis to obtain the degree of
Doctor of Engineering

by

M.Sc. Steffen Schütte

Reviewers:

Prof. Dr. Michael Sonnenschein
Jun.-Prof. Dr. Sebastian Lehnhoff
Prof. Dr.-Ing. Axel Hahn

Public defense on: 27th November 2013

Steffen Schütte
Kreisstraße 27
D-31655 Stadthagen
Germany

E-Mail: mail@steffenschuette.de

Zusammenfassung

Das intelligente Stromnetz der Zukunft (engl. Smart Grid) muss eine Vielzahl verschiedener Akteure integrieren und koordinieren, wie z.B. dezentrale erneuerbare Erzeuger, Speicher, traditionelle Lasten oder flexible (steuerbare) Lasten. Neuartige Kontrollmechanismen müssen entwickelt werden, die den Zustand aller Akteure analysieren und entsprechende Maßnahmen zur Stabilisierung des Stromnetzes ausführen können. Aus Kosten- und Sicherheitsgründen können solche Kontrollmechanismen üblicherweise nicht in großmaßstäblichen Feldversuchen getestet werden. Modellierung und Simulation sind hier geeignete Konzepte, um Kontrollmechanismen im großen Maßstab zu testen, bevor der Einsatz im echten Stromnetz erfolgen kann. Eine Vielzahl von Simulationswerkzeugen zur Netzzustandsanalyse existiert bereits. Die bestehenden Werkzeuge haben jedoch einige Nachteile. Zum einen gestaltet sich die Kombination von Simulationsmodellen schwierig, die mit verschiedenen Werkzeugen erstellt wurden. Dies ist jedoch häufig gewünscht, da gerade in Forschungsprojekten verschiedene Projektpartner zusammenarbeiten und diese oft bestehende Modelle aus ihrem Forschungsgebiet in das Projekt mit einbringen. Zum anderen ist die Spezifikation von großmaßstäblichen Szenarien aufwändig und mit viel manueller Arbeit verbunden, da diese Tausende von Objekten, wie zum Beispiel Netzknoten, Erzeuger oder Verbraucher enthalten.

Das Forschungsziel der vorliegenden Arbeit ist daher die Entwicklung eines Konzeptes zur Erstellung eines simulativen Testbetts für die Analyse von Smart Grid Kontrollmechanismen unter Verwendung bestehender Simulationsmodelle. Anforderungen an das zu erstellende Konzept werden durch die Analyse von Simulationsmodellen und Smart Grid Szenarien aus früheren Projekten sowie aus Literaturquellen abgeleitet. Basierend auf einer Literaturrecherche im Bereich der Komponierbarkeit von Simulationsmodellen wird ein mehrschichtiges Konzept namens *mosaik* vorgestellt. Es erfüllt die aufgestellten Anforderungen und erlaubt die Definition valider und großmaßstäblicher Szenarien unter Verwendung bestehender Simulationsmodelle. Dies wird durch die Definition zweier Metamodelle erreicht. Das *Semantische Metamodell* erlaubt die Erstellung formaler Beschreibungen für Simulatoren und ihre Modelle. Darauf aufbauend erlaubt das *Szenario Metamodell* die Beschreibung von Szenarien durch Erfassung statistischer Daten der Szenarioobjekte (z.B. wie viele Elektrofahrzeuge es pro Netzknoten gibt) sowie zusätzlicher Bedingungen (z.B. Elektrofahrzeuge nur an Knoten mit Photovoltaikanlage). Entsprechende Algorithmen, die das so beschriebene Szenariomodell interpretieren und komponieren können, werden vorgestellt.

Eine Implementierung des Konzeptes wird vorgestellt und verwendet, um das Konzept zu evaluieren. Zwei Fallstudien zeigen, dass das Szenariomodellierungskonzept auf großmaßstäbliche Szenarien realer Forschungsprojekte anwendbar ist. Die größte Studie besteht aus einem Mittelspannungsnetz mit 112 untergeordneten Niederspannungsnetzen und einer Gesamtsumme von knapp 12000 Netzknoten. Jedem Knoten sind dabei verschiedene simulierte Erzeuger und Verbraucher zugeordnet, sodass sich eine Gesamtanzahl von über 75000 simulierten Objekten ergibt. Laufzeitmessungen konnten zeigen, dass die Komposition eines solchen Szenarios in wenigen Minuten erfolgen kann und dass der Zeitaufwand linear zur Anzahl der Objektverbindungen ist.

Abstract

The intelligent future power system, often referred to as Smart Grid, has to integrate and coordinate a large number of different actors such as decentralized renewable energy producers, storages and traditional as well as flexible consumers. New control schemes have to be developed for this complex task. These have to analyze the status of all actors and schedule required actions among this ever growing number of resources to keep the power system stable. For economical as well as for safety reasons, large-scale field trials of new control schemes can usually not be carried out. Here, modeling and simulation are considered appropriate means to virtually test these control schemes prior to deployment. For power system analysis a number of simulation tools already exists. However, these tools have two major drawbacks. First, they are not designed to incorporate (reuse) existing simulation models implemented for other tools. But this may be desirable, especially in research projects that involve different project partners, each with its special field of expertise and often existing simulation models. Second, the specification of large-scale scenarios involving thousands of objects, such as nodes, consumers, producers as well as environmental models, requires time-consuming and error-prone manual specification work.

The research presented in this thesis aims to provide a concept to solve these issues. The objective is to develop a concept for building a simulative testbed for the evaluation of new Smart Grid control mechanisms by composing available simulation models. Existing simulation models and Smart Grid scenarios from past research projects as well as literature research are used to derive requirements for the concept. Based on literature research in the general field of simulation model composability, a layered concept called *mosaik* is presented. It claims to meet the established requirements and, in particular, allows to define valid and large-scale scenarios based on existing models without having to specify every object and its connections to other objects manually. This is achieved by two metamodels. The *semantic metamodel* is used to formally specify simulator and simulation model semantics. Based on this, the *scenario metamodel* can be used to specify a scenario model by capturing statistical data about the objects of the scenario (e.g. how many electric vehicles exist per node) and additional constraints (e.g. vehicles only at nodes that have a photovoltaic system) that govern the way in which the scenario is built. Algorithms interpreting the instances of these metamodels (the scenario models) and establish valid compositions are proposed.

An implementation of the mosaik concept is presented and used to evaluate the approach. Two case studies have shown that the scenario modeling approach is applicable to large-scale, real-world research projects. One of the studies involves a medium voltage power grid comprised of 112 low voltage grids that are composed hierarchically and sum up to an overall number of almost 12000 nodes. To each node a number of different simulated consumers and producers are assigned, so that the overall scenario involves more than 75000 simulated objects. Benchmarks of the different phases of the composition procedure have shown that such a scenario can be composed within a few minutes and that time scales linearly with the number of connections to make.

Contents

Part I

Setting and Foundations

“The most serious mistakes are not being made as a result of wrong answers. The truly dangerous thing is asking the wrong question” – Peter Drucker

1 Introduction

Traditionally, the electric infrastructure was dominated by and designed for electricity generation by large, centralized fossil or nuclear based power plants. The generated electricity is transported from these plants to the customers across different voltage levels. Figure 1.1 shows this traditional setup. On the high voltage level large industrial loads are connected and the medium voltage (MV) and low voltage (LV) levels distribute the energy to smaller business and residential loads.

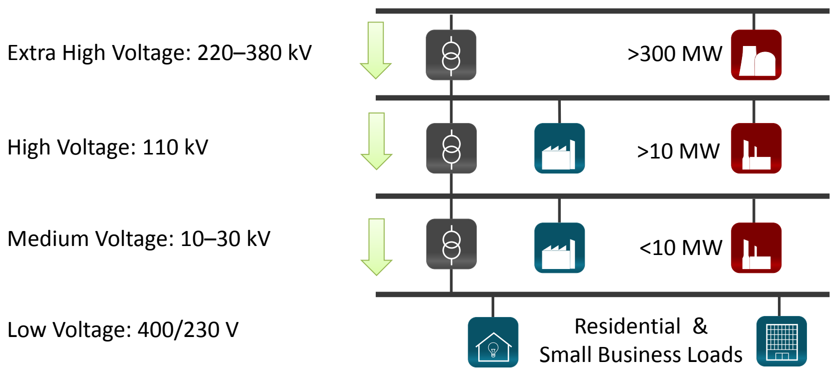


Figure 1.1: Basic topology of the power grid [HDS07]

Nowadays, the advent of more and more distributed generation units is changing this top-down paradigm. Wind-farms, biomass plants and large-scale photovoltaics (PV) are feeding electricity into the MV levels, and many smaller PV systems as well as combined heat and power plants (CHP) and other small-scale electricity sources are generating electricity on the LV level. This change in generation has a number of consequences. For example, power may also flow into the inverse direction (bottom-up) caused by the feed-in on the lower voltage levels. This requires new protection schemes. Also the increased feed-in on the LV level has an impact on the power quality, caused by voltage band violations, for example. Furthermore, the overall balancing of supply and demand in a power grid that is dominated by renewable energy sources is unequally more complicated. The reason for this is a combination of two factors. First, the electricity generated by the majority of renewable power sources directly depends on the weather conditions, causing intermittent feed-in. Second, as the grid itself cannot store electricity, measures have to be taken for bridging the times with little renewable feed-in. In the short term, conventional plants will have to be used to compensate the fluctuating feed-in from renewable sources. However, in the long run these plants shall be replaced by renewable sources completely. To achieve this, new and cheap storing technologies are required and additional flexibility on the demand side has to be exploited to be able to shift loads from low generation to high generation times. The latter is commonly known as Demand-Side-Management (DSM). Currently a lot of effort is put into both approaches. Following the definition of EU-DEEP [ED12] the term DER (distributed energy resources) will be used in the remainder of this thesis to describe distributed generation, storages as well as flexible loads.

To sum up, the traditional electrical infrastructure will have to change into a dynamic system that involves many active components from generation, over the power system equipment on the different voltage levels up to the consumers. However, a simple deployment of such active components is not sufficient. The future power grid, often referred to as Smart Grid, has to be able to collect information about these components and has to manage them in a smart way.

1.1 Smart Grid

Two definitions of the term Smart Grid are frequently used. The U.S. Department of Energy (DoE) defines a SmartGrid as a grid that "uses digital technology to improve reliability, security, and efficiency (both economic and energy) of the electric system from large generation, through the delivery systems to electricity consumers and a growing number of distributed-generation and storage resources" [U.S09]. The Smart Grids European Technology Platform [Eur10b] defines the term as "an electricity network that can intelligently integrate the actions of all users connected to it - generators, consumers and those that do both - in order to efficiently deliver sustainable, economic and secure electricity supplies." Both definitions mention the use of digital technology[1] to achieve a secure and economic energy supply. There are slight differences in that the U.S. definition puts an emphasis on reliability whereas the European definition explicitly mentions sustainability as a factor. This is due to the fact that the U.S. power grid is facing severe stability issues as the number of significant blackouts has been increasing for the past few years [SRK$^+$11]. Further, both definitions underline the need for integrating large numbers of different users and systems on all voltage levels into the management and control of the power grid.

Compared to the well-known and matured elements of the traditional power grid, a number of technologies has to be developed and tested for making the vision of a Smart Grid come true [SRK$^+$11]:

Distributed Energy Sources (DER) Small rated electricity sources based on renewable or conventional energy sources that are typically decentralized and located in close proximity to energy consumers in the distribution grid.

Energy Storage As electric energy cannot be stored directly in high quantities, it must be converted to mechanical or electrochemical energy. Different existing technologies can be used in the power grid to provide load leveling, replace spinning reserves (conventional power plants on standby) or provide ancillary services such as reactive power for voltage regulation.

Power Electronics With an ever increasing number of renewable and alternative energy sources the need for sophisticated power converter systems increases. These must offer a number of different features such as bidirectional and high efficiency power flows, synchronization capabilities, electromagnetic interference filtering, smart-metering or communication interfaces for measuring and controlling the power flow.

[1] Although this is not made explicit in the European definition, an "intelligent integration" mentioned there is not possible without digital communication and computation technologies.

Control, Automation and Monitoring While the power grid in the past was operated in a reactive fashion with only a number of critical tasks performed by human operators, the future Smart Grid will be a highly complex, nonlinear dynamic network, requiring sophisticated sensing and control strategies. In current research different strategies are being developed involving self-healing, self-organizing and self-configuring capabilities [Lue12, HSL13, Kam10]. As pointed out in the next section this thesis focuses on the use of computer simulation as a testbed for the evaluation of such strategies.

Distribution Automation and Protection "Protection deals with the detection and clearance of abnormal system conditions such as faults and overloads" [SRK+11, p.387]. Triggered by the increased number of DER, existing protection schemes can be compromised. New protection schemes have to be developed, tested and will rely on new automation capabilities in the distribution grid for being able to operate in a convenient way without much user interaction.

Communication All control and protection strategies rely on a large number of measurement values that are obtained from the various actors in the power grid. This ranges from simple information about breaker states (on/off) over detailed measurements of the power quality at different points in the grid up to forecasts of DER behavior of the next hours. After processing this information corresponding control actions have to be sent back to the different actors considering their operational constraints. Reliable, bidirectional communication mechanisms must be used for these tasks. Different wire-based and wireless technologies such as PowerLineCommunication (IEC 61334), ZigBee or 4G technologies are available and could be used.

1.2 Smart Grid Simulation

All these new technologies have to be integrated properly to work hand in hand in order to provide a sustainable and reliable power supply by the future Smart Grid. Thorough testing is required before any large-scale deployment can be started. Simulation (see Chapter 2.1) is a method that allows to do this testing on a large-scale without having to make huge investments and, what is even more important, without having to jeopardize the reliability of the real power grid. Furthermore, as the Smart Grid demands long term investments, the evaluation of these technologies is an important task, especially for future scenarios representing the grid as it is assumed to be structured in a few decades. Another argument for simulation is the fact that it offers the ability to compress long time into a shorter period [BCN05]. This is especially relevant for the evaluation of Smart Grid control strategies as the behavior of basically all DER strongly depends on the time of the season, may it be the solar irradiance that is changing the feed-in curve of PVs, a different wind situation for wind-farms or a change in the CHP behavior caused by the changing thermal demand throughout the year.

Figure 1.2 shows the approach to simulation based evaluation of control strategies on a very high level. A new concept for controlling one or more types of DER is developed (top right). Next, a number of Smart Grid scenarios (see Chapter 3.2) are specified as test cases for the control concept.

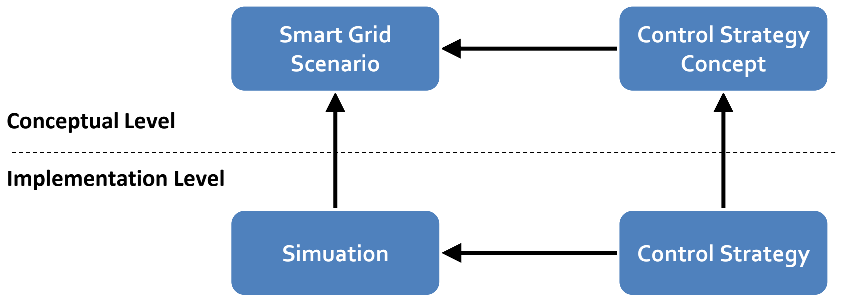


Figure 1.2: Simulative control strategy evaluation

Finally, the concept is implemented and tested against a simulated version of these scenarios.

Although there are tools that aim to cover a broad range of aspects for power system analysis, it is unlikely that a single simulation tool can provide all functionalities and models required for evaluation of Smart Grid control strategies, due to the large number of different use cases, actors and technologies. This assumption is strengthened by the results of a survey on the application of conceptualizations in energy systems modeling that was carried out jointly by the Imperial College London and the Delft University of Technology [KvD11]. It revealed that:

> "[...] people use a mixture of tools and software packages, with only 20% using a single tool and more than half (53%) using three or more different software tools."

There are several reasons for this diversity of tools (and hence models), especially in the field of academic research. As shown above, the Smart Grid is a domain with a wide technological diversity. Hence, research projects usually involve a number of partners, each with expertise in a special field, which either develop models from scratch or provide existing models for the project that have already been validated. In the latter case (reusing the existing models) it would be beneficial if an approach existed for integrating the models provided by different partners without having to convert or reimplement these, as the confidence is lost, "due to the modifications to the model required by the conversion" [RAF+00, p.3]. But even if different project partners develop models from scratch, an integration approach for these models may be desired. This may be the case when the partners do not want to disclose the source code of these models or because the different models require specialized simulation hardware that is geographically distributed among all the partners' institutes. Furthermore, an integrative approach allows the different project parties to use the platforms and tools they feel comfortable with, thus increasing development speed and model quality.

Although 92% of the respondents of above mentioned survey were from an academic background, the need for linking existing tools for Smart Grid simulation is also felt by industrial companies as an E-Mail from one of the members of the *OpenSG Simulations Working Group*[2] reveals:

[2] `http://osgug.ucaiug.org/SG_Sim/default.aspx` (accessed 05 May 2012)

"[...] there is a large installed base of mature power system simulation tools that have evolved over decades and have the trust of the energy service providers who must rely on them. New tools are certainly required but I would argue that just as important is the need to provide guidance on how the existing tool base should be used to address smart grid applications not previously modeled extensively using these tools. [..] Modeling guidelines and 'glueware' linking these tools to evaluate the impact of communications and control systems is particularly needed."[3]

Again, one of the reasons for integrating different tools is the trust in established tools which should also be used for Smart Grid simulations. As these are usually specialized tools, they are only capable of providing models for a special domain, e.g. power flow analysis. Other models have to be developed with different tools and therefore different tools need to be integrated for performing a joint simulation of all models.

1.3 Research Objective

Following the above discussion, this thesis is based on the primary assumption that simulation model reuse (by coupling the different tools that provide these models) is a desirable objective (**A 1** – *Model Reuse is Desirable*).

However, besides the mentioned benefits there is also a number of obstacles [RNP+04, Pid02]. For example, there are increased up-front costs for creating reusable models and there is little motivation for a model developer to design a model in a reusable way as he or she has no short-term benefit but usually only more work to do, for example by having to implement a standardized interface. Furthermore, a simulation model may be trustworthy and validated with respect to its original context/purpose. But this may not be the case when reusing it in a different simulation study. For these reasons, certain techniques are required to achieve the positive effects of reuse, as [RÖ8] points out. Therefore, Section 2.1.6 introduces and discusses compositional techniques from the modeling & simulation domain.

At the OFFIS, a first attempt to build a Smart Grid simulation by reusing existing models was done in the project *GridSurfer* [NTS+11]. Although the project was a success, the reuse-based approach indeed revealed a number of obstacles which make reusing models a challenging task:

- Different simulation platforms may have to be used.
- The simulation models may not provide interfaces that allow easy integration with other models (control- and data flow is not accessible from outside the simulation platform).
- Configuration parameters for the models may be defined in different formats (e.g. to select the timespan to simulate).
- The simulation models may have different temporal resolutions and fidelity.

[3]Gunther, E.: "Re: Power System Simulation Tools to consider for best practice development". Message to the OpenSG Simulations Working Group, 8 March 2011. E-Mail.

- The simulators may include different, tightly coupled Smart Grid control strategies.

To overcome these obstacles for future research projects, this thesis pursues the following research objective:

Research Objective
Develop a concept for building a simulative testbed for the evaluation of new Smart Grid control mechanisms by composing available simulation models.

Simulation composition according to Petty and Weisel [PW03b] is the "capability to select and assemble simulation components in various combinations into valid simulation systems to satisfy specific user requirements". This definition implies two important challenges that need to be solved: The formal definition of the user requirements and the semantic description of the simulation components. The latter is required to detect meaningful combinations of the components. In this thesis the specific user requirements for evaluating a control strategy are the scenarios[4] in which a control strategy is to be tested. A metamodel that allows the specification of a broad range of large-scale Smart Grid scenarios will be the main contribution of the thesis. Besides these two challenges implied by the above definition of composability, two technical aspects related to interfacing the simulators and the control mechanisms complete the set of research questions this thesis intends to answer:

RQ 1 What interface is required to integrate different simulators into composed Smart Grid scenarios?

RQ 2 What information about a simulator and its models is required to facilitate the automatic composition of valid scenarios?

RQ 3 How can (potentially large) Smart Grid scenarios be described in a formal way that allows to compose them automatically using available simulators and their simulation models?

RQ 4 How can control strategies be evaluated using the simulated Smart Grid scenario (technically)?

The mosaik concept

The mosaik concept presented in Part II of this thesis intends to provide a solution to all four areas the research questions pose a challenge on, namely syntactic interoperability, semantic-based validation, scenario modeling and control strategy integration. Figure 1.3 depicts the overarching idea of this concept, which is the separation into a physical topology and an informational topology. The physical topology involves the power grid itself and the different resources that have a physical (usually electrical) connection to the grid. It also involves other models that have a physical connection to the resources, such as climatic models (e.g. solar irradiance models required for detailed PV model operation). The informational topology comprises all other aspects

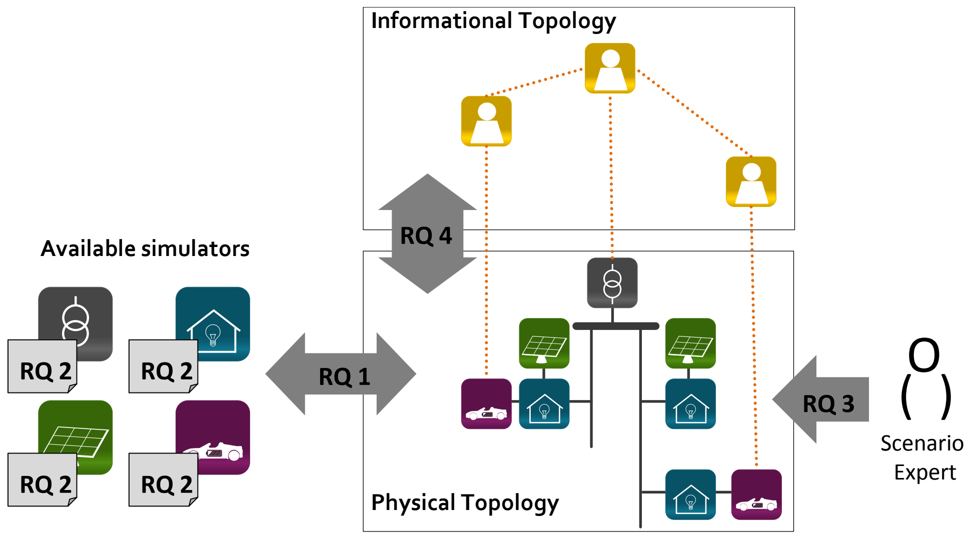


Figure 1.3: Smart Grid simulation approach following the mosaik concept

of the Smart Grid, such as multi-agent based control (see Chapter 2.2) or communication infrastructure simulations.

The first research question deals with the syntactic integration of the available simulators. The mosaik concept defines a corresponding interface called *SimAPI*, which allows a discrete-time (fixed-step size) integration of the different simulators and their models. Obviously, for being integratable into the mosaik concept, a simulator must be able to implement the SimAPI (see Chapter 6.4.1) and to conceptually map the simulation paradigm used internally to the discrete-time approach (see Chapter 6.4.2). In the evaluation it will be shown that these preconditions are given for the typical types of models that are within the focus (see Chapter 3.4) of the mosaik concept.

The second research question deals with the definition of additional semantics for the simulators and their models. The mosaik concept provides a semantic metamodel that allows to capture this information (i.e. instances of the metamodel are semantic models of the simulators and their simulation models). This information provides the basis for allowing automatic composition and validation of the specified scenarios.

The definition of the latter is within the focus of research question three. Here, the mosaik concept provides a scenario metamodel that is closely connected to the semantic metamodel and allows a Scenario Expert (see stakeholder definition in Chapter 3.1.2) to define the physical topology that is to be simulated. The mosaik concept also defines the algorithms to perform the composition (interpret scenario models) and execute the simulators (manage the data flows between these).

The fourth research question lead to the specification of an interface called *ControlAPI* which allows to implement control strategies against the simulated physical topology. Chapter 10.5 discusses how a mapping to standardized data structures and protocols allows a loose coupling between the informational and physical topologies, resulting in a realistic and reusable implementation of the former.

[4] What is meant by the term *scenario* within the context of this thesis is defined in Section 3.2.

To get a better understanding of the overall concept, Figure 1.4 gives a high-level overview of the resulting Smart Grid simulation testbed. It is implemented in Part III of this thesis to allow the experimental evaluation of the concept. The semantic as well as the scenario metamodel provide the basis for a textual scenario editor. Specified scenarios are submitted to a simulation server which instantiates the required simulator processes (integrated via the SimAPI), performs the composition and executes the simulation. During a simulation, control mechanisms can interact with the simulated objects via the ControlAPI and simulation data (outputs of the objects) is written to a database. The analysis of the simulation results is not within the scope of this thesis.

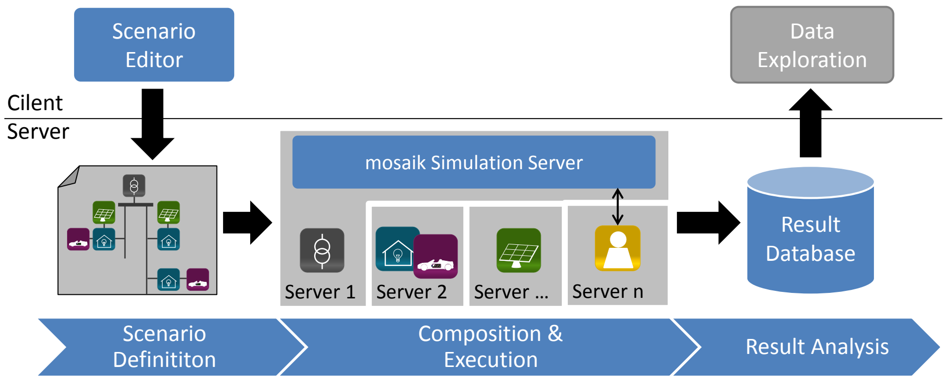


Figure 1.4: Overview of the mosaik Smart Grid simulation platform

1.4 Research Strategy

The work will follow the inductive-hypothetical research cycle [Sol82] which has already been applied successfully by Boer [Boe05] to "provide an architecture for coupling simulation models [...] in industry." The cycle consists of five steps which are depicted in Figure 1.5: initiation, abstraction, theory formulation, implementation and evaluation.

Initiation

In this first phase, the researcher perceives the current situation and creates a descriptive empirical model that reflects the relevant aspects of the problem domain. Boer [Boe05] points out that literature review and personal observations play an important role in this phase. In this sense, real-life situations are taken as the starting point for research [Dig11]. This thesis therefore analyzes different Smart Grid scenarios from past projects and literature research (Section 3.2) as well as existing simulation models (Section 3.5). These domain-specific observations are supplemented by literature research in the general field of simulation composability (Section 2.1.6) to obtain an in-depth understanding of the state-of-art for the problem domain.

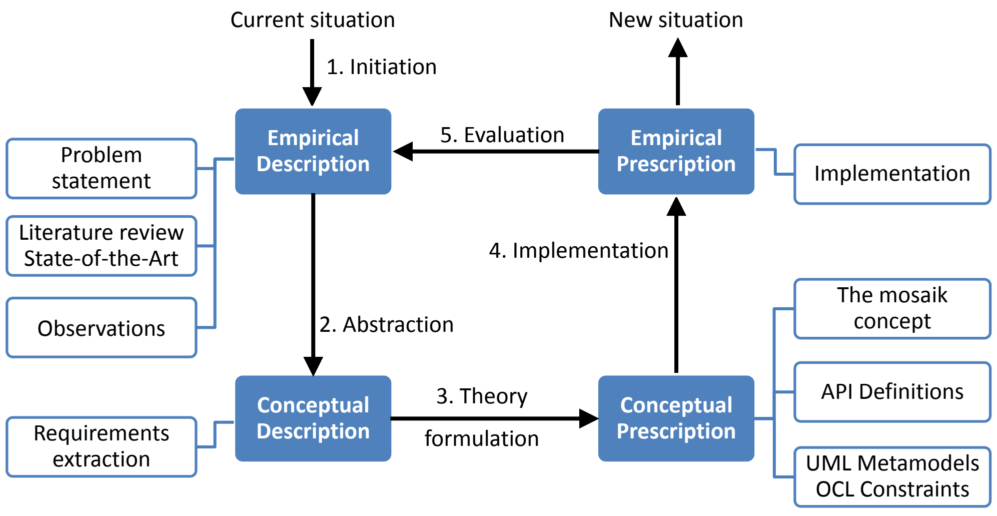


Figure 1.5: Adapted inductive hypothetical research cycle

Abstraction

The findings from the previous step are abstracted into a set of requirements which reflect the essential aspects of the problem and prescribe the features a solution has to meet in order to solve the problem adequately.

Theory formulation

As Diggelen [Dig11] points out, the aim of this activity is to find appropriate solutions for the problems that were conceptualized in the previous step. Following the research objective specified above, the term theory therefore refers to an appropriate concept that allows the creation of a simulative testbed for the evaluation of new Smart Grid control mechanisms by composing simulation models. This concept is presented in Part II of this thesis in form of API definitions as well as UML metamodels and corresponding algorithms for interpreting the instances of these models which describe the Smart Grid scenarios that are to be composed.

Implementation

The result of this phase is “an artifact [...] that aims to change the current situation” [Dig11]. The implementation phase makes the prescriptive model (the mosaik concept) operational.

Evaluation

To analyze to what extent the implementation artifact actually improves the current situation, i.e. answers the research questions, an evaluation needs to be carried out. Shaw [Sha02] has identified different types of questions, results (artifacts that aim to improve an existing situation) and validation procedures. These are depicted in Figure 1.6. The evaluation path chosen for this thesis is highlighted. The research questions defined above can be classified as design questions, because the mosaik concept provides a number of design artifact (APIs, metamodels) rather than a methodology, for example. The result is a notation (a formal language to support Smart Grid scenario modeling) and a system that can interpret the language and perform the actual simulation. According to Shaw [Sha02], evaluation includes feasibility studies/pilot projects and other criteria (such as the initially defined requirements) whereas examples are based on fictive demonstrations of how it works on a toy example which may be motivated by reality. In case of mosaik both, case studies from real world projects and fictive scenarios (inspired by reality) are used to demonstrate that the developed concept is sound. Also, a discussion of how the requirements could be met is included. The shown combination of question, result and validation type is a typical [Sha02] evaluation path which has been applied in a number of cases (for example [Boe05, RÖ8]).

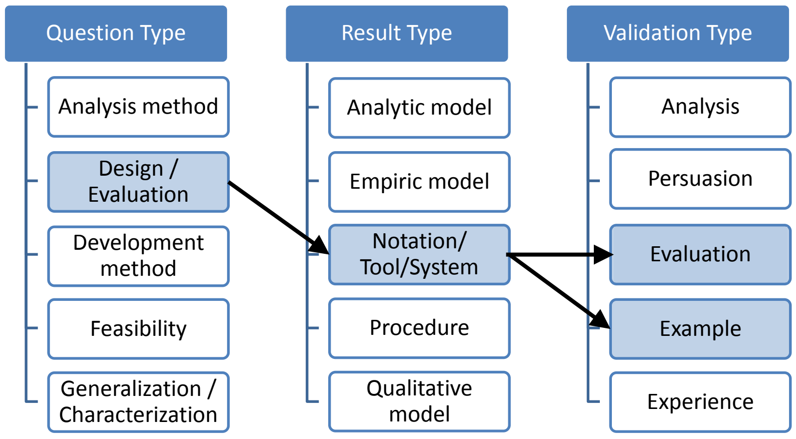


Figure 1.6: Chosen evaluation path (based on Shaw [Sha02])

1.5 Outline of the Thesis

The outline of this thesis is shown in Figure 1.7. On the highest level it is structured into three different parts which match with the different phases of the applied inductive-hypothetical research cycle. Part I starts off with this introductory chapter. Chapter 2 provides the reader with the required background knowledge for this thesis. In Chapter 3 existing simulators, models and scenarios (partly from literature) are analyzed and important requirements are extracted. Finally, Chapter 4 discusses related work in the field of Smart Grid simulation, by comparing the existing approaches with the set of requirements established before.

Part II of the thesis contains the "theory formulation" phase of the research strategy. Chapter 5 introduces the conceptual layers that make up the mosaik concept and gives an overview of related works in the field of simulator composition (Chapter 4 only deals with related works in the field of Smart Grid simulation). For the sake of structural cohesion, details of the related works are discussed in the following chapters each of which deals with a single layer of the mosaik architecture. Each chapter starts by identifying the set of requirements that is relevant for its scope and a discussion of related works. Next, a design that intends to meet these requirements is presented. Each

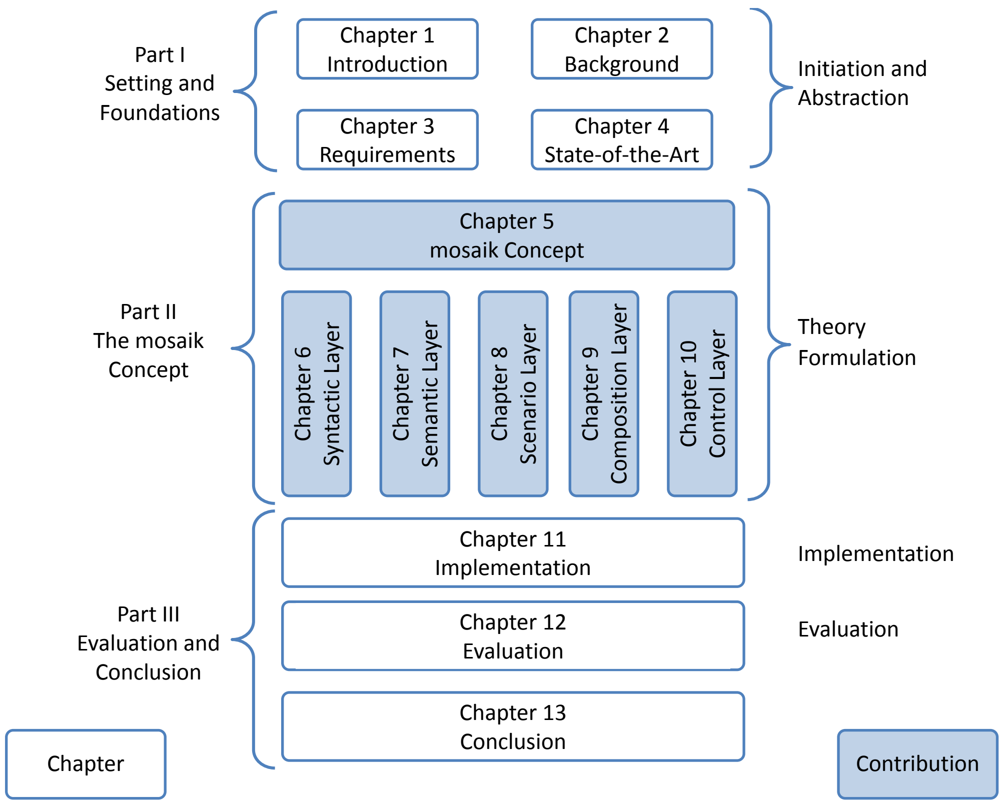


Figure 1.7: Outline of the thesis

chapter then concludes with a brief but complete discussion on how the requirements were considered and what limitations had to be made (mostly to limit design complexity and thus increase acceptance). Chapter 6 defines an interface, called *SimAPI*, for ensuring basic interoperability between the mosaik implementation and the simulators that are to be integrated. Chapter 7 presents a semantic metamodel that can be used for describing the simulator structure (its models and their entities), available configuration parameters and data flow semantics. Based on the semantic metamodel, Chapter 8 presents a scenario metamodel, the instances of which (called scenario models) are used to define the physical topology to be composed out of the existing models. This scenario metamodel represents the main contribution of this thesis as it contains a novel approach for specifying Smart Grid scenarios in a scalable fashion. Scenarios with many objects can be defined without much effort. The described scenarios can be validated in many aspects by analyzing the semantic descriptions of the participating simulators. Chapter 9 presents the algorithms required to interpret the scenario models and perform the actual simulation. Finally, Chapter 10 of Part II defines an interface for allowing control strategies to access the structure of the physical topology and interact with the simulated objects at simulation-time.

The thesis closes with Part III, Chapter 11 of which briefly describes how the mosaik concept has been implemented. In Chapter 12 two case studies and a number of theoretical scenarios (derived from use cases found in literature during requirements analysis) show that the mosaik concept allows to integrate a broad range of simulators and simulation models as well as to capture typical Smart Grid scenarios. A performance benchmark shows that the algorithms of the composition layer have a linear performance and thus meet the scalability demands of the scenario model (with a few exceptions that are being discussed). The thesis concludes with Chapter 13.

Throughout this thesis important findings are defined in a numbered fashion. This includes requirements (R), assumptions (A), OCL constraints (C) and validity aspects for the semantic layer (Validity Aspect). The latter are not abbreviated as they only occur a few times. After their definition, these findings are referred to using squared brackets and a short definition, e.g. [**R 2** – Scenario Specification Mechanism].

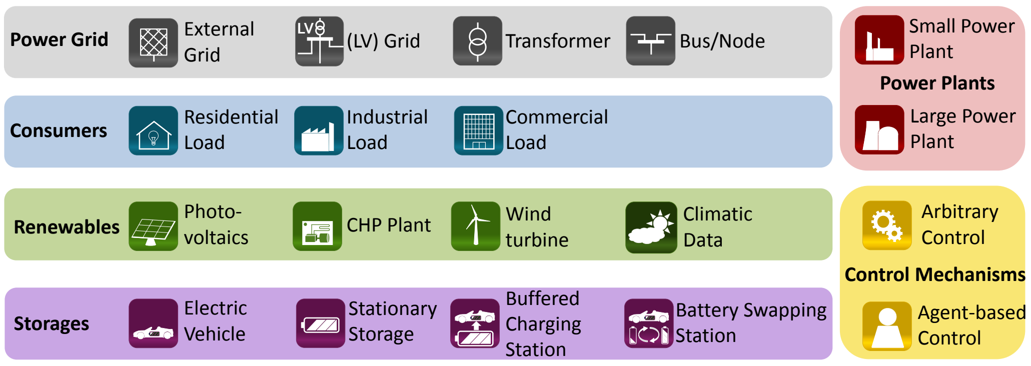


Figure 1.8: Legend of used icons

Furthermore, many figures that occur throughout this thesis are based on a number of icons that are shown in Figure 1.8. Depending on the context of the figure in which an icon is used, it may represent the respective simulated object, a simulation model including this object or the real-world object.

2 Background

In this chapter the required background for understanding the explications of this thesis is being introduced. This includes terms and concepts from the modeling and simulation domain, existing approaches to interoperability and composability as well as Smart Grid related knowledge about basic electrotechnical phenomena and relevant Smart Grid communication standards.

2.1 Modeling and Simulation

Modeling and simulation (M&S) is a widely used and accepted method for analyzing the behavior of complex systems as well as for performing what-if analyses (e.g. in future climate scenarios). Simulation follows the principle "learning by doing" [Fis95]. An executable model is created and experimented with. The big advantages of simulations are the capability to "compress long time into a shorter period" [BCN05] as well as the possibility to perform experiments with the simulation model that are not permissible with a real system, may it be for safety or for financial reasons. Furthermore, test conditions can be well defined, the tests are repeatable and they can be performed automatically [THR$^+$06, p. 23]. A key disadvantage of simulation is that a lot of effort has to be put into the creation of a suitable model and especially to ensure that the created model is valid. *Validity* is the degree to which a model faithfully reflects the behavior of the system it represents [ZPK00]. Obviously, the models for real world systems can never represent their behavior exactly for all aspects. However, it has to be ensured that the model is valid with respect to the experimental frame of interest (e.g. stationary power flow analysis). As already mentioned in the introduction, this validation effort is one of the key drivers for wanting to reuse validated models as building blocks for larger simulations rather than building these simulations from scratch.

For those readers who are familiar with the power system domain but not with the large field of modeling and simulation or simulation composability, the following subsections introduce the most important concepts and definitions.

2.1.1 M&S Concepts

Banks, Carson and Nelson [BCN05] define *simulation* as "the imitation of the operation of a real-world process or system over time." Such a *system* is a collection of entities (e.g. people or machines) that interact together to fulfill some logical goal [Law06]. In the automotive domain a system may be a vehicle that is operated by a driver to reach a certain destination. The entities here may be the driver, different sensors and actors as well as the major mechanical parts of the vehicle. In the Smart Grid domain this system could be the power grid with all its assets (transformers, lines, breakers, protection equipment) as well as the connected DER and classical loads and generators. To study the behavior of such complex systems different approaches are possible, as shown in Figure 2.1.

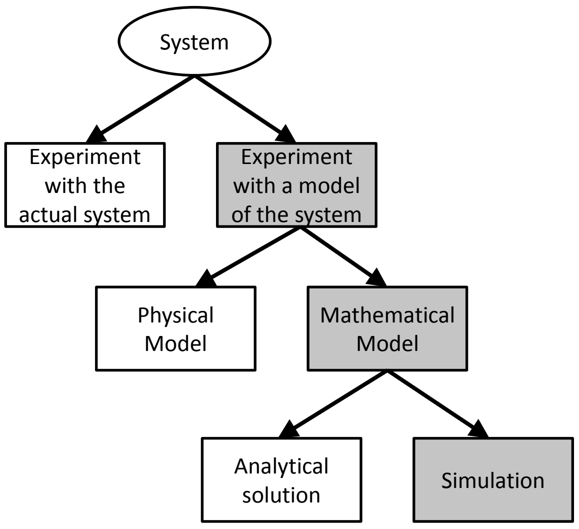

System
Experiment with the actual system
Experiment with a model of the system
Physical Model
Mathematical Model
Analytical solution
Simulation

Figure 2.1: Ways to study a system [Law06]

In case of the Smart Grid, experimenting with the actual system is limited to some smaller field trials simply because the elements characterizing the Smart Grid are not in place today on a large-scale and experiments with the real power grid are not advisable for safety reasons. Therefore, a model of the Smart Grid (or how it is supposed to look like) will have to be used. A physical model in this context may be a small-scale laboratory grid that shows similar phenomena to the real system. In this thesis, however, the term model refers to mathematical models that are studied by means of computer based-simulation, i.e. numerically exercising the model for available inputs to see how the outputs are affected [Law06].

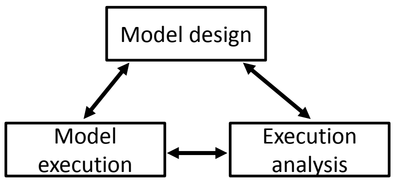


Figure 2.2: Steps of a simulation study (according to Fishwick [Fis95])

The process of creating a model is called *modeling* or *model design* [Fis95] and "comprises the conceptual work of defining the model by abstracting its data, processes, and constraints from reality" [WTW09]. Model design is the first stage of a tightly coupled and iterative process when performing a simulation-based study of a system which further involves the execution of the model and the analysis of the results that were gathered during model execution [Fis95]. This process is shown in Figure 2.2. Different methods for creating a model in the design phase will be presented in the next section. In the remainder of this thesis the terms design-time, run-time and analysis-time are used to refer to these stages of a simulation study.

A model may contain a number of *entities* representing objects or components in the system that require explicit representation [BCN05, p.68]. This is in accordance to the system definition introduced above. However, during modeling (abstracting from the real system), some entities of the system may be merged to a single entity in the model (e.g. an EV may only be modeled as a single entity and not as a complex interplay of battery, charging electronic, motor, etc...). This is a valid approach and maybe even necessary to keep the model complexity manageable as long as the level of abstraction is sufficient to answer the questions the simulation intends to answer.

In the remainder of this thesis the following M&S concepts will be used and defined as follows:

Definition 1 (Model) *A (simulation) model is an abstract representation of a system and is executed by a simulation environment (e.g. a model of a low voltage grid).*

Definition 2 (Entity) *An entity is any object or component in the modeled system that requires explicit representation (e.g. a transformer, a power line, an electric vehicle).*

Definition 3 (Simulator) *A simulator is the execution environment for one or more simulation models.*

Definition 4 (Simulation) *Simulation is the process of executing a (computer-based) simulation model.*

2.1.2 M&S Attributes

A number of attributes have to be distinguished that are commonly used (and often confused) to describe the properties of a simulation model. The primary attributes are fidelity, resolution and scale [Tol10]. *Fidelity* is used to express how good a model matches the behavior of the real system. Achieving high fidelity is not easy as usually a model cannot capture all aspects of the system it represents. However, low fidelity may be sufficient in some cases when it is caused by missing aspects of the system that are not relevant to the current subject of investigation. In this sense, a low fidelity model may be valid, as the term validity includes the requirements/sufficiency aspect. *Resolution* describes the level of granularity (or detail) in which a model captures the system it represents. A model of a car, for example, may either consider the vehicle as a single unit with weight, acceleration, speed and direction properties or model it in detail with all the components such as the engine, the gearbox, the dampers and so on. Depending on the objectives of the simulation study either may be a valid model. The term *temporal resolution* is used in this thesis to describe the minimal time-spans in which discrete simulation models (see below) can change their states and outputs. Finally, the attribute *scale* is used to describe the size of the simulated scenario. The more entities the scenario involves, the larger the scale of the simulation.

Table 2.1: A time taxonomy [ZPK00]

		Simulation/physical	
		Simulation time	**Physical time**
Local/global	**Global time**	Global, simulation: All components operate on the same abstract time base	Global, physical: All components operate on the same system clock.
	Local time	Local, simulation: A component operates on its own abstract time base.	Local, physical. A component operates on its own system clock.

2.1.3 Simulation and Time

When talking about simulations, different notions of time have to be distinguished, especially when simulation models with different notions of time are to be composed. *Physical time* refers to the time in the physical system that is being simulated. *Simulation time* (also called *logical time* [ZPK00, p.35]) is the representation of the physical time within the simulation model of this system. Finally, the term *wall-clock time* refers to the processing time of the simulation [PT06]. For example, a simulation model may be configured to simulate the behavior of the model over one year physical time. Five minutes of wall-clock time after starting the simulator, the simulation time may be advanced by 3 months. In this case the simulation time advanced faster than the physical time of the actual system. As mentioned above, this is one of the advantages of modeling and simulation as the potential behavior of a real system over a long time can be simulated in a much shorter period of time which is useful for doing forecasts, for example. A special case are real-time simulations, for example training simulators operated by humans. Here, the simulation time is synchronized in such a way that it advances in the same speed as the physical time of the simulated system. Besides this distinction into real-time and non-real-time simulations one can distinguish between a local and a global time base in case of composed simulations, as shown in Table 2.1.

Independent of the aforementioned taxonomy based on modeling flavors, simulation models can be categorized depending on the way in which they reflect the passage of time, i.e. their internal representation of physical time [vR12]:

Steady-state Time is ignored and a balance is found. For example in constraint based models, a balanced population rate (e.g. 5 coyotes, 200 rabbits) may be found.

Dynamic The modeled system evolves over time. Dynamic simulations can further be split into:

- **Continuous** State changes occur continuously (time is a real number).
- **Discrete** State changes occur at discrete points in time (time is still a real number). Discrete simulations can further be split into:
  - **Time-stepped (or Discrete-Time)** Time intervals are regular. The simulator executes the model in a loop with fixed increments of time.
  - **Event-driven (or Discrete-Event)** Time intervals are irregular. The simulator maintains an event schedule and time elapses from one event to another.

Of the dynamic simulations, the continuous approach is a classical one in natural sciences, using differential equations. Stepwise, numerical integration is used when simulating these models on a computer. This modeling approach can be "seen as the study of feedback phenomena, a form of system coupling where state changes of a component circle around to affect itself through a feedback loop" [ZPK00]. In the Smart Grid simulation domain this kind of modeling is usually employed for analyzing power system phenomena in the time-domain (see Section 2.3, below).

The time-stepped approach stands out through its simple simulation algorithm. Simulation is done stepwise and the models update their internal state (and related to this, their outputs) based on their state from the previous step and the current inputs [ZPK00].

The event-driven approach has the advantages that only those times are considered where state changes occur and only those events have to be communicated to coupled components that are relevant for their behavior [ZPK00]. This can speed up the simulation process dramatically in case of relatively few state changes (as compared to the time–stepped approach). However, calculating the event times is not always straight forward and may, depending on the nature of the modeled system, require expensive combined simulation approaches where continuous simulations have to be used to compute the event times [ZPK00].

As will be discussed in Chapter 6, the mosaik concept handles simulation time in a time-stepped fashion. The simulators that will make up the composed scenario will advance their time in regular intervals. Internally, however, these simulators may use any of the three dynamic sub categories (continuous, time-stepped or event-driven), as shown in Section 6.4.2.

2.1.4 System Types

Regardless of a particular simulation paradigm, i.e. discrete-time or discrete-event simulation, a number of system types can be distinguished. These are introduced briefly in the following as these distinctions are important for understanding the simulator scheduling options discussed in Chapter 9.6. As defined in Section 2.1.1, a model is an abstract representation of a system. In the following, the X denotes the set of possible inputs to which a model can respond and Y denotes the set of possible outputs. The variables that constitute the internal state of the model are described with the set S [Nut11b].

Moore-type systems

The output of *Moor-type systems* is computed solely on its current state and is often denoted by the function λ. As a consequence, the output of a Moor-type system is not directly dependent on the provided input values: $\lambda : S \to Y$

Mealy-type systems

The output of *Mealy-type systems* is a function of its current state and the current input: $\lambda : S \times X \rightarrow Y$

The output is therefore directly dependent on the provided input. By discarding the input that is supplied to the function a Mealy-type system can be used to imitate a Moore-type system. Hence, the Mealy type system is the more general construct [Nut11b].

Input-Free Systems

A special case are *input-free systems* which omit the input set X and thus are also called *autonomous* systems. The output depends on the state only (Moore-type). Potential state transitions are time-triggered. Given an initial system state, the response of an input-free system is uniquely determined [ZPK00]. In the context of Smart Grid simulation, environmental models or models based on time-series data are an example for input-free systems. Model configuration parameters can be used to determine the initial state.

Memoryless Systems

Another special type of system are *memoryless systems* which omit the state set S rather than the input set. The output is directly dependent on the input ($\lambda : X \rightarrow Y$) and there is no state transition function [ZPK00]. An example for such a system in the context of Smart Grid simulation are models for calculating stationary power flow, for example. These systems are also called *function specified systems* (FNSS).

Time-Invariant Systems

If the output of a system for a given input is identical, regardless of the simulation time the input was applied, a system is called *time-invariant* [Nut11b, p.47]. Although time invariance is not essential for designing simulation algorithms, time-variant (i.e. non-invariant) models must be supplied with an initial time and require access to the simulation time to allow incorporating time directly into their state transition function. As mosaik intends to compose models including their simulators, the latter (access to simulation time) is not within the scope of mosaik. But providing the initial simulation time has to be considered when describing a simulation scenario.

2.1.5 Distributed Simulation

The concept presented in this thesis has been implemented as a distributed simulation system. “The field of distributed simulation is based on the notion of combining a number of different simulation systems (each with their own models) together, to form a larger or more complex synthesized simulation space than any one of the systems could provide on their own” [TT08]. Initial efforts in the field of distributed simulation have their roots in the military domain [Boe05, p.19] where significant investments were made to make independently developed simulations work together at run-time [Pag07]. The initial trigger was the need for a simulated environment that could be used to support the training of military teams. As it was assumed that each military service (Air Force, Marines, ...) could best model its capabilities and these services where spread across the country the idea was to "move the electrons to the people rather than moving people to the electrons" to save time and costs. Besides this geographical motivation there are several other reasons for these efforts [Fuj00, p.5]:

Reduced execution time By splitting the simulation into sub-simulations that can be executed concurrently the simulation time can be reduced, e.g. to obtain the simulation results faster or to let the simulation evolve in real-time.

Fault tolerance If a simulation is distributed across different computational nodes it may be possible for other nodes to pick up those parts of the overall simulation that were executed on the faulty node, allowing the simulation to continue.

Integrating simulation models from different sources Hooking together different simulators rather than porting the simulators may be more cost effective.

The last of these reasons is the main motivation for the work presented in this thesis as the mosaik simulation concept will be developed for composing different existing, technologically heterogeneous simulation models into an overall Smart Grid simulation. Fujimoto [Fuj00] states costs as the critical factor for doing so. These costs can be traced back to two different tasks that have to be done when porting a simulation. First, the original simulation has to be analyzed and ported to another platform/programming language, etc. Second, and this may be at least as expensive as the porting itself, the simulation has to be validated again. Because by porting a simulation "the confidence in two of the three existing models is lost, due to the modifications to the model required by the conversion" [RAF+00, p.3]. As in every other domain it is necessary to use validated and trustworthy simulation models in order to generate sound and reliable results. Therefore, reusing simulation models is a desirable objective. In the next section, it will be discussed how this can be achieved by "composing" simulation models.

2.1.6 Simulation Composability

Obviously, to create a distributed simulation, different simulators and the simulation models executed by these need to be interconnected. The issues revolving around this interconnection are referred to using various terms like integration, interoperation, composition, configuration and so forth [Pag07]. There are two noteworthy publications that give a comprehensive overview of interoperability and composability as well as their historic development: [Pag07] and [Tol10]. The following discussion is based on these references.

The term composability was first used in the military domain in the mid-1990s in the "Composable Behavioral Technologies" project [Pag07]. Shortly thereafter, composability became an objective in a number of military simulation projects. Research about the impact of composability on the architecture of simulation systems [PRvdL99] as well from the computational complexity point of view [PO99] was conducted. In the early-2000s Petty, Weisel and Mielke analyzed the use of the term composability [PW03b, PWM03, PW03a, PWM05] and extended the work of Page et al. [PO99] and others. Their definition of composability became the most common one in the simulation community [Tol10, p.407]:

Definition 5 (Composability) *Composability is the capability to select and assemble simulation components [prior to run-time] in various combinations into valid simulation systems to satisfy specific user requirements.*

Figure 2.3 shows an abstract example of composability, where different simulation components can be taken from a repository and configured and combined in a number of ways to form different simulation systems. As Petty and Weisel [PW03a] point out, these components are not only limited to entity models but can also include network and user interfaces or other components. This also includes components/models from different domains and with different fidelity and resolution. Further it has to be noted that this composition already takes place at configuration-time, i.e. prior to run-time. According to Petty and Weisel [PW03a] the amount of effort put into this configuration and combination of components is an important factor to distinguish composability (less effort) and interoperability (substantial effort). They define interoperability as “the ability to exchange data or services at run-time whereas composability is the ability to assemble components prior to run-time.”

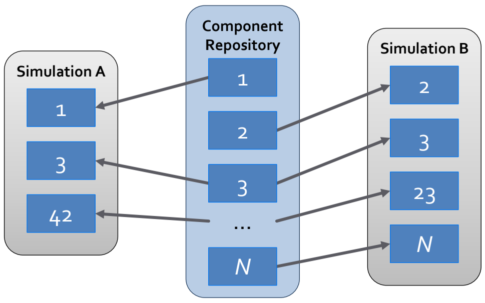


Figure 2.3: Abstract example of composability [PW03a]

Page et al. [PBT04, p.7] regard these definitions as “somewhat problematic” as it is not clearly stated how a run-time property (interoperability) can be a required precondition for achieving a design-time property (composability). They view that composability and interoperability are treated as independent properties of a model and suggest the following dimensions for what they call the simulation interconnection problem:

Composability Realm of the model (e.g. two models are composable if their objectives and assumptions are properly aligned).

Interoperability Realm of the software implementation of the model (e.g. are the data types consistent, have the little endian/big endian issues been addressed, etc.).

Integratability Realm of the site the simulation is running at (e.g. have the host tables been set up; are the NIC cards working properly).

Following these definitions, two models are composable if they are based on similar (compatible) objectives and assumptions. However, these definitions explicitly point out that the fact that two models are composable does not imply that they are interoperable, i.e. their implementations may be incompatible. Therefore composability and interoperability are independent properties but interoperability is a necessary prerequisite for composability. Tolk [Tol10] as well as Zeigler and Hammonds [ZH07] point out that systems must have a common understanding of the semantics to be interoperable which is not explicitly mentioned by the above definition of Page et al. [PBT04]. In the scope of this thesis the IEEE definition of the term interoperability is used as it implies this common understanding by claiming that systems must be able to use the exchanged information:

Definition 6 (Interoperability) *"Interoperability is the ability of two or more systems or components to exchange information and to use the information that has been exchanged." [Ins90]*

2.1.7 Approaches to Composability

Röhl [RÖ8] distinguishes three fundamentally different approaches to compose simulation models which are briefly presented in the following and discussed with respect to their relevance for this thesis.

Generic, modular and hierarchical modeling formalisms Noteworthy examples for this category are the *Discrete Event System Specification* (DEVS) [ZPK00] as well as the language Modelica [Mod]. Both allow to describe models in a modular and hierarchical fashion and have frequent, international workshops [RÖ8].

Modelica uses an equation-based approach developed to describe physical systems (electronic, hydraulic, thermal, etc...) in an object-oriented fashion. Models are defined in form of parameterizable classes that can be wired together via typed ports (e.g. a port of a hydraulic storage may define a mass flow in kg/sec). While in Modelica in models the "equations of the models are described non-causally, and [...] the equality holds for all values of time" [Bro97], the DEVS formalism allows to define discrete-event models. Based on a set of allowed inputs, outputs (grouped by ports) and states, DEVS allows to define state transition function in a state machine fashion. By allowing to specify couplings between input and output ports a modular, hierarchical modeling is enabled. The created models are then executed by some simulation tool that can understand the DEVS formalism [RÖ8].

As mosaik does not aim to provide a modeling mechanism or language for describing the behavior of models, this low level category of composition is not considered any further.

Model integration by simulator coupling This can be seen as the opposite approach to the aforementioned modeling formalisms. Models are not composed on the basis of well-defined modeling formalisms but rather as operational models in form of complete software units. The advantage of this approach is the ability to integrate very heterogeneous models [RÖ8, p.14]. This approach is a suitable one with respect to the research objective and the specific requirements of mosaik (see Chapter 3).

Obviously, for composing models this way, the corresponding simulators that execute the models have to interoperate. Therefore, Section 6 discusses related work dealing with this issue and subsequently a mosaik specific approach for interoperability is being presented.

Modeling-language independent validation of compositions The two categories for composing simulation models discussed above do not consider the validity the resulting compositions [RÖ8, p.19]. A common approach to validate compositions is to compare its behavior to a reference system. The latter can either be a real system or a fictive, ideal model. Validity is then expressed as the degree to which the behavior of the composition matches with the reference system. How mosaik ensures validity of the composition is discussed in Chapter 7.

Although these approaches are fundamentally different they may very well be combined. For example, a formalism may be used to compose a larger model from different smaller models defined in this formalism. The resulting model could be executed by a corresponding simulator that can understand this formalism. Next, the whole simulator could be coupled with another simulator using a completely different type of model description (e.g. a Java implementation). Finally, a validation formalism may be used to validate the composition by either generating the required information from the formal model or by manual creation of a model description.

2.2 Multi-Agent Systems

As the objective of this research is to create a test-bed for especially multi-agent based Smart Grid control mechanisms, this section gives short introduction into multi-agent systems (MAS) and why they are often used in Smart Grid control approaches. According to [WJ95], a computer or hardware based agent is a system that exposes the following characteristics:

Autonomy Agents operate without any direct intervention of humans or others and have some kind of control over their internal state and the actions they take.

Social Ability Agents can communicate with other agents using some agent communication language.

Reactivity Agents have the ability to observe their environment and can react to changes in their environment in a timely fashion. The environment can be the real physical world (e.g. perceived via sensors), a user-interface or the Internet.

Pro-Activeness Agents do not only respond to changes in their environment but are also able to show their own initiative to reach their goal.

Systems composed of multiple interacting agents are called multi-agent Systems (MAS) and these gained widespread recognition in the mid 1990s [Woo09]. This interest has been nurtured by the belief that the MAS are an appropriate paradigm for exploiting the possibilities offered by newly established large-scale, open and distributed systems such as the Internet. For MAS, different organization forms exist, such as hierarchies, holarchies, coalitions, teams and others. Figure 2.4 shows some of them. The chosen organization can have a significant, quantitative effect on its performance characteristics [HL05].

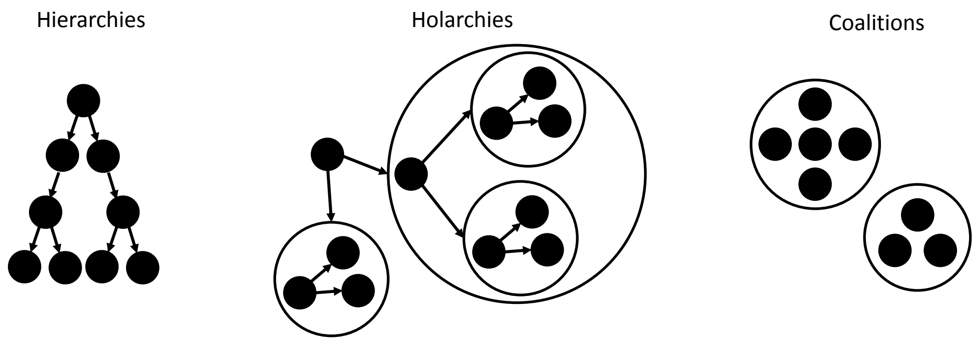


Figure 2.4: Examples for organization forms of multi-agent systems (MAS)[HL05]

MAS and Smart Grids

As the Smart Grid is also a large-scale distributed system, a number of MAS based control approaches have been developed within the Smart Grid community as well, such as [Trö10, HLR11, KRB$^+$10, LWSF11]. Pipattanasomporn et al. [PFR09] state that "in the context of power systems, multi-agent technologies can be applied in a variety of applications, such as to perform power system disturbance diagnosis, power system restoration, power system secondary voltage control and power system visualization." The belief is that MAS systems, due to their inherent architectural properties, are well suited for Smart Grid control tasks and offer a natural representation of the domain [JB03, p.67]. The organization form (see above) of the MAS can be created according to the topology that is exposed by the Smart Grid. Data, knowledge and control can be distributed and scalability can be achieved which is required as the computational power of centralized approaches would be exceeded. Furthermore, the distributed architecture of MAS has a number of characteristics that are desirable for the Smart Grid. These include robustness, scalability and flexibility. For an in-depth discussion of MAS technology in power system applications see McArthur et al. [MDC$^+$07] and Ramchurn et al. [RVRJ12].

2.3 Power System Analysis

When doing simulation studies in the Smart Grid domain, the power grid is a major component that has to be taken into account. Control mechanisms may do their job well from a market or supply/demand matching perspective. But only when taking into account the power grid, it can be made sure that the actions taken by the control mechanisms do not violate the operational constraints of the power systems' assets or adversely affect power quality (see 2.3.2). For readers who are familiar with modeling and simulation but new to the field of power system analysis, this section briefly discusses the electrotechnical terms and analysis methods used in the remainder of this thesis.

2.3.1 Operational Constraints

When operating a power system, the operational constraints of the different components, that is, the allowed states of operation, have to be met. Otherwise, the component that is operated beyond its limits will be damaged or even destroyed completely. With respect to the transmission and distribution of electricity, the main components that need to be considered are transformers, overhead transmission lines and underground cables. Operational constraints depend on the type of device and, for example in case of underground cables, depend on the environment temperature and surrounding material [HDS07]. Of course, the power generation units do also have such limits (i.e. minimal and maximal possible generation capabilities). If generation units reach their limit, power quality (see below) is likely to decrease, for example because supply can no longer follow the demand and the frequency is reduced. Overloading of components does not necessarily lead to severe damage. Some component types tolerate short-term overloading but this most likely results in a reduced life-time [MA12, p.494].

For Smart Grid simulation two types of models have to be distinguished. In the first class of models, the operation limits are part of the model. Regardless of the models inputs (e.g. commands from the control mechanisms), the simulated entities cannot be operated outside their limits. In this case commands that request operation outside the limits are either ignored, the next possible state is selected or the model raises an exception and terminates. In any case, it would be beneficial if the control mechanism had the information about the possible operation modes. The other class of models can be operated outside the allowed operation limits because a mechanism to keep the operation state within the limits is not part of the model. An example for such type of model are power grid models (e.g. within stationary power flow analysis tools as discussed below). As with the real power grid, the modeled lines and transformers cannot do anything about being operated outside the limits on their own. In the real grid, it is the task of the system operator to activate load shedding, reduce generation or reconfigure the power grid by routing the power differently. For the power grid models used in Smart Grid simulation this functionality is usually not part of the model but rather part of the control strategy that is to be evaluated using the simulation models. In this case it is essential that the operational limits of the different entities of the power grid are known to the control strategies.

2.3.2 Power Quality

Besides the aforementioned operational constraints that have to be met, power quality is an important aspect to consider. Perfect power quality is given when “the voltage is continuous and sinusoidal, with a constant amplitude and frequency” [Lar12]. In Europe, the voltage quality for the different voltage levels of the power grid is defined by EN 50160 [Eur10a]. The ubiquitous frequency of the power system is constantly changing because of loads that are switched on and off as well as intermittent distributed generation. Under normal conditions, the frequency f is allowed to vary within the allowed range of $49.8Hz \leq f \leq 50.2Hz$ as defined by [Eur09]. A number of control mechanisms (namely primary, secondary and tertiary control) that are activated hierarchically, as described in [Eur09], ensure that this frequency is met. A loss of power quality may lead to damage of devices on the consumer side and can also cause damage to the assets of the power system [BPA11].

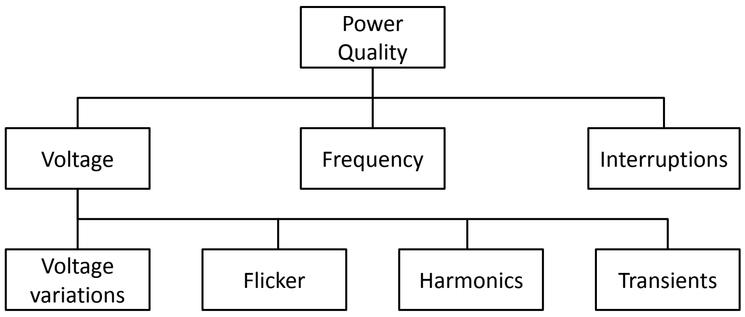


Figure 2.5: Classification of power quality phenomena [Lar12]

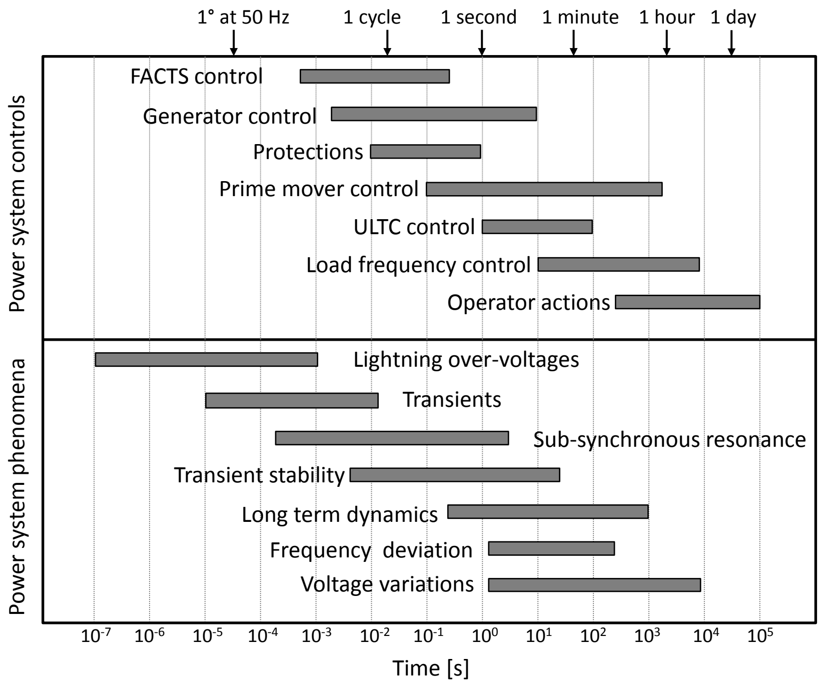


Figure 2.6: Power system controls and phenomena (based on Milano [Mil10])

There is a number of power system phenomena that adversely impact the power quality. Figure 2.5 shows a classification of these phenomena which can be broadly split into voltage related phenomena, frequency deviations (i.e. a mismatch between demand and supply) and interruptions. Counter measures for most of these phenomena exist. Figure 2.6 depicts the time scales of these phenomena and a number of controls that are of interest in power system analysis [Mil10].

As "taking into account the dynamics of each phenomena in an unique model would results in an intractable system" [Mil10], special tools for analyzing different types of

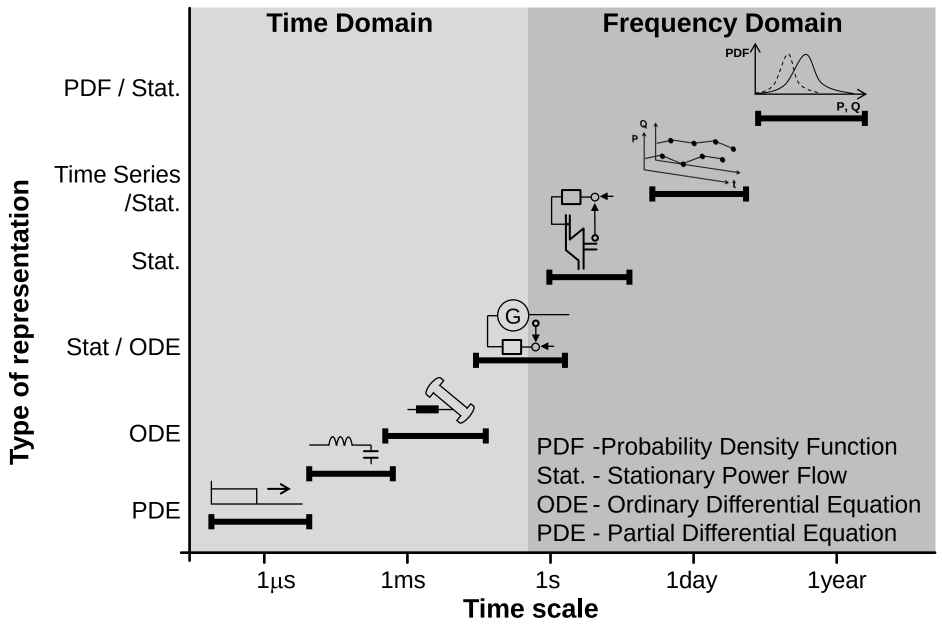


Figure 2.7: Model representations for different time scales [DBK12]

phenomena are used. Figure 2.7 gives an overview of the modeling techniques that these tools employ for capturing the phenomena on the different time scales. These can be divided into the two broad categories of *time-domain* and *frequency-domain* simulation. The term frequency-domain thereby represents the simulations on a larger time scale as it is assumed that the power system has a fixed frequency and frequency variations (harmonics, transients, subsynchronous resonance, etc...) do not occur. As will be shown in Section 3.4, the mosaik concept presented in this thesis focuses on the composition of simulation models for the analysis of power system phenomena in the frequency-domain. The stationary power flow is a commonly applied technique in this domain (see Figure 2.7) and will therefore be introduced in the next section.

2.3.3 Stationary Power Flow Analysis

Stationary load flow analysis, as the name suggests, assumes a steady state. That is, all time derivatives are considered zero [KN12]. Hence, it is not suitable to analyze the dynamic phenomena such as electromagnetic transients or harmonics. The goal of stationary power flow analysis is to find all branch currents and all nodal voltages and their angles. It can be used to calculate the utilization of the power system resources and the power quality with respect to the voltage band constraint. In the real world such analysis may be done for anticipating the effects of future operation decisions. In the simulated Smart Grid as it is in the focus of this work, the power flow analysis is a vital tool for simulating the sensors that measure line currents and node voltages in the real power system.

Basis for performing a power flow analysis is a topological representation of the electricity network that is to be analyzed. There are basically two models to represent electrical networks:

Node-Breaker Model The node/breaker model is able to describe every physical component within that network, e.g. switches, ground breakers, conductors or single transformer windings. Two components that are connected in their node/breaker representation do not necessarily have a physically conducting connection. If there is, for instance, a switch located between them, its status (open or closed) determines whether or not an electrical current can flow between those components. There are various proprietary formats to store and exchange node/breaker representations of an electrical network. An open and standardized way to do this is using CIM (see Section 2.4).

Bus-Branch Model The level of detail provided by the Node-Breaker models is usually not required for performing power flow analysis (or power system simulation in general) as most of the elements contained in these models do not influence the behavior of the grid [MAE+04]. Therefore, it is desirable to transform topologies available in these models into so called Bus-Branch models that contain only the minimal required components for performing the analysis. This includes transformers, generators, loads, busses (nodes) and branches (power lines) whereas an explicit representation of switches and ground breakers is omitted as they do not influence the behavior of the circuit. Usually, the transformation between these models is not bidirectional due to the loss of structural information. The transformation involves three different actions [MAE+04]:

1. *translate* one component from the source model (node-breaker) to one component of the target model (bus-branch)
2. *amalgamate* two or more components from the source model into a single component of the target model
3. *ignore* a component in the source model

Figure 2.8 shows a small example of the two different network representations and how the components of the node-breaker model translate to the bus-branch model by stepwise amalgamation of node-breaker components. The open-source tool chain for stationary power flow analysis discussed in Chapter 3.5.5 uses such a translation when importing topologies in the CIM/XML format.

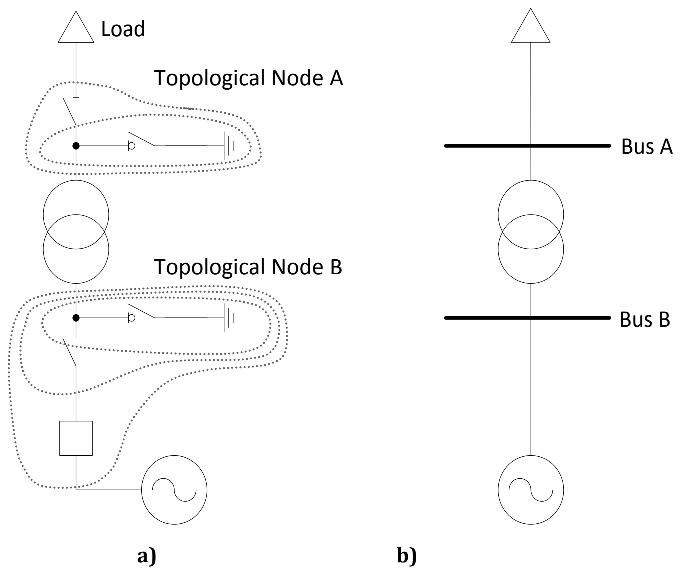


Figure 2.8: Substation feeder bay in a) CIM/XML node-breaker format and b) bus-branch format

Figure 2.9 shows an example for a power flow analysis and the given and calculated values. Loads are defined in terms of consumed active and reactive powers (P-Q load) and generators are defined by a constant voltage magnitude and active power injection (P-V generator[1]) or by a constant active and reactive power generation with the voltage being variable within a specified limit [Mil10]. For being able to perform the power flow analysis, for one generator (also called slack or swing bus) a constant voltage and phase angle has to be defined. The phase angle has to be fixed as otherwise the system equations are under-determined (see Milano [Mil10, pp.63] for detailed mathematical approach to power flow analysis). The active power is not specified as it cannot be fixed a priory because the transmission losses are unknown at that point in time. In some tools these load or generator types are also called P-Q/P-V bus [Mil10]. As Milano [Mil10] points out, "these expressions mix together two different concepts: (i) the concept of bus, which provides a topological information and (ii) the concept of generators or loads, which are devices connected to a bus. Actually, there is no real matter in connecting a P-Q load and a P-V generator at the same bus. Thus, strictly speaking, expressions as 'P-V bus' or 'P-Q bus' are misleading and should be avoided" [Mil10]. Furthermore, the branch characteristics have to be provided in terms of length, resistance (R), inductance (X) and capacitance (C).

System representations

Real-world power lines usually have different phases as, for example, less material is required to transfer the same amount of power [Ger08]. For stationary power

[1] To avoid confusion with the acronym PV referring to photovoltaics, a hyphened notation is used when referring to the power flow generator types. P-Q is also hyphened to achieve consistency.

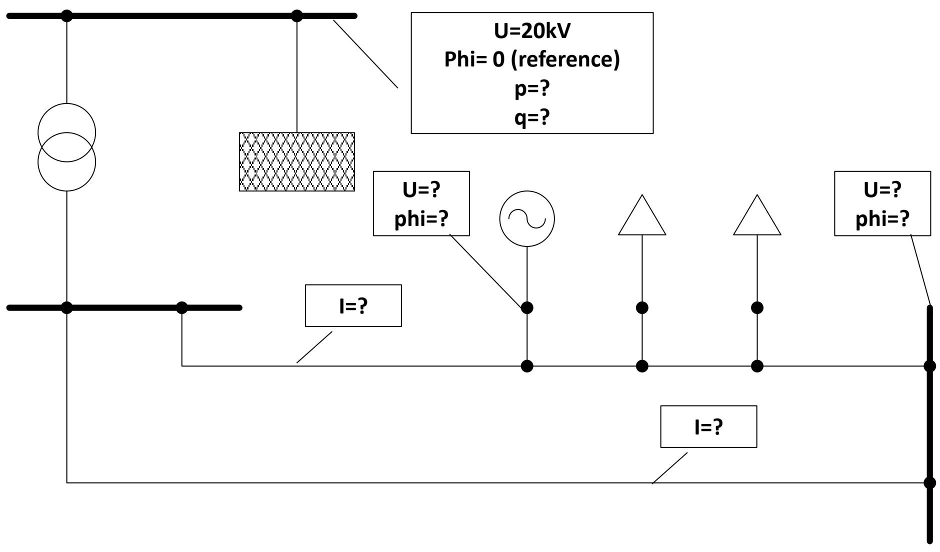


Figure 2.9: Example of a simple power flow analysis

flow analysis often simplified single-phase equivalent models are used, assuming that the power system is balanced among all three phases (i.e. same utilization). These differences in detail have to be considered when composing models in the power system domain.

2.4 Smart Grid Standards

Finally, this section briefly introduces two important standards that are widely used in the Smart Grid domain and which are referred to in different parts of this thesis.

IEC 61970/61968 Common Information Model

Recent Smart Grid standardization road maps (e.g. [Nat10], [SG310]) are emphasizing the importance of standardized data models and protocols for enabling a Smart Grid wide interoperability of the different actors. According to [SG310, p.109] the IEC 61970/61968 Common Information Model (CIM) is one of the core standards in the future smart grid. The CIM was developed in the mid 90's by the Electric Power Research Institute (EPRI) and is now maintained within the IEC mainly in the Working Group (WG)13 - "Energy management system application program interface (EMS-API)" and WG14 - "System interfaces for distribution management (SIDM)". The idea was to provide a common information model to support the information exchange between different EMS and therefore prevent vendor lock-ins. Its large data model provides the possibility to model physical (like cables, switches) and abstract objects (like documents, schedules, consumer data) in the energy domain. The CIM is actually designed in UML using Sparx Systems' Enterprise Architect as the preferred modeling tool. Overall, the CIM provides an integration framework. The CIM not only includes the platform independent data model but also several technology mappings and interface specifications, which are standardized in different subparts within the CIM standards family. It can be said that the following three use cases are the main ones for using the CIM [URS^{+}09]:

Exchange of electrical network topologies The CIM allows to define profiles (subset of classes and relations). Currently, there are two officially standardized profiles to model transmission network topologies (CPSM) and distribution networks (CDPSM). Additionally, the serialization into an XML/RDF format is provided by the CIM. Thus it is possible to exchange electrical network topologies in a standardized way.

Coupling of different systems Through standardized generic interfaces provided by the CIM it is possible to generate standard compliant implementations to couple different energy systems.

XML-based message exchange It is possible to define XML Schema (XSD) files based on the syntax and semantics of the CIM domain ontology to derive CIM compliant messages, which can be used to exchange energy domain specific information between systems within a service-oriented architecture (SOA).

In this thesis the CIM will be considered when providing common semantics for simulation model composition (see Chapter 7 and Chapter 12.3).

IEC 61850

The IEC 61850 standard family for "Communication networks and systems in substations" was originally designed to achieve interoperability between intelligent electronic devices (IEDs) in substations, but meanwhile is used in the general field of power system automation. The standard enforces an architecture including a dedicated communication layer that can be replaced without having to change the internal data model of a device. This makes it easier to keep up with the latest communication technologies, whereas the functionality changes less frequently during a typical lifespan of the (substation) equipment (typically about 30 years). For these reasons, the standardization efforts focus on domain specific object and data models for commonly used functionality rather than on communication technologies [USR^{+}13]. This is somewhat similar to the mosaik approach, which also focuses on the composition of the physical topology (the structure and functionality is rather fixed, governed by the laws of physics), whereas the communication topology is out of focus as it represents the field of innovation which advances rapidly and can include a potentially arbitrary number of technologies, structures and functionalities.

As shown in Figure 2.10, the IEC 61850 data model is hierarchical. Each physical device is mapped to a logical device which in turn consists of a number of logical nodes which group the attributes for a certain functionality. Usually one server instance runs in an IED and the functions supported by an IED are "conceptually represented by a collection of primitive, atomic functional building blocks called logical nodes" [LC08].

CIM and IEC 61850

Both, the CIM and 61850 are published standards developed within the IEC. However, they were developed independently and do not share a common data model or modeling approach [EPR10]. Harmonization of both standards is an ongoing process, first results of which can be found in [EPR10]. The approach extends the CIM UML model to include key concepts/objects from the IEC 61850 standard, so that objects between applications using these standards can be interchanged seamlessly.

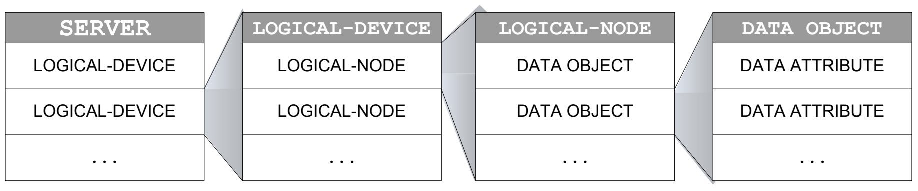


Figure 2.10: Hierarchy of the IEC 61850 data model [LC08]

3 Requirements

In this chapter the requirements that the mosaik concept has to meet will be discussed. The chapter starts with an analysis of the context in which mosaik will be used. This includes a classification of the different system stakeholder. Next, Section 3.2 and 3.3 discuss Smart Grid scenarios and control strategies that mosaik must support and identify related requirements. Based on this discussion, the scope of mosaik is presented in Section 3.4. Next, different types of simulation models are analyzed and corresponding requirements for integrating these are identified. Finally, additional requirements will be presented that are related to the analysis of simulation results as well as derived from experience gained in a former project. A summary of all defined requirements can be found at the end of this chapter.

3.1 Context of mosaik

As already mentioned in the introduction it is the research objective of this thesis to develop a concept for building a simulative testbed for the evaluation of new Smart Grid control mechanisms by composing different simulation models. This effort is part of the mosaik project which has been initiated at OFFIS. The presented concept has been implemented (see Chapter 11) and the resulting simulation platform is intended to serve as the main simulation platform for simulation studies in upcoming and future research projects.

3.1.1 Applicability to future simulation studies

As a consequence, the mosaik concept has to be general enough to be also applicable in future simulation studies, although the requirements elicitation in this chapter can only be based on the analysis of existing models, simulators and scenarios. In Section 2.3 it was shown that many different power system phenomena and control mechanisms exist. For simulating the different phenomena and control mechanisms, modeling approaches with different levels of detail exist. A general framework that is able to compose simulation models that capture all these different aspects is likely to be very complex and difficult to use. Therefore, Section 3.4 will restrict mosaik to certain types of Smart Grid simulation studies for which it has to provide a solution. As this restriction corresponds to the Smart Grid research focus of the OFFIS, it is assumed that the concept will also be applicable in the longer term. Furthermore, the scenario analysis in Section 3.2 will start with an abstract view on different categories of models in the Smart Grid domain. By analyzing different scenarios including models from all of these categories, it is expected that the concept is also applicable to scenarios that include other (yet unknown) models from these categories.

3.1.2 System Stakeholder

According to IEEE1471-2000 [Ins00] a system stakeholder is “an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.” Concerns in this context “are those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.” Inspired by the minimum types of stakeholders specified by [Ins00] (users, developers, maintainers) a number of stakeholders for mosaik can be identified. It has to be noted that these are the roles that people dealing with mosaik have. Theoretically a single person could perform all these roles.

The stakeholders of mosaik will be researchers and students who want to use the platform for evaluating new control strategies or analyzing other aspects of existing control approaches. In order to do this, they may also need to add new models into the platform. Furthermore, they only have a limited time frame for their work as students have a limited time frame for a thesis and researchers usually only spend a limited time at a research institute. A survey carried out by Keirstead and Van Dam [KvD11] has shown, that “in most cases it would take a new user days to months to become familiar enough with the models and tool in order to build and run a new analysis scenario”. And these results do not even focus exclusively on simulation composition but also include the learning curve for single tool approaches. Therefore, an important requirement is to keep the mosaik system as simple as possible while still allowing to reach the objective of the simulation studies. This will reduce the learning effort required for the fluctuating users of mosaik and ease the maintenance (**R 1** – *Keep It Simple*).

Users

This category contains all stakeholders that operate the system but do not do any implementation work. They use the provided user interfaces, command line interfaces or other high-level specification mechanisms (e.g. the *MoSL* DSL presented in Chapter 11.2.1) provided by mosaik. According to the three steps of a simulation study (model design, model execution and execution analysis – see Figure 2.2), the users of mosaik can further be subdivided into:

Scenario Expert A user who has the expert knowledge for specifying the scenarios that have to be composed and simulated. This includes knowledge about the real-world scenario that is to be simulated as well as knowledge about the available simulation models that can be used as building blocks to create the scenarios. Obviously, the mosaik concept must offer a means to allow the specification of the scenarios [**R 2** – *Scenario Specification Mechanism*]. The requirements for this will be discussed in Section 3.2 and the concept will account for them on the scenario layer (Chapter 8). Such a scenario specification mechanism is also the basis to allow the effortless recombination of simulation models as discussed in Chapter 2.1.6.

Simulation Operator A user who is responsible for initiating the simulation and monitoring its progress. This implies a number of requirements that are related to technical aspects (e.g. selection of the scenario file to execute). As these are not relevant in the context of this thesis they are not elaborated any further.

Scenario Analyst A simulation run is always related to a specific question. For example, "how well does our new control strategy for electric vehicles perform with respect to the load reduction in a particular part of the grid." For being able to answer such a question, it is the task of the Scenario Analyst to evaluate the simulation run by analyzing the data that has been generated. Therefore he or she uses certain metrics. The related requirements are discussed in Section 3.8 below.

Developers

So far, only the basic idea of mosaik has been presented in the introduction and the different problem areas have been identified (e.g. see Figure 1.3). Obviously, no detailed developer roles can be identified before the concept has been developed and presented in Part II. However, based on the identified problem areas a number of development roles can already be defined. These roles represent those stakeholders that are involved in the implementation of components required for performing a simulation study. This will include the development of the basic components of the mosaik platform as well as the simulation study or project specific elements such as new simulation models or data analysis tools.

Core Developer A Core Developer is responsible for implementing the mosaik concept and extending it according to new requirements that may arise in the future.

Modeler A Modeler is responsible for implementing a simulation model using a particular simulation framework/tool/platform. As the mosaik concept is based on the idea of composing existing simulation models (and thus models implemented in different technologies) he or she has the freedom to choose a preferred technology for the specific modeling needs and/or that allows to implement the model as fast as possible due to personal experiences with a certain platform. Thus, the time for learning a new technology can be avoided (see [**R 1** – Keep It Simple] which has already been introduced above).

Integrator It is the task of the Integrator to integrate a model into the mosaik platform. As will be shown further below, he or she has to implement a well-defined API (see Chapter 6.3) for allowing mosaik to interact with the simulation model. This API is accompanied by a structural and semantic description of the simulation model and the simulator that executes this model (see Chapter 7 and Chapter 11.2.1). In case of models that will be developed after mosaik has been implemented, the Integrator and the Modeler may well be the same person. In case of existing models (developed at OFFIS, University or other open- or closed-source models) the Integrator will likely be another person. Again, integrating a model should be as easy as possible (see [**R 1** – Keep It Simple]). A number of other requirements which have been identified in a past research project can be related to this role. These are discussed in Section 3.9 below.

Control Algorithm Developer The Control Algorithm Developer implements a control algorithm that needs to be evaluated simulatively. Therefore, he or she has to have a well-defined API for accessing the simulated objects (**R 3** – *Control Strategy API*). The detailed requirements for this API are discussed in Section 3.3.

Maintainers

Again, as the concept and the implementation have not been presented yet, only a general system maintenance role can be identified. More detailed maintenance roles will be presented when the concept is being discussed in Chapter 7.11.

3.2 Smart Grid Scenarios

Smart control algorithms are the key components that distinguish the vision of a future Smart Grid from today's power grid. To ensure that these strategies behave as intended and also to compare the performance of different strategies (with respect to different criteria, e.g. the impact on power quality), different Smart Grid scenarios need to be defined and simulated. Figure 3.1 shows the broad types of entities that are involved in such a scenario. These include entities of the power grid (e.g. transformers and power lines), the different types of DER as well as models of the environment (e.g. temperature, clouds, etc.) and of course the control mechanisms that observe some of these entities and may send control commands to them.

The mosaik concept aims to compose the physical part of the power grid by using different existing simulation components. This means that the scenario specification mechanism [**R 2** – Scenario Specification Mechanism] must be able to describe a wide range of physical topologies. It has to be pointed out that the definition of control algorithms is not part of the mosaik concept as these are external units to be tested with mosaik but not developed with it.

Falkenhainer and Forbus [FF91] define a *scenario* as "the system or situation being modeled." In the following sections a number of different situations of the power grid and their characteristics will be analyzed. These originate from literature research as well as from ongoing and planned research projects at OFFIS. The goal of this analysis is to come up with requirements for the scenario definition part of the mosaik concept allowing to model a broad range of these situations.

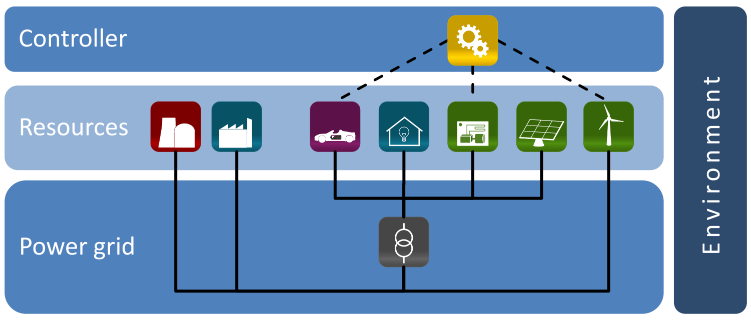


Figure 3.1: Abstract building blocks of a Smart Grid scenario

3.2.1 An e-Mobility Scenario

In a past project [NTS+11] an electric vehicle simulation model has been developed (see Section 3.5.2). It was used in combination with some other simulation models that will be presented below, such as PV, residential loads and power flow analysis models, to analyze the potential of EVs for increasing the share of locally produced green electricity in low voltage grids. In future projects similar scenarios will be simulated and may also be extended by new technologies. Figure 3.2 shows how such a scenario may look like.

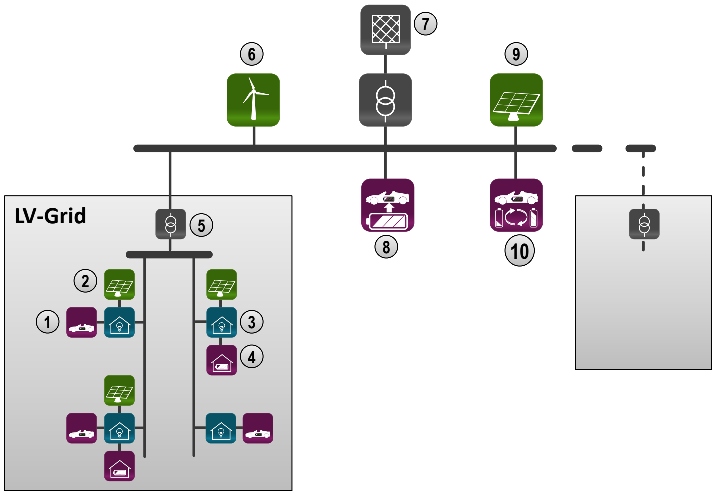


Figure 3.2: An e-mobility scenario

EVs (1), PV systems (2) and residential loads (3) are connected to nodes of a low voltage distribution grid (LV grid). Additionally, stationary battery storages (4) may be part of the scenario. A LV substation (transformer, 5) connects the LV grid to the medium voltage distribution grid (MV grid). In times when the local PV generation cannot cover the local demand, this transformer supplies the LV grid with power. If there are sufficient PV systems (or other local producers) to cover the local demand or maybe even exceed it, the excess power is fed into the MV grid. The MV grid also has a transformer which provides the remaining power that is required to balance supply and demand by drawing it from the transmission grid (7). As these higher voltage levels of the transmission grid are not within the focus of interest in any current or planned projects, it is simplified to an unlimited power source here and in the remainder of this thesis.[1] On the MV level larger power generation units are connected such as wind farms (6) or large-scale PV plants (9). Also, in future eMobility scenarios the MV level may be the connection point for battery buffered fast charging stations (8) or battery changing stations (10) which require a high power connection that cannot be offered by the LV grids.

Two observations can be made from Figure 3.2. First, the structure of the power grid is hierarchical. A large number of LV grids is connected to the MV grid, which again is connected to the transmission grid. Second, depending on the size of the power grid that is to be simulated, a large number of simulated entities and interconnections between

[1] For scenario analysis the average power mix (green, nuclear, coal) of the transmission grid may be available from an external data source to perform a sustainability analysis or such.

these need to be described when defining the simulation scenario. Therefore, the scenario definition mechanism has to be scalable with respect to the scenario size[2] (**R 4** – *Scalable Scenario Definition*). What concepts can be used to achieve this scalability will be described in Chapter 8. Second, some entity types may only be connected to nodes of the power grid that have certain other entity types connected to it. In the aforementioned example this is the case for the PV systems that should only be connected to nodes that have residential loads (e.g. because the scenario only contains roof-mounted PV). It must be possible to express these dependencies in the scenario definition (**R 5** – *Entity Dependencies*).

3.2.2 Grid Friendly Appliances

Another category of scenarios is related to the operation of resources based on information that these can obtain by locally measuring the power quality [Dyn05], [Kup08] and [Pac]. Figure 3.3 depicts the system proposed by Kupzog [Kup08]. Based on the local measurement of the ubiquitous grid frequency the load of a device is reduced or increased. Although such a frequency-based approach has the advantage of being able to operate without time-critical communication infrastructure, as the measurement and control components can be part of the device itself, some communication mechanism is required to parameterize the system [Kup08, p.66]. For example, to set the frequency/response ratio which is dependent on the number of deployed devices in the grid.

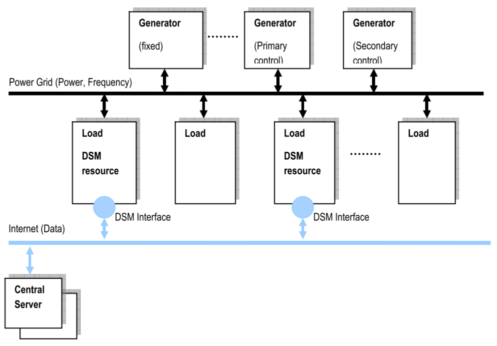


Figure 3.3: System with primary control provided by generators and a DSM system consisting of multiple participants coupled by the system frequency f and a communication network [Kup08]

Besides such frequency-controlled approaches there are also ideas of adjusting the (reactive) power behavior based on local voltage measurements [Bas08, KHH$^+$10]. The control mechanism is similar but the primary use case is to ensure voltage stability. From a more abstract point of view, P(F) controllers that regulate power depending on frequency and P/Q(U) controllers which aim to control local grid voltage by regulating active and/or reactive power can be distinguished.

[2] This notion of scalability has been defined in Chapter 2.1.2 and must not be confused with scalability of the implementation to provide good performance for large scenarios. The latter is not within the scope of this thesis.

With respect to simulation an important aspect is the capability of the devices to not only consume or feed in power but also to detect the grid frequency and/or voltage. In the real world such functionality is implemented within the resource itself using corresponding hardware [Kup08, p.107]. For simulating such a scenario the frequency information may be provided by the power grid model. As a consequence there is a data flow from the device models to the power grid model (the power drawn/fed in by the device) and from the power grid model to the devices (the local node voltage or the frequency). This way a cyclic dependency between the simulation models arises as depicted in Figure 3.4. Mosaik has to be capable of handling the resulting cyclic data flows between these entities (**R 6** – *Cyclic Data Flow Support*).

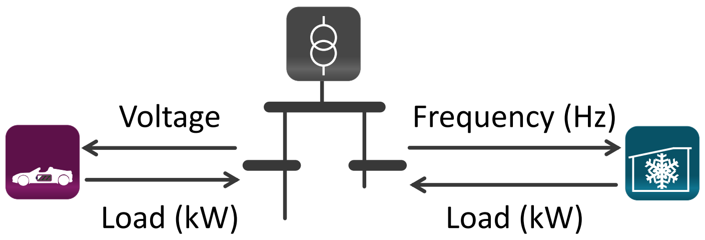


Figure 3.4: Data flow for simulating frequency or voltage controlled devices

3.2.3 Integration of Environmental Models

The simulation of cloud transients for grid areas that have a high penetration of PVs is another Smart Grid topic. A rapid decay or incline in PV power generation occurs when a large number of PV installations in close proximity is covered or uncovered by clouds. This puts a challenge on voltage stability [RVRJ12, GMD+10]. While Godfrey et al. [GMD+10] use a simple model for the solar ramp (see Figure 3.5), advanced modeling approaches for clouds also exist [BBFF12]. To test strategies for compensating these gradients, a combination of cloud transient, PV and power grid models as well as other loads (and generation) is required. Figure 3.6 depicts such a setting.

Assuming that a cloud model provides cloud density and movement for discrete geographic cells it would be convenient if the connection between these cells and the PV module entities could be established automatically during composition based on geo-coordinates of the PV modules. Obviously, a PV simulation model should be independent of any location for being universally usable. When the PV modules are connected to a power grid model of a real-world network (providing geo-location data), the location information for the PV panels should therefore be obtained from the nodes to which the modules are connected. Mosaik must be capable of exploiting this information for establishing connections between different simulated entities by comparing their properties (**R 7** – *Attribute Based Composition*).

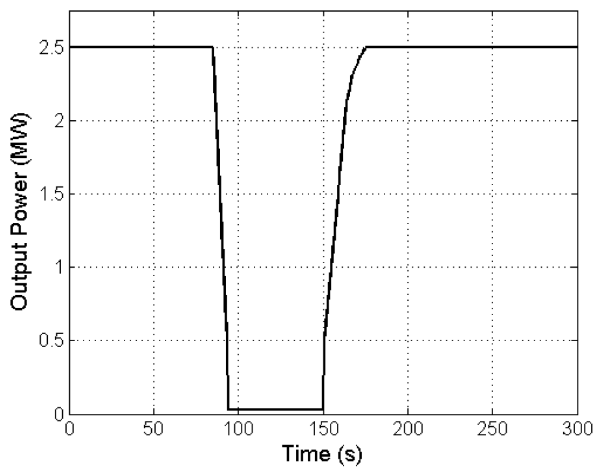


Figure 3.5: Simple solar ramp characteristics [GMD+10]

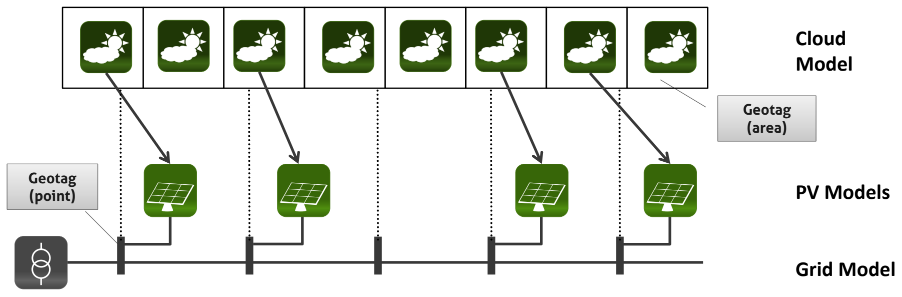


Figure 3.6: Detailed PV modeling by using a cloud model

3.3 Control Strategies

The scenarios discussed above only focused on the physical structure of the Smart Grid. But in addition to these elements, control strategies for all kinds of power grid resources, starting from a circuit breaker up to a pool of several thousand electric vehicles, are the major components that distinguish the vision of the smart grid from today's less controlled power grid. This has already been discussed in the introduction and the fourth research question asks for a solution that allows mosaik to be used as testbed for such controls strategies. A key requirement for meeting this goal is to provide an interface to allow control strategies to interact with the Smart Grid components simulated by mosaik [**R 3** – Control Strategy API] in a standardized way (i.e. the meaning and syntax of the exchanged data must be well defined and independent of any particular model).

As presented in Chapter 2.2, agent-based control strategies are seen as an appropriate paradigm to meet the complex and distributed nature of control tasks in the Smart Grid. As a consequence, a large number of research projects have developed and evaluated different agent-based control strategies, such as [KRB+10, Trö10, HLR11, LWSF11]. Also, the current Smart Grid research activities at OFFIS largely focus on agent-based control strategies. For this reason, especially the requirements for integrating agent-based control strategies are to be considered for mosaik.

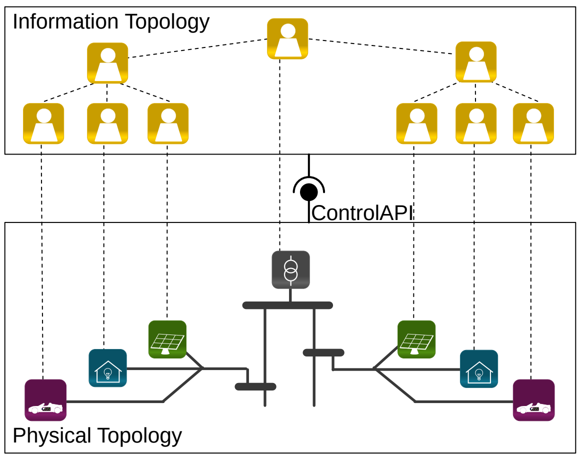


Figure 3.7: Separation into physical and informational topology

Topology awareness

As discussed in Chapter 2.2, different organization forms for agent based systems exist. There are agent based systems that form coalitions based on a topology-related proximity measure, for example considering the line distances between the agents' devices in the grid [BSA11, NS13]. This topology awareness allows the agents to take location-related aspects into account, allowing to provide ancillary services such as voltage control and reactive power supply. For such scenarios knowledge about the topology of the power grid is required by the agents. As a consequence it must be possible for the agents to access the relations of the simulated entities that are defined within the simulation model. The information topology in Figure 3.7 shows a simple example for an agent based control strategy using a hierarchical organization form. There is one agent for the complete distribution grid (transformer agent) and subordinate to this agent is one agent per grid node (node agent). Again, subordinate to the node agents there are resource agents each of which is assigned to one resource in the power grid. Besides the question what communication is required between the resources of the physical topology and the agents (see below), the creation of the information topology is an important aspect. How do the agents get to know their communication partners? This question is independent of the chosen organization form. When the agent system is started, an initial set of agents need to be created and initial connections between the agents have to be established. This can be done hard-coded by some management system of the corresponding agent-platform or by some bootstrapping component (e.g. another agent) [BÖ3]. The latter will usually be the preferred option as usually a large number of simulation runs is done with different parameters (seasons of the year, other scenario variants) as well as with different grid topologies. Manual bootstrapping is not an appropriate method in this case as simulating a new scenario requires changing the MAS setup. Some agent relations may be independent of the simulated scenario and specific to the chosen organization form but others are derived directly from the physical topology. For the latter, mosaik must offer appropriate functions for the bootstrapping component to navigate the physical topology and their elements (**R 8** – *Physical Topology Access*).

In the above example, the agent system bootstrapper (the component initializing the system [BÖ3]) may ask mosaik for all transformer entities and create an agent for each transformer. Next, the transformer agent itself may ask mosaik for the nodes of the grid that are related to the transformer and create a node agent for each of them. This process continues until each node agent has a list of subordinate resource agents.

The physical topology is defined on different levels. First, a model instance may include a number of entities that are connected to each other in some way. Consider the model of a LV grid, for example. A single instance of the grid model may contain a transformer entity and a number of interconnected bus and branch entities. Second, the topology is defined (extended) through the composition of the scenario, for example, when a consumer of a residential load model is connected to a bus of a power grid model. So the overall physical topology is a combination of entity topologies defined within the models and connections created by the composition. To avoid a separate, tool specific interface for retrieving the model internal relation information, the mosaik control strategy API should offer this information (**R 9** – *Intra-Model Topology Access*). For the agents, the overall physical topology shall be made available in a transparent way, that is the agents should not have to deal with the fact that some relations originate from model internal relations, while others have been created by the composition (**R 10** – *Combined Physical Topology*).

Operational limits

Once the information topology has been established, the simulation can start and the agent system needs to do its work. Therefore the agents need to access the status of their corresponding resource in the mosaik simulation and need to be able to send control commands to these (if appropriate). An agent must only operate the simulated devices within their operational limits. To ease the reuse of an MAS with different simulation scenarios, it has to be made sure that the agents are able to obtain these limit information from the device dynamically, that is without having to encode these limits manually into the source code of the agent. These limits are static properties of the entities, that is the data does not change during the simulation. Hence, this data needs to be retrieved from the simulators only once and will be called *Static Data* in the remainder of this thesis. Mosaik has to provide this data to the agents (**R 11** – *Static Data Access*). Furthermore, as this static data is time independent, it can be made available before the simulation has started, e.g. to compose the entities in a certain way. In that sense this requirement is a prerequisite for [**R 7** – Attribute Based Composition] which postulates a mechanism for using static data in the composition phase.

Control flow

Frameworks and platforms for developing such MAS usually do not account for simulation specific issues. In particular they provide no means for synchronizing the agents with the simulation time which does not progress implicitly as it would be the case in non-simulated, real-world control tasks [SGW08]. When do the agents react and how is their clock advanced? Does each agent have its own clock or do all agents follow a central approach? If not, how do the agents synchronize? Mosaik must support mechanisms that provide such a synchronization (**R 12** – *Synchronization of Control Mechanisms*).

3.4 Scope of mosaik

In Chapter 2.3 different aspects that are of interest when analyzing power systems have been presented. These include the operational constraints of the power system equipment as well as power system phenomena that can be observed on different time scales (see Figure 2.6). Dynamic phenomena such as electromagnetic transients or transient stability that need to be analyzed on a smaller time scale are usually modeled within special tools such as PSS/E, PowerFactory or PSLF (for transient stability) and ATP, EMTP-RV or PSCAD-EMTDC (for electromagnetic transients) [KN12]. These phenomena require continuous modeling and simulations are performed in the time domain. Time steps are in the range of microseconds and a simulation lasts less than a few seconds [HWG+06]. “Due to the limitation of computer storage and computation time, a complete representation of a large power system in an electromagnetic transients program is very difficult and won’t give additional information” [DA06]. Hence, these kinds of simulations are not well suited for the simulation of large scale scenarios. An exception are simulations for the analysis of subsynchronous resonance effects [AAV99, JK02] which require modeling a large part of the power grid for performing frequency scans, i.e. for determining the frequency at which such effects occur.[3]

On the other end of the scale ($\geq$ 1 second), the stationary power flow analysis (see Chapter 2.3.3) assumes a steady state of the system. In other words, the state of the simulation solely depends on the inputs and does not depend on previous system states and there are less dependencies between the modeled components. For example, there is no cyclic feedback between a wind turbine that generates power and the bus which it is connected to. This “lack of detail” speeds up the calculation process and allows to simulate larger scenarios. Models on this end of the time scale can be categorized as frequency domain models.

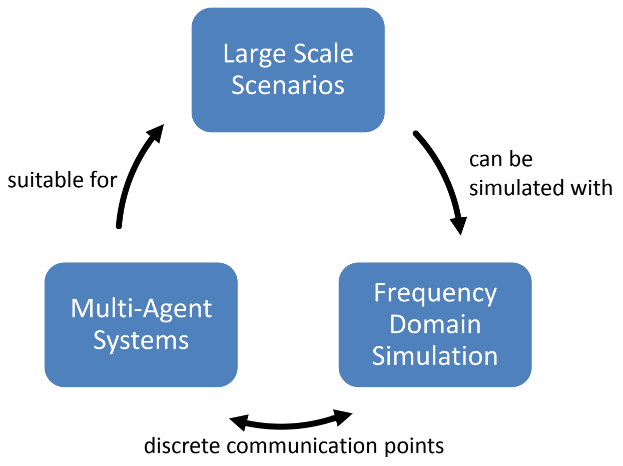


Figure 3.8: MAS and frequency domain models for large-scale simulation

As discussed in Chapter 2.2, multi-agent systems (MAS) offer a natural representation of the domain [JB03, p.67] and are thus well suited for large-scale control tasks. The

[3] Schöne, J.: “RE: RE: RE: [OPENSG-SIMSWG] SRS Framework”. Message to the author. 30 Sep. 2012. E-Mail

use of frequency domain models allow the simulation of large-scale scenarios and therefore seem to be well suited for the evaluation of MAS based control mechanisms. Furthermore, large-scale MAS for the Smart Grid are likely to rely on existing communication infrastructures (e.g. the Internet or in-home networks). The reasons for this are cost reduction and better adoption of new technologies [LAh11]. Hence, communication will cross multiple networks (WLAN, power line communication, Internet) to reach its destination. As a consequence there are longer delay times and the overall *Sense-Think-React* cycle of MAS (see Chapter 2.2) will also be on a larger time scale (from milliseconds to minutes). This is especially true for large scenarios where inter-agent communication (which is not part of the mosaik concept) takes place. Finally, communication of control algorithms with the simulated physical entities takes place at discrete points in time. This also matches well with frequency domain simulation where the models represent the system at discrete points in time. Figure 3.8 summarizes these arguments.

As the research focus at OFFIS is on multi-agent systems as well as phenomena that can be studied using frequency domain modeling (e.g. voltage stability, supply-demand-matching, offering products at electricity markets), the mosaik concept focuses on the composition of these types of simulation models (**R 13** – *Composition of Frequency Domain Simulation Models*) to keep the system as simple as possible [**R 1** – Keep It Simple].

Applicability to Time-Domain Simulations

Although the focus is on frequency domain simulations this does not prohibit the use of mosaik for the composition of more detailed (from the electro technical point of view) time-domain simulation models. However, in this case mosaik does not offer the best performance for large-scale scenarios as the simulation of time-domain models can be accelerated when not using a fixed-step size approach but rather a variable step size. To support this, more functionality than the mosaik simulator interface (SimAPI) presented in Chapter 6 offers is required. For example, it is required that the simulation models support *rollbacks*. The reason for this is that advanced variable step size algorithms need to execute the same simulation interval multiple times with different step sizes, allowing to assess the loss in precision caused by a larger step size [CASB12]. However, this also requires the simulators that are integrated to support this rollback.

Besides this performance aspect, the composition of time-domain simulation models also requires more complex scheduling algorithms as the cyclic dependencies between the models need to be solved iteratively. This complexity can be avoided when focusing on frequency domain simulation as the interconnections mostly have a distinct direction.

For the composition of continuous (time-domain) models, a new but already widely accepted standard is the Functional Mockup Interface (FMI) [MOD12b] which will also be discussed in Chapter 6.

3.5 Simulation Models

In this section available simulation models that have been developed in different research projects are presented and analyzed. Of course, the mosaik concept has to be applicable for a wider range of simulation models. As the individual simulation models do in most cases not lead directly to the definition of a new requirement there is no bias toward the particular simulation models. Rather, the model summary in Section 3.5.6 identifies a number of requirements derived from the different characteristics of the models. The summary section also contains a tabular overview of the different characteristics of the presented simulation models.

3.5.1 Photovoltaic Systems

Photovoltaics is a method to directly convert light into electrical power on the atomic level which had its first serious usage in the 1960s for powering spacecrafts [Kni02]. When light strikes a photovoltaic cell (or solar cell) and an electrical circuit is formed by connecting conducting equipment to both sides, electrical current begins to flow. As a single cell typically produces only a few watts, a number of cells can be connected in a serial fashion to form a PV module [CBA95, pp.44]. Modules, again, can be connected in a serial or parallel fashion to form a PV array. The amount of energy produced by such a PV array depends on a number of parameters. For example, the chemical and material related parameters as well as the module angle are relevant. But as these are fixed, they can be provided as model parameters and do not pose a special challenge for model composition. Solar irradiance, the azimuth of the sun or the environmental temperature are time variant parameters and directly influence the performance. As such environmental data (e.g. resulting solar irradiance and module temperature) may not be part of the PV model but rather be provided by specialized models (e.g. weather simulation) these are examples for data flows between composed models. The azimuth has impact on the spectral composition of the sunlight, affecting the energy the light can provide, while the solar irradiance and the angle of the module are both related to the amount of light the module is exposed to [KSW06].

For the Forschungsverbund Energie Niedersachsen (FEN) a PV model has been implemented in Matlab/Simulink[4]. As it is implemented as a continuous model it can theoretically be sampled with an arbitrary high resolution (very small-time steps). However, due to performance reasons and a lower resolution of other models, the PV model has usually been sampled in fixed discrete time steps of 1 or 15 minutes (i.e. used as time-discrete simulation model). The model provides the generated active power (in Watt) as output. A simple random weather model is part of the PV model. However, external weather data could also be used as input, overriding the values of a simple internal weather model. Doing so, the PV model could be used in combination with a climate model as already discussed in Section 3.2.3.

[4] `http://www.mathworks.de/products/matlab/` (accessed 18 Feb. 2013)

3.5.2 Electric Vehicles

Electric vehicles (EV) are seen as a key technology to reduce carbon dioxide emissions as electric engines are inherently more efficient than conventional combustion engines and they are “future proof” in the sense that their emissions are reduced automatically when electricity becomes cleaner [RVRJ12]. Furthermore, they reduce the dependency of the transportation sector on foreign oil imports and help to reduce local emissions in large cities. Different types of EVs exist [DH10]. In so called “mild hybrid” vehicles the electric engine only supports the combustion engine (which also charges the battery during low power times or when breaking). Plugin-hybrid electric vehicles have a larger battery, can be charged via connection to the power grid and also travel a short distance using electrical power only. Finally, pure electric vehicles have no combustion engine anymore and either have a battery or hydrogen powered fuel-cell as energy storage. Except for the hybrid vehicles all technologies need connection to the power grid.

As a consequence, EVs will become a considerable additional load for the power grid in the near future. This gets worse by the fact that vehicle charging will probably take place within certain periods of the day [SL10], thus creating high demand peaks in the distribution grid. Therefore, a controlled charging of the EVs may be required to avoid stressing the power grid. A large number of projects and studies that investigate such scenarios are available and still under development, such as [KEJ08], [NTS+11], [CHD09], [KT07].

But EVs do not necessarily place a burden on the power grid. They can also provide services to the power grid [KD06] as such vehicles can not only draw power from the grid but may also be used to feed power back to the grid if required [Bro02], [KT05]. This is commonly referred to as vehicle-2-grid (V2G). Therefore, an EV can be seen as power generator as well as power consumer. Furthermore, when equipped with a 4-quadrant charger converter, EVs can be used for voltage regulation without affecting the state of charge (SOC) of the vehicles [CT10]. In order to provide such grid services the vehicles need to be controlled.

In the project GridSurfer [TSS+11] a discrete-event EV simulation model that simulates the vehicle movements by using an empirical trip generator [SL10] has been developed and implemented using SimPy[5]. The finest temporal resolution is limited to 1 minute. Compared to other resources, EVs are not connected to a fixed point in the power grid topology but can have many different connection points. This has to be considered as a special requirement when defining scenarios including EVs (**R 14** – *Moving DER*). To determine the location of the vehicles in the power grid in case of this particular model, the model provides a location attribute which can have one of the values HOME, WORK, DRIVING, MISC. Currently the model only outputs the active power (in Watts). As the EV model supports V2G, the active power output can also be negative to represent power feed-in. Other outputs include the traveled distance (km), the state of charge (SoC in %) and whether the vehicle is plugged in or not. As input, the model accepts charging and discharging commands for certain time points in the future. This way an operating schedule can be submitted to the vehicle to influence its electrical behavior. Compared to the inputs and outputs of other models discussed above, such operational schedules

[5] `http://simpy.sourceforge.net` (accessed 27 May 2013)

require the transmission of complex data types (i.e. a data type that is composed of other existing data types) to the EV entities. The mosaik API for simulator integration must therefore support such complex data types (**R 15** – *Transmission of Complex Data Types*). In case of the EV schedule this is an array of structured primitive types indicating the type, duration and start point of a charging/discharging action. The reason for this requirement is that the entities offered by the simulators are not necessarily pure physical components but may expose a certain autonomy. In case of the EV this is the ability to automatically follow a given schedule. To allow the control of such "intelligent" objects complex data types must be supported.

3.5.3 Domestic Appliances

As mentioned above, the need for a more flexible energy demand is growing with the ever increasing number of intermittent renewable energy sources. As about 30% of Europe's electricity consumption can be traced back to the residential sector and especially to domestic appliances [SBP^{+}08, p.228], these should be considered in the future Smart Grid. The Smart-A project [SBP^{+}08], supported by the European Commission, assessed the load-shifting options for ten different types of appliances found across Europe. These included washing machines, tumble dryers, dishwashers, refrigerators, freezers, water heaters and so on. A number of aspects were considered and may be further analyzed in future simulation studies. These include the increased usage of locally generated renewable electricity, the load-shifting potential (in terms of time and energy amount) as well as additional energy consumption due to a "smart" operation. According to the type of controllability the appliances can be divided roughly into the following categories [KWK05]:

Shiftable operation devices This category includes batch-type devices. Once triggered, these run for a certain amount of time but the triggering time can be shifted to some extent. Examples are washing machines or tumble dryers.

External resource buffering devices Devices of this category produce some kind of resource other than electricity which is buffered in some way. This includes refrigerators, freezers, heatings or other thermal processes which need to keep the temperature within a certain limit.

Electricity storage devices These include dedicated storages based on conventional technologies, advanced fly-wheel or super-capacitor technologies. To be economic, they must be operated anti-cyclically in such a way that energy is buffered in times of low prices and fed back to the grid in times of high prices.

Stochastic/User-action devices From the DSM manager point of view these devices show stochastic behavior as their energy demand directly depends on the user actions. Examples for this class of appliances are audio, video, lighting or computers.

At OFFIS a number of domestic appliance models for some of these categories are available. These include dish-washers, refrigerators, freezers and washing machines. They were developed in the context of a PhD thesis [Lue12] which demonstrated a device type independent and stochastic control scheme that used these models to prove its feasibility. Again, the output includes the active power drawn from the power grid as well as device specific status information, e.g. the current temperature of a fridge. The inputs include device specific configuration parameters and load shifting commands. In case of the fridge this is the lower and upper temperature limit the fridge has to meet, for example.

3.5.4 Residential Loads

The simulation of the demand side (commercial, industrial and residential loads) is an important part of Smart Grid simulation as these account for a large part of the load that is put on the grid. Although an individual load can hardly be predicted the aggregation of a larger number of these can be predicted very well (law of large numbers) [Kam10]. For this reason energy suppliers and grid operators use different types of standardized load profiles to predict the load behavior of residential and commercial customers. The profiles usually contain samples in 15 minute intervals. Only for larger industrial customers individual profiles and actual measurements are reasonable as there are fewer of them.

The use of such standardized load profiles for Smart Grid simulation studies is common practice [Kup08, KWN$^+$10]. For this purpose the profiles are either scaled according to the assumed annual energy consumption of the respective customer (resulting in identical profile shapes but different amplitudes for the loads) or individual profiles are derived in a randomized fashion that sum up to the shape of the original load profile. The latter type of profiles has been provided to OFFIS by a project partner in the Grid Surfer Project [NTS$^+$11]. An alternative to synthetic load profiles is to use profiles based on real measurements [CHD09, TU 13]. These may provide smaller sampling intervals, e.g. 1 second in case of the ADRES data set [TU 13]. In any case, synthetic or based on real measurements, the data set provides values for discrete, equidistant time steps.

3.5.5 Power Flow Analysis

In Chapter 2.3.3 the stationary power flow analysis for calculating nodal (bus) voltages and branch currents was introduced. For past and current projects [NTS$^+$11, SNo12] an open source based approach, called GridSim, for stationary power flow analysis has been developed. The grid topology is described in the CIM XML/RDF format (see 2.4) and a path to such a topology file has to be defined as simulation parameter. For the load flow analysis, the open-source software package *Pylon*[6]. At simulation-time, the power that is fed-in or taken from each bus of the grid can be specified, the calculation can be triggered and power flows for each individual branch as well as the bus voltages can be retrieved. Figure 3.9 shows the relevant entities of the power flow simulation model that are of interest when performing a power flow analysis. For the mentioned projects this is usually done on a 1 or 15 minute basis. For these calculations a simplified single phase (symmetrical) representation is used, i.e. the power system is assumed to be balanced

[6] `https://pypi.python.org/pypi/Pylon` (accessed 10 July 2013)

among the three phases (see Chapter 2.3.3).

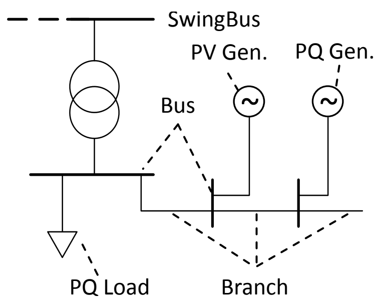


Figure 3.9: Elements of the available open-source power flow simulation model

The model of the power system that is maintained within the power flow analysis tool contains a number of different entities. In other words, as already pointed out in Section 2.1.1, a single model does not necessarily correspond to a single entity of the system it represents. As power flow analysis tools are usually configured by providing a power grid topology in form of some file (CIM files in case of the model presented in this section) the number of these entities a model contains is not fixed. As a consequence, the number of inputs and outputs provided by the power flow model also varies as these are entity specific (i.e. the model itself has no inputs and outputs but the entities have). Mosaik has to cope with such simulation models that contain more than one and especially a varying number of entities (**R 16** – *Variable Entity Quantities/Entity Specific Inputs/Outputs*).

In case of composed scenarios that are within the scope of this thesis, the actual generator or load parameters for each simulated time step are provided by simulation models that are external to the power flow simulation. The outputs of these models (e.g. active and reactive power) are the input for the corresponding entities in the power flow simulation. Special attention has to be paid to the modeling of the different power flow directions (e.g. in case of prosumers such as EVs that support V2G or other kinds of storages). One model may define that the direction of a power flow is implicitly encoded via the algebraic sign (e.g. generation is negative and power consumption positive or vice versa) while other models may use the opposite signs or a separate variable to indicate the direction. Mosaik has to deal with different encoding options (**R 17** – *Power Flow Encoding*). The outputs of a node may include voltage information (angle and magnitude) and the overall load (generation - consumption).

As the calculation of power flows and voltages is a quite complex task there are a lot of commercially available implementations that have more functionality and are known to deliver valid results. These may be used in the future instead of the open source solution. Therefore it can be said that mosaik must support the integration of Commercial Off-The-Shelf (COTS) simulators (**R 18** – *COTS Integration*). In general, such COTS simulators are popular because of their rich feature set, ease of use and their cost effectiveness [HWG⁺06]. With respect to the input/output behavior, power flow models can be categorized as memoryless systems (see Chapter 2.1.4). They have no state transition function and the output solely depends on the given input values (e.g. load, generation or switch positions).

3.5.6 Simulation Model Summary

In the last sections different simulation models that were available in the beginning of this thesis have been introduced briefly and analyzed with respect to their characteristics. As mentioned above, the mosaik concept has to be applicable to a wider range of simulation models, i.e. models that will be developed in future projects or supplied by other project partners. It is assumed that this will be possible as the analyzed models in this section have a wide diversity:

- They are implemented using different technologies, for example, Matlab as representative of a COTS tool or SimPy as a representative of a general purpose event-discrete simulation framework.
- They are modeled in the three major paradigms Discrete-Event (EV simulation), Discrete-Time (residential loads) and Continuous (PV).
- They have different temporal resolutions.
- They represent different types of Smart Grid actors, such as storages (EVs), consumer, producer and the power grid.

Table 3.1: Characteristics of currently available simulation models

Model	Temporal Resolution	Time	Implementation
EV	$\geq$ 1 minute	Discrete-Event	SimPy
PV	Continuous	Continuous	Matlab
Domestic Appliances	$\geq$ 1 minute	Discrete-Time	Python
Residential Loads	15 minutes	Discrete-Time	CSV-Data
Power Flow Model	None/Functional	n/a	Pylon

Table 3.1 gives an overview of these simulation models. The categories and the specific characteristics of the simulation models shown in the different columns of the table lead to a number of requirements. As different models have different temporal resolutions, mosaik must support the composition of simulation models with different temporal resolutions. This way a scenario can benefit from models with high temporal resolution (small step sizes) while at the same time allowing to use models with lower temporal resolutions (**R 19** – *Different Temporal Resolutions*). Further, the available simulation models employ different representations of time. Some are based on continuous models whereas most are implemented as discrete models using either the event-driven or time-stepped approach. Mosaik must be able to integrate simulation models with all of these different representations of time (**R 20** – *Different Representations of Time*). Finally, the available models are implemented using a number of different languages, tools and frameworks. Mosaik must therefore support the integration of simulators with heterogeneous implementations (**R 21** – *Heterogeneous Implementations*).

3.6 Compositional Requirements

This section deals with requirements related to the composition of simulation models that arise independently of a specific simulator, model or scenario.

3.6.1 Time Handling

Lin et al. [LSS$^+$11] point out that "especially in power system analysis, dynamic simulation runs in a step by step manner [...]." The analysis of the simulation models has shown one of the reasons for this: although there are continuous models available, the models for residential and other loads are usually based on synthetic or measured time series which contain data for fixed-length, discrete time intervals. As a consequence, the state of the power grid cannot be determined for arbitrary small intervals and the simulation runs in the mentioned "step by step manner". As the overall state of the simulation can only be determined for discrete points in time, the overall simulation can be classified as discrete simulation (see Chapter 2.1.3) although continuous models may be part of the composition. However, interacting with a simulator in a discrete-time fashion (as opposed to discrete-event) can have a number of disadvantages:

Synchronization overhead Not all state variables of a model change at every time step but synchronization takes place anyway [PT06]. A discrete-event based simulator coupling could improve on this by only communicating (raising an event) in case of state changes.

Missed events The opposite case can also happen: the fact that simulators may internally use a continuous or discrete-event approach and thus can simulate in a higher resolution than the discrete-time steps. As a consequence, critical events, e.g. unusually high load peaks may be undetected from the clients point of view. The fact than an EV charges for only 5 minutes may be completely unconsidered in a discrete-time simulation with 15 minutes step size, for example. If such missed events are critical or not, has to be decided by the Scenario Expert from case to case and the resolution in which a model is sampled has to be chosen accordingly.

Delayed events Besides the fact that a charging-event of an EV may be unconsidered, state changes in discrete-event models (e.g. a PV system stopping operation due to a defect) are delayed until the next sampling point. Again, whether such delayed events are critical or not has to be decided by the Scenario Expert from case to case and the step size has to be chosen accordingly. For the simulation studies that are within the scope of mosaik (see Section 3.4 and the step sizes of the discussed models (1 to 15 minutes) this issue is known and accepted by the researchers. It is a simple trade off between performance, modeling complexity and objectives of the simulation study.

On the other hand, the time-stepped approach "stands out through its simple simulation algorithm" [ZPK00]. This matches well with the need for simplicity [**R 1** – Keep It Simple]. Furthermore, there are many entities that change their state at (almost) every time step, for example loads, wind turbines and, as a consequence, also the power grid. The same holds for PV systems (except for the night when there are longer times with no generation). Also, EVs usually charge for a longer time, such that the missed event problem is likely to occur in rare cases only.

Finally, when implementing the time-step mechanisms in such a way that only value changes are exchanged between the simulators, a good compromise between discrete-event and time-stepped can be achieved. The participating simulators can internally benefit from the discrete-event speedup (e.g. the EV simulator) and thus rapidly detect if deltas have to be communicated when the get_data method is invoked. If this is not the case the method can return immediately without needing to exchange data and the overhead is very low.

For the above reasons, the mosaik concept will provide a mechanism to interact with simulators in a time-stepped fashion (**R 22** – *Discrete-Time Simulator Integration (fixed step size)*).

3.6.2 Formal Simulator Description

To integrate different simulators into an overall composition a well-defined, generic simulator interface (**R 23** – *Well-defined Simulator API*) is needed to control the simulators and access attributes of the simulated entities. As this API is generic for all simulators, an additional formal description is required to let the Scenario Expert as well as the composition engine know about the available elements and their semantics (**R 24** – *Formal Simulator Description*). Such a formal description of components that implement a generic API is common practice and can be found in the WebService domain [NM02] as well as in simulation specific standards (FMI/HLA, see Chapter 6). As discussed in Chapter 2.1.7, it also forms the basis to enable an automatic validation of the scenario the user has described.

3.7 Different Model Resolutions

In Chapter 2.1.2 the term resolution was defined as a description of the level of granularity (or detail) in which a model captures the system it represents. For the Smart Grid domain, Figure 3.10 shows different types of entities and some possible levels of detail. For the models discussed above, the respective level of abstraction is highlighted.

To achieve a maximal level of composability it should be possible to mix elements across different levels of detail. This is trivial, if the differences in detail only relate to the internal representation of a model. For instance it does not impact the composability if a PV model internally uses a simple sinus function or a more detailed climate model. When the level of detail influences the inputs and outputs, additional measures may be required.

Figure 3.11 shows such an example. The power flow of an EV entity is modeled in detail including the active (P) and reactive (Q) power share for all three phases. The bus entity to which the vehicle is to be connected is part of a less detailed power flow model using only a simplified single phase representation, such as the model presented in Section 3.5 above. As other entities of the scenario that is to be simulated may also only provide single phase power flow data, the usage of a more complex power flow

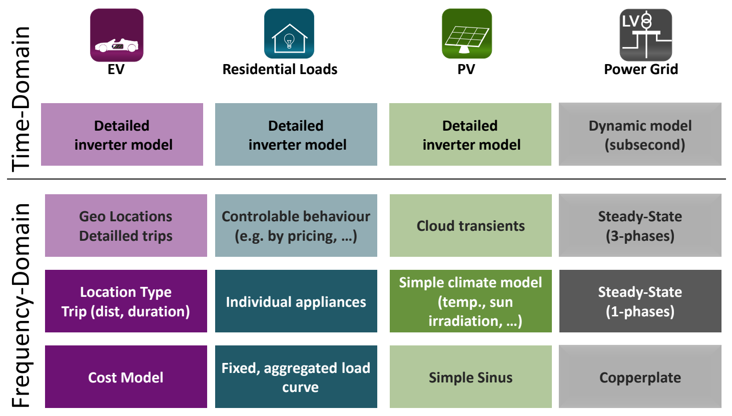


Figure 3.10: Entity types and different levels of detail

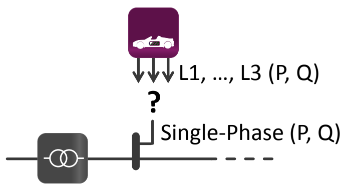


Figure 3.11: Model types and possible abstraction level

model may not be reasonable. However, in the example it is possible to bridge the detail gap by converting the 3-phased power flow of the EV to a single-phased equivalent flow to allow a composition. This can be done as follows, assuming that the phase angle (the share of reactive power) is distributed symmetrically among the three phases of the EV:

$$P_{simple} = \sum_{i=1}^{3} P_{Li} \quad Q_{simple} = \sum_{i=1}^{3} Q_{Li}$$

P_{simple} and Q_{simple} will then be the less detailed input for the bus entity of the single-phased power flow model. For the opposite direction (i.e. single-phased to 3-phased) a different calculation and assumptions about the distribution among the three phases (symmetrical or asymmetrical) have to be made. Or alternatively, a random connection to a single of the three phases can be made. This may be decided from case to case.

To support such multi-resolution compositions, mosaik must offer a mechanism for specifying such conversion algorithms (**R 25** – *Multi-Resolution Composition*).

3.8 Simulation Result Analysis

For being able to evaluate the performance of a control strategy, different metrics can be used. Although a metric could already gather the required data during a simulation run, all available entity data should be logged persistently (**R 26** – *Data Logging*) to allow the calculation of metrics after a simulation has completed. This way the metrics do not have to be defined in advance. The different metrics can be classified as follows:

Reliability Metrics These deal with interruptions, i.e. the complete loss of voltage, the most common of which are SAIFI, SAIDI and CAIDI, defined in IEEE Standard 1366 [KKOM04]. The CAIDI or customer average interruption duration index, for example, describes the average time it takes to restore the electricity supply at a customer site per interruption. For an in-depth overview of these metrics see Kueck et al. [KKOM04].

Power Quality Metrics Deal with the rating of power quality. According to Kueck et al. [KKOM04] the IEEE Standard Dictionary of Electrical and Electronics Terms defines power quality as "the concept of powering and grounding sensitive electronic equipment in a manner that is suitable to the operation of that equipment." As discussed in Chapter 2.3.2, power quality is ideal if "the voltage is continuous and sinusoidal, with a constant amplitude and frequency" [Lar12]. So these metrics describe how much the voltage and frequency deviates from the ideal values.

Sustainable Equipment Operation Metrics of this category rate the stress that is put on the power system equipment (e.g. to rate the stress a control strategy puts or takes from the equipment).

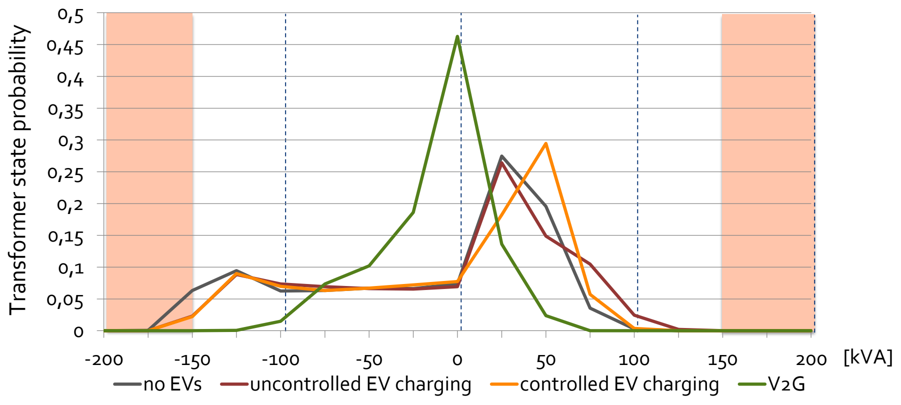


Figure 3.12: Probability density function describing transformer utilization

An illustrative example for a metric that rates the sustainability of equipment operation is a probability density function describing the utilization of a transformer[7]. Figure 3.12 shows this function for different EV charging strategies [SSW+10] that have been developed in the GridSurfer project [TSS+11]. The transformer load in kW is plotted on the x axis. The y axis indicates the percentage of time during which the corresponding load was placed on the transformer. The operational limits (see Chapter 2.3.1) of the transformer require the load to stay within certain limits (here +/-150 kW). To allow the automatic or manual interpretation of the resulting probability density curves, these limits need to be known. One way is to hard-code the values into the diagram. However, as mosaik has access to static entity data describing such operational limits [**R 11** – Static Data Access], it is much less error prone and more convenient to obtain these values also from the logged simulation data (**R 27** – *Logging of Static Data*). Finally, it has to be pointed out that further technical aspects of logging and analyzing the simulation data are not within the scope of this thesis.

[7] Further metrics of this category can be found in Schulte [Sch09]

3.9 Project Experiences

In the early stages of the GridSurfer project [TSS+11] the need for composing an overall simulation out of some of the aforementioned simulation models was detected. Different obstacles were identified during a first manual attempt to simulator coupling. These obstacles motivated this thesis and the development of an automatic approach for simulation composition so that the composition process can be done more easily, less error prone and hopefully faster. These obstacles are presented briefly in the following section and corresponding requirements for the composition concept are defined.

The first obstacle was the fact that the different simulation models were initially not developed with composability in mind. Hence, either no or at least no well-defined interface for accessing and controlling the simulation models was available. A standardized interface to achieve integratability [**R 23** – Well-defined Simulator API] was not available. Hence, in the beginning of the project the simulations were started independently and the produced results were gathered manually in form of single files. As long as no mutual influences between the simulated entities have to be modeled, this approach produces valid results. Nonetheless, this manual approach still had to cope with several other obstacles.

Due to license reasons the Matlab based PV simulation model (see Section 3.5.1) could not be executed on the PCs of all users/developers but had to remain on a remote server while the other simulation models could be executed locally. Hence, an automatic composition approach should be able to start simulator processes on different machines and integrate them into the composition (**R 28** – *Execution on Multiple Servers*), e.g. for license reasons. As a side effect, this feature can also be used to increase the speed of the composed simulation as the degree of parallelism is increased by distributing the simulations across different machines.

Another obstacle is related to the definition of the configuration parameters for the different simulation models involved in the composition. Usually, a model is being developed with a certain problem setting in mind. The model is intended to answer certain questions for the given problem setting. Often this focus on a certain problem setting limits the ability to reuse a model. Making a model parameterizable can increase the reusability by making the model adjustable to a wider range of settings [RÖ8]. The different simulation models all had their own location as well as format for specifying such parameters. This makes it a time consuming task to define the overall configuration of a simulation run as well as to keep track of the configuration when a few runs with different settings have been made. An automatic composition approach should therefore be able to provide a mechanism to define all possible and required parameters in a central place (**R 29** – *Central Parameter Definition*).

3.10 Summary

The following requirements have been elicited in this chapter and have been considered for the conception (Part II) and implementation (Chapter 11) of mosaik.

R 1	Keep It Simple
The mosaik system should be kept simple to account for the relatively short period of time that researchers and students have for learning and applying the system, while still allowing to reach the objective of the simulation studies.	

R 2	Scenario Specification Mechanism
Mosaik must offer a means to allow the formal specification of the scenarios that are to be composed out of the available simulation models.	

R 3	Control Strategy API
Mosaik must offer a dedicated API for allowing control strategies to interact with the simulated physical part of the Smart Grid in a standardized way.	

R 4	Scalable Scenario Definition
Mosaik must offer a mechanism to describe large-scale scenarios without much effort.	

R 5	Entity Dependencies
When defining a scenario, it must be possible to define connections between entities dependent on other connections that have already been defined before.	

R 6	Cyclic Data Flow Support
Mosaik must offer a way to deal with cyclic data flows.	

R 7	Attribute Based Composition
Mosaik must offer a mechanism to establish connections between entities based on their attributes. For example, a PV module should be connected to the corresponding cell entity of a spatial climate model based on its location attribute (if available).	

R 8	Physical Topology Access
Mosaik must allow the control mechanisms to query/traverse the physical topology created during the composition process.	

R 9	Intra-Model Topology Access
A simulator must offer the possibility of getting access to the relations between the entities of a model for building the complete physical topology.	

R 10	Combined Physical Topology
For the agents, the overall physical topology shall be made available in a transparent way, i.e. the agents should not be aware of model internal and mosaik scenario based topological relations.	

R 11	Static Data Access
Whenever possible, the simulation models shall provide information about the static properties of the entities to mosaik.	

R 12	Synchronization of Control Mechanisms
Mosaik must offer a way to let the control strategies be aware of the simulation time.	

R 13	Composition of Frequency Domain Simulation Models
Mosaik has to support the composition of frequency-domain simulation models.	

R 14	Moving DER
Mosaik must support the specification of scenarios with moving resources (e.g. EVs).	

R 15 Transmission of Complex Data Types

It must be possible to submit (and receive) complex data types from the entities of the participating simulators, for example to support the transmission of operation schedules and other complex structures.

R 16 Variable Entity Quantities/Entity Specific Inputs/Outputs

Mosaik must be able to cope with models that have a variable number of entities, e.g. depending on the parameter configuration of the model.

R 17 Power Flow Encoding

Different approaches for encoding power flow data have to be considered when developing the composition approach.

R 18 COTS Integration

Mosaik must allow the integration of COTS simulators.

R 19 Different Temporal Resolutions

Mosaik must allow to compose simulation models with different temporal resolutions.

R 20 Different Representations of Time

It must be able to compose simulation models that use different paradigms with respect to the handling of time.

R 21 Heterogeneous Implementations

It must be possible to integrate simulators with different implementations (i.e. frameworks, languages, tools).

R 22 Discrete-Time Simulator Integration (fixed step size)

Mosaik has to be able to step simulators in a time-stepped fashion with fixed step size.

R 23 Well-defined Simulator API

A standardized, generic API is the basic requirement for achieving any other interoperability and composability objectives.

R 24 Formal Simulator Description

Mosaik must provide a formal simulator description to automatically infer information about each simulator (and its models) that implements the standardized API. As pointed out in chapter 7 this information must be available prior to run-time to achieve composability as defined by Definition 5.

R 25 Multi-Resolution Composition

Mosaik shall support the composition of models with different levels of detail.

R 26 Data Logging

Mosaik must log all data that is provided by the simulators for later evaluation.

R 27 Logging of Static Data

The logged data must include the static data of the different entities.

R 28 Execution on Multiple Servers

Mosaik must be able to start and execute simulators across multiple computation nodes (simulation servers). These servers may have different operation systems such as Linux, Windows and OS X, which need to be supported. It has to be noted that this requirement is different from [**R 21** – Heterogeneous Implementations] as a single simulation server (e.g. running Linux) can include multiple simulation platforms, such as Matlab or Python based simulators.

R 29 Central Parameter Definition

The parameters for the different simulators that are part of the composition must be defined in a consistent and centralized fashion.

4 State-of-the-Art in Smart Grid Simulation

In this chapter related works from the field of Smart Grid simulation (that is simulations involving power system and control aspects) are being discussed. Related works from other simulation domains may also be applicable to the field of Smart Grid simulation composition. But to increase cohesion, these are discussed when the different parts of the mosaik concept are being presented in Part II.

In recent years a number of approaches toward Smart Grid simulation have been published. These were driven by the need to determine the behavior and impact of new Smart Grid technologies (see Chapter 1.1) such as DER, energy storage solutions and different control mechanisms. The diversity of simulation approaches and Smart Grid technologies also leads to a variety of categories which can be used to classify these works. For the purpose of this section, the related works are split into the two broad categories "Single-Simulation Approaches" and "Co-Simulation Approaches". Classical power system analysis tools (such as PowerFactory[1], NETOMAC[2] or PSAT[3]) will not be discussed explicitly in this section as they do not deal with the composition of existing models into an overall simulation but rather are examples of power system models from the perspective of this thesis. However, such tools are involved in the Co-Simulation approaches category discussed below. All approaches will be discussed with respect to the different aspects of the research questions (Chapter 1.3) that this thesis tries to answer. These aspects are:

Integratability To what extent does the approach offer support for integrating existing simulation models (that are potentially implemented in a different technology) into an overall simulation [**R 23** – Well-defined Simulator API]?

Interoperability & Composability Does the approach offer a formal description of the simulator and its models, in order to facilitate automatic composition [**R 24** – Formal Simulator Description]?

Formal scenario description Does the approach offer a mechanism to formally describe scenarios [**R 2** – Scenario Specification Mechanism] or do these have to be defined by changing the implementation of the participating components? If so, is a mechanism defined that is scalable with respect to scenario size [**R 4** – Scalable Scenario Definition]?

Evaluation of control strategies Does the approach offer a way to integrate and thus evaluate control strategies [**R 3** – Control Strategy API], [**R 12** – Synchronization of Control Mechanisms], [**R 8** – Physical Topology Access]?

As will be shown, no approach can meet all the requirements but a few approaches offer good ideas with respect to single requirements. These approaches will be picked up again when the corresponding part of the mosaik concept is being presented in the different chapters of Part II.

[1] `http://www.digsilent.de/index.php/products-powerfactory.html` (accessed 4 May 2012)
[2] `http://www.netomac.com/pubdown.htm` (accessed 11 Feb. 2013)
[3] `http://www3.uclm.es/profesorado/federico.milano/psat.htm` (accessed 18 Feb. 2013)

4.1 Single-Simulation Approaches

Approaches of this category are characterized by the use of a single simulation tool to analyze Smart Grid aspects. Other tools may be used for data pre- or post-processing (i.e. analysis of the results) but the approach itself relies on a single simulation tool.

4.1.1 GridIQ

As part of his bachelor thesis, Bankier [Ban10] developed this open-source framework for interfacing existing multi-agent platforms with power system simulation tools. GridIQ acts as a bridge between the agent platform and a power simulation tool. It provides an abstract framework which has to be adapted to specific tools by implementing corresponding adapter classes. The framework is being demonstrated by coupling PSAT for power flow simulation and JADE [Tel12] as agent platform. Figure 4.1 shows this setup.

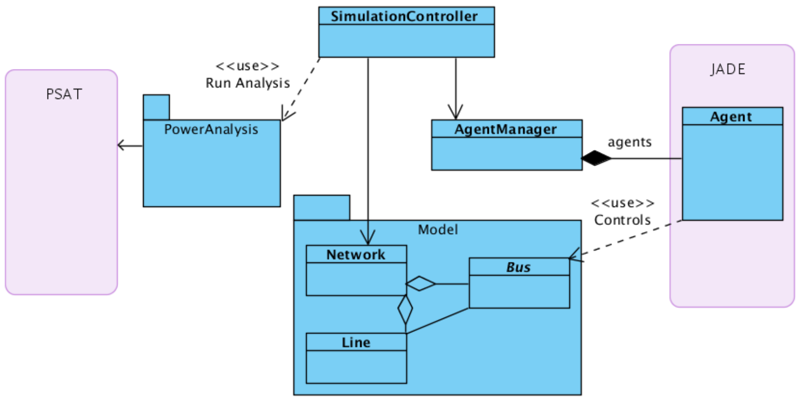


Figure 4.1: Key components of the GridIQ framework [Ban10]

Integration & composition

GridIQ is discussed here and not in the co-simulation section as the JADE MAS is not a simulation and hence, GridIQ does not couple two simulation tools but rather uses a power system simulation as test bed for MAS-based control strategies. As a consequence, GridIQ does not meet the requirements related to the composition of simulation models. A *Disturbance* class can be used to specify load profiles for certain busses at different time steps. This data can be read from arbitrary sources (e.g. CSV files) but this is not a mechanism for integrating external simulators.

Formal scenario description

Obviously, there is no formal scenario definition available. The physical aspects of the Smart Grid are to be specified in simulation tool specific topology files. There is an XML based project description but besides some configuration information this is currently limited to assigning agents statically to busses. There is no possibility for a dynamic bootstrapping process.

Evaluation of control strategies

All JADE agents have to be derived from the base class *GridIQAgent* which allows the agents to interact with GridIQ [**R 3** – Control Strategy API] but at the same time limits the agents to be used with GridIQ only. The agents have no information about the grid topology [**R 8** – Physical Topology Access] and cannot form corresponding structures (e.g. see Chapter 3.3). A simple synchronization between the power system simulation and the MAS is achieved by assuming a fixed behavior of the agents. They are notified by the framework after the power system simulation has finished a time step. Next, the MAS performs its actions (which are assumed to be timeless or to behave in real-time, as JADE has no explicit notion of time) and the control is explicitly given back to the framework. Assuming that the agents are aware of the step size of the power system simulation the synchronization requirement [**R 12** – Synchronization of Control Mechanisms] is fulfilled.

4.1.2 GridLAB-D

GridLab-D [Pac13a] is an open-source power distribution system simulation and analysis tool developed by the US Department of Energy at Pacific Northwest National Laboratory (PNNL). Version 2.2.0 has been released in September 2011 and the project is still under active development. More than 5.000 downloads of this version [Sou13] and more than 10.000 downloads overall [Pac12] indicate that the tool is widely known and addresses a wide range of modeling issues. In fact, GridLAb-D provides models from four important domains, namely power systems, control systems, buildings and markets. These models allow to model not only the power system but also the other relevant systems that directly or indirectly influence the power system. Each of these models is described by multiple differential equations integrated into the GridLAB-D simulation engine and solved independently. Chassin et al. [CSG08] claim that the advantages of this algorithm over "traditional finite difference-based simulators" are its increased accuracy, the ability to handle widely disparate time scales (from sub-second to many years) as well as the ability to easily integrate with new modules and third-party systems. The reason for the latter is "that it is not necessary to integrate all the device's behaviors into a single set of equations that must be solved" [CSG08].

Integration & composition

Although this ability to easily integrate third-party systems is mentioned, no explicit interface for this is presented. However, the process to integrate "other simulation modules" into GridLAB-D and the requirements that these components must meet are informally discussed [CW09]. Due to this fact and as the tool is open-source, requirement [**R 23** – Well-defined Simulator API] can probably be met by adding a well-defined API for simulator integration. But what is still missing in this case is the ability to formally describe the integrated simulation models [**R 24** – Formal Simulator Description]. Features to use GridLAB-D in a co-simulation manner together with a commercial transmission system simulator as well as an open-source communication system simulator are announced for version 3.0 which is scheduled to appear in the year 2013 [Pac12].

Formal scenario description

Among the simulation approaches discussed in this chapter, GridLAB-D has the most advanced mechanism to specify the scenario that is to be simulated. The user uses so called GLM (Grid Lab Model) files to describe the simulation setup [Cha09]. These files have a C-like syntax and are easy to read as they only have little markup overhead compared to an XML based representation, for example. The first part of a GLM file contains the definition of the simulation time range. The resolution is fix at one second. Next, the required modules containing the different classes of object types that GridLAB-D can simulate are included. These classes can then be instantiated with specific properties using the *object* keyword. The properties can either be constant values or randomly sampled using different distribution functions that are available. Also, it is possible to specify hierarchies of objects, e.g. a house object with a dishwasher and other consumers as child objects. Custom classes can be modeled inside a GLM file but require the installation of a C/C++ compiler as these are not interpreted but compiled into the GridLAB-D simulation engine [Pac09]. Although there are some more features, this is already sufficient to say that requirement [**R 2** – Scenario Specification Mechanism] is met.

With respect to scalability (i.e. describing large scenarios with few lines of code), GridLab-D offers the ability to include other GLM files using a macro directive (i.e. a pre-processing step merges the referenced file into the file using the macro directive). This allows the reuse of models (e.g. grid topologies) and avoids redundancies. Furthermore, it is possible to instantiate an arbitrary number of instances with a single *object* statement by specifying a range of IDs for these objects [Pac13b]. Listing 4.1 contains an example from the GridLab-D documentation illustrating how this feature is used.

Listing 4.1: GLM object definition example [Pac13b]

```
object node:1 {
    type 3;
    name "Feeder";
}

//Create 10 objects of class office and connect them to the feeder
object office:1..10 {
    parent Feeder;
}

//Create 10 recorder objects and connect each to 1 office object
object recorder:..10 {
    parent office:*;
}
```

First, a *node* object (representing a bus in the power grid) is created. Next, ten objects of the class *office* (representing an office building, available in the commercial load module of GridLab-D) are instantiated. As the ID is specified as a range (1..10), these will have the IDs *office:1* through *office:10*. Finally, ten *recorder* objects (which can be used to log data values into a CSV file, for example) are instantiated. Each of these is assigned to one office object using the parent property and the *<class>:** syntax. This has "the effect of using the first as-yet childless object with that name" [Pac13b]. However, once used, this syntax cannot be used again for the same objects. To sum up, it can be said that GridLAB-D has the most advanced scenario specification mechanism among the discussed works, although it is used to model the GridLAB-D internal models and not to compose models from external simulators. The inclusion of other GLM files and the ability to create a range of objects allow for a certain grade of scalability but this functionality is still very limited (see Chapter 8 for a more detailed discussion). Hence, requirement [**R 4** – Scalable Scenario Definition] is only partially met.

Evaluation of control strategies

The statement that "GridLAB-D provides a valuable testbed for evaluating control strategies and cost-to-benefits ratio" [Pac12] is certainly true. However, the control mechanisms are assumed to be modeled within GridLAB-D. There is no API for the easy integration of external control mechanisms [**R 3** – Control Strategy API]. As a consequence [**R 12** – Synchronization of Control Mechanisms] as well as [**R 8** – Physical Topology Access] are not met.

4.1.3 Homer

HOMER is a tool for evaluating different designs of off-grid and grid-connected power systems [Nat11] that has been developed by the U.S. National Renewable Energy Laboratory. The user can add different power resources (e.g. wind turbines, CHP plants, diesel generators) to the grid and provide a range of parameters for these resources. HOMER then simulates the behavior of these resources for all hours in a year by making energy balance calculations. The tool allows to automatically simulate all valid parameter combinations to explore the configuration space. A detailed analysis of the results is supported by different configurable table and graph views.

Integration & composition

The list of available resources the user can choose from for his or her scenario is fixed. It is not possible to integrate different models available in other simulation frameworks/packages into the simulation.

Formal scenario description

HOMER is primarily suitable for economic studies [Nat11] as the electrical grid is not modeled in detail. It is possible to model different voltage levels using a transformer resource but power lines and other assets (such as protection equipment) cannot be modeled. The configuration of a scenario is done via a graphical user interface by manually adding the desired components [**R 2** – Scenario Specification Mechanism]. But as a consequence, the creation of large scenarios requires much effort and [**R 4** –

Scalable Scenario Definition] is not met.

Evaluation of control strategies

The control strategies used by HOMER cannot be set by the user [BGL+04]. Thus, it cannot be used to evaluate new control strategies.

4.1.4 IDAPS

The Intelligent Distributed Autonomous Power System (IDAPS) is a multi-agent based concept for an intelligent microgrid[4] [PFR09]. IDASPS is comprised of four different types of agents implemented with the ZEUS agent toolkit [NNL98]. A control agent is responsible for monitoring power quality and can control a circuit breaker to isolate the microgrid from the utility if an outage is detected. DER agents represent distributed energy resources that are part of the microgrid and provide information about their states to other agents. User agents make features of IDAPS available to actual users (e.g. allowing the user to specify load priorities or to monitor the consumption of the loads). Finally, a database agent is responsible for persisting system information and exchanged messages. In order to demonstrate that the IDAPS system works properly in case of an upstream outage, a simulation model of the microgrid and its components has been implemented in Matlab. It consists of a generator, loads and load circuit breakers, a distribution transformer, the main circuit breaker and the utility grid. The agents are executed on a separate computer. At runtime they interact with the simulated components via a TCP/IP connection. Figure 4.2 shows this setup.

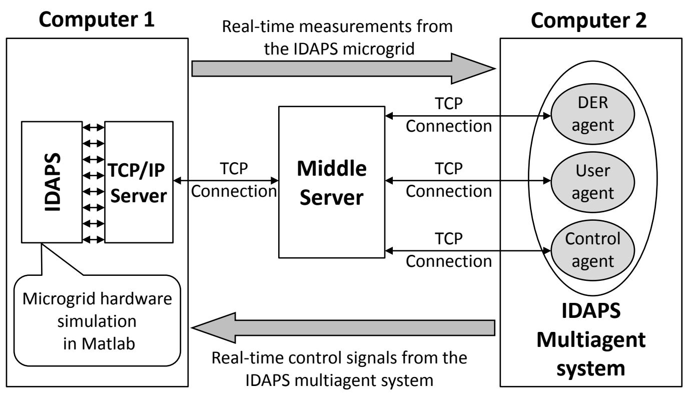


Figure 4.2: IDAPS Simulation Setup (according to [PFR09])

Integration & composition

Similar to GridIQ, IDAPS is not considered a co-simulation as the agent system is not a simulation. The approach does not meet the requirements related to the composition of simulation models.

[4] “A microgrid is an integrated energy system consisting of interconnected loads and distributed energy resources which as integrated system can operate in parallel with the grid or in an intentional island mode.“ [Agr06]

Formal scenario description

The IDAPS approach does not offer any means to scenario definition as the power system model is entirely created in the Matlab environment and the connection between agents and the simulated assets is hard-coded in the agent system.

Evaluation of control strategies

The used TCP based connection is a manual, tool specific interface to the simulated assets. From the mosaik perspective this is not an API for simulator integration but rather an API for evaluating control strategies [**R 3** – Control Strategy API]. However, the agents seem to have no ability to get information about the grid topology [**R 8** – Physical Topology Access] and cannot form corresponding structures (e.g. see Chapter 3.3). Synchronization of time between the agents and the simulated power system [**R 12** – Synchronization of Control Mechanisms] is not required as both systems are running in real-time. While this enables realistic simulations it does not allow to execute simulations faster than real-time which is possible with GridIQ, for instance.

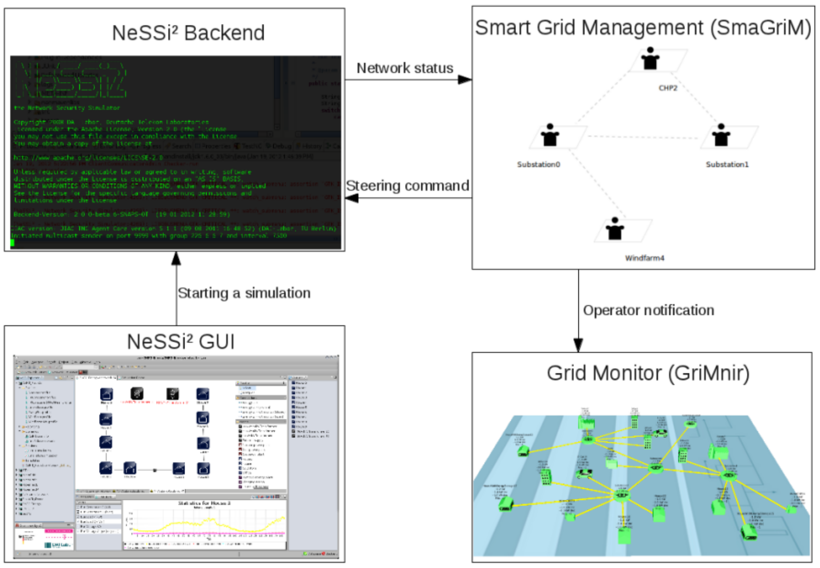


Figure 4.3: Overview over the ILIas framework [KGC+12]

4.1.5 ILIas

The ILIas framework [KGC+12] uses the open-source (communication) network simulation environment NeSSi2[5]. Although it is the main focus of NeSSi2 to act as a realistic packet-level simulation environment [GS12], the developers have shown that it can also be used for power system simulation [CBS+11] as it is based on an abstract graph model that allows domain specific extensions. For the power system domain this graph model has been extended to offer the link type *power transmission line* and the node types *producer*, *consumer*, *prosumer*. Furthermore, the load flow model of

[5] http://www.nessi2.de (accessed 29 May 2013)

the InterPSS project[6] has been integrated. The ILIas framework uses the resulting simulation system as a testbed for an agent-based Smart Grid management approach called SmaGriM. It is a hierarchical approach with two different types of agents. Reactive agents are representing the device nodes of the network and special transformer agents are responsible for managing the network. These agents monitor the state of their node agents and can also negotiate with other transformer agents for exchanging power. They can also communicate with the device agents which can, depending on the type of device, shed the load, control the production, control switches or store energy. The agents are implemented using the JIAC V framework[7] as this is also the basis for the NeSSi^2 simulator. Figure 4.3 shows the resulting setup.

Integration & composition

The ILIas framework does not offer a mechanism dedicated to the integration and composition of different simulation models. Although an HLA-like federation mechanism (discussion of HLA in Chapter 6.2) is mentioned [CBS$^+$11], it seems to be limited to the creation of projects with multiple networks and a mapping of nodes between these networks [GS12].

Formal scenario description

The NeSSi^2 environment offers different graphical editors. The *network editor* allows to use the available link and node type to create a network graph (e.g. a communication network or a power system topology). The *scenario editor* allows then to map so called profiles to each node which govern the behavior of the nodes (e.g. a standardized BDEW demand profile [(BD13] for simulating a residential load [CBS$^+$11]). So there is a modeling formalism for creating the physical topology of the Smart Grid [**R 2** – Scenario Specification Mechanism] but it is not based on composing existing simulators and as each node and link has to be created manually, the creation of large scenarios requires much effort. Hence, requirement [**R 4** – Scalable Scenario Definition] is not met.

Evaluation of control strategies

The agents of SmaGriM interact with special connectors with the simulated devices in NeSSi^2. This allows to exchange the connectors, for example to integrate one or more real physical devices into the agent system without affecting the agents [KGC$^+$12]. Although not explicitly mentioned, the interface between the connectors and the agents must be well-defined to enable such a functionality, corresponding to [**R 3** – Control Strategy API]. However, it is not mentioned that the agents can dynamically access the topology of the simulated power grid [**R 8** – Physical Topology Access]. The mapping from network elements to agents seems to be done via the network or scenario editor as well. How the time of the simulation advances (fixed steps, etc...) is not explicitly mentioned. To avoid that the simulation runs faster than the agent system (which has no explicit notion of time), the simulator does only advance if it has received a command from all agents. Similar to GridIQ, if one assumes that the agents are aware of the

[6] `http://community.interpss.org` (accessed 29 May 2013)

[7] `http://www.jiac.de/agent_frameworks/jiac_v/` (accessed 29 May 2013)

step size of the simulator, the synchronization requirement [**R 12** – Synchronization of Control Mechanisms] can be seen as fulfilled.

4.1.6 IPSYS

The analysis of autonomous power systems with a high share of renewable energy sources is the main purpose of IPSYS [BGL$^+$04]. As such systems can be quite complex with respect to configuration options and operation strategies, IPSYS claims to allow flexible modeling of control strategies. It features a full load flow analysis including active and reactive power share and allows time steps in the order of a few seconds.

Integration & composition

Although IPSYS allows the user to integrate new control mechanisms, these have to be hard-coded into the overall system. There is no support for integrating external simulation models into IPSYS and subsequently composing these.

Formal scenario description

Controller parameters can be specified in configuration files. A scalable and formal mechanism to define the grid topology and the participating components is not mentioned.

Evaluation of control strategies

How the user can implement and integrate control mechanisms is not described in detail. At least there is no API that can be used externally to evaluate control mechanisms implemented on other platforms.

4.1.7 Smart Cities

The Smart Cities approach is a “simulator based on software agents that attempts to create the dynamic behavior of a smart city” [KN09]. The simulation is intended to serve as a basis for further testing of DER management algorithms, business models and behavior analysis. The current objective is the evaluation of the chosen simulation approach for Smart Grid infrastructures rather than the evaluation of a particular management algorithm. All simulated entities are implemented as agents using the JADE platform. They offer a web interface to interact with them from outside the platform.

Integration & composition

The approach does not consider the integration of existing models but rather requires to model all entities of a Smart Grid scenario as agents on the JADE platform. So none of the requirements from this category are met.

Formal scenario description

It is mentioned that the simulator gets its initial values and scenario from a database but the details about how a scenario is described are not mentioned.

Evaluation of control strategies

Although it is stated that services offered by the agents (e.g. turning the device represented by the agent on or off) can be invoked via a web-service interface, it is not specified if the service methods are standardized or dependent on the agent implementation [**R 3** – Control Strategy API]. Details on how external control mechanisms using this interface are synchronized with respect to simulation time are not given [**R 12** – Synchronization of Control Mechanisms]. Finally, it is mentioned that "logical limits" such as city boundaries can be defined, but not if a topological power grid model is used. Thus, [**R 8** – Physical Topology Access] is inapplicable and therefore not met.

4.2 Co-Simulation Approaches

"Co-simulation [co-operative simulation] is an approach for the joint simulation of models developed with different tools (tool coupling) where each tool treats one part of a modular coupled problem. Intermediate results (variables, status information) are exchanged between these tools during simulation where data exchange is restricted to discrete communication points. Between these communication points the subsystems are solved independently" [BCWS11].

So, from the functional point of view, a co-simulation is a simulation that is comprised of different models that are simulated by different simulator processes. But this does not imply that co-simulations are necessarily composable in the sense of the composability definition (Def. 5) by Petty and Weisel [PW03a]. The reason for this is that a formal simulator description [**R 24** – Formal Simulator Description] and/or the ability to define different scenarios [**R 2** – Scenario Specification Mechanism] is not intrinsic to co-simulation approaches. But these features are required to have the ability to easily assemble the participating simulation components in different and especially valid combinations as demanded by the composability definition.

The Smart Grid combines the two domains of telecommunication and power systems. However, "[...] neither stand-alone power system simulation nor stand-alone communication network simulation is sufficient to precisely model a fully interconnected power system and communication network" [Lin12]. As mature tools for both of these domains exist, it is a natural approach to couple these tools. All works discussed in this section deal with approaches for achieving a co-simulation of power system and communication network simulators. It is important to point out that such a coupling is not within the focus of the mosaik concept presented in this thesis. Rather, the mosaik concept aims to couple different power system simulators and simulators including power system component models. Figure 4.4 points out this role of mosaik by showing that the resulting mosaik simulation platform can be used as one component of a communication co-simulation but solves a different problem: The composition of power systems based on existing component models and not the co-simulation of specialized

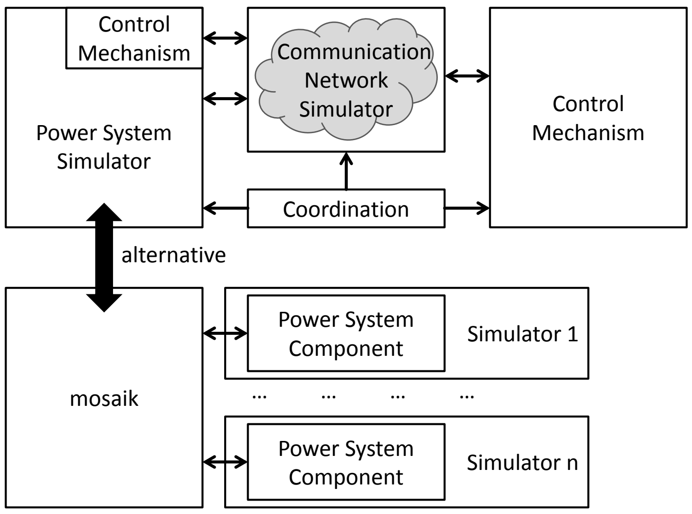


Figure 4.4: Abstract communication co-simulation framework and mosaik

tools from different domains. Due to this fact, requirements regarding the formal scenario description ([**R 2** – Scenario Specification Mechanism] and [**R 4** – Scalable Scenario Definition]) cannot be met by these tools. With the exception of two noteworthy contributions, namely EPOCHS (which pioneered the Smart Grid co-simulation efforts [Lin12]) and GECO (which was the latest available contribution by the time this thesis was written) they are only discussed briefly.

4.2.1 EPOCHS

Among the works discussed in this chapter and according to Hopkinson et al. [HWG+06] as well as Lin [Lin12], the electric power and communication synchronizing simulator (EPOCHS) [HWG+06] is the first (i.e. oldest) co-simulation approach that offers a combined simulation of communication and power system aspects. It relies on non-intrusive techniques to integrate the participating simulators into the co-simulation. The feasibility of the approach is demonstrated by integrating the tools PSCAD/EMTDC [PQ 13], PSLF [Gen13] and ns-2 [Uni13b]. These components allow the users to either simulate electromagnetic scenarios involving communication by coupling PSCAD/EMTDC and ns-2 or to simulate electromechanical scenarios using PSLF and ns-2.

A custom middleware called RTI is used for synchronization and message exchange. A self-made, discrete-event based multi-agent environment called AgentHQ has been developed especially for the EPOCHS approach and can be used to model multi-agent control strategies [TWH+03]. Using AgentHQ and the RTI, the agents can get and set power system values and can send and receive messages to each other. Figure 4.5 shows this architecture. According to Bergmann et al. [BGG+10] the EPOCHS software setup "is limited to simulate only a few scenarios in the area of protection of power transmission in the high and medium voltage range."

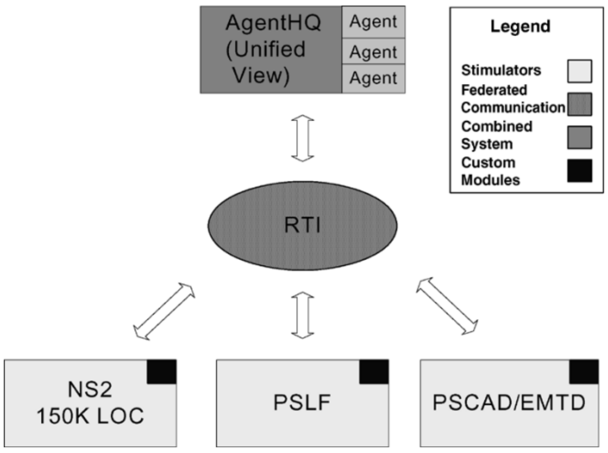


Figure 4.5: Architecture of the EPOCHS approach [HWG+06]

Integration & composition

The EPOCHS approach is the only one that seems to define an interface for integrating simulators [**R 23** – Well-defined Simulator API]. Although this interface is neither described in detail nor mentioned explicitly, the fact that a middleware (the RTI) is used leads to the natural assumption that this middleware offers a well-defined interface that is used to integrate the simulators. But it is not sufficient to ensure validity when being used for composing power system component models, as a formal description of the interfaced simulators and their models [**R 24** – Formal Simulator Description] is missing. However, the EPOCHS approach shows that "non-intrusive techniques can be used to federate simulation engines using only their built-in application programming interfaces (APIs)" [HWG+06]. Possible approaches for integrating simulators will be discussed in Chapter 6.

Formal scenario description

As mentioned above, the definition of scenarios as required by [**R 2** – Scenario Specification Mechanism] is not part of any of the co-simulation approaches. These assume that the power system is modeled in the respective simulator that is used.

Evaluation of control strategies

The EPOCHS approach explicitly mentions an agent interface. It is split into two types of methods. Event callbacks are used to notify the agents if messages arrive or the power system has finished its next time step. Communication methods are used to send messages to other agents or to issue control comments to the power system. Thus, the requirements [**R 3** – Control Strategy API] and [**R 12** – Synchronization of Control Mechanisms] are met. If it is possible to access the topology of the power system remains unknown but could certainly be added [**R 8** – Physical Topology Access].

4.2.2 GECO

Lin et al. [LSS+11] argue that the Global Event-driven Co-simulation (GECO) "approach improves the EPOCHS approach in accuracy while keeping intact the idea of federation [...]". Thus, similar to the aforementioned EPOCHS approach, PSLF and ns-2 are used as examples to demonstrate the GECO approach. The accuracy can be improved as ns-2 is extended to act as a global event scheduler. That is, the power system iterations (solving) are integrated as events into the ns-2 event list. The EPOCHS approach only synchronized the two simulators at larger time steps leaving the integration time steps of the dynamic power flow solver internal to PSLF. With respect to mosaik, the requirements are similar to the ones of the EPOCHS approach. For a detailed description of the GECO approach see Lin [Lin12].

4.2.3 Other approaches

There are a number of other co-simulation approaches. But as these use tool-specific couplings and not a federate approach, such as EPOCHS, these are of limited use for the mosaik concept and discussed only very briefly.

ADEVS Nutaro et al. [NKM+07] and more recently Nutaro [Nut11a] present a discrete-event based "technique for integrating continuous models of power system dynamics and discrete event models of digital communication networks." Again, ns-2 is used as network simulator and continuous power system components and control/monitoring components have to be modeled manually using the *adevs* framework. This framework specific modeling prevents the easy use of most commercial power system modeling tools as these are not discrete event based [Lin12].

Bergmann et al. [BGG+10] An external API for the power system simulator NETOMAC is being presented. It allows to monitor and control certain simulated objects in an IEC 61850 (see Chapter 2.4) compliant way. Java based control applications can then interact with these objects via a communication network simulated using ns-2. The use of an energy standard based API (here IEC 61850) is an interesting and promising approach as it enables an easy recombination of power system models and control mechanisms. Thus, requirement [**R 3** – Control Strategy API] is met.

Godfrey et al. [GMD+10] Here, a co-simulation involving a stepwise cyclic interaction between ns-2 and OpenDSS is presented. The scenario that is being simulated includes the wireless control of stationary storages for buffering strong PV gradients caused by cloud transitions.

PowerNet Liberatore and Al-Hammouri [LAh11] present a co-simulation of ns-2 and Modelica where ns-2 events govern the time-steps in which the Modelica models are solved. Power system as well as control modeling takes place in Modelica and ns-2 is used to determine the communication delay in the control loop.

VPNET Li et al. [LMLD11] have created a co-simulation with power system aspects being modeled in the tool VTB [Uni13a] and the communication delay in the control loop is modeled using ns-2, similar to the idea of the PowerNet approach.

4.3 Summary

Although the field of Smart Grid simulation is quite a young discipline, a number of approaches have already been developed and published in the recent past. This emphasizes the need for new Smart Grid specific simulation concepts, such as the one presented in this thesis.

Table 4.1 summarizes the discussed approaches regarding the degree to which they meet the major requirements listed in the beginning of this chapter. It can easily be seen that there is no approach dedicated to the composition of Smart Grid simulation models. Therefore, it is sufficient to rate the fulfillment degree on a simple scale without having to go into the details of the different requirements. This scale includes the symbols *n/a (not applicable as requirement not in the focus of the approach), - (requirement not met), + (requirement met or at least partially met)* and *? (unknown/not enough information).* Also, a flexible and scalable scenario specification mechanism is missing. Such a mechanism will be the main contribution of this thesis.

Table 4.1: Summary of related Smart Grid simulation approaches

Approach	Composability		Scenario Formalism		Control Strategies		
	R 23	R 24	R 2	R 4	R 3	R 12	R 8
GridIQ	-	-	-	-	+	+	-
GridLAB-D	(+)[8]	-	+	(+)[9]	-	-	-
HOMER	-	-	+	-	-	-	-
IDAPS	-	-	-	-	+	n/a[10]	-
ILIas	-	-	+	-	+	+	?
IPSYS	-	-	-	-	-	-	-
SmartCities	-	-	?	-	-	-	-
Co-Simulations							
EPOCHS	+	-	-	-	+	+	?
GECO	+	-	-	-	+	+	?
Others	-	-	-	-	(+)[11]	-	-

[8] Can easily be added as the simulator is open-source and the integration of other models has been foreseen.
[9] The GLM files offer some scalable aspects but this functionality is very limited.
[10] Not applicable as simulators run in real-time.
[11] Only true for Bergmann et al. [BGG+10].

Part II

The mosaik Concept

“Have no fear my intrepid adventurer, the complexity is layered.” – http://docs.jboss.org

5 Conceptual Architecture

The mosaik conceptual architecture presented in this part of the thesis is inspired by two layered approaches being in this chapter, namely the Levels of *Conceptual* Interoperability (LCIM) model and the *Architecture* for Modeling and Simulation. Therefore, the mosaik layers are called a "Conceptual Architecture". According to Malan and Bredemeyer [MB04] a conceptual architecture directs attention to an appropriate decomposition of the system without going into details of the system specification. Indeed, the resulting architecture does not reflect the detailed architecture of the implementation (see Chapter 11) but rather breaks down the overall task into distinct work packages/aspects that can be dealt with individually. How the conceptual architecture of mosaik is derived from the approaches found in literature is discussed in the following chapter. Subsequently, an overview of the architecture is given and a final overview of works related to individual layers is provided.

5.1 Foundation / Existing approaches

As layered models have a long tradition in computer sciences it is no surprise that several researchers started to create layered approaches to understand the simulation interconnection problem [Tol10, p.407].

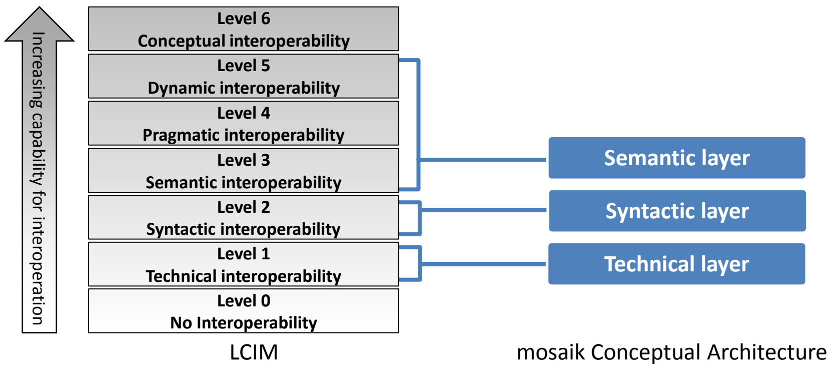

Increasing capability for interoperation
Level 6
Conceptual interoperability
Level 5
Dynamic interoperability
Level 4
Pragmatic interoperability
Level 3
Semantic interoperability
Level 2
Syntactic interoperability
Level 1
Technical interoperability
Level 0
No Interoperability
LCIM
Semantic layer
Syntactic layer
Technical layer
mosaik Conceptual Architecture

Figure 5.1: The LCIM [Tol06] as the foundation of the mosaik conceptual architecture

5.1.1 The Levels of Conceptual Interoperability Model (LCIM)

One of these models is the LCIM which was first published in Tolk and Muguira [TM03] and was subsequently improved, e.g. by incorporating the works of Page et al. [PBT04]. The LCIM has already been applied in various domains and international standardization efforts [TTDW06].[1] In the Smart Grid domain, the U.S. Department of Energy (DoE) project GridWise [Gri08] uses a derivative of the LCIM [TT08]. Figure 5.1 shows the current version of the LCIM. According to Tolk [Tol06] it was first presented by Turnitsa

[1] For a discussion of LCIM compared to other layered models of interoperability (e.g. the Levels of Information Systems Interoperability model (LISI)), see Tolk and Muguira [TM03].

[Tur05]. In the following the different layers are described based on discussions in Tolk [Tol06], Turnitsa and Tolk [TT08], Wang et al. [WTW09] and Tolk [Tol10].

Technical Interoperability This layer deals with network and infrastructure issues, ensuring basic data exchange between the systems (integratability). The underlying networks and protocols for this basic data exchange are unambiguously defined.

Syntactic Interoperability This layer defines a common structure (data format) to exchange data in a syntactically correct way (order and form), but the meaning of the exchanged items is still unknown.

Semantic Interoperability The exchanged data can be semantically parsed and a common understanding of the exchanged data is provided. Using a common information exchange reference model, the meaning of the exchanged data is unambiguously defined. This corresponds to the interoperability as defined in Definition 6. In the Smart Grid domain, the CIM (see Chapter 2.4) has been developed specifically for this purpose and will also be considered by the approach presented in this thesis to achieve semantic interoperability (see Chapter 7.4.1).

Pragmatic Interoperability This level is achieved when the interoperating systems have knowledge about the context in which the data is applied by the participating systems. Zeigler and Hammonds [ZH07] define pragmatics as “the use to which data will be put”.

Dynamic Interoperability is achieved when the systems that interact are able to understand the effect of interchanged information in the participating systems and the related state changes that occur over the time and the systems are able to reorient information production and consumption accordingly. In other words, this is the capability of being aware of and able to cope with changes of the Pragmatic Interoperability layer over time.

Conceptual Interoperability This is the highest level of interoperability. It is achieved “[...] when complete understanding of the concepts inherent within the target and source data models is shared, or shareable, between the data models” [TTD08]. This requires that all conceptual models (i.e. the implementation independent models) are fully specified and not just “a text describing the conceptual idea” [TT08].

Figure 5.1 also introduces the first layers of the mosaik conceptual architecture based on the LCIM model. To keep the architecture compact, the mosaik semantic layer combines those three layers of the LCIM that are related to semantics (understanding the meaning of data and the behavior of other systems).

Wang et al. [WTW09] describe two different roles in which the LCIM can be used. In the *descriptive role*, it can be used to describe how existing systems are interoperating and what level of the LCIM is achieved by the corresponding system. In this role, the LCIM can serve as a basis for rating the interoperability degree of a system. For the purpose of this chapter, the *prescriptive role* of the LCIM is relevant. It prescribes approaches and requirements that a (non-existing) system must satisfy to achieve a certain degree of interoperability. Wang et al. [WTW09] provide a set of domain unspecific engineering approaches to achieve the different levels of interoperability. In the next chapters appropriate approaches for each layer are chosen based on literature research and with respect to the mosaik specific requirements.

Limitations

The mosaik conceptual architecture as it is presented here does not include the "Conceptual Interoperability" layer of the LCIM. As pointed out by Tolk et al. [TTD08] this requires that the underlying conceptual models are fully documented to enable their interpretation and evaluation by other engineers. Tolk et al. further state that currently there are no generally accepted concepts to document the conceptual models in such a way that an automatic model selection (with respect to composability) is possible. As improving on this situation requires substantial effort and the focus of the research presented in this thesis is on a scalable scenario definition concept (Research Question 3, see Chapter 1.3) this layer has been left out. Wang et al. [WTW09] propose the use of a standardized documentation framework for describing the models on the conceptual level, such as the Department of Defense Architecture Framework[2] (DoDAF) as it was developed to provide a means for the standardize collection of information about a system [Han12].

The LCIM presented above deals with different levels of interoperability. While interoperability is the ability to exchange data or services at run-time, composability is the ability to assemble components prior to run-time [PW03a]. "It can be seen that interoperability is necessary but not sufficient to provide composability" [PW03a]. Petty and Weisel [PW03a] therefore defined composability as "the capability to **select and assemble** simulation components in various combinations into **valid** simulation systems to satisfy specific **user requirements**" (see Definition 5) . While the used layers of the LCIM are helpful to ensure syntactic interoperability and provide semantics to ensure validity (thus, covering the first two research questions defined in Chapter 1.3), the aspect of selecting and assembling simulation components (to satisfy the users simulation requirements) is not covered by the LCIM.

5.1.2 The Architecture for M & S

Figure 5.2 therefore presents an alternative layered approach called "Architecture for M&S" (Modeling & Simulation) proposed by Zeigler et al. [ZPK00]. It is also comprised of six layers, but other than LCIM it is intended to provide a framework for implementing comprehensive collaborative environments for M&S and takes a more architecture centric view on simulation systems. This model accounts for the selection and assembly aspect on the *Design and Search Layer*. In the following the different layers are described based on Tolk [Tol10] and Zeigler et al. [ZPK00].

Network Layer Contains everything that makes up the infrastructure, including computers and the network.

Execution Layer Contains the software to execute the simulation models (in the

[2] `http://dodcio.defense.gov/dodaf20.aspx` (accessed 2 January 2014)

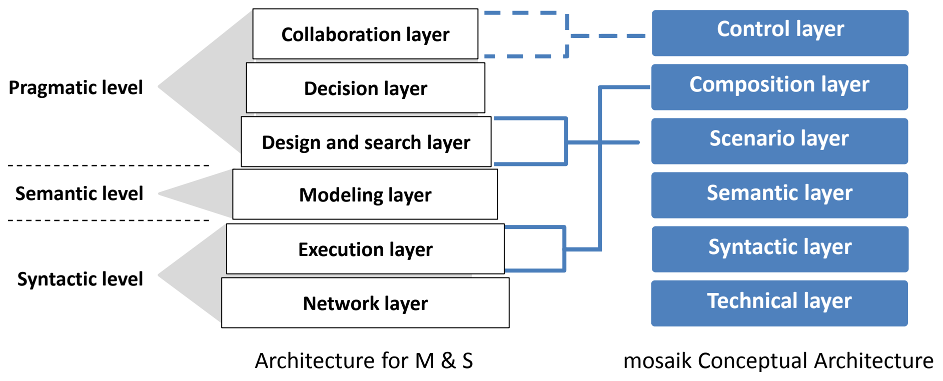


Figure 5.2: Architecture for M&S [Tol10] and its influence on the mosaik conceptual architecture

original contribution by Zeigler et al. [ZPK00] it was therefore called Simulation Layer). This also includes protocols for distributed simulation, databases for logging the results as well as means to analyze the results.

Modeling Layer Allows to specify models in formalisms that are independent of the actual implementation of the execution layer. The specified models are then executed by the execution layer.

Design and Search Layer This layer supports the design of a composed simulation system based on architectural constraints and requirements. This layer also allows to investigate large families of alternative models and find the most appropriate one.

Decision Layer Here, the capability to select and execute large model sets is added. This layer allows to perform explorative ("what if") simulations as well as simulation based optimizations.

Collaboration Layer This layer "allows experts – or intelligent agents in support of experts – to introduce viewpoints and individual perspectives to achieve the overall goal [of the simulation efforts]" [Tol10].

The figure also shows three additional linguistic levels defined by Zeigler and Hammonds [ZH07] two of which (syntactic and semantic level) can be related to the layers of the LCIM. According to Tolk [Tol10, p.411] the meaning of the pragmatic level differs among both models. The pragmatics defined in the Architecture for M&S are related to the context of the system whereas the pragmatics in the LCIM terminology refer to the data exchange within the application. As mentioned above, the issues covered by the three layers related to the pragmatic level are not within the scope of the LCIM as they deal with the use of the system [Tol10].

While the modeling layer (specifying models in some kind of formalism) is not relevant to the objective of this research, the Design and Search Layer (specifying the composition) and the Execution Layer (executing the composition) provide inspiration for extending the mosaik conceptual architecture to cover the aspects of scenario specification as well as performing and executing the composed simulation (Research

Question 3).[3] Figure 5.2 shows the extended conceptual architecture for mosaik and how it relates to the Architecture for M&S. The mosaik *Scenario layer* deals with the specification of (potentially large-scale) scenarios and the *Composition layer* will allow to perform and execute a composition (i.e. to interpret and execute a scenario description). Finally, the top-most layer, the *Control layer* has been added to allow the interaction with control mechanisms (Research Question 4). It is somewhat related to the Collaboration layer of the Architecture for M&S as collaboration may take place after a simulation has finished, for example to interpret the results, or it may take place while the simulation is running, for example to collaboratively interact with the simulation. In the context of mosaik and Smart Grid simulation in general, the simulated power system is intended to be operated by autonomous control mechanisms. Especially in case of MAS based strategies this can be seen as collaborative effort and hence, the control layer can be related to the collaboration layer.

5.2 Overview of the Conceptual Architecture

Figure 5.3 gives an overview of the six layers of the mosaik conceptual architecture. As mentioned above, they do not reflect the detailed architecture of the resulting implementation but rather represent distinct functional aspects that have to be dealt with. The figure also indicates how the research questions defined in Chapter 1.3 are related to the different layers. As the conceptual architecture covers all research questions and is based on the widely known[4] LCIM, it is assumed that the concept is sound and valid. Chapter 12 will show that a first implementation of the concept can meet the requirements and could be applied successfully in actual research projects. To get a basic understanding of the different layers they will be explained briefly in the following. Each layer will then be elaborated in the following chapters.

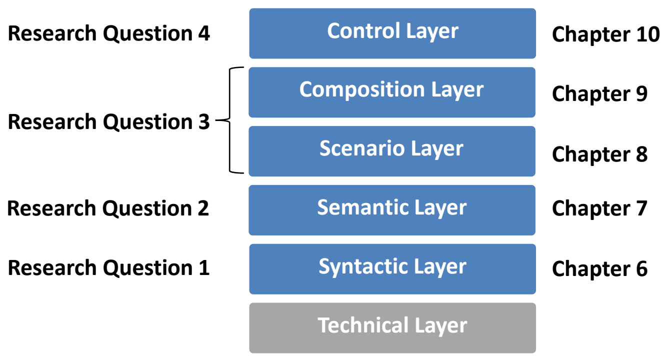


Figure 5.3: Conceptual layers of the mosaik approach

Technical layer The technical layer includes computational and networking hardware

[3] The aspects of the decision layer, e.g. support for simulation based optimization (i.e. optimizing the parameters of the composed models), is not within the scope of this research and may be subject to future work.

[4] By the end of 2013 the original publication by Tolk and Muguira ([TM03]) was cited almost 300 times according to Google Scholar (`http://scholar.google.de/scholar?hl=de&q=The+Levels+of+Conceptual+Interoperability+Model&btnG=&lr=`)

as well as those parts of the mosaik software that deal with the management of the simulator processes. The latter includes the initialization, monitoring and proper termination of simulator processes as required for the composition. This corresponds to the lowest levels of both, the LCIM (Level 0, Level 1). However, in the mosaik conceptual architecture this layer also includes the management of the simulator processes (e.g. starting, stopping) as well as a distributed simulation infrastructure [**R 28** – Execution on Multiple Servers]. Details and development of this layer are not part of this thesis and have been published in [SS12]. The implementation is briefly described in Chapter 11.1.

Syntactic layer The syntactic layer provides a well-defined interface called *SimAPI* which has to be implemented by a simulator in order to be integratable into the mosaik platform. In accordance to the related layered models, this interface only ensures a syntactically correct exchange of simulator data but does not make any implications about the content or the meaning of the exchanged data. This is identical to the syntactic layer of the LCIM.

Semantic layer On the semantic layer, a reference data model will be used to achieve a common understanding of the exchanged simulator data and hence interoperability as defined by Definition 6. This layer can therefore be related to level 3 (semantic interoperability) of the LCIM. This reference model is then used to create formal simulator descriptions. Each simulator description includes information about the possible temporal resolutions of the simulator, the contained models as well as their entities and their data flows. As will be shown, the widely used concept of ports is used to group data flows, attach additional semantics and provide a means for checking if entities are composable. The entity descriptions also include a mapping to a taxonomy of conceptual entity types defined in the reference data model and for each port it can be defined what conceptual entity types are a valid target (context). Hence, this layer can also be related to level 4 (pragmatic interoperability) of the LCIM (knowledge about the context in which the data is applied by the participating systems). The semantic layer will be the basis for the scenario definition and the automatic composition and validation as defined in the next layers. Both, the reference data model and the taxonomy described above are proposed as appropriate means for the semantic and pragmatic layer according to Wang et al. [WTW09]. As shown in Figure 5.1 level 5 of the LCIM (dynamic interoperability) is also part of the semantic layer. Mosaik focuses on the composition of physical Smart Grid entities (and thus data flows that represent physical relations). The analysis of the scenarios and simulation models in Chapter 3 revealed no need to capture changes in which models react to these flows. Hence, the semantic layer of mosaik does not provide any support for dynamic interoperability. However, as discussed below, the scenario layer provides some support for dynamic, model-state dependent behavior.

Scenario layer The purpose of the scenario layer is to formally capture all simulation intentions by the user. This includes the number of models to use as well as their configuration and interconnection. The information provided by the semantic layer will allow an automatic validation of many aspects of the scenario definition. A rule based approach will be presented that allows the specification of Smart Grid scenarios which can be composed and validated based on the available, semantic simulator descriptions of the semantic layer. Therefore, the scenario layer achieves composability according to Definition 5 as it allows the user to "select and assemble simulation components [prior to run-time] in various combinations into valid simulation systems". As described above, the scenario layer can, to a certain extend, be related to level 5 (dynamic interoperability) of the LCIM, as the scenario layer allows to specify conditions that dynamically enable and disable entity connections (e.g. to simulate different charging locations of an EV).

Composition layer This layer includes the concepts and methods required for automatically composing and validating as well as executing the overall simulation. Based on a formal scenario description, the mosaik engine has to initialize the required simulators, interpret the scenario model and analyze the resulting data flows in order to determine the required execution order of the simulators. The execution layer of the Architecture for M&S can be seen as equivalent as it "comprises the software used to implement the simulation" [Tol10].

Control layer The control layer defines an interface called *ControlAPI* which can be used by any kind of control mechanism to access and manipulate the state of the simulated entities. The ControlAPI decouples the simulated scenario from the control mechanisms, such that a scenario can be reused for the evaluation of different control mechanisms. As the ControlAPI allows to access the structure of a scenario (the connections between the simulated objects) as well as the data semantics defined on the semantic layer, a control mechanism can be implemented in such a way that it can also be evaluated with different scenarios.

5.3 Works Related to the Individual Layers

In Chapter 4 only related work with respect to Smart Grid simulation has been discussed to motivate the research presented in this thesis. However, there is also a large number of related works from other domains. These are summarized in Table 5.1. As these works only deal with aspects of individual layers of the mosaik conceptual architecture, these works were not discussed in this chapter. To increase cohesion, the table only provides an overview of these works and indicates for which layer of the mosaik conceptual architecture the respective work is relevant. A detailed discussion is done in the corresponding sections below when the relevant layer is presented in detail.

Table 5.1: Related works dealing with simulator interoperability or composability

Approach	Layer				
	Syntax	Semantic	Scenario	Composition	Control
BOM [Sim06]		X			
Clauß et al. [CASB12]				X	
CODES [TS07]		X	X		
CoMo [RÖ8]		X	X		
EPOCHS [HWG+06]	X				X
FAMAS [Boe05]	X		X		
Fishwick [Fis95]				X	
FMI [MOD12b]	X				
GridIQ [Ban10]					X
GridLAB-D [CSG08]			X		
HLA [Ins10b]	X				
IAM [SBO+06]		X	X		
IEC 61850	X				
IDAPS [PFR09]					X
ILIas [KGC+12]			X		X
IPSSS [BGL+04]					X
Nutaro [Nut11b]				X	
OpenMI [Moo10]	X	X	X		
Scenario Matching Approaches			X		
Schmitz et al. [SKdJdK11]	X			X	
Smart Cities [KN09]					X
Zeigler et al. [ZPK00]				X	

6 Syntactic Layer

As mentioned above, the design of the technical layer was not within the scope of this thesis as it is not related to any of the research questions. Hence, the syntactical layer is the first layer that is presented in this thesis, determining the possible interactions with the simulators and its models. It provides a well-defined interface, called *SimAPI*, that simulators must implement for being integrated into the mosaik platform.

6.1 Requirements

Before related works that are relevant to the design of this layer are discussed and the SimAPI is presented, the requirements that influence this layer will be recalled.

R 9 – Intra-Model Topology Access When a simulator allows to access the relations between the entities of its simulation models, the SimAPI must be able to retrieve these relations. This allows to create a complete topology including both, the entity relations resulting from the composition as well as the relations defined within the simulation models. For the Smart Grid domain this is especially relevant for power system simulators as the power grid topology can be accessed by mosaik and offered to the control mechanism (see Chapter 10). This way a MAS, for example, can dynamically adapt its organization form in a manner that considers the power system topology. Without this feature, the organization form has to be hard-coded into the MAS and a change in the mosaik scenario definition triggers implementation effort in the MAS.

R 11 – Static Data Access Whenever possible, the simulation models should provide information about the static properties of the entities to mosaik. The SimAPI has to provide this capability. From a syntactical point of view these static attributes may be the same as those attributes that are changing dynamically after each simulated step. However, when the API offers a dedicated method to retrieve the static attributes only, this can be done before the simulation actually starts (i.e. before any dynamic attributes are available), allowing to use them in the scenario definition to influence the composition (see Chapter 8.5.5). Furthermore, by making these attributes explicit they can be used to parametrize control strategies. In case of properties representing operational limits, for example, these can be used by the control strategies to detect violations of operational limits and perform corresponding actions. Again, this avoids hard-coding operational limits into a MAS that are specific to the scenario or a model, for example.

R 15 – Transmission of Complex Data Types Mosaik aims to support the integration of a wide range of models into the overall simulation. This may also include models that are not strictly physical but also offer a certain degree of intelligence. The EV model discussed in Chapter 3.5.2 is such an example. It allows to receive operational schedules for charging and discharging and automatically follows them. The transmission of such information requires the support of complex data types. Not all of the related works discussed in the next section support such data types.

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs A power system model may contain entities of different types, such as busses, transformers, power lines or switches. The exact number of these cannot always be defined prior to the initialization of the simulation model as it may depend on the configuration parameters of the power grid model (e.g. a CIM topology file used to define the power grid topology). Therefore it is not possible to offer a fixed set of data outputs or inputs for such a model as it is usual for other standards such as OpenMI or FMI (see discussion of related works below). The SimAPI has to account for this by explicitly supporting the notion of entities as introduced in Chapter 2.1.1.

R 18 – COTS Integration Mosaik must allow the integration of COTS simulators. Although COTS simulators are not necessarily closed source software, a number of established simulation tools in the Smart Grid domain belong to this closed source category, for example power flow analysis tools such as PowerFactory or simulation environments like Matlab. As a consequence this requirement implies that the SimAPI has to be applicable to closed source simulators as well. This can be achieved by avoiding to require access to internal details of the simulator.

R 20 – Different Representations of Time It must be possible to compose simulation models that use different paradigms for representing the simulation time. This includes continuous simulation models as well as discrete models that employ a time-step or discrete-event mechanism. As the SimAPI handles the simulator processes in a fixed time step fashion [**R 22** – Discrete-Time Simulator Integration (fixed step size)], different paradigms can be included easily. Continuous models are sampled at the fixed time steps and discrete-event based models process the event queue internally until all events that are scheduled before the next time step are processed. A more detailed discussion of this aspect can be found in Section 6.4.2.

R 21 – Heterogeneous Implementations It must be possible to integrate simulators with different implementations (i.e. frameworks, languages, tools). This requirement does not influence the design of the SimAPI presented in this section but has to be considered when actually implementing the SimAPI as presented in Chapter 11.1.

R 22 – Discrete-Time Simulator Integration (fixed step size) The SimAPI has to provide a method for stepping the simulator in a discrete-time fashion. When the method is called, the simulator executes its models for the next time step and waits until the method is called again (see Section 6.4.2).

R 23 – Well-defined Simulator API The definition of the SimAPI will provide a standardized way to access the different simulators, thus ensuring basic interoperability. To achieve composability and allow the automatic validation of a composition an additional formal description of structure and semantics of a simulator (i.e. models, entities and their inputs and outputs) is required. The concept for this is part of the semantic layer and is presented in Chapter 7.

R 29 – Central Parameter Definition As Röhl [RÖ8] points out, making a model parameterizable can increase the reusability by making the model adjustable to a wider range of settings. To avoid that a scenario expert needs knowledge about the implementation details of a simulator/model, a central and standardized way of parameterizing the simulators and models has to be offered. Although this requirement is mainly implemented by the scenario specification concept (Chapter 8), the SimAPI has to provide a way to communicate the parameter values to the simulator when initializing it.

6.2 Related Work

Different domain independent as well as domain specific approaches exist for integrating simulations on the syntactic level. These will be discussed in the following, before the mosaik simulation API will be presented. For each approach it is discussed if it is applicable to discrete-time simulation [**R 22** – Discrete-Time Simulator Integration (fixed step size)] and if the other requirements can be met.

6.2.1 FAMAS (First All Modes All Sizes)

The *FAMAS Simulation Backbone Architecture* [Boe05] was developed as part of the *FAMAS.MV2* project which aimed to design automated container terminals for an extension of the Rotterdam container port. FAMAS was developed in order to create a distributed simulation that allowed the simulation of such a complex container terminal in advance. It provides a peer to peer communication between the simulation components [Boe05, p.139] and offers a simulator time management as well as several other services, such as logging and visualization. The time management service provides two types of synchronization, which are conservative and real-time synchronization [Boe05, p.145]. Optimistic synchronization has, although it could be added to the backbone, not been implemented.[1] As the integration of COTS simulators was a major requirement the implementation of optimistic synchronization for this category of simulators was considered too complex [Boe05, p.146]. For example, many COTS packages do not allow to implement the rollback mechanisms required for optimistic synchronization. With respect to the mosaik syntactical layer it has to be said that the backbone could be used for time-stepped simulator execution (by scheduling the simulator events in fixed intervals). However, the source code has not been made publicly available[2].

6.2.2 FMI (Functional Mock-up Interface)

In the automotive industry it has been common for quite a while to use so called Digital Mock-ups in early stages of the design process. These are usually 3D CAD (Computer-Aided Design) models used to shorten development time, improve communication between different development teams and reduce the costly development of real, physical mocks [KKM97, SBD]. Driven by the idea to not only have geometrical but also functional Mock-ups, the Modelisar project (a European initiative to improve the design of embedded systems and software in the automotive domain) published an open

[1] For details about conservative and optimistic synchronization in the domain of discrete-event simulation please refer to Fujimoto [Fuj90] or Zeigler et al. [ZPK00].
[2] Boer, C. A.: "Re: FAMAS". Message to the author. 9 June 2012. E-Mail

interface standard called FMI [MOD12b] in January 2010. By the end of 2012, the second version of the standard was released. The number of FMI use cases in industry presented on the 2012 Modelica conference [Mod12a] indicates that the standard has already been adopted by a number of tool vendors and finds broad acceptance in a number of domains.

Besides a common basis, the FMI standard is split into two parts. The first part "FMI for Model Exchange" allows a modeling environment (e.g. Modelica or Matlab) to export a dynamic system model as C-Code that implements the FMI interface. Such an exported model is called FMU (Functional Mock-up Unit) and can subsequently be used by other simulation environments that support the integration of FMUs. Such an FMU only contains the model described by differential, algebraic or discrete equations and has to be solved by the integrators of the using system. As mosaik is neither a modeling environment, nor does it aim to act as a solver for such FMUs, this part of the standard is not considered any further. The second part of the FMI standard, however, intends "to provide an interface standard for coupling two or more simulation tools in a co-simulation environment" [MOD12b]. While this sounds promising in the first place, there are three important requirements that this standard cannot meet in the current version.

R 15 – Transmission of Complex Data Types Version 2.0 Beta 4 of the FMI standard only supports the exchange of primitive data types [MOD12b]. Therefore, this requirement cannot be met. However, it is indicated that other data types may be included in future versions. This may also include complex data types.

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs The FMI standard does not support the decomposition of models into different entities. Therefore, the integration of a power system model on entity level is not possible and the requirement cannot be met. A workaround could be to write a generic FMU adapter that splits the data provided by an FMU onto different entity instances by relying on naming conventions of the data. However, this is an error prone solution which requires the manual creation of such an adapter for every FMU that is to be integrated.

R 9 – Intra-Model Topology Access As the standard does not make the distinction between models and entities, the access of relations between entities is not possible.

For these reasons, mosaik will not rely on the FMI standard for the syntactical layer. However, as there is already a large number of tools that support export of models as FMU[3], the integration of FMI compliant models should still be possible. This will, for example, enable the integration of models implemented in the Modelica[4] language, which is becoming popular for power system modeling as well [Lar00, CSLS00, PITY12, HS12, EWP12]. Figure 6.1 shows that the integration of FMI into the mosaik concept can be done in a way that complies to a proposed use case of the FMI standard. The gray components *Master* and *Application Server* are not part of the FMI standard but represent a distributed simulation infrastructure. This will be provided by the technical

[3] `https://www.fmi-standard.org/tools` (accessed 6 March 2013)
[4] `https://www.modelica.org/` (accessed 18 Feb. 2013)

layer of mosaik. The Master component also contains the simulation logic. This is part of the composition layer of mosaik. In the evaluation (Chapter 12.2.2.3) there will be shown how a generic FMI adapter can be implemented that is used to interface FMU compliant simulation models. This adapter is integrated into mosaik like any other simulator, using the simulator API of the syntactic layer presented below.

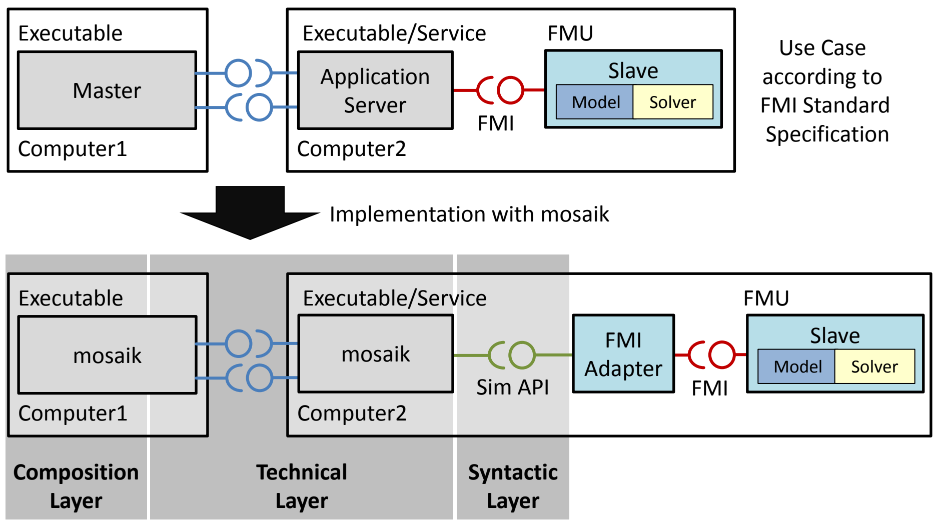


Figure 6.1: FMI compatible co-simulation with mosaik (based on [MOD12b])

6.2.3 HLA (High level Architecture)

The IEEE 1516 High Level Architecture [Ins10b] is an international standard describing an architecture for the interconnection of distributed simulations operating in a discrete-event based fashion. Its development has been initiated by the U.S. Department of Defense (DoD) in 1995 to prevent the development of proprietary simulation interconnection interfaces in the domain of military simulations [Pag07]. The need for the development of simulation interconnections is based on the insight that it will not be possible for any single, monolithic simulation to meet all requirements of the users. Also, not all desired combinations of simulation components can be foreseen in the beginning, requiring a flexible approach for combining the simulations. Furthermore no simulation engineer will be able to gain deep knowledge in all areas of a domain, making it necessary to join simulations from different developers and/or institutions which are experts in their particular field [MH02].

Any simulation or component participating in the simulation interconnection is called a *federate*. The sum of all federates being interconnected according to the protocols defined by the HLA is called the *federation*. In a federation, the participating federates do not communicate directly with each other but through a central communication infrastructure called RTI (Run-Time-Infrastructure) in an event-based fashion. Federates can register their interest in events and also create and publish events. Different commercial and freely available implementations of the RTI exist [MH02]. The HLA standard is comprised of three parts.

HLA Interface Specification Describes the services the RTI offers to the federates and vice versa. The standard defines a number of services that any RTI has to provide [Boe05, p.28]:

1. *Federation Management* - Basic functions for initiating, joining, resigning and managing a federation execution.
2. *Declaration Management* - Controls the way in which federates declare their intent to produce or consume data.
3. *Object Management* - This service is used by federates to register new object instances (=simulated entities), update their attributes and send and receive interactions.
4. *Data Distribution Management* - Used to distribute data in the federation in an efficient way, i.e. distributing data only to interested federates instead of broadcasting it.
5. *Time Management* - Manages the simulation time in such a way that data is delivered to federates in a causally correct and ordered fashion. This includes management of the simulation time and event delivery for each federate.
6. *Ownership Management* - Is used by federates to transfer the ownership of objects to other federates.

Object Model Template (OMT) Providing a common method for recording information about the federates and the federation. [RTI00]

HLA Rules A set of rules for federates and federations that defines the basic principles of communication. For example, each federate must provide a *Simulation Object Model (SOM)* according to the OMT that defines the objects and interactions the federate offers. Furthermore, every federation has to provide a *Federate Object Model (FOM)* that is used to define all shared information (e.g. objects and possible interactions) [RTI00] in the federation as well as interoperability issues such as data encoding schemes.

Although the HLA specifies this large set of services the "HLA, for example, does not specify any tools to design or deploy a federation. [...] As a result, these frameworks require a significant amount of tedious and error-prone hand-developed integration code" [HNN$^+$12]. The deployment of a federation includes starting the federate processes and connecting them to the RTI. In mosaik, the technical layer provides services for managing the simulator processes [SS12]. Obviously, using the HLA instead of the syntactical layer presented in this section would still require such a technical management layer.

Boer [Boe05], who developed the FAMAS approach (see Section 6.2.1), did extensive research regarding the usage of the HLA in the industrial domain. Based on observation of COTS tools he came up with the initial hypothesis that HLA is hardly applied in industry. Related to this hypothesis he tried to answer three questions: 1. Is this hypothesis true? 2. Why is this the case? 3. What can be done to change this? He first conducted a survey in which 19 COTS simulation package vendors participated [BdBV06a]. The survey contained a number of questions that were designed to prove the hypothesis (question 1) and if so, to find some first answers why HLA is hardly applied in industry and what can be done to change this (question 2 and 3). Of the 19 vendors, 7 stated that according to their knowledge successful projects have been carried out using HLA to integrate their simulation package with others. Nonetheless, most of them also stated that they intended to stay with low level technical protocols such as WinSock, COM or .NET instead of HLA [BdBV06a, p.1056]. Those that refused to consider HLA as a future feature gave the following reasons:

- The cost/benefit ratio is too low.
- As it is very military specific many features are not required in other use cases, making it unnecessarily complex.
- There are performance issues.
- System management for running the model is complex and not easy to integrate in a general architecture.
- There are representation problems of the different attributes that are to be exchanged.

Based on these results he conducted a second, telephone interview based survey to get more detailed arguments [BdBV06b]. The survey again included COTS simulation package vendors but also industrial and defense simulation practitioners, HLA designers and developers as well as commercial HLA vendors. The COTS vendors stated that their customers wanted to create models as fast and cheap as possible and that these were of a "throw away" type and hardly maintained during their life cycle. Also, they were often too specific for being reused in a broader context. Therefore, and because customers do not have much experience with HLA or simulation interoperability in general, the use of HLA is perceived to cause more problems than providing benefit. However, tool vendors do recognize that combining COTS models is reasonable when combining them with other applications, when doing collaborative design and development (as it is the case in the military domain where each division is developing parts of the overall simulation) or for combining the models with specialized simulation tools (e.g. power flow analysis tools in the Smart Grid domain). The COTS vendors stated that a good architecture should be simple, offering only minimal functionality and time synchronization. Integration should be painless, efficient and lead to understandable models and clear communication between them [BdBV06b, p.1064]. Industrial simulation practitioners strengthened the point that "HLA offers too much irrelevant functionality for industry" [BdBV06b, p.1064]. A similar statement is made by Hopkinson et al. [HWG+06], claiming that modifying existing simulators to integrate them using HLA is often difficult due to limited input and output options and difficulties in accessing the simulators internal states. As a consequence they also define a custom interface and do not use the HLA (see discussion of EPOCHS in Chapter 4.2.1). Also, the problem of mapping/representation of objects and attributes has been mentioned by the tool vendors. Boer, de Bruin and Verbraeck [BdBV06b] state that this could be solved by defining standard object descriptions which is exactly what the semantic layer of mosaik does (Chapter 7).

Obviously, the commercial industry domain cannot be compared directly with the scope in which the mosaik concept is mainly being developed, namely Smart Grid simulation within research projects at institutes and universities. However, the major arguments discussed above are still valid:

Irrelevant functionality For the Smart Grid simulation requirements presented above, the majority of features is not required either. The HLA time management allows to synchronize simulators in an event-based fashion, whereas for mosaik a simple time-stepped approach is sufficient [**R 22** – Discrete-Time Simulator Integration (fixed step size)]. Although this can be achieved by scheduling events in equidistant time steps using HLA, the protocol overhead is unnecessarily big. The HLA ownership management accounts for scenarios where simulated objects move from the responsibility of one simulator to another. For mosaik this is not a requirement. Also, as the control strategies are not part of the mosaik composition concept, the interactions between the entities are less complex than in the military domain. The argument that users do not have much experience with HLA or simulator interoperability in general also holds for mosaik, as the future users are students and researchers using it for evaluating their control strategies, scenarios or models (see 3.1.2).

Integratability As demanded by the COTS vendors in the FAMAS survey, mosaik will provide a lightweight simulator API that offers only the minimal required functionality, making it easy to integrate simulators to mosaik.

To sum up, the HLA standard is unnecessarily complex for the requirements that mosaik aims to meet. The Smart Grid scenarios are characterized by simulated entities being arranged in a static topological fashion (with the exception of EVs) and relatively simple interactions in form of physical flows (e.g. power or solar irradiance). This is completely different from the military scenarios that were the starting point for the development of HLA where entities can move freely in a three dimensional space and have much more complex interactions. As [**R 1** – Keep It Simple] demands for a simple solution to the composability problem, the HLA standard is not considered in the following.

6.2.4 EPOCHS

The EPOCHS approach has already been discussed in Chapter 4.2.1. Although it is a co-simulation approach for Smart Grid simulation based on the federation of simulators, the focus is not on composing different physical component models into a power system topology but rather on coupling a power system simulator with communication network simulations. As the used interface is neither available nor described in detail, it is not considered for the design of the syntactic layer. However, for the evaluation of the EPOCHS approach different power system simulators have been integrated using only their built-in APIs, indicating the feasibility of the mosaik concept.

6.2.5 IEC 61850

From the Smart Grid domain, the IEC 61850 standard introduced in Chapter 2.4 could be used on the syntactic layer to provide syntactic interoperability (for IEC 61850 with respect to the semantic layer, see Chapter 7.4.4). However, there are some aspects that make IEC 61850 not the ideal choice.

1. The IEC 61850 standard is very complex and "defines a relatively complicated communication structure [..]" [LC08]. This makes development of an IEC 61850 compliant communication stack difficult and does not match well with [**R 1** – Keep It Simple]. Buying a commercial stack is not ideal for research projects involving many parties and the academic users that mosaik targets (see Chapter 3.1.2).
2. The depth of the hierarchical data model of IEC 61850 (see Figure 2.10) matches the hierarchical structure of simulator, model and entities (see Figure 6.3) as discussed in Chapter 2.1.1 and required by [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs]. At a first glimpse, simulators could be represented by *Servers*, the models executed by a simulator could be represented by *Logical-Devices* and the entities of a model could be the *Logical-Nodes*. But as servers and the logical devices do not have properties, how are simulators and models represented by these nodes parametrized? However, in simulations the entities represent the physical objects that make up the Smart Grid and to be IEC 61850 compliant these have to correspond to the logical-devices and not to the logical-nodes suggested above. Furthermore, the abstraction level of simulation models can vary. An entity can be a part of a substation, a complete substation or even a complete LV grid. So an entity of the simulation may correspond to a single IED or a number of IEDs and a consistent mapping to the objects of the IEC 61850 standard is not possible. All in all, this would be different from the way the standard is used, eliminating the benefit of a standard compliant API.
3. Finally, to the best of the author's knowledge, there is no trend toward IEC 61850 compliant simulation models for Smart Grid components so that model integration would not benefit from an IEC 61850 compliant interface.[5]

Despite these disadvantages, the evaluation of IEC 61850 compliant control mechanisms can be done with mosaik when adding a corresponding mapping on top of the control layer, as discussed in Chapter 10.5.

6.2.6 OpenMI

The Open Modeling Interface standard intends to provide a standardized way for linking environment-related models [Moo10]. In OpenMI terms, a model is an entity that can provide data and/or accept data as input. For linking a number of models OpenMI defines an interface that a simulation engine (an engine component in OpenMI terms) needs to

[5] However, there are simulators for IEC 61850 based systems, such as substations (e.g. Juarez et al. [JRMRM12])

implement to become “linkable”. From an architectural point of view OpenMI enforces a pull-based pipe and filter architecture with memory based data exchange [Moo10, p.12]. A component (e.g. a model or a database) requests data from another linked component and this component replies with the corresponding values. For calculating the reply values a component may in turn issue a request to another linkable component. As every linkable component only handles one request at a time OpenMI is single-threaded.

The OpenMI specification states that “[...] a model can be regarded as an entity that can provide data and/or accept data” [Moo10]. Hence, OpenMI does not distinguish between models and entities. A model, accessed through the OpenMI API, is similar to a single entity, for example a river model. Thus, requirement [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs] cannot be met. Furthermore, as discussed in Section 8 the OpenMI standard is designed for the composition of a relatively small number of models. As the Smart Grid scenarios contain a potentially large number of simulated entities of different types it should be possible to execute them in parallel whenever possible to reduce simulation time. Hence, the single-threaded approach of OpenMI is not optimal. Therefore, OpenMI is not used for the syntactic layer. However, as shown later, a number of ideas from the OpenMI standard influenced the mosaik semantic layer that is presented in the next section.

6.2.7 Schmitz et al. [SKdJdK11]

Schmitz et al. [SKdJdK11] present a framework for coupling model components with fixed time steps, as demanded by [**R 22** – Discrete-Time Simulator Integration (fixed step size)]. While the paper focuses on different scheduling schemes for the coupled components (and is therefore picked up in Chapter 9.6), they also present a very simple API for interfacing the components in a discrete-time fashion. However, they also do not distinguish between models and entities [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs].

6.2.8 Related Work Summary

Table 6.1 shows the related work that has been analyzed for the design of the SimAPI. Although most of these originate from a specific domain, some of them have been developed as general standard that can be applied to other domains as well. However, the source code is either not available (FAMAS), the interfaces are too complex (HLA) or other requirements (e.g. [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs]) cannot be met. The EPOCHS approach has a different scope than mosaik and the syntactic interface for the participating co-simulators is not published. However, it has shown that the mosaik approach of integrating existing simulators into a Smart Grid co-simulation is feasible from the syntactic point of view. As a consequence, a custom simulator API is developed for the mosaik concept, considering only the relevant requirements to keep the integration of simulators as simple as possible.

Table 6.1: Related work in the field of simulator interfaces

Criterion	FAMAS	FMI	HLA	EPOCHS	OpenMI	Schmitz et al.
Time Handling	Discrete-Event	Discrete-Time	Discrete-Event	Discrete-Time	Discrete-Time	Discrete-Time
Domain of origin	Logistics	Automo-tivey	Military	Smart Grid	Environ-mental Model-ing	?
Domain specific-ness	Medium	Low	Low	High	High	Low
Complexity	?	Medium	High	?	Medium	Low
Availability	No	Open Source	Open Source	?	Open Source	?
R 15	?	-	+	?	+	?
R 16	-	-	+	?	-	-
R 29	+	+	-	?	(+)	?

6.3 The mosaik Simulator API

In Chapter 3.6.1 it was defined that the simulators will be interacted with in a discrete-time fashion [**R 22** – Discrete-Time Simulator Integration (fixed step size)]. This allows to create a SimAPI that is relatively simple, especially as the scope of mosaik does not require additional features such as rollbacks or variable step sizes (see discussion in Chapter 3.4). This promises to allow the integration of a broad range of simulation models and to minimize integration effort and reduce implementation errors. Reduced integration effort is especially important as “new approaches are unlikely to be accepted [...] if they are connected with tremendous migration costs due to programming efforts” [TM04]. The following subsections describe the different functions that the SimAPI offers. A detailed definition of the API functions can be found in Appendix A. The API functions can be split into *Initialization-Time* functions that are only called once in the beginning and *Run-Time* functions that are called repeatedly until the simulation has reached its end.

6.3.1 Initialization-Time Functions

Requirement [**R 29** – Central Parameter Definition] demands for a central, simulator independent parameter definition. This mechanism will be presented on the scenario layer (see Chapter 8). Independent of the implementation details of this mechanism, the SimAPI must offer a method to transfer the simulator and model initialization parameters to the simulator. This is done by the initialization method *init* which is the first method that is called. As it cannot be assumed that a simulator can be started without already initializing the models it was decided to put all required initialization parameters for the simulator as well as its models into a single initialization function. An implementation of the SimAPI can then perform the required simulator specific initialization procedure without having to wait for any further method calls. Besides these parameters, the arguments include the simulator step size and the number and type of models to instantiate. The method is expected to return unique IDs for the instantiated entities as well as their types [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs]. Entities cannot be simulated directly but rather are the result of initializing a model in the simulator. For performing the composition (see Chapter 9.5.1) mosaik must be able to determine what entities are part of the same model as well as what model configuration was used for instantiation. Therefore, the returned entity IDs are grouped into model instance specific dictionaries for each set of initialization parameters that has been passed to the method by mosaik. It is assumed that the returned entity IDs are unique for each simulator to ease the use of the get- and set-data methods introduced below. This approach is illustrated in Figure 6.2.

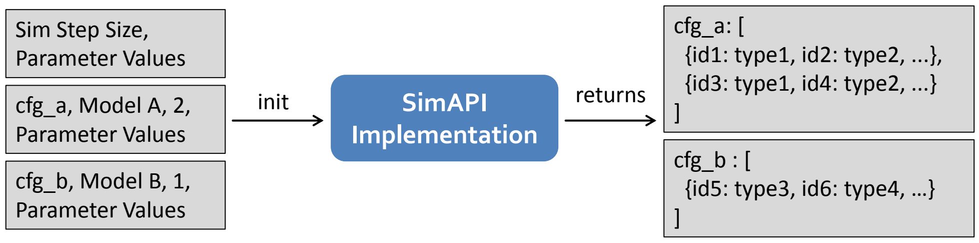


Figure 6.2: Simulator initialization procedure

Once the initialization phase is completed, the SimAPI allows mosaik to retrieve the static entity attributes [**R 11** – Static Data Access] using the *get_static_data* method. If the simulation model does not provide these attributes the SimAPI is allowed to return nothing. The same applies to the retrieval of the model internal entity relations [**R 9** – Intra-Model Topology Access] that the *get_relations* method offers. If supported, the relations are returned in form of entity tuples. So far, the type of relation cannot be defined any further and is assumed to be derived from the types of the related entities (**A 2** – *Typeless entity relations are sufficient*).

6.3.2 Run-Time Functions

After these methods have been called the *Initialization-Time* stage has been completed and the *Run-Time* methods are called until the simulation has reached its end. These methods include the *set_data* method that is called to set dynamic properties of the different entities, the *step* method that is used to advance the simulator by one time step and the *get_data* method that is called after a step has completed to retrieve the new dynamic data of the entities. Figure 6.3 shows the structure of a simulator implied by the SimAPI and the methods it offers to interact with the simulator. As already pointed out above, the usage of the SimAPI can be split into two phases. Figure 6.4 shows this dynamic perspective using a UML Activity Diagram. The activities of the *Initialization-Time* part are performed only once after the method *init* has been called. The example

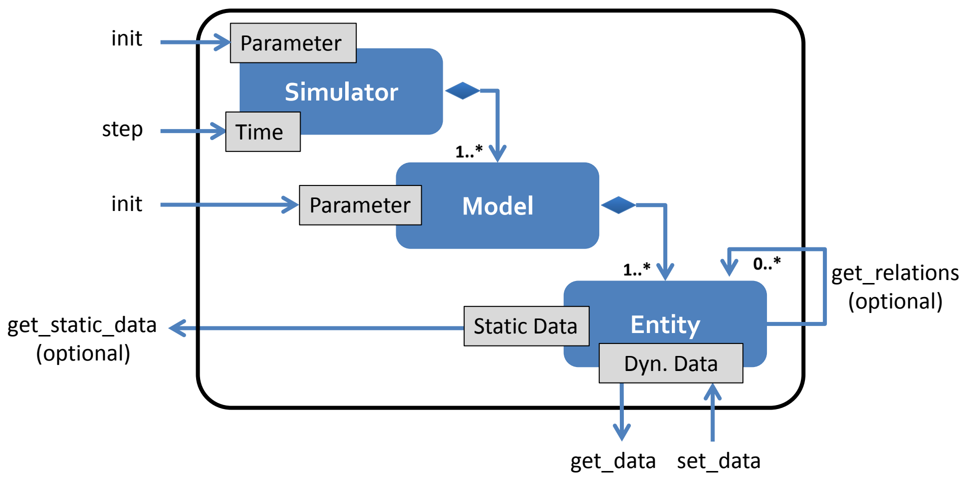


Figure 6.3: Simulation features exposed by the API

shows the two activities a simulator is likely to perform during initialization. First, the simulator process is initialized with the provided parameters. Next, the simulation models are initialized. After the initialization of all models has been finished, the entity relations [**R 9** – Intra-Model Topology Access] as well as the static entity properties [**R 11** – Static Data Access] are available. In the *Run-Time* part of the diagram, the methods *set_data*, *step* and *get_data* are to be called repeatedly by the simulation engine until the end of the simulation is reached. As shown in Chapter 11.1, the values that are transferred by the get and set functions are serialized in the JSON[6] format such that complex data types can also be transferred [**R 15** – Transmission of Complex Data Types].

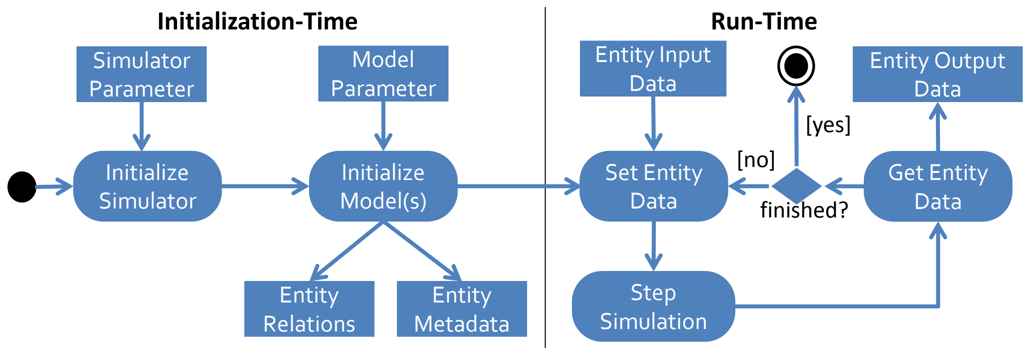


Figure 6.4: Activity chart for using the SimAPI

6.4 Using the SimAPI

In this section possible use cases for the SimAPI are discussed. The section is subdivided into a discussion of the technical perspective (how can the SimAPI be implemented for a simulator) and a discussion of the conceptual perspective (how can the SimAPI be used

[6] `http://www.json.org` (accessed 11 May 2012)

to integrate simulators with different paradigms).

6.4.1 Technical perspective

Straßburger [Str99] discussed four different concepts for integrating simulators with the HLA. Although the HLA is not used for mosaik, the problem setting is similar and can also be applied to the SimAPI presented here. Figure 6.5 summarizes these concepts from most desirable on the left to less desirable on the right (regarding implementation effort and/or performance).

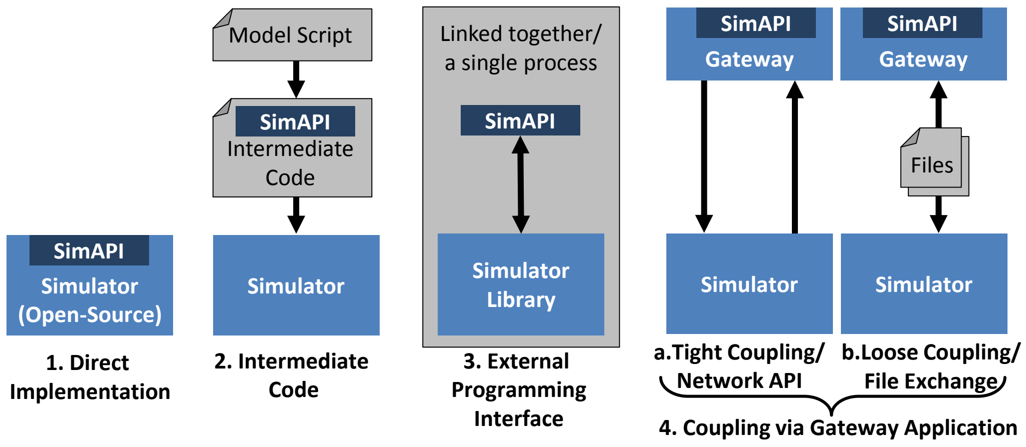


Figure 6.5: Simulator integration concepts (based on [Str99, Syr05])

The first approach assumes that the simulator is open-source and the required interface functionality can be added directly to the simulator. If the source code is not available or should not be modified for any other reason (e.g. a too complex make procedure or lost confidence in the outcome) the second variant may be used. This approach works if the simulator translates models that are defined in a tool-specific modeling language into what Straßburger [Str99] calls "intermediate code", e.g. in C++. This code can then be extended by the interface functionalities and compiled into the final executable simulation. For the Smart Grid domain this concept has been successfully applied by Hopkinson et al. [HWG$^+$06] for integrating the ns-2 simulator into their EPOCHS co-simulation.

If this approach is not possible, the third approach can be applied if the simulation tool offers a library interface (DLL on Windows). This library can be used to link the simulation interface together with the simulator into an executable unit. Hopkinson et al. [HWG$^+$06] have also applied this approach for integrating the PSCAD/EMTDC simulator.

Finally, the fourth and last variant is the use of a gateway process. The gateway process implements the simulator interface and interacts with the simulator process via means such as pipes, files or network sockets, depending on the capabilities of the simulator. This approach can be split further into a loosely and tightly coupled variant [Syr05, pp.48]. The tightly coupled variant relies on network communication to connect to the gateway. The loosely coupled variant relies on file exchange and is, for example, applied by Bankier [Ban10] when integrating the PSAT power flow simulator into the GridIQ framework or by Hopkinson et al. [HWG+06] when integrating the PSLF simulator into their EPOCHS co-simulation. A major drawback of this approach is a reduced performance due to file IO. Also, depending on the tool that is used, the simulator process may have to be restarted for each time step as it is done by GridIQ and PSLF [Ban10, p.37]. In case of mosaik, a loosely coupled approach could use files to set the input data for the next simulation step and to return the data to mosaik after the simulation has done the next step. The simulator parameter received in the *init* method of the API may be passed as command line arguments to the simulator process. The initialization results (entity IDs and types, etc..) could be parsed from the console output of the process or also be retrieved via files.

6.4.2 Conceptual perspective

In Chapter 2.1.3 it was shown that there are different ways in which simulation models represent time. Meeting requirement [**R 22** – Discrete-Time Simulator Integration (fixed step size)], the SimAPI advances the simulators in time-stepped fashion. Thus, all simulators internally working in a time-stepped fashion can easily implement the SimAPI. Simulators offering continuous (differential equation based) simulations or event-driven simulations can also be integrated [**R 20** – Different Representations of Time] as shown in Figure 6.6. For the latter, the value of the last output event that occurred prior to a time-step defines the value that is read by the SimAPI for a time step. In the figure the used value is marked by a filled circle. For continuous simulators, the used value is the continuous output value at the time of the time step.

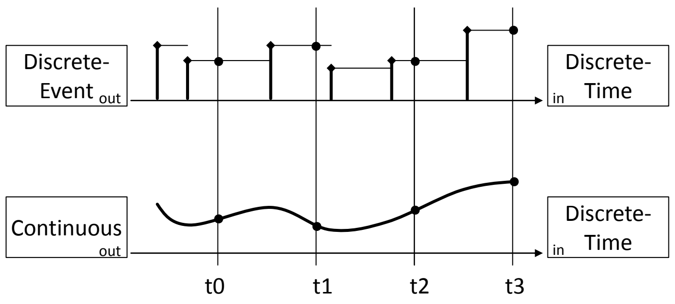


Figure 6.6: Handling different simulation paradigms in a time-stepped fashion (based on [ZPK00])

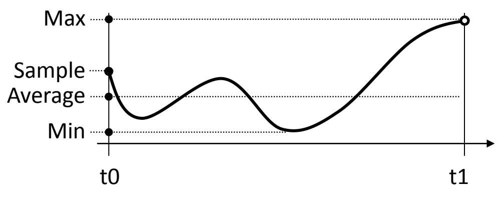


Figure 6.7: Options to calculate output data

Instead of taking the last event value or a single continuous value at that time point it is also possible to return a minimal, maximal or average value for an interval. Using an average value, for example is usual for load curves in the power grid domain. The different resulting values are shown in Figure 6.7 for an interval of a continuous model. Which variant is used by a SimAPI implementation can be defined on the semantic layer for each output value individually. To avoid using a value twice, one end of the interval has to be open. To avoid ambiguity when implementing the SimAPI, Definition 7 precisely defines the expected behavior when implementing the SimAPI:

Definition 7 (Step Semantic) *Calling the* step *function of a simulator advances its time from t_0 to t_1 with $t_1 - t_0 = \Delta t$ being the step size of the simulator specified in the* init *function that was previously called. The data that is subsequently provided by the* get *function is valid for the interval* $[t_1, t_2[$.

6.5 Discussion

In this section the SimAPI, a well-defined interface [**R 23** – Well-defined Simulator API] providing basic simulator interoperability for the interconnection of physical models from the Smart Grid domain, has been presented. How the relevant requirements have been considered is discussed briefly in the following. To keep the implementation of the discussed concepts as simple as possible [**R 1** – Keep It Simple], the SimAPI offers only a minimal set of functionality needed to meet the requirements presented in the beginning of this section. Also, certain methods are optional to support the integration of simulators that do not provide the required information.

6.5.1 Requirements

In the following it is discussed how the requirements have been met by the SimAPI.

R 9 – Intra-Model Topology Access, R 11 – Static Data Access The SimAPI offers methods that allow accessing the static entity properties and retrieving the model internal entity relations, e.g. the interconnections of busses and power lines in case of a power grid model. These are optional as they provide functionality that may not be provided by all simulators or models, especially in case of COTS simulators.

R 15 – Transmission of Complex Data Types As the arguments and return types of the get-/set-data methods are not yet specified, the SimAPI does not place any restriction on the type of data that can be exchanged. While the actual type and structure of the data are defined on the semantic layer (see next section), the implementation chapter (11.1) will show how complex data types are being submitted.

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs To account for the fact that the number of entities that a model contains is not known a-priory in all cases, the initialization method returns the IDs and types of entities specific for each model instance. All following methods (such as get-/set-data) operate on entity level by using these returned entity IDs. This way a composition on entity level can be achieved which is required for composing complex models consisting of many different types of entities such as the power grid.

R 18 – COTS Integration Different concepts for integrating COTS or other closed-source simulation tools have been discussed (see Figure 6.5). The applicability of these concepts for the Smart Grid domain was already demonstrated by Bankier [Ban10] and Hopkinson et al. [HWG+06], for example.

R 20 – Different Representations of Time In Section 6.4.2 it was shown how simulators that contain models with paradigms other than time-stepped can also be integrated using the SimAPI. Models that represent special types of systems, such as input-free or memoryless systems (see Chapter 2.1.4), can also be integrated easily, by simply ignoring the *step* or *set_data* calls.

R 21 – Heterogeneous Implementations For being able to integrate simulators implemented on different platforms or with different programming languages it is important to implement the SimAPI with a technology that is available on a broad range of platforms and for many languages. The chosen technology is presented in Chapter 11.1.

R 22 – Discrete-Time Simulator Integration (fixed step size) The step size of the simulator can be specified during initialization and the *step* method allows to advance the simulator time in a discrete-time fashion.

R 29 – Central Parameter Definition The offered initialization method allows to provide all required configuration parameters for both the simulator as well as the models that it has to execute. This is the first step to meet this requirement. On the semantic layer presented in the next chapter the available parameters, their meaning as well as allowed value ranges can be defined. Based on that, the scenario layer presented in the subsequent chapter allows to define the parameters of all simulators and models that are part of a composed scenario in a central place.

6.5.2 Limitations

The resulting design of the SimAPI is sufficient to meet the requirements of mosaik, but it implies a few limitations that are discussed briefly in the following.

Typeless relations During the design it was assumed that the type of relation between any two entities is not specified any further as no such requirement could be identified [**A 2** – Typeless entity relations are sufficient]. A MAS that uses the entity relation information for dynamic bootstrapping (see Chapter 3.3) has to infer the type of relation from the context, i.e. the type of related entities. If this turns out to be insufficient, this information could easily be added at a later stage by not only returning a tuple of connected entities but, for example, a triple including the type of relation. A well-defined list of potential types has to be established in this case, e.g. as part of the reference data model presented in the next chapter (semantic layer).

Progress in time Like all other discrete-time approaches discussed in this chapter (with the exception of FMI), the SimAPI assumes that the components only progress forward in time (see Definition 7). A reasonable coupling of components that require close interaction within a time step (for example for solving differential equations between the components) can thus not be done adequately [SKdJdK11]. However, in Chapter 3.4 it has been explicitly stated that mosaik does not focus on time-domain simulation which requires such a tight coupling. If this requirement arises later and the scope of mosaik is extended, the FMI standard is a good starting point to extend the SimAPI, as it already offers a solution to this problem.

Missed/delayed events In Chapter 3.6, the advantages and disadvantages of interacting with simulators in a discrete-time fashion were already discussed. Obviously, these disadvantages also apply to the design of the SimAPI. However, it was also argued that a discrete-time approach was seen as a good compromise between disadvantages and complexity, for the given scope of mosaik.

7 Semantic Layer

The syntactic layer presented in the last section provides the basis for exchanging data between the different simulators, allowing to integrate them into the mosaik concept. However, to achieve interoperability (see Definition 6), there must be a common understanding of the exchanged data [ZH07, p.xii]. The semantic layer presented in this section allows to create a reference data model that is used to define such a common understanding (and format) for the data exchanged by the simulated entities. But still, this is not sufficient for achieving composability (see Definition 5). As composability is a design-time property, all information required for determining how to use a simulation model and combine its entities in a valid and reasonable way has to be made available prior to run-time. This includes information such as:

- What step sizes does the simulator support?
- What parameters are available for initializing the simulator and its models?
- What is their data type and unit?
- What value ranges are permissible?
- What models can a simulator execute?
- How many instances of each model can a simulator execute?
- Can these instances have different parameter values?
- What and how many entities does a model contain?
- What data flows does each entity offer and require?

To capture this information, the semantic layer allows to formally define these properties for the integrated simulators and their models. This information is used on the scenario layer (see Chapter 8) to determine the validity of the compositions that are part of a scenario description and to ensure that the Scenario Expert specifies valid simulator and model configurations. So far, it has not been clearly defined what is meant by the term validity. Therefore, in Section 7.3 related works dealing with simulation model composability are discussed and a number of validity aspects are defined. The semantic layer must capture all information that is required to automatically check these aspects when defining a scenario on the scenario layer.

Figure 7.1 shows how the semantic layer is used in the context of the other layers to allow the creation of Smart Grid scenarios and the automatic validation of these. Starting from the bottom right of the figure, a Modeler (stakeholder see Chapter 3.1.2) creates a simulation model using a modeling tool of his or her choice for a specific simulator. The Integrator (maybe the same person) implements the SimAPI offered by the syntactic layer which has been introduced in the last section. If the simulator has been integrated already and only a new model has been added this step is skipped. Next, the Integrator creates a formal description of the simulator and the newly created

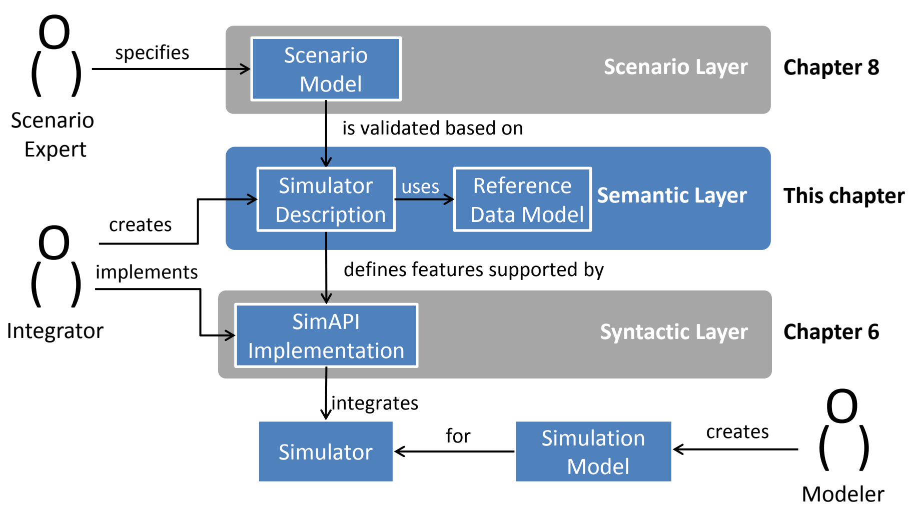


Figure 7.1: Context of the semantic layer

model. If the simulator was integrated before, only the model description is added. This description uses the data elements defined by the reference data model. If the data model does not provide required elements, the Maintainer (not shown in the Figure) has to extend the data model as discussed in Section 7.11. Once the simulator description is available, the new model can be used as a building block for scenario models.

7.1 Metamodeling

A metamodel is a model of models [Fav04]. The metamodel presented here allows to create models that capture the semantics of the entities of a simulation model and their data flows. To prevent the ambiguous use of the word "model" the terms *model* and *simulation model* will be used synonymously throughout the remainder of this thesis when referring to a model that can be executed by a simulator. The term *semantic model* will be used when referring to a description of the semantics of a simulator and its models. The semantic layer presented in this section and the scenario layer presented in the next section will each define a metamodel. To avoid confusion the metamodel presented in this section will be called *semantic metamodel* and the metamodel of the scenario layer will be called *scenario metamodel*. Figure 7.2 depicts this nomenclature.

The metamodel will be defined using the UML [Obj07] notation, standardized by the Object Management Group (OMG). For several parts of the metamodel the UML syntax alone is not expressive enough to specify the metamodel semantics unambiguously. In these situations additional constraints for the metamodel are specified in addition to the UML diagrams by using the Object Constraint Language (OCL)[Obj12] in version 2.3.1 as defined by the OMG. A brief introduction of OCL and its syntax can be found in Appendix B. All OCL constraints for the semantic layer can be found in Appendix C. The conception of the metamodel has been done using the Xtext [Ecl12] domain-specific language framework that has been chosen for implementing the semantic as well as the scenario layer of mosaik. The presented UML class-diagrams have been automatically generated by Xtext. By doing the conceptual modeling in Xtext and not in a separate

UML tool, a manual conversion step from the conceptual model to the implementation is avoided and it is ensured that the diagrams (being part of the documentation) are always up to date. More details on Xtext and domain-specific languages are given in the implementation chapter. Before the metamodel is presented step by step in the next sections, the relevant requirements and related works in the field of semantic based simulator coupling are presented and their applicability with respect to the requirements of mosaik are discussed.

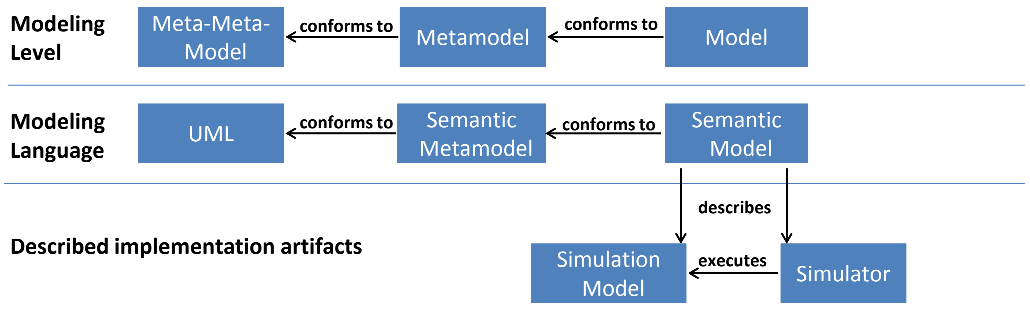


Figure 7.2: Hierarchy of metamodels and models for the semantic layer[1]

7.2 Requirements

A list of requirements that were gathered before the conceptual work began was shown in Chapter 3.10. Requirements relevant for the conception of the semantic layer are identified and discussed briefly in this section.

R 7 – Attribute Based Composition, R 11 – Static Data Access The simulator description must contain information about the static data that the different types of entities of a model provide. By providing this information on the semantic level, the available attributes can already be used during scenario design-time to define attribute specific compositions as will be shown in Chapter 8.

R 14 – Moving DER The simulation of EVs led to this special requirement as these are probably the only type of distributed energy resources that do not have a fixed location. As a consequence, the scenario specification mechanism described in Chapter 8 must be able to define the different locations depending on the value of a dynamic data item provided by the EV entities. Although this is not the task of the semantic layer, the possible values (e.g. HOME, WORK, MISC in case of the EV model used in this thesis) have to be known at design-time, as otherwise the scenario expert cannot specify the conditions properly. Therefore, the semantic layer must offer a means to define the range of possible values for the data provided by such entities.

[1]The semantic model is not a metamodel of the simulation model as it does not provide any concept that can be used for creating the simulation model. Therefore, the simulation model is placed in a separate row underneath the semantic model. In case of a simulation model implemented in Python, for example, the Python grammar would be its metamodel. To put it differently: The semantic model is just an implementation independent description of some aspects of the model and not the modeling formalism.

R 15 – Transmission of Complex Data Types As pointed out during requirements analysis, some models expect input in form of a complex data type, e.g. to represent operational schedules. The semantic layer must consider this and offer a means to define the structure of such complex data types.

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs As the entities will be the elements that are being composed to form a Smart Grid Scenario, the semantic layer has to provide information about the number of entity instance each model instance contains. This way the scenario expert can determine the required number of other model instances to use in a scenario. In case of a power grid model, for example, the scenario expert can choose a corresponding number of load models when he or she knows the number of bus entities of a power grid model.

R 17 – Power Flow Encoding As pointed out in Chapter 3.5.5, different ways of encoding power flows between simulated DER and busses of a power system simulation exist. How mosaik deals with these differences is discussed in detail in Section 7.10.

R 22 – Discrete-Time Simulator Integration (fixed step size) In the requirements analysis it has been defined that the simulators will advance in a discrete-time fashion. Also, simulators with different step-sizes (temporal resolutions) should be composable [**R 19** – Different Temporal Resolutions]. Therefore, the simulator description must provide information about the step sizes that each simulator supports. This information can be used on the scenario level to avoid invalid simulator parametrization as well as to configure the step-sizes as homogeneous as possible in order to avoid inaccuracies.

R 24 – Formal Simulator Description As already discussed, a formal simulator description is required to add semantic as well as structural information to the generic SimAPI in order to achieve composability, i.e. the capability to create and validate simulation model connections prior to run-time. This semantic metamodel presented in this section meets this requirement.

R 25 – Multi-Resolution Composition In Chapter 3.7 it was shown that models of a real world system can have different levels of abstraction. To support the specification of scenarios that include models of different abstraction levels, the semantic layer should provide information on how to bridge the data flows between these levels. For example, a detailed model of an electrical load and a less detailed model of a power grid are available. If the power grid model is sufficient for the objectives of the simulation study it would not make sense to put effort in the creation of a less detailed load model. Rather a combination of both models supported by a data flow conversion method (e.g. convert the three different phases of the load models power flow to a simplified single phase flow for the power model) is desirable. The required mechanism is presented in Section 7.10.

R 29 – Central Parameter Definition Mosaik must offer a mechanism to define the required parameter values of all simulators that make up a scenario. The SimAPI allows to set these parameters but it does not provide any information about what parameters are available. Furthermore, the scenario expert must decide about the values that are assigned to the available parameters. As Zeigler and Hammonds [ZH07] point out “[...] values assigned usually make sense only when we know more about the possible range and units of the variable’s values.” Therefore, the semantic layer will have to allow the definition of the value range (as already discussed for [**R 14** – Moving DER]) as well as the units of the parameters.

7.3 Related Work

As pointed out in Chapter 4 no Smart Grid specific approach dealing with the composability of simulation models exists. However, a number of approaches for composability have been developed in other domains. In the remainder of this section these works are discussed with respect to their applicability for Smart Grid simulation.

7.3.1 CODES

The COmposable Discrete-Event scalable Simulation (CODES) is “an approach to component-based modeling and simulation that supports model reuse across multiple application domains” [TS07]. Similar to the SimAPI of mosaik, the simulation components of CODES are black-boxes that only provide input and output channels. Each component is described by a meta-component in terms of available attributes and behavior. For doing so, the authors propose an XML-based markup-language called COML (COmponent Markup Language). A supportive ontology called COSMO (COmponent-oriented Simulation and Modeling Ontology) is used for component discovery as well as for ensuring semantic correctness of the component compositions. For adding a new application domain, this ontology has to be extended by the domain’s basic components. Furthermore a composition grammar has to be created which determines the domain specific composition rules. The actual specification of a composition is then done in a graphical user interface by connecting the models manually (see Chapter 8.2). Four types of constraints can be formulated to define the semantics of each component: Data type, value ranges, origin/destination types and behavior. For ensuring the semantic validity of a composition these constraints are checked as follows:

Data Type Matching This is a trivial check. Obviously the data type of any output must be identical or convertible to the data type of the connected input. This will also be true for Smart Grid simulation composition (**Validity Aspect 1** – *Data types must match*).

Origin/Destination Matching For each input and output the required origin or destination types can be specified. If this constraint is not satisfied, i.e. the type specified in the composition is not of the expected type, the semantic checking fails. For Smart Grid simulation composition there are scenarios where this is required too (**Validity Aspect 2** – *Origin/destination types must match*). For example, an EV entity may have two outputs: one to the power grid, indicating the drawn power for charging, and one to a separate battery model, indicating the power fed into the battery. Both outputs may be modeled as float values with additional unit information, say kW. Without additional source/sink type semantics it cannot be decided automatically which output should be connected to the power grid and which to the battery model. When the EV entity accounts for charging electronic losses both values are not identical and the correct semantic matters. Also, if the direction of the power flows could be indicated by the algebraic sign, a wrong connection would lead to inverse power flows (e.g. charging instead of discharging).

Range Matching If specified, the range of possible values for the consuming component has to be equal or wider than the range of the incoming values by the producing component. With respect to Smart Grid simulation composition such range checks are also useful (**Validity Aspect 3** – *Ranges must match*). For example, the EV model (see Chapter 3.5.2) outputs negative power values in case of power feed-in and positive values indicate that the EV draws power, e.g. during charging. However, not every power flow simulation may be able to handle/interpret negative load values correctly. If both simulation models provide range information for their data such incompatibility can be detected during the composition process. Obviously in this case, a manual mapping[2] of the incompatible data flows may be required.

Inputs Satisfied CODES checks if every input of the components is satisfied by a corresponding output that meets the other constraints. Obviously this is a required check for every domain as a model can only produce viable outputs if required inputs are made available (**Validity Aspect 4** – *Inputs satisfied*). There may be optional inputs. For example, in a PV model the solar irradiance may be internally modeled in a very simple way. This internal value may be optionally overwritten by providing the input from a more sophisticated climate model.

Behavior The behavior of components can be defined as state machine. A simulation component produces output after a state change. State changes happen when input from another component is received or a time interval has elapsed. "It is checked if every component's input arrives at certain time intervals as otherwise the receiving component cannot proceed“ [TS07]. This means that CODES also validates the aspect of data availability (when is data available) whereas the time semantics in the OpenMI standard (see below) describe the time point or span for which the data is valid (when is data applicable). However, mosaik does not capture the internal state relationships of the models, as mosaik focuses on the composition of models for the physical topology. Outputs of these models are available continuously (physical interconnections are of continuous nature compared to discrete data packages exchanged within the informational topology). Also, state changes do usually occur on a random basis (e.g. EV arrival/departures, stochastic loads or fluctuating renewable energies) without external input, making a state based validation of compositions non-trivial or even impossible. However, there are cases where availability semantics are also applicable, e.g. to define that a certain forecast or control schedule has to be provided until a certain deadline has been reached. This aspect (**Validity Aspect 5** – *Temporal input availability*) is therefore covered in more detail in Section 7.8 .

[2] As will be shown later, this mapping feature will also be required to meet the requirement of composing models with different levels of abstraction [**R 25** – Multi-Resolution Composition].

Limitations

The CODES approach is not used directly for this layer as it does not meet all the requirements of mosaik. In the first place, the CODES ontology as well as the composition grammar do not account for the simulator-model-entity structure that mosaik assumes [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs]. CODES only knows a single *Component* type. Furthermore, the definition of complex data types is not considered [**R 15** – Transmission of Complex Data Types]. Also, definition of static attributes [**R 11** – Static Data Access] is not possible.

The impact on mosaik

To keep mosaik as simple as possible [**R 1** – Keep It Simple] the final implementation of mosaik should integrate the different conceptual layers seamlessly. As CODES uses different specification artifacts (the COSMO OWL ontology and the composition grammar), these will not be reused but a single semantic metamodel will be created for mosaik such that the semantic as well as the scenario layer can be implemented seamlessly using the Xtext framework, as pointed out in the beginning of this chapter. As discussed above, all four types of checks from the CODES approach can be applied to the Smart Grid domain and will therefore be considered when designing the semantic metamodel. Section 7.12 will discuss the presented concept of the semantic layer and point out how the information required for performing the different validity aspects can be defined. It has to be pointed out that checking the validity of compositions is not the task of the semantic layer as the compositions are not yet defined. But the semantic layer must offer all information that is required to perform these checks on the scenario and composition layer.

7.3.2 CoMo

ComponentModels (CoMo) is a Java based tool that allows the composition of platform independent simulation models to a single, executable, platform specific model. It implements a formal composition concept presented by Röhl [RÖ8]. The concept allows to create formalism-independent interface descriptions of models and associate these with corresponding implementations in a specific modeling formalism. Based on these abstract definitions, the simulation models can be composed and the composition can be validated. Hence, CoMo meets requirement [**R 24** – Formal Simulator Description]. The CoMo platform allows to specify these definitions in XML and corresponding converters can be implemented to convert the specific models into the PDEVS[3] formalism so that the overall composition can be actually created and executed. Due to the use of XML, CoMo allows the definition of complex data types as demanded by [**R 15** – Transmission of Complex Data Types]. The use of so called *Configurators* allows to forward parameters to subcomponents of a component and thus allows to define configuration parameters in a single place [**R 29** – Central Parameter Definition] without having to specify it in the models specific formalisms.

[3] Parallel DEVS - A formalism for the specification of parallel discrete-event simulation models.

Limitations

Nonetheless, the CoMo approach is different from the one of mosaik as the conversion of modeling formalisms is not an objective of mosaik but rather the reuse of simulation models including their simulators. Therefore, [**R 22** – Discrete-Time Simulator Integration (fixed step size)] cannot be met, although time-stepped models can be represented with CoMo by scheduling events in equidistant time points. CoMo only offers the notion of a *Component* but does not allow to describe the simulator properties or the further decomposition of models into entities [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs], although this could be represented in CoMo by hierarchical components with the root being the model and the child components the entities. Finally, CoMo does not distinguish between dynamic and static data which can be used for defining a composition as described in Chapter 8 [**R 11** – Static Data Access].

The impact on mosaik

For these reasons, the CoMo metamodel will not be used directly for mosaik. However, there are aspects of CoMo that mosaik can benefit from, for example the use of a common type definition to ease validity checking for compositions or the idea to group data into logical ports. CoMo also introduces the notion of *Roles* which can be compared to the types used for origin/destination checking in the CODES approach discussed above and which were considered useful for Smart Grid composition as well. More detailed references and comparisons to CoMo and CODES will be given below, when the different parts of the mosaik metamodel will be developed.

7.3.3 IAM

Soma et al. [SBO$^+$06] present an approach for an Integrated Asset management (IAM) system in the domain of the oil and gas industry. The objective is to support a user with a single, easy-to-use interface to specifying and executing different workflows, such as reservoir simulations or economic evaluations. The data sources for these workflows are very diverse, ranging from real-time measurements from temperature, flow, pressure and other sensors on actual physical assets such as oil pipelines to abstract data such as maintenance schedules, market prices and so on [SBO$^+$06]. The central idea of the approach is a domain-specific modeling language that allows to capture all information about the available data sources. The corresponding domain-specific metamodel (from which a graphical modeling environment is generated, see Chapter 8.2) can be split into three different parts. The first part, called "data schema" allows to define primitive as well as complex [**R 15** – Transmission of Complex Data Types] data elements and transformations among these. The latter need to be specified as sub-routines in C. The second part, the "data composition schema", is used to define connections (maybe using a transformation) between the available data types. Finally, the "domain model schema" lets the user specify the elements of the model database (reflecting the status of the assets) and how these can be updated by the results of a composition.

Limitations

Although the metamodel of IAM can be used to formally describe the available data sources [**R 24** – Formal Simulator Description], it is not clear how simulations can be integrated into the approach, as there is no notion of a time-advance mechanism for a component (e.g. fixed-step size, et...).

The impact on mosaik

The idea of separating the definition of domain specific data types, domain assets that are made up of these types and the actual definition of compositions seems to be a reasonable approach. The semantic metamodel presented in this section covers the first two aspects. It is divided into a domain specific data model (see Section 7.4.1) (comparable to the IAM data schema) and the simulator descriptions (see Section 7.4) that reference elements of the data model. The scenario metamodel introduced in the next chapter (scenario layer) can be compared to the data composition schema of the IAM approach.

7.3.4 OpenMI

As mentioned in the last chapter the OpenMI standard defines an interface for linking simulation components. This interface allows a generic access to the data of linked components. On this level, nothing more than the data type is known about the data. Additional semantics can be added by so called *extensions*. These are additional interfaces a linkable component can implement [Moo10, p.16]. OpenMI defines the semantics of the available data in terms of quantities (what it represents), geometry (where it applies) and time (when it applies).

What To describe the physical semantics of the data a so-called value definition is used. Each value definition is either a quantity combined with a unit or a quality combined with a list of allowed values for the quality. In the environmental domain examples for quantities are water level in meter or the amount of flow in cubic meter per hour. An example for a quality is the type of land use (e.g. forest, road, residential area and so forth). In the Smart Grid domain quantitative and qualitative values can also be found. A quantitative value, for example, is the capacity of a battery storage (e.g. kWh) or the flow of power through a line or into a battery (e.g. in kW). Qualitative values could be the location of an electric vehicle (e.g. *home* or *work*) or the state of a circuit breaker (*open*/*closed*) in the power grid. The metamodel should therefore support the definition of qualitative and quantitative values for both, the data provided and consumed by the entities as well as the simulator and model parameters to classify the data more precisely. For the parameters, this additional information allows the Scenario Expert to easily identify valid parameter values, e.g. compared to a regular integer attribute which could take all possible integer values. For the entity data, this allows to validate if data flows between any two entities have compatible units (**Validity Aspect 6** – *Data flow units match*).

Where In order to describe the space where the values of a model component apply, OpenMI uses an ordered list of elements, each of which having a unique ID and an optional geo-coordinate in a given co-ordinate system. Such an element could be a point, a line, a polygon or other 2 to 3 dimensional geometric shapes. This is different from the requirements for the analyzed Smart Grid scenarios (see Chapter 3.2) where the entities are related to each other in a strictly topological fashion, i.e. determined by electrical topology. This aspect is therefore not relevant to the semantic metamodel.

When OpenMI allows to define points in time for which the outputs provided by a component are valid. This can be specified as either a time point or a time-span. In the Smart Grid domain such information may be required for describing the validity of scheduling commands or other forecasts, for example, considering a component that delivers a load-curve which is a forecast of its future behavior. Without time semantics it is not even clear that the list of load values is a forecast. This aspect is covered in more detail in Section 7.8 (**Validity Aspect 7** – *Temporal data validity*).

The definition of so called *DataOperations* allows the conversion of data by means of temporal, spatial or other user defined operations. These data operations could probably be used to meet [**R 25** – Multi-Resolution Composition]. As OpenMI follows a pull based approach, i.e. a component requests (pulls) data from another component for a specific time point, these data operations can also be used to integrate simulation components with different temporal resolutions [**R 19** – Different Temporal Resolutions].

Limitations

As stated above, the semantics of the available data can be queried via the extension interfaces. As a consequence, the simulation component needs to be initialized in order to access this information being somewhat in conflict with the definition of composability which requires validity checking prior to run-time. Therefore, OpenMI does not fully meet [**R 24** – Formal Simulator Description]. Also, OpenMI is limited to Microsoft .NET components and the formal simulator description should ideally be implementation independent. To the best of the author's knowledge, OpenMI does not provide a mechanism to define static data [**R 11** – Static Data Access], i.e. time invariant attributes that could be used for the mosaik scenario specification as demanded by [**R 7** – Attribute Based Composition]. OpenMI exchanges values as a two dimensional array representing time and space. However, complex datatypes as demanded by [**R 15** – Transmission of Complex Data Types] are not possible. Also, a mechanism for defining the parameters a model offers is not known [**R 29** – Central Parameter Definition]. Only a mechanism to "calibrate" models at run-time is mentioned [Moo10] based on the defined data flows. Again, the distinction between models and their entities is not possible as OpenMI only defines a *ILinkableComponent* interface. Hence, requirement [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs] is not met.

The impact on mosaik

Similar to CODES and CoMo approach discussed above, the OpenMI metamodel is not used directly for mosaik due to the identified limitations. However, the classification of data types into qualities and quantities will be picked up in the mosaik semantic metamodel.

7.3.5 BOM

The SISO 003-2006 Base Object Model (BOM) standard [Sim06] extends the HLA standard (see Chapter 6.2.3) by a conceptual modeling description. The main contribution of this extension is the ability to represent "discrete patterns of interplay among conceptual entities within a simulation environment" [Tol10]. As BOM relies on XML to describe these patterns, BOM descriptions are implementation independent. Conceptual entities and conceptual events are used to describe state machines representing the internal behavior of components and interaction patterns that describe interactions among these. As already discussed for the CODES approach, mosaik does not intend to describe the state machines for the participating simulation models as this is not relevant for the composition of the physical topology. However, if the scope of mosaik is extended to include the composition of agents or other components that make up the informational topology, the BOM standard seems to be a good basis for describing the potentially complex interactions among agents.

7.3.6 Related Work Summary

Table 7.1 gives an overview of the discussed works that deal with semantic simulation composition in general or in other domains. CODES and CoMo are domain independent approaches for composable modeling and simulation and could be applied to the Smart Grid domain as well. However, all approaches are missing the required decomposition of models into a variable number of entities [R 16] and the explicit definition of static data [R 11]. Also, most approaches do not consider the definition of complex data types [R 15]. With the exception of the BOM standard, all approaches have contributed ideas and concepts to the development of the mosaik semantic metamodel that will be presented in the next sections.

Table 7.1: Related work dealing with semantics for simulation composition

Approach	Domain of origin	Domain specificness	R 11	R 15	R 16
CODES	?	n/a	-	-	-
CoMo	?	n/a	-	+	-
IAM	Petroleum Industry	Medium	-	+	-
OpenMI	Environmental Modeling	Medium	-	+/-	-
BOM	Military	Low	n/a	n/a	n/a

7.4 Simulator Decomposition and Type System

A major criticism of the related works was the missing decomposition of models into entities which is required to compose models of complex systems such as the power grid. In the mosaik semantic metamodel shown in Figure 7.3, the classes[4] *Simulator*, *SimulationModel*[5] and *EntityType* account for this decomposition and lay the foundation for the remainder of the metamodel. Each of these classes has a name attribute that allows to identify it unambiguously within its context. To ensure unambiguity, Simulator names must be globally unique, SimulationModel names must be unique within the context of their simulators and EntityType names must be unique within their model. These and other constraints cannot be expressed using UML alone and are therefore defined as additional OCL constraints and presented in Appendix C.

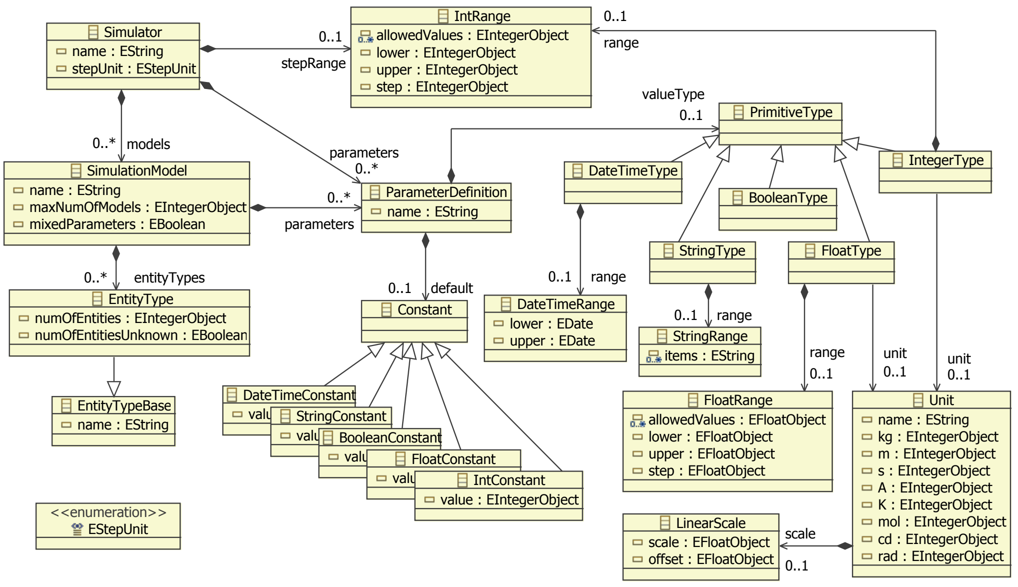


Figure 7.3: Metamodel for basic simulator structure and parameter definition

For a Simulator an arbitrary number of available *models* can be defined. The parameter *maxNumOfModels* is used to indicate the maximal number of model instances that the simulator can execute for this type of model. Although not shown in the diagram, the implementation will later allow to use the symbol '*' to specify the maximum integer value in order to indicate an unlimited number of possible model instances. A SimulationModel can define an arbitrary number of EntityTypes which define the type of objects that are explicitly modeled. The instances of these EntityTypes will be the building blocks for describing scenarios on the scenario layer (see Chapter 8). For each EntityType it can be defined how many instances of this type are created for each model instance [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs]. The *numOfEntitiesUnknown* attribute can be set to indicate that the number of entities is not known prior to run-time, e.g. because it depends on a configuration parameter such as a

[4] Similar to other terms, class names are written in *italics* at their first occurrence.

[5] To avoid any confusion with the term *model* in the context of simulation and metamodeling the class name *SimulationModel* is used in favor of *Model*. Although the latter would be sufficient in the context of the UML diagrams.

topology file in case of a power grid model.

As pointed out by Röhl [RÖ8] making a model parameterizable can increase the reusability as it can be adjusted to a wider range of settings [RÖ8]. The same applies to the simulators. The *ParameterDefinition* class of the metamodel allows to define the available parameters. Each ParameterDefinition has a name and references a *PrimitiveType*. The latter is the base class of a type system that allows to specify the most common primitive data types. A ParameterDefinition can also specify an optional default value, indicating if the parameter is mandatory (no default value) or optional (default value is used). Each subclass of PrimitiveType (except for *BooleanType*) references a corresponding range class that allows to define the valid values for the defined parameter. For datetime, string and the numeric types a list of allowed values can be defined. Alternatively, for datetime and numeric types, a closed[6] interval of allowed values can be defined using the attributes *lower* and *upper*. The interval may be unlimited to one or both sides when the respective boundary is not defined. The step sizes that a simulator supports is also defined using the *IntRange* class. The unit of the *stepRange* is defined by the *stepUnit* attribute which can either be minutes or seconds. As discussed in Chapter 3.4 smaller step sizes are not within the scope of mosaik.

For the numeric types a *Unit* can be specified to add additional semantics to the parameter definition. For an EV model, a parameterizable battery capacity can be described this way by defining a parameter *capacity* in kilowatt-hours and a range from 0 to 100 kWh with a default value of 32 kWh, for example. A *step* attribute may be defined for a range to indicate that only some of the values in the interval are valid (e.g. the interval [1,10] with a step of 2 means that 1, 3, 5, 7, 9 are valid values). For each unit it can be defined how it is composed out of the SI base units[7]. The relation to an SI base unit is defined by a *LinearScale* class and the definition of the exponents of the 7 SI base units and the derived unit *rad* [MOD12b]. The definition of kW as a unit for electrical power, for example, can be expressed in SI units as follows: $1000 \cdot \frac{kg \cdot m^2}{s^3}$. The corresponding kW *Unit* object would have the attributes kg=1, m=2, s=-3 and a scale object with scale=1000 and offset=0. As the SimAPI does not provide unit information and mosaik relies on a reference model, expected to take place in the components implementing the SimAPI. The provided values have the units defined in the reference data model when they are received or sent by the simulation engine (**A 3** *– No central unit conversion required*). However, when the SI units are defined such a conversion can be done automatically in the SimAPI implementation, allowing to create generic adapters to other standards also defining SI units, for example the FMI standard (e.g. see Chapter 12.2.2.3).

OCL Constraints

The following additional OCL constraints, which can be found in appendix C, accompany the presented metamodel.

C 1 Simulator.name must be globally unique.

[6] This is sufficient for most cases. Support for open interval boundaries could be added later (e.g. by adding two boolean flags. To reduce complexity only closed intervals are supported in the scope of this thesis.

[7] `http://www.bipm.org/en/si/base_units/` (accessed 22 Feb. 2013)

C 2 SimulationModel.name must be unique within the context of its Simulator.

C 3 EntityType.name must be unique within the context of its SimulationModel.

C 4 The default value of a ParameterDefinition must be compatible to the valueType.

C 5 The default value of a ParameterDefinition must be in the allowed range of the valueType, if specified.

C 6 If DateTimeRange.allowedValues is defined, the lower and upper values must be *null*. If lower and upper range are defined $lower \leq upper$ must hold.

C 7 If FloatRange.allowedValues is defined, the lower and upper values must be *null*. If lower and upper range are defined $lower \leq upper$ must hold.

C 8 If IntRange.allowedValues is defined, the lower and upper values must be *null*. If lower and upper range are defined $lower \leq upper$ must hold.

C 10 Unit.name must be globally unique.

C 11 The step size of a simulator can only be positive.

7.4.1 Reference Data Model

As yet, the semantic metamodel allows to define the simulator structure and the required parameters for initializing it and its models. On the scenario layer described in the next section, a metamodel will be presented that allows to define Smart Grid scenarios that are composed of these simulation models (their entities to be precise). In order to provide semantics about what entities can be connected and thus avoid invalid compositions, the meaning of the entities' inputs and outputs has to be defined. What is needed, however, is a common understanding of the data that is to be exchanged between the entities. For example, it does not make sense to use the output of an entity providing the solar irradiance (e.g. see Chapter 3.2.3) as an input for the environmental temperature of a PV module. Although this may be syntactically valid, e.g. temperature as well as irradiance may be a float value or otherwise compatible numeric types, the meaning (semantic) is completely different.

The usage of reference data models to define a common semantic in the field of simulation composability is widely accepted and proposed in the context of the LCIM [WTW09] and by Tolk, Turnitsa and Diallo [TTD08], Turnitsa [TT08] as well as Mcdonald and Talbert [MT00]. Figure 7.4 illustrates how the reference data model serves as a common basis for defining the semantics and structure of the data exchanged with each simulator via the SimAPI. The formal descriptions of the simulators and their models (i.e. instances of the metamodel presented in this section) use elements of the reference data model to define the input and output data of the entities as will be shown in Section 7.5.

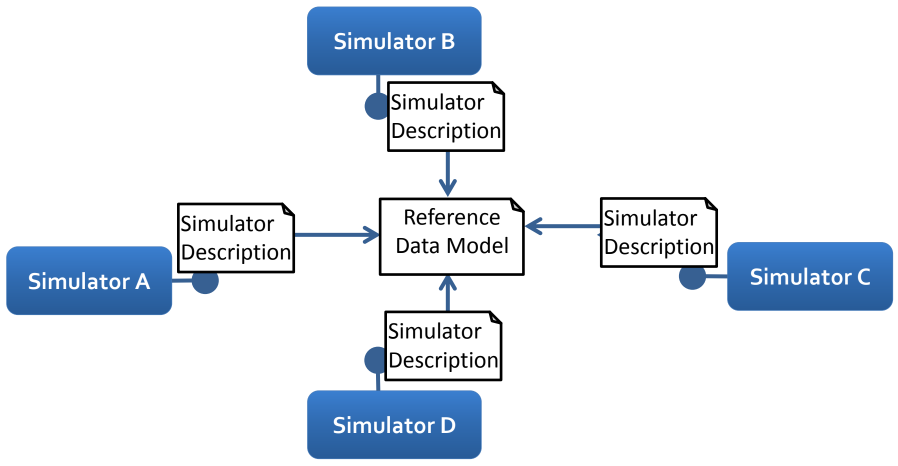


Figure 7.4: Reference data model for the semantic description of simulators

7.4.2 Ontologies

As Miler et al. [MHC07] point out, it is only feasible to define the meaning of something if the related terms are already defined and the relations between different terms are defined as well. Such a definition is called *Ontology*. An ontology is “an engineering artifact, constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary words” [Gua98]. Since the interest in ontologies began to rise around the year 2000 [GW10], a number of ontologies for different domains have already been defined in the past years [MHC07]. This also includes ontologies for the modeling and simulation domain, such as DeMO [SMH+10] and DeSO [GW10]. The DeMO ontology can be seen as a high-level simulation language supporting the different paradigms of state-, event-, activity- and process-based simulation [GW10]. The main purpose of DeSO is to provide a basis for evaluating discrete event simulation languages by using more abstract concepts from a foundational ontology called eUFO. For the mosaik semantic layer, however, these ontologies are not useful, as it is neither the intention to define the behavior of a simulation model nor to evaluate existing modeling languages. These modeling aspects are internal to the simulators and the syntactic layer abstracts from them by allowing to handle all simulators in a time-stepped fashion.

7.4.3 Ontologies versus Data Models

There is no metric or single fact allowing to clearly distinguish between a data model and an ontology [AGK06, SMJ02]. Based on definitions of the term “ontology” found in literature, Atkinson et al. [AGK06] propose five different characteristics to identify an ontology:

1. Conceptualization (an ontology is a model of relevant concepts of a real world phenomenon)
2. Explicit (these concepts and constraints on them are made explicit)

3. Machine Readable (the ontology must be machine readable to allow reasoning)
4. Based on First-order Logic
5. Shared (the ontology represents consensual knowledge, i.e. is not limited to a private individual but accepted widely)

However, at the same time Atkinson et al. [AGK06] point out that these characteristics are not sufficient as they allow to classify a too wide range of data models as ontologies. For example all computer based UML models are machine readable, intended to be shared among all stakeholders and may also contain first order logic if accompanied by OCL constraints[8]. Therefore, Atkinson et al. [AGK06] "propose to use the term 'ontology' to characterize models which are intended to capture some standard or universally applicable information of the kind that might be found in an encyclopaedia or widely accepted authority." Following this definition, an ontology must have a very wide acceptance range. But they also point out that information representations deserving the term ontology are for example domain ontologies which are representing "a standardized model of the respective domain."

7.4.4 Designing the Reference Data Model

In case of mosaik a Smart Grid domain specific ontology is needed to establish this common understanding. The IEC 61970 Common Information Model presented in Chapter 2.4 is often referred to as ontology [USR+13, RUA10, NDWB06]. Mosaik aims to make use of the CIM[9] to describe structure and meaning of data flows as it will be one of the core standards in the future smart grid [SG310, p.109]. Warmer et al. [WKK+09] state that existing models such as the CIM seem to be a good starting point for integration efforts. Especially with respect to the use of mosaik as a test-bed for control mechanisms it is a good choice to build upon existing standards. This way the control mechanisms can interact with the mosaik entities in a way that is a real-world standard and less effort is required for migrating control mechanisms to production environments. Therefore, the semantic metamodel has to be extended in such a way that it can capture the elements of the CIM as well as other data structures that are to be defined additionally.

Extending the semantic metamodel

At run-time it must be possible to actually interchange the data via the SimAPI. As will be shown in Chapter 11, the SimAPI implementation relies on JSON for interchanging data. This led to the idea of using JSON-Schema[10] for describing the elements of the reference data model. JSON-Schema is "a JSON based format for defining the structure of JSON data" [JSO10]. This is basically what is needed for the metamodel. As additional benefit, it is made sure that this way all specified data structures can easily be

[8] Beckert et al. [BKS02] define a translation of UML class diagrams with OCL constraints into first-order predicate logic to enable reasoning about UML models.
[9] As there are efforts to merge IEC 61850 into 61970 (see Chapter 2.4), the 61850 standard is not considered here.
[10] `http://json-schema.org` (accessed 25 Feb. 2013)

exchanged by the simulators. As the JSON-Schema specification is very complex and its syntax is quite verbose and difficult to read, the Orderly project [Ord10] developed a specification based on a subset of JSON-Schema. It can be converted to JSON-Schema but due to a different syntax and by using only a subset of the JSON-Schema specification it is "extremely easy to learn and remember" [Ord10]. As shown in the next section, this subset is sufficient for capturing the elements of the CIM, for example. This way the metamodel is kept simple while at the same time offering the opportunity to use the simple Orderly syntax for the implementation (see Chapter 11.2.1). Figure 7.5 shows the extension of the metamodel partially based on structures defined by the Orderly specification. Newly introduced classes are highlighted in a darker color. This visual highlighting will be used throughout the remainder of this thesis to ease diagram interpretation.

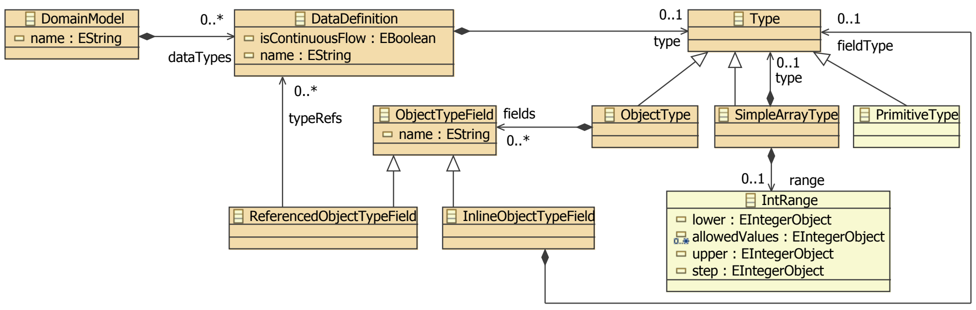


Figure 7.5: Metamodel extension for complex data types based on Orderly [Ord10]

The *DomainModel* class represents the domain-specific reference data model. The *name* attribute allows to create multiple reference data models and identify them unambiguously, e.g. when creating compositions based on different reference data models originally developed at different companies. The DomainModel can contain an arbitrary number of data types defined by the *DataDefinition* class, each of which must have a name that is unique within its DomainModel. This sounds trivial but is an important constraint as the ability to identify the data definitions unambiguously is essential to use them for validating a composition.

The PrimitiveType, already introduced above, is the simplest building block for a DataDefinition, representing a single value. Arrays of a specific type can be defined by the *SimpleArrayType*. An optional IntRange of positive integer values can be defined to indicate the minimal and maximal number of elements occurring in the array. The *ObjectType* class is used to specify complex types as they can be found in the CIM. An ObjectType has got a number of named *fields*. Each of these are either an *InlineObjectTypeField* which directly specifies an anonymous type structure or a *ReferencedObjectTypeField* referencing another DataDefinition. This class structure is a variation of the well known Composite Pattern [GHJ94, p.163] commonly used to create hierarchical data structures.

To sum up, the DataDefinition class serves two basic purposes. Firstly, it assigns a name to a *Type* making it identifiable for being used as reference in a ReferencedType-Definition. Secondly, as shown below (see Section 7.7) it allows to attach additional information to a type defined in the reference data model which is not required for the primitive types used for specifying a simulator or model parameter. Hence, it is not reasonable to attach this information directly to the Type class. The chosen design is the best option to meet all modeling requirements.

OCL Constraints

The following OCL constraints complete this part of the metamodel:

C 12 Each DomainModel must have a globally unique name.

C 13 Each DataDefinition must have a unique name within the context of its DomainModel.

C 14 The IntRange for SimpleArrayTypes must be positive (arrays cannot have negative length).

C 15 Each ObjectTypeField must have a unique name within its defining ObjectType.

7.4.5 Using the metamodel

It is important to point out that "an ontology will only help you in structuring your knowledge, but it will not replace the knowledge engineer" [RDA^{+}08, p.709]. To put it differently, the use of an ontology does not solve the composability problem on its own but the content has to be chosen carefully to achieve a composable solution. This content is not part of the discussion in this chapter as it only presents a metamodel that allows to define the content. However, in the evaluation (Chapter 12) a proposal for the reference data model (i.e. the content) will be given. In this section it is shown how the metamodel can generally be used to capture data structures defined in the CIM.

Currently, the CIM is expressed as a data model using the UML notation.[11] Hence, the metamodel extension for capturing the data structures defined in the CIM has to be capable of expressing any object oriented class structures. These structures have a number of concepts that the metamodel presented above allows to capture as follows:

Classes are represented by a combination of a *DataDefinition* and an *ObjectType*.[12]

Attributes are modeled by defining corresponding *fields* of the *ObjectType* with the field type being either a *SimpleArrayType* or a *PrimitiveType*. Both defined via an *InlineTypeDefinition*.

Associations are attributes where the type is another class definition. In the above metamodel associations are modeled for an *ObjectType* by defining a *FieldTypeDefinition* which is either an *InlineTypeDefinition* (an anonymous class) or a *ReferencedType-Definition* that is a reference to *ObjectType* defined elsewhere in the data model.

[11] "At some point in the future the CIM may be maintained and evolved using OWL" [NDWB06].

[12] In the Orderly specification the name *ObjectType* has been chosen because it defines the allowed/required attributes of an object serialized to a JSON string.

Inheritance is not supported by Orderly as it “purposefully ignores all features of JSONSchema which aren’t useful for validation” [Ord10]. It is assumed that an Orderly definition of a subclass defines all the attributes of the superclass. The semantic metamodel sticks with this approach. The disadvantage is that it results in redundant attribute definitions as an attribute occurring in two subclasses cannot be pushed up to the superclass. But on the other hand, the actual attributes of a class can be seen much more easily. In general the reference data model is easier to grasp when inheritance is not supported as all attributes are defined explicitly in each class and subclass.

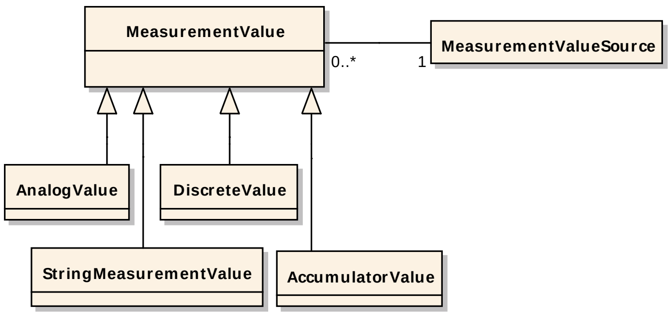


Figure 7.6: CIM MeasurementValue class illustrating an association to a super class

Figure 7.6 shows an example for a special case to be considered for associations. It shows a CIM data structure that can be used to represent measurements. The association from *MeasurementValueSource* references the superclass *MeasurementValue*. But at run-time the association will target any of the four subclasses. Although the metamodel does not support inheritance, such polymorph associations can be modeled by specifying more than one possible type definition for a ReferencedTypeDefinition (notice the 0..* multiplicity for the *typeRefs* role). In Orderly/JSON-Schema this construct is called *Union*. This way a number of classes that are valid targets of an association can be defined and this special case can be captured.

7.4.6 Entity Taxonomy

As discussed in Section 7.3 the CODES approach proposed by [TS07] allows to specify constraints for valid component (entity) types acting as origin or destination of data flows. As argued above this is also necessary for some cases in the Smart Grid domain. Figure 7.7 (A, B) illustrates this case using the aforementioned example. As the EV is controlling the charging process it offers two outputs that represent a power flow in kW. One flow is intended as input for an entity representing some connection point of a power grid model and the other represents the power that the EV charges or discharges from its battery. Negative values indicate a power flow from the EV to the respective target. Based solely on the data flow type the mapping cannot be created unambiguously. Both variants A and B shown on the left hand side of the figure are valid as the flows have the same type. Additional semantic is required to remove ambiguity and enable a valid automatic composition.

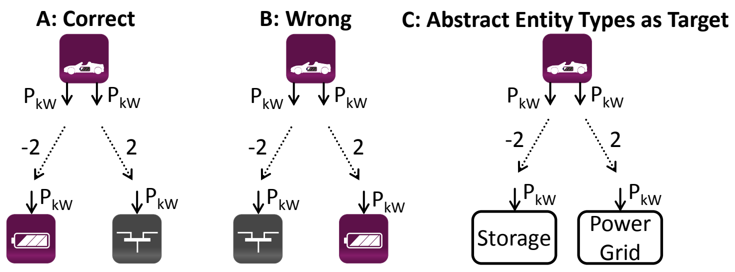


Figure 7.7: Example for ambiguous connections when no abstract entity type is defined as target

As proposed by Teo and Szabo [TS07], this additional semantic can be added by specifying what type of entity is valid for each data flow. Of course, the description of the data flows of an EntityType should not use another EntityType as these are specific for each simulation model. Therefore abstract entity types (as opposed to the model specific entity types introduced so far) have to be used. Obviously these abstract entity types cannot be defined freely by the different simulation developers as otherwise a matching will not be possible. These types must also be part of the reference data model. The top left part of Figure 7.8 shows how the metamodel is extended accordingly. The name *AbstractEntityType* was chosen to denote the non-existing, conceptional entity types. Examples for such abstract entity types are *EnergyConsumer, EnergyProducer, Storage, ElectricVehicle or Transformer*. Each AbstractEntityType can have an arbitrary number of super types via the *super* association. This way a taxonomy of abstract entity types can be defined in the reference data model. Of course, this taxonomy determines the composability of the entities on the scenario level. Therefore, the taxonomy design itself as well as the assignment of an AbstractEntityType to an EntityType of the simulator description has to be made thoughtfully. The evaluation chapter proposes such a domain-specific taxonomy (see Chapter 12.3.1). The other classes shown in the figure are introduced in the next section.

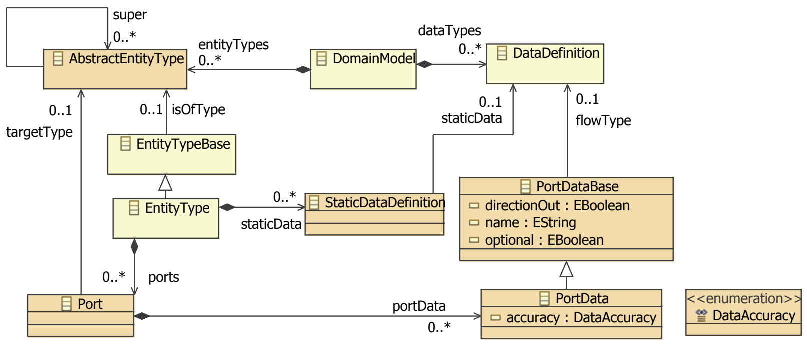


Figure 7.8: Metamodel for entity taxonomy and ports providing additional connection semantic

An additional OCL constraint is required to ensure that the taxonomy of abstract entity types is free of cycles. The *closure* operation introduced with OCL version 2.3.1 can be used for this:

```
C 16| context AbstractEntityType inv acyclicAbstractEntityTypes:
    super->closure(super)->excludes(self)
```

7.5 Defining Static and Dynamic Data

As yet, the semantic metamodel allows to create a reference data model. In this section the corresponding extensions are made so that EntityTypes can use the elements of the data model to describe their static data. As discussed before, entity attributes can be subdivided into static (time-invariant) and dynamic data (changing over time). Static attributes are defined via the associative class *StaticData* (see Figure 7.8) that references a corresponding DataDefinition and assigns a name to it that is used to identify the datum in the SimAPI implementation. The DomainModel itself does not define what DataDefinition is static or dynamic, as a position data type, for example, could be a static attribute for a transformer or a house but a dynamic attribute for an EV. To avoid semantic ambiguity an additional OCL constraint **C 18** ensures that a DataDefintition is used only once per EntityType.

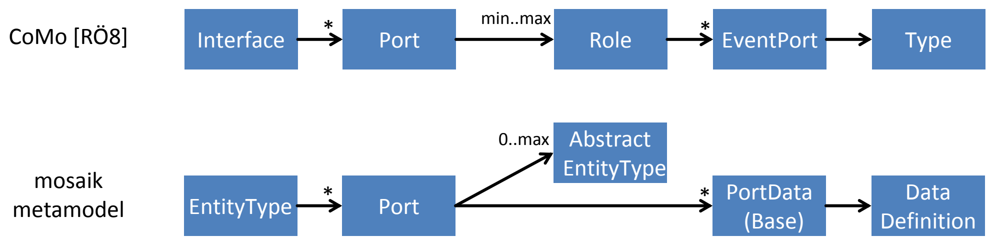


Figure 7.9: Metamodels of CoMo [RÖ8] and mosaik

The description of dynamic data flows is more complex as these are used for determining the composability of entities. Inspired by the concept of ports used in UML, Röhl [RÖ8] points out that ports can be used to decide about the compatibility of models (entities in case of mosaik) as these contain information about the required and offered data flows. Inspired by this idea and by the need to make use of the AbstractEntityTypes introduced in the last section, mosaik also employs the concept of ports. However, as shown in Figure 7.9 the structure is slightly different from the one defined by CoMo. An *Interface* in CoMo can be compared to the EntityType in mosaik with the difference that CoMo specifies additional bindings that bind interfaces (that are implementation independent) to specific model implementations. In mosaik the EntityTypes are implementation specific as they belong to a specific model that is integrated into mosaik. Hence, the binding takes place at composition-time when the mosaik engine instantiates the corresponding model via the SimAPI. A CoMo port specifies a *Role* and a minimal and maximal number of instances with this role that can be connected to this port. The Role is similar to the AbstractEntityTypes of mosaik. For the time being, mosaik will only allow to specify a maximal number of connections as this is sufficient for the identified use cases. However, it is possible to specify if a data flow (a referenced DataDefinition) is optional or mandatory which is similar to defining minimal values of zero or one but on a finer grained level, as CoMo does this per port only. In CoMo, a Role defines the data flows that are possible for this Role. For mosaik the AbstractEntityType does not define data flows but these are defined independently of it in a *Port*. The advantage of this approach is that the definition of AbstractDataTypes is independent of the level of abstraction of the entity type. The EntityTypes of a very detailed model of a power system can use the same types as those of a less detailed model. Only the data definitions need to be different. This makes the AbstractEntityType taxonomy less complex and easier to understand. Figure 7.8 shows the corresponding extension of the metamodel. Similar to CoMo the elements of the reference data model are not directly referenced from a Port (or a role in CoMo) as the data flows have additional attributes. These include a name used to unambiguously identify the flow at run-time in the SimAPI implementation, a direction flag indicating if it is an incoming or outgoing data and the optionality flag mentioned above. These attributes cannot be added to the Port as they are different for each data flow. Also they cannot be assigned to the DataDefinition as it will be used as input by one entity and as output by another. Hence, an association class *PortData* is required. Actually, the *PortDataBase* class from which the PortData is derived carries these attributes. Why this super class is needed will be explained in Chapter 8.5.4. The *optional* attribute only applies to input data flows and indicates if the input is mandatory or optional. In case of a PV module entity, for example, the solar irradiance may be an optional input if the entity has a simple internal irradiance model that can optionally be overridden by a more precise external irradiance value. The *accuracy* attribute allows to define how the value is calculated, following the four options discussed in Figure 6.7 above. This information, however, is not used to judge about the validity of a composition as this is likely to restrict composability too much as often alternative models are not available. Therefore, this attribute is only being used as an indicator for the quality of the results. During composition, a later implementation of the mosaik concept should point the user to potential quality problems in the following cases:

- When the accuracy of a used output is set to *unknown*.
- When an input data flow defines an accuracy flag that is different from the one of the output data flow that it is connected to.
- When the accuracy of an input is defined as *unknown* and two outputs with different accuracy values are connected to it.

In CoMo, the definition of compatibility of roles adds additional semantics to the ports. Informally spoken, two CoMo roles are compatible (can be connected) if for every event port of one role a type-compatible event port of the other role with opposite flow direction exists. Transferring this semantic to the mosaik metamodel classes implies that placing data flows into a port ensures that at run-time all data flows have to be satisfied by a single instance of another entity type. Figure 7.10 shows this additional port semantics. Entity Type A requires two flows *flow1* and *flow2* to operate properly. In the upper part of the Figure the Entity Types B and C each offer one of these flows. There may be cases where this is not the intended behavior and both flows have to be satisfied by a single entity instance. By placing both flows into a port (lower part of the Figure) this composition behavior can be forced. The formal definition of compatibility of ports is given in the next paragraph. It differs from the compatibility definition of CoMo in that mosaik allows two ports to be connected even though not all outputs of a port are connectable. The composition is assumed to be valid as long as all non-optional inputs can be mutually satisfied (see **C 24** defined below). Additionally, the *targetType* of the ports must be mutually compatible, meaning the EntityType of the opposite port must be a similar or more specialized subtype than the *targetType* requires. If a bus entity of a power grid has the *targetType EnergyConsumer*, for example, the *EntityType* of a potential other Port must extend the *EnergyConsumer* type or a subtype of it, such as *ResidentialLoad* (see Chapter 12.3), for example.

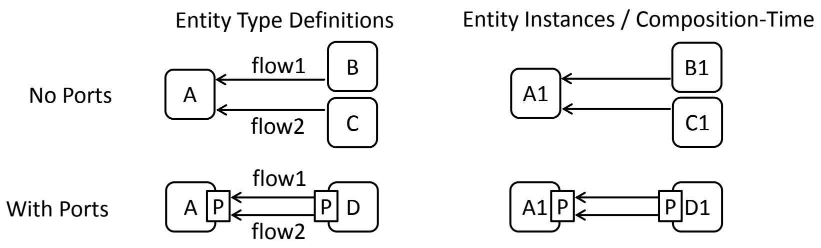


Figure 7.10: Using ports for providing additional grouping semantics

Finally, it has to be pointed out that the Port class has a *mandatory* attribute which indicates if the port as a whole has to be connected in a composition or not. Although an *optional* attribute for the PortData objects exists, the *mandatory* attribute is sill required. The reason for this is that making all data flows (=PortData objects) of a Port optional to indicate that the whole Port is optional influences composability. A Port defining a power flow, for example, may have P (active power share) as mandatory flow and Q (reactive power share) as optional (assuming it is zero). This is reasonable to increase composability with entities that do not provide a reactive power value. If P and Q are made optional, connections to entities with reactive power flows may also be possible. While this may not be critical in the given example, the additional attribute at the Port allows to model all cases precisely.

OCL Constraints

Again, a number of OCL constraints is required to ensure that the metamodel can only be instantiated in a valid way.

C 18 Ensures that any two static data definitions of an EntityType reference a different DataDefinition to prevent semantic ambiguity, as the *name* attribute of a Static-DataDefinition is used for identifying the attribute syntactically (via the SimAPI) but does not carry any meaning itself. Therefore it cannot be used to distinguish static data attributes.

C 19 Ensures that the names of all static data items defined for an EntityType are unique, allowing to unambiguously identified them via the SimAPI.

C 20 As Ports do not have a name, the names of the PortData objects must be unique across all Ports of their defining EntityType. Also, to keep model integration simple, the SimAPI does not make use of the Port concept. This constraint ensures that the data flows have unique names per EntityType (and not only per Port) such that these can be passed unambiguously via the SimAPI.

C 21 A DataDefinition must only be used once per Port and direction. Otherwise the semantic is ambiguous, as the *name* attribute of the PortDataBase class is used for identifying the attribute syntactically (via the SimAPI) but does not carry any meaning itself.

C 22 In addition to **C 21**, multiple input or output flows may only reference the same DataDefinition object, if the referenced *targetTypes* are different. They are different if one is not the same or a subtype of the other in the AbstractEntityType taxonomy. Specifying no *targetType* is not valid in cases with more than one input or output referencing identical DataDefinitions.

C 23 The optional attribute may only be set if the PortData defines an input. As outputs do not need to be connected to ensure validity, these have no optional flag.

For the Port class **C 24** specifies five additional functions. These functions will be used on the scenario layer (see Chapter 8.5.3) to determine if two EntityTypes have compatible ports. This includes checking if the target types are compatible [**Validity Aspect 2** – Origin/destination types must match], if the limits of all data flows match [**Validity Aspect 3** – Ranges must match] and if the ports can mutually satisfy their mandatory inputs [**Validity Aspect 4** – Inputs satisfied].

7.6 Dynamic Data Limits

The PrimitiveTypes that are the atomic building blocks for the DataDefinitions in the reference data model allow the definition of value ranges as introduced in Figure 7.3. These ranges should be used to define validity with respect to real world physics. The definition of a temperature type in Kelvin, for example, should limit the possible values to positive ones. When a DataDefinition is used for defining dynamic data of an EntityType additional limits can be specified that override this default range specification with model specific values. In the example of the temperature type, the model may only be capable of or validated for a smaller range of temperature. Figure 7.11 shows a corresponding extension to the PortData class that implements this additional range definition inspired by the CODES approach (see Section 7.2). At composition-time it has to be checked if the value range of the consuming entity is equal or wider than the range of the incoming values by the producing component (see function *limitsMatch* defined in OCL constraint **C 24** introduced below). If the DataDefinition defining a DataFlowLimit uses an ObjectType, the *ObjectTypeFieldReference* class is used to recursively specify a path to a PrimitiveDataType that is directly or indirectly (via nested ObjectTypes) defined.

OCL Constraints

Once more, additional OCL constraints are required to ensure validity of the metamodel instances:

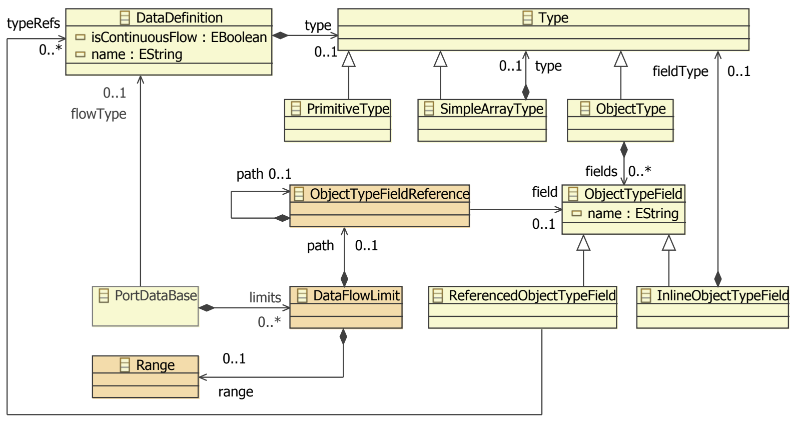


Figure 7.11: Specification of valid ranges for dynamic data

C 24 The OCL function *limitsMatch* has been added to this constraint and *mandatoryInputsSatisfied* has been refined to use this function to ensure compatibility of defined data limits.

C 28 If a DataFlowLimit defines a path, it must reference an ObjectTypeField that directly or indirectly exists within the referenced ObjectType.

C 31 If an ObjectTypeFieldReference *A* defines a path, this path must reference an ObjectTypeField that exists within the ObjectType referenced by *A*.

C 29 The DataFlowLimit must be specified for PrimitiveTypes only. For ObjectTypes each (primitive) field must be limited separately.

C 30 The defined range is type compatible to the type referenced by the DataFlowLimit.

7.7 Domain-Specific Constraints

There may be situations where the reference to a DataDefinition alone is not sufficient to specifying a valid composition. Figure 7.12 (A) shows such a situation where a PV entity can still be connected to different voltage levels. Obviously, the PV entity is modeled for a specific voltage level (typically low or medium voltage). But if the reference data model defines a power flow type in Watts or Kilowatts, this is not enough semantic to unambiguously determine the allowed connection points, as node entities on all voltage levels of a power grid model use the same definition.

Although the Scenario Expert may be aware of this ambiguity, mosaik will allow to define additional constraints on the data flows to resolve these issues and prevent him or her from specifying invalid compositions. The voltage-level information in this case can be provided by the PV system as static data attribute, for example. If the entities representing the connection points of the power grid provide the same voltage level information, an automatic check can be performed prior to executing the simulation to ensure that the composition is valid. As these constraints have to be checked at composition-time, i.e. before the simulators are stepped, these constraints can only be

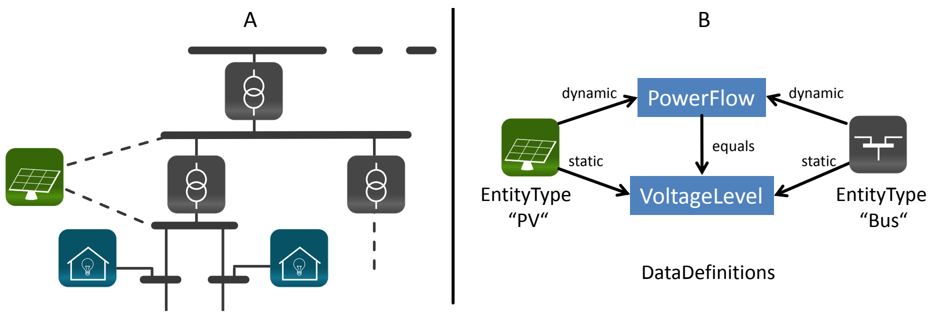


Figure 7.12: Example for a modeling situation requiring domain-specific semantics

defined for the static entity data. Figure 7.13 shows the corresponding extension of the reference data model.

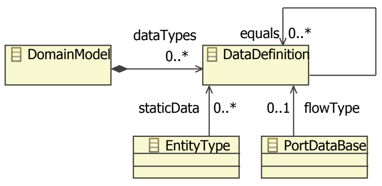


Figure 7.13: Domain specific constraints for dynamic data

For each DataDefinition *A* it is possible to reference an arbitrary number of other DataDefinitions as constraints *C1, C2, ...* via the *equals* association. If these other definitions are used as static data by two EntityTypes *E1, E2,* a data flow based on the definition *A* can only be established between these if the static data values of *C1, C2, ...* match for both entities. To illustrate how this part of the metamodel is used, Figure 7.12 (B) shows how a constraint is modeled and used in the PV example. Currently equality is the only possible operation. Others may be added later if required by introducing an association class into the *equals* association which specifies the comparison operation. As such constraints cannot be validated prior to run-time (static values must be available), this is not defined as an OCL constraint on the metamodel but part of the composition layer discussed in Chapter 9.

7.8 Temporal Semantics

As already discussed in the related works for this layer (see Section 7.3), there are at least two aspects related to temporal semantics of data flows that can be distinguished. The first is related to the availability of data. A simulation model can make certain data available at every simulation step or only sporadically. Once data is available, the temporal validity of the data has to be considered. It may be valid for the time point when it became available only, or until the next data is made available or for a specific time point/span only.

Figure 7.14 shows a scenario that is comprised of different representative types of entities and an arbitrary control mechanism. The data flows between the entities can be categorized as follows:

Informational Flows (A, C) provide information about the state of a simulated entity (A) and are thus available at every simulation step (continuous availability) and valid for this step. They may be used by the control algorithms to determine the next actions. In the case of environmental models (C) there may also be both, continuous flows to the control strategies (e.g. the current weather conditions) as well as sporadic flows, e.g. simulated weather forecasts that are made available at certain points in time (a preview on future states). The latter, however, may also be available as continuous flow only changing its values and temporal validity sporadically (e.g. a forecast is provided at every step but updated only sporadically).

Command Flows (B) include commands issued by control algorithms to a number of entities that can be controlled. Although depending on the used control strategy, these commands do usually occur at certain points in time (sporadic availability). Temporal validity for this type of data flow is complex as it depends on the type of command. Absolute commands, such as a switching command (e.g. open/close), may be valid until the next command is issued, whereas relative commands, such as reduction or increase of electrical production/consumption by a certain amount relative to the current state (e.g. increase by 5 kW), should only be processed once by the receiving entity. The point in time when the command is to be processed may vary as well. A schedule submitted by an EV charging strategy, for example, may contain numerous commands for future time points. The schedule may be valid until a new schedule is received (same semantics as absolute commands such as switch positions). This shows the variety of aspects to be considered for command flows.

Physical Flows (D, E) describe physical dependencies between entities. These can be data flows representing the flow of electrical power or other physical flows provided by environmental models, such as solar irradiance, for example. Data flows of this type are continuous, i.e. they are available at every time-step. As all physical elements of the Smart Grid are exposed to the environment in some way, environmental models do not only occur in combination with PV models. The air temperature, for example, may also influence EV models (e.g. battery state and consumption) or grid assets such as power lines or transformers. They are valid for the current time step.

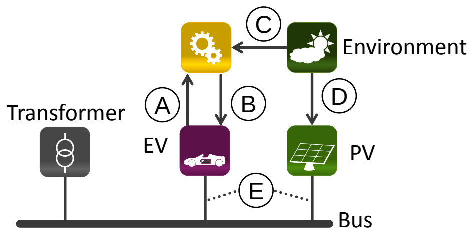


Figure 7.14: Types of data flows between the simulation entities

Table 7.2 summarizes the characteristics for the different flow types that have been discussed above, using the example shown in Figure 7.14.

Table 7.2: Temporal classification of different flow types

Flow Type	Availability	Temporal Validity
Informational	Continuous/Sporadic	For current step/ Schedule-like
Command	Sporadic	Until next update/ Only once/ Schedule-like
Physical	Continuous	For current step

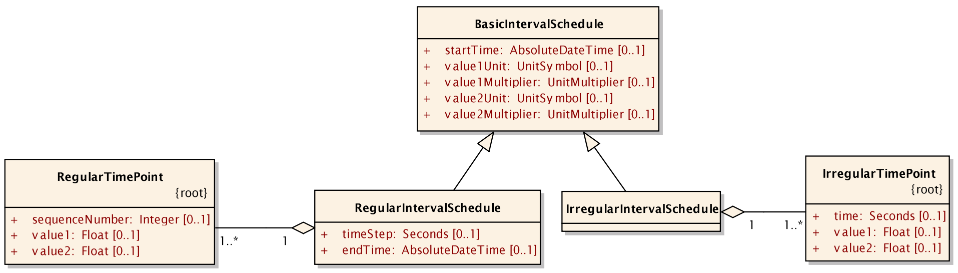

BasicIntervalSchedule
+ startTime: AbsoluteDateTime [0..1]
+ value1Unit: UnitSymbol [0..1]
+ value1Multiplier: UnitMultiplier [0..1]
+ value2Unit: UnitSymbol [0..1]
+ value2Multiplier: UnitMultiplier [0..1]
RegularTimePoint
{root}
+ sequenceNumber: Integer [0..1]
+ value1: Float [0..1]
+ value2: Float [0..1]
1..*
1
RegularIntervalSchedule
+ timeStep: Seconds [0..1]
+ endTime: AbsoluteDateTime [0..1]
IrregularIntervalSchedule
1
1..*
IrregularTimePoint
{root}
+ time: Seconds [0..1]
+ value1: Float [0..1]
+ value2: Float [0..1]

Figure 7.15: CIM classes for modeling schedules

7.8.1 Validity

For command flows it is assumed that temporal validity information is given by the commands data definition **A 4** – *No explicit temporal validity*. Figure 7.15 shows how such a data definition for a schedule looks like using the CIM ontology. Here the *startTime* attribute of the *BasicIntervalSchedule* implies the start of the schedule's temporal validity and the *time* attribute of the *IrregularTimePoint* is assumed to have the semantics as defined in the attributes comments of the CIM ("The time is relative [to] the BasicTimeSchedule.startTime."). In other words, the semantics are either modeled as attributes of the data definition (startTime) or implied by the standardized meaning of the used ontology and do not require explicit representation in the reference data model. The same assumption is made for sporadic informational flows such as forecasts. These must therefore use data definitions that are known to be forecasts. For physical flows it is assumed that data is valid for the current time step.

7.8.2 Availability

Availability semantics can only be partly based on conventions of the reference data model. A distinction between physical flows and informational/command flows can be made in the data model as this is related to the semantics of each data definition (e.g. compared to a schedule type, a power flow is a physical flow by convention). The DataDefinition class has a boolean attribute *hasContinuousAvailability*[13] which defines if it is a physical flow with continuous availability or a command or informational flow with the possibility of having non-continuous availability. This very basic flag

[13] For DataDefinitions referenced in field definitions of ObjectTypes this additional attribute is ignored at composition-time and the continuity is defined by the root DataDefinition directly referenced by the DomainModel.

is essential to optimize the communication on the technical layer, for example. To minimize the data that has to be transfered on this layer, one option is to transfer only deltas (i.e. value changes). Without the information of continuity, a receiving component does not know if not receiving values for a certain data definition means that the value stays the same (i.e. for continuous flows) or if it was available only once (i.e. for non-continuous flows).

A further description of the availability of non-continuous flows has to be defined in the context of the EntityType using that data definition as it may vary from model to model. To find a syntax to define this behavior different approaches from other projects/domains are being discussed briefly.

UNIX crontab [The97]

GridLAB-D (see 4.1.2) uses the UNIX crontab standard to define so called *Shapers* [Cha09]. These are objects that define load shapes (curves). This standard allows to specify specific time points in the resolution of minutes. Wildcards can be used to specify time intervals or a list of time points, for example $0\ 0\ 1, 15 * 1$ "would run a command on the first and fifteenth of each month, as well as on every Monday" [The97].

ASAM FIBEX [Ass11]

The Association for Standardisation of Automation and Measuring Systems (ASAM) has defined a standard for modeling the structure and communication behavior of ECU (Electronic Control Unit) network systems, called FIBEX (Field Bus Data Exchange Format). These networks consist of a number of ECUs that interchange signals (values) grouped into units called FRAME and/or PDU (Protocol Data Unit). The standard supports the description of various network systems used in industry, including CAN, Flexray, LIN and MOST. The protocols employed by the different systems can be split into:

non-time-triggered/event-triggered (e.g. Ethernet, CAN) where the ECUs can try to send a FRAME/PDU any time, e.g. when a calculation has finished or a sensor detected a change in the environment.

time-triggered (e.g. Flexray, LIN) where well-defined slots exist that are repeated in fixed time intervals, in which a specific message can be sent. This way deterministic communication behavior can be achieved.

Besides the definition of technical aspects of these systems, the FIBEX standard defines data structures that allow to describe the timing behavior of the FRAMES and PDUs, i.e. when these are sent or received by an ECU. This is very similar to the availability semantics that can be described for simulation data. Actually, when performing HIL (Hardware-In-The-Loop) tests [LW00] involving such bus systems, these frames are indeed the inputs and outputs of simulation models. The available timing types are:

ABSOLUTLEY-SCHEDULED-TIMING It allows to specify the absolute positions (slots) and cycles in which a message is sent. This is done by specifying the slot index (each cycle is subdivided into a fixed number of time slots), the base-cycle and the cycle-repetition. Cycles are numbered in a modulo fashion, e.g. repeating every 64 cycles in case of Flexray. E.g. base-cycle = 4 and cycle-repetition = 2 means that a message is sent every second cycle starting from cycle 4.

RELATIVELY-SCHEDULED-TIMING As the LIN protocol is less complex than Flexray, a simple relatively scheduled timing is used to specify the order in which messages are sent, e.g. message A after message B.

CYCLIC-TIMING For non-time-triggered protocols (and for special segments of the Flexray protocol) cyclic timings are used to define intervals in real time in which messages are sent, e.g. send message A every 2 seconds. The interval value is specified in the XML standard *duration*[14] format and thus can range down to microseconds or less.

EVENT-CONTROLLED-TIMING If the messages are not sent periodically, this timing type allows to specify conditions to trigger message transmission. These conditions can either relate to system states (e.g. car key turned to ignition position) or signal states (e.g. message B has been received).

Applicability for Smart Grid simulation

The crontab standard could be used as it is to describe the timing behavior of periodic/deterministic data and for simulators with a step size of a minute or more. The standard syntax could be extended by an additional field for seconds.

The FIBEX standard does not have the flexibility of the crontab wildcards but provides concepts to describe other timing types such as event-triggered timings as well as relative timings. Although the standard is specific to bus systems (e.g. the available conditions for events, etc..), the basic ideas of these timing types can be applied to the Smart Grid domain. The event-triggered timing could be used to specify that data is only provided sporadically when a model-internal event occurs (as models are black-boxes, internal events cannot be predicted and are to be considered stochastic/sporadic) or after a certain input has been received (signal event). The relatively-scheduled-timing could be used to define a fixed order in which commands have to be submitted (e.g. data A is provided before data B). Finally, when considering each step of a simulation as cycle, the absolutely-scheduled-timing of FIBEX could be used to define timings in relation to the number of simulation steps. For example, a weather forecast that is provided or expected at the beginning of each day could be encoded for a simulator step size of 15 minutes with a base-step of 0 and a step-repetition of 96 (= 24 hours). However, both values depend on how the simulator and the model are parametrized as the start time of the simulation as well as its step-size may be adjustable. So the base-step cannot be a simple step count but must be an absolute value with respect to the simulation time, e.g. 00h 00m 00s to define midnight. The first availability of the data would then be the first step at which the simulation time (see Chapter 2.1.3) reaches the value of base-step. The step-repetition also has to be a value in seconds or minutes such that this timing type is just a less complex version of the crontab approach.

[14] `http://www.schemacentral.com/sc/xsd/t-xsd_duration.html` (accessed 28 Feb. 2013)

Conclusion

Although both, FIBEX and crontab, are applicable to the field of Smart Grid simulation, the mosaik semantic metamodel will be limited to the *hasContinuousAvailability* flag introduced above. More advanced availability semantics are not considered in the scope of this thesis as the known models do not require this. Models that are not provided with schedules at fixed points in time are expected to idle or fall back to a default behavior if no valid schedule is available. Data flows for which the flag is set to true have to be provided/received at every time step.

7.9 Aggregation and Interpolation

At run-time, data flows between entities cannot always be processed in a simple copy fashion. There are some cases where aggregation and interpolation functions are applicable. This is mainly related to the possibility of having entities that are simulated by simulators with different step sizes [**R 19** – Different Temporal Resolutions]. Although processing the data flows is the task of the composition layer, the semantic layer has to provide information about what aggregation or interpolation function to employ.

Aggregation

Figure 7.16 depicts the three situations in which aggregation functions may or have to be applied. The symbol at the bottom represents a bus entity in a low-voltage power grid to which PV modules are connected to.

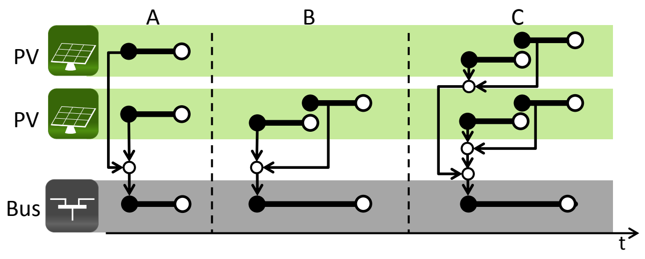


Figure 7.16: Data flows where aggregation functions can be applied

The first situation in which an aggregation function has to be used arises when an incoming data flow of an entity is connected to more than one entity providing the corresponding data (A). Mosaik expects that this aggregation is done within the simulation model (the SimAPI implementation) of the consuming entity. The reason for this is that some aggregations may require a complex, model specific handling that cannot be provided by mosaik. For example, in case of the bus entity, PU and PQ generators (see Chapter 2.3.3) have to be summed up separately within the power flow model as these have to be treated differently. If an EntityType can cope with multiple connections, this is indicated in the semantic simulator description by the *max_connections* attribute of each port of the EntityType (see Figure 7.8).

Aggregation functions may also be required in cases where only a single entity is providing data to another entity if the providing entity steps faster than the consuming entity (B). This is depicted in situation B of the example. In this case, instead of only using the value that is provided when the time points match, it may be desirable to use an average value of all output data that has been produced in the same interval. Finally, situation C depicts the combination of these cases. As both situations can be handled independently of each other this does not imply new requirements. The aggregations in situation A are handled in the simulator for each consuming entity and situation B will be handled by mosaik by aggregating the data flows for each producing entity.

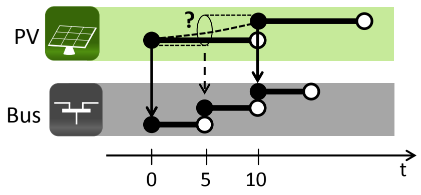


Figure 7.17: Data flows where interpolation functions can be applied

Interpolation

Figure 7.17 shows a case where the consuming entity steps faster (the opposite of the aggregation case). In this case the bus entity is now simulated at a finer resolution (e.g. with a step size of 5 minutes). This may be the case, as other entities not shown here also have this step size and the state of the grid model should be updated accordingly. Now the bus requires an input from the PV entity at every step. However, the PV entity does not provide this data at every bus step. A simple solution to deal with this is to assume that the last provided value is used again, so at $t = 5$ the Bus will consume the data that was provided by the PV entity at $t = 0$. The linear interpolation of a value is theoretically possible by advancing the slower stepping simulation to the next time step (or even the next but one for polynomial interpolation) but this increases control and data flow complexity and is likely to yield wrong values for most cases as the domain mostly involves models with non-linear behavior. Loads show stochastic behavior and charging an EV, for example includes constant power consumption in the first phase only but changes to a non-linear charging in the end phase. An exception may be the fill level of a storage during charging (except for the non-linear end phase in case of a chemical storage). Such data, however is likely to be used as state information for the control strategies only and not part of the physical composition of the Smart Grid. As will be shown below (Chapter 9.6.5) the control strategies will only interact with the simulated entities in intervals the length of which is governed by the simulator with the largest step size. For these reasons, interpolation functionality is not considered in the scope of this thesis **A 5** – *Interpolation is not reasonable in the majority of cases*. In Chapter 9.7 it will be discussed briefly what has to be changed to support interpolation at a later stage.

Metamodel extension

There are three different options to specify, if aggregation functions are to be used and what types of functions are to be used. It could be specified at the EntityType providing a data flow, at the EntityType consuming a data flow or in the scenario description which is introduced on the scenario layer in the next section. In the OpenMI specification it is argued that the “[..] providing component knows best how to convert data in time or space” [GBB$^+$05]. This is certainly true for interpolation functions as the real-world behavior of the providing component governs the choice of the interpolation function. For aggregations, however, this statement is arguable as it rather seems to depends on the receiving component. A number of power values in Watt could be averaged as input for a battery charging process whereas the same values as input for a circuit breaker, for example, would be better aggregated using a maximum function to determine if it trips or not.

So for mosaik, the aggregation behavior is defined at the inputs of an EntityType. Furthermore, aggregation functions can first of all be defined for primitive types only or for array types that reference primitive types to reduce complexity. As pointed out below (see Chapter 12.3) such primitive types will be used for defining physical flows and thus allow the specification of aggregation functions. For non-physical flows (e.g. control commands) object types may be used but aggregation for such flows does not seem to be reasonable/required (why/how to aggregate two control commands?). If required later, aggregation for object types could be added by re-using the ObjectTypeFieldReference mechanism (see Figure 7.11) to specify what function is to be used for which field.

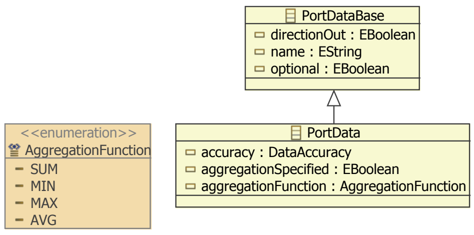


Figure 7.18: Simple metamodel extension to define aggregation

Figure 7.18 shows the corresponding extension of the metamodel that allows to specify the aggregation behavior for each data flow. As enumeration types cannot be null in Xtext, two additional boolean flags are used to indicate if a function is active or not. The fact that aggregation behavior can only be defined for the inputs of an EntityType has to be accounted for by two additional OCL constraints. The same is true for the limitation to continuous, numeric types. The aggregation functions supported are the common aggregate functions (average, minimum, maximum, sum) that are well known from the SQL query language in the database domain [Int11]. If no aggregation function is specified, the default behavior is to use the first value of the interval (which matches the time point of consuming and providing entity). This is also the only possible behavior for non-numeric values.

OCL Constraints

C 32 checks if aggregation functions are only specified for incoming data.

C 33 ensures that aggregation functions are only specified for continuous, numerical data flows.

7.10 Multi-Resolution Composition

In literature, the term *Multi-Resolution Modeling* (MRM) refers to the process of "building a single model, a family of models, or both to describe the same phenomena at different levels of resolution[15]" [DB98]. MRM is a process with the explicit goal of having multiple representations with different resolutions of the same phenomenon, e.g. an electrical power grid. The appropriate representation with respect to the simulation objectives is either chosen beforehand or can be changed dynamically at run-time [HK12]. The latter can be used, for example, to increase simulation performance by only switching to more detailed models in case of interesting events that need to be studied in more detail. While such functionality is certainly interesting for Smart Grid simulations as well, this is not part of this thesis. The term multi-resolution composition described in this section deals with the problem of integrating models of different phenomena and different levels of resolutions [**R 25** – Multi-Resolution Composition] (whereas MRM deals with different models of the same phenomenon). This is caused by the fact that mosaik is used to create composite simulations based on different existing models that were developed independently and thus may have different resolutions. For cases where the inputs and outputs of these models do not have the same level of resolution, the definition of a mapping may still allow for the composition of the models. In Chapter 3.7 an example for such a mapping was given. As these models are not alternatives for the same phenomenon, a dynamic exchange of these models at run-time is not required.

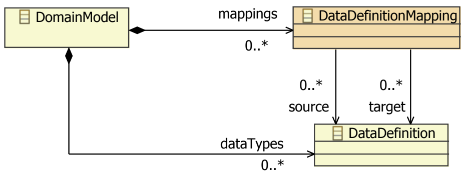


Figure 7.19: DataDefinition mapping to support multi-resolution compositions

Figure 7.19 shows how the semantic metamodel is extended to capture such mappings between data definitions. A *DataDefinitionMapping* specifies what output DataDefinitions can be converted/mapped to other DataDdefinitions. The mapping algorithm itself is only declared in the metamodel, as the algorithm can best be defined in a general purpose programming language rather than a complex metamodel extension. The mappings specified in the metamodel are used on the scenario layer (i.e. prior to run-time) for determining composability only. The actual conversion will then take place at run-time, provided by corresponding callback functions. A port *A* that is providing data can be mapped to a port *B* that is consuming data, if a corresponding mapping function exists that provides all mandatory inputs for port B and can be invoked with all parameters provided by the outputs of port A. This mapping extension can be used to resolve the mismatch between different resolutions of power flows, for example (see Chapter 3.7). However, it has to be pointed out that, although technically possible, the power flow encoding problem [**R 17** – Power Flow Encoding] (see Chapter 3.5.5) should not be solved with this mapping mechanism. It is a purely syntactic problem (same meaning, same resolution) which can be solved much more easily when implementing the SimAPI on the syntactical layer, avoiding the need for a separate mapping function.

[15] See definition of the term resolution in Chapter 2.1.2

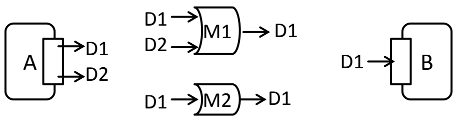


Figure 7.20: Ambiguous DataDefinitionMappings

Although a description of how the mappings are used for achieving composability will be given in Chapter 6, there are two basic options. One possibility is to try to automatically determine a mapping that can be used to connect two ports. An additional OCL constraint could therefore be defined, ensuring that the specified mapping functions have unique signatures (i.e. unique combinations of referenced input and output types). However, Figure 7.20 shows a situation where two mappings (in the middle) have different signatures but could both be used to connect the ports, as both provide the require input for B and the required inputs for both mappings are provided by A. One could argue that mapping M1 should be chosen as it makes use of all provided ports (and is likely to yield better results), but from the current knowledge this is a blatant assertion that may cause undesired results. Also, from the user perspective it is not ideal, as it is hard to figure out what mapping is used. Each DataDefinitionMapping therefore has to have a unique name attribute (**C 34**) which is used on the scenario layer to select the desired mappings.

7.11 Maintenance

The semantic layer presented above defines two different types of artifects: the domain specific reference data model and simulator descriptions. The reference data model represented by the class DomainModel is a shared resource among the simulator descriptions as it represents the common data flow semantics. As Röhl [RÖ8] points out, such references to a central definition are the only option to put components into a common context. But as a consequence, changes to the reference data model may trigger changes in the simulator descriptions or influence the composability and thus the validity of the scenario definitions described in the next section.

7.11.1 Possible Changes

In the following paragraphs possible types of changes and their impact are discussed briefly split into changes to the reference data model and changes to simulator descriptions. An in depth-discussion of this matter is not subject of this thesis, as these problems are not specific to the mosaik approach but rather occur in every approach that is based on a common data model.

Changes to the reference data model

Table 7.3 lists the potential changes to the reference data model and the severity of their impact. The change column refers to different changes, depending on the type of element. *Breaking (syntax)* indicates changes that cause one or more simulator descriptions to become inconsistent (e.g. dangling references to a deleted element) and *Breaking (semantic)* refers to changes that violate semantic OCL constraints of related scenario models. Adding elements is usually not requiring changes to related artifacts, with exception of the domain specific constraints introduced in Section 7.7. Adding such a constraint can break a scenario model semantically at run-time by making two EntityTypes uncomposable that were (possibly erroneously) composable before the change. For the deletion case the effects are inverse. Deleting a domain specific constraint does not break anything as the constraints are not referenced directly, whereas the deletion of any other element causes a syntactic break as referenced elements are no longer available. However, deleting a domain specific constraint may allow semantically invalid compositions in later scenarios. Renaming, where applicable, usually causes breaking changes (again references break) but could be automatically performed as a refactoring [FBB+99] operation. For AbstractEntityTypes changing the super types does not break the scenario syntactically but may cause the composability constraints to fail (e.g. constraint **C 24**). Changing the internal structure of a DataDefinition (e.g. using an integer array instead of a single float value) will lead to syntactical errors at run-time. The SimAPI implementation of the simulators using the DataDefinition has to be changed accordingly. Changes to the signature of a UserDefinedFunction causes syntactic errors as the arguments provided by the scenario model (see Chapter 8.5.5) are no longer valid. Changing the implementation of a UserDefinedFunction can cause the scenario to break at run-time when the mapping semantic is changed. For the DataDefinitionMappings a change to the DataDefinitions referenced as input and output causes a breaking change in the scenario model as EntityTypes become uncomposable that were composable before the change.

Table 7.3: Potential changes to the reference data model and their impact

Element	Add	Delete	Rename	Change
Unit	ok	breaking (syntax)	refactoring	n.a.
AbstractEntityType	ok	breaking (syntax)	refactoring	breaking (semantic)
DataDefinition	ok	breaking (syntax)	refactoring	breaking (SimAPI)
Dom.-spec. constraint	breaking (semantic)	ok	n.a.	n.a.
UserDefinedFunction	ok	breaking (syntax)	refactoring	breaking (syntax/semantic)
DataDefinitionMapping	ok	breaking (syntax)	n.a.	breaking (semantic)

With the exception of structural changes to a DataDefinition, changes to the reference data model do not affect the implementation of the SimAPI, as a simulator description maps the elements of the reference data model to implementation specific names used for interacting with the SimAPI. Scenario definitions from former projects (i.e. that have not been continuously adapted to changes in the reference data model) are mostly unaffected by changes to the reference data model as they contain references to the simulator descriptions. An exception are the references UserDefinedFunctions as well as DataDefinitionMappings.

Table 7.4: Potential changes to the simulator description and their impact

Element	Add	Delete	Rename	Change
Sim Param	breaking (syntax)[16]	breaking (syntax)	refactoring+ SimAPI	breaking (syntax)[17]
Model	ok	breaking (syntax)	refactoring+ SimAPI	(see next rows)
Model MaxNum	ok	n.a.	n.a.	ok
Model Param	breaking (syntax)[18]	breaking (syntax)	refactoring+ SimAPI	breaking (syntax)[19]
EntityType	ok	breaking (syntax)	refactoring+ SimAPI	(see next rows)
Entity Num	n.a.	n.a.	n.a.	breaking (semantic)
Static Data	ok	breaking (syntax)	refactoring+ SimAPI	breaking (syntax)
Port	ok	breaking (semantic)[20]	n.a.	breaking (semantic)
Input	breaking (semantic)	(ok)	refactoring+ SimAPI	breaking (semantic)
Output	ok	breaking (semantic)	refactoring+ SimAPI	breaking (semantic)

Changes to a simulator description

Accordingly, Table 7.4 lists the potential changes to a simulator description and their impact. A few aspects need special explanation. *Refactoring+SimAPI* indicates changes where a refactoring in the artifacts may have to be performed and the SimAPI implementation has to be adapted. This is the case for all renames, as the names of the simulator descriptions are the links to the implementation. Deleting an input or adding an output does not break anything, as the resulting entities can still mutually satisfy their data flows. An exception is the case where all inputs of a port are deleted. In this case the entity may no longer be composable. Changing the number of entities that a model provides may cause a scenario to become invalid as the multiplicity constraint of a ConnectionRule (introduced in Chapter 8.5.3 below) may be violated. Adapting old scenario models to the latest changes of the simulator descriptions will in the most cases be limited to simple changes of the names of the related elements to resolve dangling

[16] Not a breaking change if a default value is provided
[17] Breaking if the data type is changed to a non-compatible type, e.g. changing from int to string
[18] Not a breaking change if a default value is provided
[19] Breaking if the data type is changed to a non-compatible type, e.g. changing from int to string
[20] Assuming that the Ports data flows are deleted, too

references. In cases where composability constraints are broken, an alternative model may be available or a DataDefinitionMapping has to be added.

7.11.2 Simulator Integration Process

Based on the discussion above, the steps for integrating a simulator and its models into the mosaik concept are depicted in Figure 7.21. Depending on the type of simulator (open-source, closed-source) and the interfaces it offers, one of the integration approaches discussed in Chapter 6.4.1 is chosen. Now that the features that the simulator can expose are known, the formal description can be created. This step may require a number of substeps, if the reference data model has to be extended. If the extensions are breaking other simulator descriptions may have to be changed as well which may also trigger changes in their SimAPI implementations (as discussed above). When further models are added, the MoSL description has to be extended and the SimAPI implementation may have to be changed (depending on how generic it is).

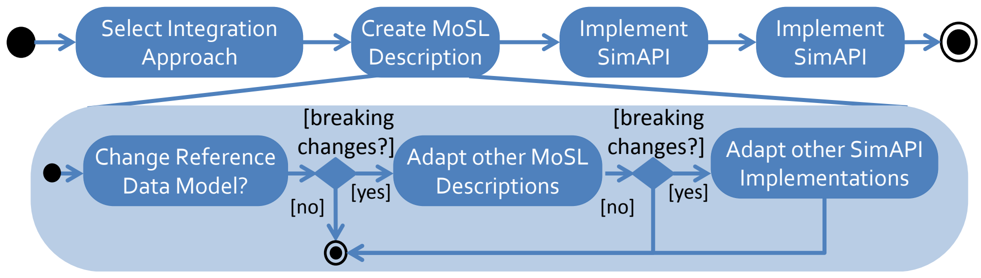

Select Integration Approach
Create MoSL Description
Implement SimAPI
Implement SimAPI
Change Reference Data Model?
[breaking changes?]
[yes]
[no]
Adapt other MoSL Descriptions
[breaking changes?]
[yes]
[no]
Adapt other SimAPI Implementations

Figure 7.21: Process for integrating a simulator and its models into the mosaik concept

7.11.3 Maintenance Process

To avoid the advent of surprising changes to the reference data model or the simulator descriptions and the resulting consequences on the composability of scenarios, the role of a *Semantic Data Manager* is proposed in addition to the different stakeholders already presented in Chapter 3.1.2. His or her task is to coordinate change requests to the data model and to the simulator descriptions. As the Semantic Data Manager has a deeper understanding of the reference data model he or she can thus advise a Model Integrator on what elements of the reference data model to use and thus avoid the introduction of redundant elements with identical meaning. This avoids consolidation of the data model and breaking changes at a later stage. Figure 7.22 depicts a process that is proposed for maintaining the artifacts of the semantic layer and should be carried out by the Semantic Data Manager. This process assumes that all artifacts are managed in a version control system. Initially, revision 1 of Sim A and the reference data model are created. After a while, a project requiring a simulation study is started (Study 1). In the beginning of the project a new simulation model Sim B is being created. At t_1 it has been implemented and it is noticed that it requires a breaking change to the reference data model (e.g. the taxonomy of AbstractEntityTypes has to be refined). Therefore, the reference data model is updated to revision 2 and this triggers a corresponding change of Sim A which now references the updated taxonomy and is updated to revision 2. At t_2, the start of another project (Study 2) requires to extend the functionality of Sim B to revision 2. Due to the use of a versioning system, Study 1 can still use revision 1 of Sim B and is not affected. The new functionality requires a non-breaking change to the reference data model (e.g. add a new DataDefinition) which leads to revision 3 of the data model. Sim A can be used in revision 2 by both studies. At t_3, a bug is detected in Sim B by the team working on Study 1. A fix leads to revision 4. As Study 2 is also affected, a patch is created for revision 2 as well, which leads to revision 4. Although in this example Study 1 could also use revision 4 as it only contains an extension of Sim B, the patch operation is shown for illustration purposes. In reality it may not always be possible to use the latest version as related changes may require revalidation of the model or a new data preparation process is required.

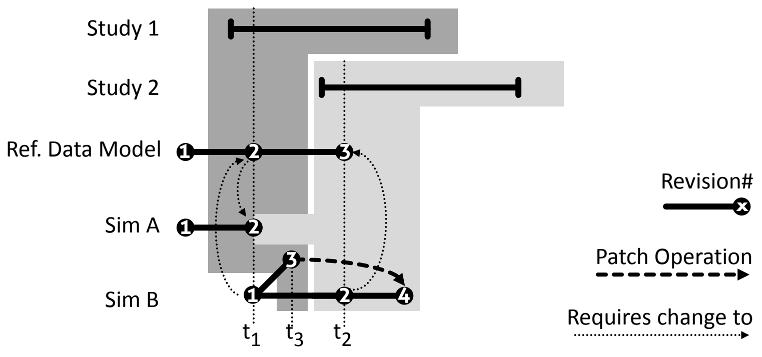


Figure 7.22: Process for versioning the different artifacts of the mosaik concept

The bottom line of the presented process is that the use of a version control system is mandatory, to avoid influencing ongoing studies when extending mosaik for a new study. Also, there should be a list indicating what versions of each simulators are compatible/tested with which version of the reference data model. Finally, the Model Integrators and the Semantic Data Manager should be granted time to merge potential branches (such as revision 3 of Sim B) such that for new studies all bug fixes and extended functionalities are available.

7.12 Discussion

In this section the semantic metamodel of mosaik was presented which is the formal basis for creating well-defined, machine-readable simulator descriptions [**R 24** – Formal Simulator Description]. It allows to create a reference data model and descriptions of the simulators that implement the SimAPI. These descriptions provide information about the available models a simulator can execute and the entity types that these models are comprised of. To define the static and dynamic data of an EntityType, the DataDefinitions provided by the reference model are used, thus providing a shared syntax and semantic of the data.

Figure 7.23 gives an overview of the major classes of the semantic metamodel developed in the last sections. For reasons of readability some classes have been left out, for example the different subclasses of Constant and their corresponding range

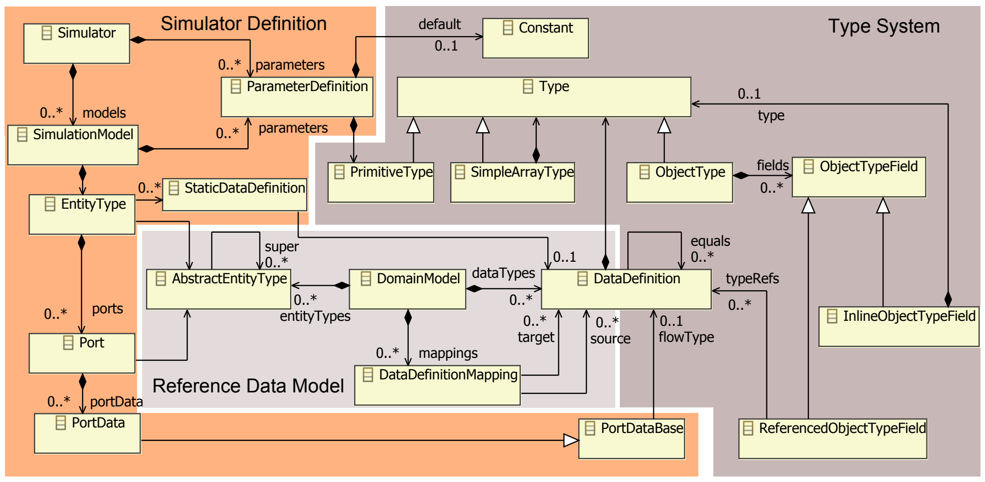


Figure 7.23: Summary of the major classes of the semantic metamodel

classes as well as the DataFlowLimit and its related classes. The shown diagram splits the metamodel into different parts, each dealing with different modeling aspects. The domain specific reference data model is in the center of the diagram, represented by the class DomainModel, providing the AbstractEntityTypes and the DataDefinitions for the simulator definition. The latter is comprised of the Simulator and SimulationModel classes which can define a number of parameters. The SimulationModel then defines a number of EntityTypes which again specify incoming and outgoing data flows grouped into Ports as well as static data describing the non-changing entity properties, such as a location or operational limits. Finally, the type system allows to define data types including unit and range information (not shown in the diagram) that are used for the ParameterDefinition as well as for defining dynamic and static entity data by using the DataDefinition class. The latter has a mixed role as the defined types are part of the data model but the class is also used within the type system for defining the fields of an ObjectType when no anonymous inline type is used (i.e. when a DataDefinition is reused as complex type for a field in an ObjectType).

7.12.1 Requirements

In Section 7.2 the requirements relevant for the semantic layer have been presented. In the following it is discussed how the requirements have been met by the semantic metamodel.

R 7 – Attribute Based Composition, R 11 – Static Data Access As it cannot be assumed that a Scenario Expert knows all available static entity attributes, the simulator description has to make the available static data explicit. This is achieved by referencing the DataDefinitions of the DomainModel via the associative class StaticDataDefinition. By explicitly defining static data this way the automatic comparison of static data for the domain specific constraints (see Section 7.7) is enabled as the structure and meaning of the data is well defined. This also allows static entity attributes to be used for defining a Smart Grid scenario (see Chapter 8).

R 14 – Moving DER Depending on the location attribute of an EV, for example, different connections to the power grid have to be possible. By allowing to specify ranges for dynamic data flows, the Scenario Expert knows what values are available and can define valid conditions (see Chapter 8.5.6 for these connections).

R 15 – Transmission of Complex Data Types The type system of the metamodel allows to define complex data types including array types and polymorphic associations. In Section 7.4.5 it was shown how the corresponding elements of the metamodel can be used to capture the different structures of object oriented class structures as they occur in the CIM standard specification, for example.

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs To support the Scenario Expert in determining the required number of model instances to use in a scenario, information about the number of entities can be provided for each model (see *numOfEntities* attribute of EntityType). In cases where this number depends on parameter values (e.g. the number of bus entities in a power grid model may depend on the topology file provided as parameter), the *numOfEntitiesUnknown* flag indicates this variability. In the next chapter it will be shown how additional, parameter specific entity instance information can be provided to allow the composition engine to validate the composition in such cases as well.

R 17 – Power Flow Encoding In Section 7.10 it was pointed out that the encoding variants for power flows should be unified when implementing the SimAPI and not being dealt with on the semantic layer, as this is a purely syntactic issue. Models that model the power flow in different levels of detail may be composed by using corresponding mapping functions that can be specified in the DomainModel as well.

R 19 – Different Temporal Resolutions The metamodel allows to specify the possible step sizes for each Simulator via the associated IntRange class. As the primitive data types define range information, this range class of the primitive IntegerType has been reused for specifying the simulator step range, rather than creating a new class for this purpose. An explicit constraint **C 11** has to ensure that the values of the specified range are greater than zero. For convenient modeling, the step size can be defined in seconds or minutes, as indicated by the *stepUnit* attribute of the simulator.

R 25 – Multi-Resolution Composition As discussed in Chapter 3.7, models of a real world system can have different abstraction levels. To support the specification of scenarios that include models of different abstraction levels, the reference data model allows to specify mappings between data definitions to support the composition engine in determining what data flows (their data structure definitions) are compatible. The actual conversion for these data flows is assumed to be implemented separately in the language of the mosaik engine as it is not reasonable to extend the metamodel to allow the definition of all possible transformations.

R 29 – Central Parameter Definition The definition of parameters including name, type, unit and an optional range is possible for the simulator definitions and their models. This information allows the scenario expert to specify all required values for instantiation of simulators and the desired models. The range and type information can furthermore be used to automatically validate the provided parameter values. The corresponding constraints **C 46** and **C 47** are introduced in the next chapter as they are part of the scenario metamodel.

7.12.2 Validity aspects

In Section 7.3 a number of validity aspects were identified that are also applicable to the Smart Grid domain. While most of these aspects will be picked up and considered by the scenario layer presented in the next section, some of these aspects already influenced the design of the semantic metamodel.

Validity Aspect 1 – Data types must match This aspect has implicitly been considered when deciding to use a reference data model which not only ensures a common semantic for the contained elements but also defines their data structures. Therefore, if two data flows of a port are connectible as they reference the same element of the data model, the data types also match.

Validity Aspect 2 – Origin/destination types must match To support this validity aspect when specifying a composition on the scenario layer (see next chapter), the EntityType references an AbstractEntityType it represents and each Port can reference an AbstractEntityType as target type (see Figure 7.10).

Validity Aspect 3 – Ranges must match To support range validation, the DataDefinitions used in a Port can be limited to certain values using the DataFlowLimit class (see Figure 7.8).

Validity Aspect 4 – Inputs satisfied As a Port will only be connectible if all required inputs can be satisfied (see **C 24**), the PortData class allows to mark inputs as optional in order to increase composablity.

Validity Aspect 5 – Temporal input availability A special flag indicating if a DataDefinition is expected to be available/consumed continuously or only sporadically, has been added to the DataDefinition class. However, in case of sporadic availability, no further information about the availability can currently be provided, as discussed in Section 7.8.

Validity Aspect 6 – Data flow units match Although matching of units is implicitly ensured by relying on common data definitions, these allow to specify unit information for two reasons. First, this allows the Model Integrator (see Chapter 3.1.2) to identify if a unit conversion has to be implemented when implementing the SimAPI. Second, the unit information (including the mapping to SI units) allows to implement generic adapters for integrating simulation models that are compatible to another standard, such as FMI (see Chapter 12.2.2.3).

Validity Aspect 7 – Temporal data validity This aspect is currently unconsidered for the reasons discussed in Section 7.8.

7.12.3 Limitations

Although it has been shown how the different requirements have been considered for the design of the semantic metamodel, there are currently a number of limitations. First of all, the semantic metamodel does not offer means to capture the temporal validity of data flows. It was assumed [**A 4** – No explicit temporal validity] that physical flows are always valid for the current instant in which they are provided and that the validity of control flows is defined by corresponding data structures of the used DataDefinition and therefore does not require separate means in the metamodel to be captured.

Another limitation, discussed in Section 7.9, is a missing support for interpolation [**A 5** – Interpolation is not reasonable in the majority of cases]. It was argued that it is possible to interpolate, by advancing the slower stepping simulation to the next step. However, as most models do not show linear behavior, such an approach is likely to introduce new errors or at least does not perform better than just assuming that the last values still holds. Also, scheduling of the simulators (see Chapter 9.6) gets more complex. For these reasons, interpolation is not supported as yet. If experience with the mosaik concept shows that interpolation is required, it could be added similar to the specification of the aggregation functions. However, in case of interpolation the function that is to be used should be defined at the data flow definition of the providing EntityType as it depends on the real-world behavior of the object it represents.

Finally, in Section 7.10 a mapping mechanism for making EntityTypes composable, that have data flows with different levels of abstraction. The current mapping design of the semantic metamodel, however, does not account for the validity aspect [**Validity Aspect 3** – Ranges must match], as the mapping does not define valid value ranges for the two flows that are to be mapped. However, this could be added by not only referencing the DataDefinitions that are to be mapped, but also allowing to define valid ranges for each definition in a similar way as it is done in the ports of the EntityTypes (see Section 7.6).

8 Scenario Layer

During the requirements analysis in Chapter 3.2 different Smart Grid scenarios were discussed. Mosaik must be able to simulate these and other scenarios when provided with the corresponding simulation models. As these scenarios were derived from real-world settings they did not contain any notion of simulation. For simulation based scenarios different definitions of the term scenario exist in literature. As already mentioned above, Falkenhainer and Forbus [FF91] provide an abstract definition of the term *scenario* by defining it as “the system or situation being modeled.” In the context of HLA Topçu and Oğuztüzün [TO10] define a scenario as “the description of an [military] exercise providing entities and a series of events related to those entities.” Lofstrand et al. [LKS+04] use a similar definition which additionally points out that besides entities and events, initial state information is part of a scenario. Following the two last definitions, the term scenario refers to a description of a real world situation whereas Falkenhainer and Forbus [FF91] use the term for the actual situation that is to be simulated. They introduce the term *scenario model* to clearly distinguish between the scenario (actual situation) and its description (the scenario model). To avoid any further confusion the term scenario model is also used in the remainder of this thesis when referring to a description of a real world scenario.

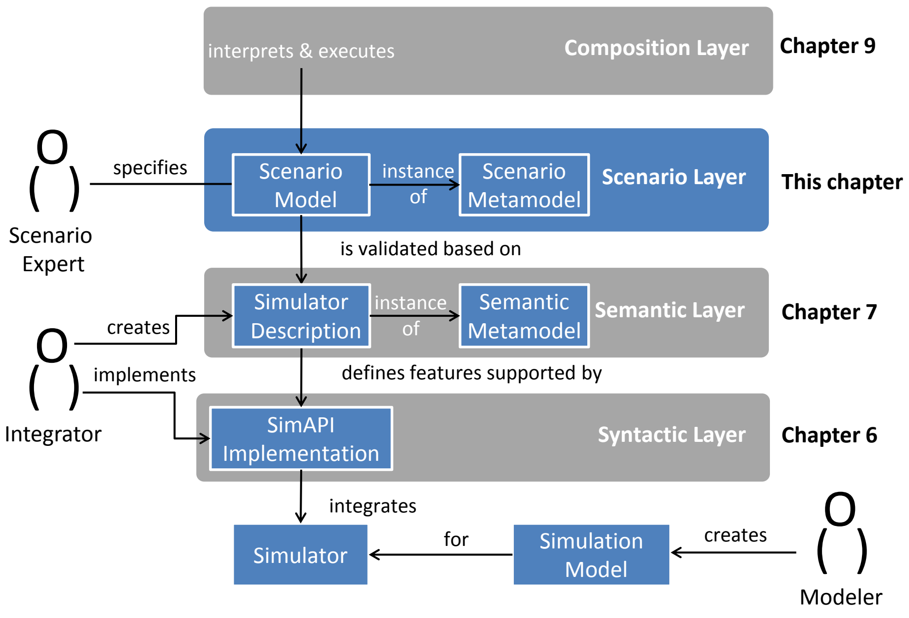


Figure 8.1: Context of the scenario layer

The development of a metamodel that allows to create large-scale scenario models (instances of the metamodel) by composing available simulators is within the scope of this chapter and the major contribution of this thesis. To assess the validity of the composed simulation models, a scenario model will have to reference the elements of the simulator descriptions. These dependencies are shown in Figure 8.1. To avoid confusion with the semantic metamodel presented above, the metamodel presented in this chapter will be called *scenario metamodel*.

8.1 Requirements

This section briefly discusses those requirements that influence the design of the scenario metamodel. After that, works related to simulation scenario definition are analyzed and finally the mosaik scenario metamodel is being presented.

R 2 – Scenario Specification Mechanism The existence of a scenario specification mechanism is the basis for enabling the automatic composition of simulation models to form a simulated SmartGrid scenario. The scenario metamodel developed in this chapter represents this requirement.

R 4 – Scalable Scenario Definition This is one of the most important requirements as Smart Grid scenarios can have very different sizes. They range from a handful of entities in a small low voltage grid up to thousands of entities in a medium voltage grid. To account for this fact, the scenario metamodel must allow to capture large scenarios without requiring the user to specify an equally large scenario model. As will be shown below, the key to meet this requirement is the use of so called *EntitySets* and a set of rules that connect entities from these sets as well as the possibility of reusing compositions as building blocks for other compositions in a hierarchical fashion. This scalability distinguishes the presented scenario metamodel from other scenario description approaches as discussed below.

R 5 – Entity Dependencies As will be discussed below, the scenario metamodel follows a random based composition approach for large-scale scenarios. For example, 200 PV entities can be randomly distributed within a power grid. To avoid invalid connections resulting from this randomness, the scenario metamodel must account for the fact that certain entity types can only be connected to each other dependent on other connections. The mentioned PV entities, for example, may only be connected to those busses of a power grid model to which residential loads are connected (i.e. roof-mounted PV).

R 7 – Attribute Based Composition In Chapter 3.2.3 it was shown that environmental models can be integrated easily into a Smart Grid scenario when entities can be composed based on their attributes (e.g. by comparing geographic positions). The metamodel has to account for such attribute comparisons.

R 14 – Moving DER The scenario model must provide a mechanism to account for the fact that EVs can charge at different nodes of the power grid.

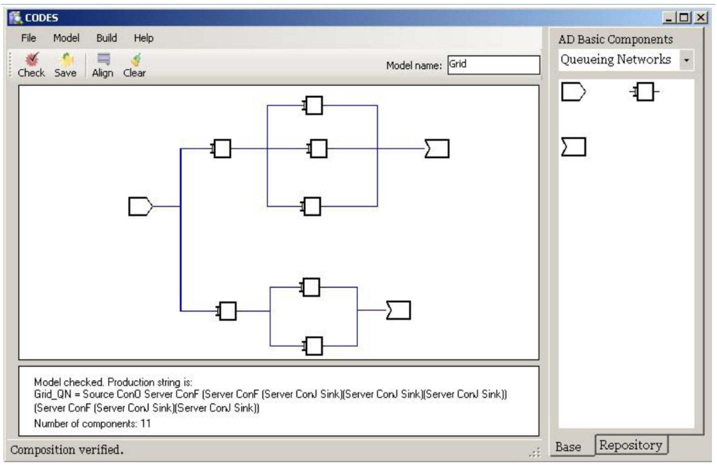
R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs The fact that models may have a different number of entity instances as well as different EntityTypes has already been considered during the conception of both, syntactic and semantic layer. This also poses a special challenge on the scenario layer as the scalability requirement [**R 4** – Scalable Scenario Definition] can only be met, if these potentially large numbers of heterogeneous entities are integrated into the scenario model in a way that avoids the manual connection of entities.

R 19 – Different Temporal Resolutions The scenario model must allow to specify the step sizes of the simulators (within the validity range specified by the simulator description) and in particular it must be possible to specify different step sizes for different simulators.

R 29 – Central Parameter Definition The scenario model must offer the possibility of providing the parameter values for all participating simulators. Experiences from past projects without mosaik have shown that it is time consuming and error prone to maintain different simulator configuration files. The mosaik scenario model should solve this problem by allowing to define all parameters as part of the scenario description rather than in simulator specific configuration file locations and formats.

8.2 Related Work

According to Topçu and Oğuztüzün [TO10] “scenario development is generally a domain specific activity, which is typically carried out by a domain specialist or trainer.” Hence, scenario definitions require domain specific constructs. Although no Smart Grid specific approaches that focus on simulation composition exist (see Chapter 5.3), some kind of scenario definition is also required for monolithic simulation approaches such as GridLAB-D. Furthermore, a number of domain-independent scenario definition approaches as well as approaches from other domains can be found in literature and are discussed in the following.



```
Model checked. Production string is:
Grid_QN = Source ConO Server ConF (Server ConF (Server ConJ Sink)(Server ConJ Sink)(Server ConJ Sink))
(Server ConF (Server ConJ Sink)(Server ConJ Sink))
Number of components: 11
```

Composition verified.

Figure 8.2: CODES Model Composition GUI [ST07]

8.2.1 CODES

Although the term “scenario” is not used in the CODES approach [TS07, ST07], the specification of the component composition can be seen as a scenario, as it specifies the interplay of the components and their structure. The application of CODES to a new domain, for example Smart Grids, requires the extension of the CODES ontology (called COSMO) to capture the basic elements of this domain and the extension of the composition rule grammar [TS07]. Once this is done, for each *base component* (i.e. each model implementation in the model repository) a meta-component has to be created via the XML based COML language that uses the COSMO elements. A graphical icon can be specified for each of these base components and the user can use a GUI to specify the number of components to instantiate as well as their interconnections as shown in Figure 8.2. The specified composition, called *model component*, can be reused as building block in other component models.

With respect to the definition of Smart Grid simulation scenarios two observations can be made. First, requiring the user to manually specify every component connection is only appropriate for small scenarios. This approach is not scalable enough for larger scenarios that can be found in the Smart Grid domain. Even a small LV grid usually has more than 50 interconnections just between loads and busses. Second, the possibility of hierarchically reusing compositions is a feature that can very well be applied to the definition of Smart Grid scenarios, matching the natural, hierarchical structure of power systems. This way, for example, it is possible to specify a number of LV grid compositions and reuse them as building blocks for MV grids, and so on. Also, residential loads and PV systems could be bundled as building blocks. A detailed discussion for the use of hierarchical scenario models is done in Section 8.5.2.1 below. For the time being it is sufficient to capture that hierarchical scenario models should be supported by the mosaik scenario layer (**R 30** – *Hierarchical Scenario Modeling*).

8.2.2 CoMo

While CoMo [RÖ8] allows to construct components hierarchically by using other components (meeting [**R 30** – Hierarchical Scenario Modeling]), CoMo relies on XMl files to specify the individual connection between the ports of different components. Hence, the scalability requirement cannot be met [**R 4** – Scalable Scenario Definition].

8.2.3 FAMAS

The FAMAS Simulation Backbone Architecture [Boe05] specifies a so called *scenario object* and a corresponding graphical user interface to create the scenario object. However, it only allows some simple settings: First, the participating simulators and their possible parameters can be specified. Next, an initialization script that determines the order in which the simulators connect to the backbone can be specified. Finally, events that are executed at a predefined simulation time can be defined in a scenario section. Each event can change the value of a parameter of a simulator, e.g. change the weather parameter from 'sunny' to 'rainy'. The mosaik concept as it is presented in this thesis does not have the requirement to offer such event functionality as it is assumed that any interaction is initiated by the control strategies for which a separate interface is being created. The requirements [**R 4** – Scalable Scenario Definition] and [**R 30** – Hierarchical Scenario Modeling] are not met by the simple FAMAS scenario specification.

8.2.4 GridLAB-D

In Chapter 4.1.2 it was shown that the scenario definitions of GridLAB-D (GLM files) provide some features that enable scalability. For example the *<class>:n..m* syntax can be used to create a large number of instances of a GridLab-D model component. However, wildcards to connect other models to these instances can only be used once. This is a severe limitation, as a large number of busses in a power grid model often has to be connected to more than one other type of model, for example to PVs, EVs, residential loads and so on. As a consequence, GLM files for a realistic scenario can still be over 100.000 lines long, and require the users to make use of additional scripts[1] to generate these files.[2] Furthermore, the wildcard syntax does not allow to provide additional constraints such as dependencies on other connections [**R 5** – Entity Dependencies] or the connection of components based on their attributes [**R 7** – Attribute Based Composition]. However, the ability to make use of distribution functions in stead of fixed values for the creation of such a range of instances is an interesting feature that is picked up by the mosaik scenario metamodel (**R 31** – *Randomized parameter values*). Also, GridLAB-D allows to define nested objects (e.g. appliances as part of a house) and thus meets [**R 30** – Hierarchical Scenario Modeling].

8.2.5 IAM

For the Integrated Asset Management framework that has already been discussed above, Soma et al. [SBO+06] present a graphical front-end (Figure 8.3) that allows to manually connect the available model components for reservoir forecasting scenarios. The editor is automatically created by the GME environment, based on the underlying, domain-specific metamodel presented in Chapter 7.3.3). As with all graphical approaches discussed in this section, the manual creation of objects and their connections does not satisfy the scalability required for the mosaik scenario definition [**R 4** – Scalable Scenario Definition].

8.2.6 ILIas

The ILIas framework [KGC+12] offers a graphical user interface for creating and modifying the network topologies and their connected components that are to be simulated. As a matter of fact, the user is required to specify each individual connection and [**R 4** – Scalable Scenario Definition] cannot be met.

8.2.7 OpenMI

As introduced above, OpenMI provides a standardized interface for linking simulation components, i.e. any entity that can provide and/or accept data, e.g. a simulation model or a database. All examples for composed simulations with OpenMI that are given in its

[1] For example the `http://gridlab-d.svn.sourceforge.net/viewvc/gridlab-d/Taxonomy_Feeders/PopulationScript/Feeder_Generator.m` (accessed 17 Aug. 2011)

[2] Broeer, T.: "RE: A question regarding GridLAB-D". Message to the author. 17 Aug. 2011. E-Mail.

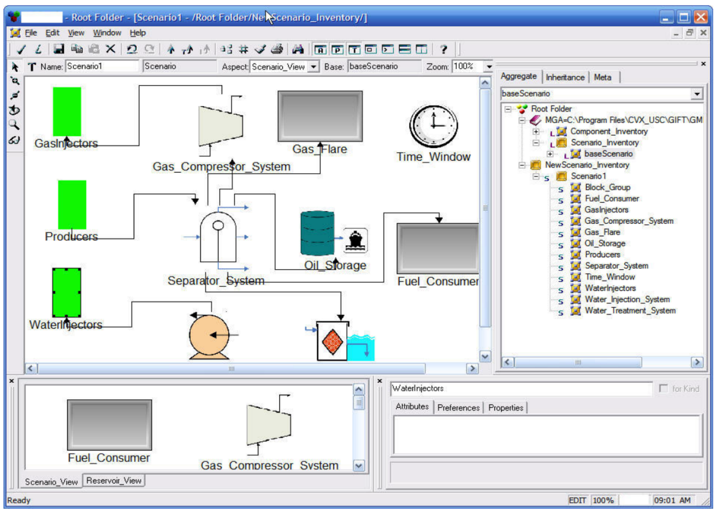


Figure 8.3: GME based editor for modeling reservoir forecasting scenarios [SBP07]

specification include less than a handful of linkable components. An example for such a composition is the linkage of a ground water model and a river model [Moo10, p.24]. It can be assumed that the number of components to be composed in the environmental domain is smaller but their interconnection is more complex. They are not necessarily related in a topological fashion as in the scenarios analyzed for mosaik but may also be related to each other using complex geometric shapes, for example the ground area/shape of a river (see semantic analysis of OpenMI). OpenMI offers an editor[3] (see Figure 8.4) that requires the manual connection of the linkable components, similar to the CODES approach presented above. Therefore, requirement [**R 4** – Scalable Scenario Definition] cannot be met. Also, a hierarchical composition is not possible.

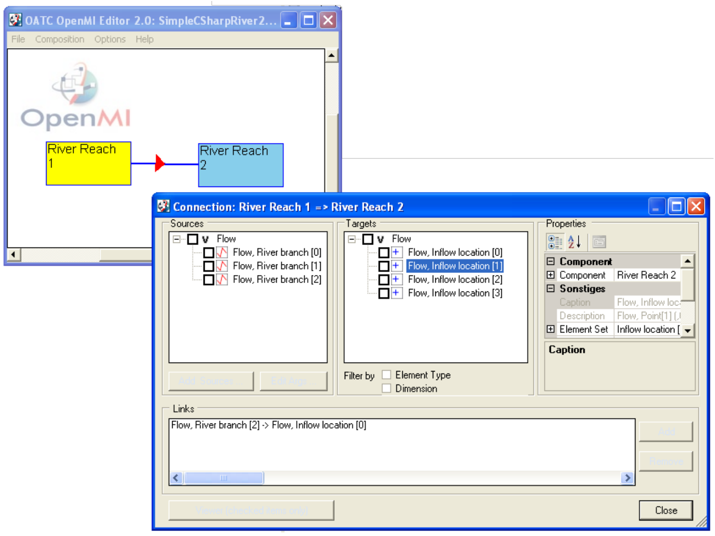

OATC OpenMI Editor 2.0: SimpleCSharpRiver2...
File Composition Options Help
OpenMI
River Reach 1
River Reach 2
Connection: River Reach 1 => River Reach 2
Sources
Flow
Flow, River branch [0]
Flow, River branch [1]
Flow, River branch [2]
Targets
Flow, Inflow location [0]
Flow, Inflow location [1]
Flow, Inflow location [2]
Flow, Inflow location [3]
Properties
Component
Component River Reach 2
Sonstiges
Element Set Inflow location
Caption
Filter by Element Type Dimension
Links
Flow, River branch [2] -> Flow, Inflow location [0]
Close

Figure 8.4: OpenMI Configuration Editor

[3] `http://www.openmi.org` (accessed 14 May 2012)

8.2.8 Scenario Matching

There is a body of work [FF91, Mor08, BB11] dealing with concepts to describe scenarios independently of any model implementations and trying to find those implementations that best match the scenario requirements in a subsequent step. The idea of just coming up with a set of questions and composing a simulation out of a relevant set of objects that answers these questions [FF91] is appealing. However, for mosaik it is assumed that the number of models is limited and that finding the appropriate model implementations for composing a scenario is not a task that has to be supported by the platform. Therefore, the mosaik scenario metamodel uses the models of actual simulators as building blocks and not conceptual objects. Otherwise, the semantic metamodel would have to be constructed differently, e.g. by assigning data flows to the AbstractEntityTypes (similar to the CoMo structure, see Figure 7.9) and using these as building blocks. However, to keep the system simple [**R 1** – Keep It Simple] this option was not considered.

Related Work Summary

Table 8.1 gives an overview about the presented approaches dealing with scenario definitions in the general field of simulation. With the exception of GridLAB-D, none of the discussed approaches provide the required scalability for describing large-scale Smart Grid scenarios that possibly involve hundreds or thousands (see Chapter 12.4.6.2) of entities that have to be connected. A manual specification of the individual connections is error prone and time consuming. A good aspect of the GridLAB-D modeling mechanisms is the ability to create a large number of objects with a single statement and to be able to use random distribution functions for parameterizing these objects differently. However, as discussed above, the scalable (i.e. effortless) creation of interconnections between the created objects is still very limited. A more powerful and scalable Smart Grid scenario specification mechanism is therefore presented in the remainder of this section and is a major contribution of this thesis.

Table 8.1: Related work dealing with scenario/composition definition

Approach	Scalability (R 4)	Hierarchical scenarios (R 30)
FAMAS	No	No
CODES	No	Yes
CoMo	No	Yes
GridLAB-D	Some	Yes
IAM	No	No
ILIas	No	No
OpenMI	No	No
Scenario Matching	No	No

8.3 Scenario Modeling Concept

As mentioned before, the Smart Grid scenarios needing to be simulated can become quite large. A single LV segment fed by a single transformer may very well have about 70 nodes to which consumers (e.g. private households) and potential DER can be connected. Assuming that each household has got a PV system and an electric vehicle, such a segment has about 210 entities (plus the power grid entities) and the same number of connections has to be specified (from each simulated entity to a node of the power grid). If a branch of an MV grid which about ten substations is to be simulated and each substation is feeding an LV segment modeled in this granularity, the overall number of entities and connections to specify is 2100. This number increases further, when more DER are added or a household is simulated in more detail by means of single appliance models. As a consequence the requirement of scalability [**R 4** – Scalable Scenario Definition] influences the scenario definition concept the most. An approach has to be developed that allows to specify such potentially large scenarios without having to specify each entity instance and its connections to other entities in a manual fashion. Of course, when simulating even higher voltage levels in this detail, the number of entities and connections increases even further. However, due to the required computational resources for such detailed large-scale simulations, it can be assumed that the level of detail will be reduced or other methods, such as model abstraction approaches (e.g. [BEDM98]), will be pursued for such scenarios. Figure 8.5 shows three potential ways for creating a Smart Grid scenario model which are discussed in the following.

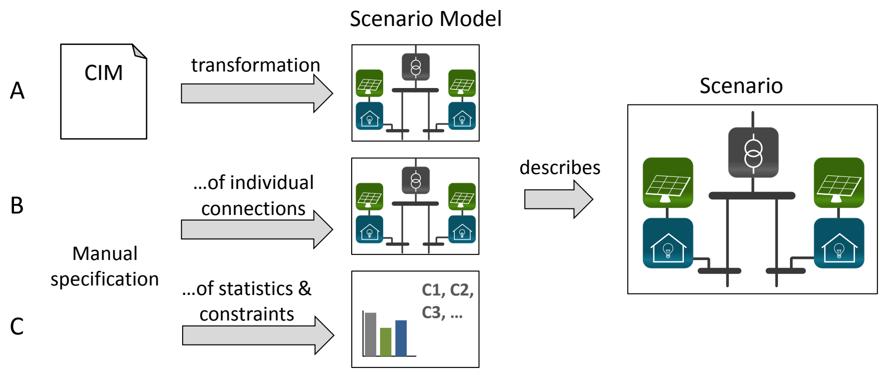


Figure 8.5: Three ways to create a scenario model

Transformation (A) This approach assumes that a standard compliant description (e.g. using the CIM) of the power grid and all connected consumers and DER can be provided, for example exported from the database of a grid operator. Based on this file, the scenario model can be automatically generated. However, in reality such standardized files currently only exist for the power grid topology and assets, if at all. For describing a complete scenario additional information about the loads and DERs connected to the grid would be required. Furthermore, simulation studies in the Smart Grid domain often deal with future scenarios especially in the field of academia and research that mosaik targets (see Chapter 3.1.2). So even if a transformation was possible, the resulting scenario model would have to be extended or modified manually to meet the future setup. As more DER are likely to be deployed in the future, their number may still be too large for a convenient manual scenario model modification.

Manual Specification of Elements and Connections (B) As comprehensive files which can be transformed into a scenario model do not exist, a manual creation of a scenario model is the most common option, which is also employed by the related works discussed above. This includes textual (e.g. the GLM files of GridLAB-D) as well as graphical approaches (e.g. the editors offered by OpenMI or CODES). The manual efforts include the specification of model parameters, the specification of model instances as building blocks for the scenario and the specification of individual connections between these instances. As discussed above, the manual specification of individual connections is not the ideal solution for Smart Grid scenarios, as these are characterized by a large number of objects. Therefore, the mosaik scenario concept introduces a third option.

Manual Specification of Statistics and Constraints (C) For this option, the scenario model does not directly reflect the structure of the scenario that is to be simulated. Instead, statistical data about the number and distribution of entities (e.g. every third household has got an EV) and additional constraints are used to describe a scenario. At run-time, this information is processed by a composition engine (see composition layer described in Chapter 9) which instantiates the required models and creates connections in a random fashion, whilst obeying the constraints. The result is a scenario that is representative within the boundaries of the given statistics and constraints.

But usually, not all aspects of a scenario model are subject to randomness. The use of random based composition can only be justified when no specific data is available (i.e. for future scenarios). Therefore, a combination of statistically distributed and manually specified connections has to be possible. However, even in cases where most connections are known, the random based approach can be applied and ease scenario model creation if applied properly. For example, a simulation study may be based on a real-world, representative power grid. It is known that at node A and B industrial loads are connected and all remaining nodes have residential loads. The remaining residential loads can be distributed randomly without having to specify each connection manually with only a single additional rule: A node may not have more than one load (residential or industrial). When all residential loads are modeled by the same type of model such an approach is viable.

In addition to this random and constraint based composition approach, scalability will be achieved by supporting hierarchical reuse of compositions as building blocks in other compositions. Section 8.5.2.1 discusses potential use cases for hierarchical modeling and presents a corresponding concept.

8.4 Example Scenario

Before the scenario metamodel that allows to capture such statistic and constraint-based scenario aspects is presented, an example scenario is being introduced that will be used for illustration purposes throughout the remainder of this thesis. It is depicted in Figure 8.6.

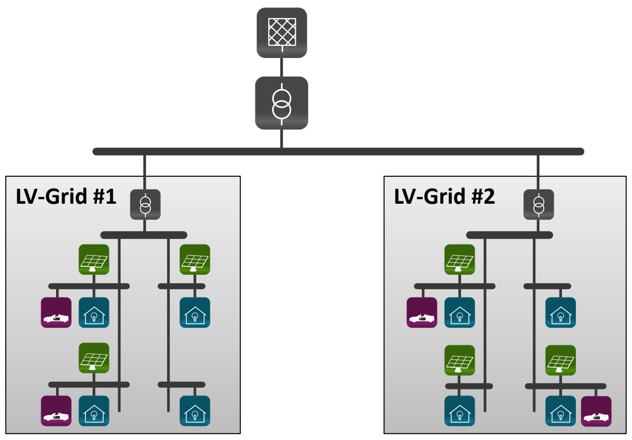


Figure 8.6: Example scenario used throughout this chapter

The example has been chosen in such a way that most parts of the scenario metamodel can be illustrated. The scenario is comprised of an MV grid to which two LV grids are connected. It is assumed that the LV grids are simulated separately from the MV grid (different model instances). The SwingBus of the LV grid is the connection point to the MV grid. Each of the LV grids is described by the following rules:

- Each bus in the LV grid has a residential load connected to it.
- 75% of these busses (3 out of 4) have a PV system connected to it (roof mounted PV).
- In this example it is assumed that homes with PV systems are more likely to have an EV than others. Therefore, EVs are only connected to busses that have PV systems connected to it.
- Furthermore it is assumed that again 75% of the busses which can have EVs (those with PV systems) actually have an EV.
- As the LV grids are assumed to be residential areas, the EVs shall only be connected there when their location is “home”.

As will be shown later, it is sufficient to model a single LV grid and reuse it twice for building the example scenario.

8.5 The mosaik Scenario Metamodel

In the following subsections the details of the mosaik scenario specification concept are presented and the corresponding parts of the scenario metamodel are shown. Again, UML diagrams are used for illustration accompanied by additional OCL constraints to ensure the validity of metamodel instances.

8.5.1 Parameterization

As motivated in Chapter 7.4 and shown in Figure 7.3, a simulator as well as its models may define one or more parameters. For being able to use a simulator and initialize its models, values for these parameters have to be provided (parameterization). So at first, the scenario metamodel will have to provide a mechanism to define these parameters. For this purpose, the notion of model and simulator parameter sets is being introduced. These are used to define a set of coherent parameter values to parametrize a *Simulator* or a *SimulationModel*.

Definition 8 (SimulatorParameterSet) *A* SimulatorParameterSet *is a set of non-redundant, mandatory simulator parameter values and a number of subordinate ModelParameterSets.*

Definition 9 (ModelParameterSet) *A* ModelParameterSet *is a set of non-redundant, mandatory parameter values for a SimulationModels of the Simulator that is referenced by the parent SimulatorParameterSet.*

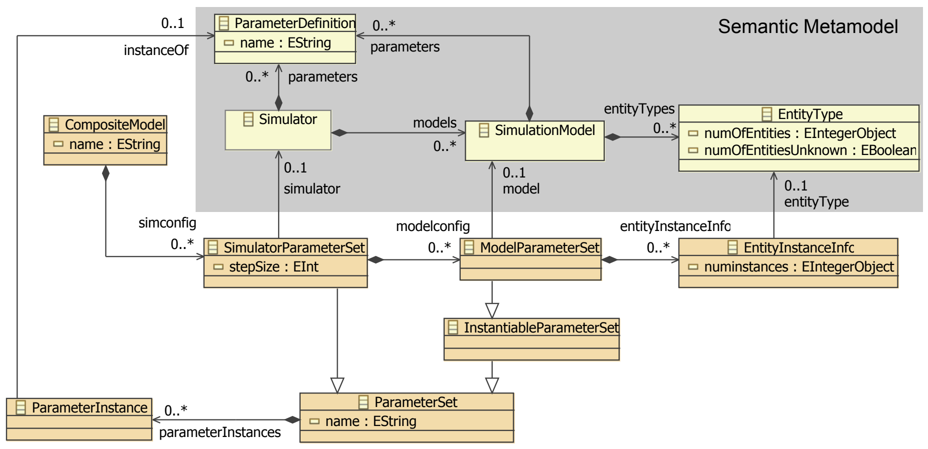


Figure 8.7: Definition of parameter values for simulators and models

Figure 8.7 shows how these sets are integrated into the metamodel. Values for a ParameterDefinition are provided by the *ParameterInstance* class (see Figure 8.8, below). As it cannot be precluded that the choice of parameter values for a simulator influences the behavior of its models, the model parameter values are to be defined subordinate to the simulator parameter values. To express this in the metamodel, specific subclasses for model and simulator parameter sets have been defined and via the *modelconfig* association a number of ModelParameterSets can be specified subordinate to a SimulatorParameterSet. In other words, a ModelParameterSet is always defined in the context of a SimulatorParameterSet. A SimulatorParameterSet is part of a *CompositeModel* which represents a composition and can be used hierarchically as building block within other composite models to meet [**R 30** – Hierarchical Scenario Modeling]. Details of hierarchical modeling will be introduced in Section 8.5.2.1. The *InstantiableParameterSet* will also be motivated in Section 8.5.2.1. By referencing the elements from the semantic metamodel instead of working with name strings, it is made sure that only simulators, models and parameters are used that are actually available. Additional OCL constraints ensure that homomorphic ParameterInstance-ParameterDefinition relations are created and consistency is ensured.

As discussed in Chapter 7.4, there may be models where the number of entities is not model specific but rather depend on a parameter value. A power grid model, for example, may have a varying number of bus, branch and transformer entities when a topology file is used to parametrize the model. In such cases, the ModelParameterSet allows to define a number of *EntityInstanceInfo* objects. Each of these defines the number of entity instances for an EntityType that result from the instantiation of the referenced SimulationModel for the given ModelParameterSet. This way precise information about the number of entities per model instance can be defined, allowing the Scenario Expert to choose the required number of models and, what is even more important, enable the automatic validation of the composition with respect to constraints that are part of the connection rules introduced below.

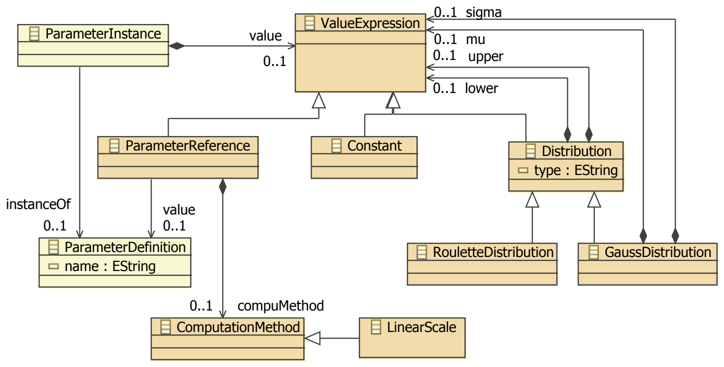


Figure 8.8: Parameter instances for configuring simulators and models

Figure 8.8 shows how a ParameterInstance provides a value for a defined parameter via a *ValueExpression*. Such an expression can either be a *Constant* (see Figure 7.3), sampled from a random *Distribution* function (in case of numerical parameter types) or obtain its value from *PrameterReference*. The usage of the latter will be motivated in Section 8.5.2.1. Furthermore, variables can have a *ComputationMethod*[4] allowing to scale the variable value, prior to using it. So far, only linear scaling is available but other methods could be added accordingly.

OCL Constraints

The following OCL constraints accompany the classes of the scenario metamodel introduced in this section:

C 35 SimulatorParameterSet.name must be unique within a CompositeModel.

C 36 ModelParameterSet.name must be unique within the context of its SimulatorParameterSet.

[4] This name is taken from the ASAM FIBEX standard [Ass11] which defines such computation methods for scaling transmitted values in a similar fashion in the domain of automotive bus systems.

C 37 A SimulatorParameterSet must only define ParameterInstances for ParameterDefinitions that are part of the Simulator that is referenced (homomorphism).

C 38 A ModelParameterSet must only define ParameterInstances for ParameterDefinitions that are part of the SimulationModel that is referenced (homomorphism).

C 39 The ParameterInstances of a ParameterSet must not reference a ParameterDefinition more than once (no redundant parameters).

C 41 A SimulatorParameterSet must define ParameterInstances for all ParameterDefinitions of its Simulator that have no default values (completeness).

C 42 A ModelParameterSet must define ParameterInstances for all ParameterDefinitions of its SimulationModel that have no default values (completeness).

C 46 The values provided by the ValueExpression of a ParameterInstance must match the type of the corresponding ParameterDefinition.

C 47 The values provided by the ValueExpression of a ParameterInstance must be within the range of allowed values specified for the Type of the corresponding ParameterDefinition.

C 43 For each EntityInstanceInfo the referenced EntityType must be part of the referenced SimulationModel.

C 44 For each ModelParameterSet an EntityInstanceInfo must be specified at most once for each EntityType.

C 45 An EntityInstanceInfo may only be specified for EntityTypes with an unknown number of instances.

C 48 Distribution.lower must be greater than or equal to the upper value.

C 49 The attributes *lower* and *upper* of a Distribution must not be a Distribution to avoid confusing setups.

C 50 Also, the GaussDistribution attributes (sigma and mu) must not be a Distribution to avoid confusing setups.

8.5.2 Instantiation

Entities will be the building blocks for compositions, as these are the objects defining inputs and outputs that can be connected. At run-time, entities can be obtained by instantiating a model in a simulator. A ModelParameterSet contains everything that is needed to perform such an instantiation. It references a specific model, provides all required parameters for it and from its context (a SimulatorParameterSet) the simulator containing this model can be identified, started and parametrized. This way, a ModelParameterSet represents a specific real world system (e.g. a LV grid or a specific type of EV) by providing potentially generic models with specific values. To meet [**R 4** – Scalable Scenario Definition], a mechanism to manage large sets of entities resulting from the instantiation of models has to be found. The mosaik scenario metamodel therefore introduces the notion of an *EntitySet*.

Definition 10 (EntitySet) *An* EntitySet *represents all entities that will at run-time result from the instantiation of one or more SimulationModels or CompositeModels for a given ParameterSet.*

Two things are noteworthy about this definition. First, the entities that will result from instantiation are represented. Hence, an EntitySet does not actually contain entities that will exist in the simulator processes. However, the number and type of entities in an entity set can be inferred from the simulator description provided on the semantic layer. The reason for this is that at scenario design-time no simulator processes are being instantiated. As the scenario is not necessarily modeled on the server that contains the simulators, either all simulators would have to be available on the computers of all scenario developers (which is a very large administrative overhead and thus not feasible) or a connection to the simulation server is required for scenario validation and an unnecessary load is placed on the simulation server. Second, an EntitySet only represents the entities from a single ModelPrameterSet. This way it is made sure that the large number of entities in a set can be handled uniformly as all entities originate from model instances with the same parameter values and thus, they are identical with respect to their possible use in a scenario.

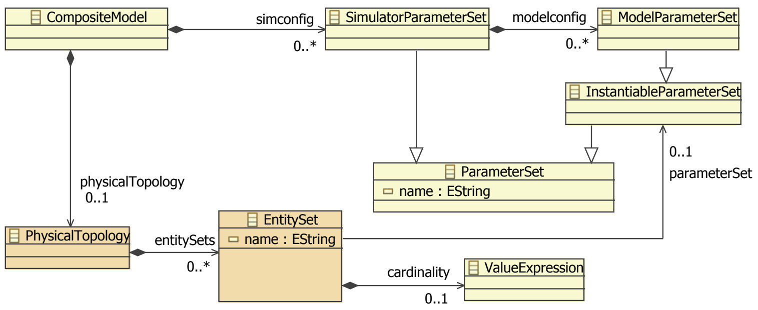


Figure 8.9: EntitySets as logical container for entities resulting from simulation model instantiation

Figure 8.9 shows how the EntitySet concept is integrated into the metamodel. Basically it attaches a *cardinality* (the number of instances to create) to a ModelParameterSet. This is done by referencing the abstract class *InstantiableParameterSet* that will be motivated in the next section. The *cardinality* can be any ValueExpression and especially a ParameterReference. This is an important aspect that will also be picked up in the next section, when the hierarchical modeling approach is presented. An EntitySet cannot reference a SimulatorParameterSet. A simulator is only seen as a container for the models which are the building blocks for each scenario model. The fact that a simulator may be capable of executing many instances of a model and maybe even models with different parameters is hidden from the Scenario Expert and dealt with on the composition layer by creating as many separate simulator processes as necessary. The required simulator processes to provide the number of model instances are determined at composition time (see Chapter 9.5.1). An EntitySet is not directly contained in a CompositeModel but via a *PhysicalTopology* class. The latter accounts for the fact that the mosaik approach may be extended later to include additional elements for definitions of communication topologies, for example.

Example

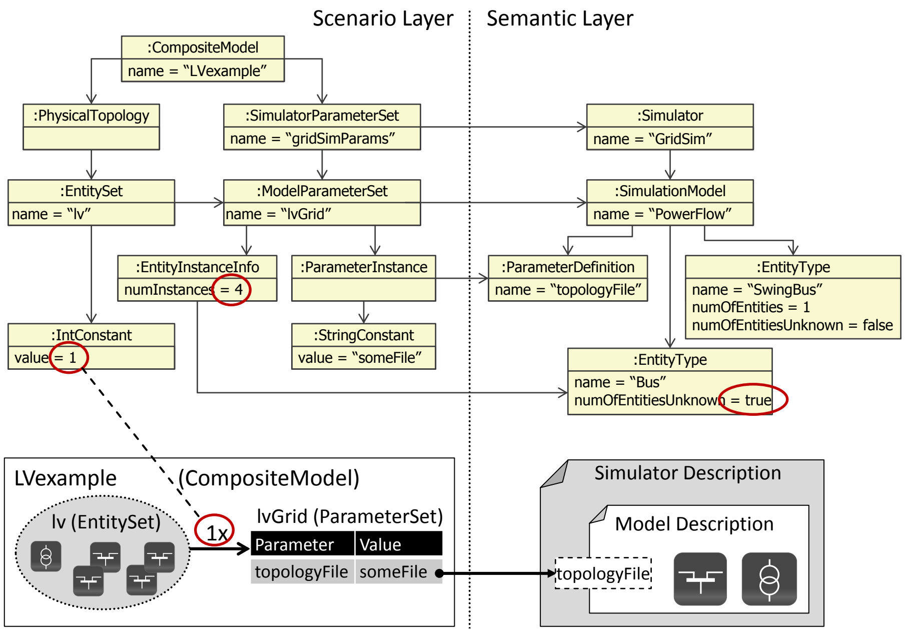


Figure 8.10: Example of how EntitySets are defined

The upper part of Figure 8.10 contains a UML object diagram illustrating the use of an EntitySet and other classes that have been introduced above. The right part of the object diagram depicts how a possible description of the power flow simulation GridSim (see Chapter 3.5.5) can be defined. The simulation model can be parametrized with a CIM compliant grid topology. As the simulation can currently only deal with low voltage grids that contain a single transformer (SwingBus), the number of *SwingBus* entities is fixed per model instance. The number of regular busses (to which other resources can be connected) depends on the provided topology file, the number of entities is defined as unknown using the *numOfEntitiesUnknown* attribute. On the left part of the figure, a scenario metamodel instance is shown which defines a composition with parameter sets for the GridSim and an additional EntityInstanceInfo object that provides information about the number of bus entities for the given topology file. With this information it is already possible to infer the number and type of entities that the defined EntitySet *lv1* will contain at design-time, as depicted at the bottom left part of the figure.

OCL Constraints

The following OCL constraints are required to ensure that an EntitySet definition is valid:

C 51 The name of an EntitySet must be unique within a CompositeModel.

C 52 The ValueExpression used to define the cardinality of an EntitySet must resolve to a numeric integer value equal or greater than zero. If a ParameterReference is used, this check makes sure that the range of this parameter is restricted to positive values only. If a float parameter is referenced, the semantic assumes that the decimal places are stripped (i.e. 1.6 becomes 1) like a cast operation from float to integer, for example in Java or Python, would do it.

8.5.2.1 Hierarchical Composition

In Section 8.2 it was identified that hierarchical reuse of compositions is a requirement that can also support scalability as it matches the hierarchical structure of the power grid [**R 30** – Hierarchical Scenario Modeling]. To satisfy this requirement, it must be possible to create instances of other CompositeModels within a CompositeModel. To maximize the reusability of a CompositeModel as building block in different scenarios, the parameters for the simulation models used in a CompositeModel should be adjustable from outside the composition as well.

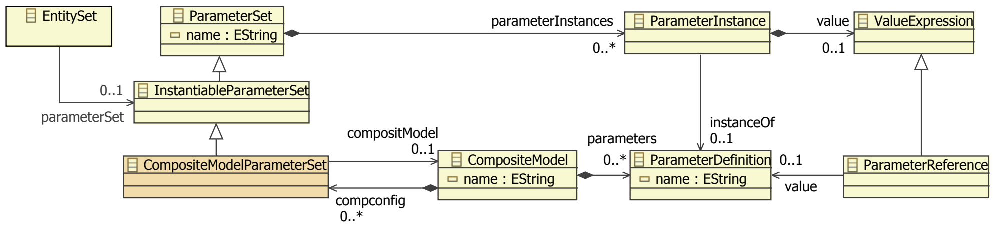


Figure 8.11: Parameterizable CompositeModels as hierarchical building blocks

Figure 8.11 shows the required extensions to the scenario metamodel. At first, the CompositeModel class has been extended to allow the definition of parameters. Second, to achieve a uniform handling of SimulationModels and CompositeModels, the EntitySet concept is being used for the instantiation of CompositeModels as well. As an EntitySet references the InstantiableParameterSet class, a *CompositeParameterSet* subclass can be added easily. Similar to the ModelParameterSet, a CompositModelParameterSet can now be referenced from an EntitySet to specify instances of CompositeModels. The *compconfig* association allows a CompositeModel to define parameter values for the instantiation of another CompositeModel. By using a ParameterReference instead of other constant value expressions, it is possible to use parameters defined by a CompositeModel in its ParameterSets.

Example

Figure 8.12 shows this mechanism using the example introduced above. The LV grid composition that was already shown in the last object diagram above is now extended by an EntitySet with three PV systems and used as building block in the medium voltage level composition. Here, the number of PV instances and their peak power are defined by parameters of the CompositeModel. How the PVs are connected to the bus entities of LV grid and how the swing busses of the LV EntitySet are connected to the bus entities of the MV grid (not shown in the figure) is presented in the next section. The set semantics of an EntitySet referencing a CompositeModel are illustrated in the bottom left part of the figure. It contains subsets which are unions of those EntitySets contained in the CompositeModel instances that have the same name. How these unions can be used for scenario specification will be shown in the following section.

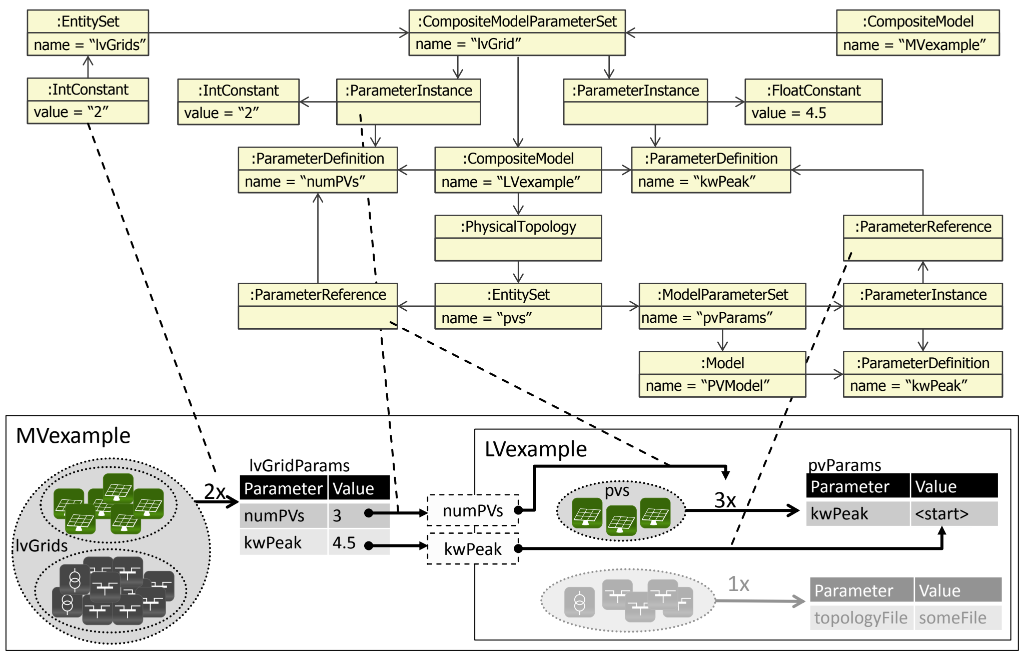


Figure 8.12: Example for hierarchical reuse of composite models

OCL Constraints

To ensure that a CompositeModelParameterSet is valid, the following OCL constraints have to be met:

C 54 CompositeModelParameterSet.name must be unique within a CompositeModel.

C 55 A CompositeModelParameterSet must only define ParameterInstances for ParameterDefinitions that are part of the CompsiteModel being referenced.

C 56 A CompoisteModelParameterSet must define ParameterInstances for all ParameterDefinitions of its CompoisteModel that have no default values.

In addition to these constraints, the constraints **C 39** (no redundant parameters), **C 46** (types match) and **C 47** (range matches) also apply to the CompositeModelParameterSet as they are defined for its base class.

8.5.3 Connection Rules

So far, the scenario model allows to specify ParameterSets that provide the required values for the simulators and their models as well as for CompositeModels. Furthermore, the concept of EntitySets allows to specify how many instances of simulation models and composite models are to be created at run-time. But the major part of the scenario model is still missing. What is needed is a mechanism for composing the entities contained in the EntitySets in such a way that the Smart Grid scenario that is to be simulated can be defined easily. Therefore the concept of *ConnectionRules* is introduced. Besides the EntitySets, ConnectionRules are the heart of the mosaik scenario concept.

Definition 11 (ConnectionRule) *A* ConnectionRule *defines how the instances of two EntityTypes from two different EntitySets are to be connected. Therefore the ConnectionRule allows to define a multiplicity constraint as well as additional, topology and entity related constraints.*

In addition to the possibility of hierarchical compositions, the required scalability of the scenario mechanism is achieved by the ConnectionRules which can operate on an arbitrarily large number of entities. They allow the user to specify rules that are applied by the simulation composition engine for establishing n:1 relations between the entities. The analysis of the Smart Grid scenarios discussed in Chapter 3.2 has shown that it is sufficient to use n:1 relations for describing the scenarios (**A 6** – *It is sufficient to specify n:1 relations to capture Smart Grid Scenarios*). Figure 8.13 shows how the ConnectionRule concept is integrated into the scenario metamodel.

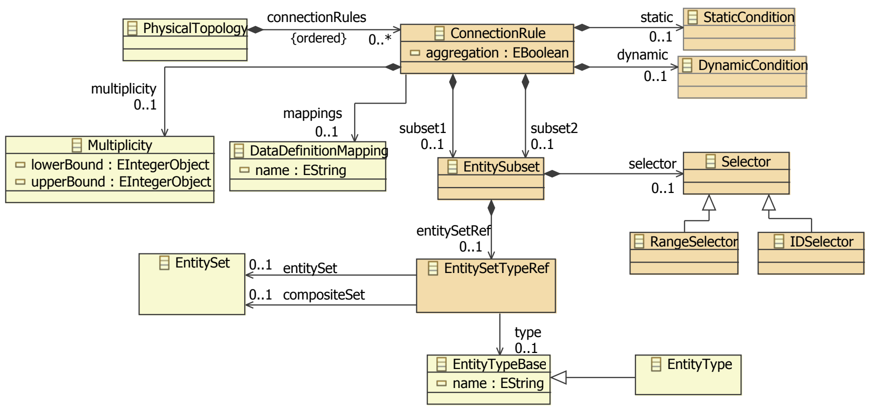


Figure 8.13: The ConnectionRule concept

Each ConnectionRule references two *EntitySubsets subset1* and *subset2*. These represent all entities of a specific type of another EntitySet. The EntityType and the EntitySet used by the subset is specified for each subset by an *EntitySetTypeRef* (a reference to an EntitySet and EntityType). Such a reference can be used to reference entities from two different sources:

Entities from local EntitySets can be referenced via the *entitySet* association when referencing an EntitySet that references a SimulationModel.

Entities from CompositeModels can be referenced via the *compositeSet* associatio, when referencing an EntitySet that references a CompositeModel.

In both cases, the type of entities to use from the referenced EntitySet is restricted by the *type* association. Figure 8.14 in the next paragraph illustrates both cases.

Furthermore, the multiplicity for the first EntitySubset has to be specified. A *Multiplicity* with *lowerBound* = 1 and *upperBound* = 3 (in the following written 1..3), for example, means that at least one element of *subset1* has to be connected to an entity of *subset2*. As mentioned above, the multiplicity in the opposite direction is fixed to 1, i.e. all entities of *subset1* have to be connected. What specific entities are being connected is determined at run-time in a random fashion. Details of this process are discussed below, when the composition layer is presented (Chapter 9.5.3).

If not all elements of *subset1* are to be connected, this is done via a *Selector*. The *RangeSelector* can be used to select a number of entities from a subset to reduce this further. For the later implementation it is important to ensure that any two EntitySubsets using the same range selector will result in the selection of the same entities. This implies that non-overlapping ranges will also result in disjunct EntitySubsets which is an important aspect to allow the user to specify different ConnectionRules for different entities of an EntitySet (e.g. as discussed in Chapter 12.4.7). If the overall scenario is small, the *IDSelector* can be used to pick a single entity with a certain ID (must be defined as a static data item) from an entity set. This way individual connections can be specified and, for example, real-world scenarios can be modeled precisely.

For capturing the additional constraints underlying a scenario model (e.g. the constraints of the example scenario defined in Section 8.4) each ConnectionRule allows to specify different types of additional constraints for selecting the entities (*StaticCondition*, *DynamicCondition*). These will be introduced in the next sections. ConnectionRules are processed by the simulation engine in deterministic sequential order (one at a time in the order in which they are defined in the scenario model).

Example

Figure 8.14 shows how ConnectionRules are modeled using the well known example scenario. The figure assumes that all required EntitySets for the example scenario have been created as discussed above and only depicts the resulting EntitySets and the elements relevant for the ConnectionRules. The right part of the figure shows a ConnectionRule that connects the residential load entities in a one-to-one fashion to the bus entities of the LV grid model. The left part shows a connection rule that selects the bus entities of the MV grid EntitySet *mvGrid* and connects them to zero or one swing bus of the *lvGrids* EntitySet that represents the two instances of the CompositeModel *LVexample*. As the opposite direction has the fixed multiplicity of one, all swing busses will be connected but not all MV grid busses will have an LV grid in this example.

OCL Constraints

To make sure that a ConnectionRule specifies a valid composition, the OCL definition **C 57** defines a corresponding invariant *connectionRuleValid* and a number of related functions. To avoid redundant OCL definitions in the appendix, this OCL definition already contains required functionality for the CompositeEntityTypes introduced in the next section. The invariant first checks if not both types referenced by the subsets of the ConnectionRule are CompositeEntityTypes. This limitation has been made to reduce complexity of the OCL constraints, as explained below. A function *checkComposability* has been overloaded and can be invoked for checking if two EntityTypes or an Entitytype and a CompositeEntityType are composable. The former uses the OCL function *compatibleTo* already defined in **C 24**. The latter checks if the type that the CompositeEntityType represents is compatible and if the ports are compatible. This check also uses OCL definition **C 62** which determines if two ports are mappable using

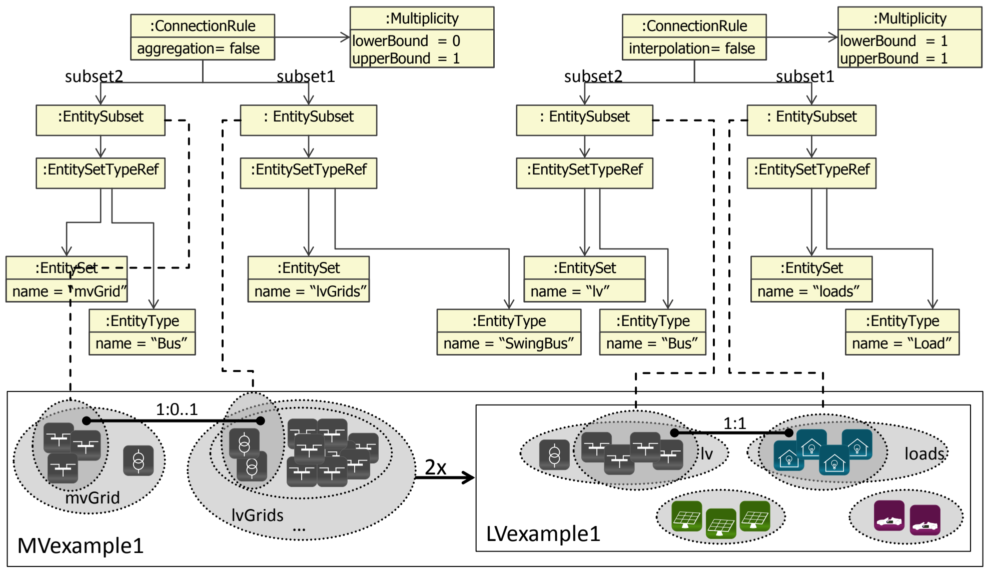


Figure 8.14: Example for modeling a connection rule

the mapping function concept defined in Chapter 7.10.

In addition to this essential constraint for ensuring valid compositions, a number of other constraints needs to be defined to ensure a proper use of the metamodel:

C 58 Multiplicity.lowerBound must be smaller or equal than the upper bound. The lower bound must be non-negative.

C 59 The EntityType referenced by an EntitySetTypeRef must be in the referenced EntitySet. Again, this constraint already includes the required check for the CompositeEntityTypes introduced in the next section.

C 60 This constraint ensures that the EntitySet referenced via the *compositeSet* association is a CompositeModel and the EntitySet referenced by the *entitySet* association is actually defined in the referenced CompositeModel. Also, if a CompositeModel is referenced, the EntitySet referenced by the *entitySet* association must not be related to another CompositeModel. The latter constraint limits a ConnectionRule to only use EntityTypes from EntitySets that are defined in the same CompositeModel as the ConectionRule or in one of the instantiated CompositeModels (but not in CompositeModels deeper down in the hierarchy). This limitation reduces complexity and allows the Scenario Expert to keep an overview of what the overall composition looks like.

C 61 This constraint validates if the lower and upper bound of the multiplicity can be met. This can only be checked if there is no static condition (see below), as these can only be evaluated at run-time. Also, the number of entities in the sets must be known. This is not the case if the cardinality of the referenced EntitySets uses a ParameterReference or a distribution function. If the number of entities is known, three cases can be distinguished (see comments in OCL definition). 1: The

ConnectionRule references a CompositeEntityType. As there is one instance per CompositeModel, the number of instances is equal to the cardinality of the set. 2: The ConnectionRule references a ModelParameterSet which is locally defined. Thus, the number of entities is the set cardinality times the number of entity instances in the model. 3: The ConnectionRule references a ModelparameterSet within a locally defined CompositeModelParameterSet (hierarchical composition). In this case the number of entities is the cardinality of the CompositeModelParameterSet times the cardinality of the ModelParameterSet times the number of entity instances in the model.

8.5.4 Composite Entity Types

The hierarchical composition mechanism works well if a CompositeModel contains a model that is hierarchical itself. This is the case for the LV grid used in the example. The swing bus entity is at the root of a hierarchical LV grid topology (LV grids usually have a radial topology configuration) and both swing busses in the example could be connected to an MV grid easily using a single ConnectionRule. Also, connections that incorporated all entities of a certain type could be specified with a single rule. For example, with only a single ConnectionRule, the PV entities of both LV grid compositions can be connected to an entity of a cloud model which provides them with detailed irradiance data.

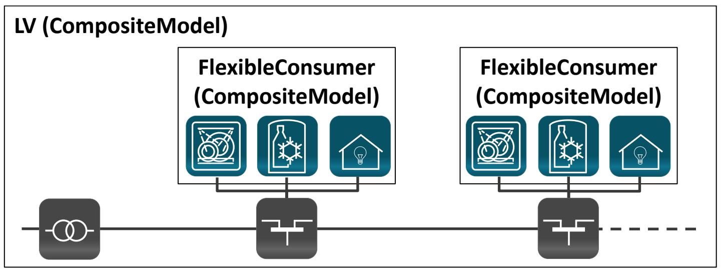

LV (CompositeModel)
FlexibleConsumer (CompositeModel)
FlexibleConsumer (CompositeModel)

Figure 8.15: Flexible Consumer - A non hierarchical example of a composite model

But the concept, as it is now, is not ideal when a composition of non hierarchical entities is to be used as building block. Figure 8.15 shows such an example using the appliance models presented in Chapter 3.5.3. A CompositeModel *FlexibleConsumer* has been created which can be used as alternative to the simple loads used in the example scenario. Each flexible consumer is comprised of three different entities (from left to right): A controllable dishwasher, a controllable fridge and an aggregated model of the remaining residential loads. Although the resulting scenario is hierarchical, the CompositeModel itself has no hierarchy as all models/entities are on the same hierarchy level and cannot be connected internally (within the CompositeModel). With the scenario metamodel so far, three different ConnectionRules are necessary to connect the entities to busses defined outside the CompositeModel. Besides this inconvenience the use of three different ConnectionRules also means that it cannot be guaranteed that the entity instances from a single CompositeModel instance are connected to the same bus entity (due to the random selection mechanism). While this may be valid for some cases it may not be for others, for example, when all models are parametrized via a common parameter defined for the CompositeModel. To avoid these inconveniences and potential causes of errors, this section introduces the concept of *CompositeEntityTypes*.

Definition 12 (CompositeEntityType) *A* CompositeEntityType *represents a group of EntityTypes that occur within a CompositeModel and can be used instead of the individual EntityTypes when defining a ConnectionRule.*

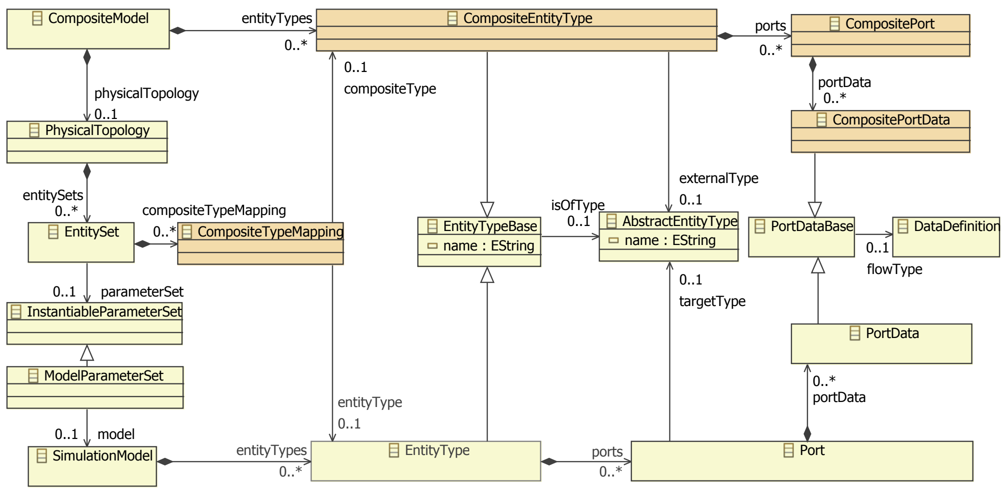


Figure 8.16: CompositeEntityTypes as logical groups of EntityTypes

As shown in Figure 8.16, a CompositeModel can define a number of CompositeEntityTypes. For an EntitySet, a number of *CompositeTypeMappings* can be defined. These can be used to map specific EntityTypes of an EntitySet to a CompositeEntityType of the corresponding CompositeModel. For the example given above, a CompositeEntityType *FlexibleConsumer* can now be defined and the three different load types can be mapped to this new type. To connect all three entities to a bus of a grid model, a single ConnectionRule is now sufficient by referencing the FlexibleConsumer type. The metamodel allows this, as the CompositeEntityType is derived from the class *EntityTypeBase* just like the regular EntityType. Therefore, the two different usage variants for an EntitySetTypeRef that were summarized in Section 8.5.3 have to be extended by a third option:

Referencing CompositeEntityTypes CompositeEntityTypes are referenced by an EntitySetTypeRef by using the *compositeSet* association to reference a CompositeModelParameterSet and referencing a CompositeEntityType within the corresponding CompositeModel (the *entitySet* association remains unused as the CompositeEntityTypes are defined directly for a CompositeModel).

To achieve consistent semantics, the use of a CompositeEntityType within a ConnectionRule is identical to specifying the same ConnectionRule separately for each EntityType. To ensure that this results in a valid composition, the EntityTypes that are mapped to a CompositeEntityType must have some similarities. As defined above, the composability of two EntityTypes is determined on a per port basis. Therefore, the CompositeEntityType also allows to define a number of *CompositePorts* which can reference DataDefinitions from the reference data model via the *CompositePortData* class. The classes Port and PortData cannot be reused as these have a number of attributes that are not applicable to the specification of ports for the CompositeEntityType. An EntityType can now be mapped to a CompositeEntityType when all mandatory data flows of all CompositePorts can be satisfied and vice versa. This is a stricter definition as for the composability of EntityTypes which only required that the data flows for at least one Port can be satisfied. The reason for this is shown in an abstract example in Figure 8.17. Two hypothetic EntityTypes A and B are mapped to a CompositeEntityType using the less strict definition. This is possible as both can satisfy one port of the CompositeEntityType. A ConnectionRule between the defined CompositeEntityType and another EntityType1 should have the same result as two individual ConnectionRules between EntityTypes A and 1 and B and 1. However, as EntityB does not offer a Port A, this will not be a valid composition. Therefore, only if all ports of a CompositeEntityType can be satisfied, an EntityType can be mapped to it. Details of this checking procedure can be found in the OCL definitions below.

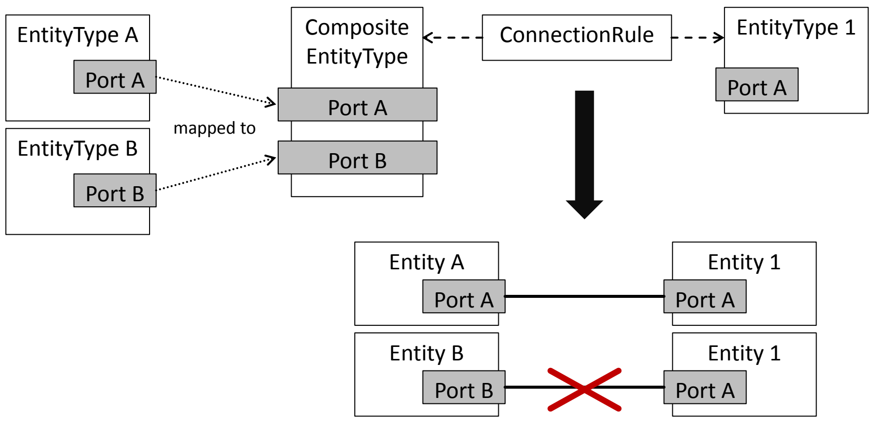


Figure 8.17: Example for insufficient composability constraints for CompositeEntityTypes

Besides the port based data flow compatibility two other validation mechanisms were introduced before. These are related to AbstractEntityTypes and the defined value ranges for the data flows. How these validation mechanisms can be applied/kept for the concept of CompositeEntityTypes is discussed in the following paragraphs.

AbstractEntityTypes

Each Port of an EntityType can define an allowed *targetType* (see Figure 7.8). This constraint can also be used for validating a ConnectionRule that connects a regular EntityType with a CompositeEntityType, as the latter can also define an AbstractEntityType via the *isOfType* association inherited from the common base class EntityTypeBase. However, for the EntityTypes that are mapped to a CompositeEntityType it also has to be made sure that a common target type that is compatible to all mapped EntityTypes is available. Therefore, the *externalType* association allows to specify the AbstractEntityType that the CompositeEntityType can be connected to. So all in all, a CompositeEntityType references two AbstractEntiyTypes: the one it represents when being used outside the CompositeModel and the one reflecting the target type of the EntityTypes that are mapped. Figure 8.18 illustrates this duality.

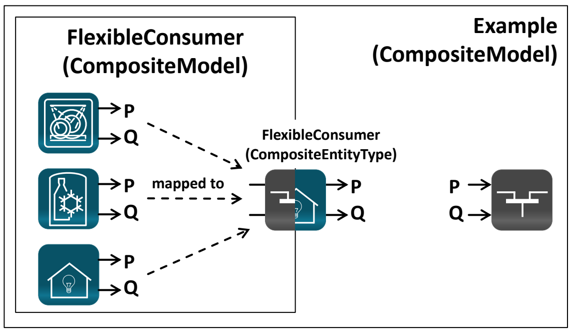

FlexibleConsumer (CompositeModel)
Example (CompositeModel)
P
Q
P
Q
P
Q
mapped to
FlexibleConsumer (CompositeEntityType)
P
Q
P
Q

Figure 8.18: CompositeEntityTypes have a dual identity

Data Flow Limits

For incorporating the data flow limits (see Figure 7.11) into the CompositeEntitySet concept, two options exist. One option is to dynamically infer the data flow limits from the mapped EntityTypes. Doing so, the data flow limit of a flow defined for a CompositeEntityType is the intersection of all data flow limits specified by EntityTypes that provide or receive that flow. This approach has two big disadvantages. First, the implementation becomes more complex, as an additional inference mechanism for the data flow limits has to be implemented. Second and most important, the limits that apply to a composite data type cannot be immediately seen by the Scenario Expert.

The second option is to allow the definition of limits for the DataDefinitions used by a CompositeEntityType. Besides additional specification effort, this way the limits are made explicit and no inference mechanism is required. For these reasons this option has been implemented. When specifying a CompositeTypeMapping or a ConnectionRule involving a CompositeEntityType, these limits are included in the validation and ensure that the resulting composition is valid. Details of this checking procedure can be found in the OCL paragraph below.

Design alternative

An alternative to the class structure presented above is to let the CompositeModel inherit from EntityTypeBase. By doing so, the whole CompositeModel would represent an EntityType. However, this would not be as transparent as the chosen approach, as a SimulationModel can have a number of EntityTypes and the chosen approach allows to define a number of CompositeEntityTypes for a CompositeModel as well. The symmetry of Figure 8.16 nicely illustrates the similar structures of CompositeModels (upper part) and SimulationModels (lower part).

Example

Figure 8.19 shows how a CompositeEntityType is used for the flexible appliances example introduced in this section. It is important to understand that the actual composition at run-time does not include the CompositeEntityType anymore but has been resolved to individual connections between the entities.

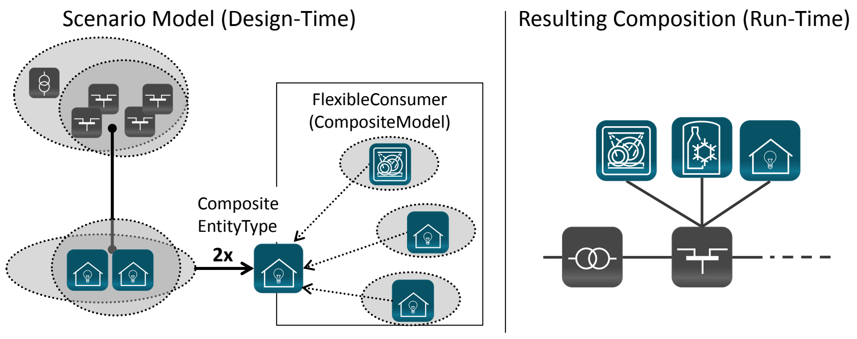

Scenario Model (Design-Time)
Resulting Composition (Run-Time)
FlexibleConsumer (CompositeModel)
Composite EntityType
2x

Figure 8.19: Example for using a CompositeEntityType

OCL Constraints

To ensure that the usage of CompositeEntityTypes also results in valid compositions, a number of additional constraints is required. First of all, the CompositeTypeMapping is accompanied by two constraints **C 63** and **C 64** ensuring that only those EntityTypes can be mapped to a CompositeEntityType that can satisfy all of its ports, meet the data flow limits and are compatible to the specified external type. For the purpose of this thesis, the connection of two CompositeEntityTypes is not allowed (see invariant *connectionRuleValid* in **C 57**) to ease implementation and reduce complexity. At a later point such connections could be added by extending the constraints.

8.5.4.1 Topological Conditions

If two entities are to be connected or not, often depends on other connections that have already been established before [**R 5** – Entity Dependencies]. In the example scenario introduced in the beginning of this section, for example, EVs are only to be connected to those busses of the power grid that have a PV system connected to it. The scenario metamodel must offer a way to specify such conditions.

At composition-time, the application of the ConnectionRules creates connections between different entities that exist within the models of simulator processes. When interpreting these entities as nodes and creating edges between these if any data flow exists (abstracting from specific data flows), the result is a undirected graph. This topology graph shows what entities are connected by some data flow. This graph roughly corresponds to the topological images used in this thesis when describing the Smart Grid scenarios during requirements analysis, e.g. Figure 3.2. The required conditions can now be expressed by subgraphs in the topology graph. Hence, they are called topological conditions. Figure 8.20 shows this using the example scenario introduced above.

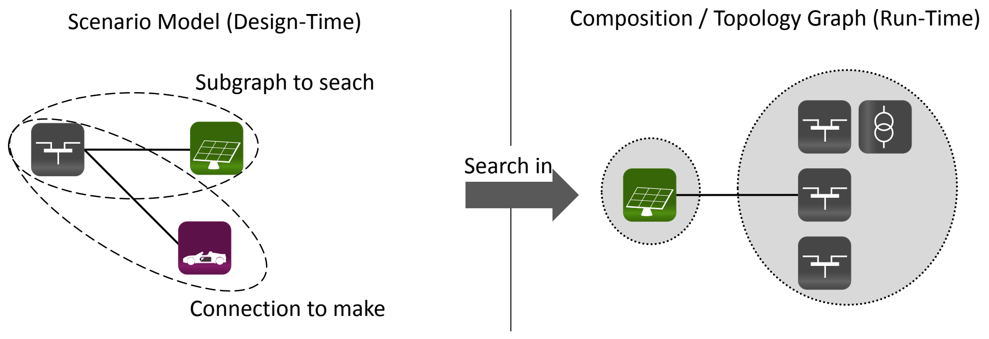

Scenario Model (Design-Time)
Subgraph to seach
Connection to make
Search in
Composition / Topology Graph (Run-Time)

Figure 8.20: Expressing topological conditions as subgraphs

Obviously this is a very simple example. Mosaik allows to specify more complex topological conditions by not only allowing to specify conditions with simple, direct connections between two entities but also connections that indirectly connect two entities via a path of different other entities. Furthermore, instead of simple *exists* operation, the number of entities that have to be connected can be specified. In the used example an EV module may only be connected to busses that have one or more PV systems. Figure 8.21 shows the corresponding extension to the scenario meta model.

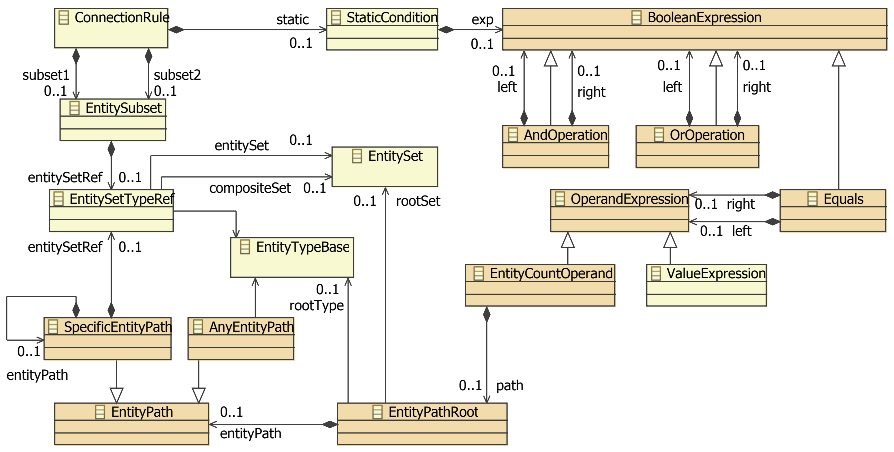


Figure 8.21: Metamodel extension for defining topological conditions

As mentioned above each ConnectionRule specifies two EntitySubsets containing those entities of an EntitySet that have a specific type. A topological condition is now defined by using the *EntityCountOperand* class. It defines the root of the path (EntitySet and EntityType to start from) by referencing one of the EntityTypeBase objects that are part of one EntitySubset of the corresponding ConnectionRule (see OCL constraint **C 59**). Later, at composition-time, for each of the entities in the referenced set, the referenced *EntityPath* objects (if specified) is being evaluated. That means it is checked if the topology graph contains a connection to an entity of the EntitySet specified by the topological path. This can either be a specific path (using the class *SpecificEntityPath*) considering specific types end sets of entities or any reachable EntityType (or CompositeEntityType) regardless of the type of entities encountered along the path (using the class *AnyEntityPath*). At composition time, the

EntityCountOperand returns an integer number indicating the number of entities that could be reached from the root EntitySet/Type of the path. The following example illustrates this approach.

Example

In case of the example scenario the EntityPathRoot references a bus and defines a specific path to the EntityType PV. Figure 8.22 shows this part of the scenario model.

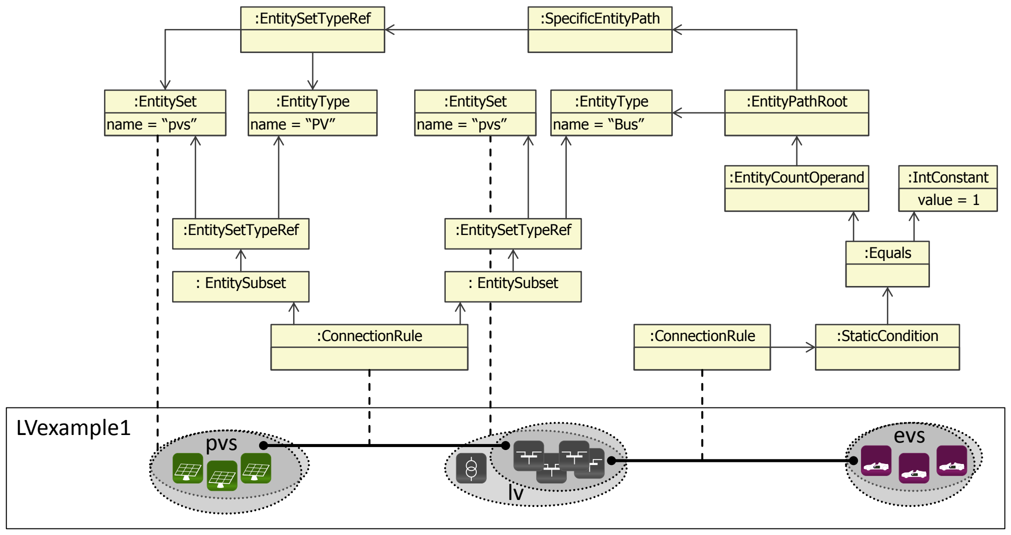


Figure 8.22: Example for using topological conditions

OCL constraints

To ensure that the topological conditions specified in a static condition are valid, the following OCL constraints must hold:

C 65 The EntitySet referenced by the *rootSet* association must be part of ConnectionRule that specifies the static condition which includes the EntityPathRoot object.

C 66 The referenced *rootType* must be in the *rootSet*.

Also, the ValueExpressions used in the expression tree of the StaticCondition must have compatible types. For example, it is not sensible to compare string and integer expressions. This is ensured by OCL constraint **C 67**.

8.5.5 Static Data & User Defined Functions

For providing even more flexibility, the conditions of a ConnectionRule can be extended by user defined functions operating on the entities static data. Figure 8.23 shows a scenario where such functionality can be used to keep the scenario specification short. It was already shown in Chapter 3.2.3 and can, for example, be used for analyzing measures (e.g. stationary electric storages) to smoothen load gradients caused by cloud transients in areas with much PV feed-in.

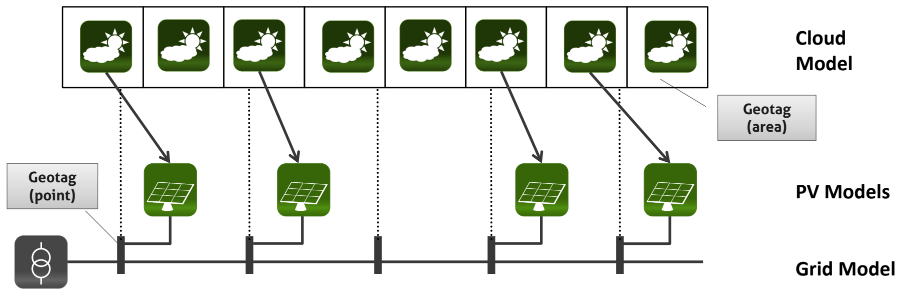


Figure 8.23: Detailed PV simulation with a cloud model providing solar irradiance data

It includes three different models, a power grid model, a PV model (both known from the former examples) and a cloud model that models the solar irradiance in a cell based fashion for different geographical regions. The cells have to be connected as input to the PV modules to refine the overall scenario. It is assumed that the grid model offers static data about the geographic location of a bus and the cell entities of the cloud model provide the cell area as static data as well. For large scale scenarios it would be cumbersome and error prone to connect cells and PV modules manually. Hence, a mechanism for automatically creating a connection between a cell and those PV modules that are geographically located in the area of the cell has to be created.

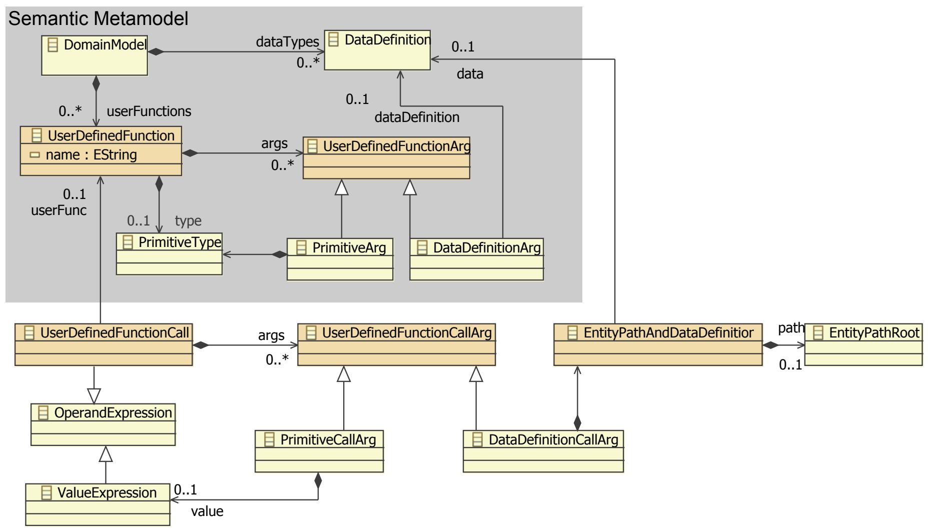


Figure 8.24: User-defined functions for incorporating static entity data into ConnectionRules

As this is only one example for static data and the static data items can be defined freely by the users of mosaik in the reference data model, mosaik cannot offer a mechanism to match different static data items a priori. Figure 8.24 therefore shows an extension to the semantic as well as to the scenario metamodel which allows to specify UserDefinedFunctions and incorporate them into the static condition of a ConnectionRule. Each function defines a return type via the *type* association and can define an arbitrary number of arguments (*UserDefinedFunctionArg*). The argument types are either existing DataDefinitions (via the *DataDefinitionSubclass*) or primitive types. Similar to the mapping functions already introduced in Chapter 7.10, each UserDefinedFunction can only be declared in the metamodel but the actual implementation has to be defined in general purpose language and provided to the later implementation of mosaik in a plug-in fashion. A UserDefinedFunction is incorporated into the static condition of a ConnectionRule by the *UserDefinedFunctionCall* class, which is derived from the OperandExpression class and can thus be used in any relational expression that is defined. Actual argument values are specified via the subclasses of the *UserDefinedFunctionCallArg* class (either specifying a ValueExpression (see Figure 8.8) or an *EntityPathAndDataDefinition*. The latter references a DataDefinition that is declared as static data for an EntityType referenced via the EntityPath concept already introduced above.

Example

Figure 8.25 illustrates how the proposed user-function mechanism works, using the above example of the PV simulation that is fed with region specific sun irradiance values. For the sake of clarity, the ConnectionRule itself is not part of the illustration. Assuming that a ConnectionRule between PV entities and the buses of the power grid has already been defined, the connection between PV entities and the cell entities of the cloud model can be defined by calling a UserDefinedFunction *pointInArea*. It returns true, if a given point is in a given area. As mentioned above, the area is defined as static data for the cell entities and can thus be directly specified as argument for the function call using the EntityPathAndDataDefinition class. As the PV entities do not define a location themselves, the EntityPath is used to navigate to the connected busses and the static position data defined for these can be used. At composition-time, the user-defined function is invoked for each bus-cell pair that exists and a cell entity matching the location of a bus is connected to the related PV module.

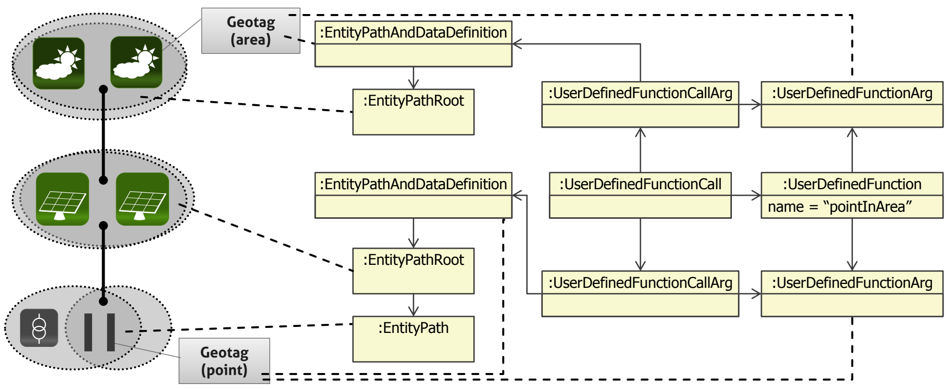


Figure 8.25: Example for using a UserDefinedFunction

OCL constraints

OCL constraint **C 68** ensures that the type of the provided argument for a UserDefinedFunctionCall matches with the arguments in the definition of the corresponding UserDefinedFunction.

8.5.6 Dynamic Condition

EVs are a very special type of entity in Smart Grid scenarios, as these are the only entities moving between different nodes (charging locations) of the power grid. So far, to support this functionality, mosaik allows to specify a dynamic condition for a ConnectionRule. In contrast to the topological conditions and user-defined functions presented above, which operate on static data and are only evaluated once at composition-time, the dynamic condition is reevaluated at every simulation step and operates on the dynamic data of the entities. Figure 8.26 shows how the scenario metamodel has been extended in a simple way to support dynamic conditions. Currently a ConnectionRule can only specify a single dynamic condition as this seems sufficient for the EV use case (which currently is the only reasonable case for using such conditions). Each condition references a PortData object which must be defined in one of the EntityTypes that are to be connected. OCL constraint **C 69** has been added to ensure this. If the DataDefinition of the referenced PortData is an ObjectType, the concept of the ObjectTypeFieldReferences (see Chapter 7.6) can be used to pick a specific field of the object type. Finally, a constant value has to be provided for which the condition becomes true and the ConnectionRule thus active. OCL constraint **C 70** ensures that the type is compatible to the referenced PortData. This allows to enable a ConnectionRule based on a single attribute value. Support for more sophisticated conditions (e.g. multiple attribute conditions) could be added similar to the expression structure used for the static conditions. As this does not require new concepts this has not been considered, yet. Also, the current design only allows to specify dynamic conditions for EntityTypes and not for CompositeEntityTypes. As dynamic conditions have been added specifically to support EVs this is first of all sufficient.

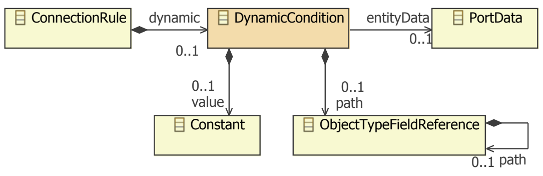


Figure 8.26: Metamodel extension for dynamic conditions

8.5.7 Simulation Study

To actually perform one or more simulation runs, some basic information is required. At least a CompositeModel that represents the physical topology of the Smart Grid has to be selected and its parameters have to be provided with values. Figure 8.27 shows the *SimulationStudy* class that finalizes the scenario metamodel and represents the entry point for performing simulation runs. Details of the shown structure are disussed in the

Table 8.2: Combination types of SimulationStudy values

Parameter Values		start 2011-01-01	stop 2011-03-31	numEvs 1, 2	numPVs 1, 2
Combination Type	#Run				
Linear	1	2011-01-01	2011-03-31	1	1
	2	2011-01-01	2011-03-31	2	2
Cartesian	1	2011-01-01	2011-03-31	1	1
	2	2011-01-01	2011-03-31	1	2
	3	2011-01-01	2011-03-31	2	1
	4	2011-01-01	2011-03-31	2	2

following paragraphs.

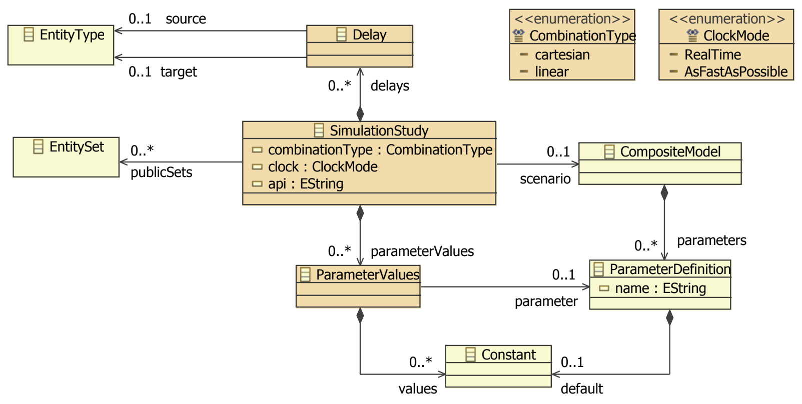


Figure 8.27: A SimulationStudy object defines specific parameters for the instantiation of a CompositeModel.

Parametrization

To be able to compose the referenced CompositeModel at run-time, its *parameters* have to be provided. The SimulationStudy allows to define a number of *ParameterValues* each of which references a single ParameterDefinition and provides a number of Constant values for each ParameterDefinition. At run-time, the simulation engine executes multiple simulation runs for the different parameter values. The *combinationType* attribute governs the way in which the values are combined. A *linear* combination type executes simulation runs for the first, second, third, etc. value of each ParameterDefinition. The *cartesian* combination type executes one simulation run for each possible value combination. Table 8.2 illustrates these two possibilities for three example parameters. The parameters *start* and *stop* must be defined for every CompositeModel (see **C 71**) as otherwise potentially time-variant simulation models (see Chapter 2.1.4) do not behave correctly and simulators do not know how many steps to perform which may be an important information for caching, preparing input data, and so on.

The linear combination type is useful when different time spans have to be analyzed (winter, summer, etc.) as these can be specified for the start/stop parameters and the simulation is executed for each time span. Cartesian combination is not suitable in this case as it would produce combinations where the start is after the stop time. Cartesian combination is in particular useful when a large number of other parameters has to be combined. In this case different SimulationStudy objects have to be used to additionally simulate multiple time spans. A more sophisticated combination mechanism with partly linear and partly cartesian combination possibilities for these situations can be added to the SimulationStudy but is not within the scope of this thesis.

Controllable entity sets

As described in the next chapter dealing with the composition layer, it is beneficial to describe what EntitySets contain EntityTypes needing to be accessed by the control mechanism that is to be tested. In many cases involving simulators with different step sizes this allows to shorten the time span in which control mechanisms can interact with the physical topology. The SimulationStudy allows to specify these EntitySets via the *publicSets* association.

Handling cyclic data flows

In Figure 3.4 of the requirements analysis it was shown that compositions with cyclic data flows may occur. Hensen [Hen95] presents two approaches for dealing with such cycles, as shown in Figure 8.28:

The "Ping-Pong" method The entities are simulated in sequence. Each entity uses the output data generated by the other entity in the previous step. It has to be decided which entity runs first.

The "Onion" method For single time steps the entities are simulated in an iterative fashion until satisfactory small errors are achieved.

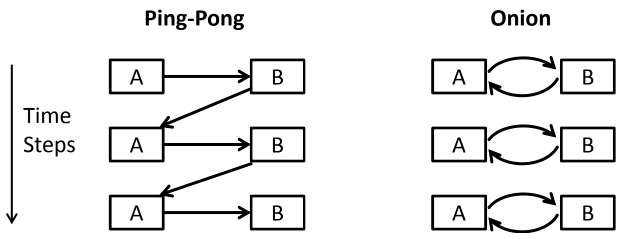


Figure 8.28: Schematic representation of ping-pong vs. onion approach [Hen95]

In Chapter 3.4 it was argued that rollback functionality is required to support simulations with variable-step sizes. The same is required to support the "Onion" method. In Chapter 6.2.1 it was argued that implementing such a functionality is complex and not possible for many COTS packages. Therefore, the concept developed in this thesis currently focuses on the implementation of the "Ping-Pong" method. This limitation was already discussed in Chapter 6.5.

The "Ping-Pong" method requires one of the entities involved in a cycle to be simulated based on data from the previous time step (the entity that is being simulated first). The metamodel allows to specify this via the *Delayed* class defining a data flow from one EntityType to another one that is causing the delay. For the example in Figure 8.28 the *source* type would be B and the target type A, as A processes the data from B with a delay of one time step.

OCL Constraint

The following OCL constraints have been defined to ensure that a SimulationStudy can be interpreted properly on the composition layer:

C 71 A CompositeModel must at least define the parameters *start, stop* of type DateTime so that these can be set in the SimulationStudy to let the simulation engine know what timespan (how many steps) to simulate.

C 72 A SimulationStudy must only reference those ParameterDefinitions that are defined by the referenced CompositeModel.

C 73 The SimulationStudy provides all required (non-default) parameters of the referenced CompositeModel exactly once.

C 74 In case of linear value combination, all specified ParameterValues must define the same number of values or only a single value. Otherwise it is not clear, how these are to be combined.

8.6 Discussion

In this section the scenario metamodel acting as a well defined scenario specification mechanism [**R 2** – Scenario Specification Mechanism] has been presented. It references the simulators, simulation models and their entities defined on the semantic layer. The scenario metamodel allows to create scenario models that specify parameter values for simulators and models, defines how many instances of each model are to be used and how these are to be connected. By relying on the OCL constraints defined on the semantic layer it is ensured that ConnectionRules are only defined for those EntityTypes that are compatible (e.g. see **C 57**). Also, additional OCL constraints introduced on the scenario layer ensure that only valid parameter values can be provided.

Figure 8.29 shows the most important elements of the scenario metamodel. It can largely be divided into five parts. The *Parametrization* part defines the parameter values required to instantiate simulators and models defined on the semantic layer as well as CompositeModels defined in a scenario model. The *Composition* part, starting at the CompositeModel, defines the PhysicalTopology by specifying EntitySets which govern model instantiation and ConnectionRules among the EntityTypes and CompositeEntityTypes defined in these EntitySets. The *Hierarchy* part highlights those classes that allow hierarchical reuse of CompositeModels. Conditions for ConnectionRules can be defined using the *ConditionExpression* section and values that are used as operands within these conditional expressions or as parameter values for ParameterSets are defined in the *ValueProvider* section. This includes simple Constant values and Distributions as well as complex value expressions such as the EntityCountOperand used for defining

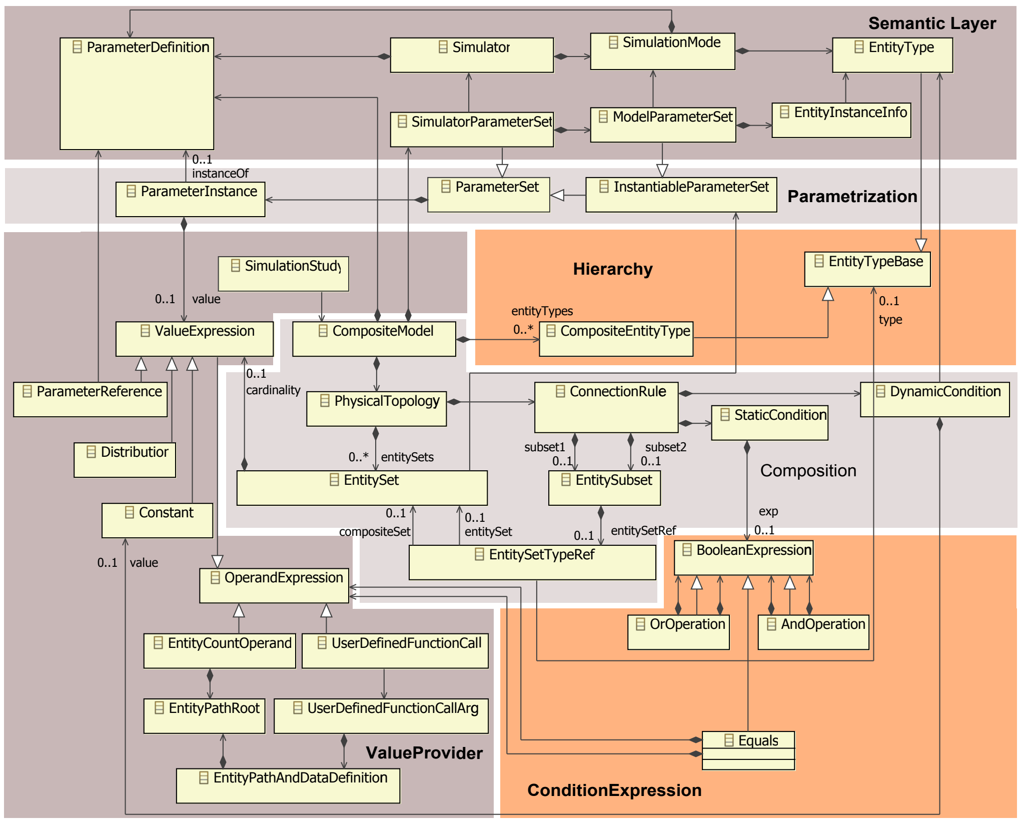


Figure 8.29: Summary of the major classes of the scenario metamodel

topological conditions and UserDefinedFunctionCalls used to compare static entity data. Due to layout restrictions, the CompositEntityTypeMapping class has been left out. Also, the different subclasses of BooleanExpression (e.g. GreaterThan, SmallerThan, etc.) are not shown.

8.6.1 Requirements

In Section 8.1 the requirements relevant to the scenario layer have been presented. In the following it is discussed how the requirements have been met by the design of the scenario metamodel.

R 4 – Scalable Scenario Definition The most important requirement and the unique feature of the mosaik scenario layer is the capability of specifying large-scale scenarios without much effort. This means for scenarios involving a large number of entities, an explicit instantiation and specification of every connection between two entities is not required. Rather, the scenario metamodel allows to specify potentially large sets of entities and ConnectionRules which connect two EntityTypes of two sets in a randomized fashion while adhering to additional constraints. These can be multiplicity, topological or static data based constraints. Furthermore, it is not required to specify individual data flows but mosaik automatically connects all compatible data flows between two entities on a port-to-port basis. The details of the underlying connection algorithm are defined on the composition layer in the next section. Such a random based connection is especially useful when the exact distribution of elements in a power grid is not known, e.g. for future scenarios. But it is also useful when not using a real power topology but representative ones (e.g. as done in [NTS+11]). However, the IDSelector class introduced in Figure 8.13 also allows to select one or two specific entities for a ConnectionRule, allowing to accurately model a real-world setup. The corresponding EntityTypes are expected to have a static attribute named *id* in this case. Therefore, all three scenario modeling options presented in Figure 8.5 can be implemented with the scenario metamodel. The ability to instantiate CompositeModels as building blocks in other CompositeModels allows hierarchical modeling which matches the natural structure of the power grid. Furthermore, the definition of CompositeEntityTypes allows to connect an arbitrary number of entities with a single ConnectionRule (see Figure 8.19).

R 5 – Entity Dependencies The mechanism for specifying topological constraints introduced in Section 8.5.4.1 allows to account for the fact that connections between entities may depend on other connections previously made. This was illustrated in Figure 8.22 using the example of EVs that are only connected to those busses that already have a PV system connected to it.

R 7 – Attribute Based Composition In Chapter 3.2.3 it was shown that environmental models can be integrated easily into a Smart Grid scenario when entities can be composed based on their attributes (e.g. by comparing geographic positions). The scenario metamodel accounts for this requirement by offering the mechanisms of UserDefinedFunctions that can be used to implement comparison operations on static data items. These functions are to be provided by the user in a plugin-fashion as it is also the user who defines static data types in the reference data model. Mosaik cannot anticipate the defined types and possible operations for these beforehand.

R 14 – Moving DER The metamodel allows to define dynamic conditions that can be used to activate the connections created by a ConnectionRule depending on dynamic data of entities.

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs For some model types the number of entities a model instance will contain depends on their parametrization. The scenario metamodell accounts for this by allowing to attach entity instance information (class EntityInstanceInfo) to the ModelParameterSets. This allows to validate the multiplicity constraints of ConnectionRules involving even these models. The fact that a model can contain more than one type of entities has been accounted for by having to select a specific EntityType for a ConnectionRule (classes EntitySubset and EntitySetTypeRef).

R 19 – Different Temporal Resolutions As simulators may support different step sizes, the SimulatorParameterSet allows to select the step size to use.

R 29 – Central Parameter Definition The scenario model allows to specify all required parameters for the simulators and their models. Values can be constants or randomly sampled at run-time from distribution functions (inspired by the GridLAB-D scenario mechanism). Additional flexibility is gained by allowing to use parameter definitions of a CompositeModel as values. This way, simulator and model parameters can still be changed when using a CompositeModel as building block in hierarchical scenario model.

8.6.2 Validity aspects

As mentioned in Chapter 7.12, most of the identified validity aspects needed to be considered by the scenario layer presented in this chapter. In the following it is briefly summarized how these were considered by this layer.

Validity Aspect 1 – Data types must match As already mentioned in the last chapter, the usage of a common reference data model which also defines the detailed data structures (including specific data types) ensures that data types match as composability constraints operate on the elements of this data model.

Validity Aspect 2 – Origin/destination types must match OCL constraint **C 57** which validates the composability of the EntityTypes specified for a ConnectionRule considers the AbstractEntityTypes that can be specified as expected target types for each Port. Also, the concept of CompositeEntityTypes introduced in Section 8.5.4 considers the AbstractEntityTypes by defining both, an internal and an external type, such that the validity aspect could be integrated (see OCL constraints **C 63** and **C 64**).

Validity Aspect 3 – Ranges must match OCL constraint **C 57** considers this aspect by using the port related function *compatibleTo* which was already defined in **C 24** on the semantic layer. How range validity has been considered for CompositeEntityTyeps has been discussed in a dedicated paragraph of Section 8.5.4.

Validity Aspect 4 – Inputs satisfied Again, OCL constraint **C 57** considers this aspect by using the port related function *compatibleTo* which was already defined in **C 24** on the semantic layer. For CompositeEntityTypes **C 64** additionally checks this aspect for the mapping of EntityTypes to CompositeTypes class.

Validity Aspect 5 – Temporal input availability The semantic layer defined a simple flag for the elements of the reference data model, indicating continuous or sporadic availability. Therefore, this aspect is implicitly covered by relying on the reference data model. Hence, it is implicitly made sure by the composability constraints that only continuous->continuous or sporadic->sporadic flows are connected. No further information about the availability is currently provided, as discussed in Chapter 7.8.

Validity Aspect 6 – Data flow units match Similar to [**Validity Aspect 1** – Data types must match], this aspect is implicitly covered by relying on the reference data model. Potential unit conversions have to be done in the SimAPI implementation and thus cannot be validated automatically.

Validity Aspect 7 – Temporal data validity This aspect is currently unconsidered for the reasons discussed in Chapter 7.8.

8.6.3 Conflicts

Although suggested by the concept of ConnectionRules introduced in this chapter, the process of interpreting a scenario model is not identical to classical rule-based systems [FH]. The ConnectionRules are neither activated based on boolean conditions nor are they processed repeatedly. Rather, they are processed once in a predefined sequential manner. Therefore, no conflict resolving for multiple activated strategies is required. Also, the ConnectionRules cannot directly contradict each other as they only allow to add connections and not to remove or modify existing connections. However, there may be indirect dependencies in that sense that changing or adding a ConnectionRule that is processed before another one may cause the static condition of a ConnectionRule to break (i.e. when it defines a topological condition). This can cause the composition to fail at run-time. Due to the stochasticity of the composition procedure, checking for such interdependencies is non-trivial and in many cases impossible, for example when the number of entities in a set is not known at design-time or when a static condition is used which involves static entity data. Such checks have therefore not been considered. Also, in many cases such interdependencies can be avoid when using greater then or less than instead of equal operands in the static conditions (i.e. connect an EV to a bus when there are >= 1 PV systems). In such a case adding new ConnectionRules (connecting more entities) is not an issue.

8.6.4 Limitations

Besides this limitation intrinsic to the random approach, there are a number of limitations that have either been made deliberately to reduce complexity (while still allowing to specify the identified types of Smart Grid scenarios) or are intrinsic to the random-based composition approach that was chosen to allow a compact description of large-scale scenarios.

Connection limits cannot be checked The Port class defines a *maxConnections* attribute to indicate the maximal possible number of connections that this Port can handle. This attribute is currently not checked at scenario design-time. Due to the random based selection mechanism this aspect could only be validated in cases where no static conditions are defined and the number of entities is known (i.e. specified by the SimulationModel or defined through an EntitySetInstanceInfo). As this is not always the case, a runtime-check has to be implemented anyway. Therefore, to avoid redundancies, a check at run-time only is more reasonable.

ConnectionRule multiplicity check incomplete In Chapter 6 OCL constraint **C 61** was introduced, which checked if the lower and upper bound of a ConnectionRules multiplicity can be met. Checking this constraint is impossible if the number of entities that a model instance contains is not defined or if the number of model instances is specified as random distribution. The constraint also does not work when the number of model instances (the cardinality of an EntitySet) is provided by a ParameterReference. However, this could be checked if the invariant is defined for a SimulationStudy and works in a recursive, top-down fashion such that the values of the referenced parameters are known. As yet, this has not been specified, to keep the constraints as simple as possible and because a run-time check for the multiplicity bounds has to be implemented anyway (see Chapter 9.5.3) due to the above cases in which the bounds cannot be checked.

No transitive, CompositeEntityTypes For the time being, a CompositeEntityType can only be mapped to regular (non-composite) EntityTypes. Although the class structure would also allow to map CompositeEntityTypes to other CompositeEntityTypes, this transitive mapping is currently not supported to ease implementation and limit complexity.

CompositeEntityType composability For similar reasons, it is currently not possible to specify a ConnectionRule that connects two CompositeEntityTypes.

Limited hierarchical composability When specifying an EntitySet for a Composite-Model, the EntitySetTypeRef class can be used to access the EntitySets defined within this model. However, it is currently not possible to access EntitySets that are defined in a CompositeModel of these EntitySets. Again, this is to limit the complexity of constraints and in particular, to keep the scenario description understandable for the Scenario Expert. Allowing connections of EntitySets that are arbitrarily deep in the composition hierarchy would allow compositions that are only hard to understand.

Composition of implementations Falkenhainer and Forbus [FF91] discuss two variants for creating a scenario model. In the first variant, a scenario model is created for a specific simulator. The second variant, a domain model is built first and the scenario model build is built by composing elements of this model. This way, the scenario model is independent from an actual implementation. The presented scenario concept can be compared to the first variant, as the mosaik scenario metamodel relies on references to the actual simulator descriptions. This way, exchanging the simulation models that are used for a scenario is not possible without touching the scenario model. In the current concept the ParameterSet and the EntityTypes used in the ConnectionRules have to be changed when a different model is being used.

To implement the second variant described by Falkenhainer and Forbus [FF91], the following changes could be made:

1. All EntityTypes of a SimulationModel must define different AbstractEntity-Types.
2. The EntitySetTypeRef is changed to not operate on the EntityTypes but on their AbstractEntityTypes (as these are defined in the domain specific reference model and not implementation specific).This way the desired decoupling could take place.
3. To avoid that ParameterSets used in the scenario model are still tied to specific simulation models, the parameters of the simulators and models also need to reference elements of the reference data model. This way, a mapping from implementation independent, conceptual parameters used in the scenario model to actual parameters of the simulators/models could be implemented.

As these changes increase complexity of the scenario metamodel and do not change the basic concept of constraint and random-based composition of large-scale scenarios, an implementation independent scenario specification has not been implemented but may be subject to future work.

9 Composition Layer

The composition layer is responsible for a number of tasks. It has to interpret a scenario model, initialize as many simulator processes and models as needed to simulate the scenario, establish the data flows between the simulated entities and advance the simulators in the correct order. As the scenario model adheres to the scenario metamodel and the additional OCL constraints, algorithms for the composition layer can be defined that perform these tasks automatically. The different algorithms are presented in the remainder of this section. The actual implementation of these algorithms and the other conceptual layers is described in the Chapter 11. Figure 9.1 shows the context of the composition layer.

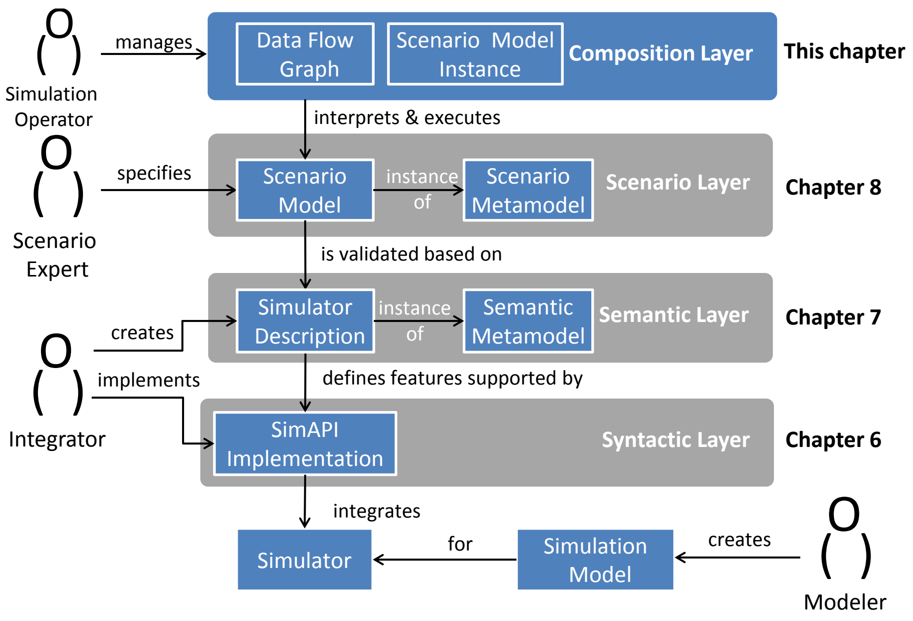


Figure 9.1: Context of the composition layer

9.1 Requirements and Related Work

The first sections of this chapter deal with the interpretation and instantiation of scenario models. The presented algorithms have to meet requirements [**R 6** – Cyclic Data Flow Support] and [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs]. As the algorithms are specific to the new scenario modeling concept of mosaik presented in the last chapter, these algorithms have not been influenced by related works. At the end of this chapter an algorithm for simulator scheduling is presented in Section 9.6. To increase coherence, relevant requirements and related works are discussed in Sections 9.6.1 and 9.6.2, just before the algorithm is presented.

9.2 Constructing a Data Flow Graph

In this section, an algorithm for creating a data flow graph (DFG) is being presented. It will represent the data dependencies between the different EntityTypes. This is the first step of the composition phase and it serves two purposes. First, the DFG allows to detect cycles in the composition (see Section 9.4). Second, there may be EntityTypes in different sets that are to be connected by a ConnectionRule and originate from different models of the same simulator. These models must then be executed by two distinct simulator processes to ensure that these can be stepped one after another. The DFG allows to detect such situations as well (see Section 9.3). As information about such EntitySets is needed prior-to instantiation, the DFG is constructed solely based on the scenario model. This also allows to detect cycles and check if these are resolved by corresponding delays specified in the SimulationStudy without the overhead of simulator instantiation, which can be very time consuming, especially for larger scenarios. By doing this prior to instantiation the simulation operator can obtain immediate feedback. Also, the algorithm for the detection of cycles will perform faster as the DFG only includes nodes representing the EntityTypes and their data flow dependencies and not the final, potentially large number of actual entity instances.

Definition

According to Quan [Qua12], A DFG is "a directed graph that shows the data dependencies between a number of functions". In case of mosaik, these functions are the EntityTypes and their inputs and outputs (entity instances are not available at this stage). The DFG is formally defined as follows:

Definition 13 (Data Flow Graph) *$DFG = (V, E)$ with V being a set of vertices (nodes) and E a set of directed edges connecting two vertices each.*

The semantics of a DFG are such that each node (representing a function) can be executed as soon as all required inputs are available. There may be times when many nodes are ready to be executed [Qua12]. As a model described on the semantic layer may contain more than one EntityType, a model can only be simulated for the next time step when the inputs for all instances of its EntityTypes are available. This is captured in the DFG by adding additional nodes representing the EntitySets (which represent one or more model instances) as shown further below.

DFG creation

The algorithm for creating the DFG starts with the CompositeModel referenced by the SimulationStudy (see Figure 8.27) that is to be executed. The algorithm is recursively applied in a depth-first fashion for all the CompositeModels referenced by an EntitySet. When a CompositeModel only made up of SimulationModels (and no CompositeModels) is reached, the contained ConnectionRules are analyzed. For each of the two EntityTypes referenced by the ConnectionRule a node is added to the DFG if a similar node does not already exist. Next, for each possible data flow direction between the two nodes a directed arc is added from the node providing the data to the node representing the receiving EntityType. To identify if two nodes are similar each node has a name that is the name of the EntityType, prefixed by the composition hierarchy of the EntitySets.

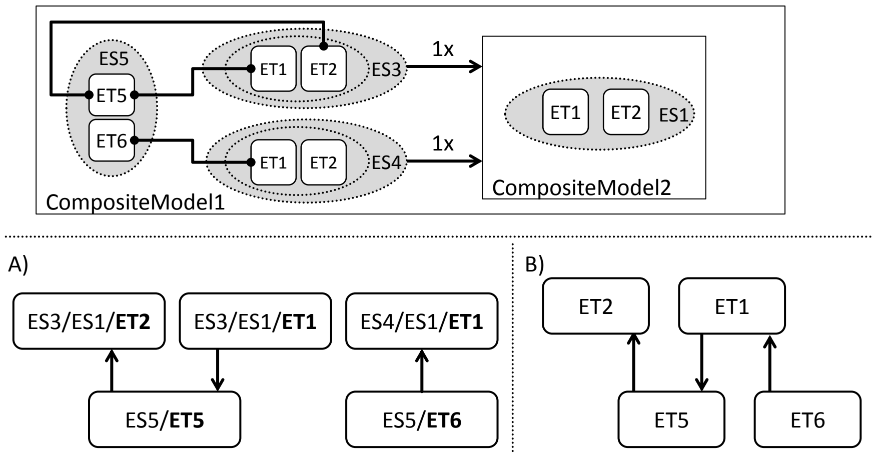


Figure 9.2: Context specific (A) and context independent (B) naming schemes for the nodes of the data flow graph

Figure 9.2 illustrates this approach using a hypothetic example. Part A of the figure shows the resulting DFG for the composition structure depicted in the upper part of the figure. It contains three ConnectionRules connecting the EntityTypes ET5-ET1, ET5-ET2 and ET6-ET1. As a ConnectionRule does not directly reference an EntityType but rather a combination of EntitySet and EntityType (see Figure 8.13), it would be insufficient to only use the name of an EntityType as node identifier as shown in the right part of the figure (B). In the latter case, the same node ET1 is referenced by both, ET5 and ET6, although at run-time there will be distinct entity instances that ET5 and ET6 reference as they reference different EntitySets, namely ES3 and ES4. As discussed in the following, this can lead to an erroneous detection of cycles that do not exist at run-time.

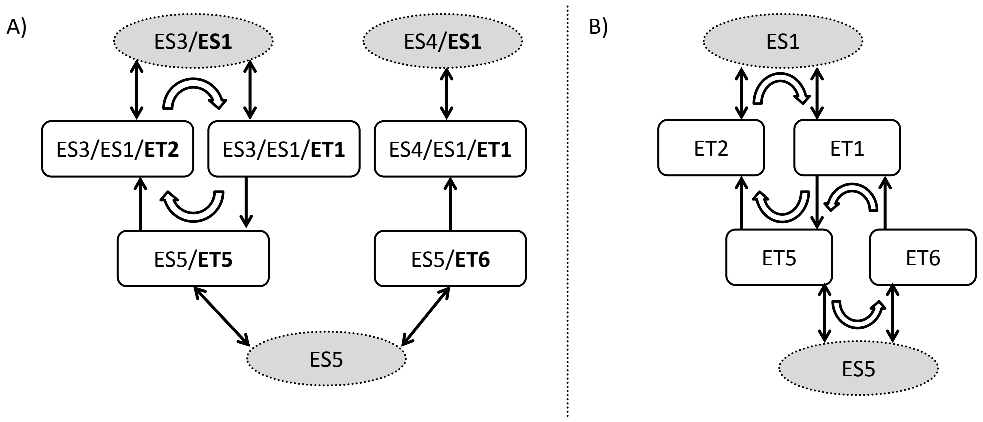


Figure 9.3: Integrating EntitySets as nodes into the data flow graph

To complete the DFG, a second type of nodes has to be added. It accounts for the fact that the entities in an EntitySet can only be stepped together as they are part of the same model instance. In Figure 9.2, for example, ET1 and ET2 will at run-time be instances from the same model, therefore a cyclic dependency between these entities arises. ET2 requires input from ET5 but ET5 can only be stepped after ET1 has provided input. However, ET1 can only be stepped with ET2 which is waiting for ET5 and so on. But the DFG as shown in part A of the figure does not reflect this cycle as it currently has no means to express that two EntityTypes are part of the same model instance. As no model instances exist at this stage, the EntitySet can be used to do so. Figure 9.3 (A) shows how the cycle is detected correctly by adding EntitySet nodes and connecting them to the contained EntityTypes using bidirectional edges. Part B of the figure shows the same extension when not using context specific node names. As discussed above, this leads to a cycle involving ET6 that does not exist due to the different EntitySets referenced by ET5 and ET6 for connecting to ET1.

Algorithm 1 Data flow graph creation

```
1: procedure CreateDFG(c:CompositeModel, id:String, dfg:DirectedGraph)
2:     $id \leftarrow id$+'/'+$c.name$
3:     for $es \in c.physicalToplogy.entitySets$ do
4:         $setId \leftarrow id$+'.'+$es.name$
5:         $paramSet \leftarrow es.instantiableParameterSet$
6:         if $paramSet$ is a $CompositeModelParameterSet$ then
7:             CreateDFG($paramSet.compositeModel, setId, dfg$)          ▷ Recursion
8:         else
9:             Add node $setId$ to $V$
10:            for $et \in paramSet.model.entityTypes$ do
11:                $nodeId \leftarrow setId$+'.'+$et.name$
12:                dfg.addNode($nodeId$)                   ▷ Add node of EntityType
13:                dfg.addBidirectionalEdge($nodeId, setId$)     ▷ Connect to EntitySet
14:            end for
15:        end if
16:    end for
17:    for $cr \in c.physicalTopology.connectionRules$ do
18:        for $source, target \in resultingDataFlows(cr)$ do
19:            dfg.addDirectedEdge($source, target$)  ▷ For CompositeEntityTypes each
   node of their presented EntityTypes is to be connected
20:        end for
21:    end for
22: end procedure
```

The final algorithm for creating the DFG is presented in Algorithm 1. It is a recursive algorithm that is initially invoked with the CompositeModel of the SimulationStudy, an empty DFG and an empty string as initial identifier (id). Initially the *id* is extended to include the name of the CompositeModel, as the names of the contained EntitySets are only unique within a CompositeModel (see OCL constraint **C 51**). As the algorithm operates in a depth-first fashion, all nodes required for the connection of the data flow edges do already exist. The depth-first behavior is achieved simply by recursing into those EntitySets referencing CompositeModelParameterSets first. Further, it has to be noted that the presented algorithm may still detect cycles that do not occur at run-time because the static conditions of the ConnectionRules as well as ID- and RangeSelectors cannot yet be evaluated. But the algorithm does not miss potential cycles and can therefore be classified as conservative cycle detection. The algorithm also works for dynamic conditions (e.g. moving EVs) as it includes all ConnectionRules at the same time. If no cycles arise in this case, no cycles can arise if some of the resulting data flows are deactivated dynamically. For these reasons, the user does not need to intervene once the simulation is started, regardless of the outcome of the actual composition.

9.3 EntitySets and Simulator Processes

Figure 9.4 shows the DFG for another scenario model. Two EntitySets representing a medium and low voltage power grid model have been specified. If the corresponding simulator process allows the execution of more than one model instance (determined by the attribute *maxNumOfModels* of class SimulationModel, see Figure 7.3) and the EntitySets reference the same SimulatorParameterSet, both models could be simulated by the same simulator instance. In this case, however, a data flow between the two EntitySets occurs as the SwingBus (transformer) of the LV grid has been connected to a Bus of the MV grid. The two underlying models must not be simulated by the same simulator instance as otherwise these cannot be stepped sequentially as required by the data flow.

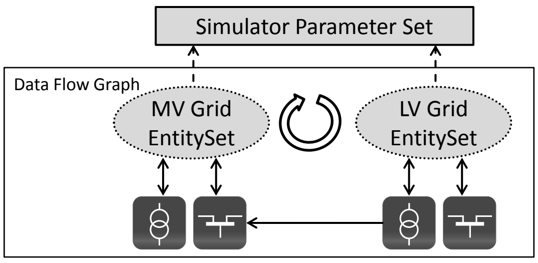


Figure 9.4: Using cycle detection to distribute EntitySets across multiple simulator processes

The DFG can be used for determining those EntitySets that cannot be placed in the same simulator instance. This is achieved for each SimulatorParameterSet by iterating over the two-tuples of the combination[1] of all nodes representing an EntitySet using this SimulatorParameterSet. This can be implemented easily by extending Algorithm 1 to maintain a map with the SimulatorParameterSets as keys and a list of nodes representing the EntitySets that use the respective SimulatorParameterSet as values. Any two non-identical nodes of each list connected via a path in the DFG have to be simulated by different processes.

[1] A combination here is the Cartesian product of all EntitySet nodes for a SimulatorParameterSet excluding symmetrical and reflexive tuples.

9.4 Detecting Cycles in the Data Flow Graph

In Figure 3.4 it was shown that certain scenarios can contain cyclic data flows between involving two or more entities. Requirement [**R 6** – Cyclic Data Flow Support] demands a solution for such cases. This was already considered when creating the scenario metamodel by allowing to specify delayed data flow directions between two EntityTypes as described in Chapter 8.5.7. Now that the DFG has been created, an algorithm is required that checks if it contains cyclic data flows. If so, the SimulationStudy must defined delayed connections between at least two EntityTypes involved in each cycle such that the scheduling algorithm of the simulators (see Section 9.6.4) can determine the simulator that is to be stepped first.

In order to do so, an algorithm is needed that finds elementary circuits (also called simple cycles) in the DFG. A circuit is a path in a graph in which the first and last vertices are identical (closed path). A circuit is said to be elementary if no vertex except for the first and last occurs more than once. Two elementary circuits are distinct if they are not cyclic permutations of each other [Joh75]. For a human it is easy to "detect" such simple cycles in the graph (at least for smaller graphs). For a computer this is not that easy. Johnson [Joh75] presents an algorithm that can detect such cycles and can be applied to directed graphs such as the DFG presented above. Hawick and James [HJ08] present an improved version of Johnson's algorithm that can handle graphs containing edges that start and end at the same vertex as well as vertices connected by more than one edge. However, this is not required for the DFG graph presented here. The implementation of the mosaik concept (see Chapter 11.3) uses a graph library that provides Johnson's algorithm out of the box.

According to Johnson [Joh75], his algorithm is time bounded by $O((v + e) \cdot (c + 1))$ and space bounded by $O(v + e)$ where v and e are the number of vertices and edges and c the number of elementary circuits in the graph. As the DFG construction algorithm is based directly on the scenario model and not on the resulting instantiation, the DFG is small even for large scenarios. Therefore, Johnson's algorithm can be applied without any performance issues. In the evaluation (Chapter 12.4.5) this will be proved.

To obtain proper results by Johnson's algorithm, the bidirectional edges between the nodes representing the EntitySets and the EntiyType nodes have to be removed. These were only required for determining if two EntitySets have to be placed in different simulator processes as discussed above. Otherwise the algorithm would detect a circuit for each of these bidirectional connections. Furthermore, all edges that correspond to the delays specified in the SimulationStudy are being removed. Once these preparatory steps have been done Johnson's algorithm can be applied to the DFG and the composition phase can continue if no elementary circuits are found. As Johnson's algorithm does not only detect elementary circuits but also returns them, these can be logged and the Scenario Expert can use this as a basis to decide what additional delays to specify in the SimulationStudy until all cycles are resolved. By limiting the detection to elementary circuits (instead of finding all circuits) the Scenario Expert is only provided with a minimal set of data flows causing a cycle and can more precisely specify the required delays.

9.5 Scenario Instantiation

The different linear and Cartesian variants to combine the parameter values specified by a SimulationStudy have already been discussed in Chapter 8.5.7. Given a resulting set of parameter values for a CompositeModel, the scenario that is represented by the CompositeModel can be instantiated. Figure 9.5 shows the different steps that are required for scenario instantiation, including the step of creating a DFG as discussed in the last section. The subsequent steps are described in the next sections, starting with the instantiation of the simulators.

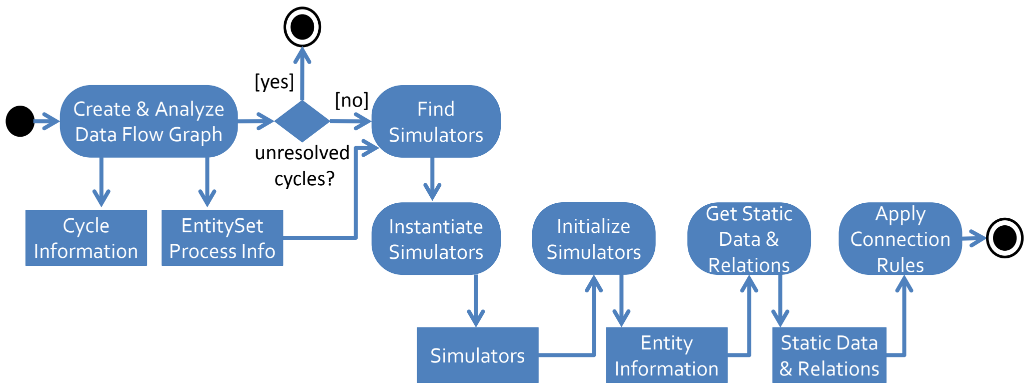


Figure 9.5: Activity diagram for scenario instantiation (scenario model interpretation)

9.5.1 Instantiating the Simulators (creating EntitySetInstances)

Given a resulting set of specific parameter values for a CompositeModel, the next step is to determine the number and type of simulator processes that are required to simulate the overall scenario. As multiple instances of a CompositeModel can occur (when an EntitySet references a CompositeModel rather than a SimulationModel) it is also possible that an EntitySet occurs more than once (i.e. once per instance of the CompositeModel). In the example introduced in Figure 8.4, for example, the EntitySets representing the entities of the LV grid occur as often as the number of LV grid compositions used on the MV level. Therefore, the concept of an *EntitySetInstance* has to be defined. An EntitySetInstance is a specific instance of an EntitySet defined in a scenario model. In the following it is represented as a tuple containing an ID to identify the EntitySetInstance, a set of entity IDs returned by the simulators *init* function and a reference to the EntitySet of which the EntitySetInstance is an instance of:

$$EntitySetInstance = (id, \{eid_1, ...eid_n\}, es \in O_{EntitySet}) \tag{9.1}$$

In the following, $O_{ClassName}$ denotes the set of objects that exist for the given class name. The specific IDs of the entities are obtain by calling the initialization function of the corresponding simulator. As shown in Chapter 6.3.1, it is required to pass the step size, the simulator parameter values as well as a list of model configurations when initializing a simulator. Each element of this list is a tuple containing a unique configuration ID, the name of the model to initialize, the number of instances to initialize

Algorithm 2 Algorithm for determining the required simulator processes

```
procedure FindSimulators(cm:CompositeModel, id:String, context:Map)
    $id \leftarrow id$+'/'+$c.name$ + $createUID()$
    for $es \in cm.physicalTopology.entitySets$ do
        $setID \leftarrow id$+ '.' + $es.name$
        $paramSet = es.instantiableParameterSet$
        $values$ =resolve($paramSet, context$)
        $cardinality$ =resolve($es.cardinality, context$)
        if $paramSet$ is a $CompositeModelParameterSet$ then
            for $i = 0 \rightarrow cardinality$ do
                FindSimulators($paramSet.compositeModel, id, values$)
            end for
        else
            SimulateEntitySet($es, cardinality, id, values$)
        end if
    end for
end procedure

procedure SimulateEntitySet(es:EntitySet, cardinality:Int, id:String, context:Map)
    $esi \leftarrow (id, \{\}, es)$                                    ▷ EntitySetInstance
    $simParamSet \leftarrow es.parameterSet.simulationParameterSet$
    $stillToAssign \leftarrow cardinality$
    $sims_{simParamSet} \leftarrow \{(id, stepSize, sp) \in simulators | sp = simParamSet\}$
    for $simInst \in sims_{simParamSet}$ do
        $assigned \leftarrow$AssignToSimulator($simInst, stillToAssign, esi, context$)
        $stillToAssign \leftarrow stillToAssign - assigned$
    end for
    while $stillToAssign > 0$ do
        $values$ =resolve($simParamSet, context$)
        $simInst = (createUID(), simParamSet.stepSize, values)$
        $simulators \leftarrow simulators \cup simInst$
        $assigned \leftarrow$AssignToSimulator($simInst, stillToAssign, esi, context$)
        $stillToAssign \leftarrow stillToAssign - assigned$
    end while
end procedure
```

and the model parameter values as name-value pairs. In addition to this initialization information, a set of EntitySetInstances to which the returned entity IDs will be assigned is defined:

$$CfgList = \{(id, name, instances, values \in String \times Object, \{esi_n \in O_{EntitySetInstance}\}), ..\} \tag{9.2}$$

To keep track of the simulator processes that are required, a SimulatorInstance is defined as follows:

$$SimInstance = (id, stepSize \in \mathbb{N}^*, sps \in O_{SimulatorParameterSet}, CfgList) \tag{9.3}$$

Algorithm 2 shows an algorithm based on these definitions that can be used to perform the scenario expansion. Similar to the algorithm for creating the data flow graph, the CompositeModels are processed recursively by the procedure *FindSimulators*, starting from the CompositeModel referenced by the SimulationStudy. The given *id* is initially an empty string and the *context* contains key value pairs of the parameters defined in the SimulationStudy. The function *resolve* is not specified in detail. It is used whenever a ValueExpression is encountered and resolves it to specific values in the given context. The procedure *SimulateEntitySet* is invoked for every EntitySet that is referencing a ModelParameterSet (i.e. not referencing a CompositeModel but a SimulationModel). This procedure can be split into two parts. In the upper part involving the for-loop, the procedure tries to assign the model instances of the given EntitySet to existing simulator processes with the same SimulatorParameterSet. How many instances of the model represented by the EntitySet can be handled by a simulator process is determined by the procedure *AssignToSimulator* which is presented below. If there are still instances to assign, the bottom part of the procedure creates data for new simulator instances until all instances could be assigned.

How many instances of a model a simulator can handle depends on three factors that are checked by the procedure *AssignToSimulator* shown in Algorithm 7 in Appendix E:

1. Hwo many model instances does the simulator allow to be simulated (*SimulationModel.maxNumOfModels*)?

2. Does the simulator support models with different configurations (*SimulationModel.mixedParameters*)? If not, do the model instances already assigned to the simulator have the same parameter values as existing model instances?

3. Would a cyclic data flow be created when simulating the EntitySet in this simulator (see Section 9.3)?

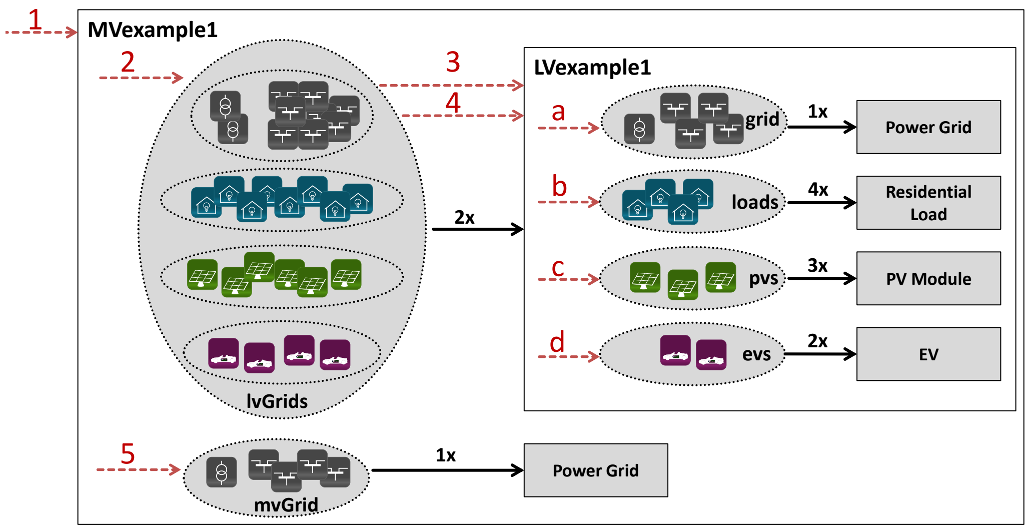


Figure 9.6: Schematic representation of the recursive scenario model expansion

Figure 9.6 shows the order in which the scenario models and their EntitySets are analyzed by the algorithm. The numbered arrows indicate the order in which the recursion processes the scenario model. The EntitySetInstances that are created during this process are as follows (with the subscript being the arrow number in Figure 9.6):

$$\begin{aligned}
EntitySetInstance_{3a} &= (/MVexample1uid1.lvGrids/LVExample1uid1.grid, \{\}, grid)\\
EntitySetInstance_{3b} &= (/MVexample1uid1.lvGrids/LVExample1uid1.loads, \{\}, loads)\\
EntitySetInstance_{3c} &= (/MVexample1uid1.lvGrids/LVExample1uid1.pvs, \{\}, pvs)\\
EntitySetInstance_{3d} &= (/MVexample1uid1.lvGrids/LVExample1uid1.evs, \{\}, evs)\\
EntitySetInstance_{4a} &= (/MVexample1uid1.lvGrids/LVExample1uid2.grid, \{\}, grid)\\
EntitySetInstance_{4b} &= (/MVexample1uid1.lvGrids/LVExample1uid2.loads, \{\}, loads)\\
EntitySetInstance_{4c} &= (/MVexample1uid1.lvGrids/LVExample1uid2.pvs, \{\}, pvs)\\
EntitySetInstance_{4d} &= (/MVexample1uid1.lvGrids/LVExample1uid2.evs, \{\}, evs)\\
EntitySetInstance_{5} &= (/MVexample1uid1.mvGrid, \{\}, grid)\\
EntitySetInstance_{2} &= (/MVexample1uid1.mvGrid, \{\}, lvGrids)
\end{aligned}$$

The lists of entity IDs are still empty at this stage, as the simulator initialization calls have not been made, yet. The simulator instance tuples that are created during this process are as follows:

$$\begin{aligned}
&SimInstance_1 = (sim1, 15, gridSimParams, \{\\
&\quad (cfg1, grid, 1, lvParams, \{EntitySetInstance_{3a}\})\\
&\})\\
&SimInstance_2 = (sim2, 15, loadSimParams, \{\\
&\quad (cfg3, H0, 8, loadParams, \{EntitySetInstance_{3b}, EntitySetInstance_{4b}\})\\
&\})\\
&SimInstance_3 = (sim3, 15, pvSimParams, \{\\
&\quad (cfg4, PV, 6, pvParams, \{EntitySetInstance_{3c}, EntitySetInstance_{4c}\})\\
&\})\\
&SimInstance_4 = (sim4, 15, evSimParams, \{\\
&\quad (cfg5, EV, 4, evParams, \{EntitySetInstance_{3d}, EntitySetInstance_{4d}\})\\
&\})\\
&SimInstance_5 = (sim5, 15, gridSimParams, \{\\
&\quad (cfg2, grid, 1, lvParams, \{EntitySetInstance_{4a}\})\\
&\})\\
&SimInstance_6 = (sim6, 15, gridSimParams, \{\\
&\quad (cfg6, grid, 1, mvParams, \{EntitySetInstance_{5}\})\\
&\})
\end{aligned}$$

9.5.2 Initializing the Simulators (populating EntitySetInstances)

Once the simulator processes have been started, they can be initialized with the required parameters and the models can be initialized. This can be done based on the information gathered by the algorithm described above. As described in Chapter 6.3, the result of calling the *init* function is a data structure that contains the entity IDs and types for each ID representing a model configuration. Based on this, the related EntitySetInstances can be identified and the entity IDs can be added. Afterwards, the static attributes and relations between the entity instances can be retrieved via the SimAPI of each simulator process. Figure 9.7 show an example of how such an assignment may look like. It can be seen that an EntitySet can have entities from more than one model instance (e.g. model instances A1 and A2 or C1 and C2 in this case).

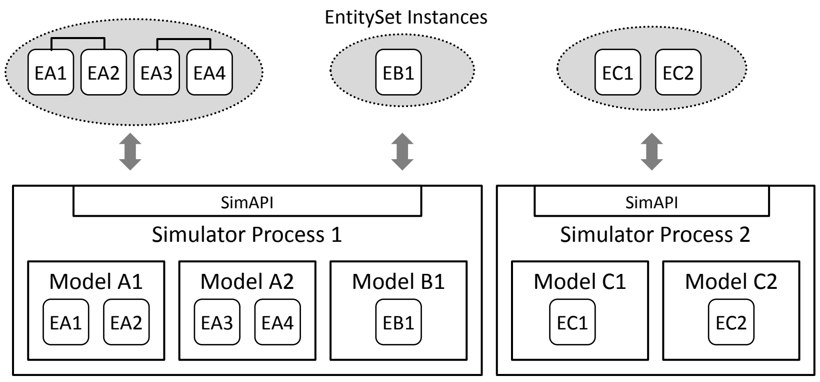


Figure 9.7: EntitySetInstances and their simulator processes

9.5.3 Applying the ConnectionRules

Now that the EntitySets are populated the ConnectionRules can be applied. For processing the ConnectionRules a recursive algorithm is used again, similar to the one used for the DFG creation and scenario instantiation. Therefore it is not mentioned in detail here. By constructing the IDs identically to the ones created in Algorithm 2, the corresponding EntitySetInstances for each ConnectionRule can be identified. It is also important for this algorithm to operate in a depth-first fashion as the ConnectionRules in those CompositeModels that do not contain any other CompositeModels have to be evaluated first. The reason for this is that the ConnectionRules in the higher CompositeModels may specify static conditions that are querying connections established by these ConnectionRules deeper down in the hierarchy.

9.5.3.1 Random Connections

As defined in the scenario metamodel, the ConnectionRules operate on two EntitySubsets. In the following these subsets are called *set*1 and *set*2. Each set therefore only contains entities of a specific type. This can be achieved easily by removing all entity instances from the referenced EntitySetInstances that do not have the EntityType specified by the ConnectionRule. The sets may be reduced further if the ConnectionRule specifies Range- or IDSelectors.

The remaining entities in the two sets now need to be connected on a random basis, but obeying the multiplicity constraint specified by the ConnectionRule. With respect to the combination possibilities of the entities contained in the two sets, different variants have to be distinguished. These are illustrated in Figure 9.8 and depend on the expression structure of a static condition used in a ConnectionRule and if a static condition is defined at all. The entities in the gray shaded ovals are those that can be connected to some other entity. The dashed line indicates the original set before a static condition was applied.

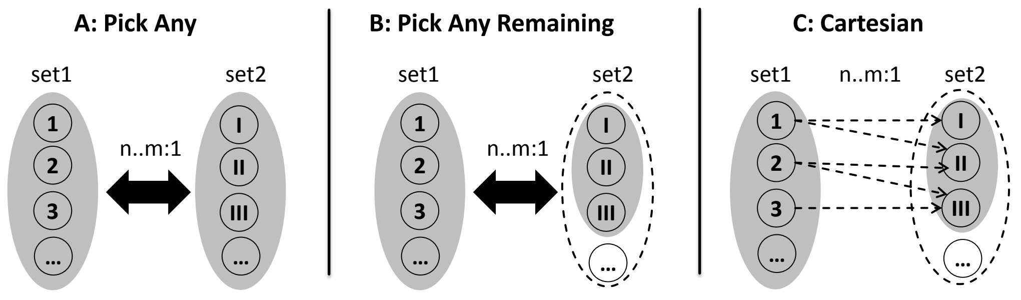


Figure 9.8: Combination variants for two EntitySets being connected by a ConnectionRule

In Variant A (Pick Any) the ConnectionRule does not specify a static condition and thus can operate freely on the two sets. In other words, any two entities can be randomly picked from the sets and connected as long as the multiplicity constraint is met in the end.

In Variant B (Pick Any Remaining), the ConnectionRule contains a static condition. As a consequence, the entities outside the gray oval of *set*2 cannot be picked at all as they do not fulfill the specified condition. In the example shown in Figure 8.22, these entities would be the Bus entities that do not fulfill the condition of having a PV system connected to it. The remaining entities can be connected as in variant A. For both variants (A and B) it can be checked easily if the lower (n) and upper (m) bound of the multiplicity constraint specified by the ConnectionRule can be met:

$$Cn_{PickAny}(set1, set2) : |set1| \geq n \cdot |set2| \tag{9.4}$$

$$Cm_{PickAny}(set1, set2) : |set1| \leq m \cdot |set2| \tag{9.5}$$

The algorithm for selecting the connections has to make sure that the lower bound is met first by assigning *n* entities of *set*1 to each entity of *set*2 and remaining entities can be connected randomly to those entities of *set*2 that have less than *m* connections to entities of *set*1. The formal definition of this algorithm is shown in Appendix E (Algorithm 8). It can be applied for both, variant A and B.

In Variant C (Cartesian) the ConnectionRule contains a static condition that may not only restrict the selectable entities of one set or both sets (here *set*2) but also contains an expression that can only be evaluated when entities from both sets are considered in combination. As a result, only certain entity combinations are valid, indicated by the dashed lines in Figure 9.8. An example for such a condition is the UserDefinedFunction already introduced in the cloud model example (see Figure 8.23). It was used to match the position of PV modules (their connected bus to be precise) with a cell of the cloud model. Obviously in such a case where PV modules and cells are to be connected it is not possible to randomly pick any two entities from the two sets. Each PV module can only be connected to a specific cell. To find out what entities can be connected (before performing the actual connection in a random fashion) the static condition has to be evaluated for all entity combinations (Cartesian product of *set*1 and *set*2). How the lower and upper bound constraints for variant C are defined is shown further below.

9.5.3.2 Expression analysis

The computational complexity for evaluating all these combinations is $O(|set1| \cdot |set2|)$. So in case of equally large sets it is quadratic.[2] If done without any further considerations, the complete static condition is being evaluated for each combination. As a static condition may include both, expressions restricting the elements of one set as well as expressions operating on both sets, techniques from the domain of database systems – from the field of SQL (ISO/IEC 9075) query optimization, to be precise – can be applied to optimize the combination procedure. The reason for this is that Variant C is similar to a database query that performs a join on two tables. A figure with database tables would look pretty similar to Variant C in Figure 9.8 except that the sets would be tables and the entities would be rows of these tables.

Hellerstein [Hel94] provides a good introduction into the terminology and procedures of SQL query optimization (predicate placement). In SQL systems, the *WHERE* clause (which is a boolean expression) is typically converted into conjunctive normal form (CNF). That is, the clause is converted in such a way that it is comprised of a conjunction (AND) of clauses that only consist of disjunctions (OR) or negations (NOT). For example, the clause $\neg(A \vee B)$ is converted into $\neg A \wedge \neg B$. In these CNF clauses, each conjunct is considered to be a separate predicate [Hel94, p. 326]. To optimize the overall clause, these predicates are then classified according to their table references:

Selection Predicates - are referencing a single table and select some of its tuples.

Join Predicates - are referencing multiple tables and specify a join of these tables.

With respect to the expression of a static condition in a ConnectionRule, the selection predicates do only operate on a single EntitySet and the join predicates operate on both sets. It seems to be advantageous to evaluate the selection predicates first, as the number of combinations of the Cartesian product may be reduced dramatically. This approach is called “join-ordering optimization” or “selection pushdown” [Hel94]. The latter term refers to the representation of the *WHERE* clause as a tree structure where the selection predicates are “pushed down” to the bottom as far as possible. In traditional database systems this approach can “frequently save orders of magnitude of time” [Ull88, p.65].

[2] The evaluation in Chapter 12 discusses this issue for actual research projects. For the considered projects the combinatoric complexity is not an issue.

However, as Chaudhuri and Shim [CS99] point out, this approach is not sufficient when a database system allows to define user-defined functions in a query expression. The costs (i.e. time and memory) for evaluating such predicates can be much more expensive. Especially, the costs are not known to the database system a-priory (in contrast to the built-in SQL predicates) as these functions can include more or less arbitrary source code (e.g. invoke a web-service). The same is true for the UserDefinedFunctions that can be used in the static conditions of a ConnectionRule in a mosaik scenario model. However, in case of mosaik, the scenarios analyzed in Chapter 3.2 can be defined with static conditions that only use a single selection or join predicate. Therefore, optimization approaches for predicate placement (i.e. to dynamically evaluate the costs of predicates and place them correspondingly) are not within the scope of this thesis. Mosaik will stick with the approach of evaluating selection predicates first. However, an easy to implement conclusion that can be drawn from this discussion is that selection predicates with UserDefinedFunctions may be more expensive than the built-in functions. Therefore, selection predicates without such functions should be evaluated first. The following enumeration presents the proposed order in which predicates should be evaluated by an implementation of the composition layer (from least expensive to most expensive):

1. Selection: Static attribute compared to constant/variable
2. Selection: Single topological condition (counting related entities)
3. Selection: UserDefinedFunction invoked with static data from single EntitySet
4. Join: Comparison of static data from two EntitySets
5. Join: Comparing two topological conditions (counting related entities) that are originating from different EntitySets
6. Join: UserDefinedFunction with two or more inputs from different EntitySets

Finally it has to be pointed out that a conversion of the initial condition into CNF is not always beneficial. An example given by Vorwerk and Paulley [VP02] shows that a simple expression such as $(a \land b) \lor (c \land d)$ where each letter represents a predicate, can result in the much more complex CNF expression $(c \lor a) \land (c \lor b) \land (d \lor a) \land (d \lor b)$. If the resulting conjunct does not lead to improvements of the query processing strategy, the converted expression is likely to be more expensive to evaluate (time and/or memory consumption) [VP02]. However, at least for the domain of database systems, "the conjunctive normal form is more practical as query qualifications usually include more AND than OR predicates [TV11, p.223]". As mentioned above, the scenarios analyzed in Chapter 3.2 can be defined with static conditions that only use a single selection or join predicate and are thus very simple. Therefore, expression optimization such as described by Vorwerk and Paulley [VP02] is not within the scope of this thesis. Mosaik will split the given expression into different AND conjuncts as it is. Thus, if it does not contain AND operators on the root level, the whole expression is treated as a single predicate. If problematic situations are identified in the future, the user may rearrange the static condition manually or a corresponding extension offering expression optimization may be added.

9.5.3.3 A Constraint Satisfaction Problem

In case of the existence of a join predicate another issue may arise even after the possible candidates for composition have been identified. This problem is depicted in Figure 9.9 with dashed lines showing possible connections. Entities 1 and 2 were connected to II and III, respectively. However, entity 3 cannot be connected (only a connection to III is possible) without violating the 1:1 multiplicity constraint although another combination (i.e. 1-to-I, 2-to-II, 3-to-III) would be possible. But this may only occur in rare cases.

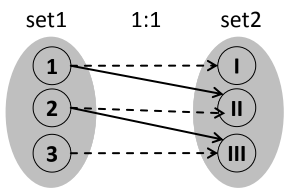


Figure 9.9: Constrained EntitySets that may not be connected in a purely random fashion

Luckily, due to limiting the ConnectionRule multiplicity to n..m:1 relations [**A 6** – It is sufficient to specify n:1 relations to capture Smart Grid Scenarios], the entities of one EntitySet are only assigned to 1 other entity. Therefore, this problem can be formulated and solved as classical discrete, finite domain Constraint Satisfaction Problem (CSP). Formally, a CSP is defined by a set of variables $V = \{X_1, ..., X_n\}$, a non-empty domain $D_i = \{v_1, ...v_k\}$ of possible values for each variable X_i and a set of constraints $C = \{C_1, C_2, ..., C_m\}$ [RN03]. A possible solution for the CSP is any variable-value assignment $A : V \rightarrow \bigcup_i D_i$ that does not violate any of the constraints. In case of a ConnectionRule, the entities of *set*1 can be treated as variables. For each variable, those entities of *set*2 that fulfill the static condition of the ConnectionRule constitute the domain of possible values. Furthermore there is a single global constraint (a constraint involving all variables) C_{Mult} that is true, if the multiplicity lower and upper bound is met when considering all variable assignments. A standard CSP solver can now be used to obtain valid connections. By shuffling the EntitySets prior to passing them to the CSP solver, a random solution can be achieved.

$$V = \{l_1, ...l_n | l_i \in set1\} \tag{9.6}$$

$$D_i = \{r_1, ...r_n | r_j \in set2; cond_{stat}(l_i, r_j) = True\} \tag{9.7}$$

$$C_{mult} : n \leq |\{l | l \in A^{-1}(r)\}| \leq m \quad \forall\, r \in \bigcup_i D_i \tag{9.8}$$

9.5.3.4 Connection algorithm

As solving the CSP can also be time intensive, it is advisable to identify the situations in which no CSP occurs. Obviously, these are the situations in which the lower and upper bound can be met in any case, regardless of the random selection. For the upper bound, this is the case if the number of possible candidates for all entities $e \in set2$ is less or equal the upper bound (m):

$$Cm_{Cartesian}(set1, set2) : \forall e \in set2 : |possibleCandidates(e)| \leq m \tag{9.9}$$

For the lower bound this check is slightly more complex. It can only be made sure that the lower bound (n) is met if for all entities $e_2 \in set2$ there are at least n elements in the set of possible candidates for e_2 that can only connect to e_2:

$$\begin{aligned}&Cn_{Cartesian}(set1, set2) : \forall e_2 \in set2 :\\ &|\{e_1 \in possibleCandidates(e_2) : |possibleCandidates(e_1)| = 1\}| \geq n\end{aligned} \tag{9.10}$$

The connection of PV modules to the cloud model presented in Figure 8.23 is an example where these checks can be used to avoid the usage of a CSP solver, as a cell can be connected to any number of PV modules (0..*). Thus, both constraints (9.9 and 9.10) are met. Checking these constraints and using a simple random selection is faster and especially performs linear with increasing set size than always using the CSP solver. This is shown in Figure 9.10, where an implementation of both variants has been benchmarked for different set sizes. Therefore, these checks are to be considered in the connection algorithm to avoid unnecessary performance reduction by the CSP solver for such cases.

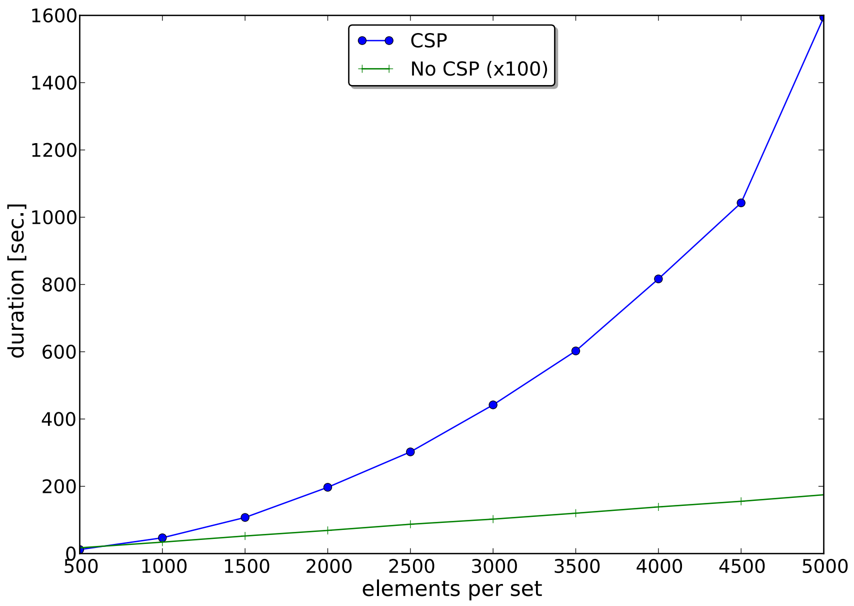


Figure 9.10: Runtime for CSP solving versus checking if no CSP solving is required

The final algorithm for connecting the elements of two EntitySets is depicted in Figure 9.11 and works as follows. If the static condition of the ConnectionRule contains Selection Predicates (as discussed above), these are applied first to reduce the set sizes. If no Join Predicates are specified, it can easily by checked if the lower and upper bounds of the multiplicity constraint can be met ($Cn_{PickAny}$, $Cm_{PickAny}$) and any two entities can be

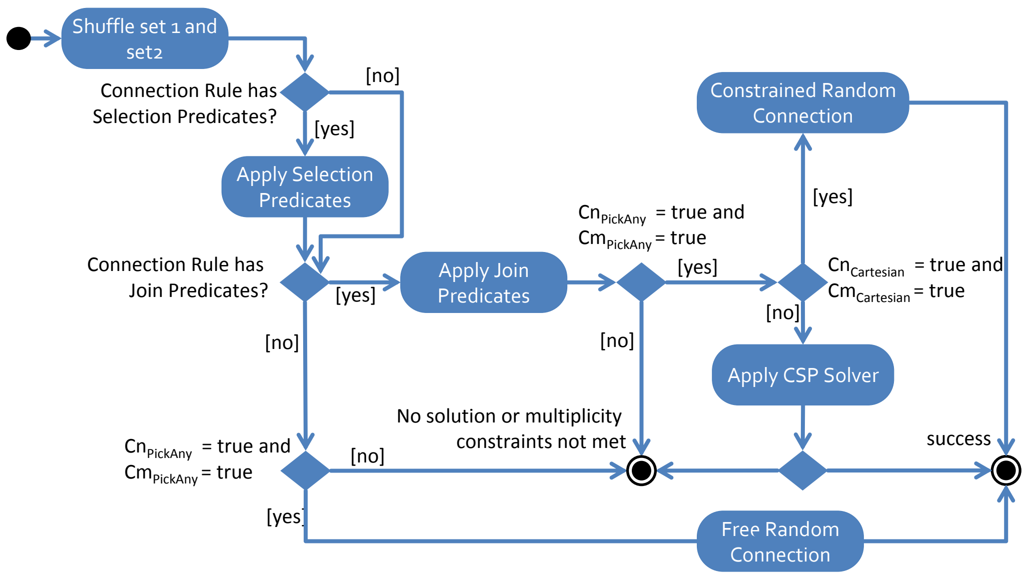

Shuffle set 1 and set2
Connection Rule has Selection Predicates?
[no]
[yes]
Apply Selection Predicates
Connection Rule has Join Predicates?
[yes]
Apply Join Predicates
[no]
$Cn_{PickAny}$ = true and $Cm_{PickAny}$ = true
[yes]
[no]
Constrained Random Connection
[yes]
$Cn_{Cartesian}$ = true and $Cm_{Cartesian}$ = true
[no]
Apply CSP Solver
No solution or multiplicity constraints not met
success
$Cn_{PickAny}$ = true and $Cm_{PickAny}$ = true
[no]
[yes]
Free Random Connection

Figure 9.11: Flow chart for connecting EntitySets

randomly selected and connected (variant A shown in Figure 9.8). If Join Predicates are specified, these are applied, and the constraints from the *PickAny* variant can be applied to allow an early exit in cases where not enough entities are remaining. However, as these constraints do not consider the actual combinations that the Join Predicates allow, meeting these constraints is not a sufficient condition to guarantee that the lower bound constraints can actually be met. Next, the constraints $Cn_{Cartesian}$ and $Cm_{Cartesian}$ are applied and if successfully passed, the *Constrained Random Connection* algorithm can be applied and the CSP solving overhead can be avoided. This algorithm is defined in Appendix E (Algorithm 9). Otherwise, a CSP solver has to be used to solve the situation as defined in Equation 9.6, above.

It has to be pointed out that the constraints $Cn_{Cartesian}$ and $Cm_{Cartesian}$ can only detect some special cases that do not require a formulation as CSP. However, they cannot detect cases that need a CSP. Thus, the presented algorithm may still apply the CSP solver to situations that do not require it. Such a situation is shown in Figure 9.12. An algorithm allowing to detect exactly those situations where CSP solving is the only possible option is subject to future work. However, the defined algorithm is sufficient for the artificial examples as well as the real-world case studies used for the evaluation in Chapter 12. No CSP case occurs in these scenario.

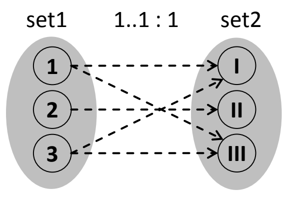


Figure 9.12: Two EntitySets for which the constraints $Cn/m_{Cartesian}$ do not hold but still no CSP arises

9.5.3.5 Domain-Specific Constraints

Domain-specific constraints (see Chapter 7.7) may be defined for one or more data flows of the entity types that are to be connected. However, for two reasons the evaluation of these constraints should not be part of the connection procedure discussed above. First, the fact that domain specific-constraints involve static attributes from both entities would result in a Cartesian combination of the two EntitySets even if no join predicate was used in the static condition. This would mean a large performance loss for the composition of most scenarios. For example, a basic scenario involving 1000 nodes of power grid and as many consumer entities would mean to check one million combinations. Second, and even more important, the chosen approach allows more flexibility. A user can still specify a static condition that involves the same static attributes as the domain-specific constraint to use these as selection criterion, for example, to connect a load only to those nodes that have the same voltage level. But as this is not the default behavior the composition process is usually accelerated and checking the constraints afterwards still prevents the simulation of invalid scenarios. This approach assumes that the Scenario Expert knows what he or she is doing and the constraints are only used as additional safety measure and not for defining the scenario itself.

9.5.3.6 Connection Limits

The same handling is applied for the *maxConnections* attribute of the Port class. If the algorithm for applying a ConnectionRule encounters an entity with a limited Port and the limit is already reached, the composition will be canceled and a corresponding error message should be raised such that the user can identify the ConnectionRule that causes the limit violation.

9.5.3.7 Establishing Data Flows

Once the entities that have to be composed are chosen, and all checks are passed, the data flows between the entities can be established. The algorithm is similar to the one of OCL check **C 57** but instead of returning a boolean result indicating the composability of the two entity types the algorithm returns the names of the data flows (PortData objects) of both entity instances. The *name* attribute of these objects can be used to parametrize the get and set-data calls of the SimAPI to actually perform the data flow.

9.5.3.8 Completeness

Finally, after the data flows have been established, it is the task of the composition layer to check if the composition is complete. For a composition to be complete, every continuous mandatory input of all mandatory ports of the entities has to be connected to at least one other entity providing the required data (derived from the definition of completeness in Röhl [RÖ8]). While this check sounds straight forward, the problem is that mandatory inputs that are not connected to other entities outputs may still be provided with data at run-time by a control mechanism. As discussed below, control strategies may not be able to interact with all models at every time step if the models are simulated in different step sizes. Therefore, the completeness check has to be implemented in such a way that it incorporates the synchronization step size Δt_{sync} (see next section) of the control mechanisms.

Definition 14 (Completeness) *The composition is considered complete, if all continuous mandatory inputs of all mandatory ports are connected or the step size of all entities with an unconnected continuous and mandatory input of a mandatory port is similar to the synchronization step size of the control mechanisms.*

9.6 Scheduling the Simulators

Because of data flow dependencies between the entities of the different simulator processes it is crucial to find the right sequence for executing the step function of the simulator processes [ZPK00]. In this section an algorithm for creating a schedule graph is presented. The resulting graph can then be processed rapidly node by node without the need for any further scheduling operations. Before the scheduling algorithm is presented, the relevant requirements are recapitulated and related works dealing with the coordinated execution of coupled simulators will be discussed.

9.6.1 Requirements

R 19 – Different Temporal Resolutions The scheduling algorithm has to account for the fact that the simulators may advance their simulation time in different increments. For example, a scenario contains EV entities that can operate at a step size of 1 minute while residential loads may only be stepped in 15 minute intervals (e.g. based on load curves). In other words, invoking the *step()* method of a simulator's API can cause different advances in simulation time of the respective simulator. The exact amount of time that is simulated by this method has been defined in the SimulatorParameterSet during scenario specification.

R 6 – Cyclic Data Flow Support As shown in the requirements chapter there may be scenarios that lead to cyclic data flows between two or more entities (see Figure 3.4). The scheduling algorithm that determines the simulator execution order has to be able to cope with such cyclic flows. As already mentioned in Chapter 8.5.7, the scenario modeler can specify a delay for certain data flows of the cycles so that these can be resolved (see Section9.4).

[R 12 – Synchronization of Control Mechanisms] Mosaik focuses on the integration of agent-based control mechanisms. Frameworks and platforms for developing such MAS usually do not account for simulation specific issues. In particular they provide no means for synchronizing the agents with the simulation time which does not progress implicitly as it would be the case in non-simulated, real-world control tasks [SGW08]. This will be the task of the control layer presented in the next chapter. However, the scheduling algorithm has to account for this synchronization task such that the control layer can forward this information to the control mechanism.

9.6.2 Related Work

Clauß et al. [CASB12] present a so called Master program for coupling simulators via the FMI interface. However, the execution order of these simulators has to be specified manually. They state that an automatic evaluation of the dependencies is subject to future work.

Fishwick [Fis95] presents a "Time-Slicing" algorithm for the simulation of function-based models, which solves the simulation ordering issue by creating a cyclic graph that determines the order in which the blocks are calculated. However, the approach does not consider simulation time. It is assumed that all blocks are timeless as their outputs only depend on the given input and not on simulation time or previous states (see memoryless systems, Chapter 2.1.4). Although they may require some time for computation this is not related to the simulator step sizes mosaik has to account for.

Zeigler et al. [ZPK00] present an algorithm for coordinating networks of discrete-time simulators. Other than the FMI-Master presented by Clauß et al. [CASB12], the algorithm automatically determines the execution order. Also, compared to Fishwick [Fis95], the algorithm explicitly considers simulation time (and not only memoryless systems). However, it is only applicable if all participating simulators have the same step size. Therefore this algorithm is not directly applicable to mosaik due to requirement [**R 19** – Different Temporal Resolutions].

Nutaro [Nut11b] presents two algorithms for managing networks of discrete-time components. The first operates in a top-down fashion on a hierarchy of composed simulation models where only the leafs of the hierarchy are atomic models. The second algorithm flattens the network in a sense that it only operates on the atomic models and uses the hierarchy only to find out what communication connections exist between the atomic models. This approach seems to be applicable to the given scheduling problem, as the hierarchy in the mosaik scenario models only exists at scenario design-time. The result of the scenario interpretation presented above is a flat list of simulation models and data flow connections among these. Although the relations may correspond to the voltage hierarchy of the analyzed power grid, the simulators itself are atomic. Similar to the approaches discussed above there is no explicit consideration for components with different step sizes.

Schmitz et al. [SKdJdK11] propose an approach that can handle discrete-time simulators with different step-sizes. The presented approach also considers aggregation of data flows. There is a central execution component which maintains a shared time for all participating simulators and advances the simulators in a step-by-step fashion according to a previously generated schedule. Schmitz et al. [SKdJdK11] point out that this approach offers more flexibility then the pull-based approach used in OpenMI (see Chapter 6.2.6), for example concurrent execution of the participating simulators. This presented approach matches well with the requirements of the composition layer. However, as Schmitz et al. [SKdJdK11] do not present a formal definition of their scheduling algorithm, this is done in Section 9.6.4, below.

9.6.3 Domain-specific scheduling aspects

Although not yet discussed explicitly, the above approaches are assuming Moore-type models (see Chapter 2.1.4).[3] This means that they "compute their output solely on the state information and therefore impose a delay on the effects of the input on the output" [ZPK00, p.164]. This is because a change in the output is not directly caused by a changed input but rather by a state change. For discrete-time systems it is assumed that such state changes consume simulation time (not physical time) and thus the output cannot be provided at the same instant. For discrete-event systems, the state transitions can usually have a delay of zero [Nut11b].

Besides the fact that a zero-delay state transition may be unrealistic in many cases, the reason for using Moore-type systems (and not the more general Mealy-types) can be found in the handling of cyclic data flows. As Mealy-type systems can only produce an output when an input is given, a deadlock occurs in case of cyclic dependencies, whereas in Moore-type systems the outputs can be provided any time just from the current states. This makes scheduling easier. Algorithm 3 shows the most important steps of the Moore-type simulation algorithm proposed by Nutaro [Nut11b].

Algorithm 3 Discrete-time Moore-type simulation algorithm (based on [Nut11b])

```
1:  for $t = startTime \rightarrow stopTime$ do
2:      1. Clear inputs of all components
3:      2. Calculate outputs:
4:      for all $c \in Components$ do
5:          $y_t = \lambda_t(s_t)$
6:      end for
7:      3. Transfer outputs to connected inputs
8:      4. Update states:
9:      for all $c \in Components$ do
10:         $s_{t+1} = \delta_t(s_t)$                ▷ $\delta$ is the state transition function
11:     end for
12: end for
```

Figure 9.13 shows the resulting state trajectories for three (partially cyclic) interconnected simulation models that are advanced according to Algorithm 3. Identical numbers indicate that these steps can be executed in parallel (although the algorithm shows them in sequential manner). In step 1 the outputs of the models are generated from their current states. Next in step 2, the state transitions can be calculated and the states for the next time instant are obtained. As a consequence, a change in a state of the upper model can only propagate slowly through the chain of connected models, as state changes are assumed to take time and outputs can only change if states change. For illustration purposes it is assumed that the state simply reflects the received input value and the output is identical to the current state.

As the type of studies that are within the focus of mosaik (stationary power flow analysis or simple energy balancing analysis) are usually 1 or 15 minutes, such an

[3] For the approach presented by Schmitz et al. [SKdJdK11] the assumed model type cannot be identified clearly.

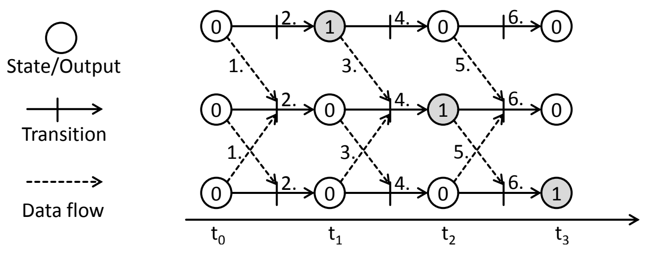


Figure 9.13: Exemplary data flow for delayed scheduling algorithm (Moore-type)

approach is likely to induce large errors. For example, when considering the PV/cloud model scenario shown in Figure 8.25. Using a resolution of 15 minutes for all simulators, a change in the solar irradiance would take 30 minutes to propagate through the models: 15 minutes till the PV output changes and another 15 minutes until the power system model updates its output. The real world components (e.g. the PV panel and its inverter), however, may only take a few milliseconds to react. As Zeigler et al. [ZPK00] point out "in Mealy networks the effects of an input can propagate through space in zero time [...]". The mosaik scheduling approach therefore interprets the connected simulation models in a Mealy-style (same state + new input = new output). This is achieved by not transferring the outputs of all models in parallel and subsequently stepping all models (as done by Algorithm 3), but by incrementally transferring outputs and stepping the receiving models only. The next section presents the algorithm in detail. Figure 9.14 shows the resulting output behavior for the PV/cloud model scenario. It has to be noted that the circles in this figure do not represent states but the outputs only. The states are not shown to improve understandability of the figure. In case of a PV model the state may be the time of the day, for example, which also influences the output[4], but by handling the models in a Mealy-style, the output can directly be influenced by the solar irradiance provided by the model, for example. For power flow models, which can be characterized as a memoryless (function-specified) system where the output directly depends on the input ($\lambda : X \rightarrow Y$, see Chapter 2.1.4), direct response to the provided inputs represents the natural model behavior.

It has to be noted that the chosen Mealy-style approach only influences the order in which scheduling algorithm processes outputs and steps the simulators. The models integrated via the SimAPI do not have to be Mealy models internally. A discrete-event model, for example, can schedule an undelayed event as response to an input. In this case, the output changes in a Moore fashion, but from the outside (via the SimAPI) this is seen as an immediate (Mealy-style) response to a change of the input.

Cyclic Mealy-style coupling

As already discussed in the beginning of this section, trouble comes when connecting Mealy-type components in a cyclic fashion [Nut11b, p.90], [ZPK00, p.46]. As the cycles have no delays a deadlock arises. However, in Chapter 8.5.7 a mechanism for resolving this issue has been foreseen: the Scenario Expert can specify delayed data

[4] Remember that the output function of Mealy-Type components ($\lambda : S \times X \rightarrow Y$) depends on the input X and the state S (see Chapter 2.1.4).

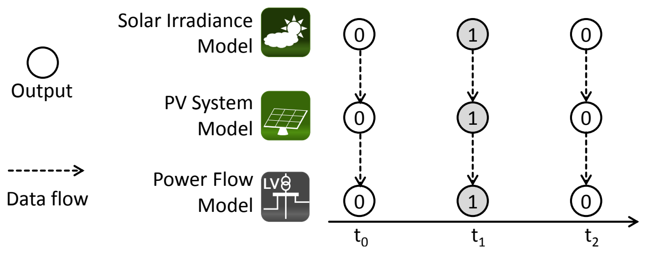


Figure 9.14: Exemplary data flow for non-delayed scheduling algorithm (Mealy-type)

flows between any two EntityTypes, thus resolving the deadlock. As already mentioned above (Chapter 2.1.4), Mealy-type components are the more general type of components. The same holds for the mosaik scheduling approach. By specifying delays for all connections, an identical scheduling as with Algorithm 3 can be obtained.

9.6.4 Mosaik Scheduling Algorithm

The scheduling algorithm is based on a directed graph. The nodes represent the different simulator processes at different points in simulation time. The edges represent dependencies between the nodes. In the following this schedule graph is called SG and defined as follows:

Definition 15 (Schedule Graph) *$SG = (V, E)$ with V being a set of vertices (nodes) and E a set of directed edges connecting two vertices each.*

The SG is constructed after the ConnectionRules have been processed and the corresponding data flows have been established. For executing the simulation the graph is simply traversed from the beginning (root node) to the end repeatedly until the final simulation time (specified by the simulation study) is reached. The simulator process that each node represents can only be advanced one step if all predecessor processes (nodes providing incoming edges to the node) have been processed. As a consequence, the algorithm for constructing the schedule has to be executed only once and at simulation-time the graph can be traversed rapidly. It is only required to keep track of what simulator processes (nodes) have already been processed.

Before the algorithm for creating the SG will be presented in detail, an important limitation has to be presented. It is assumed that the step sizes of any two participating simulators are integer multiples (**A 7** – *Any two simulator step sizes are integer multiples*). If there are three simulator processes involved in a simulation, for example, the simulator step sizes (2, 4, 8) are valid, whereas step sizes of (2, 4, 6) are not valid as 6 is not a whole multiple of 4. There are a number of reasons for making this limitation:

1. The computability of aggregation functions is much more difficult if arbitrary simulator step sizes occur.
2. For step sizes that have a large common multiple, the SG can become very large until the repetition point is reached. As a result, the schedule is hard to debug in cases of errors or for the validation of the simulation run.

3. Common step sizes in the domain are 1 minute or 15 minutes so that this is no actual limitation in most cases.

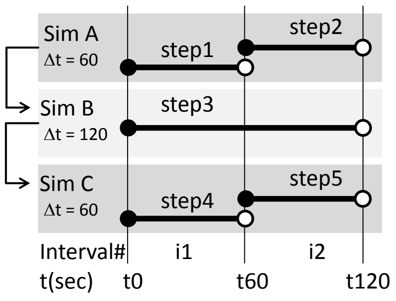


Figure 9.15: Time semantic of a simulator step

The semantic of calling the *step* function of the SimAPI has been defined clearly in Definition 7. Figure 9.15 depicts the step behavior according to this definition using an example of three simulator processes SimA, SimB and SimC with different step sizes. The example assumes that SimA provides data that is required by SimB and SimB provides data for SimC. The order in which the simulators have to be stepped is indicated in the figure by enumerating the steps from *step1* to *step5*. Step3 can only be executed after data for both intervals (i1 and i2) has been made available by SimA. The reason for this is the data can then be aggregated as input for SimB according to an aggregation function potentially specified by the corresponding PortData object. When implementing this behavior it has to be made sure that corresponding buffers are used to hold previous values as input for the aggregation. Finally, SimB has to be stepped to provide input data for the two steps of SimC.

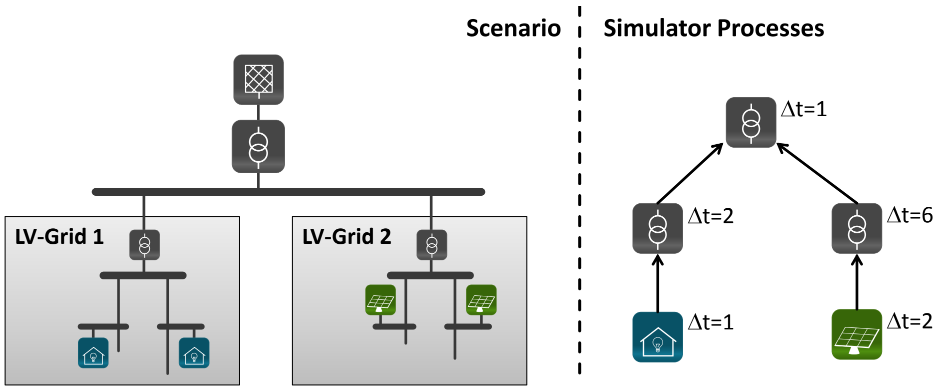


Figure 9.16: Simplified example scenario for illustrating the scheduling algorithm

To illustrate the algorithm for creating an SG that operates in this way, the example scenario presented in Figure 8.6 is used in a slightly simplified version. To reduce the complexity of the graph for illustration purposes, the LV grids of the scenario do either contain residential loads only or PV systems only. This setting is shown in Figure 9.16. The actual number of loads or PV systems is not important, as the corresponding simulator processes are capable of simulating an arbitrary number of these entities and thus the nodes of the SG remain unchanged. For better illustration, different step sizes for the simulators are assumed. Of course, a scenario with such heterogeneous step sizes will probably not occur in a realistic simulation scenario.

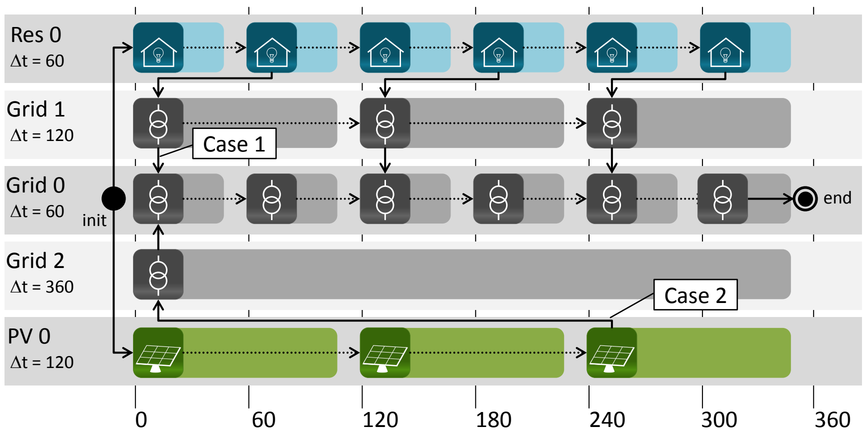


Figure 9.17: Sample schedule

The corresponding SG is shown in Figure 9.17. Each symbol represents a node of the SG, which again represents a simulator process at a certain point in time. The schedule starts at the initial *init* node and the simulator processes *Res0* and *PV0* can be stepped as these have no input dependencies. They provide data for the interval [0, 60[and [0, 120[respectively. Depending on the maximum step size $step_{max}$ (equation 9.11) of all simulator processes, the schedule repeats at a certain period of time. This point is marked by the *end* node. If the end node is reached and the stop time of the simulation is not yet reached, the schedule is repeated again. Of course, the simulator time is not reset. Therefore, the time scale shown in the bottom of the figure only denotes the relative time elapsed since simulation start for the first cycle of the schedule.

$$step_{max} = max(s.stepSize : s \in simulators) \tag{9.11}$$

Listing 4 shows the complete algorithm for schedule graph creation. The algorithm starts by creating a node sim_t^s for each simulator process s and time step t until the schedule repeats. Next, for each created node, directed edges from the preceding nodes are created. For being able to maintain the time/state information of each simulator process, this also includes an edge to the preceding node of the same simulator process (Line 10). The algorithm for connecting to the predecessors of other simulators is shown below in Algorithm 5. Once these edges have been added, the *init* node marking the start of the schedule is integrated by adding directed edges to all root nodes of the SG. The roots are those nodes that have no incoming edge yet. Next, special synchronization nodes are added that represent the time points at which a control mechanism is allowed to interact with the simulated entities (i.e. read their state and send control commands) as all simulators have reached a consistent time point. Details of this procedure are given in Section 9.6.5, Algorithm 6. Finally, the *end* node, marking the end of the schedule graph is added and connected to the leaf nodes. These are the nodes that have no outgoing edge yet.

Algorithm 4 Algorithm for simulator scheduling

```
 1: for all $s \in simulators$ do                      ▷ Create a node for each simulator and step
 2:     for $t = 0 \rightarrow step_{max}$ step s.stepSize do
 3:         Add node $sim^s_t$ to V
 4:     end for
 5: end for
 6:
 7: for all $sim^s_t \in V$ do                          ▷ Connect to predecessors of same simulator
 8:     if $t = 0$ then                                 ▷ First node has no predecessor
 9:         $t' = t - s.stepSize$
10:         Add directed edge $sim^s_{t'} \rightarrow sim^s_t$ to E
11:     end if
12:     connectToPredecessors($sim^s_t$)                 ▷ See Algorithm 5
13: end for
14:
15: Add node $init$ to V                                ▷ Create and connect start node
16: for all $root \in rootNodes(SG)$ do
17:     if $root! = init$ then
18:         Add directed edge $init \rightarrow root$ to E
19:     end if
20: end for
21:
22: addSyncNodes($SG$)                                  ▷ See Algorithm 6 in Section 9.6.5
23:
24: Add node $end$ to V                                 ▷ Connect end node
25: for all $leaf \in leafNodes(SG)$ do
26:     if $leaf! = end$ then
27:         Add directed edge $leaf \rightarrow end$ to E
28:     end if
29: end for
```

Algorithm 5 shows the procedure to connect a node to predecessors of other simulator processes. The algorithm can be split into four different cases depending on the step sizes of the simulators and whether a delay is specified for a data flow from a predecessor or not. Two of these cases (Case 1 and Case 2) deal with non-delayed data flows and are highlighted in Figure 9.17. The other two cases dealing with delayed data flows are shown in Figure 9.18 with the dashed edges being dependencies caused by delayed flows. In Case 3 (A) the predecessor to which a delayed flow exists has the same step size. In Case 3 (B), the predecessor steps slower and in Case 4 the predecessor steps faster. THe algorithm connects the predecessors in such a way that the preceding node is the last available step the predecessor made before the time at which the successor requires the input. It has to be noted that Case 3 and 4 only add the dashed, delayed edges. The solid edges have already been added before by Case 1 or 2 (see Algorithm 5). This algorithm also works for cycles that involve more than two simulator processes.

Algorithm 5 Procedure for connecting nodes to predecessors

```
procedure CONNECTTOPREDECESSORS($sim_t^s$)
    for all $pre \in predecessors(s)$ do
        if $s.stepSize \leq pre.stepSize$ then                    ▷ Case 1
            if $t\%pre.stepSize = 0$ then
                Add directed edge $sim^pre_t \rightarrow sim_t^s$ to E
            end if
        else                                    ▷ Case 2: Predecessor steps faster
            $t_{pre} = t + s.stepSize - pre.stepSize$
            Add directed edge $sim_{t_{pre}}^{pre} \rightarrow sim_t^s$ to E
        end if
    end for

    for all $pre \in predecessorsWithDelay(s)$ do
        if $s.stepSize \leq pre.stepSize$ then          ▷ Case 3: Delayed predecessor
            if $t\%pre.stepSize = 0$ then
                $t_{pre} = t - pre.stepSize$
            end if
        else                           ▷ Case 4: Delayed predecessor steps faster
            $t_{pre} = t - pre.stepSize$
        end if
        if $t_{pre} \geq 0$ then
            Add directed edge $sim_{t_{pre}}^{pre} \rightarrow sim_t^s$ to E
        end if
    end for
end procedure
```

9.6.5 Control Strategy Scheduling

So far, the algorithm provides a valid schedule for the different simulators. To meet requirement [**R 12** – Synchronization of Control Mechanisms], it needs to be extended in such a way that:

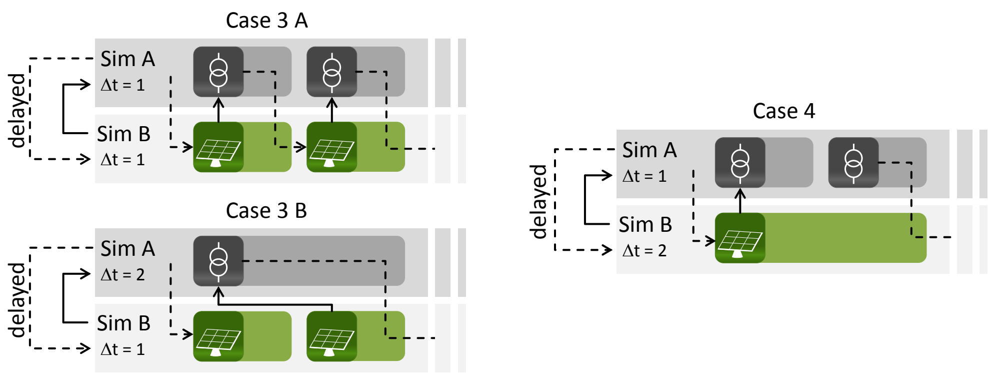


Figure 9.18: Scheduling of simulators with delayed data flows

- Simulators existing in parallel branches of the SG do not advance independently but are synchronized (stopped) at those points in simulation time where control strategies can interact with the simulated entities.
- The simulator processes must not advance in time unless the control strategy has finished interaction as otherwise control commands may be based on outdated state information or arrive at the simulators at different points of simulated time.

The presented algorithm allows to account for these subrequirements in a simple and straight forward way: First, additional synchronization nodes (sync nodes) are added to the SG, representing those points in simulation time at which the interaction with the control strategies can take place. Second, for each sync node, the dependencies from the preceding simulator nodes and to their successor nodes are added as directed edges. This way it is ensured that the sync nodes are only executed when all preceding simulators have finished and that these simulators only advance after the interaction has taken place. For this process, only those simulator processes are considered that contain at least one entity of an EntitySet that is declared "public" by the *publicSets* association of the SimulationStudy (see Chapter 8.5.7). As shown in Figure 9.19 two different strategies can be pursued for placing the sync nodes:

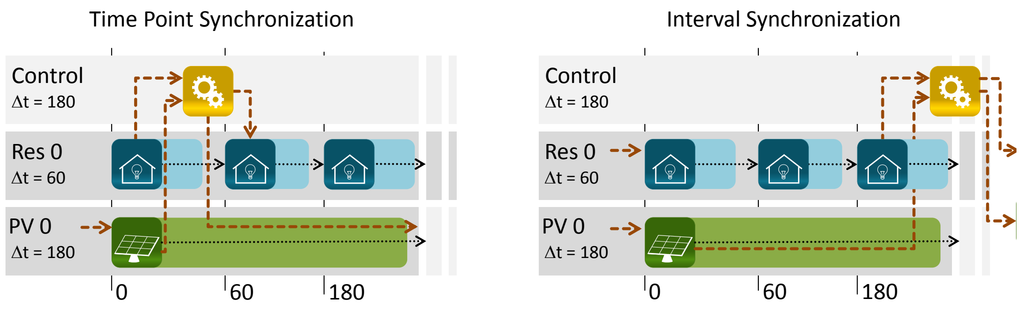


Figure 9.19: Alternative approaches for placing synchronization nodes in the simulator schedule graph

Time Point Synchronization Following this strategy, the sync nodes are placed as soon as all public simulators have provided data for the same simulation time point ($t = 0$ in the example). The control mechanism(s) can read the status of the simulated entities for this point in time and submit control commands. These commands are received/processed by the entities at their next time step.

Interval Synchronization An alternative to this approach is to place the sync nodes as soon as all public simulators have provided data for the same interval ($[0, 180[$ in the example). Compared to the aforementioned strategy this has the advantage that a control mechanism has the possibility to perform aggregating calculations for outputs of the faster stepping entities such that the interval of the aggregated data matches with the interval of the outputs from the slower stepping entities. Also, potential control commands can be issued for those time points at which all simulators provide the next outputs. As calculating average values for consistent intervals is important (e.g. for observing if loads or producers fulfilled a schedule), the Interval Synchronization approach is pursued. For cases with all public simulators having the same step size both approaches result in the same schedule.

Calculating the synchronizations step size

To determine how many synchronization nodes are needed, the synchronization step size Δt_{sync} has to be calculated. This can easily be done by simply taking the maximum step size of all public simulators. Here, the slowest public simulator governs the synchronization step size. Algorithm 6 shows the resulting algorithm for integrating the synchronization nodes, once Δt_{sync} has been calculated.

$$\Delta t_{sync} = max(s.stepSize : s \in public(simulators)) \tag{9.12}$$

Algorithm 6 Procedure for adding synchronization nodes

```
1:  procedure addSyncNodes($SG$)
2:      for $t = 0 \to step_{max}$ step $\Delta t_{sync}$ do
3:          Add node $sync_t$ to V                              ▷ Create sync node for t
4:          for $s \in public(simulators)$ do
5:              $t_{pre} = t + \Delta t_{sync} - s.stepSize$
6:              Add directed edge $sim^s_{t_{pre}} \to sync_t$ to E        ▷ Add edge from predecessor
7:
8:              $t_{succ} = t + \Delta t_{sync}$
9:              if $t_{succ} < step_{max}$ then
10:                 Add directed edge $sync_t \to sim^s_{t_{succ}}$ to E     ▷ Add edge to successor
11:             end if
12:         end for
13:     end for
14: end procedure
```

Figure 9.20 shows the resulting overall schedule of the example scenario, including the sync nodes. For illustration purposes it is assumed that only the PV and residential load entities are declared public, i.e. accessible by a potential control strategy.

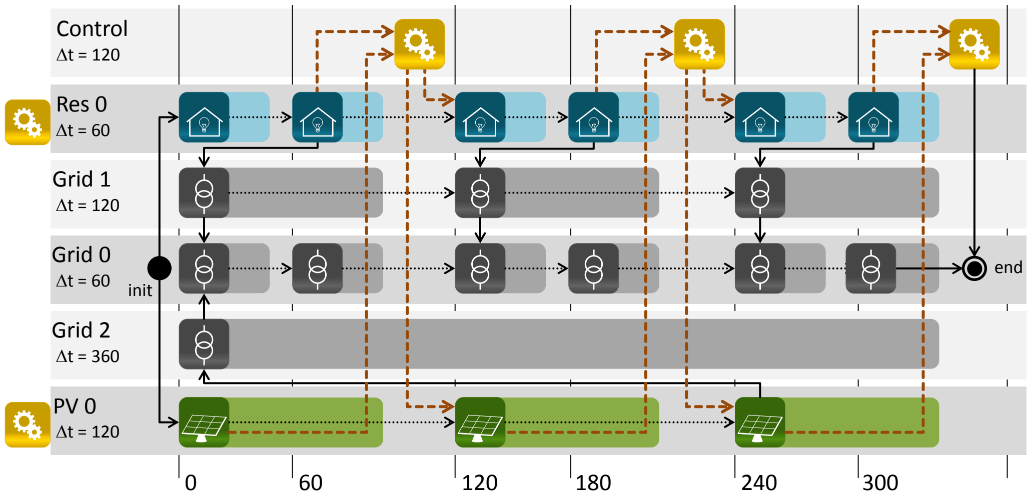


Figure 9.20: Sample schedule with control strategy synchronization points

9.7 Discussion

In this Chapter, a concept for implementing the composition layer has been presented. It can largely be split into three subsequent parts. In the first part, a data flow graph (DGF) was created. It is used to efficiently (i.e. without instantiating the scenario and simulator processes) determine if the data flows of a SimulationStudy contain unresolved cycles (see Section 9.4). Also, the DFG can be used to detect what EntitySets need to be simulated by distinct simulator processes (see Section 9.3).

The second part dealt with the actual instantiation of the SimulationStudy. An algorithm for determining the required number of simulator processes and collecting their initialization data (Algorithm 2) was presented. Also, the task of applying the ConnectionRules was discussed which involved the subtasks of expression analysis (see Section 9.5.3.2) and CSP solving in rare cases.

In the third and last part, an algorithm (Algorithm 4) for creating a schedule graph (SG) was presented. It is created once before the simulation is started and processed node by node until the end of the simulation is reached. This way the potentially large number of data flows only has to be analyzed once in the beginning. The algorithm supports simulators with different step sizes [**R 19** – Different Temporal Resolutions] as the nodes of the SG represent distinct steps of the simulator processes and the dependencies between the nodes are created accordingly (see Algorithm 5). Also, the SG considers specified delays to resolve cyclic data flows [**R 6** – Cyclic Data Flow Support]. Finally, the SG has been extended to contain synchronization nodes, so that an API allowing control strategies to interact with the simulated entities can be implemented [**R 12** – Synchronization of Control Mechanisms]. The concept of such an API is part of the control layer presented in the next section.

Limitations

Although the composition layer meets the relevant requirements, a number of limitations can be identified. However, as will be shown later, these are minor limitations that are not relevant for the Smart Grid scenarios considered in the evaluation chapter.

Expression optimization In Section 9.5.3.2 different approaches for optimizing the expression used in static conditions were discussed. It was argued that for the Smart Grid scenarios identified during requirements analysis only very simple expressions are required. Therefore, no further expression optimization approach has been integrated into the concept besides a fixed processing order for different predicate types.

CSP detection In Section 9.5.3.3 it was shown that a CSP can occur for certain ConnectionRules. Although a check was presented that can identify situations in which no CSP occurs (to avoid complexity of CSP solving), there is no check available that can identify situations in which the solcing the CSP is the only possible option. However, for most cases the CSP does not occur. This is also true for all scenarios that are used in the evaluation chapter.

Simulator step sizes Assumption [**A 7** – Any two simulator step sizes are integer multiples] limits the current simulator scheduling approach to simulators the step sizes of which are whole multiples (e.g. 1, 5, 10, 20, ...). For those simulators identified during requirements analysis and used for evaluation purposes, this limitation is not an issue, as according to experience, usual step sizes in the domain are 1 or 15 minutes.

Simulator consolidation Algorithm 2 implicitly assumes that it is beneficial to let a single simulator process simulate as many model instances as possible. However, depending on the used hardware (multi-core) there may be cases where it is faster to use separate simulator processes and let them execute in parallel. While this can be controlled by the *maxNumOfModels* attribute of the SimulationModel definition, there is currently no automatism to determine the ideal number of simulator instances.

Interpolation support In Chapter 7.9 it was argued that interpolation of data flows need not be supported [**A 5** – Interpolation is not reasonable in the majority of cases]. The scheduling algorithm presented above would have to be extended in order to support interpolation. Simulators that have at least one outgoing data flow that is subject to interpolation would have to advance on step further to enable triggering their successors. This way two values would be available to allow linear interpolation.

10 Control Layer

The control layer is the topmost layer of the mosaik concept. When its mechanisms come into play the simulators have been initialized and the composition has been performed according to the scenario definition. Its purpose is to provide an interface called *ControlAPI* that can be used by control mechanisms to access the state of the simulated entities and submit control commands to them.

10.1 Requirements

The requirements that have to be met by the control layer are briefly recapitulated in the following. All of these have been discussed before for one or more of the other conceptual layers, as these provide the functional base for the control layer presented in this section.

R 8 – Physical Topology Access One of the advantages that makes multi-agent systems a suitable paradigm for the control of Smart Grids is their inherent capability to match its structure [JB03, p.67]. The agents of an MAS can be arranged according to the topology of the Smart Grid elements that are to be controlled and tasks can be distributed among these agents accordingly, resulting in a scalable solution. As pointed out in the requirements analysis, manual bootstrapping is not an ideal option for initializing an MAS (providing an initial set of agents and relations among them). For being able to automatically create an initial set of agents and relations among them, a bootstrapping component needs to have information about the physical topology and its elements. The ControlAPI must provide functionality to retrieve this topology information.

R 10 – Combined Physical Topology The topology information that can be accessed via the ControlAPI has to include the model-internal entity relations (retrieved by the *get_relations* function of the SimAPI) as well as those relations that are created by the ConnectionRules.

R 11 – Static Data Access By accessing the topology information, the MAS can be bootstrapped for a scenario without providing additional information. By also allowing to access the static entity attributes via the ControlAPI, the MAS can also be parameterized dynamically, e.g. by accessing operational limit attributes of the entities or capacity data in case of storage entities. Of course, in both cases, topology and static data, this dynamic configuration only works properly if the simulators implementing the SimAPI can provide the required information via the SimAPI.

R 12 – Synchronization of Control Mechanisms In the last section it has been shown how synchronization points for the synchronized, consistent interaction of control mechanisms with the simulated physical entities have been created. The ControlAPI must provide a mechanism for synchronizing control strategies at these synchronization points.

10.2 Related Work

In Chapter 4 different works dealing with Smart Grid simulation were discussed. Several of these works also offer an interface to control integrate control mechanisms and power system simulations. However, non of the approaches explicitly addresses the aspect of dynamically bootstrapping a topology aware, multi-agent based control mechanism as discussed in Chapter 3.3. The requirements for this have been recapitulated above. Table 10.1 gives an overview of those works and if they meet these requirements. As the presented works do not give many details about synchronization and no details about multi-agent system bootstrapping, the control layer as it is presented below has barely been influenced by these works.

Table 10.1: Related work with respect to the requirements of the control layer

Approach	R 8	R 11	R 12
EPOCHS [HWG+06]	?	?	+
GridIQ [Ban10]	-	-	(+)
IDAPS [PFR09]	?	-	n/a
ILIas [KGC+12]	?	-	+
IPSSS [BGL+04]	?	?	?
Smart Cities [KN09]	?	?	?

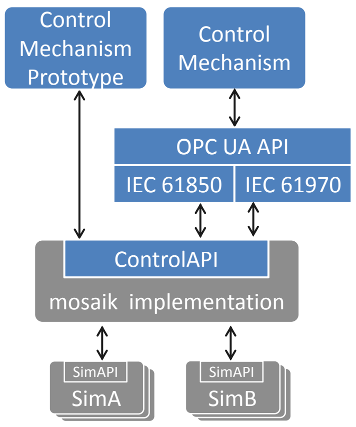


Figure 10.1: Architectural concept of the ControlAPI

10.3 ControlAPI Definition

For a number of reasons that will be discussed in detail in Section 10.5, an energy standard compliant integration of control mechanisms is beneficial. However, as there are a number of standards available (e.g. see Chapter 2.4), the control layer of the mosaik concept defines the ControlAPI independent of any standard, but directly based on the data structures of the reference data model defined on the semantic layer. The ControlAPI can then be used to implement mappings to the desired energy standard compliant data models. The latter include semantic standards (data models) as well as syntactic standards (protocols). Figure 10.1 depicts this concept using the examples of IEC 61850 and IEC 61970 (CIM) data models and the OPC Unified Architecture (OPC UA) which is proposed for the SmartGrid automation, for example, by Rohjans et al. [RUA10].

A detailed description of the functions offered by the ControlAPI can be found in Appendix F. Besides functions to get and set dynamic entity data, the ControlAPI includes the following functions to meet the aforementioned requirements and to allow dynamic bootstrapping of agent systems:

get_entity_types() Allow to retrieve a list of EntityTypes that are available. The returned list includes the names of the EntityType classes to allow retrieving entities for specific models as well as the names of the related AbstractEntityTypes to allow the creation of model independent mappings (see Section 10.5).

get_entities_of_type(type) Returns a list of ID strings for all entity instances of the given EntityType or AbstractEntityType name. In combination with the above function this allows, for example, to retrieve all electric vehicle entities or all transformer entities.

get_related_entities(entity_id) Returning a list of entity IDs of entities that are related to the entity with the given ID. An implementation of this function must consider model internal relations as well as relations resulting from the composition. In combination with the above function, it is for example possible to navigate a grid topology from a transformer via the related lines down to the subordinate consumers and producers. This assumes that the power grid simulator allows access to the model internal entity relations.

get_static_data(entity_id) Returning a dictionary with attribute name/value pairs for the static attributes of the entity with the given ID. As the names of the static attributes are specific for the EntityType, the referenced DataDefinition's name is used. In combination with the above function it is possible, for example, to read out operational limits of the transformers or line segments and parametrize the control strategy accordingly.

get_available_flows(entity_id) Returns a list describing the available data flows for the entity with the given ID. As the names of the entities PortData do not carry any semantics, the related DataDefinitions are returned as well. To avoid ambiguity, the DataDefinitions are grouped Port-wise and accompanied by the AbstractEntityType that the corresponding Port references as target type. The OCL constraints **C 21** and **C 22** introduced in Chapter 7.5 ensure that the returned combination of DataDefinition and AbstractEntityType representing the target type of the Port is unique per EntityType.

10.4 ControlAPI Usage

The usage of the ControlAPI is discussed in the following two sections. The first section describes the usage for initializing a control strategy and the subsequent section describes the control flow at simulation run-time.

10.4.1 Bootstrapping

In general, there are two different ways to bootstrap (set up) a control mechanism for interacting with the simulated entities via the ControlAPI. The easiest way is to code a control strategy for a specific scenario with specific simulation models. In such a case, only the *get_entity_types* method is used to retrieve the IDs of the entities that are to be controlled and the *get_dynamic_data* and *set_dynamic_data* methods are called with the returned IDs and hard-coded data flow names (attribute *name* of the class PortDataBase). Obviously, this way of interacting with the ControlAPI can be implemented quickly but is likely to break if different models are used as the names of the data flows are model specific. Therefore, this approach is only recommended to quickly test new control algorithm prototypes or test a specific model but not for implementing control mechanisms that require much effort or are likely to be used in the longer term.

The recommended way for such control mechanisms (and especially for complex MAS) is to use a dynamic bootstrapping based on model independent elements defined by the reference data model. This way, the control mechanism can be reused with different scenarios and models without having to change the code that ties it to the ControlAPI. As a simple centralized control mechanism can be considered a special case of an MAS with just one agent, only the MAS case with multiple agents is discussed in the following. The algorithm starts similar to the one above by querying entity IDs for different types. However, instead of using specific EntityTypes for the query, the AbstractEntityTypes defined in the reference data model are used. Next, it has to be checked if the different entities provide the input and output flows the control mechanism expects. This is done by invoking the method *get_available_flows* and compare the returned flow data with a list of expected DataDefinition-AbstractEntityType pairs that are defined in the reference data model. If an entity does not offer all expected flows the bootstrapping procedure may either exit with an error or a different type of agent requiring less flows can be used (i.e. with reduced control features). After this stage has passed one agent may have been instantiated for every entity, depending on the organization form of the MAS. Finally, for each agent a set of initial relations to other agents can be created by invoking the *get_related_entities* function. Also, each agent can be parametrized, for example according to the operational limits of the entity it represents, by retrieving the static data of its entity. Figure 10.2 summarizes the dynamic bootstrapping procedure that has been presented in this section in form of an UML sequence diagram.

10.4.2 Run-Time

The ControlAPI offers three functions to manage the control flow between mosaik and its clients (ControlAPI users). Compared to the functions mentioned above, these are invoked by mosaik and not by the clients.

set_client(client) Called by mosaik to register a client (a user of the ControlAPI). It is assumed that the client is created by mosaik as mosaik has to provide the client with information about how to connect to the ControlAPI (for invoking the functions defined above). Hence, mosaik can directly connect the client at the ControlAPI.

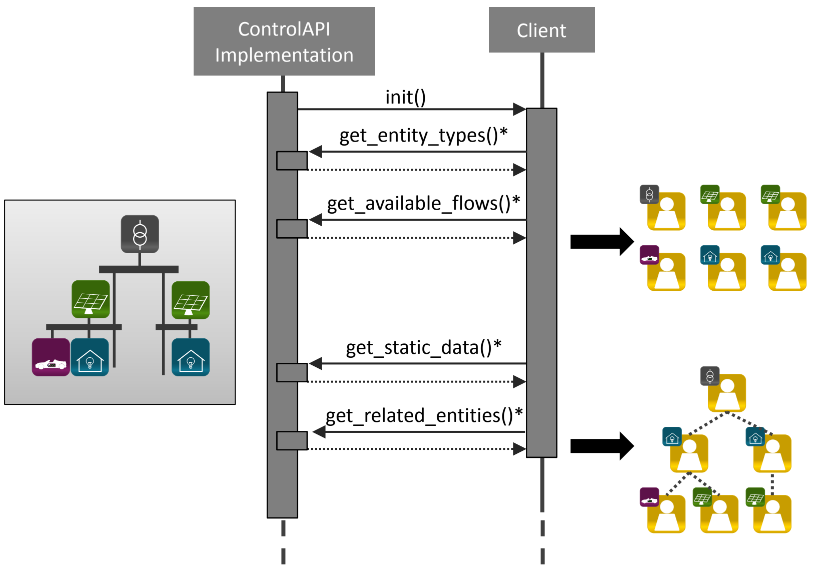


Figure 10.2: Using the ControlAPI to bootstrap an MAS

init(absolute_start_time) This function is called by mosaik to initialize the control mechanism as soon as the simulators have been initialized and static data as well as entity relations are available. To do so, the ControlAPI invokes a blocking *init* function on the registered client. When the function returns the simulation can start.

step(seconds_since_start) At every reached synchronization point (see Chapter 9.6.5) mosaik invokes this method which passes the elapsed time since simulation start to the ControlAPI. The latter invokes a blocking function *activate* on the registered clients to trigger the control functionality. As soon as all of these functions have return, e.g. after all agents finished their work and potentially submitted commands to the simulated entities, the simulation can continue. This approach is comparable to the one presented by Konnerth et al. [KGC$^+$12] discussed in Chapter 4.1.5. Their ILIas framework also stops the simulator until it has received a command from all agents.

Figure 10.3 depicts the usage of the ControlAPI using the syntax of a UML sequence diagram. The client has to offer the methods *init* and *activate*. The ControlAPI, as it is presented here, only allows to register a single client (hence the name *set_client* and not *add_client*). This client may be another API, for example mapping the ControlAPI to a standardized API. In case of a MAS the different agents may be seen as individual agents that like to register at the ControlAPI. However, as the individual agents should not have to deal with mosaik specific synchronization details, the current ControlAPI design is based on the assumption that a dedicated synchronization agent implements the activate function and manages the synchronization of the other agents. Different approaches for synchronizing MAS (which usually have no explicit notion of time) have been summarized by Rohlfs [Roh13] and are not part of the mosaik concept. Although only a single client can be registered, the different agents can directly use the get and set data functions of the ControlAPI, once they are activated. To enforce consistency, the

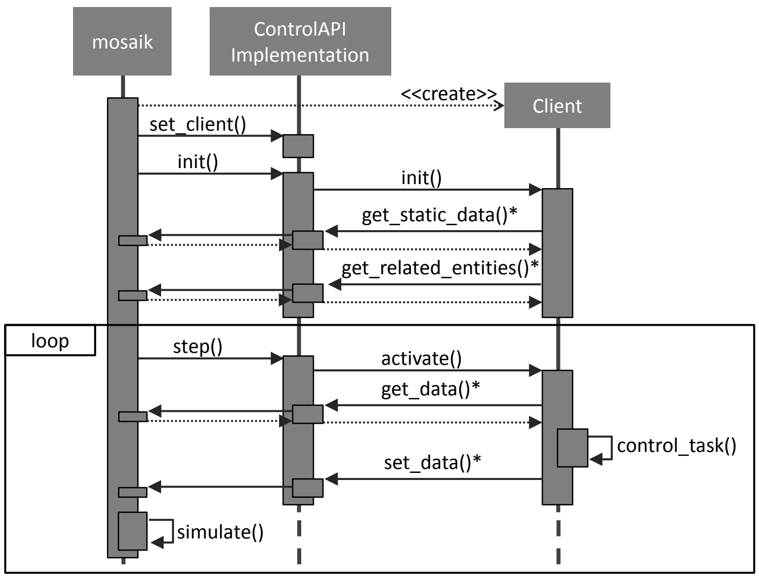


Figure 10.3: Using the ControlAPI

get and set data functions should be implemented in such a way that an error is raised as soon as they are called after the activation function has returned.

10.4.3 Parallel and sequential control

Rohlfs [Roh13] has identified two different types of control strategies which can both be integrated into mosaik using the ControlAPI, as illustrated in Figure 10.4.

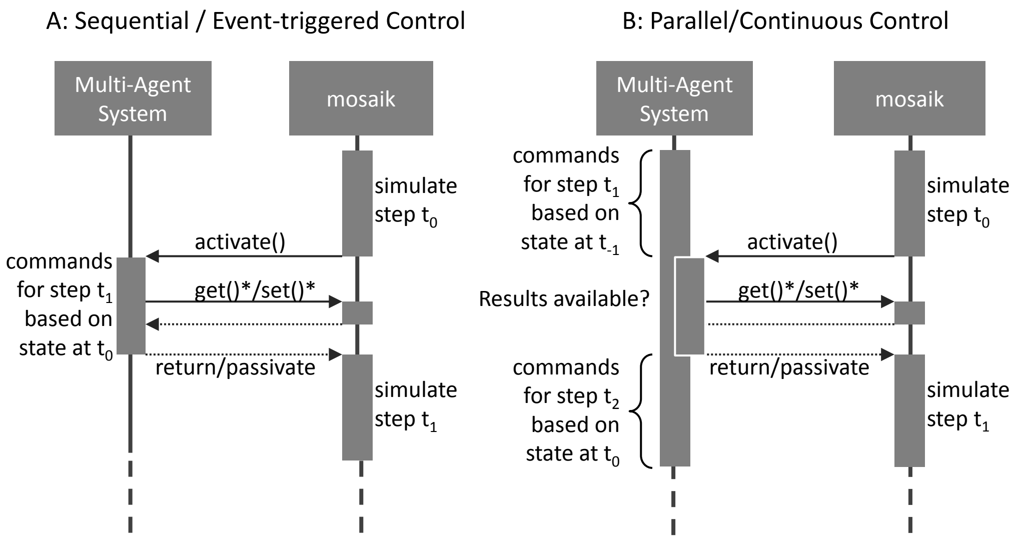


Figure 10.4: Parallel and sequential execution of control strategies

Sequential (or event-triggered) control strategies do only perform actions in response to an event (e.g. a state change of a simulated entity or a certain advance in simulation time). If the time taken to perform the actions is small compared to the synchronization step size with mosaik, this time can be neglected and no simulation time has to elapse during these operations. Hence, the control strategy can do its work while the simulation is halted. Mosaik and the control strategy run in a sequential manner. An example for such a control strategy is the calculation of the confidence in a forecasted behavior of a group of energy consumers and producers for providing a virtual product on the energy market [Ott12]. For certain events, the confidence may change and the calculation is triggered again.

As the name suggests, parallel (continuous) control strategies are continuously active, e.g. working on some kind of optimization problem. An example for such a strategy is the continuous rescheduling algorithm presented in Nieße and Sonnenschein [NS13], in which the optimization goal is to identify a schedule that leads to a similar grid usage as the original schedule for clusters of small distributed generation units, consumers and electrical storages. As such optimization procedures usually run for a long time span, time cannot be neglected and simulation time also has to proceed (in parallel). Usually, there is a specific deadline at which the best as yet available optimization result is used although optimization could continue. When a synchronization point is reached and the control strategy is informed about the current simulation time there are different options to match this behavior:

1. Simulation time is used to detect a defined deadline. If simulation time has reached the defined deadline, the optimization is stopped, corresponding commands are submitted to mosaik and the simulation can continue. Depending on the start criterion, the optimization may be started again with the new entity states or wait for another event to be triggered again.
2. In most cases simulation time will elapse faster than wall-clock time. Therefore, using the simulation time for deadline detection will result in poorer optimization results as the optimization procedure runs less long. Hence, wall-clock time may be used to determine the deadline. If simulation time has already reached the deadline, the optimization procedure may still continue but mosaik is blocked until the optimization timespan of wall-clock time and simulated time match.

Due to their duration, such strategies cannot always work with the latest data available. When using mosaik, this means that the control strategy may not retrieve new data at every synchronization point but continue optimization based on old data. However, as Figure 10.4 (B) shows, even if the control strategy returns results and reads new data at every synchronization point, the result is based on data that is one synchronization time span older than in the sequential approach. This additional delay cannot be avoided for the parallel execution of simulation and control mechanisms.

10.5 Standard Compliant Access

The dynamic bootstrapping procedure presented in Section 10.4.1 allowed to reuse a control strategy with different scenarios and models of mosaik. However, the integration in such a case is still specific to the syntax of the ControlAPI. By relying on syntactic and semantic standards that can be found in the energy domain, it is possible to implement and test a control strategy independently of a specific test bed such as mosaik. There are a number of benefits that a standard compliant API offers:

Thorough validation If a control strategy is using a well known standard to interface its test bed, it requires no or only few effort to test it with other simulation tools offering the same standardized interface. Using different tools (and thus models) for simulating the same scenario allows to identify tool and model specific errors and behaviors (biases) [HRU10]. Using a standardized API (syntactically and semantically) would allow to evaluate a control mechanism with mosaik as well as with other simulation tools that can simulate the same scenario.

Protocol overhead Another benefit of using a standardized API is the capability of measuring the overhead that specific standards introduce. This overhead is twofold. First, data structures are likely to get more complex when mapping to a semantic standard. Secondly, the use of syntactic standards may introduce additional protocol overhead which can then be measured for a realistic communication scheme.

Migration Finally, when implementing a control strategy against a standardized API, it is likely that the migration of the control strategy into a production environment, for example a small field trial or a hardware-in-the-loop (HIL) test setting, is easier to accomplish.

Mapping to a syntactic standard usually requires to expose the functions offered by the ControlAPI using a different protocol. Rohlfs [Roh13] has shown how this can be accomplished for the OPC UA standard by creating as many nodes in the OPC address space [Int07] as the scenario contains entities and copy data in a one-to-one fashion into the address space.

In addition to this, a mapping to a semantic standard (such as the IEC 61970 CIM) can be implemented. Ideally, such a mapping should be implemented independently of the specific models that are used to simulate a scenario. Otherwise, a standard specific mapping has to be create for each new model that is used in mosaik. The API function *get_available_flows(entity_id)* can be used to retrieve the semantics of the available flows independently of the name of a data flow for the specific entity. As discussed in Section 10.3 OCL constraints ensure that the returned combination of DataDefinition and AbstractEntityType is unique for each EntityType. As the API function *get_entity_types()* also returns the AbstractEntityTypes, it is possible to implement mappings to the data structures of a semantic standard in a model independent, rule-based fashion. For example, for each entity with the AbstractEntityType *Consumer* that has a data flow of type *ActivePower* a mapping to a corresponding consumer data structure of a standard could be specified.

10.6 Discussion

The following requirements are met by the ControlAPI presented in this chapter:

R 8 – Physical Topology Access, R 10 – Combined Physical Topology The Control-API offers a function *get_related_entities(entity_id)* to retrieve a list of entities that are related to the entity with the given id. To meet requirement [**R 10** – Combined Physical Topology], the entity relations defined in the model itself as well as those that have been established by a ConnectionRule are returned by the ControlAPI.

R 11 – Static Data Access The ControlAPI also offers a dedicated function to retrieve static entity data.

R 12 – Synchronization of Control Mechanisms The synchronization between the simulators and the control strategy has been discussed in Section 10.4.2 and is implemented by the explicit invocation of a step function which activates the control strategy and returns when the control strategy has finished interaction.

Part III

Evaluation and Conclusion

“Everything is theoretically impossible, until it is done.” – Robert A. Heinlein

11 Implementation

In the last chapter a layered concept for the mosaik platform has been presented. This chapter describes how the different layers of the concept have been implemented.

11.1 Technical and Syntactic Layer

Although the conception of the technical layer is not part of this thesis, it is described here briefly to be complete. Also, requirement [**R 28** – Execution on Multiple Servers] that was elicited in Chapter 3 is implemented by this layer. To do so, the technical layer has been implemented to act as a middleware that can span multiple simulation servers and offers an API to start or connect to specific simulators, interact with them via the syntactic layer and terminate them once the simulation is finished or has failed. The development team chose Python 3 to implement this layer. ZeroMQ was chosen as messaging library as it promises a good performance and implementations for various languages exist, including C/C++, Python, Java and .NET [iMa12]. Figure 11.1 depicts the architecture of the technical layer and the ZeroMQ based communication between the different components.

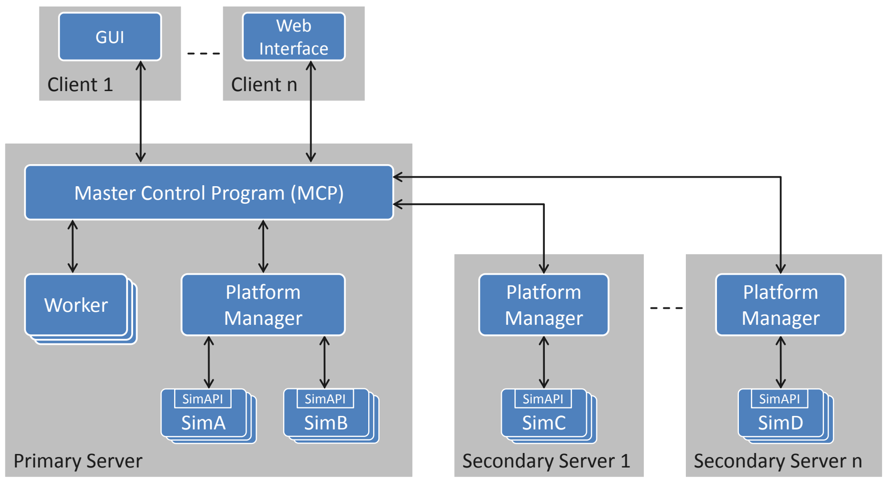


Figure 11.1: Architecture of mosaik and the technical layer.

A primary server is running a central process called *Master Control Program (MCP)*. It manages the connection to the clients and the scenarios being simulated. Starting and controlling simulators is handled by the *Platform Managers (PMs)*. One PM is started automatically when the MCP starts. PMs on remote machines need to be started manually and connect themselves to the MCP. The composition and execution of each scenario is done by a dedicated *Worker* process. They automatically shut down when the simulation has finished. If a simulation study specifies different parameter combinations, one worker is instantiated for each combination so that these can be simulated in parallel. Different clients can connect to the MCP to upload simulation studies, monitor the progress of the corresponding workers and retrieve simulation results, logged by the workers.

Syntactic Layer

In Chapter 6 the SimAPI, a discrete-time simulator interface providing basic integratability, has been presented. The methods of the SimAPI can be found in Appendix A. Besides the requirements discussed during conception of the SimAPI, the implementation of the SimAPI meets two important requirements:

[R 21 – Heterogeneous Implementations] The SimAPI also relies on ZMQ as it is available for more than 40 programming languages [iMa12] and supports a wide range of platforms, such as Linux, OS X and Windows.

[R 15 – Transmission of Complex Data Types] As identified in Chapter 3.5.2 and supported by the semantic metamodel presented in Chapter 7.4.4, the SimAPI must allow the transmission of complex data types. This is achieved by interchanging method parameters according to the JSON (JavaScript Object Notation)[1] format. JSON is a lightweight, textual data-interchange format that is, although being based on a subset of a JavaScript Programming Language, language independent. The ZeroMQ technology used for the implementation of the mosaik technical layer natively supports JSON as data-interchange format.

As yet, the syntactic layer implementation of mosaik offers libraries for Java and Python that contain abstract base classes implementing the SimAPI via ZeroMQ. An Integrator (see Chapter 3.1.2) can use these libraries to implement the SimAPI for a specific simulator without having to learn and deal with ZeroMQ specifics. More libraries for other languages such as C++ will be developed in the future.

11.2 Semantic and Scenario Layer

The implementation of the semantic as well as the scenario layer has to provide a convenient mechanism for the mosaik users to specify simulator descriptions and scenario models that are valid with respect to the semantic and scenario metamodels defined in Chapter 7 and 8.

11.2.1 The Mosaik Specification Language (MoSL)

Based on these metamodels, the Xtext framework[2] has been chosen to implement a domain specific language (DSL) that can be used to define the simulator descriptions as well as scenario models. As the scenario models reference elements of the simulator descriptions, it is reasonable to include both metamodels in a single DSL. The Xtext framework is based on the Eclipse modeling Framework and belongs to the family of language workbenches [Tho11]. A language workbench is an environment to support the development of new DSLs, combined with a set of tools for using these DSLs effectively [FP10, p.22]. This includes tools for compilation, interpretation, code generation, model validation, error handling and syntax highlighting, for example.

[1] `http://www.json.org/` (accessed 10 April 2013)
[2] `http://www.eclipse.org/Xtext` (accessed 12 April 2013)

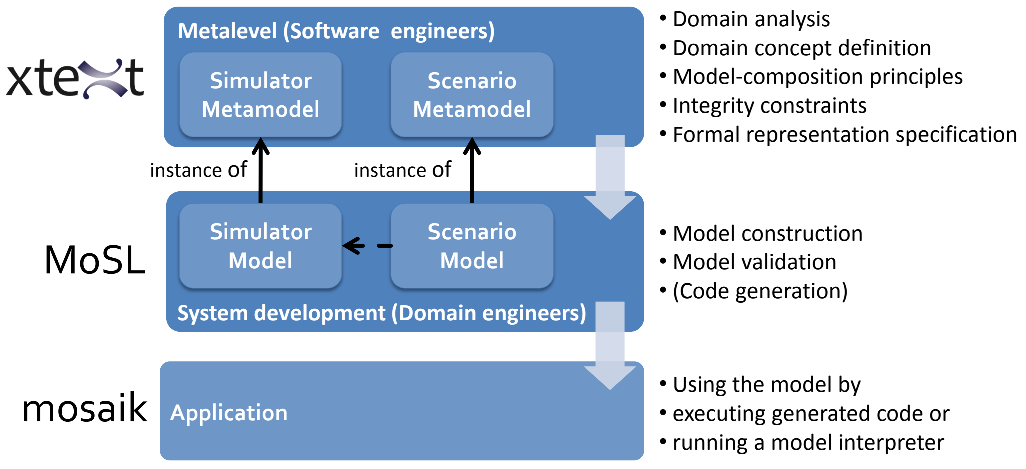


Figure 11.2: A textual DSL based MIC-Approach

Using Xtext, the process of implementing the semantic and scenario layer follows the Model-Integrated Computing (MIC) approach presented by [SK97]. This process is depicted in Figure 11.2. On the *Metalevel*, software engineers define the domain concepts (the abstract syntax of the DSL), the principles for composing models out of these concepts as well as integrity constraints and a formal representation of the concepts (the concrete syntax of the DSL). Based on this information, an automatic synthesis of a domain specific modeling environment can be performed. This environment is then used by domain engineers to build the model. In case of Xtext the generated modeling environment is an editor that can be exported as Eclipse-plugin. The users are those who want to integrate simulators or describe a Smart Grid scenario. Using the integrity constraints defined on the metalevel, the model can be automatically validated. Finally, for the application step, the model is interpreted by the target application or code for the target application is being generated. In case of mosaik, this step involves the interpretation of the scenario model and performing the simulation of the resulting composition.

Design alternative

An alternative approach to this textual definition is the usage of a graphical DSL/environment, for example the Generic Modeling Environment (GME) [Van] which has evolved from the initial MIC approach proposed by [SK97]. The IAM approach described by Soma et al. [SBO+06] (see Chapter 8.2.5) is based on GME, for example. For a general discussion of the advantages of textual languages see Gronninger et al. [GKR+07]. For mosaik the choice was made for a textual DSL for a number of reasons. Xtext is widely used and has a very large community and integrates seamlessly with the other parts of the Eclipse Modeling Project[3]. This allowed the automatic generation of the UML diagrams as well as an Eclipse based evaluation of the syntax and functionality of the OCL constraints presented in this thesis (also GME supports OCL, too). Also, Xtext allows to change the DSL and the underlying metamodel quickly and corresponding

[3] `http://www.eclipse.org/modeling/` (accessed 10 April 2013)

model instances can be adjusted easily, e.g. using find and replace operations. This is much harder in graphical environments and hence, Xtext was better for rapid prototying during the development of the mosaik metamodels. From the user perspective it can be argued that a graphical DSL offers a better overview of the structure of the model. However, due to the scalability requirement and the resulting concept of EntitySets and ConnectionRules, the structure of a mosaik scenario model does not exactly match the resulting scenario. Therefore, the visual advantage of a graphical DSL is limited. Finally, the artifacts created by a textual DSL (the simulator descriptions, the scenario models, etc.) can be managed conveniently by standard versioning systems, as demanded in Chapter 7.11.

11.2.2 DSL Implementation

The implementation of the MoSL DSL with Eclipse Xtext is comprised of two different projects, *mosaik.mosl* and *mosaik.mosl.ui*, which implement different parts of the concept and are described briefly in the following paragraphs. The basic structure of both projects was generated by the Xtext project wizard and has been extended accordingly.

mosaik.mosl

This project contains a number of files that define the metamodel and concrete syntax of the DSL, implement the constraints defined for the metamodel and provide two code generators for further processing of the models created with the DSL.

MoSL.xtext In case of mosaik and Xtext the metamodel (abstract syntax) as well as the concrete syntax of the DSL is defined by an EBNF-like[4] syntax. This file contains the definition for the MoSL DSL. It is listed in Appendix G. For the part of the DSL that allows to define the complex data types of the reference data model, an existing Xtext project *orderly_json_xtext* [CRE12] has been used as a basis. The metamodel classes DomainModel, Simulator, CompositeModel and SimulationStudy make up the root elements of the DSL. That is, instances of these classes can later be defined in separate files. This allows to create a file structure that is similar to the structure in which the objects are referenced. A SimulationStudy references a CompositeModel which references one or more simulator descriptions which again are based on a referenced DomainModel. All these objects must have unique names for being able to reference them unambiguously. This is achieved by allowing to declare a package name for each file. For example *de.offis.energy.simulator* can be used as a package name for simulator descriptions to avoid name clashes with simulators provided by other parties in future projects.

MoSLScopeProvider.java For associations referencing elements defined in other parts of the DSL (and not created directly as part of an aggregation), the scope provider contains methods that are used to retrieve the allowed association target objects. The OCL constraint **C 37**, for example, defines that a SimulatorParameterSet must only reference those ParameterDefinitions that belong to the referenced Simulator.

[4] EBNF = Extended Backus-Naur Form

Listing 11.1 shows the corresponding scoping function. A number of other OCL constraints is also implemented by the scope provider and not as validation checks as it is more convenient for the user if the system offers only valid objects as association targets instead of offering all objects and raising an error if the user picks the wrong one.

Listing 11.1: Scoping example: SimulatorParameterSet

```
public IScope scope_ParameterInstance_instanceOf(
    SimulatorParameterSet pi, EReference ref) {
  return Scopes.scopeFor(pi.getSimulator().getParameters());
}
```

MoSLJavaValidator.java This file contains the *MoSLJavaValidator* class that implements additional validation checks that have not been covered by the scope provider. It is derived from the *AbstractMoSLJavaValidator* which is generated by Xtext specifically for the MoSL DSL (based on the definitions in *mosl.xtext*) and contains abstract validation methods that can be overridden in the subclass to implement the checks. Thus, the different methods reflect the different *context* classes of the OCL constraints. The checks are specified using plain Java code. The integration of checks defined in OCL syntax is possible when using the *CompleteOCLEObjectValidator* class. However, this OCL support is in a prototypical stage and has not been used for the current implementation.

MoSLGenerator.xtend This file contains a code generator defined in the Xtend[5] syntax that is used to generate a single YAML[6] file for each SimulationStudy which contains all directly or indirectly related objects of the scenario model. Therefore, it is sufficient to pass the single YAML file to the simulation engine to simulate a simulation study.

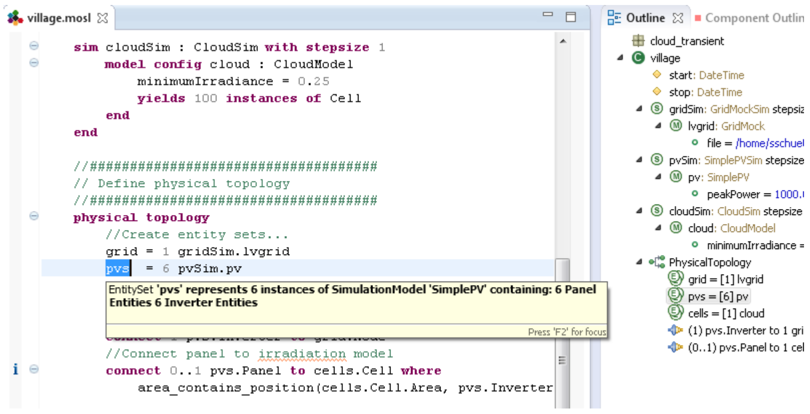
MoSLPythonGenerator.xtend As the YAML file is a serialization of the metamodel instance created by the Xtext DSL editor, corresponding classes have to be available when deserializing (loading) the file in the simulation engine (the composition layer implementation). As the latter is written in Python, the *MoSLPythonGenerator* generates Python classes for the defined metamodel. While the YAML generator is executed every time a simulation study is defined, these have to be regenerated only if the metamodel is changed (e.g. by the developers of mosaik and not by the users).

mosaik.mosl.ui

Based on the artifacts of the *mosaik.mosl* project, Xtext automatically generates a number of files which constitute the editor of the MoSL DSL. It can be deployed as Eclipse plugin and thus be used by the mosaik users for describing the reference data model, the available simulators and of course, to define the scenario models. Additional files have been added to provide more information to the user. This includes information that is displayed for individual elements when the mouse hovers over it and a tree view

[5] `http://www.eclipse.org/xtend/` (accessed 12 April 2013)
[6] `http://www.yaml.org/` (accessed 12 April 2013)

that shows the outline of the DSL file that is currently edited. Figure 11.3 shows an example for such additional information.



village.mosl

```
sim cloudSim : CloudSim with stepsize 1
    model config cloud : CloudModel
        minimumIrradiance = 0.25
        yields 100 instances of Cell
    end
end

//####################################
// Define physical topology
//####################################
physical topology
    //Create entity sets...
    grid = 1 gridSim.lvgrid
    pvs  = 6 pvSim.pv
```

EntitySet 'pvs' represents 6 instances of SimulationModel 'SimplePV' containing: 6 Panel Entities 6 Inverter Entities

Press 'F2' for focus

```
    //Connect panel to irradiation model
    connect 0..1 pvs.Panel to cells.Cell where
        area_contains_position(cells.Cell.Area, pvs.Inverter
```

```
cloud_transient
village
    start: DateTime
    stop: DateTime
    gridSim: GridMockSim stepsiz
        lvgrid: GridMock
            file = /home/sschuel
    pvSim: SimplePVSim stepsize
        pv: SimplePV
            peakPower = 1000.
    cloudSim: CloudSim stepsize
        cloud: CloudModel
            minimumIrradiance =
    PhysicalTopology
        grid = [1] lvgrid
        pvs = [6] pv
        cells = [1] cloud
        (1) pvs.Inverter to 1 gri
        (0..1) pvs.Panel to 1 cel
```

Figure 11.3: MoSL editor plugin generated by Xtext with syntax-highlighting, outline view (right) and custom pop-up information

11.3 Composition Layer

Similar to the technical and syntactic layer presented above, the composition layer has been implemented using Python. Following the composition concept presented in Chapter 9, the data flow graph has been implemented by using the *NetworkX*[7] library. It also offers the function *simple_cycles* to return the elementary circuits of a graph, as required for the composition procedure. Also, the function *shortest_path* was used for implementing the procedure checking if two EntitySets require different simulator processes as described in Chapter 9.3. The scheduling algorithm presented in Chapter 9.6 has also been implemented based on the NetworkX graph library. An open-source CSP library called *python-constraint*[8] was used to implement the CSP solving part of the composition layer.

Figure 11.4 shows a UML diagram with the major classes of the implementation. The *mosl* package contains the Python classes generated by the Xtext *MoSLPythonGenerator* presented above. These are required to deserialize the scenario model when being passed to the *Experiment* class of the *composer* package. This package contains all logic that is required to compose and execute the experiment (one variant of a SimulationStudy) according to the algorithms defined in Chapter 9. The *scenario* package contains all classes required to interpret the deserialized scenario model and instantiate the corresponding scenario. Therefore, corresponding instance classes of the scenario model classes have been created. When the ConnectionRules (not shown in the diagram) are interpreted, these operate on the instance classes (e.g. *EntitySetInstance*). As the interpretation of a scenario model and the composition of the entities is independent of the chosen discrete-time simulation approach, the implementation is framework-like and simulator and entity subclasses containing the discrete-time specific operations need

[7] `http://networkx.github.io/` (accessed 16 April 2013)
[8] `http://labix.org/python-constraint` (accessed 16 April 2013)

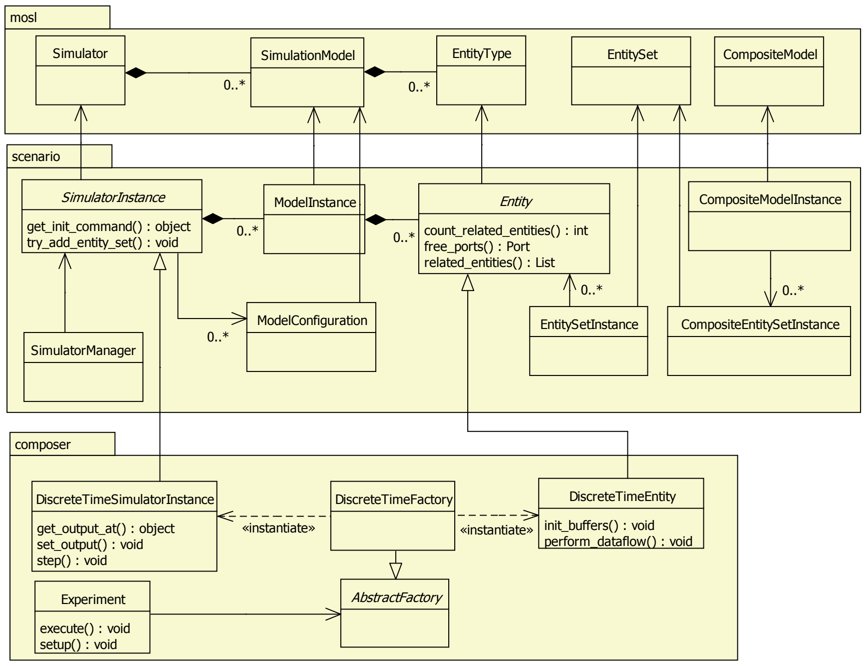


Figure 11.4: Class diagram of the composition layer implementation

to be derived from the abstract classes of the scenario package. This is the case for the *SimulatorInstance* and the *EntityInstance* class. As these only contain the logic for initializing the simulator processes and composing the resulting entities, the subclasses in the composer package carry the logic to perform the data flow in discrete time steps. In Chapter 12.4.4 it will be demonstrated that this framework structure works as intended to enable the use of the scenario specification mechanism of mosaik with other simulation engines.

Once the composition phase has finished (and the simulation can start), the implementation of the composition layer saves a number of files to a scenario specific folder (scenario name + time stamp). These files include a serialized copy of the data flow graph, a serialized copy of the schedule graph (both exported in the *graphml*[9] format) and a CSV based list of data flows grouped by source and target entity type. This list also includes information about how the data for each flow is calculated (see Figure 6.7). These files are intended as information source for the Scenario Expert to judge if the scenario has been composed and simulated as intended and also as debug aid for the developers. The relations of the entities (the primary outcome of the composition phase) are stored in an HDF5[10] database which also contains all output data of the entities as demanded by requirements [**R 26** – Data Logging] and [**R 27** – Logging of Static Data]. This relation information may be required for result visualization, for example to create a grid topology specific heat map. By integrating it into the database, no additional files but the database file are required for result analysis.

[9] `http://graphml.graphdrawing.org` (accessed 18 July 2013)
[10] `http://www.hdfgroup.org/HDF5` (accessed 17 July 2013)

11.4 Control Layer

Although control strategies could be implemented in a plug-in fashion, that is they are part of the same process as the mosaik engine, the current implementation starts them as separate processes and offers access to the ControlAPI presented in Chapter 10 via ZMQ. Therefore, the technical layer does not need to distinguish between simulator and control strategy processes and can be used to integrate both. Integrating control strategies as separate processes has a number of advantages. First, they do not have to be implemented in Python but can be implemented in any language that is supported by the ZMQ framework (see Chapter 11.1). Second, as MAS based control strategies are in the focus and an MAS can span multiple computational nodes, remote access to the ControlAPI is beneficial. Finally, when running the control strategy in a separate process, control strategies that are running continuously (i.e. in parallel to the simulated entities, see discussion in Chapter 10.4.3) can be implemented more easily.

11.5 Summary

In this chapter the implementation of the mosaik concept has been presented. Figure 11.5 summarizes the described implementation by showing the architecture of the mosaik platform from a high-level perspective. The semantic as well as the scenario metamodel are implemented as a DSL called MoSL, using the Xtext framework. The Scenario Expert uses the Eclipse based DSL editor to create a scenario using MoSL. To simulate a scenario it is serialized in the YAML format and transferred to the server (using a remoting capable command line client) which interprets the scenario model and composes and executes it by using the simulators made available by the different servers that are integrated into the mosaik platform. During simulation, control strategies can interact with the simulated entities. Similar to the simulators, the control strategy process is started by the mosaik platform and communicates via ZMQ. However, instead of using the SimAPI offered by the syntactic layer, a control strategy uses the ControlAPI offered by the control layer.

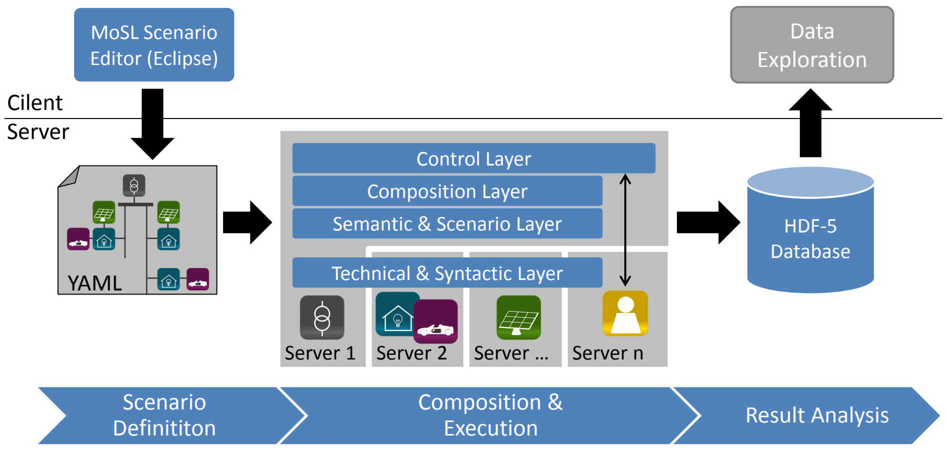


Figure 11.5: Architecture of the mosaik platform

12 Evaluation

In this chapter it is verified that the implementation presented above shows the behavior defined in the concept and that the resulting system is valid with respect to the four research questions that were initially defined. Besides the case studies, the simulators and scenarios identified in the requirements chapter, two case studies of real-world projects are used for the evaluation. After having introduced these case studies in Section 12.1, they are picked up again in the following sections, each dealing with the evaluation of one of the research questions. At the beginning of each section the respective research question is recapitulated and a number of criteria for the evaluation are presented. As each layer of the mosaik concept can be related to a research question (see Figure 5.3), the resulting evaluation covers all layers. Figure 12.1 shows the resulting structure of this chapter. Finally, a first performance benchmark of the scenario defined for the Smart Nord case study is presented and the assumptions that were made in this thesis are discussed. This chapter concludes with a brief summary of the evaluation results.

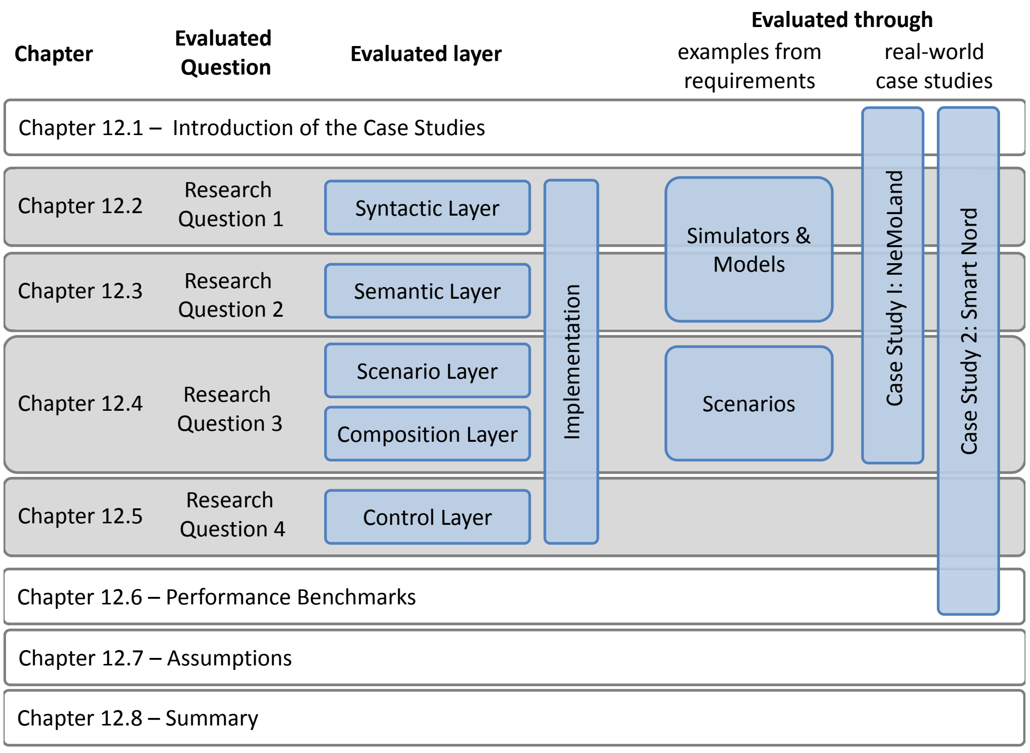


Figure 12.1: Structural overview of the evaluation

12.1 Case Studies

As examples of actual practice are more compelling than examples of idealized problems [Sha02], two case studies from current research projects are used for evaluation. These are briefly introduced in the following as the case studies are picked up almost all subsequent sections.

12.1.1 Case Study I: NeMoLand

Workpackage 3 of the project “Neue Mobilität im ländlichen Raum (NeMoLand)” [NeM11] aims to identify potential synergies of EVs in commercially operated fleets and local PV installations and to analyze the impact of such fleets on the local power grid [HSB12]. In order to do so, different models have to be integrated into an overall scenario. As the simulative part is carried out at OFFIS, mosaik was chosen for this task. In the following sections there will be shown that the simulators required for carrying out this project could be integrated into mosaik and that the required scenarios could be described in MoSL and be composed and simulated correctly. Figure 12.2 shows the structure of the simulated scenarios. Four different representative LV grids are to be analyzed, involving a different number of EV fleets, residential or commercial loads, PV installations and micro CHP generation. Detailed topologies of the grids are not available. However, the number of nodes is given and node specific consumption and PV/CHP feed-in profiles are given (see Section 12.2.3.1), so the overall power balance of the LV grid can be calculated and analyzed with respect to the operational limits of the local substation. The models that are used and the scenarios that are to be composed will be introduced and discussed below. The fleet model that is used includes a charge controller which can be used by a central control mechanism to reduce the fleets power consumption in case of exceeding the operational limits of the substation.

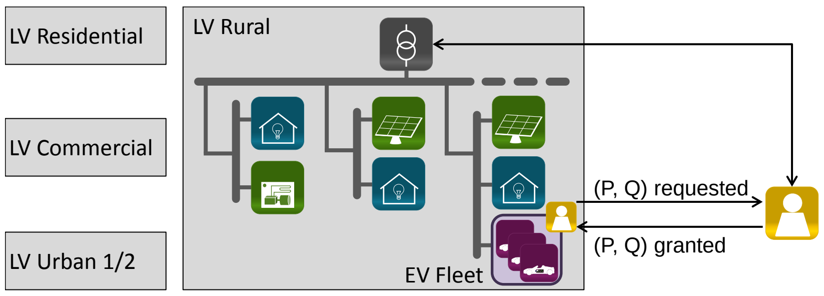


Figure 12.2: Scenario of the NeMoLand project

12.1.2 Case Study II: Smart Nord

While the NeMoLand scenarios only involve a single LV grid, the Smart Nord[1] case study poses a bigger challenge on the scenario specification mechanism. Figure 12.3 depicts the structure of the Smart Nord power grid. An MV grid provided by a project partner (Leibniz Universität Hannover) makes up the base of the scenario. It is derived from an MV rural distribution network introduced by Rudion et al. [ROSS06] as a benchmark for distributed generation studies. Connected to each node of the MV grid are a number of 8 different LV grids such that an overall number of 112 LV grids results. These LV grids have been provided by another project partner (TU Braunschweig). In the following sections the case study is picked up again and details are given of the used models and how they are to be connected to the overall power grid.

The objective of the Smart Nord project is to propose concepts and algorithms for the

[1] `http://www.smartnord.de` (accessed 04 July 2013)

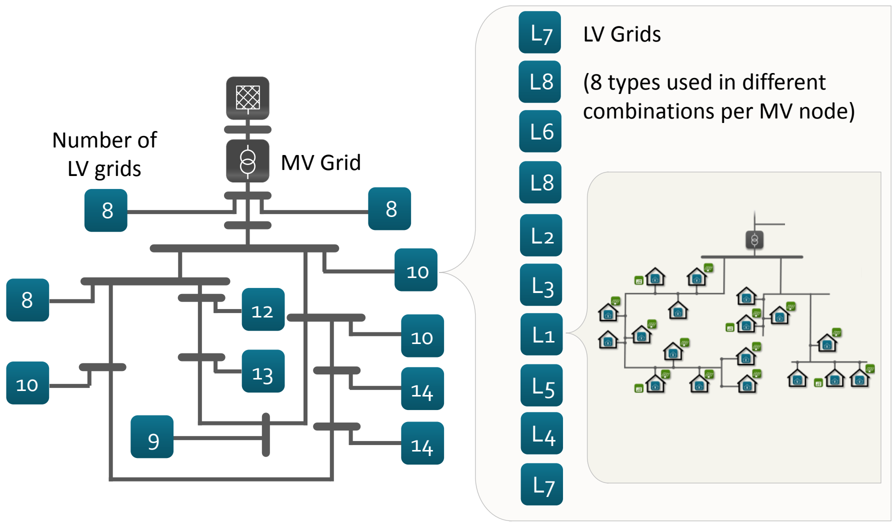


Figure 12.3: Power grid topology of the Smart Nord project (based on [BBD$^+$13])

decentralized provision of real, reactive power as well as spinning reserve in distribution networks. Special focus is put on the dynamic aggregation of consumers and producers to meet agreed schedules while at the same time considering the operational state of the power grid. An MAS is being developed which intends to provide these capabilities. A simulation based on the mosaik implementation is the basis for evaluating this MAS. In Section 12.5 it will be shown that the required functionalities are offered by the control layer of the mosaik implementation and that they work as intended.

12.2 Syntactic Integration

RQ 1 – What interface is required to integrate different simulators into composed Smart Grid scenarios?

This section deals with the evaluation of the first research question and thus, with the implementation of the syntactic layer. While the requirements [**R 23** – Well-defined Simulator API], [**R 22** – Discrete-Time Simulator Integration (fixed step size)], [**R 11** – Static Data Access] and [**R 19** – Different Temporal Resolutions] are met by the availability and the design of the SimAPI itself, a number of other criteria can be evaluated using the actual implementation of the syntactic layer. These are derived from the other requirements (arrows indicate the section(s) in which the criterion is evaluated):

R 15 – Transmission of Complex Data Types Can complex data types be used as inputs or outputs of simulation models? ► 12.2.1.1

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs Can simulation models with more than one entity type and varying numbers of entities be integrated? ► 12.2.1.3, 12.2.2.2

R 18 – COTS Integration Can COTS simulators be integrated? ► 12.2.2

R 20 – Different Representations of Time Can simulators with different simulation paradigms (i.e. event-discrete, continuous time handling) be integrated? ► 12.2.1.1, 12.2.2.3

R 21 – Heterogeneous Implementations Can simulators implemented on different platforms/languages be integrated? ► 12.2.2.1

The following evaluation is subdivided into three subsections dealing with the integration of open-source simulators, closed source-simulators and the integration of the simulators used in the case studies. The first two subsections cover the simulation models presented in Chapter 3.5 and the integration of the COTS power-flow simulator (CERBERUS) which is used as a representative for other COTS power-flow tools. For each integrated simulator/simulation model, the used approach according to Figure 6.5 is mentioned and the corresponding MoSL description file can be found in Appendix I. It has to be pointed out that these descriptions are based on elements of the reference data model which has not been discussed yet, but is presented in Section 12.3. This section order has been chosen nonetheless as it follows the layers of the mosaik concept and thus the structure of the conceptual part of this thesis.

12.2.1 Interfacing Open-Source Simulators

Option 1 of the integration concepts shown in Figure 6.5 could be employed to integrate all simulation models into mosaik that use open-source simulation frameworks.

12.2.1.1 EVSim

The electric vehicle simulation already introduced in Chapter 3.5.2 could be integrated into mosaik easily, as it has been developed at OFFIS and the complete source code is available. A new *EVSim* class has been added which implements the SimAPI and invokes the existing simulator class of the EV simulation in a similar fashion as the alternative GUI version of the EV simulation does it. Although the EV simulation package contains distinct classes for the vehicle itself, the battery and the battery controller, the current implementation of the SimAPI represents the complete vehicle (including battery and controller) as a single entity. Therefore, the MoSL description of the EV simulation, shown in Appendix I.5, only defines a single EntityType *EV* for the defined model. Static data includes the storage capacity of the EV and the voltage level, as defined in Section 12.3 below. Dynamic data outputs include the charging/discharging power of the EV and the status information (traveled distance, SoC, etc.). As optional dynamic input a schedule can be submitted which is a complex data type derived from the CIM type *IrregularIntervalSchedule* (see Section 12.3). As actual schedules are submitted to the simulation in Section 12.5.1, this shows that [**R 15** – Transmission of Complex Data Types] is met.

Figure 12.4 illustrates how the EVSim is integrated from a conceptional point of view, following the discussion in Chapter 9.6.3. In the shown example a schedule is submitted to the EV at t_1, which tells the EV to charge until t_2. Next, following the set, step, get order of the SimAPI, the step function is called. According to Definition 7 calling the

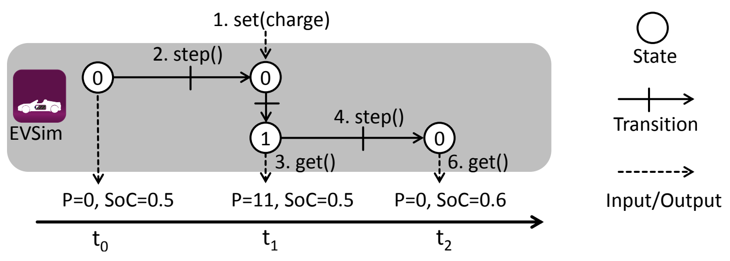


Figure 12.4: Discrete-time integration of the EVSim

step function has to advance the simulator time by the defined step size (here from t_0 to t_1) and the subsequently returned data must be valid for the new interval (here [t_1, t_2[). In case of the EVSim the returned power and SoC values are sampled values valid for the first time point of the interval. Therefore, the step function processes all events of the EVSim up to t_1 (included). As the charging command is available at t_1 and the delay for processing such a command is only a few seconds [Bro02], it can be neglected when simulating with a step size of 1 minute or more. Therefore, the EVSim does not account for such delays and charges immediately. Hence, the data retrieved by the get function (3.) already accounts for the charging power (here 11kW). The SoC of the battery is unchanged as charging is a slow gradual process which cannot be neglected. After the next step call (no further command is submitted in this example), charging has ended and P goes back to zero and the SoC of the Battery has changed accordingly. The EVSim shows that a discrete-event model can be integrated into mosaik [**R 20** – Different Representations of Time].

12.2.1.2 Appliance Sim

For the scenario of the Smart Nord case study (introduced below), the ApplianceSim introduced in Chapter 3.5.3 had to be integrated. As the source code is available the syntactic integration was straight forward and similar to the integration of the EVSim. So far, only the *Fridge* model has been integrated. The other appliance models can be integrated in a similar fashion. As the fridge model is very detailed, a number of different EntityTypes are defined in the MoSL description. The electrical components (consumer), the fridge compartment (thermal), the thermostat controlling the regular operation and an entity type representing the usage behavior of the fridge (e.g. when it is opened, closed) are the basic types. For using a fridge in load-shifting scenarios, a scheduler type for processing provided schedules as well as a sampler, which can provide a number of possible operation schedules for the fridge[2], are available. The MoSL description of the ApplianceSim is shown in Appendix I.1. Some DataDefinitions that are specific to the fridge model have first of all been defined in a separate data model file (also shown in Appendix I.1) and may be moved to the reference data model when other models require the same structures.

[2] Such randomly sampled operational schedules can be used to create a search space model representing the individually constrained feasible operational schedules of DER, as discussed in Bremer and Sonnenschein [BS13]

Open issues

The integration of the sampler entity has revealed a number of drawbacks that the current concept (caused by the SimAPI) imposes. As the calculation of the operational schedule samples offered by the sampler entity is a time consuming computational task, it is not advisable to update the returned sample values at every simulation step. Also, as a control mechanism may not require samples at every time step and the sample data structure can become quite large (depending on the sample size), providing new samples at every time step causes unnecessary performance losses. Therefore, the input *sample_size* has been added which allows to indicate how many samples are to be calculated. If it is set to zero, no samples are calculated and returned. However, this is still no ideal solution as this means that a control mechanism has to request a sample one step in advance. The control has to be handed back to the simulation which advances to the next time step to update the sample output. This complicates requesting the samples. Here, an extension of the SimAPI which allows control mechanisms to request dynamic data on demand from an entity would be desirable, especially as the other appliance types of the ApplianceSim also include a sampler entity. This could not be considered during the requirements analysis presented in Chapter 3.5 as this functionality was added later specifically for the Smart Nord project.

12.2.1.3 CSV Sim

In many cases, simulation studies do not exclusively involve simulation models, but also time-series data for different types of entities (e.g. based on real measurements). This is the case for climate data such as solar irradiance or environmental temperatures, for example, as well as for the standardized residential load profiles discussed in Chapter 3.5.4. From the perspective of the mosaik simulation engine it does not matter if the data that is queried via the SimAPI is created by some stochastic, discrete-time simulation algorithm or simply read from a CSV file. Therefore, a generic simulation called *CSVSim* has been implemented that acts as an adapter for integrating such time-series data into a simulation scenario. Generic in this case means that the implementation of the CSVSim does not have to be changed for different CSV files if a MoSL description of the contents of the file is provided and the file is formatted as follows:

```
EntityType         ;Household          ;       ;Household          ;
StaticData         ;float_VoltageLevel ;0.4    ;float_VoltageLevel ;0.4
                   ;<type>_<name>      ;<value>;                   ;
DynamicData        ;float_p            ;float_q;float_p            ;float_q
2004-07-12 00:00:00;928                ;273.07 ;743                ;250.83
2004-07-12 00:15:00;206                ;56.62  ;104                ;45.33
...
```

The name of the EntityType defined in the first row indicates a column starting a specific entity. Static entity data is defined from the second row on (until a row starting with “DynamicData” is hit) with the type/name in the first column of the entity followed by the specific value in the next column. Finally, an arbitrary number of dynamic data time-series can be defined followed by the specific values for each time step. This information is sufficient to allow a generic implementation of the SimAPI. For each CSV file, a corresponding MoSL description for a CSV file has to be created and the occurring EntityTypes can be used in a scenario model as any other EntityType. The MoSL file tells the simulation engine what EntityTypes and data flows are available and the CSVSim can map corresponding requests to the specific values of the CSV file based on the header information. Appendix I.4 shows the MoSL definition for the above example of residential household profiles. For different CSV files different model descriptions are added to the MoSL file, matching the contents of the file (i.e. EntityTypes, static and dynamic data). As both, SimAPI and CSV time-series follow the discrete-time paradigm, the integration is straight forward. At each call of the *step* function a new row is read and the subsequent *get* call returns these values. An initial *start* parameter allows to define an initial row by matching the parameter with the date entries in the first column. The CSVSim shows that requirement [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs] can be met, as a CSV file can contain any number of entities/entity types.

12.2.2 Interfacing COTS/Closed-Source Simulators

In this section it is shown how different closed-source simulators are integrated into the mosaik platform [**R 18** – COTS Integration].

12.2.2.1 PVSim (Matlab)

The Matlab/Simulink based PV simulation model (running on a Windows 7 virtual machine) has been integrated into mosaik. For interfacing Matlab/Simulink a number of options exists. As the PV model itself is available, one option is to create an S-Function block that implements the SimAPI and to visually connect it to the blocks of the PV model. However, this requires a good piece of Matlab expertise and some changes to the existing model initialization scripts. Therefore, a less complex and non-invasive option has been chosen which corresponds to option 3 of the integration concepts shown in Figure 6.5. It relies on a third-party library *matlabcontrol*[3] that allows to interface Matlab from Java. With this library a SimAPI implementation has been created that invokes Matlab and controls the PV model without having to change any model code. Based on the elements of the proposed reference data model (see Section 12.3), a MoSL simulator description has been created for the Matlab/PV model combination. It is shown in Appendix I.8. As the mosaik implementation is based on Linux/Python and the Matlab integration is based on Windows/Java this is an example for [**R 21** – Heterogeneous Implementations], which is met for all languages and platforms supported by the used ZeroMQ remoting library.

12.2.2.2 CERBERUS

Besides Matlab, the power flow simulator CERBERUS[4] has been integrated into mosaik as a representative of COTS power-flow tools. A similar integration approach is applicable to other power system simulation tools (e.g. PowerFactory [MHG+12]), as

[3] `https://code.google.com/p/matlabcontrol/` (accessed 17 April 2013)

[4] see `http://www.adapted-solutions.com/web/ASProduktCerberusUeberblick.html` (accessed 14 June 2013)

these are based on similar modeling concepts and objects. However, other tools have not been integrated yet, due to the complexity of the automation APIs and missing licenses.

CERBERUS could be integrated without much hassle following option 4.a of the integration concepts shown in Figure 6.5. Again, an adapter has been created which implements the SimAPI and adapts and forwards the calls to CERBERUS, using its http-based automation API. In the initialization phase CERBERUS is started as console application and a grid topology file is passed as argument. It has to be created manually beforehand using the CERBERUS user interface. Next, the contained entities are retrieved using the http-API and returned to mosaik. For the different entities that make up a power grid model in CERBERUS, parameters and outputs can be distinguished. Outputs are changed by a power flow calculation, whereas parameters are set initially (when loading the power grid model) and do not change through a power flow analysis. Therefore, outputs represent the dynamic output data returned by the SimAPI and parameters are either returned as static data or may be set as dynamic data inputs. CERBERUS does not define explicit inputs, as it was not designed for being used in combination with other tools. Generation or consumption behavior of consumers and producers modeled in CERBERUS is therefore changed by adjusting the parameter values and not through inputs from other tools. The corresponding parameters of these entities represent the dynamic input data set by the SimAPI. The types of entities that have been exposed by the SimAPI (and their names) correspond to the model elements that CERBERUS uses. Section 12.2.3.2 shows that exposing the entities via the SimAPI in such a direct way is not always the preferred case. For CERBERUS only a subset of the available elements has been exposed by now which is sufficient for testing the mosaik integration. When being used in an actual project further elements may be added to the SimAPI implementation. The resulting MoSL description of the simulator can be found in Appendix I.2. The *get_relations* function could be implemented for CERBERUS, as the http automation API of CERBERUS allows to retrieve connection (pin) information of the different entities. The example of integrating CERBERUS shows that [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs] can be met. Different power system topologies involving different numbers and types of entities can be handled without adapting the implementation of the adapter.

12.2.2.3 FMI Compliant Simulators

The FMI standard for co-simulation defines a generic API for interfacing FMUs (see Chapter 6.2). As the standard does currently only support primitive (or scalar) data types and the reference data model proposed in Section 12.3 limits itself to use primitive data types for the physical flows, a generic adapter can be implemented easily that maps the FMI interface to the SimAPI defined by mosaik.

To show that this is feasible, first a very simple Modelica model of a pumped hydroelectric storage has been created based on technical data given in [RWE]. It can operate in pump mode (154 MW, 102 m^3/sec) or turbine mode (153 MW, 110 m^3/sec) and has a ramp up time of 60 seconds. Next, a generic adapter has been created based on the PyFMI library that is part of the JModelica[5] project. An FMU

[5] `http://www.jmodelica.org` (accessed 30 May 2013)

corresponds to a simulation model of mosaik and a single entity. The adapter expects a *file* parameter defining the FMU that is to be loaded. The names of the inputs and outputs that a specific FMU offers need to be specified in the corresponding MoSL description (see Appendix I.6 for the pumped hydro storage FMU used here). When these are requested by mosaik via the get and set methods of the SimAPI, the adapter performs corresponding get and set calls on the FMU via the PyFMI classes. Figure 12.5 shows a plot generated from the resulting simulation logfile for a scenario containing a single hydro storage that is operated in turbine and pump mode for ten minutes each. A simple control strategy based on the ControlAPI presented in Chapter 10 issues the corresponding commands. The FMI adapter is an example for integrating models following the continuous time paradigm [**R 20** – Different Representations of Time]. Figure 12.6 shows how the continuous model is sampled in discrete time intervals. Input data is assumed to be valid for the interval $[t_0, t_1[$. The step function is called and the output trajectory is returned by PyFMI for this interval. Finally, the get function returns the value at t_0.

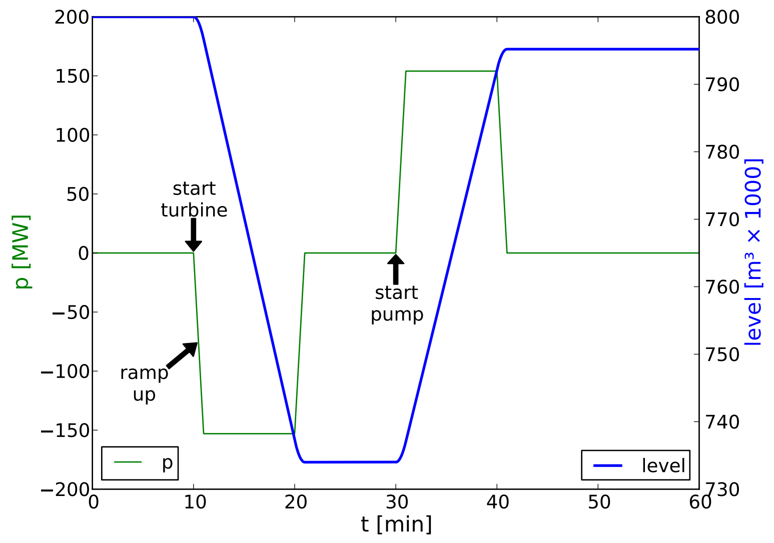


Figure 12.5: FMI pumped hydro storage example

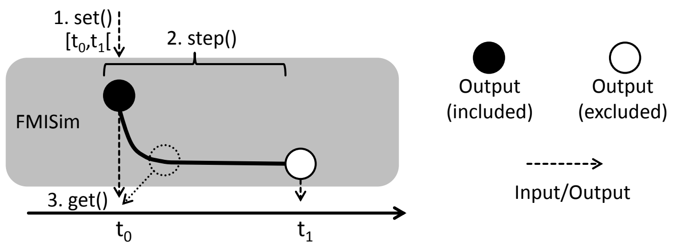


Figure 12.6: Discrete-time integration of the FMISim

Open issues

Although it could be shown that the integration of FMI based models works in principle, there are a number of possible improvements. For example, in Chapter 7.4 it was argued that the semantic metamodel should allow to specify how a unit is decomposed into the corresponding SI base units to allow the creation of generic simulator adapters such as the one presented in this section. The FMUs can also contain information about the SI base units of the inputs and outputs [MOD12b]. Provided with the used MoSL description, the FMI adapter could automatically check if the units are identical and perform factor conversions automatically (e.g. transform the power from MW to W in this example). In the current example, this conversion was hard-coded into the adapter. This should be improved in a later version. Also, when specifying control commands as complex data types (e.g. compliant to the CIM), the FMI adapter may have to be provided with plugins that convert this to signals that the FMU can understand. Finally, the pump storage has a very slow behavior, therefore it was valid to simply sample the values at the given time instants. For quickly reacting systems (e.g. like the EV discussed above) this sampling may not be the optimal strategy as returning average values over a certain interval or a slightly delayed sampling (indicated by the dashed state in Figure 12.6) may be more desirable. This has to be decided from case to case.

12.2.3 Case Studies

So far, a number of models that were already known during requirements analysis and a COTS power flow simulator as well as an FMI compliant model have been integrated. To show the applicability of mosaik for new projects, the following sections discuss the integration of simulators required for the case studies already introduced above.

12.2.3.1 Case Study I: NeMoLand

The example of the NeMoLand case study (see Section 12.1.1) allows to demonstrate the possibility of integrating models of very different internal complexity into mosaik.

CSV Time-Series

CSV files with time-series data for the residential and commercial loads as well as for the PV and micro CHP generation were provided by a project partner for each of the four different power grids to analyze. To integrate these into the NeMoLand scenarios (see below), the generic CSV model already described in Section 12.2.1.3 could be used. A small Python script was created to bring the files into the expected structure and corresponding MoSL descriptions were created (see Appendix J.4). The script also performed the conversion from KiloWatts to Watts to match the power flow definition of the reference data model proposed in Section 12.3. As the conversion script was required anyway, the assumption that mosaik does not need to support unit conversion to ensure composability (see [**A 3** – No central unit conversion required], discussed in Chapter 7.4) was valid in this case.

Mambaa

While the time-series data represents the a very simple type of simulation model the project also demands the simulation of EVs operated in commercial fleets. A corresponding simulator called *Mambaa* [HHS13, HH12b] that has been extended with a project specific EV fleet model was provided by the OFFIS R&D division Transportation. Compared to the generic CSV-based model with a very simple (deterministic, input-free) internal behavior, Mambaa is a real simulator with much more intrinsic complexity. “Mambaa is a multi-agent based simulation framework that includes data models of vehicles and fueling infrastructure. It is designed to simulate the user’s mobility patterns, the negotiation between the user and the charging infrastructure and the charging process itself” [HSB12]. So although mosaik is used as a testbed for MAS, some simulators can be MAS as well. Especially for such complex simulation models the chosen distinction between models and entities is beneficial, as the different objects can be made explicit via the SimAPI. In case of Mambaa, this could be the charging stations and the vehicles of a model representing a single commercial fleet, for example. However, for the NeMoLand project it is sufficient to expose a single *ChargingStation* entity per fleet model. Similar to the integration of the EV simulation already described above, Mambaa is based on a discrete-event simulation framework (JASON [HH12a] in this case). Therefore, a similar behavior of the SimAPIs step function has been implemented to step Mambaa in discrete-time fashion. The MoSL descriptions of Mambaa can be found in Appendix J.4.

GridMockSim

As will be described in Section 12.1.1, the NeMoLand project does not consider a specific grid topology. To ease the calculation of nodal power balances (e.g. to make a statement about local PV consumption) as well as the overall power balance of the LV grids, a simple grid simulator called *GridMockSim* has been created. It contains a single transformer as well as a number of nodes that can be configured via a model parameter. Each node is directly connected to the transformer via a single line. Figure 12.7 shows this approach. For each node as well as for the transformer the power balance is provided as output values, whereas the node can also have a number of power inputs, e.g. provided by PV, EV and residential models. As any other grid simulation it can be classified as a memoryless model (see Chapter 2.1.4). The simulator was given the name Grid<u>Mock</u>Sim as it is an ideal simulator to serve as mock for power system simulators in automatic test cases, for example. This is because the model does not account for line losses such that the outputs of other models and the output of the transformer can be automatically compared easily. Appendix I.7 shows the MoSL description of the GridMockSim.

12.2.3.2 Case Study II: Smart Nord

In the Smart Nord project, the scenario is composed out of a number of different models which are partly provided by project partners. In the following it is discussed how these are integrated into mosaik. As the Smart Nord project is still in progress by the time this thesis is written, the integration approach for some models is discussed only

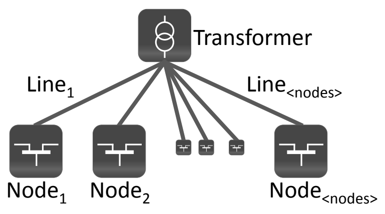


Figure 12.7: Star topology of the GridMockSim power system model

theoretically. All available models have actually been integrated and are used in the subsequent sections. The resolution of the simulation models will be one minute for all models.

CHP Plants & Heat Pumps Models for these types of resources are still under development by the time this thesis is written. The implementation will be in Matlab or Python, such that an integration similar to the PVSim (Matlab) or the EVSim (Python) will be possible, as already discussed above.

Domestic Appliances The ApplianceSim, which has been already discussed above, will be used to simulate domestic appliances. For the purpose of this evaluation, the model of a fridge has been integrated as example.

Electric Vehicles To simulate EVs, the EVSim which has already been discussed above will be used.

Photovoltaics By the time this thesis is written, the PV model used for Smart Nord is still under development. However, as it is developed by the project team using Python, it is open-source and integration will be similar to the EVSim from a technical point of view. From the structural and semantic point of view it will be similar to the PVSim discussed above but with an additional input to reduce PV feed-in. The EVSim integrated above shows that submitting such commands is possible (an actual example is given below).

Power Grid (load flow analysis) For simulating the power grid, the Smart Nord project relies on the open-source power flow simulation package *PyPower*[6]. It is supposed to be faster than the Pylon library[7] discussed in Chapter 3.5.5 and is therefore used instead of it. To integrate PyPower into mosaik, an adapter has been implemented, following integration approach 1 shown in Figure 6.5. The power system model of PyPower shows some structural differences to the one of CERBERUS, as PyPower internally uses a more abstract Bus-Branch model, whereas CERBERUS uses a Node-Breaker model, allowing to capture the grid assets that are part of the real-world topology in greater detail. Differences of these approaches have been discussed in Chapter 2.3.3. In PyPower only three different types of objects exist, namely *buses*, *branches* and *generators*. The branch objects are used to model transformers as well as transmission lines, as these can be treated in a similar fashion from the mathematical point of view [Mil10]. For each bus object the type P-Q, P-V or Ref

[6] `https://pypi.python.org/pypi/PYPOWER` (accessed 08 July 2013)
[7] `https://groups.google.com/forum/#!topic/pypower/7_KJzfdbkhw` (accessed 30 July 2013)

(=slack bus) can be specified. For P-V and Ref buses, a generator object has to be associated. All other buses are P-Q buses for which P and Q can be directly specified. To ease scenario modeling and the subsequent analysis of simulation data, the PyPower mosaik adapter defines a distinct EntityType for the Ref buses as well as for branch objects that represent transformers. Figure 12.8 shows this mapping. The corresponding simulator description can be found in Appendix I.9.

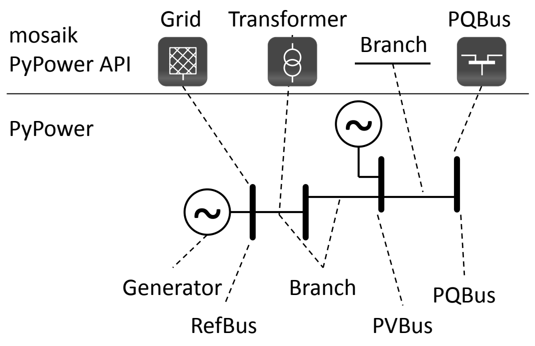


Figure 12.8: EntityTypes used for integrating PyPower

Residential Loads Have been provided as CSV time-series with 15 minutes resolution. An approach to interpolate the data to obtain data with 1 minute resolution is subject to current research by the project team. In any case, the integration can use the CSVSim discussed above. Details of how this data has been integrated will be given below when the scenario layer is evaluated.

Stationary Storages A model for stationary storages will be provided by a project partner for Matlab in a later stage of the project. Technically, the integration will therefore be similar to the Matlab PV model which has already been discussed above.

Wind Energy Converter (WEC) Similar to the PV simulation model, the WEC model used for Smart Nord is under development by the time this thesis is written. Again, it is developed by the project team using Python and therefore, the integration approach is similar, too.

12.3 Semantic Integration

RQ 2 – What information about a simulator and its models is required to facilitate the automatic composition of valid scenarios?

Although a number of requirements influenced the design of the semantic layer, it is only possible to judge its appropriateness when putting it into the context of specific scenarios. Therefore, this section only proposes a reference data model (and does not evaluate its usefulness) which is then being put to the test in the subsequent section dealing with research question three in which actual scenarios are composed. How the requirements have been considered for the design of the semantic metamodel has already been discussed in Chapter 7.12. The following sections discuss the major aspects of the different parts of the reference data model. The complete model can be found in Appendix H. The resulting simulator descriptions for the simulators that have been integrated above can be found in Appendix I.

12.3.1 AbstractEntityType Taxonomy

As motivated in Chapters 7.4.6 and 7.5 a taxonomy of AbstractEntityTypes is used to prevent semantic ambiguity when establishing the data flows. In this section a proposal for a taxonomy is made which is used in the remainder of the evaluation.

Taxonomy semantics

The AbstractEntityType class offers an association to one or more super types. Intuitively this seems like an inheritance relation in object-oriented programming (OOP). In OOP these are usually interpreted as “is a” relations. However, in case of the reference data model it is not advisable to create the taxonomy by strictly following the “is a” metaphor. *Variant 1* shown in Figure 12.9 is an example where this approach is misleading[8]. The port of the shown EntityType *Bus* accepts EntityTypes that are of type *Prosumer* or subtypes of these (see OCL constraint **C 24**). However, although the “is a” approach seems reasonable, the Bus cannot be connected to plain *EnergyConsumer* or *EnergySource* (producer) entities as these are no subtypes but super types. *Variant 2* shows an alternative semantic for the taxonomy relationships. By interpreting a subtype as a “subset of” the super type, the composition semantics work as intended and it is also reasonable from a linguistic point of view. A *Photovoltaics* entity, for example, is a subset (regarding its type) of all possible EnergySources which again is a subset (regarding its functionality) of a Prosumer.

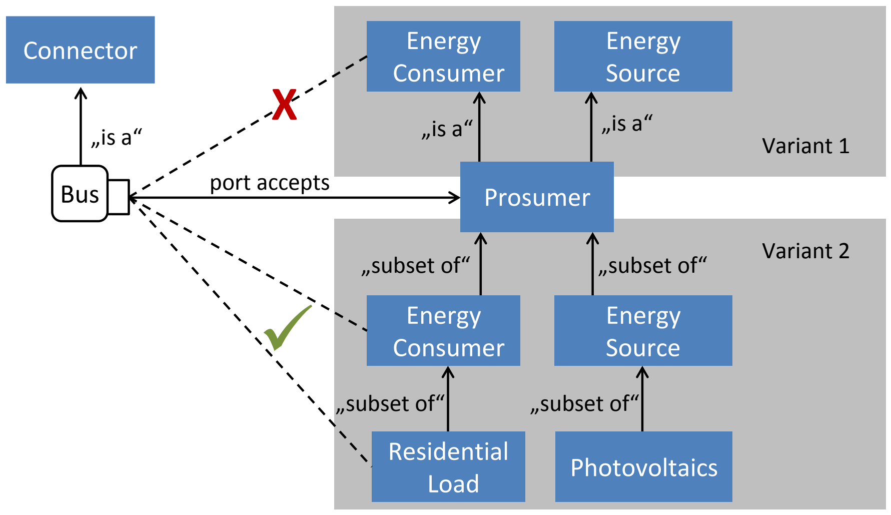


Figure 12.9: Variants for creating a taxonomy of AbstractEntityTypes

[8] In OOP the discussion if a square “is a” rectangle or vice versa is a well-known similar discussion, e.g. see `http://cseweb.ucsd.edu/classes/wi10/cse130/lectures/lecture-19.pdf` (accessed 08 May 2013)

Power system models

In Chapter 2.3.3 it was discussed that different types of resources exist, namely P-Q loads, P-V generators and P-Q generators. As the type of a resource influences the outcome of the power flow calculation, power flow tools either offer distinct entity types for these resources or corresponding configuration options have to be set and determine the behavior. Consumption and generation values for the power flow objects are provided by entities of other models in case of simulation model composition (see Chapter 3.5.5). When defining the composition, it has to be made sure that entities representing P-Q loads, P-Q generators or P-V generators are mapped to the corresponding P-Q load, P-Q generator or P-V generator entities of the power flow model. As discussed above, the power flow simulation tools integrated by now (CERBERUS and PyPower) have slightly different representations of the power grid. However, from the point of view of the entities that are to be connected to the power grid model, the required data flows (see next section) and the AbstractEntityTypes are similar. Figure 12.10 illustrates this for PyPower and CERBERUS, using the EntityTypes that the mosaik adapter of the respective simulator exposes. The *PQBus* in PyPower is used as connector for all objects (i.e. branches as well as load and generators) and is able to behave as P-Q load and generator (by setting positive or negative P and Q values). In CERBERUS, the connections are done by a *Node* object which cannot act as load or generator. Therefore, explicit load (LAST), generator (EEA) or storage (SPEICHER) objects have to be placed in the power grid model. To describe potential connection points for P-Q loads/generators or P-V generators regardless of their tool specific characteristics, the two AbstractEntityTypes *PQProxy* (for P-Q loads or generators) and *PVProxy* (for P-V generators) are proposed. The term proxy shall suggest that the entity may only represent the corresponding load or generator object in the power flow model (e.g. in case of CERBERUS) but uses the values provided by a simulated entity in a different model. The fact that a load in CERBERUS can only receive positive values (no feed-in), but has the same AbstractEntityType (PQProxy), can be accounted for by limiting the corresponding data flow types (introduced below) to positive values only (as discussed in Section 7.6).

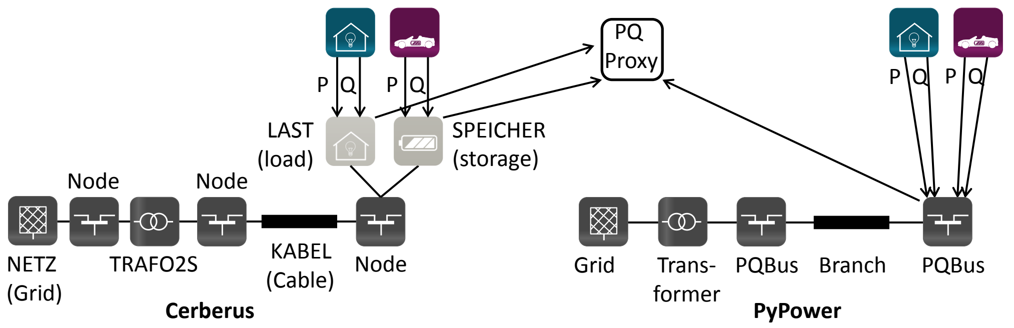


Figure 12.10: Composing PyPower/CERBERUS and other load/generator entities

CIM-based taxonomy

Figure 12.11 shows the proposed taxonomy, based on the types that were discussed above and extended by classes and inheritance relations used in the IEC 61970 CIM (see

Chapter 2.4) *Wires* package. Finally, an AbstractEntityType *Environment* is proposed to represent environmental entities, such as entities representing temperatures or solar irradiance values for distinct geographical regions.

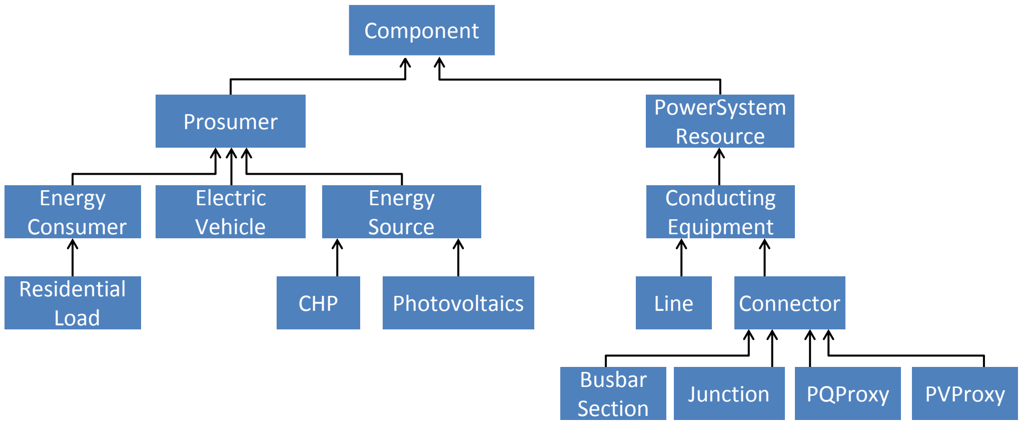


Figure 12.11: Proposed taxonomy of AbstractEntityTypes

12.3.2 Data Definitions

Besides the AbstractEntityTypes discussed above, DataDefinitions are the major elements used to check if two Ports are compatible and thus two EntityTypes can be composed. The creation of these DataDefinitions follows two simple rules:

1. For physical data flows (e.g. power, temperature, etc.) simple data types (scalars) should be used to ease the implementation of generic simulator adapters (e.g. see CSV and FMI integration in Section 12.2.1.3 and 12.2.2.3, respectively).

2. In case of informational flows (i.e. commands sent from agents to a simulated entity) complex data types may be defined if required. In this case, structures defined in the IEC 61970 CIM should be used wherever possible to increase understandability and ease the implementation of standard compliant mappings on the control layer (see 10.5).

As the power grid is the central element of Smart Grid scenarios (which connects almost all other objects), capturing its data flow semantics in such a way that it can be composed in different ways is a central challenge. Two important aspects have to be considered. First, it should be possible to compose different power grid models hierarchically (e.g. see Figure 3.2) to match the natural structure of the power system. Second, power flow analysis requires to distinguish between different types of loads and generators, namely P-V and P-Q resources as already discussed above.

Starting with the latter aspect in a bottom-up fashion, the DataDefinitions for connecting entities representing P-U/P-Q types have to be defined. For a simplified single-phase representation (assuming a 3-phase balanced system) the following DataDefinitions *VoltageMagnitude* and *Re/ActivePowerSinglePhase* are proposed as shown in Table 12.1. Negative values of the power definitions indicate a feed-in in case of producer entities. A

DataDefinition *VoltageLevel* is proposed and added as domain-specific constraint to the active and reactive power flows to allow checking if composed entities have compatible voltage limits. It is defined as enumeration involving commonly used voltages occurring in low and medium voltage grids. The second aspect, hierarchical compositions of power grid models, can be covered with these DataDefinitions as well, by exposing the slack bus entity of the corresponding grid as P-Q prosumer entity.

As discussed in Chapter 2.3.3, the goal of stationary power flow analysis is to find all branch currents and all nodal voltages and their angles. To retrieve this information from the node and branch entities of the used power flow model (for logging and later analysis or for accessing them during run-time via the ControlAPI), the DataDefinitions *VoltageAngle*, *Current* and *CurrentLimit* are proposed. The latter is intended to be used as to describe the current limit of a branch (static data).

Table 12.1: Details of proposed DataDefinitions

Name	Type	Unit	Limit
VoltageMagnitude	float	V	[0,∞]
ActivePowerSinglePhase	float	W	-
ReactivePowerSinglePhase	float	VAr	-
VoltageLevel	float	kV	{.4, .6, 3, 6, 10, 15, 20, 30}
VoltageAngle	float	degree	[-90.0, 90.0]
Current	float	A	[.0, ∞]
CurrentLimit	float	A	[.0, ∞]
PositionPoint	float	deg	[-180.0, 180.0]
	float	deg	[-90.0, 90.0]
Area	PositionPoint[]	-	[3, ∞]

To account for scenarios involving geo-located power grid topologies and environmental models with distinct regions, the CIM type *PositionPoint* is proposed to indicate locations of nodes in the power grid. It defines a position in terms of WGS84[9] coordinates. To define the influence area for entities of environmental models, an *Area* type being an array of 3 or more PositionPoints is proposed.

As mentioned above, the complete model can be found in Appendix H. It involves some more DataDefinitions to capture storage status and sizes of hydro and chemical storages and several other definitions, e.g. to describe operational limits of transformers. In the CIM, a *type* attribute of the operational limit classes is used to define whether a limit is an upper or a lower limit (high/low). However, as such an attribute value would only be available at run-time, precise semantics cannot be ensured at design-time. Therefore, the reference data model proposed here uses two limit definitions using a HIGH and LOW postfix for each limit type (e.g. ApparentPowerLimit_HIGH). Derived from the CIM class *IrregularIntervalSchedule*, the DataDefinition *StorageSchedule* has been added which is used in the MoSL description of the EVSim, for example, to allow sending schedules to the electric vehicles. To support entities with a more detailed three-phased (unbalanced) power flow, the types Re/ActivePower3Phase have been added, with the data type being an array with three elements.

[9] `http://en.wikipedia.org/wiki/World_Geodetic_System` (accessed 24.06.2013)

12.3.3 User-Defined Functions and Mappings

To allow geo-location specific mappings, a user-defined function *areaContainsPosition* is proposed that takes an Area and a PositionPoint as input and returns *true* if the position point is within the given Area. In Section 12.4.2 the usage of this function will be demonstrated. Also, a mapping function *PhaseMapping3_1* has been specified which allows to connect consumer/producer entities with detailed three-phased power flows to a simplified-single-phased grid model, as discussed in Chapter 3.7.

12.4 Scenario Modeling & Composition

RQ 3 – How can Smart Grid scenarios be described in a formal way that allows to compose them automatically using available simulators and their simulation models?

This section deals with the evaluation of the scenario and composition layer of the mosaik concept. Requirement [**R 2** – Scenario Specification Mechanism] is met by the scenario metamodel as a whole. To show that the other requirements have not only been considered during conception (as discussed in Chapters 8.6 and 9.7) but are also met by the mosaik implementation, this section specifies and actually simulates a number of artificial scenarios. Finally, it is described how the scenarios for the two case studies can be modeled to prove that the approach can not only be applied to theoretic scenarios but be used in actual projects. Again, derived from the requirements that influence the layers in question, a number of evaluation criteria can be identified (arrows indicate a section in which the criterion is evaluated):

R 4 – Scalable Scenario Definition Can large-scale scenarios be described without many lines of MoSL specification? ► 12.4.6.2

R 5 – Entity Dependencies Can entities be composed by considering existing entity connections? ► 12.4.1

R 6 – Cyclic Data Flow Support Is the implementation able to cope with scenarios that include cyclic data flows? ► 12.4.3

R 7 – Attribute Based Composition Can entities be composed based on their static attributes? ► 12.4.2, 12.4.6.1

R 14 – Moving DER Can the implementation cope with the dynamic conditions used to specify moving resources (EVs)? ► 12.4.1

R 16 – Variable Entity Quantities/Entity Specific Inputs/Outputs Can the implementation handle models with different EntityTypes and variable numbers of entities? ► 12.4.1

R 19 – Different Temporal Resolutions Does the implementation support simulators with different step sizes? ► 12.5.1[10]

[10]To avoid redundancies, this feature is demonstrated as part of the control layer evaluation in Section 12.5.

R 25 – Multi-Resolution Composition Is it possible to compose models with different abstraction levels? ► 12.4.2

R 26 – Data Logging, R 27 – Logging of Static Data Is all dynamic and static data of the entities logged? ► 12.4.2

R 28 – Execution on Multiple Servers Can a composition involve simulators running on different servers? ► 12.4.1

R 30 – Hierarchical Scenario Modeling Do the hierarchical scenario modeling concepts work as intended? ► 12.4.1 (CompositeModels), 12.4.6.2 (CompositeModels + CompositeEntities)

R 31 – Randomized parameter values Does the mechanism for parameterizing model instances in a random fashion work as intended? ► 12.4.1

12.4.1 Example Scenario

This section shows how the example scenario introduced in Figure 8.6 can be composed out of the simulators that have been integrated into the mosaik platform as shown in Chapter 12.2. The corresponding MoSL files containing the scenario definition can be found in Appendix J.1. According to the rules defined in Section 8.4, each node has a residential load and 75% of the nodes have a PV system. EVs are created for 75% of the nodes that have a PV system. The scenario is composed out of four different simulators, namely GridMockSim, EVSim, PVSim as well as the CSVSim for integrating timer series based residential load profiles from the Grid Surfer project, as discussed in Chapter 3.5.4. This scenario can be used to evaluate a number of the criteria defined above:

Entity Dependencies (topological conditions) The rules for the example scenario (see 8.4) require that EVs are only connected to those buses that have a PV system connected to it. In the scenario description of the LV grid (see LVexample.mosl in Appendix J.1) this has been accounted for by adding a static condition using an EntityCountOperand (see Chapter 8.5.4.1) to determine the number of PV systems connected to a bus. Figure 12.12 shows the resulting composition of the example scenario, automatically visualized from the simulation results using a small Python script and the open-source graph visualization tool Gephi[11]. It can be seen that EVs only occur at nodes (black circles) that also have a PV entity connected to it. Hence, requirement [**R 5** – Entity Dependencies] is met.

Moving DER (dynamic conditions) In addition to the static condition, the ConnectionRule for the EVs also specifies a dynamic condition to achieve that the resulting connections are only active if the EVs location is *Home*. Figure 12.13 shows that this mechanism works as intended using the example of two EVs located in one of the LV grids. The rectangular shapes of the right (secondary) y-axis indicate the locations of the two vehicles. The lines belonging to the left (primary) y-axis indicate the load

[11] `https://gephi.org` (accessed 09 July 2013)

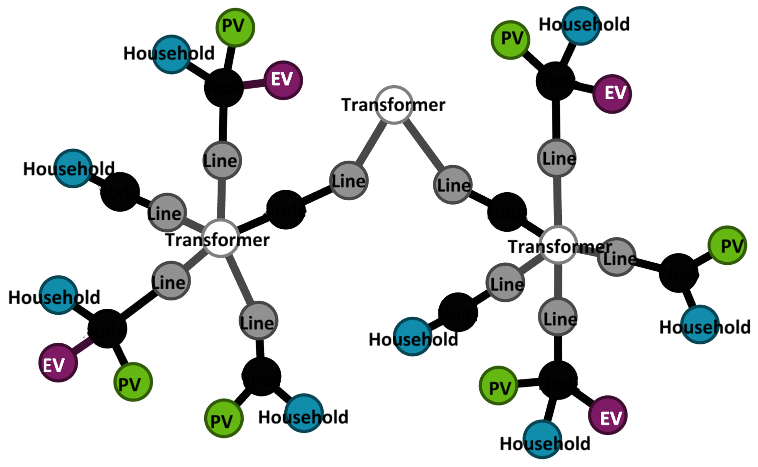


Figure 12.12: Entity relations of the example scenario generated from simulation results

the EVs draw from the power grid and the overall load at the transformer. From the latter, the PV feed-in and residential loads have been subtracted to allow focusing on the EVs. At t = 160, EV1 arrives at work (1) and starts charging. The load curve of EV1 increases to 2 kW (2). However, as the dynamic condition is not fulfilled, the load of the transformer remains unchanged, as it is the transformer of the vehicles home location (the work location is not part of the scenario). At t = 295 EV2 arrives back home and starts charging (3). The transformer load rises as expected. Finally, at t = 370, EV1 also arrives at home and charges as expected. Hence, requirement [**R 14** – Moving DER] is met.

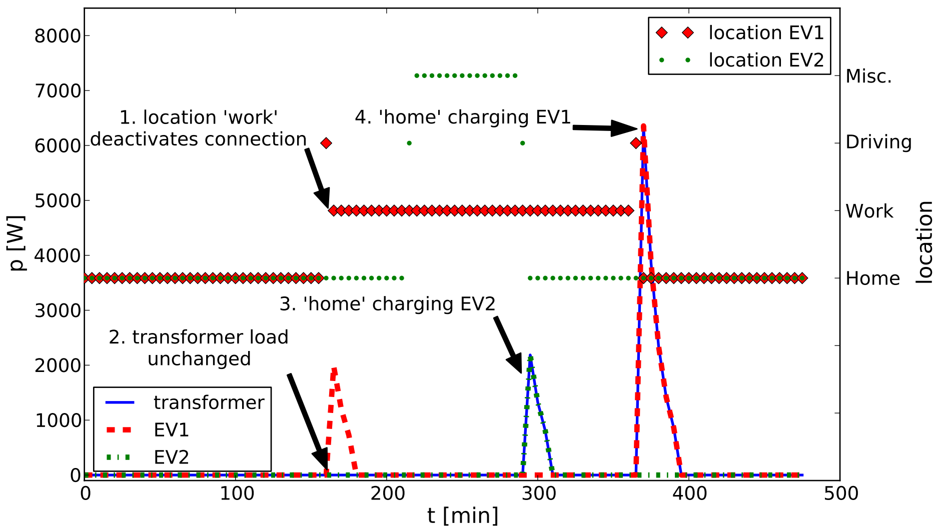


Figure 12.13: Dynamic activation of connection rules for EV charging

Variable entity quantities/types The GridMockSim introduced above allows to specify the number of nodes the internal power grid model contains. Also, the GridMockSim contains a number of different EntityTypes (lines, branches and a transformer). The example in this section shows that mosaik can handle such models and hence, Requirement [**R 16** – Variable Entity Quantities/Entity Specific Inputs/Outputs] is met.

Execution on multiple servers As the PVSim is running on a different server (see Section 12.4.5) this scenario shows that Requirement [**R 28** – Execution on Multiple Servers] is met.

Hierarchical Modeling The example also shows that the concept of CompositeModels works as intended, as the two LV grids of the example are instances of a single CompositeModel *LVexample1* defined in the file *LVexample.mosl* (see Appendix J.1). Figure 12.12 shows that the CompositeModel has been instantiated and composed as intended.

Random Parameter Values The maximum peak power current (IMPP, see [Ger07, p.58]) of the PV model parameter set has not been set to a constant value but is sampled from a uniform probability distribution (roulette) in a range from four to nine Amperes. In combination with the default maximum peak power voltage (UMPP) of the PVSim (23.80 V) and the number of strings and modules per string, the maximum power that the PV system generates is determined. To verify that the random parameter sampling is working properly, Figure 12.14 shows the PV feed-in curves for the six PV systems of the example scenario and for a single day. The diagram has again been generated from the simulation logfile. As the current model does not involve cloud transients, the curves are almost identical, but the different amplitudes prove that different IMPP values have indeed been sampled.

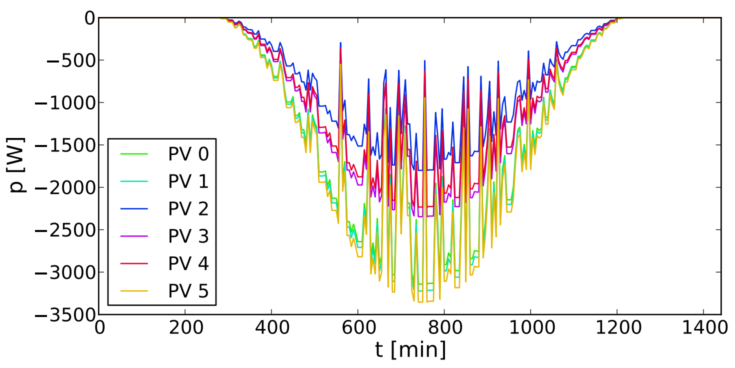


Figure 12.14: Random PV model parametrization

12.4.2 Cloud Transient Scenario

In this section the scenario introduced in Section 3.2.3 will be simulated with mosaik and it is shown that the mechanism of UserDefinedFunctions (see Chapter 8.5.5) works properly in combination with static entity data and thus meets [**R 7** – Attribute Based Composition]. The complete MoSL definition of this scenario can be found in Appendix J.2. Listing 12.1 shows the physical topology part of the definition (omitting the parameter set definition section). The scenario uses the GridMockSim again, a simple sine-based PV model with power output linear to solar irradiance and a cloud transient model that provides irradiance data. These can be found in Appendix I.3 and I.10 and were created for this demonstration.

Listing 12.1: MoSL specification of the cloud transient scenario

```
physical topology
  //Create entity sets...
  grid = 1 gridSim.lvgrid
  pvs  = 6 pvSim.pv
  cells = 1 cloudSim.cloud

  //Connect inverters to power grid
  connect 1 pvs.Inverter to grid.Node mapped by PhaseMapping3_1
  //Connect panel to irradiance model
  connect 0..* pvs.Panel to cells.Cell where area_contains_position(
    cells.Cell.Area, pvs.Inverter--grid.Node.PositionPoint)
end physical topology
```

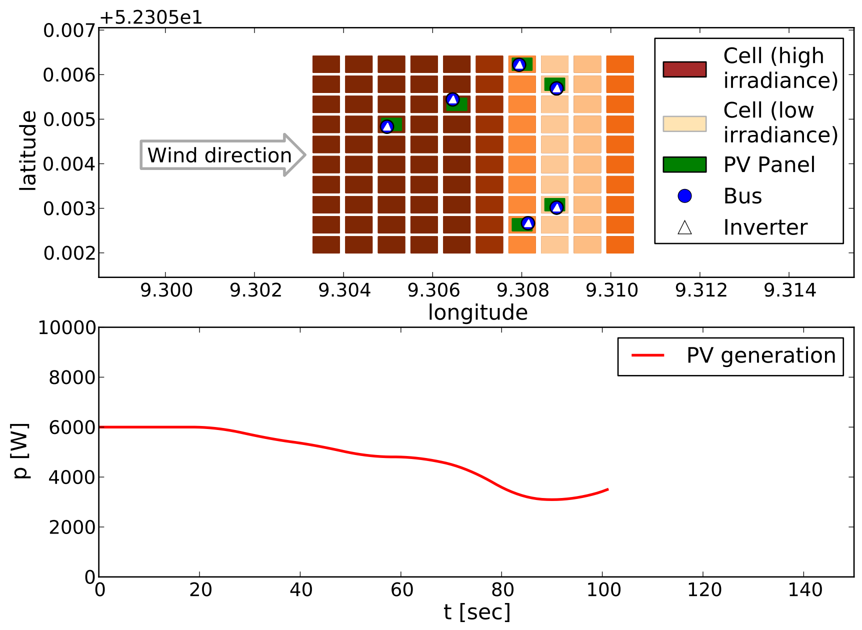


Figure 12.15: Visualization of a cloud transient PV scenario

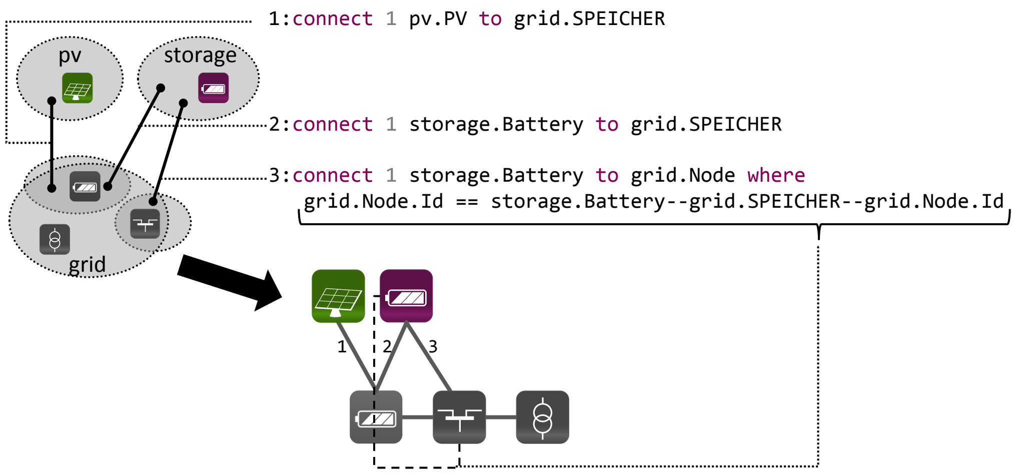
Figure 12.15 shows a graphic that was automatically generated from the log file of the simulation run after the simulation had finished. It includes static data (position information) as well as dynamic outputs from the different Cell and PV entities, showing that Requirements [**R 26** – Data Logging] and [**R 27** – Logging of Static Data] are met. Circles represent the buses of the power grid, triangles represent the inverters of the PV systems. Their positions have been determined by following the connection to the buses. The green (smaller) rectangles indicate cells that contain a PV panel. The figure shows that the UserDefinedFunction based assignment of PV panels to cells of the cloud model works well as all panels are located in a cell covering the position point of the bus to which the respective inverter is connected to. As the user function represents a join predicate, the combinatoric complexity can increase rapidly as already discussed. Section 12.4.5 shows some actual measurements for such a scenario involving more PV entities and a larger number of cells for the cloud model. This scenario also demonstrates that Requirement [**R 25** – Multi-Resolution Composition] is met, as the inverter of the PV model and the nodes of the power grid use DataDefinitions that represent power flows in different level of detail.

12.4.3 Grid Friendly Appliances Scenario

To demonstrate that Requirement [**R 6** – Cyclic Data Flow Support] is met by the implementation, a scenario as shown in Figure 3.4 has been created. The related MoSL description can be found in Appendix J.3. It uses the COTS power flow simulator CERBERUS and a fictive model of a storage that charges/discharges depending on the nodal voltage using a simple formula which determines the charging/discharging power linearly to the voltage difference of the node and an upper/lower threshold:

$$P(U) = \begin{cases} pf * (U - U_{lower}) & \text{discharging if } U \leq U_{lower}, \\ pf * (U - U_{upper}) & \text{charging if } U \geq U_{upper}, \\ 0 & \text{idle otherwise.} \end{cases}$$

The coefficient *pf* is the power factor indicating how many Watts the charging/discharging power changes per Volt deviation. In the given scenario the power factor is set to $400W/V$ and the upper and lower voltage limit is set to 235 and 225, respectively. The formula automatically yields negative values in case of discharging, thus matching the power flow encoding assumed by mosaik.



```
1:connect 1 pv.PV to grid.SPEICHER
2:connect 1 storage.Battery to grid.SPEICHER
3:connect 1 storage.Battery to grid.Node where
   grid.Node.Id == storage.Battery--grid.SPEICHER--grid.Node.Id
```

Figure 12.16: Scenario modeling approach for a voltage-controlled storage

To connect the Battery entity to the power grid, a dynamic condition using an EntityPath (see Chapter 8.5.4.1) is necessary, as the CERBERUS grid model (other than PyPower) requires the storage to connect to two different entities. The STORAGE entity receives the power data and the related Node entity offers the required voltage information. The Battery entity is composable in this case, as the voltage data flow and the power data flow have been defined in different ports. Otherwise, not all mandatory flows of a single port could be satisfied by the entities of CERBERUS. Although this approach (using an EntityPath) is not required in this example (as the power grid only has one node) this shows how the Battery could be deployed in a scenario using a real-world grid topology. Figure 12.17 shows that the simulation has been performed as intended. When PV feed-in (negative values) increases over the day, the voltage raises. At $t = 435$, the Voltage reaches the defined threshold of 235 Volts. As a delay has been specified from the Node to the Battery to resolve the cyclic data flow (see *vctrlstorage.study.mosl* in Appendix J.3), the Battery starts charging at $t = 436$. The enlarged part shows this delayed behavior for a different time point which fits better for illustrative purposes. To make things more interesting, the PV model steps in 15 minute intervals whereas the Battery and grid model steps every minute, causing nice oscillations, illustrating the feedback between these systems.

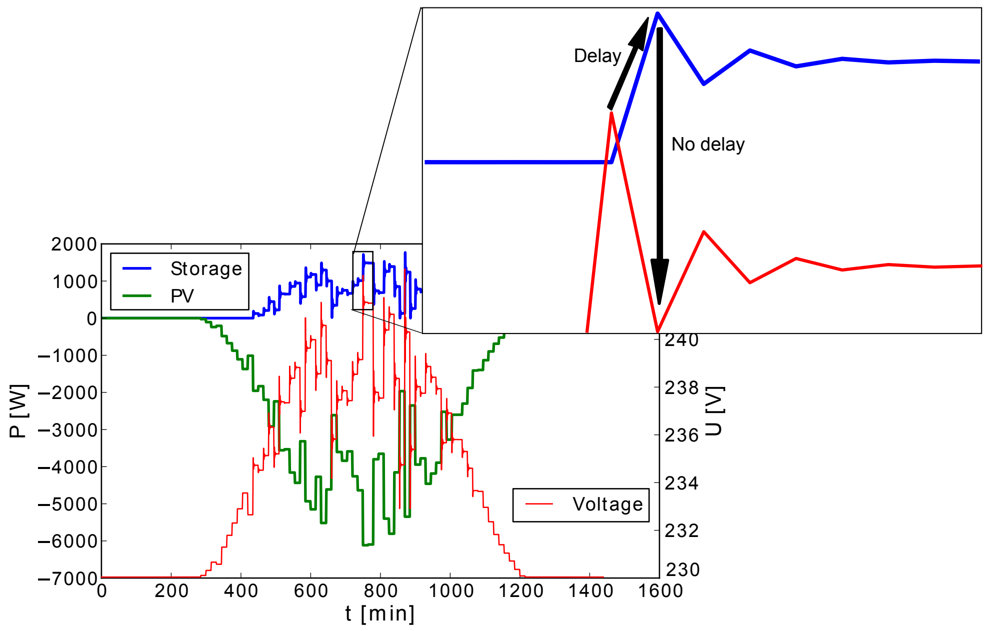


Figure 12.17: Visualization of the voltage controller storage scenario

12.4.4 GridLab-D

To demonstrate the applicability of the scenario specification concept presented in Chapter 8 and the chosen implementation design presented in Chapter 11.3, this section shows how the MoSL DSL can be used to define scenarios for the execution with GridLab-D. This use case also shows that MoSL can be used to compose simulation models within a single simulator rather than combining numerous simulators. However, this section does not cover the complete GridLab-D functionality but only a small example showing that the process works in principle.

As the considered objects in the GLM files are connected via parent-child relations, the reference data model is very simple, just containing a DataDefinition representing such parent-child relations. For this example it is assumed that the power grid is not modeled in MoSL but rather included as a separate GLM file. Therefore, this single DataDefinition is sufficient. What elements allow what parents, is governed via the AbstractEntityTypes taxonomy. Appendix K.1 shows the resulting domain model. Next, the simulator description for GridLab-D has to be created. It is shown in Appendix K.2. It includes the models and their parameters and entities. As GridLab-D does not distinguish between models and entities, all models have a single entity, except for the grid model which represents the GLM file that contains the power grid and the entities are the models included in this file (e.g. transformers, nodes, lines). To test if this basically works, a composition representing a residential load comprised of a house with a waterheater (see Appendix K.3) has been created, based on the example file *residential_loads.glm* shipped with the GridLab-D installation. Two of these houses are instantiated and connected to a simple test grid (also based on the example file). This scenario is shown in Appendix K.4.

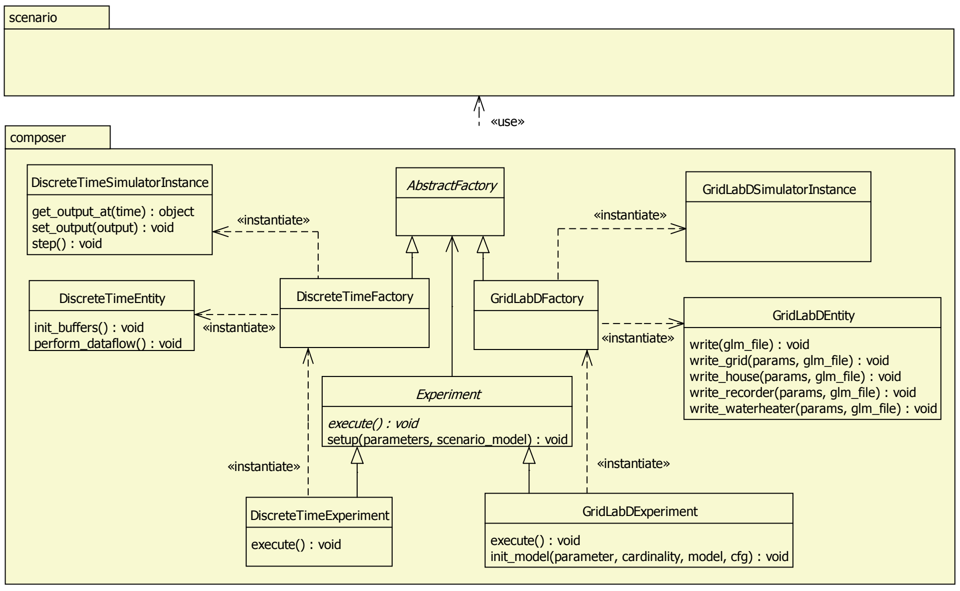

scenario
«use»
composer
DiscreteTimeSimulatorInstance
get_output_at(time) : object
set_output(output) : void
step() : void
AbstractFactory
GridLabDSimulatorInstance
«instantiate»
«instantiate»
DiscreteTimeEntity
init_buffers() : void
perform_dataflow() : void
DiscreteTimeFactory
GridLabDFactory
GridLabDEntity
write(glm_file) : void
write_grid(params, glm_file) : void
write_house(params, glm_file) : void
write_recorder(params, glm_file) : void
write_waterheater(params, glm_file) : void
«instantiate»
«instantiate»
Experiment
execute() : void
setup(parameters, scenario_model) : void
«instantiate»
«instantiate»
DiscreteTimeExperiment
execute() : void
GridLabDExperiment
execute() : void
init_model(parameter, cardinality, model, cfg) : void

Figure 12.18: Extension to the composition layer to create GLM-files for GridLab-D

Figure 12.18 extends the diagram shown in Figure 11.4 by GridLab-D specific subclasses for the composition package. The *setup* procedure which analyzes the scenario model and comes up with a list of required simulator processes and model configurations (according to Algorithm 2) is identical with the one for composing models of different simulators. However, the *execute* procedure is substantially different. Therefore, the *Experiment* class has been made abstract and specific subclasses have been created that implement the corresponding execute procedure. While the original execute procedure started the simulators, initialized their models and thus retrieved a list of available entities, the GridLab-D execute procedure simply calls the *init_model* procedure without instantiating any simulator process. This method mimics the *init* call of the SimAPI and as GridLab-D does not distinguish between models and entities, this method can simply assume one entity per model instance. The EntityType that is to be returned for each model can be inferred from the simulator description that is part of the scenario model and therefore this method can be implemented in a generic fashion. This means that more GridLab-D model types can be added by just extending the MoSL files without having to touch the implementation. The current implementation assumes that an existing GLM file containing the power grid is used as power grid model. In case of a grid model, the *init_model* procedure scans this file for the contained models (=entities) and returns their names as IDs.

Next, a new GLM file is being created and the *write...()* procedure is called for each EntityInstance which appends the required GLM statements to describe the corresponding object type to the file. As GLM does not support forward references, the EntityInstances are iterated in a topologically sorted fashion, following the data flow directions. The sorting can easily be done by invoking the *topological_sort()* method offered by the NetworkX library (see Chapter 11.3). It is invoked on the entity graph that is internally managed by the classes of the scenario package. Finally, the generated GLM file (see Appendix K.5) is passed as argument to the GridLab-D executable which performs the simulation run and returns the results of the recorder objects (if specified). Figure 12.19 shows a plot for the example scenario discussed above.

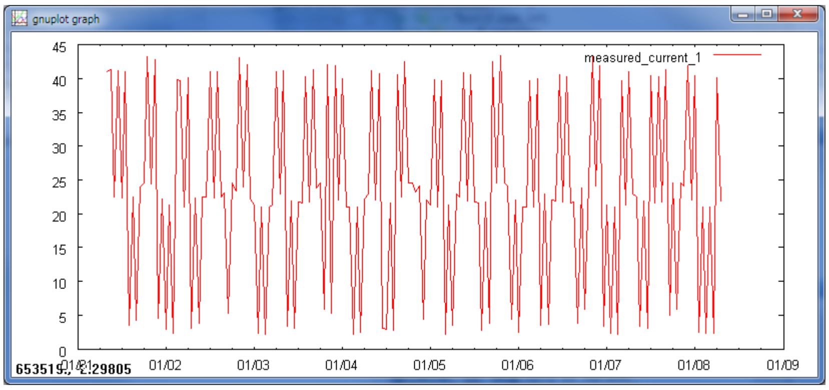


Figure 12.19: Plot generated by GridLab-D for the presented example (current of 1 phase)

12.4.5 Algorithmic Complexity

In Chapter 9 a number of algorithms was presented that are used to analyze the provided scenario model, instantiate the simulators and interpret the ConnectionRules. Figure 9.5 showed the related activities that have to be performed by the composition layer. As the mosaik scenario modeling approach has to be scalable, it must not only offer a compact way to describe large-scale scenarios but also be able to actually compose these scenarios in a reasonable amount of time. Therefore it is important that the above algorithms perform linearly, i.e. with a complexity of O(n). An exception is the CSP solving algorithm, for which a benchmark has been shown in Figure 9.10 and the evaluation of the join predicates which can only be solved in O(n^2) as discussed below.

Design of the experiment

Although not analyzed yet, the algorithms are assumed to perform linearly. To prove this assumption, the actual times that the mosaik implementation needs for performing a composition have been measured. There are two factors that influence the performance of the composition procedure: the number of ConnectionRules to process and the number of entities in the EntitySets that the rules have to operate on. As discussed in

Table 12.2: “One factor at a time” setup for the benchmark experiment

	Factor	
Run	numOfNodesPerLV	numOfLVGrids
1	100	1
2	400	1
3	700	1
4	1000	1
5	1300	1
6	1600	1
7	1900	1
8	100	10
9	100	20
10	100	30
11	100	40
12	100	50
13	100	60
14	100	70

Chapter 8.6, the ConnectionRules have no direct dependencies. But as they add edges to the topology graph (i.e. connections between entities which resolve to specific data flows in the final stage of the composition) they may influence the result of a dynamic condition if it uses an EntityCountOperand. However, this only changes the composition result but not the algorithmic complexity of the ConnectionRule as the dynamic condition has to be evaluated in any case. Therefore it is not required to perform a benchmark for different ConnectionRules, but it is sufficient to let the same ConnectionRule occur multiple times. To achieve this, the benchmark is based on the well known example scenario already used above (see Appendix J.1). The CompositeModel defining the LV grid does not use a constant value to specify the number of nodes in the grid but uses a parameter *numOfNodesPerLV* defined by the CompositeModel. Similarly, the CompositeModel defining the MV grid defines a parameter *numOfLVGrids* to specify the number of LV grid instances to create. The number of ConnectionRules directly depends on this parameter. As each LV grid composition contains three ConnectionRules there are *numOfLVGrids* ·3 ConnectionRules plus the one connecting the MV grid to the EntitySet of LV grids. In the above discussion of the scenario the default values were used. For performing the benchmark, these parameters are varied as shown in Table 12.2.

These values were chosen as they are equally spaced and thus allow to easily recognize linear behavior and as they are in a realistic magnitude for real-world scenarios. An exception is the number of nodes per LV grid which may not be above 400 in real grids but has been further increased to have enough samples for judging th algorithmic complexity. This experimental setup is called “one variable at a time” [Ant03] or “one factor at a time” [Kap08], as only one factor is varied while the other is fixed. Compared to a more complex, full factorial design, this approach does not allow to identify interactions[12] among factors [Nat05]. However, this is not the objective of

[12] “You have an interaction whenever the effect of one independent variable depends on the level of the

this benchmark experiment, as the interaction between the number of LV grids and the number of nodes in an LV grid is known. Doubling the number of nodes has a different effect if there are more LV grids specified, as for each LV grid composition the double amount of nodes has to be connected. This is a linear dependency. Rather, the objective of the benchmark experiment is to prove the assumption that the algorithms for creating the composition scale linear with the number of entities in the respective EntitySets. This can be shown by runs 1 to 7 of the experiment. Runs 8 - 14 can be used to show that there are no dependencies between the ConnectionRules of the different LV grids.

Due to the use of parameters, the MoSL files defining the LV and MV grids can remain unchanged and merely a new file defining the SimulationStudy has to be created as shown in Listing 12.2. The linear value combination feature (see Chapter 8.5.7) is used to automatically execute benchmark experiments for the given values.

Listing 12.2: MoSL of the SimulationStudy for the benchmark experiment

```
study of scenarios.example.MVexample using linear value combination
  start = [2004-07-14 00:00:00]
  stop  = [2004-07-15 00:00:00]

  numOfNodesPerLV = [100, 400, 700, 1000, 1300, 1600, 1900,
                     100, 100, 100,  100,  100,  100,  100]
  numOfLVGrids    = [  1,   1,   1,    1,    1,    1,    1,
                      10,  20,  30,   40,   50,   60,   70]
end
```

Performance metrics

A number of metrics have been integrated into the source code implementing the composition layer by wrapping a timing decorator[13] around relevant function calls. For the sake of clarity, the benchmark results for the different functions have been aggregated into five different metrics enumerated in the order in which they are executed during the composition phase:

1. **Data flow analysis** Required time for the creation of the data flow graph (see Algorithm 1), detecting cyclic data flows (see Chapter 9.4) and determining EntitySets that have to be simulated in separate processes (see Chapter 9.3).

2. **Expand CompositeModel** Required time for the recursive analysis of the CompositeModel specified by the respective SimulationStudy object and for gathering simulator process and model configuration information (see Algorithm 2 and Algorithm 7 in Chapter 9.5.1).

3. **Get static data/relations** Required time for retrieving static data and entity relations from all simulators.

4. **Apply ConnectionRules** Required time for evaluating all ConnectionRules (see Chapter 9.5.3).

other" [Cal05].

[13] *Decorator* design pattern, see Gamma et al. [GHJ94]

5. Create schedule Required time for creating the simulator schedule graph (see Algorithms 4, 5 and 6.

Performance results

Figure 12.20 shows the results for runs 1 to 7 (one LV grid composition with increasing number of nodes/PVs/EVs). The different metrics are plotted with different line width and styles. For each variant five repetitions have been made to reduce the influence of other incidentally occurring tasks performed by the operating systems of the involved servers[14]. The plots represent average values over the five repetitions. The standard deviation (i.e. the distribution of individual values around the mean [Str96]) is shown by using vertical error bars. The thin straight plots have been added for each metric using linear regression with the Python library NumPy[15] to illustrate the perfect linear case.

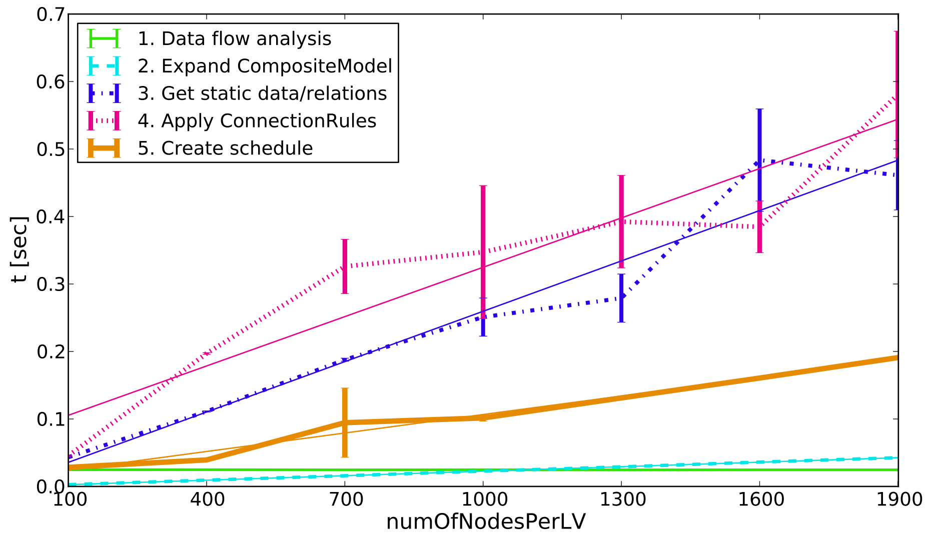


Figure 12.20: Performance for different parts of the composition procedure and an increasing number of nodes and connected entities in an LV grid (Run 1 to 7)

A number of things are noteworthy about the obtained results. First of all it can be seen that the time required for the data flow analysis runs in constant time O(1) as it is performed on the elements of the scenario model itself (as described in Chapter 9.2) which remains constant and thus does not depend on the factor that is varied. Next, all other plots indicate linear O(n) behavior (or at least close to linear). This is especially important for the expansion of the CompositeModels and the application of the ConnectionRules as these involve/operate on the final number of simulated entities. For x=1900 these are one transformer, 1900 lines, 1900 nodes, 1425 PV systems and 1068 EVs as well as three entities that make up the MV grid (=6297 entities). Finally, the absolute time taken for the composition is less than one second for all cases which

[14] The simulation involves two servers. Server 1 (4x AMD Opteron 2.4 GHz/64 GB RAM/Ubuntu 10.04 LTS 64bit) running the mosaik simulation engine and hosting all simulators except for the PV simulator and server 2 (Windows 7 64bit) hosting the PV simulator running on a virtual machine (VM Ware).

[15] `http://glowingpython.blogspot.de/2012/03/linear-regression-with-numpy.html` (accessed 10 July 2013)

is more than satisfying, as according to experience from actual research projects, a usual LV grid only has about 50 to 250 nodes (e.g. see the LV grid topologies in Appendix J.5.2 that will be introduced below). Therefore, a deeper (numerical) analysis of the benchmark results is not pursued.

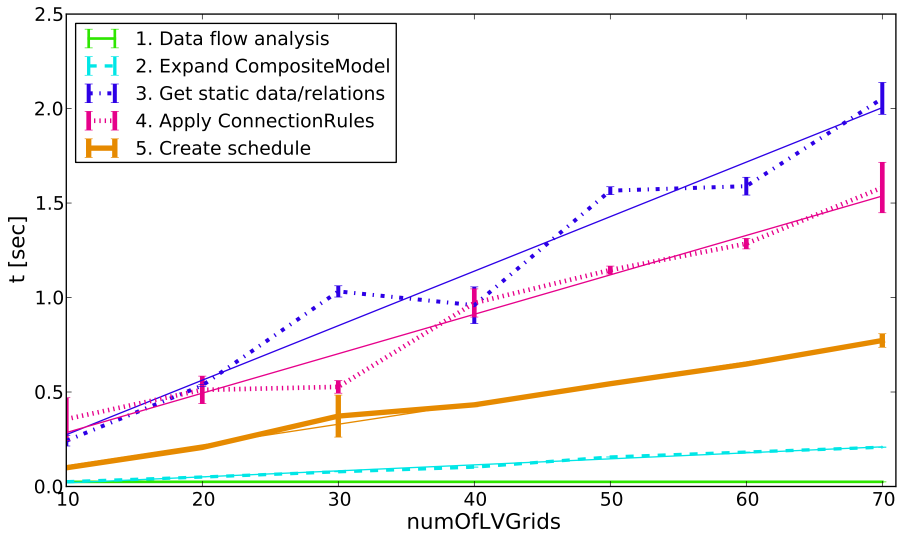


Figure 12.21: Performance for different parts of the composition procedure and an increasing number of LV grid compositions (Run 8 to 14)

Of course, mosaik must also be able to not only simulate a single, large-scale LV grid but also MV grids including a large number of LV grids. Therefore, runs 8 to 14 of the benchmark experiment are increasing the number of LV grid compositions connected to the LV grid while keeping the number of nodes (100), PVs (75) and EVs (56) stable for each grid. Figure 12.21 shows the results for run 8 to 14. For x=7000 the overall scenario involves 70 MV/LV transformers, 7000 LV nodes, 7000 LV lines, 5250 PV systems, 3937 EVs and 141 (70 lines, 70 nodes, 1 transformer) entities that make up the MV grid (=23398). Again, the results indicate a linear behavior, ensuring scalability of the scenario modeling approach also for hierarchical compositions. For both studies, the number of involved simulator processes is constant, as all simulators can handle multiple model instances.

It has to be pointed out that the results shown in Figure 12.20 and Figure 12.21 do not involve the time taken for instantiation and initialization of the simulator processes (including the communication between them and mosaik). As this takes unequally longer due to the involved VM running Matlab for the PV simulation, these times are shown separately in Figure 12.22. Again, a linear complexity can be observed. Also, there is an offset of about 20 seconds that account for the start of Matlab. This startup is not part of the "Instantiate simulators" metric as this only starts the SimAPI adapter for Matlab, which is started on-demand when the init-call is made. The remaining, linear increase of the initialization time is related to simulator specific initialization effort, mostly caused by the EVSim, as will be shown in Figure 12.35 below.

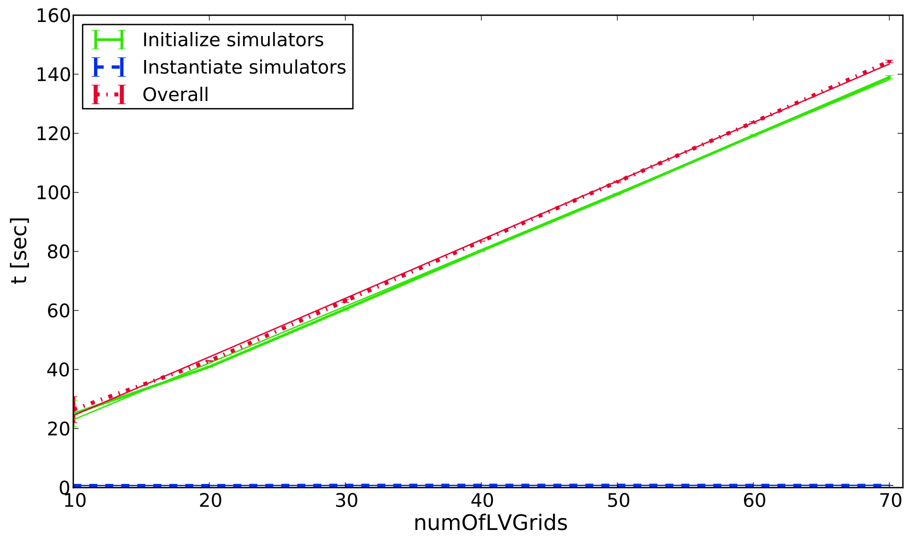


Figure 12.22: Performance for simulator instantiation and initialization and an increasing number of LV grid compositions (Run 8 to 14)

Performance for join predicates

Although it is obvious that scenarios involving a ConnectionRule with a join predicate in the static condition cannot be interpreted in linear time, a benchmark has been performed to find out how the performance actually is (in absolute numbers). Figure 12.23 shows the resulting plot for the cloud transient scenario introduced above (see Section 12.4.2). The values on the x-axis indicate the number of deployed PV systems and the width of the cell array used in the CloudSim. So for $x = 20$, the locations of 20 PV systems have to be joined with a 20 x 20 cell array (=400 cell entities). As indicated by the black dashed line[16], the plot shows a complexity close to $O(n^3)$. This matches the expectation as the number of combinations to test is x PV entities times $x \cdot x$ cell entities and thus x^3.

These results indicate that for larger EntitySets being involved in join predicates the performance will become a problem. However, as shown below this is not an issue for the scenarios of the case studies, as ConnectionRules are specified for individual LV grids and thus the number of entities is relatively small. Nevertheless, the implementation should be extended to issue an information message and possibly some progress information in case of such big joins.

While this combinatoric complexity is intrinsic to the attribute based composition of entities, the static ordering processing order defined in Chapter 9.5.3.2 allows some simple countermeasures that the scenario expert can take to reduce combinatoric complexity. The cloud model, for example, may cover a larger area than the power system that is subject to the analysis. Therefore it is possible to reduce the number of cell entities to only those covering the desired area of investigation to increase composition performance. This can easily be done by extending the static condition of the ConnectionRule. Instead of involving only the UserDefinedFunction that matches

[16] c is a constant factor to match the scale of the benchmark plot

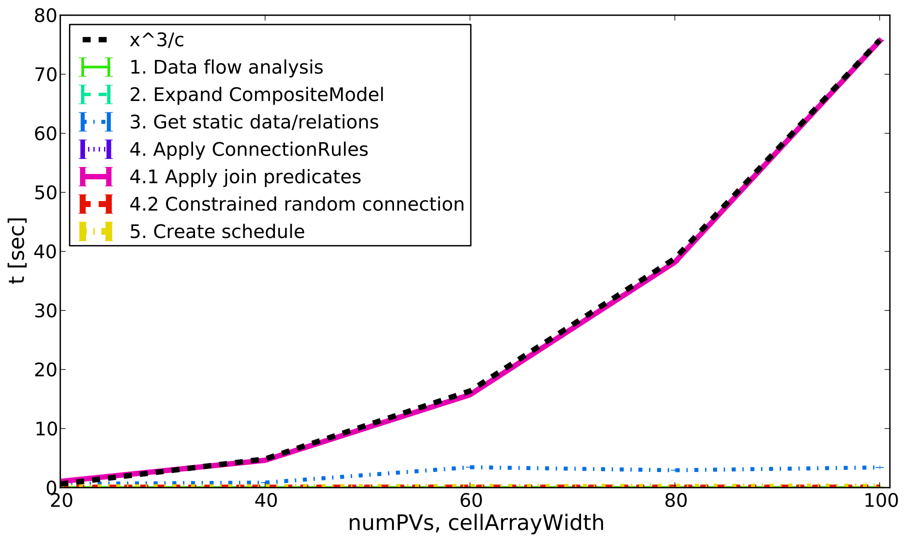


Figure 12.23: Performance for different parts of the composition procedure and a connection rule involving join predicates

cell area and PV position, AND operators are used to add conditions on the static area data of the cells. As these will be identified as selection predicates (involving only the cell entity) they will be evaluated first and therefore reduce the number of cells in question. Figure 12.24 shows this approach.

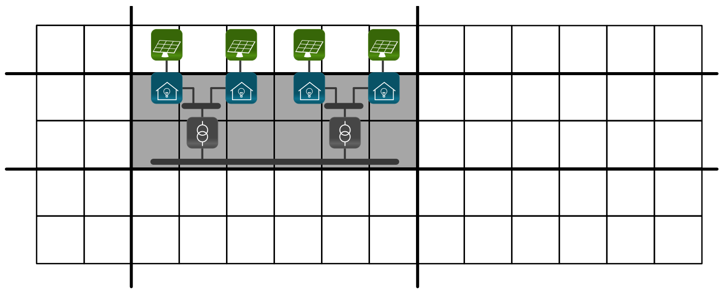


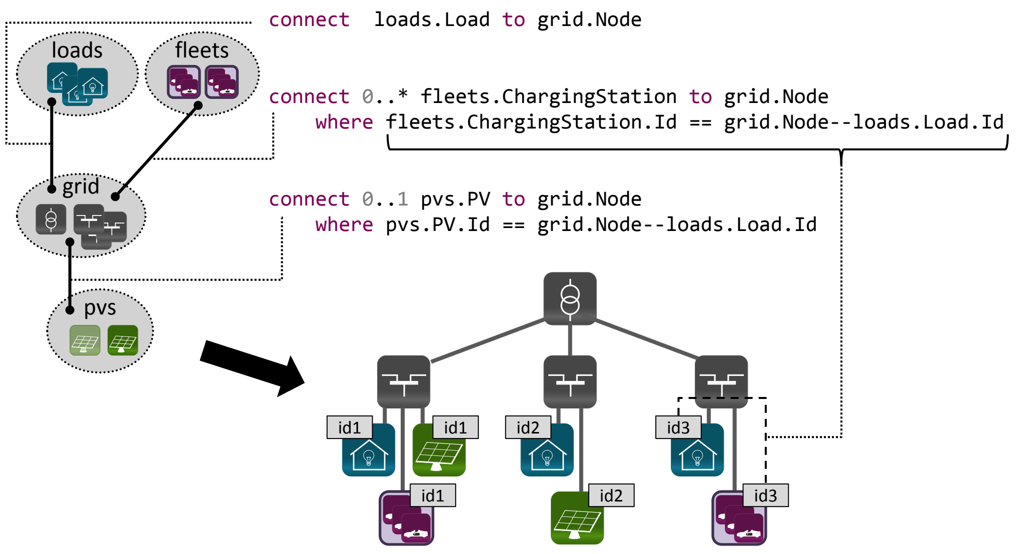
Figure 12.24: Example for reducing combinatoric complexity of join predicates by explicitly restricting the set of possible entities

12.4.6 Case Studies

In the following sections there will be shown how the mosaik scenario modeling concept can be used to define the scenarios to be simulated for the presented case studies.

12.4.6.1 Case Study I: NeMoLand

As already shown in Figure 12.2, the NeMoLand scenarios are comprised of a number of different loads, PV installations, micro CHPs and EV fleets. The values of the time-series for the different loads as well as PV and CHP entities are based on the study of actual areas of the municipality Ganderkesee (Lower Saxony). Therefore, the different objects of the time-series have node IDs that are logical names for their actual locations. Although no specific power grid topology is present, the power grid model (GridMockSim, see Section 12.2.3.1) should contain as many nodes as required for the particular area (rural, commercial, ...) and only entities with the same node ID should be placed together on one node. This allows to easily calculate the energy balance per node, for example, to obtain verifiable statements about the local PV consumption and potential synergies of different consumer and generator types.



```
connect  loads.Load to grid.Node

connect 0..* fleets.ChargingStation to grid.Node
    where fleets.ChargingStation.Id == grid.Node--loads.Load.Id

connect 0..1 pvs.PV to grid.Node
    where pvs.PV.Id == grid.Node--loads.Load.Id
```

Figure 12.25: Scenario modeling approach for NeMoLand

The EntityPath concept introduced in Chapter 8.5.4.1 can be used to conveniently model a scenario in this manner. Figure 12.25 illustrates this for a part of the resulting scenario modeling approach (micro CHP and parameter set definitions are not shown for the sake of clarity) and Appendix J.4 contains the complete NeMoLand scenario MoSL file. As each node has a *Load* entity, these are connected first in a 1:1 fashion. Next, the other EntityTypes can be connected in an arbitrary order. As the node entities of the generic GridMockSim do not “know” about the IDs used in the provided time-series data, the ConnectionRules specified for connecting the other EntityTypes must use the EntityPath concept to navigate to the Load entity of each node and use its ID which is provided as static data as specified in the CSV file. This ID is compared with the ID of the entity that is to be connected and thus it is made sure that a node has only connections to entities that have the same ID as the Load that was initially connected. A manual analysis of the composition (comparing the IDs of the entities connected to the same node) showed that the scenario has been composed correctly. This again shows that Requirement [**R 7** – Attribute Based Composition] is met.

To show that the nodal power balance is calculated correctly, Figure 12.26 shows the load/feed-in curves for the entities of a single node in a commercial grid. It can be seen that the nodal balance (Node sum) matches with the sum of the other plots.

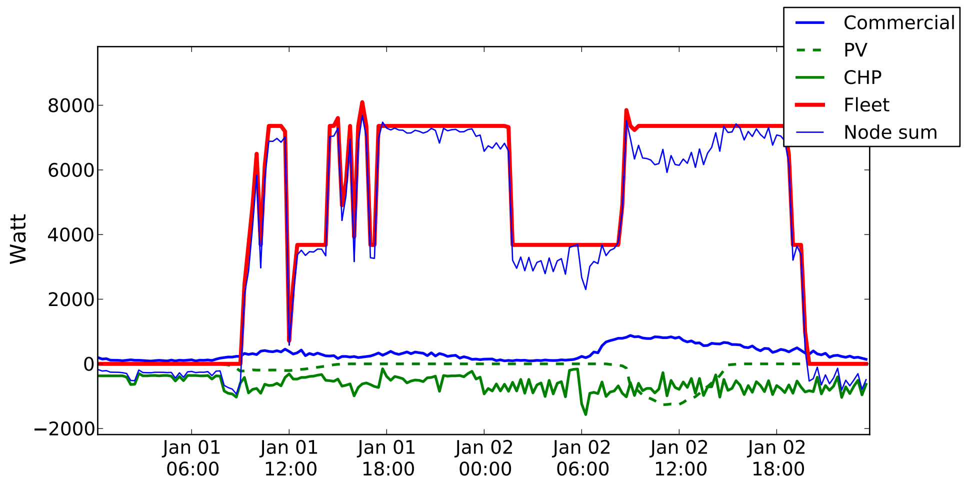


Figure 12.26: Consumption/generation at a node of an EV fleet

Open issues

The relational operators mosaik offers (==, <= and >=) can only operate on two single values and not on a set of values. The evaluation of an EntityPath, however, can result in a number of reachable entities. The current implementation raises an exception in this case, as this situation does not occur in the examples used for the evaluation and has thus been left out to ease implementation. To resolve this issue and allow to defined precisely what the result in case of an EntityPath returning multiple entities will be, quantifier functions could be integrated as another expression type combining two expressions and a relational operator as follows:

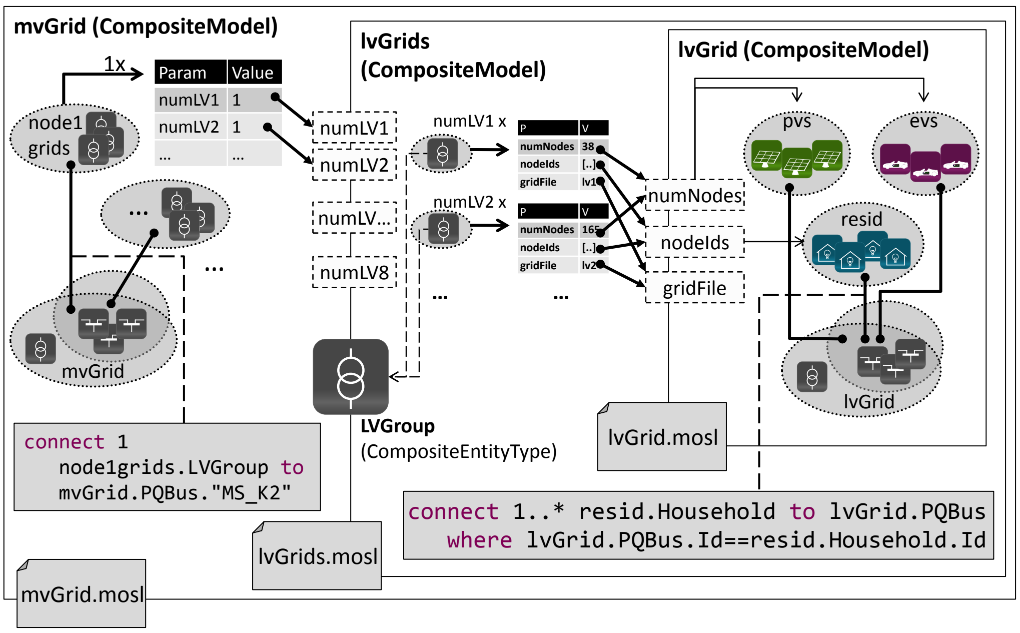
```
any (grid.Node -- loads.Load.Id == 42)
most(grid.Node -- loads.Load.Id == 42)
all (grid.Node -- loads.Load.Id == 42)
<quantifier function>(<expression> <relational-operator> <expression>)
```

12.4.6.2 Case Study II: Smart Nord

The power grid for the Smart Nord scenario could be modeled as large, single grid or each individual LV grid (see Section 12.1.2) and the MV grid could be modeled separately. As mosaik was designed to support the hierarchical composition of power grids, the Smart Nord team decided to go with the latter approach. Having each grid as a single model also increases modularity. This allows to create small test cases using only one individual grid and also allows an easy reuse of the grids in future projects.

LV grid modeling

For each LV grid, a CSV file with load profiles has been provided. Each column in these files has a header indicating to which node of the grid the values belong. However, the values for the different nodes repeat every 67 columns. Especially, as the data will be interpolated to a resolution of 1 minute (see Section 12.1.2), a large amount of redundant data is the result. To avoid this, the team decided to only place the 67 different columns into a single CSV file and create a project specific version of the CSVSim (a branch in the simulator repository) which allows to provide the CSVSim with a list of IDs as configuration parameter. The CSVSim then assigns each ID to a column of the CSV file in a modulo fashion (when more than 67 IDs are passed, assignment starts from the first column again), such that the output of the simulation is similar to a large CSV file with individual column IDs. Similar to the NeMoLand scenario model, the IDs are available as static data and can be used in a ConnectionRule to easily define all specific mappings. The bottom right part of Figure 12.27 shows this ConnectionRule. It is part of the file *lvGrid.mosl* (see Appendix J.5) which specifies a CompositeModel that can be parametrized in such a way that it can be used to define all of the eight LV grid types. This is achieved by allowing to pass the IDs for the residential CSV file, the LV grid file for PyPower and the number of nodes that the grid for this file contains. The latter is used to define a node specific number of entities such as PVs, for example. By using the ComputationMethod that a ParameterReference can specify (see Figure 8.8) it is also possible to specify fractions or multiples of *numNodes* as cardinality for an EntitySet and thus to deploy more or less than *numNodes* entities per node.



```
connect 1
    node1grids.LVGroup to
    mvGrid.PQBus."MS_K2"
```

```
connect 1..* resid.Household to lvGrid.PQBus
    where lvGrid.PQBus.Id==resid.Household.Id
```

Figure 12.27: Scenario modeling approach for Smart Nord

The presented scenario modeling approach implicitly assumes that entities are deployed in this "per node" fashion (e.g. n entities per node entity). If a node of the power grid represents a connection point of a single building, this approach works well for entity types that can be deployed on a "per building" basis, such as PV systems, for example. Although this is the case for the Smart Nord grid (1 node = 1 building), there are many nodes with multiple household load profiles assigned, that is the buildings contain more than one household (i.e. apartment buildings). This case was not considered during the design of the mosaik scenario modeling concept in Chapter 8. The deployment of entities "per household" (such as fridges or EVs) cannot be specified directly in a scenario model. This is because the concept does not offer a means to specify the multiplicity constraint for the ConnectionRules of such entities in a way that incorporates the number of household entities connected to a node, for example.

However, to integrate the fridges into the scenario on a per household basis, a workaround is possible.[17] As a workaround, a number of identical ConnectionRules has been added which also lead to the desired result. There are as many ConnectionRules required as the maximum number of households per node in the overall scenario. In this case, one LV grid type has a node with eight households. Therefore, seven additional ConnectionRules have been specified which connect each node to exactly on fridge when then node has at least one household but less fridges than households connected to it:

```
|lvGrid.PQBus--resid.Household| > 0 AND
|lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
```

Also, the set of possible fridges to connect is narrowed down to those fridges that have not been deployed yet:

```
|fridges.consumer--lvGrid.PQBus| == 0
```

However, this workaround only works if there are one or more entities per household to be deployed. As fridges have a saturation level of 106% [SBP+08] (six of 100 households have two fridges) this is the case. However, for the purpose of this evaluation one fridge per household is assumed to ease validation of the results (simply compare the number of fridges and households per node as shown in Figure 12.29 discussed below). But for the final Smart Nord scenario it may be desirable to additionally deploy a number of additional fridges in a random fashion to reach the factor of 1.06. This can easily be done by increasing the number of fridges in the EntitySet by multiplying *numHouseholds* with a constant factor of 1.06 and add another ConnectionRule at the end of the scenario model which randomly deploys the remaining fridges (without check if there are less fridges than households).

Due to this limitation this workaround cannot be used for the deployment of EVs as these have a share of only 33% even for 2050 scenarios [NPS+10] and the Smart Nord project considers 2030 for the most distant future scenario. So although an average German household has 1.2 vehicles [Ins10a] there will be less than one EV per household. As the final Smart Nord scenario has not been specified, the scenario presented here assumes a single EV per node.

As for other entities (e.g. appliances) a per household deployment is also desirable, this issue should be improved on in the next version of the mosaik implementation. This could be done easily, by allowing to use the EntityCountOperand (see Chapter 8.5.4.1) not only in a dynamic condition but also as parameters for the upper and lower bound of the multiplicity constraint. As these values are required after the dynamic condition has been evaluated (when the resulting connections are made) it is possible to also evaluate the EntityCountOperand to obtain these values. This way it can easily be defined that nodes with 4 households, for example, are allowed to have 4..5 fridges ($5 = 4 \cdot 1.06$,

[17] Figure 12.27 does not show the fridge integration to its complexity. Please refer to *lvGrid.mosl* in Appendix J.5 to follow the explications.

rounded up). When the number of fridge instances is limited to *numHouseholds*·1.06 the desired deployment is obtained.

Finally, the LV grid composition is an example for which the simple wildcard approach of GridLAB-D (see Listing 4.1 in Chapter 4.1.2) is insufficient to easily deploy PVs, EVs or other resources, as the nodes have already been assigned to a connection as soon as the residential loads are connected.

MV grid modeling

The MoSL file *lvGrids.mosl* (see Appendix J.5) which is also shown in the figure is used to define individual parameter sets for each of the 8 LV grid types. Also, it instantiates a specific number of LV grid instances of each type, the number of which can be determined via the *numLV...* parameters.

Finally, the MoSL file *mvGrid.mosl* (see Appendix J.5) defines the MV grid for Smart Nord by instantiating a single PyPower instance with the MV grid topology. For each of the ten MV grid nodes, a single instance of the *lvGrids* composition is created. By parameterizing it with the number of LV grid types the respective MV node requires, the resulting EntitySet (e.g. *node1grids*) can be used to access the entities of all LV grids. To avoid the specification of eight individual ConnectionRules between the LV type specific subsets and the MV node, it can be taken advantage of the CompositeEntityTypes introduced in Chapter 8.5.4. As the *lvGrids* composition defines a CompositeEntityType *LVGroup*, representing a group of LV grids, and all transformers are mapped to this type (indicated by the dashed arrows in Figure 12.27), it is possible to connect this CompositeEntityType to the node of the MV grid with a single ConnectionRule. This is shown in the bottom left of Figure 12.27.

Although the final scenario of the project will be even bigger as soon as the other models are available, these models can be integrated similarly to the residential loads (if they are node specific) or to the EVs (if a random distribution can be assumed). The scenario presented here contains an overall number of 75.343 simulated objects[18] (11.992 buses, 11.981 EVs, 11.981 PVs, 13.381 households, 13.381 fridges, 12.268 branches, 246 transformers and 113 slack nodes). More detailed entity information for the LV grids as well as composition and simulation performance benchmarks are presented in Section 12.6 below.

Scenario validation

Figure 12.28 shows a graph of the MV level that has been automatically generated from the logged simulation data (which also includes the entity relationships). It can be seen that the scenario specification leads to the desired result. The numbers in the nodes indicate the LV grid type (1 to 8) and a manual check has shown that this matches with the overall topology in the Smart Nord scenario specification documents. To show that the LV grids have been composed correctly, similar graphs have been created for the different LV grid types. These can be found in Appendix J.5.2. To get an impression of the overall complexity of the scenario, Appendix J.5.3 shows a similar graph that has

[18] The simulated fridges have been counted as a single object although each fridge model is comprised of a number of different entities as discussed in Section 12.2.1.2.

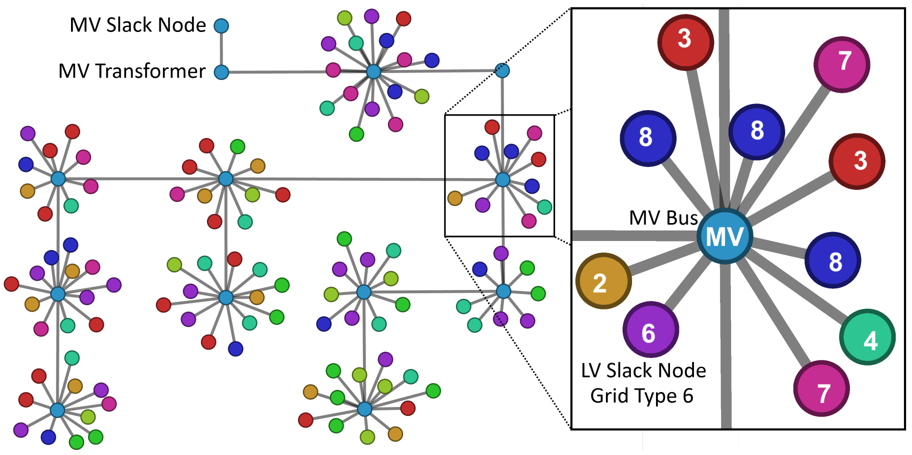


Figure 12.28: Validation of the Smart Nord composition

been generated for 22 LV grids located at four different MV nodes.[19] To validate that the deployment of the fridges works as intended, Figure 12.29 shows a part of an LV grid, also visualized from the simulation results database. It can be seen that each node has the same number of fridges and households. This is the intended result.

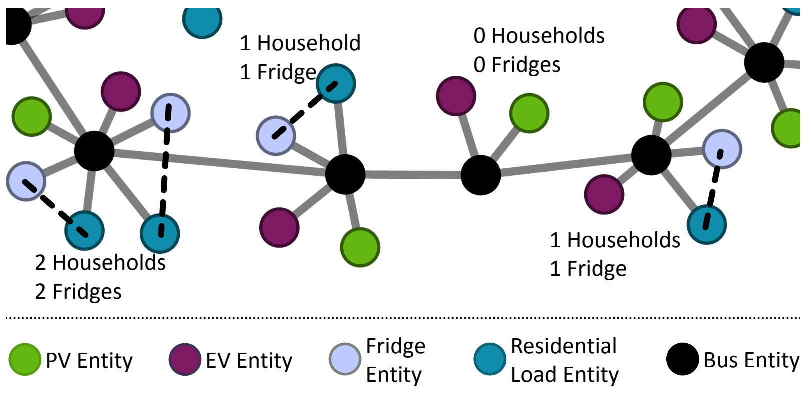


Figure 12.29: Deployment of fridges in the Smart Nord scenario

As the scenario as it is presented in this work only requires a small number of ConnectionRules, this demonstrates that Requirement [**R 4** – Scalable Scenario Definition] could be achieved by the hierarchical scenario modeling concepts introduced in Chapters 8.5.3 and 8.5.4. For MV grids with a larger number of nodes, LV grid types could either be connected randomly or, if a specific assignment is desired, a scenario specific UserDefinedFunction could be defined which is provided with the IDs of the MV node and the LV slack node as well as a reference to a mapping file. The function could then be implemented in such a way that it reads the mappings from the file and checks if the given IDs of the entities for which the function is called are contained in the mapping.

[19] A visualization of the complete set of 112 LV grids cannot be arranged in a visually appealing way.

12.4.7 Entity distribution variants

When a ConnectionRule is evaluated, entities that meet the defined constraints are chosen with uniform probability for each connection to be made (see algorithms in Appendix E). In the following, it is discussed what the implications of this approach are, i.e. how the resulting scenarios look like. Especially, the influence of the multiplicity constraint on the resulting composition and the distribution of the number of entities per node is discussed, as these aspect did not occur in the scenarios discussed above. For a Scenario Expert these are important things to understand in order to specify the scenario model correctly.

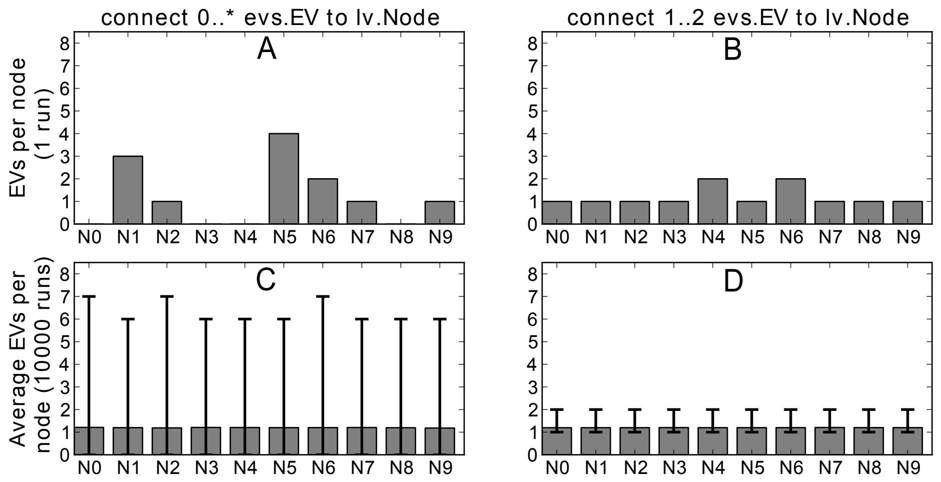


Figure 12.30: Number of EVs per node for different multiplicities of a ConnectionRule (error bars denote minimal and maximal values)

Figure 12.30 A shows how an example for a ConnectionRule with unbounded multiplicity (0..*) that connects twelve EVs to ten nodes of a LV grid[20] (EntitySets have the same name as in the scenario discussed in Section 12.4.1). The plots have been generated by using the actual algorithms of the mosaik implementation. The y axis shows the number of EVs that are connected to a specific node on the x axis (nodes N0 to N9). As the multiplicity constraint is unbounded, nodes with zero as well as up to four EVs occur (theoretically up to twelve are possible). Figure 12.30 C shows the average number of EVs as well as the maximal and minimal number of EVs (using error bars) occurring per node for 10000 runs. The average value of 1.2 EVs per node indicates that the random distribution is indeed uniform as there is no bias towards a specific node. The maximum number of EVs per node over all runs is up to seven. Higher values (up to twelve) do not occur as the small likelihood for a node to be chosen more often requires more runs for such a case to occur.

The right plots of Figure 12.30 show the influence of the multiplicity constraint when it limits the number of EVs per node to one or two. Plot B shows an exemplary distribution of EVs for a single run and plot D shows the average as well as the minimal and maximal

[20] Assuming that a node represents a single household this number of entities was chosen based on the average number of vehicles per household in Germany, which is 1.2 [Ins10a]

number of EVs per node for 10000 runs. The average number of EVs per node is again 1.2, as the number of EVs and nodes is unchanged (12 EVs for 10 nodes). However, the multiplicity constraint works as intended and prevents nodes with zero or more than two vehicles.

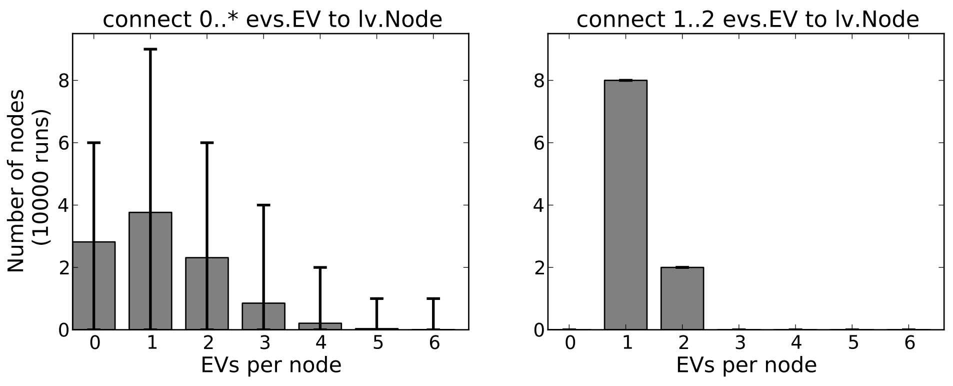


Figure 12.31: Average number of nodes with a certain number of EVs (error bars denote minimal and maximal values)

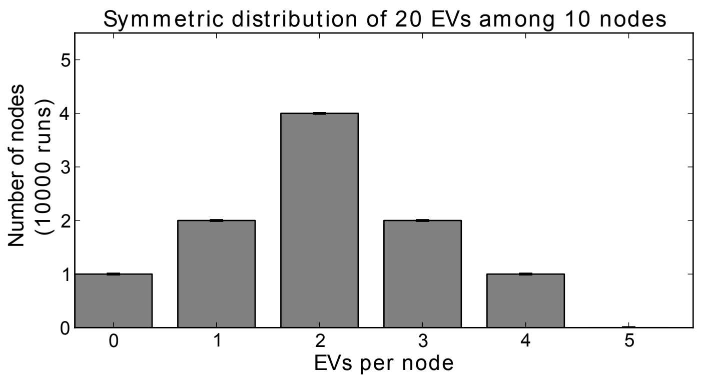


Figure 12.32: Specific distribution of EVs by using multiple ConnectionRules (error bars denote minimal and maximal values)

For the Scenario Expert it is important to understand that although the average number of EVs is identical for all nodes (and thus a random probability for selecting a node is given), the number of EVs per node is not uniformly distributed in the resulting scenarios. This is illustrated by Figure 12.31. The left plot shows the average number of nodes plotted against the number of vehicles for each of these node for 10000 runs and an unbounded multiplicity constraint. The result is a positively skewed distribution, resulting from the likelihood of a node to be selected zero, one or more times. An average composition has about three nodes without vehicles and four nodes with one vehicle and so on. The right plot of Figure 12.31 shows the average number of nodes for the bounded multiplicity case. Here, the error plots indicating the minimal and maximal number of nodes over all runs match the average values as the multiplicity constraint of 1..2 is only satisfied by eight nodes having a single EV and two nodes having two EVs.

More sophisticated means to influence the shape of the plots have not been defined in the scope of this thesis. However, in most cases only a small number of entities (e.g. one or two fridges or one PV system) has to be deployed per node/house/household. Therefore the multiplicity constraint is sufficient to meet these cases. An exception may indeed be EVs of which a household may have up to four. However, in these cases a number of ConnectionRules using the RangeSelector introduced in Chapter 8.5.3 can be used to achieve a specific distribution of EVs, e.g. as governed by official statistical data such as [Ins10a]. To do so, one ConnectionRule is defined per number of entities (e.g. EVs) to connect per node and the RangeSelector is used to select disjunct sets of nodes as shown in Listing 12.3.

Listing 12.3: Using multiple ConnectionRules and RangeSelectors to define specific entity distributions

```
connect 1 evs.EV to lv.Node[1: 3] where |evs.EV--lv.Node| == 0
connect 2 evs.EV to lv.Node[3: 7] where |evs.EV--lv.Node| == 0
connect 3 evs.EV to lv.Node[7: 9] where |evs.EV--lv.Node| == 0
connect 4 evs.EV to lv.Node[9:10] where |evs.EV--lv.Node| == 0
```

As the EntitySets are initially shuffled (and not every time a ConnectionRule is applied) the RangeSelector creates disjunct sets and the nodes that have one, two or more EVs are still randomly chosen (and differ from one simulation run to the other if no random seed is specified for the scenario). Figure 12.32 shows the resulting distribution of EVs among ten nodes. Obviously, to obtain such a distribution, a sufficient number of EVs must be created (20 in this case).

12.5 Control Strategy Evaluation

RQ 4 – How can control strategies be evaluated using the simulated Smart Grid scenario (technically)?

This section deals with the evaluation of the top most layer of the mosaik concept which offers the ControlAPI that allows to use mosaik as simulative test bed for control mechanisms. Again, a number of evaluation criteria can be derived from the requirements that influenced the design of this layer:

R 3 – Control Strategy API Can entity states be retrieved and control commands be issued? ► 12.5.1

R 8 – Physical Topology Access, R 10 – Combined Physical Topology Can control algorithms be implemented that require access to the power grid topology? ► 12.5.2

R 12 – Synchronization of Control Mechanisms Does the synchronization of the control mechanism and the simulated entities work? ► 12.5.1

To prove that mosaik meets these criteria, the Smart Nord scenario is taken as example, as it is more complex than the NeMoLand scenario and the project requires the use of all features of the ControlAPI.

12.5.1 Synchronization and Data Access

To show that reading and setting dynamic entity data works and that the control strategies are integrated into the overall control flow in a synchronized way (as discussed in Chapter 9.6.5), a simple control strategy has been implemented. To test it, type 1 of the LV grids of the Smart Nord project has been simulated (without fridges). Although the control strategy would also work with the complete Smart Nord scenario, using a this simplified single grid is sufficient for demonstration purposes and eases graphical illustration. The control strategy monitors the power flow at the slack node of the LV grid and as soon as power flow becomes negative (power is fed back to the MV grid), an immediate charge command is issued to all EVs that are part of the LV grid. When feedback stops (power flow is positive again), the EVs are told to stop charging immediately.

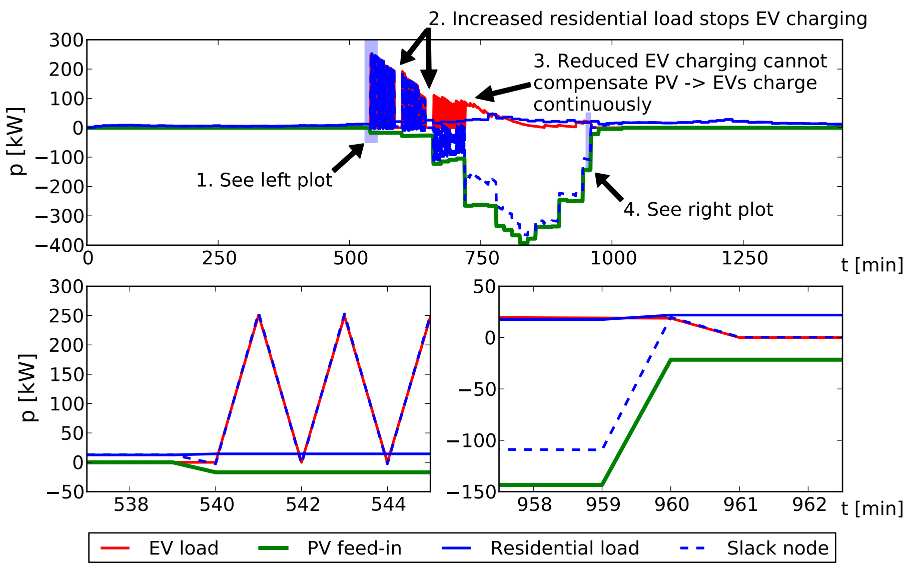


Figure 12.33: Demonstration of controlled EV charging using the ControlAPI of mosaik

Figure 12.33 shows the obtained simulation results. The two plots in the bottom part of the figure zoom in on interesting details of the overall simulation (1 day/1440 minutes). Following the numbers at the arrows shown in the figure, several observations can be made:

1. At t=540, the control strategy notices that the power is fed back into the LV grid as the corresponding dynamic value of the slack node turns negative. This is because the PV systems start to generate electricity and there is only a low consumption of the residential loads. As a consequence, the control strategy submits a charging command to all 38 electric vehicles of the LV grid. The EVs receive this command and process it immediately at the next simulated time step (see Figure 12.4), causing the power flow at the slack node to turn positive again as the EVs now use the PV power for charging and even draw additional power from the MV grid. Thus, the control strategy tells the EVs to stop charging. The resulting sawtooth plot is no

behavior that one would like to see in a real world setting but it is nice to illustrate that the synchronization between control strategy and simulators works as intended.

2. As can be seen in the top plot, this on and off behavior of the EVs stops at t=585 and t=645 for a few minutes. This is due to an increase in residential load which compensates the PV feed-in such that EVs can remain idle until PV production increases again.

3. At t=720 the on and off behavior stops again but this time the EVs do not idle but start charing continuously. This is because the accumulated charging power of the EVs (the red curve) is gradually decreasing over the day as the batteries of the EVs fill up and the battery model of the EVSim reduces charging power accordingly.[21] At t=720 the power of all EVs is no longer sufficient to take up all of the PV feed-in such that there is still a feed back of PV power to the MV grid.

4. Finally, at the end of the day when the sun sets (a winter day has been simulated) residential load is higher than PV feed-in (at t=960) and EVs stop charging completely.

Although the control strategy is quite simple, this scenario has shown that reading entity data (power flow at the slack node), submitting data to the entities (the EV schedules) and synchronization (e.g. the sawtooth behavior) works as intended.

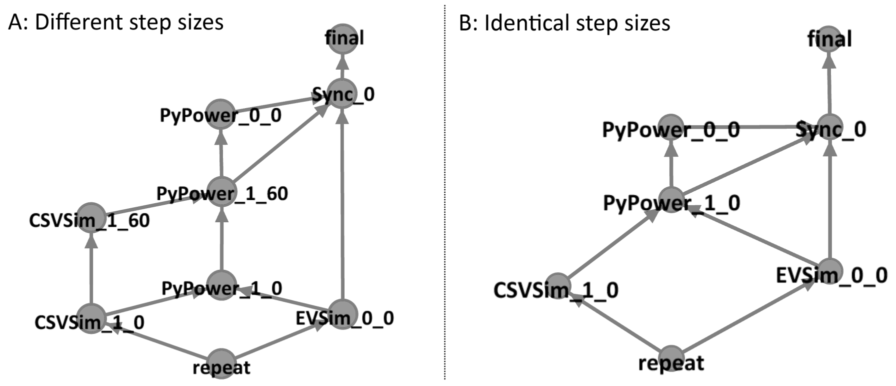


Figure 12.34: Schedule graph for a Smart Nord scenario with different simulator step sizes

Simulators with different step sizes

To make sure that the implementation also works for scenarios involving simulators with different step sizes, the Smart Nord scenario was changed such that the EVs as well as the MV grid have a step size of two minutes rather than one. These simulation models were chosen as they allow to demonstrate two different cases. In case of the EVs, the model consuming the data (the LV grid) steps faster and in case of the MV grid, the models providing the data (the LV grids) step faster. Figure 12.34 shows a part of the overall schedule graph that has been generated for both scenarios from the

[21] This is a typical charging process for Lithium-Ion batteries as they are currently used in most EVs.

graphml file dumped during the composition phase. As discussed in Chapter 9.6.4, a node represents a simulator process at a specific simulation time point. Here, the CSVSim process (residential loads) two PyPowerSim instances (PyPower_0_t = MV grid and PyPower_1_t = LV grid) and the EVSim process are shown. The left part of the figure (A) shows the scenario with different step sizes. The simulator processes with a step size of 1 minute are correctly scheduled using two nodes, representing the first and the second minute until the schedule repeats. The LV grid can be simulated for two steps [t_0, t_1[and [t_1, t_2[as soon as the EVSim has done one step producing data for [t_0, t_2[. The MV grid can step as soon as the LV grid has done its second step, allowing to use the last LV grid output as input or use an aggregation function, if specified. The synchronization point is also placed correctly, as it is reached when all simulators have provided data for the interval [t_0, t_2[. For an easy comparison the right part of the figure (B) shows the schedule of the regular Smart Nord scenario with identical simulator step sizes. Here each simulator process only has a single node.

12.5.2 Grid Topology Access

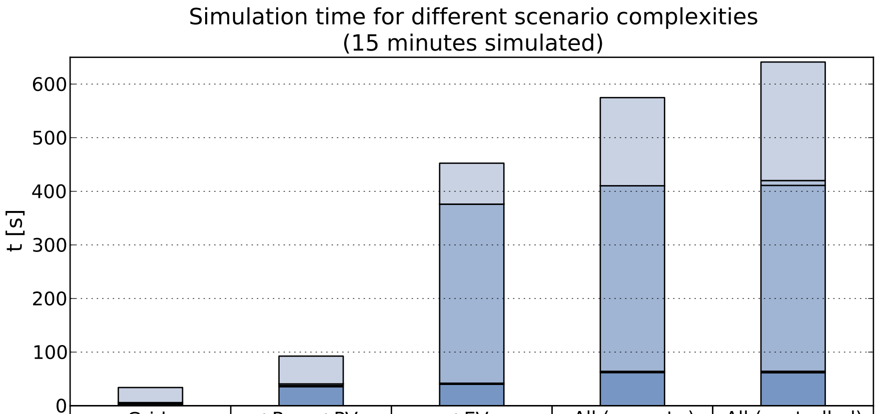
In the context of the Smart Nord project, an algorithm for dynamically rescheduling generation units in self-organized clusters has been proposed [NS13]. A special feature of this algorithm is the fact that it takes the power grid topology into account when rescheduling the units, such that the new schedule has similar electrical effects as the original one. As a consequence, this algorithm must be able to retrieve the power grid topology via the ControAPI, in order to implement it in a scenario independent way. This is desired as it eases reuse of the algorithm in other projects and scenario settings. Requirement [**R 8** – Physical Topology Access] and [**R 10** – Combined Physical Topology] demand such functionality. The function *get_related_entities* of the ControlAPI (see Appendix F) provides this access. To demonstrate that this works as intended, the topological representations shown in Appendix J.5.2 have been generated with Gephi from a *graphml* file that was created at runtime by a small script using the ControlAPI.

12.6 Simulation Performance

Although this thesis did not focus on optimizing simulation performance at run-time, the complexity of the Smart Nord scenario makes it an ideal candidate for doing some first performance benchmarks. The structure of the scenario model does not affect the performance of the actual simulation phase (the phase in which the simulators are stepped until the desired simulation end is reached). This is because after the composition phase the simulation engine simply follows the simulator schedule and processes a flat list of data flows to supply the simulators with input. Therefore, the presented benchmark does not vary any simulation model specific aspects but rather increases the number of simulated entities. As the Smart Nord scenario is used again, five different benchmark runs were performed by using the complete scenario or omitting certain entity types. Table 12.3 shows the number of entities for these runs. As the control strategy used in Section 12.5.1 only exchanges data when PV feed-in increases, the benchmark runs use another control strategy which retrieves the dynamic data of all busses (p, q, voltage angle and magnitude) at every time step and submits a charging command to every EV at every time step. Thus, a constant and quite realistic data exchange between the control strategy and mosaik is generated.

Table 12.3: Number of entities for the overall performance benchmark

Name of run	LV Nodes	Households	PVs	EVs	Fridges
Grid only	11981	0	0	0	0
Grid +Res. +PVs	11981	13381	11981	0	0
Grid +Res. +PVs +EVs	11981	13381	11981	11981	0
All (uncontrolled)	11981	13381	11981	11981	13381
All (controlled)	11981	13381	11981	11981	13381



	Grids	+Res. +PVs	+EVs	All (uncontr.)	All (controlled)
5. Simulate	28.1	51.9	76.4	164.5	221.1
4. Init. controller	0.0	0.0	0.0	0.0	9.1
3. Initialize sims	1.0	3.0	334.1	346.2	346.8
2. Instantiate sims	2.0	2.3	2.1	2.3	2.7
1. Composition	2.8	35.4	39.9	61.8	61.5

Figure 12.35: Benchmark of the simulation performance for different scenario complexities

Figure 12.35 shows the results for a simulated time span of 15 minutes (15 simulator steps) on the same server used for benchmarking the composition algorithms (4x AMD Opteron 2.4 GHz/64 GB RAM/Ubuntu 10.04 LTS 64bit). Based on these results, a number of observations can be made. First of all, the composition time for the "grid only" run is very small, as the ConnectionRules to connect the slack nodes of the LV grid to the MV grid do not use any join predicates. For all other runs, the composition time is higher. This is mostly caused by the ConnectionRule that connects the residential loads to the LV grid nodes, as this ConnectionRule uses a join predicate that compares the ID of the node to the ID of the residential load. In Section 12.4.5 it was shown that such join predicates can cause problems when the number of entities in the two sets increases. Although the overall Smart Nord scenario has such a large number of nodes and residential loads, the ConnectionRule is applied per LV grid instance and thus, the number of combinations per rule evaluation is small (261 nodes times 299 residential loads = 78.039 combinations for the largest LV type). For the runs where PVs and EVs have to be connected the composition time only increases a few seconds as their ConnectionRules do not use join predicates either.

The initialization time for the control strategy in the last run is only a few seconds. In this time, the retrieval of the entity relations (including the grid topology) is done and for illustration purposes, the graph is dumped into a graphml file, as discussed above. The initialization time of the simulators, however, increases dramatically for the runs that involve EVs. This time is consumed in the EVSim itself as it parses the statistical trip data and initializes the random trip generators for the large number of vehicles. This nicely shows that the overall performance of a composed scenario depends on external factors that cannot necessarily be influenced by the composition approach itself.

While all these times discussed above only elapse once in the beginning of the simulation, the time required for the actual simulation (i.e. advancing the simulators and exchanging data) is much more important. While the simulation of the "grid only" scenario takes about 28 seconds (about 32 times faster than real-time), the simulation of the full scenario with control strategy (which even gets bigger in the final Smart Nord project, as already mentioned above) already takes about 142 seconds and thus only runs about 3.5 times faster than real-time. Simulating a complete year would take two months. While it is certainly possible to only simulate shorter, representative time spans (e.g. a week in summer, spring and winter), these results indicate that measures to increase simulation performance have to be the next step for the mosaik platform. The discussion of Figure 12.34 (see Section 12.5.1) has shown that mosaik already schedules (and actually executes) the different simulators in parallel if the data flows allow this. Therefore, other measures to improve performance have to be identified. First ideas will be discussed briefly in the conclusion chapter below.

12.7 Assumptions

A number of assumptions were made during this thesis, mostly to limit complexity of the mosaik approach [**R 1** – Keep It Simple] but also because the scope of mosaik (see Chapter 3.4) allowed to cut back certain aspects. In the following, these assumptions are reflected based on the experiences and insights gained during the evaluation.

A 1 – Model Reuse is Desirable This is the initial assumption from which the research objective has been derived. A general discussion of simulation model reuse was not within the scope of this thesis. A number of pro and contra arguments can be found in Robinson et al. [RNP+04] and Pidd [Pid02]. However, the evaluation has shown that the composition of Smart Grid scenarios based on reused simulation models is feasible for actual research projects. Pidd [Pid02] points out that "it is vital to have a properly developed reuse strategy if any benefits are to be gained from reuse." He states four aspects that such a strategy must account for and which are (with some exceptions) supported by mosaik:

1. Abstraction – Refers to a high level description of the reusable artifacts to assist the developers in understanding their purpose, nature and behavior. In case of mosaik this description is provided to the Scenario Experts (the developers) by

the simulator descriptions defined semantic layer.

2. Selection – Refers to mechanisms to locate, compare and select reusable artifacts. In Chapter 8.2.8 it was discussed why this aspect is not supported by the mosaik concept as yet.
3. Specialization – Refers to a technique to turn reusable abstract artifacts into concrete entities. While mosaik does not directly offer a means to create specializations of models (as these are black-boxes), this aspect is supported by allowing to parametrize the models conveniently and thus turn general models into specific ones.
4. Integration – Refers to a mechanism to combine and connect the reusable artifacts and enable communication between them. In case of mosaik this mechanism is provided by the scenario and composition layer.

A 2 – Typeless entity relations are sufficient In Chapter 6.3.1 it was assumed that the entity relations can be provided as simple tuples, i.e. only returning tuples of related entities and no further relation details. This assumption is still valid, as the types of relations (e.g. electrical flow, physical flow, informational flow) are limited and can be inferred from the types of connected entities.

A 3 – No central unit conversion required In Chapter 7.4 the assumption that the mosaik platform does not have to deal with unit conversion issues was made. The argument that this can be done in the implementation of the SimAPI (i.e. the simulator adapters) is still valid. However, the creation of generic adapters such as the CSV or FMI adapter presented in Sections 12.2.1.3 and 12.2.2.3 gets more complex, as the adapters now have to deal with the unit conversion or the FMI model and the CSV files have to be changed (which is not always possible and not the desired approach).

A 5 – Interpolation is not reasonable in the majority of cases This assumption can be retained as the scenarios could all be implemented with simulation models that have the same step sizes.

A 4 – No explicit temporal validity In Chapter 7.8.1 it was assumed that the semantic metamodel does not need to provide a concept to define the time points for which data provided by entities is valid. Rather, the used DataDefinitions are expected to contain such information for non-physical flows. For the simulators used in the evaluation this only applies to the EVSim. All other flows are continuous state informations or physical flows. Therefore, no validity problems arise as part of the composition procedure and this assumption is still valid.

A 6 – It is sufficient to specify n:1 relations to capture Smart Grid Scenarios The fact that all scenarios could be specified without problems shows that this assumption can also be retained.

A 7 – Any two simulator step sizes are integer multiples The evaluation has shown that this assumption can also be retained as the simulation models used in th evaluation all meet this constraint (or can be configured in such a way that they meet it, e.g. by adjusting a suitable step size).

12.8 Summary

The integration of the simulators identified during requirements analysis and the successful specification and execution of the scenarios identified during requirements analysis showed that the concept meets the established requirements. In addition, two case studies have shown that the concept is also applicable for simulators and scenarios of actual research projects. This also shows that the proposed reference data model is valid. With respect to the syntactic layer, it was demonstrated that simulation models based on open-source as well as COTS simulators can be integrated. The integrated models also included continuous as well as discrete-event based models. The features of the control layer were demonstrated using a fictive control mechanism which was simple but suitable to show that synchronization and interaction between mosaik and the control strategy works as intended.

Although the evaluation of the Smart Nord scenario has shown that the concept as it is now is not ideal for the deployment of objects ”per household“, a possible extension to improve this issue has already been discussed and can be integrated seamlessly in a next version of mosaik. A performance benchmark of the composition phase was made to show that not only the scenario modeling approach is suitable for large-scale scenarios, but also the algorithms for the interpretation of the scenario model and the creation of the composition scale with the scenario size. The algorithms perform in linear time and thus are also applicable for scenarios involving a large number of entities. Using the Smart Nord case study, it was shown how the concepts for hierarchical scenario modeling work as intended in case of scenarios involving power grid topologies with more than one voltage level.

Based on the largest scenario of the case studies, involving about 75000 simulated objects, a performance benchmark of the simulation phase was made (including the composition phase that was already benchmarked separately). The results showed that the simulation only runs about 3.5 times faster than real time. Measures to increase simulation performance should be taken, especially to allow more computation time for realistic control strategies. First ideas to increase simulation performance are discussed in the outlook part of the conclusion chapter below.

13 Conclusion

The conclusion starts with a summary of the research presented in this thesis. This includes a description of how the initially established research questions are addressed and what the research findings are. Finally, the most important limitations of the presented concept as well as possible enhancements for future versions of the mosaik platform are presented.

13.1 Summary of the Presented Research

The research objective of this thesis is to develop a concept which allows to create a simulative testbed for the evaluation of new Smart Grid control mechanisms by composing available simulation models. This objective is based on the assumption that it is desirable to reuse simulation models. Following the definition of composability given by Petty and Weisel [PW03b] (see Definition 5), four research questions were defined, each dealing with a distinct aspect of the research objective. Chapter 3 defines the scope of the research and establishes detailed, problem specific requirements that needed to be considered. This is done by analyzing a set of existing simulation models, potential Smart Grid scenarios as well as requirements of (multi-agent) control strategies. Part I of the thesis closes with Chapter 4 in which related works dealing with Smart Grid simulation are discussed. The research findings of Part I are a comprehensive overview of related works in the field of Smart Grid simulation as well as a set of problem specific requirements. It is shown that no Smart Grid specific approaches to simulation model composability exist and, in particular, there are no approaches allowing to specify large-scale Smart Grid scenarios without much effort.

Therefore, Part II of this thesis presents a domain specific layered concept called *mosaik*. It is derived from two layered models being discussed in literature dealing with simulation composability. Five of the six layers of mosaik are dealt with in this thesis. Each of these layers can be related to one of the research questions with one question being related to two layers.

Addressing the Defined Research Questions

The first research question [RQ 1] “What interface is required to integrate different simulators in composed Smart Grid scenarios?” is dealt with in Chapter 6 (syntactic layer) which presents a well-defined interface, called SimAPI, allowing to interface simulators in a discrete-time fashion. Besides domain independent API functions for setting input data, advancing simulation time and retrieving simulation output, the API also offers functions related to domain specific requirements. This includes access to static data of the simulated entities and retrieval of model internal entity relations. This data can be used to influence the composition phase and also be requested by a control strategy, for example to allow the dynamic, power grid aware bootstrapping of a MAS. The research contributions of this chapter are an overview of existing works dealing with simulator integratability and the definition of the problem specific simulator interface that explicitly distinguishes between models and entities and allows retrieve the relations between the entities.

The second research question [RQ 2] "What information about a simulator and its models is required to facilitate automatic composition?" is dealt with in Chapter 7 (semantic layer) which presents a metamodel that serves two purposes. First, it allows to create a reference data model defining domain specific and model independent entity types and data structures. Second, it allows to create a formal description of a simulator, its models and their entities (real-world objects represented by the model). By referencing elements from the reference data model, the entity types and data flows used in these descriptions become a common meaning. The formal description includes potential parameters for configuring simulators and models as well as the potential step-sizes that a simulator supports. Data flows are defined for the entity types that a model contains. These are grouped by logical ports which are introduced to support the automatic port-wise composition of entities. Again, this chapter has two research contributions: first, a problem specific discussion of related works dealing with semantic interoperablity of simulation models and the identification of domain relevant validity aspects among these, second, the presented metamodel which allows to capture the simulator and simulation model specific information required to check the identified validity aspects.

How potentially large Smart Grid scenarios can be described in a formal way that allows to compose them automatically using the available simulators and their simulation models is the third research question [RQ 3]. It is dealt with in Chapter 8 (scenario layer), in which a new concept for specifying large-scale Smart Grid scenarios is being presented as the main research contribution of this thesis. While existing approaches for specifying composed scenarios require the definition of individual object relations, this thesis presents a novel approach based on a random, rule-based connection mechanism. The rules are accompanied by different constraints that influence the resulting outcome of the composition. These constraints include multiplicity constraints that govern how many entities of the same type are related to another type of entity (e.g. two electrical vehicles per household) and topological constraints that only allow relations if other relations already exist (e.g. electric vehicles only for those households that have photovoltaics). Additional concepts that allow hierarchical compositions (i.e. using compositions as building block in other compositions) have been defined to match the natural hierarchical structure of the power system. Chapter 9 (composition layer) is also related to the third research question as it presents algorithms for the interpretation and execution of the scenario models defined on the scenario layer. The research results of this chapter are the mosaik specific algorithms for interpreting the scenario models and an algorithm for creating a schedule graph that governs the execution order of the individual simulators.

The fourth and last research question [RQ 4] "How can control strategies be evaluated using the simulated Smart Grid scenario?" is dealt with in Chapter 10 (control layer). Here, an interface called ControlAPI is presented. It meets the requirements that were identified for the evaluation of (multi-agent based) control strategies. It allows to access the entity relations (e.g. the power grid topology) and static entity data as well as dynamic entity data that changes over the time. Also, a callback functionality to synchronize with the time of the composite simulation is presented. Functionality to map the data structures of the reference data model to established standards in the energy domain are not within the scope of this thesis as the focus is on the scenario specification mechanism. Although the ControlAPI is not the central contribution of the presented research but merely a simple interface to the information offered by the composed simulation, the presented API explicitly allows the dynamic bootstrapping of topology aware, multi-agent based control mechanisms. This is achieved by allowing access to static entity data describing operational limits of the simulated objects as well as to the relations between the entities and thus the power grid topology. Several other Smart Grid simulation approaches also offer such an interface but do not address these aspects explicitly.

Implementation & Evaluation

Finally, the third and last part of this thesis presents a working implementation of the mosaik concept in Chapter 11. The semantic as well as the scenario metamodel were implemented as a domain specific language called MoSL, using the Xtext framework. Scenarios specified with MoSL are serialized and transferred to a simulation server which implements the algorithms of the composition layer and is thus able to execute the scenarios by actually composing them out of existing simulation models. The evaluation in Chapter 12 demonstrates that the implementation works as intended and shows that the mosaik concept is applicable to actual research projects.

13.2 Limits of the Concept

Although the evaluation has shown that the scenarios of the case studies could be specified, composed and executed satisfactorily, the presented concept has a number of limitations. The most important ones are briefly summarized in the following. It has to be pointed out that these limitations are independent of the presented scenario modeling concept which is a major contribution of this thesis. The semantic layer, the scenario layer and large parts of the composition layer (those interpreting the scenario model) can therefore remain unchanged for alternative concepts that do not expose these limitations.

Discrete-time simulation In Chapter 3.6.1 the decision to integrate simulators in a (fixed step size) discrete-time fashion was deliberatively made [**R 22** – Discrete-Time Simulator Integration (fixed step size)]. It was a compromise between simplicity of the discrete-time concept and the requirements of the simulation studies mosaik targets. However, this decision limits mosaik to scenarios involving stationary power flow analysis (large step sizes, less tightly coupled models). For the simulation of dynamic power system phenomena (time-domain phenomena, see Figure 2.6) occurring on a much smaller time scale, at least a variable step approach with additional rollback functionality of the models is required, as discussed in Chapter 3.4. Nutaro [Nut11b] points out that “continuous simulation is handled in its entirety, and with surprising ease, in the context of a discrete-event worldview.” So if support for simulation studies involving dynamic power flow phenomena is desired, it may be better to switch directly to a discrete-event approach to additionally increase simulation performance by being able to skip time spans in which the model does not change its state.

Communication system co-simulation Due to the discrete-time approach and (depending on the used models) the large step sizes, the current implementation of mosaik is not suitable for the co-simulation of communication infrastructure simulators, as discussed in Chapter 4.2. As the communication infrastructure operates on the scale of milliseconds to seconds, such co-simulations usually investigate the behavior of control mechanisms that have to react in real-time or close to real-time, for example to provide new protection schemes [Lin12, HWG+06].

An exception are communication co-simulation studies, such as Godfrey et al. [GMD+10], dealing with wireless communication infrastructures and not considering message dispatching times. In these cases, reliability of transmission and coverage of wireless signals is in the focus, which can be done on a larger time scale.

Central coordinator The benchmark results presented in Chapter 12.6 show that simulation speed is an issue for large-scale scenarios. During the benchmarks, the CPU load of the worker process of mosaik (see Figure 11.1) was running at 100% CPU[1] load almost all the time while the simulator processes only worked occasionally. This indicates that the centralized approach is currently limiting the performance by causing a bottleneck for the data flow. A single process is not able to handle the whole data flow fast enough to keep the simulators continuously busy. Here, approaches to use multiple worker processes for the same scenario should be investigated, assuming that a multi-core server or multiple servers are available. As the composition performance is satisfying, the scenario could be composed centrally by a single worker which then creates subworkers and provides each with the relevant part of the overall schedule. In case of multiple servers, network communication overhead could be additionally reduced when creating simulator processes with high data flow demands on the same (possibly multi-core) server. For this purpose, data flow graph partitioning approaches from the domain of parallel computing can serve as a starting point (e.g. [KPV86]).

13.3 Future Developments

A number of desirable improvements to the mosaik concept was already identified during the evaluation. These and other extensions that supplement the concept are presented briefly in the following.

Context specific multiplicity constraints The specification of the Smart Nord scenario (see Chapter 12.4.6.2) has shown that constant multiplicity bounds for a ConnectionRule are not sufficient to deploy household specific entities (fridges and EVs in this case) on a “per node” basis. Here, a mechanism to more conveniently control the number of entities that are to be deployed for each node of the power grid is desirable. It was proposed to extend the concept accordingly by allowing the usage

[1] The server running the benchmark includes four AMD Opteron 2.4 GHz CPUs. The value of 100% refers to a single core of these CPU.

of EntityCountOperands (introduced in Chapter 8.5.4.1) to specify the multiplicity boundaries. This way, context specific multiplicity bounds (e.g. depending on the number of households connected to a node) can be defined.

Request-based data access In Chapter 12.2.1.2 a simulator for residential appliances has been integrated into the mosaik platform. The models of this simulator offer the ability to provide samples for possible operation schedules. The creation of this data is a time consuming computational task. Furthermore, the data can become quite large and it is only required sporadically. Therefore, a mechanism that allows a control strategy to trigger this computation on demand by explicitly requesting this data is desired. Such an extension would affect the SimAPI as well as the ControlAPI and the simulator description (semantic layer). The scenario layer can remain untouched.

Global constraints So far, the scenario model only allows to specify constraints for individual connection rules. However, there are situations in which errors (wrong scenario setups) can be detected when also having constraints that are global to a specific scenario. When a ConnectionRule involves static conditions, for example, there may be cases where not all entities are being connected but the multiplicity constraint is met.

Data flow optimization To increase simulation performance without having to change the discrete-time paradigm, the data exchanged between the simulators and the mosaik worker process can be improved. Even when the state of an entity changes at every time-step, this can be a change of a single attribute only, whereas all other attributes are unchanged. Currently, at every time step all data is transferred from and to the simulators. By only transferring deltas, the performance can potentially be improved. An analysis of the logged data of the case studies presented above can reveal the degree of redundancy and thus indicate the potential benefit. Only exchanging deltas can be a good compromise between the simplicity of the discrete-time approach and the performance of a discrete-event approach. By implementing this mechanism in abstract classes from which specific simulator adapters are derived, this additional complexity can be hidden from the developer who is integrating a simulator.

Simulator consolidation As already discussed in Chapter 9.7, the composition algorithms implicitly assume that it is beneficial to let a single simulator process simulate as many model instances as possible. However, depending on the used hardware (multi-core) there may be cases where it is faster to use separate simulator processes and let them execute in parallel. While the number of models per simulator can be limited/controlled by the *maxNumOfModels* attribute of the SimulationModel definition, this parameter was intended to describe the implementational limits of the simulator (i.e. how many models it is able to handle) and not to tune simulator performance. Here, an automatism to determine the ideal number of simulator instances automatically would be desirable. For example, by automatically running a number of different setups and compare the results.

Result analysis The main purpose of a simulation study with mosaik is to judge the appropriateness of a control strategy. The HDF5 database, containing the (dynamic and static) data that has been logged during the simulation runs as well as the entity relations, is the key artifact to calculate metrics and assess the performance of the employed control strategy. The database does not only contain the simulation specific entity and data flow names but for each data set also provides the names of the related types of the reference data model. This way, each data set is given a well defined semantic. Therefore, a generic and simulation model independent approach to define metrics or other Smart Grid visualization techniques (e.g. for spatial analysis [NQX$^+$12]) is a reasonable extension to the mosaik platform. This would not only allow to reuse simulation models but also to reuse metrics and visualizations to analyze different scenarios and control strategies, thus yielding results that are easier to compare.

Multi-resolution modeling In Chapter 7.10 it was discussed how a composition can incorporate models that have different levels of resolution. However, the presented concept is a static one. That is, the composition can be defined but the resolution of the models cannot change during the execution of the simulation. An extension to allow switching between models with different resolutions (see [DB98, HK12]) may be desirable. However, due to the complexity of such an extension it is only justified if it boosts performance. If a low resolution model is not clearly faster, the higher resolution alternative can be used in the composition straight away.

Adapter generation To ease the creation of simulator adapters and to avoid implementation errors, the automatic generation of adapter stubs for different programming languages is a reasonable extension for the mosaik platform. Based on the formal simulator descriptions of the semantic layer, such stubs can be created, for example also using Xtend which has already been used to generate the YAML scenario files from the scenario descriptions. In another step, the generation of unit tests (or at leasts parts of it) for a simulator adapter could be added. This would allow to check if a simulator adapter adheres to the corresponding formal description.

ControlAPI documentation Finally, the current concept assumes that the Control Algorithm Developer (see Chapter 3.1.2) knows what entities the scenario contains and what data flows the entities provide and accept. Currently, he or she has to analyze the scenario model and the descriptions of the used simulation models. To ease this process, the mosaik platform could be extended in such a way that a documentation is being generated for each composed scenario which can then be used by a Control Algorithm Developer to identify the available entities and data flows and make sure that the scenario can satisfy the needs of the control strategy and vice versa.

Informational topology The concept presented in this thesis focuses on the composition of simulation models that are part of the physical topology, i.e. models of the power system or objects that are physically connected to it. When consequently applying the concept of composability to the complete Smart Grid domain, a mechanism to formally describe the informational topology including the communication infrastructure and different software agents is a logical next step. For example, a mechanism to formally describe the interface needs of a control strategy could be added to the control layer, allowing to automatically check if a control strategy is compatible to a given scenario. If the description of the information topology also intends to describe valid interconnections of agents, the BOM discussed in Chapter 7.3.5 is a good starting point to describe the complex interactions among these.

Part IV

Appendix

A SimAPI Definition

init (step_size, sim_params, model_config):
Initialize a simulation and create the model instances. The *init* message is the first command that mosaik sends to the simulator and it is sent only once. It is used to configure the simulator and to setup the model instances.

step_size (int) – defines, how many seconds (in simulation time) **step()** will advance on each call.

sim_params (dict) – General parameters for the simulation. Contains e.g. a start time.

model_config (list) – A list of tuples describing a certain model configuration: **(cfg_id, model_name, num_instances, params)**

Returns: A **dict** that maps model configurations (*cfg_id*) to a list of dicts which describe the entities for each model instance: **{cfg_id: [{eid: etype}, ...], ...}**
The entity IDs must be unique for a given model name.

get_relations ():
Return a list of tuples for each entity relation. Each tuple contains a pair of related entities (their IDs).

Returns: A list of two-tuples for each relation: **[(eid1, eid2), ...]**

get_static_data ():
Return the values of all static attributes for all entities. Static attributes are attributes that do not change during the execution of the simulation.

Returns: A **dict** that maps entity IDs to data dictionaries. Each data dictionary maps the entities' attribute names to their values: **{eid: {attr: val, ...}, ...}**

get_data (model_name, etype, attributes):
Return the current values of all **attributes** for all entities of type **etype** and all instances of the model **model_name**.

model_name (string) – The name of the model to query.

etype (string) – Query all entities of that type.

attributes (list) – List of attribute names to query.

Returns: A **dict** that maps entity IDs to data dictionaries. Each data dictionary maps the entities' attribute names to their values: **{eid: {attr: val, ...}, ...}**

set_data (data):
Set the values of the given attributes for each entity in **data**.

data (dict) – A **dict** with the key being the entity id for which data is received. Each value is again a **dict** with *attribute: value* pairs: **{eid: {attr: value, ...}, ...}**

Returns: nothing

step ():

Advance the simulation by **step_size** steps (as defined in **init()**) and return the current simulation time.

Returns: The current simulation time as **int**.

B OCL 2.3.1 – In a Nutshell

The Object Constraint Language (OCL) [Obj12] is a textual semi-formal general-purpose language standardized by the Object Management Group (OMG) used to define different expressions that complement the information of UML models [CG12]. OCL is declarative and free of side-effects meaning that OCL expressions can query or constrain a model but not modify the state of it. In the following the most important expressions of the OCL language that are used in this thesis are briefly introduced, namely *Invariant* and *Query* expressions. For a complete overview please refer to [CG12] and [Obj12] from which parts of the explanations below are taken.

B.1 Invariants

Using OCL, integrity constraints for a model are expressed as *Invariants* which are defined in the context of a specific type (i.e. a UML class). The expression defined in the body of an invariant must evaluate to a boolean value, indicating if the invariant is fulfilled (true) or violated (words in bold are keywords).

```
context ClassA inv nameOftheInvariant: <boolean expression>
```

For example, the following invariant restricts the attribute values of all objects of ClassA to positive values:

```
context ClassA inv attributeMustBePositive:
  self.myAttribute > 0
```

The keyword *self* refers to an arbitrary instance of *ClassA* and the dot notation is used to access the attributes of that class. However, many invariants will be based on more complex conditions, e.g. using collection operations as defined below. For example, the following invariant checks if all objects in the list *myElements* have a positive value for their attribute *myAttribute*:

```
context ClassB inv attributesPositive:
  self.myElements->forAll(e | e.myAttribute > 0)
```

OCL also defines operations that can be applied to classes rather than objects. One that is frequently used in this thesis is the *allInstances* operation which can be used to retrieve a collection of all instances of a specific class. For example, the following invariant checks if the *name* attribute of all objects of a class *ClassC* is unique (see collection operations below):

```
context ClassC inv nameIsUnique:
  ClassC.allInstances()->isUnique(name)
```

B.2 Types

Besides operating on model specific types (i.e. classes or enumerations), OCL offers predefined types. These are *Integer, Real, String* and *Boolean*. To handle collections, OCL offers the types *Bag, Sequence, OrderedSet, Collection* and *Tuple*. These can be templated, i.e. be defined in such a way that all elements in a collection must be of a specific type, for example a collection of type *Bag(Integer)* can only contain integer values. For type checking OCL offers a predefined operation *oclIsTypeOf* which can be invoked on all objects to check if they are an instance of the given type. For example, the following invariant ensures that the attribute *myObject* is an instance of *ClassA*:

```
context ClassB inv checkAttributeType:
  self.myObject.oclIsTypeOf(ClassA)
```

B.3 Collection Operations

Also operations that can be applied to collections are frequently used and these are invoked using the *<collection>-><operation>* notation, as already shown in the example above. These operations are:

at(Integer) : Object Returns the element at the given index position.

collect(<expression>) : Collection Returns a collection containing the result of applying the expression to all elements contained in the collection.

exists(<boolean expression>) : Boolean True if the expression is true for at least one element of the collection.

forAll(<boolean expression>) : Boolean True if the expression is true for all elements in the collection.

includes(Object) : Boolean True if the collection includes the given object.

isEmpty() : Boolean True if the collection contains no elements.

isUnique(<expression>) : Boolean True if expression is unique for all elements in the collection.

select(<boolean expression>):Collection Returns a collection with all elements that satisfy the given expression.

size() : Natural Returns the number of elements in the collection.

Finally, the *closure* operation is used. It returns a collection that is an accumulation of the source object and the collections resulting from the recursive invocation of expression-with-v in which v is associated exactly once with each distinct element of the returned collection.

```
source->closure(v | expression-with-v)
```

C OCL Constraints - Semantic Metamodel

OCL constraints for the basic simulator structure shown in Figure 7.3

```
C 1|context Simulator inv simNamesUnique:
    Simulator.allInstances()->isUnique(name)
```

```
C 2| context Simulator inv modelNameUnique:
    models->isUnique(name)
```

```
C 3| context SimulationModel inv entityTypeNameUnique:
    entityTypes->isUnique(name)
```

```
C 4| context ParameterDefinition
  inv parameterDefaultTypeValid:
    (valueType.oclIsTypeOf(DateTimeType) implies
      default.oclIsTypeOf(DateTimeConstant))
    and
    (valueType.oclIsTypeOf(StringType) implies
      default.oclIsTypeOf(StringConstant))
    and
    (valueType.oclIsTypeOf(BooleanType) implies
      default.oclIsTypeOf(BooleanConstant))
    and
    (valueType.oclIsTypeOf(FloatType) implies
      default.oclIsTypeOf(FloatConstant) or
      default.oclIsTypeOf(IntConstant))
    and
    (valueType.oclIsTypeOf(IntegerType) implies
      default.oclIsTypeOf(IntConstant))
```

```
C 5| context ParameterDefinition
  inv parameterDefaultRangeValid:
    --Uses function 'within' defined in C 9
    (valueType.oclIsTypeOf(DateTimeType) implies
      valueType.oclAsType(DateTimeType).range.includes(
        default.oclAsType(DateTimeConstant)
      ))
    and
    (valueType.oclIsTypeOf(StringType) implies
       valueType.oclAsType(StringType).range.includes(
        default.oclAsType(StringConstant)
      ))
    and
    (valueType.oclIsTypeOf(FloatType) implies
       valueType.oclAsType(FloatType).range.includes(
        default.oclAsType(FloatConstant)
      ))
    and
    (valueType.oclIsTypeOf(IntegerType) implies
       valueType.oclAsType(IntegerType).range.includes(
        default.oclAsType(IntConstant)
      ))
```

```
C 6| context DateTimeRange
  inv dateTimeRangeConsistent:
    allowedValues->size() > 0 implies lower = null and upper = null
  inv dateTimeRangeValid:
    lower <> null and upper <> null implies lower <= upper
```

```
C 7| context FloatRange
  inv floatRangeConsistent:
    allowedValues->size() > 0 implies lower = null and upper = null
  inv floatRangeValid:
    lower <> null and upper <> null implies lower <= upper
```

```
C 8| context IntRange
  inv intRangeConsistent:
    allowedValues->size() > 0 implies lower = null and upper = null
  inv intRangeValid:
    lower <> null and upper <> null implies lower <= upper
```

```
C 9| context IntRange
  --For other subclasses of Range similar functions are defined
  --Also, within functions with signatures for other Range subclasses exist
  def: within(widerRange:IntRange):Boolean =
    if self.allowedValues->size() > 0 then
      if widerRange.allowedValues->size() > 0 then
        self.allowedValues->forAll(v|
            widerRange.allowedValues->includes(v)
        )
      else
        self.allowedValues->forAll(v|
            widerRange.lower <= v and v <= widerRange.upper
        )
      endif
    else
      if widerRange.allowedValues->size() > 0 then
        false --hard to check. if self.upper = inf this check runs very very
            long...
      else
        widerRange.lower <= self.lower and self.upper <= widerRange.upper
      endif
    endif

  --For other subclasses of Constant the definition is similar
  def: includes(c:IntConstant):Boolean =
    if allowedValues->size() > 0 then
      allowedValues->includes(c.value)
    else
      lower <= c.value and c.value <= upper
    endif

  def: includes(v:ValueExpression):Boolean =
    (v.oclIsKindOf(Constant) implies self.includes(v))
    and
    (v.oclIsKindOf(Distribution) implies
      let d:Distribution = v.oclAsType(Distribution) in
        (d.lower.oclIsTypeOf(IntConstant) and d.upper.oclIsTypeOf(IntConstant))
            implies
          (self.includes(d.lower.oclAsType(IntConstant).value) and
            self.includes(d.upper.oclAsType(IntConstant).value))
        and
        (d.lower.oclIsTypeOf(FloatConstant) and d.upper.oclIsTypeOf(
            FloatConstant)) implies
          (self.includes(d.lower.oclAsType(FloatConstant)) and
            self.includes(d.upper.oclAsType(FloatConstant)))

        --Note: Checking range when d.lower or d.upper is a ParameterReference
        --is less complex as run-time check
    )
    and
    (v.oclIsKindOf(ParameterReference) implies
      v.oclAsType(ParameterReference).value.valueType.range.within(self))
```

```
C 10| context DomainModel inv unitUnique:
  unit->isUnique(name)
```

```
C 11| context Simulator inv stepSizeGreaterZero:
  if stepRange.allowedValues->size() > 0 then
    stepRange.allowedValues->forAll(v:Integer|v > 0)
  else
    stepRange.lower > 0
  endif
```

OCL constraints for Figure 7.5 (Reference Data Model)

```
C 12| context DomainModel inv domainModelNameUnique:
 DomainModel.allInstances()->isUnique(name)
```

```
C 13| context DomainModel inv dataDefNameUnique:
 dataTypes->isUnique(name)
```

```
C 14| context SimpleArrayType
 inv arrayTypeRangePositive:
  range.allowedValues->size() > 0 implies
    (range.allowedValues->forAll(v|v >= 0) and
     range.lower <> null implies range.lower > 0)
```

```
C 15| context ObjectType inv objTypeFieldUnique:
 fields->isUnique(name)
```

OCL constraints for Figure 7.8 (AbstractEntityTypes & Ports)

```
C 16| context AbstractEntityType inv acyclicAbstractEntityTypes:
    super->closure(super)->excludes(self)
```

```
C 17| context AbstractEntityType
 def: isSameOrSubtypeOf(t:AbstractEntityType):Boolean =
  self = t or self->closure(super)->includes(t)
```

OCL constraints for Figure 7.10 (Ports)

```
C 18|context EntityType inv staticDataUnique:
    self.staticData->isUnique(staticData)
```

```
C 19|context EntityType inv staticDataUnique:
    self.staticData->isUnique(name)
```

```
C 20| context EntityType inv portDataNamesUnique:
 ports->collect(portData->collect(name))->isUnique(name|name)
```

```
C 21| context Port inv portDataDefinitionUnique:
 portData->forAll(p1, p2|
    p1 <> p2 and p1.flowType = p2.flowType implies
        p1.directionOut <> p2.directionOut
 )
```

```
C 22| context EntityType inv dataDefinitionUniquePerTargetType:
 /*If p1 and p2 are different ports, then for all flows pd1 of p1 there
  * must not be a flow pd2 of p2 with same direction, same flowType and same
      targetType.
  */
 ports->forAll(p1, p2| p1 <> p2 implies p1.portData->forAll(pd1|not p2.portData
     ->exists(pd2|
    pd1.flowType = pd2.flowType and
    pd1.directionOut = pd2.directionOut and (
     p1.targetType.isSameOrSubtypeOf(p2.targetType) or
     p2.targetType.isSameOrSubtypeOf(p1.targetType) or
     p1.targetType = null or
     p2.targetType = null
    )
 )))
```

```
C 23| context PortDataBase inv optionalFlagValid:
 optional implies not directionOut
```

```
C 24| context Port
  def: entityType():EntityType =
      EntityType.allInstances()->select(et|et.ports->includes(self))->
          asOrderedSet()->at(1)

  def: targetTypeCompatibleTo(pTar:Port):Boolean =
    --uses function defined in C 17
    (self.targetType <> null and pTar.entityType().isOfType <> null) implies
      pTar.entityType().isOfType.isSameOrSubtypeOf(self.targetType)


  def: limitsMatch(flowIn:PortData, flowOut:PortData):Boolean =
    --uses function defined in C 27
    flowIn.limits->forAll(inLim|flowOut.limits->forAll(outLim|
      (inLim.path.pathEnd() = outLim.path.pathEnd() or
       inLim.path = null and outLim.path = null)--no path for primitive types
       implies outLim.range.within(inLim.range)))


  def: mandatoryInputsSatisfiedBy(pOut:Port):Boolean =
    self.portData->select(not directionOut and not optional)
      ->forAll(flowIn|pOut.portData->select(directionOut)
         ->exists(flowOut|flowIn.flowType = flowOut.flowType
                  and limitsMatch(flowIn, flowOut)))

  def: typeCompatibleTo(p:Port):Boolean =
    self.targetTypeCompatibleTo(p) and
    p.targetTypeCompatibleTo(self)

  def: inputsMutuallySatisfiedBy(p:Port):Boolean =
    self.mandatoryInputsSatisfiedBy(p) and
    p.mandatoryInputsSatisfiedBy(self)
```

OCL constraints for Figure 7.11 (DataFlowLimit)

```
C 25| context Type
 def: fieldsOfType():Set(ObjectTypeField) =
  if oclIsTypeOf(PrimitiveType) then
   Set{} -- emtpy set, i.e. no fields for simple types
  else
    if oclIsTypeOf(SimpleArrayType) then
     --recurse into array elements
      oclAsType(SimpleArrayType).type.fieldsOfType()
    else -- It is an object type
      oclAsType(ObjectType).fields
    endif
  endif
```

```
C 26| context ObjectTypeField
  def: typesOfField():OrderedSet(Type) =
    if oclIsTypeOf(ReferencedObjectTypeField) then
      self.oclAsType(ReferencedObjectTypeField).typeRefs->collect(type)
    else -- InlineObjectFieldType currently only has 1 type
      OrderedSet{self.oclAsType(InlineObjectTypeField).fieldType} -- return type
           as set
    endif
```

```
C 27| context ObjectTypeFieldReference
  def: pathEnd():ObjectTypeField =
    if path <> null then
      pathEnd()
    else
      field
    endif
```

```
C 28| context DataFlowLimit inv dataFlowLimitPathValid:
  -- Uses functions defined in C 25 and C 27
  path <> null implies
    let portDataBase:PortDataBase = --navigate association backwards
      PortDataBase.allInstances()->select(p|
        p.limits->includes(self)
      )->asOrderedSet()->at(1) in
    portDataBase.flowType.type.fieldsOfType()->includes(path.pathEnd())
```

```
C 29| context DataFlowLimit inv dataFlowLimitReferencesPrimitiveType:
 -- Uses functions defined in C 27, C 26 and C 25
 -- End of path cannot be a union but must be inline and evaluate to a primitive
      type
  if path <> null then
    path.pathEnd().typesOfField()->size() = 1 and
    path.pathEnd().typesOfField()->at(1).oclIsKindOf(PrimitiveType)
  else --it must be a primitive type, i.e. no fields defined
    let portDataBase:PortDataBase = --navigate association
      PortDataBase.allInstances()->select(p|p.limits->includes(self))->
          asOrderedSet()->at(1) in
    portDataBase.flowType.type.oclIsKindOf(PrimitiveType)
  endif
```

```
C 30| context DataFlowLimit inv dataFlowLimitTypeRangeValid:
  --Note: typesOfField also resolves array types to their defined type
  let pathType:PrimitiveType = path.pathEnd().typesOfField()->at(1) in
     --Range must make set of allowed values smaller not wider
     --Uses function 'within' defined in C 9
    range.within(pathType.range)
    and --type must match
    (range.oclIsTypeOf(DateTimeRange) implies
     pathType.oclIsTypeOf(DateTimeType))
    and
    (range.oclIsTypeOf(StringRange) implies
     pathType.oclIsTypeOf(StringType))
    and
    (range.oclIsTypeOf(FloatRange) implies
     pathType.oclIsTypeOf(FloatType))
    and
    (range.oclIsTypeOf(IntRange) implies
     (pathType.oclIsTypeOf(IntegerType))
```

```
C 31| context ObjectTypeFieldReference inv objTypeFieldRefValid:
  -- Uses functions defined in  C 25 and C 26
  path <> null implies
    field.typesOfField()->exists(
      t:Type|t.fieldsOfType()->includes(path.field)
    )
```

OCL Constraints for Figure 7.18 (Aggregation)

```
C 32| context PortData inv aggregationValid:
  aggregationSpecified   implies directionOut
```

```
C 33| context PortData
  static def: isNumeric(t:Type):Boolean =
    if t.oclIsTypeOf(FloatType) or t.oclIsTypeOf(IntegerType) then
      true
    else if t.oclIsTypeOf(SimpleArrayType) then
      --recurse into array elements
      PortData.isNumeric(t.oclAsType(SimpleArrayType).type)
    else
      false -- It is a non-numeric type
    endif
  endif

  inv aggregationContNumOnly:
    aggregationSpecified implies
        (flowType.isContinuousFlow and PortData.isNumeric(flowType.type))
```

OCL Constraints for Figure 7.19 (DataFlowMapping)

```
C 34| context DataDefinitionMapping inv mappingUnique:
 DataDefinitionMapping.allInstances()->isUnique(name)
```

D OCL Constraints - Scenario Metamodel

OCL constraints for Figure 8.7 (ParameterSet)

```
C 35| context CompositeModel inv simParamSetUnique:
    simconfig->isUnique(name)


C 36| context SimulatorParameterSet inv modelParamSetUnique:
    modelconfig->isUnique(name)


C 37| context SimulatorParameterSet inv simParamSetHomomorph:
    /* A SimulatorParameterSet must only reference those ParameterDefinitions
     * that are defined by the referenced Simulator
     */
    parameterInstances->forAll(simulator.parameters->includes(instanceOf))


C 38| context ModelParameterSet inv modelParamSetHomomorph:
    parameterInstances->forAll(model.parameters->includes(instanceOf))


C 39| context ParameterSet inv ParamSetNotRedundant:
    parameterInstances->isUnique(instanceOf)


C 40| context ParameterSet
  def: providesTheseParameters(params:OrderedSet(ParameterDefinition)):Boolean =
    let providedParameters:Set(ParameterDefinition) =
        self.parameterInstances->collect(instanceOf) in
    params->forAll(p|providedParameters->includes(p))


C 41| context SimulatorParameterSet inv simParamSetComplete:
    --uses function defined in C 40
    providesTheseParameters(simulator.parameters->select(default=null))


C 42| context ModelParameterSet inv modelParamSetComplete:
    --uses function defined in C 40
    providesTheseParameters(model.parameters->select(default=null))


C 43| context ModelParameterSet inv eInstInfoHomomorph:
    entityInstanceInfo->forAll(model.entityTypes->includes(entityType))


C 44| context ModelParameterSet inv eInstInfoUnique:
    entityInstanceInfo->isUnique(entityType)


C 45| context ModelParameterSet inv eInstInfoNotRedundant:
    entityInstanceInfo <> null implies
      entityInstanceInfo.entityType.numOfEntitiesUnknown = true
```

OCL constraints for Figure 8.8 (ParameterInstance)

```
C 46| context ParameterInstance inv paramValueTypeValid:
    /* The values provided by the ValueExpression of a ParameterInstance
     * must match the type of the corresponding ParameterDefinition. */
    (value.oclIsTypeOf(DateTimeConstant) implies
      instanceOf.valueType.oclIsTypeOf(DateTimeType))
    and
    (value.oclIsTypeOf(StringConstant) implies
      instanceOf.valueType.oclIsTypeOf(StringType))
    and
    (value.oclIsTypeOf(BooleanConstant) implies
      instanceOf.valueType.oclIsTypeOf(BooleanType))
    and
    (value.oclIsTypeOf(FloatConstant) implies
      instanceOf.valueType.oclIsTypeOf(FloatType))
    and
    (value.oclIsTypeOf(IntConstant) implies
      instanceOf.valueType.oclIsTypeOf(IntegerType))
    and
    (value.oclIsTypeOf(Distribution) implies
      value.oclAsType(Distribution).returnType = instanceOf.valueType
    )
    and
    (value.oclIsTypeOf(ParameterReference) implies
      value.oclAsType(ParameterReference).value.valueType = instanceOf.valueType
    )
```

```
C 47| context ParameterInstance inv paramValueWithinRange:
    /* The values provided by the ValueExpression of a ParameterInstance
     * must be within the range of allowed values specified by the Type
     * of the corresponding ParameterDefinition.
     */
     --Uses function 'within' defined in C 9
     (instanceOf.valueType.oclIsTypeOf(StringType) implies
        instanceOf.valueType.oclAsType(StringType).range.includes(value))
     and
     (instanceOf.valueType.oclIsTypeOf(DateTimeType) implies
        instanceOf.valueType.oclAsType(DateTimeType).range.includes(value))
     and
     (instanceOf.valueType.oclIsTypeOf(IntegerType) implies
        instanceOf.valueType.oclAsType(IntegerType).range.includes(value))
     and
     (instanceOf.valueType.oclIsTypeOf(FloatType) implies
        instanceOf.valueType.oclAsType(FloatType).range.includes(value))
```

```
C 48| context Distribution inv limitsOfDistributionValid:
  if returnType.oclIsTypeOf(IntegerType) then
    ((lower.oclIsTypeOf(IntConstant) and upper.oclIsTypeOf(IntConstant)) implies
       lower.oclAsType(IntConstant).value < upper.oclAsType(IntConstant).value)
    --Run-time check for cases where upper or lower is a ParameterReference
  else
    --It is a float type distribution
    --Check is analog to above definitions but using different casts
  endif
```

```
C 49| context Distribution inv distributionLimitNotNested:
    not lower.oclIsTypeOf(Distribution) and not upper.oclIsTypeOf(Distribution)
```

```
C 50| context GaussDistribution inv gaussDistributionNotNested:
    not mu.oclIsTypeOf(Distribution) and
    not sigma.oclIsTypeOf(Distribution)
```

OCL constraints for Figure 8.9 (Entity Set)

```
C 51| context PhysicalTopology inv entitySetUnique:
    entitySets->isUnique(name)
```

```
C 52| context EntitySet inv entitySetCardinalityValid:
    --Uses function defined in C 53
    self.cardinality.valueRangePositiveNumeric()
```

```
C 53| context ValueExpression
  def: valueRangePositiveNumeric():Boolean =
    if self.oclIsTypeOf(IntegerType) then
      let range:IntRange = self.oclAsType(IntegerType).range in
        if range.allowedValues->size() = 0 then
          range.allowedValues->forAll(v|v >= 0)
        else
          range.lower >= 0
        endif
    else
      if self.oclIsTypeOf(FloatType) then
        let range:FloatRange = self.oclAsType(FloatType).range in
          if range.allowedValues->size() = 0 then
            range.allowedValues->forAll(v|v >= 0)
          else
            range.lower >= 0
          endif
      else
        if self.oclIsTypeOf(ParameterReference) then
          self.oclAsType(ParameterReference).value.valueType.
              valueRangePositiveNumeric()
        else
          if self.oclIsTypeOf(Distribution) then
            let lower = self.oclAsType(Distribution) in
               lower.valueRangePositiveNumeric()
          else
            false--not numeric
          endif
        endif
      endif
  endif
```

OCL constraints for Figure 8.11 (CompositeModelParameterSet)

```
C 54| context CompositeModel inv compModelParamSetUnique:
    compconfig->isUnique(name)
```

```
C 55| context CompositeModelParameterSet inv compModelParamSetHomomorph:
    parameterInstances->forAll(compositModel.parameters->includes(instanceOf))
```

```
C 56| context CompositeModelParameterSet inv compModelParamSetComplete:
    --uses function defined in C 40
    providesTheseParameters(
        compositModel.parameters->select(default=null)
    )
```

OCL constraints for Figure 8.13 (ConnectionRule)

```
C 57| context ConnectionRule
  def: typesMatch(cp:CompositePort, p:Port):Boolean =
    let cet:CompositeEntityType = --helper for reverse navigation
      CompositeEntityType.allInstances()->select(cet|cet.ports->includes(cp))->
          asOrderedSet()->at(1) in
    --uses function defined in C 17
    (p.targetType <> null and cet.isOfType <> null) implies
      cet.isOfType.isSameOrSubtypeOf(p.targetType)

  def: mandatoryInputsSatisfied(cp:CompositePort, p:Port):Boolean =
    p.portData->select(not directionOut and not optional)
      ->forAll(pFlow|cp.portData->select(directionOut)
         ->exists(cpFlow|pFlow.flowType = cpFlow.flowType))
    and --opposite direction
    cp.portData->select(not directionOut and not optional)
      ->forAll(cpFlow|p.portData->select(directionOut)
         ->exists(pFlow|pFlow.flowType = cpFlow.flowType))

  def: portsAreCompatible(cp:CompositePort, p:Port):Boolean =
    mandatoryInputsSatisfied(cp, p)

  def: checkComposability(
    et1:CompositeEntityType, et2:EntityType):Boolean =
      --ensure external type is substitutable
      --uses function defined in C 17
      et1.externalType <> null and et2.isOfType <> null implies
      et2.isOfType.isSameOrSubtypeOf(et1.externalType)

     and --At least one port is compatible
       et1.ports->exists(cp|et2.ports->exists(p|
       typesMatch(cp, p) and (
         mandatoryInputsSatisfied(cp, p) or portsAreMappable(cp, p))
      ))

  def: checkComposability(et1:EntityType, et2:EntityType):Boolean =
    --uses function defined in C 24
    et1.ports->exists(p1|et2.ports->exists(p2|
    p1.typeCompatibleTo(p2) and
    (p1.inputsMutuallySatisfiedBy(p2) or portsAreMappable(p1, p2))
    ))

  inv connectionRuleValid:
    let et1:EntityTypeBase = subset1.entitySetRef.type in
    let et2:EntityTypeBase = subset1.entitySetRef.type in

    --enforce temporary limitation that no pure CompositeEntityType connections
        can be made
    (et1.oclIsTypeOf(CompositeEntityType) implies et2.oclIsTypeOf(EntityType))
        and
    (et2.oclIsTypeOf(CompositeEntityType) implies et1.oclIsTypeOf(EntityType))
        and

    if et1.oclIsTypeOf(CompositeEntityType) then
      checkComposability(
        et1.oclAsType(CompositeEntityType), et2.oclAsType(EntityType))
    else
      if et2.oclIsTypeOf(CompositeEntityType) then
        checkComposability(
          et2.oclAsType(CompositeEntityType), et1.oclAsType(EntityType))
      else
        checkComposability(
          et1.oclAsType(EntityType), et2.oclAsType(EntityType))
      endif
   endif
```

```
C 58| context Multiplicity inv connectionRuleMultiplicityValid:
     lowerBound <= upperBound and lowerBound >= 0 and upperBound >= 1


C 59| context EntitySetTypeRef inv referencedTypeInSet:
  /*The EntityType referenced must be in the referenced EntitySet*/
  let ps:InstantiableParameterSet = entitySet.parameterSet in

  ps.oclIsTypeOf(ModelParameterSet) implies
    ps.oclAsType(ModelParameterSet).model.entityTypes->includes(type) and

  ps.oclIsTypeOf(CompositeModelParameterSet) implies
    ps.oclAsType(CompositeModelParameterSet).compositModel.entityTypes->includes
        (type)


C 60| context EntitySetTypeRef inv referencedEntitySetValid:
  compositeSet <> null implies
    entitySet.oclIsTypeOf(ModelParameterSet) and
    compositeSet.parameterSet.oclIsTypeOf(CompositeModelParameterSet) and
    compositeSet.oclAsType(CompositeModelParameterSet).compositModel.
        physicalTopology.entitySets->includes(entitySet)


C 61| context ConnectionRule

  def: numOfEntities(s:EntitySet, t:EntityType):Integer =
    if t.numOfEntitiesUnknown then
      --Try to find entity typs instance info
      let esi:Sequence(EntityInstanceInfo) =
        s.parameterSet.oclAsType(ModelParameterSet)
          .entityInstanceInfo->select(i|i.entityType = t) in
      if esi->size() =1 then
        esi->at(1).numinstances
      else
        -1
      endif
    else
      t.numOfEntities
    endif

  def: entitiesInSet(cs:EntitySet, s:EntitySet, t:EntityTypeBase):Integer =
    if s.cardinality.oclIsTypeOf(IntConstant) then
      let setCardinality = s.cardinality.oclAsType(IntConstant).value in
        if s.oclIsTypeOf(CompositeModelParameterSet) then
           --Case 1
           setCardinality
        else
          if cs = null then
            --Case 2
            setCardinality * numOfEntities(s, t.oclAsType(EntityType))
          else
            if cs.cardinality.oclIsTypeOf(IntConstant) then
               --Case 3
               cs.cardinality.oclAsType(IntConstant).value *
               setCardinality * numOfEntities(s, t.oclAsType(EntityType))
            else --Distribution or paramter reference
              -1
            endif
          endif
        endif
    else --Distribution or paramter reference
      -1
    endif
  def: entitiesInSet1():Integer =
    entitiesInSet(
        subset1.entitySetRef.compositeSet,
        subset1.entitySetRef.entitySet,
        subset1.entitySetRef.type)

  def: entitiesInSet2():Integer =
    entitiesInSet(
        subset2.entitySetRef.compositeSet,
        subset2.entitySetRef.entitySet,
        subset2.entitySetRef.type)

  inv connectionRuleMultiplicityMet:
    (entitiesInSet1() <> -1 and entitiesInSet2() <> -1 and
    self._'static' <> null) --static is also a keyword->escaped
      implies
        --Check lower bound met: |set1| >= lower*|set2|
        entitiesInSet1() >=
          multiplicity.lowerBound * entitiesInSet2()
        and
        --Check upper bound met: |set1| <= upper*|set2|
        entitiesInSet1() <=
          multiplicity.upperBound * entitiesInSet2()
```

```
C 62| context ConnectionRule

def: portsAreMappable(p1:Port, p2:Port):Boolean =
  mappings->exists(m:DataDefinitionMapping|
    --The port p1 provides outputs for all inputs of the mapping
    m.source->forAll(s|p1.portData->select(directionOut)->exists(pd|pd.flowType
        = s))
    and --All mandatory inputs port p2 are provided with outputs of the mapping
    p2.portData->select(not directionOut)->forAll(pd|m.target->includes(pd.
        flowType))
  )

def: portsAreMappable(cp:CompositePort, p:Port):Boolean =
  mappings->exists(m:DataDefinitionMapping|
    --The port provides outputs for all inputs of the mapping
    m.source->forAll(s|cp.portData->select(directionOut)->exists(pd|pd.flowType
        = s))
    and --All mandatory inputs of the port are provided with outputs of the
        mapping
    p.portData->select(not directionOut)->forAll(pd|m.target->includes(pd.
        flowType))
 )
```

OCL constraints for Figure 8.16 (CompositeEntityTypes)

```
C 63| context CompositeTypeMapping inv entityTypeSameOrSubtype:
  /* EntityType must be same or subtype to be usable instead of the
   * CompositeEntityType in all situations.
   */
  entityType.isOfType.isSameOrSubtypeOf(compositeType.isOfType)
```

```
C 64| context CompositeTypeMapping
 def: portsMappable(t:AbstractEntityType, cp:CompositePort, p:Port):Boolean =
   --types must match
   (p.targetType <> null and t <> null) implies
   t.isSameOrSubtypeOf(p.targetType)

   and --p must provide all outputs
     cp.portData->select(directionOut)
```

```
       ->forAll(out1|p.portData->select(directionOut)
         ->exists(out2|out1.flowType = out2.flowType))

    and --cp must provide all mandatory inputs
      p.portData->select(not directionOut and not optional)
       ->forAll(in1|cp.portData->select(not directionOut and not optional)
         ->exists(in2|in1.flowType = in2.flowType))

 inv allPortsCanBeSatisfied:
  --EntityType must be able to satisfy all ports of CompositeEntityType
  compositeType.ports->forAll(cp|entityType.ports->exists(p|
   portsMappable(compositeType.externalType, cp, p)
  ))
```

OCL constraints for Figure 8.21 (Topological Condition)

```
C 65| context EntityPathRoot
  static def: getParent(o:OclAny):OclAny =
    if o.left <> null then
      getParent(o.left)
    else
      if o.right <> null then
        getParent(o.left)
      else
        o.exp
      endif
    endif

  inv topologyConditionRootSetValid:
    let sc:StaticCondition = getParent(self.EntityCountOperand) in
      self.rootSet = sc.ConnectionRule.subset1.entitySetRef.entitySet or
      self.rootSet = sc.ConnectionRule.subset2.entitySetRef.entitySet


C 66| context EntityPathRoot inv topologyConditionRootTypeInSet:
    if rootSet.parameterSet.oclIsTypeOf(ModelParameterSet) then
        rootSet.parameterSet.oclAsType(ModelParameterSet).model->includes(
            rootType)
    else
        let cm:CompositeModel =
          rootSet.parameterSet.oclAsType(CompositeModelParameterSet).
              compositModel in
            cm.entityTypes->includes(rootType)
    endif


C 67| context BooleanExpression
  static def: typeStr(o:EntityCountOperand):String = 'int'
  static def: typeStr(o:BooleanExpression):String = 'bool'
  static def: typeStr(o:StringConstant):String = 'string'
  static def: typeStr(o:FloatConstant):String = 'float'
  static def: typeStr(o:BooleanConstant):String = 'bool'
  static def: typeStr(o:DateTimeConstant):String = 'datetime'

  static def: typeStr(o:IntegerType):String = 'int'
  static def: typeStr(o:FloatType):String = 'float'
  static def: typeStr(o:BooleanType):String = 'boolean'
  static def: typeStr(o:StringType):String = 'string'
  static def: typeStr(o:DateTimeType):String = 'datetime'

  static def: typeStr(o:ParameterReference):String =
    typeStr(o.value.valueType)

  static def: typeStr(o:Distribution):String =
    typeStr(o.returnType)
  def: typesAreCompatible(t1:String, t2:String):Boolean =
    (t1 = 'int'      implies t2 = 'int' or t2 = 'int') and
    (t1 = 'float'    implies t2 = 'float' or t2 = 'int') and
    (t1 = 'bool'     implies t2 = 'bool') and
    (t1 = 'string'   implies t2 = 'string') and
    (t1 = 'datetime' implies t2 = 'datetime')

  inv booleanExpressionOperandsValid:
    typesAreCompatible(typeStr(left), typeStr(right))
```

OCL constraints for Figure 8.24 (User-Defined Function)

```
C 68| context UserDefinedFunctionCall
def: argValid(supplied:UserDefinedFunctionCallArg):Boolean=
    let idx:Integer = self.args->indexOf(supplied) in
    let expected:UserDefinedFunctionArg = self.userFunc.args->at(idx) in

    (supplied.oclIsTypeOf(PrimitiveCallArg) implies
        expected.oclIsTypeOf(PrimitiveArg)
    ) and
    (supplied.oclIsTypeOf(DataDefinitionCallArg) implies
        expected.oclIsTypeOf(DataDefinitionArg)
    ) and
    if(supplied.oclIsTypeOf(DataDefinitionCallArg)) then
        supplied.oclAsType(DataDefinitionCallArg).ref.data = expected.oclAsType(
            DataDefinitionArg).dataDefinition
    else
        --function typeCompatibleTo definieren
        supplied.oclAsType(PrimitiveCallArg).value.typeCompatibleTo(expected.
            oclAsType(PrimitiveArg).primitiveType)
    endif

inv callArgsValid:
  args->size() = userFunc.args->size() and
  args->forAll(callArg|argValid(callArg))
```

OCL constraints for Figure 8.26 (Dynamic Condition)

```
C 69| context ConnectionRule inv dynCondPortDataValid:
  dynamic <> null implies
    (subset1.entitySetRef.type.oclIsTypeOf(EntityType) implies
      let et:EntityType = subset1.entitySetRef.type.oclAsType(EntityType) in
        et.ports->exists(p|
            p.portData->select(d|d.directionOut)->includes(dynamic.entityData)
        )
    )
    and
    (subset2.entitySetRef.type.oclIsTypeOf(EntityType) implies
      let et:EntityType = subset1.entitySetRef.type.oclAsType(EntityType) in
        et.ports->exists(p|
            p.portData->select(d|d.directionOut)->includes(dynamic.entityData)
        )
    )
```

```
C 70| context DynamicCondition inv constantTypeValid:
  -- Uses functions defined in C 27
  path.pathEnd().typesOfField()->size() = 1 and
  path.pathEnd().typesOfField()->at(1).oclIsKindOf(PrimitiveType) and
  (path.pathEnd().typesOfField()->at(1).oclIsKindOf(IntegerType) implies
   value.oclIsTypeOf(IntConstant)) and
  (path.pathEnd().typesOfField()->at(1).oclIsKindOf(FloatType) implies
   value.oclIsTypeOf(FloatConstant)) and
  (path.pathEnd().typesOfField()->at(1).oclIsKindOf(StringType) implies
   value.oclIsTypeOf(StringConstant)) and
  (path.pathEnd().typesOfField()->at(1).oclIsKindOf(BooleanType) implies
   value.oclIsTypeOf(BooleanConstant)) and
  (path.pathEnd().typesOfField()->at(1).oclIsKindOf(DateTimeType) implies
   value.oclIsTypeOf(DateTimeConstant))
```

OCL constraints for Figure 8.27 (Simulation Study)

```
C 71| context SimulationStudy inv simStudyDefinesStartStop:
    let names:Set(String) =
        self.parameterValues->collect(parameter.name) in

    names->includes('start') and
    names->includes('stop')
```

```
C 72| context SimulationStudy inv simStudyDefinesStartStop:
    /* A SimulationStudy must only reference those ParameterDefinitions
     * that are defined by the referenced CompositeModel.
     */
    parameterValues->forAll(scenario.parameters->includes(parameter))
```

```
C 73| context SimulationStudy inv simStudyParamsComplete:
    let providedParameters:Set(ParameterDefinition) =
        parameterValues->collect(parameter) in

    --SimStudy Provides all non default parameters
    scenario.parameters->select(default=null)->forAll(
        p|providedParameters->includes(p)
    )
    and
    --Parameter definitions are non-redundant
    parameterValues->isUnique(parameter)
```

```
C 74| context SimulationStudy inv simStudyLinearParamsSameLength:
  combinationType = CombinationType::linear implies
    parameterValues->forAll(p|
      p.values->size()=parameterValues->at(1).values->size()
      or p.values->size()=1
    )
```

E Composition Layer Algorithms

Algorithm 7 Algorithm to determine how many instances of an EntitySet can be handled by a given simulator

```
1: procedure AssignToSimulator(simInst: SimulatorInstance, cardinality:Int, esi:EntitySetInstance, context:Map)
2:     modelParamSet ← esi.entitySet.parameterSet
3:     values ← resolve(modelParamSet, context)
4:     model ← modelParamSet.model
5:     if simInst cannot simulate more instances of model then
6:         return 0
7:     end if
8:     if simInst cannot simulate models with different parameter values and this model has different parameter values than a similar model that is already assigned to this simulator then
9:         return 0
10:    end if
11:    if Data flow of esi requires other simulator processes then
12:        return 0
13:    end if
14:    if simInst contains a configuration for this model and the given values then
15:        Increase instances of existing configuration
16:    else
17:        Create a new configuration: (uid, model.name, cardinality, values, esi)
18:    end if
19:    return number of instances that could be handled by this simulator
20: end procedure
```

Algorithm 8 *Free Random Connection* algorithm for establishing entity relations on unconstrained entity sets *set*1 and *set*2 with lower bound *n* and upper bound *m*

```
1:  $R_{new} = \emptyset$
2:  for all $e2 \in set2$ do
3:      $e_{min} \subseteq set1 : |e_{min}| = n$        ▷ Select entities to meet lower bound of e2
4:      $set1 \leftarrow set1 \setminus e_{min}$        ▷ Reduce $set1$ as each $e1 \in set1$ is only connected once
5:      $R_{new} \leftarrow R_{new} \cup \{(e1, e2) : e2 \in e_{min}\}$
6:  end for
7:
8:  while $|set1| > 0$ do
9:      $e1 \in set1$        ▷ Chosen with uniform probability
10:     $set1 \leftarrow set1 \setminus e1$
11:     $e2 \in set2 : |\{(s', t) : (s', t) \in R_{new}\}| < m$        ▷ Select entity with < m connections
12:     $R_{new} \leftarrow R_{new} \cup (s, t)$
13: end while
```

Algorithm 9 *Constrained Random Connection* algorithm for establishing entity relations on entity sets *set*1 and *set*2 constrained by a Join Predicate

```
1:  $R_{new} = \emptyset$
2:  for all $e1 \in set1$ do
3:      $e2 = possibleCandidates(e1)[0]$        ▷ Order of candidates randomly shuffled
4:      $R_{new} \leftarrow R_{new} \cup \{(e1, e2)\}$
5:  end for
```

F ControlAPI Definition

F.1 Functions Called by mosaik

set_client (client):
This function is called by mosaik to register a client of the ControlAPI. This may be a wrapper for a standard compliant API or a control mechanism. At every synchronization point reached by the mosaik simulation, a callback method *activate* is called on the registered client object, allowing it to interact with the ControlAPI.

client (object) – The specific client (an API or control mechanism) to register.

init (absolute_start_time):
Called by mosaik after the initialization phase of the simulation has been performed. The implementation of this method is expected to invoke the *init* callback function of the registered client in order to allow bootstrapping and initialization of potential control strategies.

absolute_start_time (datetime) – The absolute start date and time marking the first step that is simulated.

step (seconds_since_start):
Called by mosaik when a synchronization point is reached. The implementation of this method is expected to invoke the *activate* callback function of the registered clients.

seconds_since_start (int) – The seconds that have elapsed since simulation start marking the time point at which the commands that a control strategy passes to the simulated entities are consumed.

F.2 Functions Called by the Client of the ControlAPI

get_stepsize ():
Returns: The stepsize in seconds in which the APIs callback function is invoked.

get_absolute_time ():
Returns: The datetime at which the data set by the control strategies is received/processed by the simulated entities. The data that is made available by the API is thus valid for the interval [get_time() - get_stepsize(), get_time()[. Calculated by adding the *seconds_since_start* passed by the *step* function to the absolute start time given in the *init* function.

get_relative_time ():
Returns: The seconds elapsed since start of the simulation (in simulation time) at which the data set by the control strategies is received/processed by the simulated entities.

get_entity_types ():
Can be invoked by a control strategy to retrieve a list of entity types that can be found

in the simulation. **Returns:** A list of entity types which includes the specific entity types (class EntityType) as well as their abstract types (class AbstractEntityType).

get_entities_of_type (type_to_search):
Can be invoked by a control strategy to retrieve a list of entity IDs for the given entity type.

type_to_search (string) – The entity type for which the instance IDs shall be returned as a list. Must be a type name that is contained in the returned list of get_entity_types().

Returns: A list of entity ID strings for the given type. If the given type does not exist an empty list is returned.

get_related_entities (entity_id):
Can be invoked by a control strategy to retrieve a list of IDs of those entities that are related to the entity with the given ID.

entity_id (string) – The ID of the entity whose related entities shall be returned.

Returns: A list of entity IDs that are related to the entity with the given ID. Empty list if an entity with the given ID does not exist.

get_static_data (entity_id):
Can be invoked by a control strategy to retrieve the static attributes of the entity with the given ID.

entity_id (string) – The ID of the entity to query.

Returns: A dictionary of static attribute name/value pairs for the entity with the given ID. Empty list if an entity with the given ID does not exist.

get_available_flows (entity_id):
Returns a list describing the available data flows for the entity with the given ID. As the names of the entities PortData do not carry any semantics, the related DataDefinitions are returned as well. To avoid ambiguity, the DataDefinitions are grouped Port-wise and accompanied by the AbstractEntityType that the corresponding Port references as target type. Input flows that have already reached the maximal possible number of connections must not be returned.

entity_id (string) – The ID of the entity to query.

Returns: A list where each element is again a list of elements describing the data flows of a Port of the entity that is queried: [[isOutput, name, DataDefinition.name, targetType.name], ...].

get_dynamic_data (entity_id, all):
Can be invoked by a control strategy to retrieve the dynamic attributes of the entity with the given ID.

entity_id (string) – The ID of the entity to query.

all (boolean) – If set to *true*, all values that were produced in the recent interval are returned.

Returns: A dictionary of dynamic attribute name/values for the entity with the given ID. Empty dict if an entity with the given ID does not exist. If *all* is *true*, a list of dicts carrying the values of the last interval is returned.

set_dynamic_data (entity_id, attr_name, value):
Called by a control mechanism to set dynamic entity attributes.

entity_id (string) – The ID of the entity to set attributes.

attr_name (string) – The name of the attribute to set.

name (object) – The value of the attribute to set. For complex data type, a valid JSON-encoded string is expected.

Returns: -

G MoSL Xtext Syntax Definition

This chapter describes the Xtext language definition for the MoSL DSL (see Chapter 11.2.1). The definition file contains both, the concrete and the abstract syntax of the MoSL language. The language definition is formulated using the Extended Backus-Naur Form (EBNF)-like Xtext language of the Xtext framework [Ecl12].

File *mosl.xtext*

```
grammar de.offis.mosaik.mosl.MoSL with org.eclipse.xtext.common.Terminals
generate moSL "http://www.offis.de/mosaik/mosl/MoSL"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Mosl: PackageDeclaration;

PackageDeclaration:
  'package' name=QualifiedName
  imports+=Import*
  element+=(DomainModel | Simulator | CompositeModel | SimulationStudy)*;

MoSLResource :
  PackageDeclaration | DomainModel | Simulator | CompositeModel |
      SimulationStudy;

QualifiedName returns ecore::EString: ID ('.' ID)*;

QualifiedNameWithWildcard returns ecore::EString: QualifiedName '.*'?;

Import: 'import' importedNamespace=QualifiedNameWithWildcard;

DATETIME returns ecore::EDate: ISO_DATETIME;

terminal ISO_DATETIME:  //YYYY-mm-dd HH:MM:SS
  (('0'..'9') ('0'..'9') ('0'..'9') ('0'..'9') '-'
  ('0'..'1') ('0'..'9') '-'
  ('0'..'3') ('0'..'9') ' '
  ('0'..'2') ('0'..'9') ':'
  ('0'..'5') ('0'..'9') ':'
  ('0'..'5') ('0'..'9'));

FLOAT returns ecore::EFloatObject:
  (INTEGER '.' INT SCIENTIFIC_EXT?) | MAX_FLOAT | MIN_FLOAT;

POSITIVE_FLOAT returns ecore::EFloatObject:
  (POSITIVE_INTEGER '.' INT SCIENTIFIC_EXT?) | MAX_FLOAT | MIN_FLOAT;

terminal SCIENTIFIC_EXT: ('e' | 'E') ('-' | '+') INT;

INTEGER returns ecore::EIntegerObject:
  INT | ('-' INT) | MAX_INTEGER | MIN_INTEGER;

POSITIVE_INTEGER returns ecore::EIntegerObject: INT;

terminal MAX_FLOAT returns ecore::EFloatObject: 'MAXFLOAT';
terminal MIN_FLOAT returns ecore::EFloatObject: 'MINFLOAT';
terminal MAX_INTEGER returns ecore::EIntegerObject: "MAXINT";
terminal MIN_INTEGER returns ecore::EIntegerObject: "MININT";

//*****************************************************************************
// Data Type System
// Order of the types according to grammar on http://orderly-json.org/docs
// Syntax based on https://github.com/crealytics/orderly_json_xtext
```

```
//*****************************************************************************
Type:
  PrimitiveType | ObjectType | SimpleArrayType;

PrimitiveType:
  IntegerType | FloatType | BooleanType | StringType | DateTimeType;

IntegerType:
  {IntegerType} 'integer' ('in' unit=[Unit])? range=IntRange?;

FloatType:
  {FloatType} 'float' ('in' unit=[Unit])? range=FloatRange?;

BooleanType returns BooleanType: {BooleanType} 'boolean';

ObjectType: {ObjectType} 'object' '{' (fields+=ObjectTypeField)* '}';

ObjectTypeField: InlineObjectTypeField|ReferencedObjectTypeField;

InlineObjectTypeField: name=ID ':' fieldType=Type;

ReferencedObjectTypeField:
  name=ID ':' (typeRefs+=[DataDefinition] | ( 'union' '['
    typeRefs+=[DataDefinition] (',' typeRefs+=[DataDefinition])*
  ']'));

ObjectTypeFieldReference:
  '/' field=[ObjectTypeField] path=ObjectTypeFieldReference?;

SimpleArrayType: 'array' '[' type=ArrayType ']' range=IntRange?;

ArrayType: InlineArrayType|ReferencedArrayType;

InlineArrayType: type = Type;

ReferencedArrayType: type = [DataDefinition];

StringType: {StringType} 'string' range=StringRange?;

DateTimeType: {DateTimeType} 'datetime' range=DateTimeRange?;

//*****************************************************************************
// Range Definitions
//*****************************************************************************
Range: IntRange | FloatRange | StringRange | DateTimeRange;

IntRange:
  ({IntRange} '{' allowedValues+=INTEGER (',' allowedValues+=INTEGER)* '}') |
      ({IntRange} '[' lower=INTEGER ','
  upper=INTEGER ('|' 'step' step=POSITIVE_INTEGER)? ']');

FloatRange:
  ({FloatRange} '{' allowedValues+=FLOAT (',' allowedValues+=FLOAT)* '}') | ({
      FloatRange} '[' lower=FLOAT ','
  upper=FLOAT ('|' 'step' step=POSITIVE_FLOAT)? ']');

StringRange: {StringRange} '{' (items+=STRING)? (',' items+=STRING)* '}';

DateTimeRange:
  ({DateTimeRange} '{' allowedValues+=DATETIME (',' allowedValues+=DATETIME)*
      '}') | ({DateTimeRange}'[' lower=DATETIME ',' upper=DATETIME ']');

//*****************************************************************************
// Value provider
//*****************************************************************************
ValueExpression: Constant | ParameterReference | Distribution;
```

```
// Distributions
Distribution: GaussDistribution | RouletteDistribution;

RouletteDistribution:
  type='roulette' '[' lower=(Constant | ParameterReference) ',' upper=(
      Constant | ParameterReference) ']' 'as' returnType=(IntegerType |
      FloatType);

GaussDistribution:
  type='gauss' '(' mu=(Constant | ParameterReference) ',' sigma=(Constant |
      ParameterReference) ')' ('['
  lower=(Constant | ParameterReference)? ',' upper=(Constant |
      ParameterReference)? ']')? 'as' returnType=(IntegerType
  | FloatType);

//Constants
Constant:
  StringConstant | FloatConstant | IntConstant | BooleanConstant |
      DateTimeConstant;

IntConstant: value=INTEGER;
FloatConstant: value=FLOAT;
StringConstant: value=STRING;
DateTimeConstant: value=DATETIME;
BooleanConstant: value=("True" | "False");

// Parameters and References
ParameterDefinition:
  name=ID ':' valueType=PrimitiveType ('default' '=' default=Constant)?;

ParameterInstance: instanceOf=[ParameterDefinition] '=' value=ValueExpression;

ParameterReference:
  value=[ParameterDefinition] (compuMethod=ComputationMethod)?;

ComputationMethod: LinearScale;

LinearScale:
  {LinearScale} ('*' scale=FLOAT) | (('+')? offset=FLOAT) | ('*' scale=FLOAT (
      '+')? offset=FLOAT);

//****************************************************************************
// Reference data model
//****************************************************************************
DomainModel:
  'domain model' name=QualifiedName
  ('units' unit+=Unit* 'end')?
  ('entity types' entityTypes+=AbstractEntityType* 'end')?
  'data types' dataTypes+=DataDefinition* 'end'
  ('user functions' userFunctions+=UserDefinedFunction* 'end')?
  ('mappings' mappings+=DataDefinitionMapping* 'end')?
    'end';

UserDefinedFunction:
  type=PrimitiveType name=ID '(' args+=UserDefinedFunctionArg (',' args+=
      UserDefinedFunctionArg)* ')';

UserDefinedFunctionArg: DataDefinitionArg|PrimitiveArg;

DataDefinitionArg: dataDefinition = [DataDefinition];

PrimitiveArg: primitiveType = PrimitiveType;

Unit:
  name=STRING
  //The 7 SI base units and how they make up this unit (taken from FMI
      standard)
  //For example, for kg*m^2/s^2 the attributes of BaseUnit are: kg=1, m=2, s
      =-2)
  (('kg^' kg=INTEGER)? & ('m^' m=INTEGER)? & ('s^' s=INTEGER)? & ('A^' A=
      INTEGER)? & ('K^' K=INTEGER)? & ('mol^'
  mol=INTEGER)? & ('cd^' cd=INTEGER)? & ('rad^' rad=INTEGER)?) scale=
      LinearScale?;

AbstractEntityType:
  name=STRING (':' super+=[AbstractEntityType] (',' super+=[AbstractEntityType
      ])*)?;

DataDefinition:
  ('@Equals' '(' asserts+=[DataDefinition] ')')*
  (isSporadicFlow?='sporadic')? name=ID ':' type=Type;

DataDefinitionMapping:
  name=ID 'maps' source+=[DataDefinition|QualifiedName] (','source+=[
      DataDefinition|QualifiedName])* 'to' target+=[DataDefinition|
      QualifiedName] (','target+=[DataDefinition|QualifiedName])*;

//****************************************************************************
// Simulator Definition
//****************************************************************************
UNBOUNDINT returns ecore::EIntegerObject: '*';

enum EStepUnit: min | sec;

Simulator:
  'sim' name=ID 'stepsize' stepRange=IntRange stepUnit=EStepUnit
  parameters+=(ParameterDefinition)* models+=(SimulationModel)* 'end sim';

SimulationModel:
  //If maxNumOfModels is not specified, 1 is assumed
  (maxNumOfModels=(UNBOUNDINT | INTEGER))? 'model' name=ID (mixedParameters?='
      allowing different parameter')?
  parameters+=ParameterDefinition*
  entityTypes+=EntityType+
  'end model';

EntityType:
  //If numOfEntities is not specified, 1 is assumed
  ((numOfEntities=INTEGER) | (numOfEntitiesUnknown?='*'))?
  'entity ' name=ID ':' isOfType=[AbstractEntityType]
  (staticData+=StaticDataDefinition)* ports+=Port* 'end entity';

StaticDataDefinition:
  'static' name=ID ':' staticData=[DataDefinition];

Port:
  (mandatory?= 'mandatory')? 'port' 'accepting' maxConnections=(UNBOUNDINT |
      INTEGER)
  (targetType=[AbstractEntityType])? portData+=(PortData)+ 'end';

PortData:
  (directionOut?='out' name=ID ':' flowType=[DataDefinition] ('accuracy'
      accuracy = DataAccuracy)?
  ('limit' limits+=DataFlowLimit+)?)
  |
  ((optional?='optional')?
  'in' name=ID ':' flowType=[DataDefinition] ('accuracy' accuracy =
      DataAccuracy)?
  ('limit' limits+=DataFlowLimit+)?
  //Aggregation only allowed for simple types (physical flows)
  (aggregationSpecified?='aggregation' aggregationFunction=AggregationFunction
      )?);

DataFlowLimit: (path=ObjectTypeFieldReference)? 'to' range=Range;
```

```
enum DataAccuracy: Unknown|IntervalAVG|IntervalMIN|IntervalMAX|Sample;

enum AggregationFunction: SUM | MIN | MAX | AVG;

//****************************************************************************
// Compositions
//****************************************************************************
CompositeModel:
  'composition' name=ID
  (parameters+=ParameterDefinition|entityTypes+=CompositeEntityType)*
  (simconfig+=SimulatorParameterSet|compconfig+=CompositeModelParameterSet)*
  (physicalTopology=PhysicalTopology)?
  'end';

//Definition of CompositeTypes
EntityTypeBase:
  CompositeEntityType | EntityType;

CompositeEntityType:
  'composite entity' name=ID
  'internal type' isOfType=[AbstractEntityType]
  'external type' externalType=[AbstractEntityType]
  ports+=CompositePort*
  'end composite entity';

CompositePort:
  {CompositePort}
  'port'
  portData+=CompositePortData*
  'end';

PortDataBase: CompositePortData | PortData;

CompositePortData:
  ((directionOut?='out' name=ID ':' flowType=[DataDefinition] ('limit' limits
      +=DataFlowLimit+)?)
  |
  ((optional?='optional')? 'in' name=ID ':' flowType=[DataDefinition] ('limit'
       limits+=DataFlowLimit+)?));

ParameterSet: InstantiableParameterSet | SimulatorParameterSet;

InstantiableParameterSet: ModelParameterSet | CompositeModelParameterSet;

SimulatorParameterSet:
  'sim' 'config' name=ID ':' simulator=[Simulator]
  'with stepsize' stepSize=INT
  parameterInstances+=ParameterInstance*
  modelconfig+=ModelParameterSet*
  'end sim config';

ModelParameterSet:
  'model' 'config' name=ID ':' model=[SimulationModel]
  parameterInstances+=ParameterInstance*
  ('yields' entityInstanceInfo+=EntityInstanceInfo*)?
  'end';

EntityInstanceInfo:
  numinstances=INTEGER 'instances of' entityType=[EntityType];

CompositeModelParameterSet:
  'composition' name=ID ':' compositModel=[CompositeModel|QualifiedName]
  parameterInstances+=ParameterInstance*
  'end';

//****************************************************************************
```

```
//Physical topology
//*****************************************************************************
PhysicalTopology:
  'physical topology'
  entitySets+=EntitySet+
  connectionRules+=ConnectionRule*
  'end physical topology';

EntitySet:
  name=ID '=' cardinality=ValueExpression ('new')?
  parameterSet=[InstantiableParameterSet|QualifiedName]
  ('with ' compositeTypeMapping+=CompositeTypeMapping+
    (',' compositeTypeMapping+=CompositeTypeMapping*)?
  )?;

CompositeTypeMapping:
  /* Mapping CompositeEntityTypes that occur in the set to a
   * CompositeEntityType is not possible right now to reduce complexity.
   *
   * Two types of mappings are possible:
   * 1. Map an EntityType to a CompositeType (if the EntitySet references a
   *    ModelParameterSet)
   * 2. Map an EntityType of an EntitySet in a CompositeModel (if the
   *    EntitySet references a CompositeModelParameterSet)
   */
   //Type 1 mapping:
  (entityType=[EntityType] 'as' compositeType=[CompositeEntityType])
  | //or
  //Type 2 mapping:
  (entitySet=[EntitySet]'.'entityType=[EntityType] 'as' compositeType=[
      CompositeEntityType]);

Multiplicity: //If only lowerBound is defined upperBound is assumed to be
    identical
  lowerBound=INTEGER | (lowerBound=INTEGER '..' upperBound=(UNBOUNDINT |
      INTEGER));

ConnectionRule:
  'connect' (multiplicity=Multiplicity)? //no multiplicity is defined as 0..*
  subset1=EntitySubset
  'to'
  subset2=EntitySubset
  ('mapped by' mappings+=[DataDefinitionMapping]*)?
  (static=StaticCondition)?
  (dynamic=DynamicCondition)?
  (aggregation?='using' 'aggregation')?;

DynamicCondition:
  'when' entityType=[EntityType] '.'
  entityData=[PortData] (path=ObjectTypeFieldReference)? '=' value=Constant;

EntitySubset:
  {EntitySubset}
  entitySetRef=EntitySetTypeRef
  (selector=Selector)?;

Selector: RangeSelector | IDSelector;
RangeSelector: '[' min=INT (':' max=INT)? ']';
IDSelector: '.' id=STRING;

EntitySetTypeRef:
/* A Reference to a specific EntitySet AND EntityType.
   *
   * There are 3 possible Variants:
   * 1. Reference to an EntityType in a locally defined EntitySet of a
   *    SimulationModel : setName.TypeName
   * 2. Reference to an CompositEntityType in a locally defined EntitySet of
   *    a CompositModel: setName.CompositeTypeName
   * 3. Reference to an EntityType in an EntitySet contained in locally
   *    defined EntitySet of a CompositModel:
   *      localSetName->setNameInComposition.TypeName
   * The 4th option is disabled in the scoping:
   * 4. Reference to an CompositEntityType in an EntitySet contained in
   *    locally defined EntitySet of a CompositModel:
   *      localSetName->setNameInComposition.TypeName
   */
  {EntitySetTypeRef} (compositeSet=[EntitySet] '->')? entitySet=[EntitySet] '.
      ' type=[EntityTypeBase];

//*****************************************************************************
// Static condition and related expression tree
//*****************************************************************************
StaticCondition:
  ('where'|'WHERE') exp=OrOperation;

OrOperation returns BooleanExpression:
  AndOperation ({OrOperation.left=current} ('OR'|'or') right=AndOperation)*;

AndOperation returns BooleanExpression:
  PrimaryOperation ({AndOperation.left=current} ('AND'|'and') right=
      PrimaryOperation)*;

PrimaryOperation returns BooleanExpression:
  //Parenthesis are possible
  RelationalOperation | '('OrOperation')';

RelationalOperation returns BooleanExpression:
  OperandExpression (({Equals.left=current} '==' | {GreaterThan.left=current}
      '>' | {LessThan.left=current} '<' |
  {GreaterEquals.left=current} '>=' | {LessEquals.left=current} '<=') right=
      OperandExpression)?;

OperandExpression:
  ValueExpression | EntityCountOperand | UserDefinedFunctionCall|
      StaticEntityDataReference;

StaticEntityDataReference:
  {StaticEntityDataReference}
  ref = EntityPathAndDataDefinition;

UserDefinedFunctionCall:
  userFunc=[UserDefinedFunction] '(' args+=UserDefinedFunctionCallArg (','
      args+=UserDefinedFunctionCallArg)* ')';

UserDefinedFunctionCallArg:
  /*  Similar to UserDefinedFunctionArg, a call arg is either a reference to
   *  a (static)data definition or a primitive constant value
   */
  DataDefinitionCallArg|PrimitiveCallArg;

DataDefinitionCallArg: ref = EntityPathAndDataDefinition;

PrimitiveCallArg: value = ValueExpression;

EntityPathAndDataDefinition: path=EntityPathRoot '.' data=[DataDefinition];

EntityCountOperand: '|' path=EntityPathRoot '|';

EntityPathRoot:
  {EntityPathRoot} rootSet=[EntitySet]'.'rootType=[EntityTypeBase] entityPath=
      EntityPath?;

EntityPath: AnyEntityPath|SpecificEntityPath;
```

```
SpecificEntityPath:
  {SpecificEntityPath} '--'(entitySetRef=EntitySetTypeRef entityPath=
      SpecificEntityPath?);

AnyEntityPath: '--''*' '.' entityType=[EntityTypeBase];

//******************************************************************************
// SimulationStudy definition
//******************************************************************************
enum CombinationType: cartesian | linear;
enum ClockMode: RealTime | AsFastAsPossible;

SimulationStudy:
  'study' 'of' scenario=[CompositeModel|QualifiedName] 'using' combinationType
      =CombinationType
  'value combination' ('clock' clock=ClockMode)? parameterValues+=
      ParameterValues*
  ('controllable sets' publicSets += [EntitySet|QualifiedName] (',' publicSets
       += [EntitySet|QualifiedName])*)?
  ('control strategy' api = ID)?
  ('resolve cycles' delays+=Delay*)?
  'end';

Delay:
  'delay' 'from' source=[EntityType|QualifiedName] 'to' target=[EntityType|
      QualifiedName];

ParameterValues:
  parameter=[ParameterDefinition] '=' '[' values+=Constant (',' values+=
      Constant)* ']';
```

H MoSL Reference Data Model

File *domainmodel.mosl*

```
package common

domain model domain_v1
  units
    "VAr"     kg^1 m^2 s^-3
    "VA"      kg^1 m^2 s^-3
    "W"       kg^1 m^2 s^-3
    "kW"      kg^1 m^2 s^-3 * 1000.0
    "Wh"      kg^1 m^2 s^-2 * 3600.0
    "V"       kg^1 m^2 s^-3 A^-1
    "ohm"     kg^1 m^2 s^-3 A^-2
    "A"       A^1
    "F"       A^2 s^4 kg^-1 m^-2 //Capacitance in farad
    "degC"    K^1 -237.15
    "s"       s^1
    "km"      m^1000
    "percent"
    "deg"     rad^1 * 0.0175 //=pi/180
    "w_m2"    kg^1 s^-3 //Global Horizontal Irradiance in watt/m2
    "m3"      m^3
  end
  entity types
    "Component"//Base type of every entity type

    //Resources
    "Prosumer" : Component
    "EnergyConsumer" : Prosumer
    "EnergySource" : Prosumer

    //Specific types of consumers and sources
    "ResidentialLoad" : EnergyConsumer
    "Photovoltaics" : EnergySource
    "CHP" : EnergySource
    "ElectricVehicle" : Prosumer

    //Power system components
    "PowerSystemResource": Component
    "ConductingEquipment" : PowerSystemResource
    "Line" : ConductingEquipment
    "Connector" : ConductingEquipment
    "Junction" : Connector
    "BusbarSection" : Connector
    "PQProxy" : Connector
    "PVProxy" : Connector

    //Environmental model objects
    "Environment" : Component
  end
  data types
    //-----Static Entity Data---------
    Area : array[PositionPoint][3, MAXINT]//Area must have at least 3 points
    Id : string
    kwPeak : float in kW //Maximum peak power of a generator (e.g. PV)
    Length:float in km
    Peakload: float in W //Maximum peakload of a consumer
    VoltageLevel : float {0.4, 0.6, 3.0, 6.0, 10.0, 15.0, 20.0, 30.0}
    //----End Static Entity Data---------

    //-----EV Data---------
    ConnectedToGrid: boolean
    TravelledDistance : float in km
```

```
VehicleLocationType : string {"Home", "Misc.", "Work", "Driving"}
//-----End EV Data-----

//-----Storage Data------
StorageMode : float in percent [-1.0, 1.0] #-1 = Empty; 1 = Charge; 0 =
    Idle
//Hydro
ReservoirLevel : float in m3
ReservoirCapacity : float in m3
//Chemical
StorageCapacity : float in Wh
StateOfCharge : float in percent [0.0, 1.0]
//-----End Storage Data------

//-----Transformer Data-----
KUP:integer in percent [0, 100]
KUQ:integer in percent [0, 100]
//-----End Transformer Data-----

//-----Operational Limit Data-----
//Based on Model.IEC61970_IEC61968_combined.IEC61970.OperationalLimits.
    ApparentPowerLimit
ApparentPowerLimit_HIGH:float in VA [0.0, MAXFLOAT]
ApparentPowerLimit_LOW:float in VA [0.0, MAXFLOAT]

CurrentLimit:float in A [0.0, MAXFLOAT]//Limit of current (e.g. in a line)

TemperatureLimit_HIGH : float in degC [-273.0, MAXFLOAT]
TemperatureLimit_LOW : float in degC [-273.0, MAXFLOAT]
//-----End Operational Limit Data-----

//-----State Information-----
ApparentPower:float in W
Current:float in A [0.0, MAXFLOAT]//Current current (e.g. in a line)
Temperature : float in degC [-273.0, MAXFLOAT]
SwitchPosition:integer {0, 1}
VoltageMagnitude : float in V [0.0, MAXFLOAT]
VoltageAngle     : float in deg [-90.0, 90.0]
//-----End State Information-----

//based on: Model.IEC61970_IEC61968_combined.IEC61968.Common.PositionPoint
PositionPoint : object {
  xPosition : float in deg [-180.0, 180.0] //wgs84 longitude
  yPosition : float in deg [-90.0, 90.0]   //wgs84 latitude
}

//-----Physical Flows-----
SolarIrradiance : float in w_m2 [0.0, MAXFLOAT]

ActivePowerSinglePhaseFrom : float in W
ReactivePowerSinglePhaseFrom : float in VAr
ActivePowerSinglePhaseTo : float in W
ReactivePowerSinglePhaseTo : float in VAr

//Types for simplified single phase representation of a balanced system
@Equals(VoltageLevel)
ActivePowerSinglePhase : float in W
@Equals(VoltageLevel)
ReactivePowerSinglePhase : float in VAr

//Types for 3 phase balanced/unbalanced system
@Equals(VoltageLevel)
ActivePower3Phase   : array[float in W][3, 3]
@Equals(VoltageLevel)
ReactivePower3Phase : array[float in VAr][3, 3]
//-----End Physical Flows-----
```

```
    //-----Sporadic Flows (Commands)-----
    //Based on Model.IEC61970_IEC61968_combined.IEC61970.Core.
        IrregularIntervalSchedule
    sporadic StorageSchedule : object {
      startTime : datetime //Absolute date/time
      timePoints: array[
        object
        {
          time:integer in s //seconds relative to startTime
          duration:integer in s
          command:string {"idle", "charge", "discharge"}
          activePower:float in W
        }
      ]
    }

    //Based on Model.IEC61970_IEC61968_combined.IEC61970.Core.
        RegularIntervalSchedule
        sporadic RegularStorageSchedule : object {
            startTime : datetime
            timeStep : integer in s
            timePoints : array[
                object {
                    activePower : float in W
                }
            ]
        }
  end

  user functions
    boolean areaContainsPosition(Area, PositionPoint)
  end

  mappings
    Map3To1Phase maps ActivePower3Phase to ActivePowerSinglePhase
  end

end
```

I MoSL Simulator Descriptions

I.1 ApplianceSim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*
import offis.mosaik.simulator.appliancesim.extensions.*

sim ApplianceSim  stepsize [1,MAXINT] min
    start : datetime

    *model Fridge
        entity consumer:EnergyConsumer
            static voltageLevel:VoltageLevel
            port accepting 1 PQProxy
                out P:ActivePowerSinglePhase
                out Q:ReactivePowerSinglePhase
            end
        end entity

        entity  thermal:Component
            port accepting *
                out T:Temperature
            end
        end entity

        entity  thermostat:Component
            port accepting *
                in T_min:TemperatureLimit_LOW
                in T_max:TemperatureLimit_HIGH
                in P_min:ApparentPowerLimit_HIGH
                in P_max:ApparentPowerLimit_LOW
            end
        end entity

        entity scheduler:Component
            port accepting *
                in schedule:RegularStorageSchedule
            end
        end entity

        entity usage:Component
            port accepting *
                optional in daily_pdf:ProbabilityDensitySeries
                optional in T_noise_lower:TemperatureLimit_HIGH
                optional in T_noise_upper:TemperatureLimit_LOW
            end
        end entity

    entity sampler:Component
            port accepting *
                in sample_size:sampleSize
                out current_sample:sample
            end
        end entity
    end model
end sim
```

ApplianceSim specific domain model extensions (ApplianceSim.extensions.mosl)

```
package offis.mosaik.simulator
import common.domain_v1.* //unit access

domain model appliancesim.extensions
    data types
      //Contains the probability density values of a probability density
          function
    ProbabilityDensitySeries : array[float [0.0, 1.0]]
    sporadic sample : array[
            array[
                object {
                    activePower : float in W
                }
            ][96, 96] //1 day in 15 minutes resolution
        ]
        sampleSize : integer [0, MAXINT] //0 = no sampling
    end
end
```

I.2 Cerberus.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim Cerberus stepsize [1, MAXINT] sec

    1 model PowerGrid
        file : string default = "" //Must use absolute path and backslashes

        * entity Node:Junction
          static id:Id
            port accepting *
                out Voltage: VoltageMagnitude
                out Phase: VoltageAngle
            end
        end entity

        1 entity NETZ:PQProxy //Grid/Slack-Node
            static Unenn: VoltageLevel

            port accepting *
                out Power: ActivePowerSinglePhase
                out Q_out: ReactivePowerSinglePhase
            end
        end entity

        * entity TRAFO2S:Component
            static Unenn1: VoltageLevel
            static Unenn2: VoltageLevel
            static Snenn: ApparentPowerLimit_HIGH
        end entity

        * entity LEITUNG:Line//Land line
            static Imax: CurrentLimit
            static Laenge: Length

            port accepting *
                out I: Current
            end
        end entity

        * entity KABEL:Line//Ground cable
```

```
            static Imax: CurrentLimit
      static Laenge: Length

            port accepting *
                out I: Current
            end
        end entity

        * entity LAST:PQProxy //Load
            static Unenn: VoltageLevel
            static Pnenn: ActivePowerSinglePhase
            static Qnenn: ReactivePowerSinglePhase
            static KUP: KUP
            static KUQ: KUQ

            port accepting * EnergyConsumer
                in Pnenn: ActivePowerSinglePhase limit to [0.0, MAXFLOAT]
                    aggregation AVG
                optional in Qnenn: ReactivePowerSinglePhase  aggregation AVG
                optional in KUP : KUP
                optional in KUQ : KUQ
            end
            port accepting *
                out Power: ActivePowerSinglePhase
                out Q_out: ReactivePowerSinglePhase
            end
        end entity

        * entity ASM:PQProxy //Asynchronous Machine
            static Unenn: VoltageLevel
            static Snenn: ApparentPower
            static cos_phi: VoltageAngle

            port accepting 1 EnergySource
                in Snenn: ApparentPower
                in cos_phi: VoltageAngle
            end
            port accepting *
                out Power: ActivePowerSinglePhase
                out Q_out: ReactivePowerSinglePhase
            end
        end entity

        * entity EEA:PQProxy
            //Eigenerzeugungsanlage/in-house electricity generating plant
            static Unenn: VoltageLevel
            static P: ActivePowerSinglePhase
            static Q: ReactivePowerSinglePhase

            port accepting * EnergySource
                in P: ActivePowerSinglePhase limit to [MINFLOAT, 0.0]
                    aggregation AVG
                optional in Q: ReactivePowerSinglePhase  aggregation AVG
            end
            port accepting *
                out Power: ActivePowerSinglePhase
                out Q_out: ReactivePowerSinglePhase
            end
        end entity

        * entity SPEICHER:PQProxy //Storage
            static Unenn: VoltageLevel
            static P: ActivePowerSinglePhase
            static Q: ReactivePowerSinglePhase

            port accepting * Prosumer
                in P: ActivePowerSinglePhase aggregation AVG
                optional in Q: ReactivePowerSinglePhase  aggregation AVG
            end
            port accepting *
                out Power: ActivePowerSinglePhase
                out Q_out: ReactivePowerSinglePhase
            end
        end entity

        * entity SCHALTER:Component //Switch
            static State: SwitchPosition
            static Imax: CurrentLimit

            port accepting *
                optional in State: SwitchPosition
            end
            port accepting *
                out I: Current
            end
        end entity
    end model
end sim
```

I.3 CloudSim.mosl

```
package offis.mosaik.simulator

import common.domain_v1.*

sim CloudSim stepsize [1, 1] sec
  model CloudModel allowing different parameters
    minimumIrradiance : float in percent [0.0, 1.0]
    cellArrayWidth:integer [1,MAXINT] default = 10 #e.g. 10 results in 10x10
        cells

    *entity  Cell : Environment
      static Area:Area
      port accepting *
        out irradiance : SolarIrradiance
      end
    end entity
  end model
end sim
```

I.4 CSVSim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim CSVSim stepsize [1,MAXINT] min
  start:datetime

  * model ResidentialLoads
    file:string
    numEntities:integer default = MAXINT
    * entity Household : ResidentialLoad
      static VoltageLevel:VoltageLevel
      port accepting 1 PQProxy
         out p:ActivePowerSinglePhase
         out q:ReactivePowerSinglePhase
      end
    end entity
  end model
end sim
```

I.5 EVSim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim EVSim stepsize [1, MAXINT] min
  start :datetime
  stop  :datetime

  *model ElectricVehicle
    random_seed :integer default = 42
    e_bat       :float in kW default = 32.0

    //esp = external signal planner
    planner     :string {'default', 'esp'} default = "default"
    p_crg       : float in kW default = 11.0
    plugin_home : boolean default = True    // Plug-in and load at home?
    plugin_work : boolean default = False   // Plug-in and load at work?
    plugin_misc : boolean default = False   // Plug-in and load at misc.?

    1 entity EV:ElectricVehicle
      static storageCapacity:StorageCapacity
      static voltageLevel:VoltageLevel
      port accepting 1 PQProxy
        out p :ActivePowerSinglePhase
        out q :ReactivePowerSinglePhase
      end
      port accepting *
        out location  :VehicleLocationType
        out soc     :StateOfCharge
        out distance  :TravelledDistance
        out plugged_in  :ConnectedToGrid
      end
      port accepting 1
        optional in schedule :StorageSchedule
      end
    end entity
  end model
end sim
```

I.6 FMISim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim FMISim stepsize [1,MAXINT] sec

  *model PumpStorage
    file : string
    v_max: integer in m3 default = 1600000 //Maximum reservoir capacity
    v_0  : integer in m3 default = 800000 //Initial reservoir capacity

    1 entity PumpStorage:Prosumer
      static v_max : ReservoirCapacity

      port accepting 1 PQProxy
        out p : ActivePowerSinglePhase
      end

      port accepting *
        optional in valve_ctrl : StorageMode
        out level : ReservoirLevel
      end
    end entity
  end model
end sim
```

I.7 GridMockSim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim GridMockSim stepsize [1,MAXINT] sec
  *model GridMock allowing different parameters
      // Either use nodes or file parameter
    nodes          :integer default = 0
    file           :string default = ""
    voltageLevel   :string {"LV", "MV"} default = "LV"

    max_trafo_load:integer in W default = MAXINT
    max_line_load :integer in W default = MAXINT

    1 entity Transformer:Prosumer
      static id: Id
      static maxApparentPower : ApparentPowerLimit_HIGH
      static pos: PositionPoint
      port accepting *
        out load : ActivePowerSinglePhase
      end
    end entity

    * entity Line:Line
      static maxApparentPower : ApparentPowerLimit_HIGH
      port accepting *
        out load : ActivePowerSinglePhase
      end
    end entity

    * entity Node:PQProxy
      static id: Id
      static vl:VoltageLevel
      static pos: PositionPoint
      port accepting * Prosumer
        in p : ActivePowerSinglePhase aggregation AVG
      end
      port accepting *
        out u : VoltageMagnitude
        out phi : VoltageAngle
        out load : ActivePowerSinglePhase
      end
    end entity
  end model
end sim
```

I.8 PVSim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim PVSim stepsize [1,MAXINT] min
  start:datetime
  stop:datetime

  *model PV allowing different parameters
      Inverter_maxActivePower : float in W [0.0, MAXFLOAT]
    Angle : float in deg [0.0, 360.0]
    NumberOfStrings : integer [1, MAXINT]
    NumberOfModulesPerString : integer [1, MAXINT]
    IMPP : float default = 7.55
    UMPP : float default = 23.80
    aP : float default = -0.48
    NOCT : integer default = 48

    entity PV:EnergySource
      static vl:VoltageLevel
      port accepting 1 PQProxy
        out p : ActivePowerSinglePhase
        out q : ReactivePowerSinglePhase
      end
    end entity
  end model
end sim
```

I.9 PyPower.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim PyPower stepsize [1,MAXINT] sec
  1 model PowerGrid allowing different parameters
    file          :string

    * entity Grid:Prosumer
      static id: Id
      static vl: VoltageLevel
      port accepting * PQProxy
        out p : ActivePowerSinglePhase
        out q : ReactivePowerSinglePhase
      end
    end entity

    * entity Transformer:Line
      static s_max : ApparentPowerLimit_HIGH

      port accepting *
        out p_from : ActivePowerSinglePhaseFrom
        out q_from : ReactivePowerSinglePhaseFrom
        out p_to : ActivePowerSinglePhaseTo
        out q_to : ReactivePowerSinglePhaseTo
      end
    end entity

    * entity Branch:Line
      static s_max : ApparentPowerLimit_HIGH
            static length: Length
      port accepting *
        out p_from : ActivePowerSinglePhaseFrom
        out q_from : ReactivePowerSinglePhaseFrom
        out p_to : ActivePowerSinglePhaseTo
        out q_to : ReactivePowerSinglePhaseTo
      end
    end entity

    * entity PQBus:PQProxy
      static id: Id
      static vl: VoltageLevel
      port accepting * Prosumer
        in p : ActivePowerSinglePhase aggregation AVG
        optional in q : ReactivePowerSinglePhase aggregation AVG
      end
      port accepting * PQProxy
        out p_out : ActivePowerSinglePhase
        out q_out : ReactivePowerSinglePhase
      end
      port accepting * Connector
        out vm : VoltageMagnitude
        out va : VoltageAngle
      end
    end entity
  end model
end sim
```

I.10 SimplePVSim.mosl

```
package offis.mosaik.simulator
import common.domain_v1.*

sim SimplePVSim stepsize [1,MAXINT] sec
  *model SimplePV allowing different parameters
      peakPower : float in W [0.0, MAXFLOAT]

    entity Inverter:Photovoltaics
      static VoltageLevel:VoltageLevel
      port accepting 1 PQProxy
        out load : ActivePower3Phase
      end
    end entity

    entity Panel:Photovoltaics
      mandatory port accepting 1 Environment
        in irradiance:SolarIrradiance
      end
    end entity
  end model
end sim
```

J Application Examples

J.1 Example Scenario

File *LVexample.mosl*

```
package scenarios.example
import common.domain_v1.*
import offis.mosaik.simulator.*

composition LVexample
  start:datetime
  stop:datetime

  numOfNodesPerLV:integer [0,MAXINT] default = 4

  sim config grid:GridMockSim with stepsize 900
    model config lvgrid:GridMock
      nodes = numOfNodesPerLV
      voltageLevel = "LV"
    end
  end sim config

  sim config csvSim15:CSVSim with stepsize 15 //Minutes
    start = start
    model config residentialLoads:ResidentialLoads
      file = "/home/sschuette/evaluation/data/H0Sommer.csv"
      numEntities = numOfNodesPerLV
    end
  end sim config

  sim config pv_sim : PVSim with stepsize 5
    start=start
    stop=stop
    model config  small_pv : PV //4kWpeak
       Inverter_maxActivePower = 4000.0
       Angle = 30.0
       NumberOfStrings = 4.0
       NumberOfModulesPerString = 5.0
       IMPP = roulette[4, 9] as float
    end
  end sim config

  sim config evSim:EVSim with stepsize 5
    start = start
    stop = stop
    model config E3:ElectricVehicle
      random_seed = 42
      plugin_home = True     # Plug-in and load at home
      plugin_work = True     # Plug-in and load at work
      e_bat = 32.0
    end
    model config controllableE3:ElectricVehicle
      random_seed = 42
      planner = "esp"
    end
  end sim config

  physical topology
    lv    = 1 grid.lvgrid
    loads = 1 csvSim15.residentialLoads
    pvs   = numOfNodesPerLV * 0.75   pv_sim.small_pv //75% of nodes have PV
    evs   = numOfNodesPerLV * 0.5625 evSim.E3 //75% of PV nodes have EV
    connect 1 loads.Household to lv.Node
    connect 0..1 pvs.PV to lv.Node
    connect 0..1 evs.EV to lv.Node where |lv.Node--pvs.PV| > 0
                               when EV.location = "Home"
  end physical topology
end
```

File *MVExample.mosl*

```
package scenarios.example
import common.domain_v1.*
import offis.mosaik.simulator.*

composition MVexample
  start:datetime
  stop:datetime

  //Used for benchmarks
  numOfLVGrids:integer  [0, MAXINT] default = 2
  numOfNodesPerLV:integer [0,MAXINT] default = 4

  sim config grid_sim:GridMockSim with stepsize 900
    model config mvGrid:GridMock
      voltageLevel = "MV"
      nodes = numOfLVGrids #1 node per LV grid
    end
  end sim config

  composition LVExample:scenarios.example.LVexample
    start = start
    stop  = stop
    numOfNodesPerLV = numOfNodesPerLV
  end

  physical topology
    mvGrid = 1 new grid_sim.mvGrid
    lvGrids = numOfLVGrids new LVExample
    connect 1 lvGrids->lv.Transformer to mvGrid.Node
  end physical topology
end
```

File *example.study.mosl*

```
package scenarios.example

study of scenarios.example.MVexample using linear value combination
  start = [2004-07-14 00:00:00]
  stop  = [2004-07-15 00:00:00]
end
```

J.2 Cloud Transient Scenario

File *village.mosl*

```
package cloud_transient
import common.domain_v1.*
import offis.mosaik.simulator.*

composition village
  start:datetime
  stop:datetime

  // Define configurations of models
  sim config gridSim : GridMockSim with stepsize 1
    model config lvgrid:GridMock
      file = "/home/sschuette/evaluation/GridMockSim/data/example.grid"
      yields 6 instances of Node
    end
  end sim config

  sim config pvSim : SimplePVSim with stepsize 1
    model config pv : SimplePV
      peakPower = 1000.0
    end
  end sim config

  sim config cloudSim : CloudSim with stepsize 1
    model config cloud : CloudModel
      minimumIrradiance = 0.25
      yields 100 instances of Cell
    end
  end sim config

  physical topology
    //Create entity sets...
    grid = 1 gridSim.lvgrid
    pvs  = 6 pvSim.pv
    cells = 1 cloudSim.cloud

    //Connect inverters to power grid
    connect 1 pvs.Inverter to grid.Node mapped by Map3To1Phase

    //Connect panel to irradiation model
    connect 0..1 pvs.Panel to cells.Cell
      where areaContainsPosition(cells.Cell.Area, pvs.Inverter--grid.Node.
          PositionPoint)

  end physical topology
end
```

J.3 Grid Friendly Appliance Scenario

File *vctrlstorage.mosl*

```
package scenarios
import common.domain_v1.*
import offis.mosaik.simulator.*

composition voltageControlledStorageTest
  start:datetime
  stop:datetime

  sim config cerberus:Cerberus with stepsize 60
    model config lvgrid:PowerGrid
      file = "C:\\Users\\SharedUser\\Projekte\\mosaik\\cerberus-mosaik\\
          cerberus\\test\\data\\test_case.dsn"
      yields 1 instances of SPEICHER
    end
  end sim config
  sim config storSim:VoltageControlledStorageSim with stepsize 60 //Seconds
    model config storage:Storage
      capacity = 15000
        lowerActivationVoltage = 225
        nominalVoltage = 230
        upperActivationVoltage = 235
        wattPerVolt = 400
    end
  end sim config
  sim config pv_sim : PVSim with stepsize 15
    start=start
    stop=stop
    model config  large_pv : PV
       Inverter_maxActivePower = 8000.0
       Angle = 30.0
       NumberOfStrings = 4.0
       NumberOfModulesPerString = 10.0
       IMPP = 7.0
    end
  end sim config

  physical topology
    grid = 1 cerberus.lvgrid
    storage = 1 storSim.storage
    pv = 1 pv_sim.large_pv

    connect 1 pv.PV to grid.SPEICHER
    connect 1 storage.Battery to grid.SPEICHER
    connect 1 storage.Battery to grid.Node where grid.Node.Id == storage.
        Battery--grid.SPEICHER--grid.Node.Id
  end physical topology
end
```

File *vctrlstorage.study.mosl*

```
package scenarios

study of scenarios.voltageControlledStorageTest using linear value combination

  start = [2004-07-12 00:00:00]
  stop  = [2004-07-13 00:00:00]


  resolve cycles
    delay from
      offis.mosaik.simulator.Cerberus.PowerGrid.Node to
      offis.mosaik.simulator.VoltageControlledStorageSim.Storage.Battery

end
```

J.4 Case Study I: NeMoLand

File *CSVSim.mosl*

```
package nemoland

import common.domain_v1.*

sim CSVSim stepsize [1,MAXINT] min

  start:datetime

  *model LoadProfile allowing different parameter
    file:string
    *entity Load:Consumer
      static id:Id
      static profile:LoadProfile
      port accepting 1
        out p : ActivePowerSinglePhase
      end port
    end
  end

  *model PV allowing different parameter
    file:string
    *entity PV:Producer
      static id:Id
      static kwpeak:kwPeak
      port accepting 1 PQNode
        out p : ActivePowerSinglePhase
      end port
    end
  end

  *model KWK allowing different parameter
    file:string
    *entity KWK:Producer
      static id:Id
      port accepting 1 PQNode
        out p : ActivePowerSinglePhase
      end port
    end
  end




end
```

File *MambaaSim.mosl*

```
package nemoland

import common.domain_v1.*

sim MambaaSim stepsize [15,15] min

  start:datetime
  stop:datetime

  1 model Fleet
    *entity ChargingStationImpl : Prosumer
```

```
      static max_load:Peakload
      static node_id:Id
      port accepting 1 PQNode
        out load:ActivePowerSinglePhase
      end port
    end
  end

end
```

File *nemoland.mosl*

```
package nemoland.scenario

import offis.mosaik.simulator.GridMockSim
import nemoland.MambaaSim
import nemoland.CSVSim

composition nemoland

  start:datetime
  stop:datetime

  load_file:string
  pv_file:string
  kwk_file:string

  num_nodes:integer //number of nodes in the power grid


  sim fleet:MambaaSim with stepsize 15
    start = start
    stop = stop
    model config fleet:Fleet
    end
  end

  sim grid:GridMockSim with stepsize 900
    model config lvgrid:GridMock
      nodes = num_nodes
    end
  end

  sim loadcsv:CSVSim with stepsize 15
    start = start
    model config loads:LoadProfile
      file = load_file
    end
  end

  sim pvcsv:CSVSim with stepsize 15
    start = start
    model config pvs:PV
      file = pv_file
    end
  end

  sim kwkcsv:CSVSim with stepsize 15
    start = start
    model config kwks:KWK
      file = kwk_file
    end
  end
  physical topology
    gridSet = 1 grid.lvgrid
    #fleetSet = 1 fleet.fleet
    loadSet = 1 loadcsv.loads
    pvSet = 1 pvcsv.pvs
    kwkSet = 1 kwkcsv.kwks

    connect 1 loadSet.Load to gridSet.Node

    //connect 0..* fleetSet.ChargingStationImpl to gridSet.Node
      //where fleetSet.ChargingStationImpl.id == gridSet.Node--loadSet.Load.id

    connect 0..1 pvSet.PV to gridSet.Node
      where pvSet.PV.Id == gridSet.Node--loadSet.Load.Id

    connect 0..1 kwkSet.KWK to gridSet.Node
      where kwkSet.KWK.Id == gridSet.Node--loadSet.Load.Id

  end physical topology

end composition
```

File *optimistic.study.mosl*

```
package nemoland

study of nemoland.scenario.nemoland using linear value combination
  start = [
    2012-01-01 00:15:00,
    2012-01-01 00:15:00,
    2012-01-01 00:15:00,
    2012-01-01 00:15:00,
    2012-01-01 00:15:00]
  stop  = [
    2013-01-01 00:00:00,
    2013-01-01 00:00:00,
    2013-01-01 00:00:00,
    2013-01-01 00:00:00,
    2013-01-01 00:00:00]
  load_file = [
      "/home/sschuette/nemoland/data/G_Stromlast_2030_optimistic.csv",
      "/home/sschuette/nemoland/data/L_Stromlast_2030_optimistic.csv",
      "/home/sschuette/nemoland/data/S_1_Stromlast_2030_optimistic.csv",
      "/home/sschuette/nemoland/data/S_2_Stromlast_2030_optimistic.csv",
      "/home/sschuette/nemoland/data/W_Stromlast_2030_optimistic.csv"]
  pv_file   = [
    "/home/sschuette/nemoland/data/G_PV_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/L_PV_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/S_1_PV_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/S_2_PV_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/W_PV_2030_optimistic.csv"]
  kwk_file  = [
    "/home/sschuette/nemoland/data/G_KWK_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/L_KWK_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/S_1_KWK_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/S_2_KWK_2030_optimistic.csv",
    "/home/sschuette/nemoland/data/W_KWK_2030_optimistic.csv"]
  num_nodes = [30, 9, 43, 43, 100]
end
```

J.5 Case Study II: Smart Nord

J.5.1 MoSL Scenario Files

File *lvGrid.mosl*

```
package smartnord.scenarios
import offis.mosaik.simulator.PyPower
import offis.smartnord.simulator.CSVSim
import offis.mosaik.simulator.EVSim
import offis.mosaik.simulator.ApplianceSim
import common.domain_v1.*

composition lvGrid
    start:datetime
    stop:datetime
    gridfile:string
    numNodes:integer[1,MAXINT]
    numHouseholds:integer[1,MAXINT]
    nodeIds:string

    sim config Grid:PyPower with stepsize 60  //1 Minute
        model config LvGrid:PowerGrid
            file = gridfile
        end
    end sim config

    sim config csv:CSVSim with stepsize 1  //1 Minute
        start = start
        model config ResidentialLoads:ResidentialLoads
            file = "/home/sschuette/evaluation2/snord/data/profiles.csv.bz2"
            idList = nodeIds
        end
    end sim config
    sim config csv2:CSVSim with stepsize 1  //1 Minute
      start = start
        model config PV:PV
            file = "/home/sschuette/evaluation2/snord/data/4kwpeak2011.csv"
            numEntities = numNodes
        end
    end sim config

    sim config evSim:EVSim with stepsize 1 //1 minutes
    start = start
    stop = stop
    model config E3:ElectricVehicle
      random_seed = 52
      plugin_home = True    // Plug-in and load at home
      e_bat = 32.0
      planner = "esp"
    end
  end sim config

  sim config appSim:ApplianceSim with stepsize 1
    start = start
    model config fridge:Fridge
    end
  end sim config


    physical topology
        lvGrid = 1 Grid.LvGrid
        resid = 1 csv.ResidentialLoads
        pvs = 1 csv2.PV
        evs = numNodes evSim.E3
        fridges = numHouseholds appSim.fridge


        connect 1..* resid.Household to lvGrid.PQBus WHERE lvGrid.PQBus.Id ==
            resid.Household.Id
        connect 1 evs.EV to lvGrid.PQBus
        connect 1 pvs.PV to lvGrid.PQBus

        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE |lvGrid.PQBus--
            resid.Household| > 0
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|
        connect 0..1 lvGrid.PQBus to fridges.consumer  WHERE
          |fridges.consumer--lvGrid.PQBus| == 0  AND |lvGrid.PQBus--resid.
              Household| > 0 AND
          |lvGrid.PQBus--fridges.consumer| < |lvGrid.PQBus--resid.Household|


    end physical topology
end
```

File *lvGrids.mosl*

```
package smartnord.scenarios
import offis.mosaik.simulator.PyPower
import offis.smartnord.simulator.*
import common.domain_v1.*

composition lvGrids
    start:datetime
    stop:datetime
    numLV1:integer [0, MAXINT] default = 0
    numLV2:integer [0, MAXINT] default = 0
    numLV3:integer [0, MAXINT] default = 0
    numLV4:integer [0, MAXINT] default = 0
    numLV5:integer [0, MAXINT] default = 0
    numLV6:integer [0, MAXINT] default = 0
    numLV7:integer [0, MAXINT] default = 0
    numLV8:integer [0, MAXINT] default = 0

    composite entity LVGroup
      internal type PQProxy
      external type Prosumer
      port
        out p:ActivePowerSinglePhase
        out q:ReactivePowerSinglePhase
      end
    end composite entity

    composition lv1:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 38
      numHouseholds = 41
      //Anonymized IDs. Real node IDs are not ordered and may occur more than
      // once (i.e. n households in the same house)
      nodeIds = '["node1", ..., "node38"]'
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_1_HA_41.json"
    end

    composition lv2:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 165
      numHouseholds = 139
      nodeIds = '["node1", ..., "node165"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_2_HA_139.json"
    end

    composition lv3:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 71
      numHouseholds = 67
      nodeIds = '["node1", ..., "node71"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_3_HA_67.json"
    end

    composition lv4:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 45
      numHouseholds = 57
      nodeIds = '["node1", ... , "node45"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_4_HA_57.json"
    end

    composition lv5:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 136
      numHouseholds = 169
      nodeIds = '["node1", ... , "node136"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_5_HA_169.json"
    end

    composition lv6:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 261
      numHouseholds = 299
      nodeIds = '["node1", ... , "node261"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_6_HA_299.json"
    end

    composition lv7:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 47
      numHouseholds = 66
      nodeIds = '["node1", ... , "node47"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_7_HA_66.json"
    end

    composition lv8:smartnord.scenarios.lvGrid
      start = start
      stop = stop
      numNodes = 81
      numHouseholds = 103
      nodeIds = '["node1", ... , "node81"]'//Anonymized ids...
      gridfile = "/home/sschuette/evaluation2/snord/data/120918
          _AD_NS_Land_8_HA_103.json"
    end

    physical topology
        lv1set = numLV1 lv1 with lvGrid.Grid as LVGroup
        lv2set = numLV2 lv2 with lvGrid.Grid as LVGroup
        lv3set = numLV3 lv3 with lvGrid.Grid as LVGroup
        lv4set = numLV4 lv4 with lvGrid.Grid as LVGroup
        lv5set = numLV5 lv5 with lvGrid.Grid as LVGroup
        lv6set = numLV6 lv6 with lvGrid.Grid as LVGroup
        lv7set = numLV7 lv7 with lvGrid.Grid as LVGroup
        lv8set = numLV8 lv8 with lvGrid.Grid as LVGroup
    end physical topology
end
```

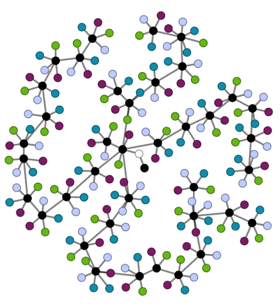
File *mvGrid.mosl*

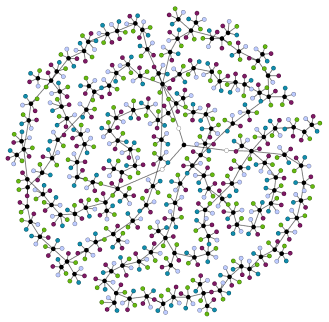
```
package smartnord.scenarios

import offis.mosaik.simulator.PyPower
import offis.smartnord.simulator.*
import common.domain_v1.*

composition mvGrid
    start:datetime
    stop:datetime

    sim config Grid:PyPower with stepsize 60  //1 Minute
        model config MvGrid:PowerGrid
            file = '/home/sschuette/evaluation2/snord/data/
                MS_data_CIGRE_Szenarien.json'
        end
    end sim config

    composition node2setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV1 = 1      numLV3 = 1      numLV4 = 2
      numLV5 = 2      numLV6 = 3      numLV7 = 4
      numLV8 = 3
     end

     composition node4setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV2 = 1      numLV3 = 2      numLV4 = 1
      numLV6 = 1      numLV7 = 2      numLV8 = 3
     end

     composition node5setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV1 = 2      numLV4 = 2      numLV6 = 3
      numLV8 = 1
     end

     composition node6setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV1 = 2      numLV4 = 2      numLV5 = 2
      numLV6 = 3      numLV8 = 1
     end

     composition node7setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV1 = 4      numLV2 = 2      numLV3 = 2
      numLV4 = 1      numLV5 = 3      numLV6 = 1
     end

     composition node8setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV1 = 2      numLV2 = 1      numLV3 = 3
      numLV4 = 3      numLV5 = 1      numLV6 = 2
      numLV8 = 1
     end

     composition node9setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV1 = 1      numLV2 = 2      numLV3 = 4
      numLV4 = 1      numLV5 = 1      numLV7 = 1
     end

     composition node10setup:smartnord.scenarios.lvGrids
      start = start   stop = stop
      numLV2 = 1      numLV3 = 3      numLV4 = 1
      numLV6  = 1       numLV7  = 2       numLV8  = 1
     end

     composition node11setup:smartnord.scenarios.lvGrids
      start = start    stop = stop
      numLV1  = 2       numLV2  = 1       numLV3  = 2
      numLV4  = 1       numLV6  = 3       numLV7  = 2
      numLV8  = 1
     end

     composition node12setup:smartnord.scenarios.lvGrids
      start = start    stop = stop
      numLV1  = 1       numLV2  = 1       numLV3  = 2
      numLV4  = 1       numLV5  = 1       numLV6  = 2
      numLV7  = 2       numLV8  = 1
     end

    physical topology
        mvGrid      = 1 Grid.MvGrid
        node2grids  = 1 node2setup
        node4grids  = 1 node4setup
        node5grids  = 1 node5setup
        node6grids  = 1 node6setup
        node7grids  = 1 node7setup
        node8grids  = 1 node8setup
        node9grids  = 1 node9setup
        node10grids = 1 node10setup
        node11grids = 1 node11setup
        node12grids = 1 node12setup

        connect 1 node2grids.LVGroup  to mvGrid.PQBus."MS_K2"
        connect 1 node4grids.LVGroup  to mvGrid.PQBus."MS_K4"
        connect 1 node5grids.LVGroup  to mvGrid.PQBus."MS_K5"
        connect 1 node6grids.LVGroup  to mvGrid.PQBus."MS_K6"
        connect 1 node7grids.LVGroup  to mvGrid.PQBus."MS_K7"
        connect 1 node8grids.LVGroup  to mvGrid.PQBus."MS_K8"
        connect 1 node9grids.LVGroup  to mvGrid.PQBus."MS_K9"
        connect 1 node10grids.LVGroup to mvGrid.PQBus."MS_K10"
        connect 1 node11grids.LVGroup to mvGrid.PQBus."MS_K11"
        connect 1 node12grids.LVGroup to mvGrid.PQBus."MS_K12"
    end physical topology
end
```

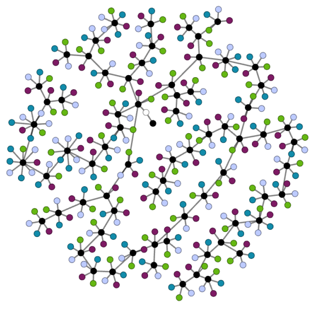
File *snord.study.mosl*

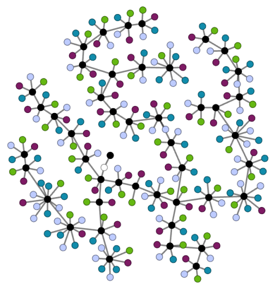
```
package scenarios.snord

study of smartnord.scenarios.mvGrid using linear value combination
  start = [2011-01-01 00:00:00]
  stop  = [2011-01-01 00:15:00]




  controllable sets smartnord.scenarios.lvGrid.evs,
    smartnord.scenarios.lvGrid.lvGrid,
    smartnord.scenarios.mvGrid.mvGrid
  control strategy snordbenchmkctrl

end
```

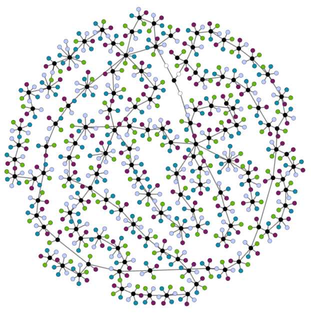
J.5.2 Smart Nord LV Grid Topologies (visualized from simulation results)

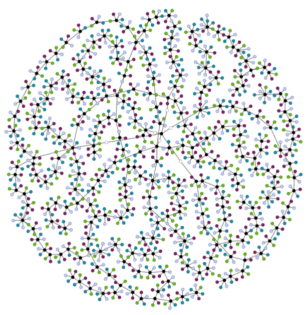

(a) LV Grid Type 1 (38 buses)

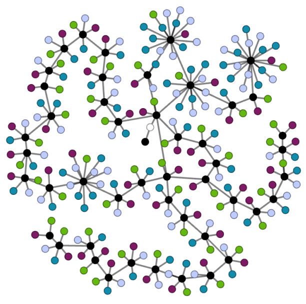

(b) LV Grid Type 2 (165 buses)

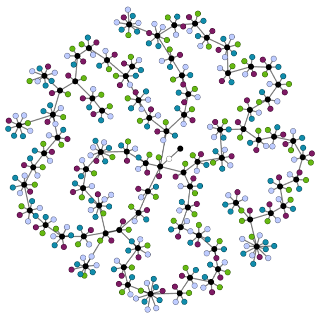

(c) LV Grid Type 3 (71 buses)


(d) LV Grid Type 4 (45 buses)


(e) LV Grid Type 5 (136 buses)

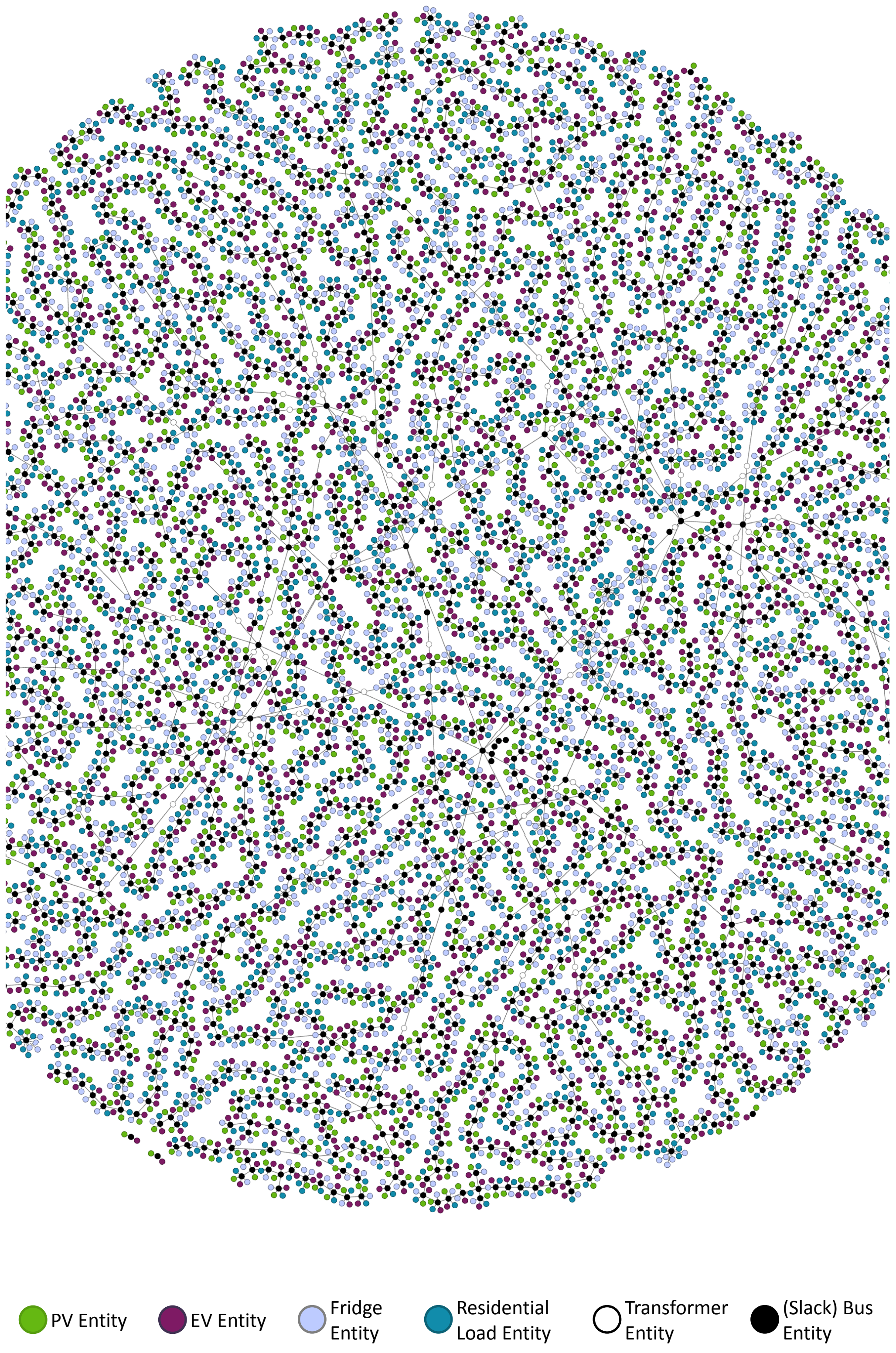

(f) LV Grid Type 6 (261 buses)


(g) LV Grid Type 7 (47 buses)


(h) LV Grid Type 8 (81 buses)

🟢 PV Entity ● EV Entity ○ Fridge Entity ● Residential Load Entity ○ Transformer Entity ● (Slack) Bus Entity

J.5.3 Partial Visualization of the Smart Nord Scenario



K GridLab-D MoSL Files

K.1 gridlab_domain.mosl

```
package gridlabd
domain model domain_v1
  units
    "sf"
    "degF"
    "W"
    "s"
  end
  entity types
    "ResidentialLoad"
    "Node"
    "Line"
    "Transformer"
    "CommercialConnectionPoint"
    "ResidentialConnectionPoint"
    "Appliance"
    "Recorder"
  end
  data types
    name:string
    parentChildRelation:object{}
  end
end
```

K.2 gridlabd.mosl

```
package gridlabd
import gridlabd.domain_v1.*

sim GridLabD stepsize [1,1] sec
  1 model grid
    file:string
    * entity node:Node
      static name:name
    end entity
    * entity overhead_line:Line
      static name:name
    end entity
    * entity  transformer:Transformer
      static name:name
    end entity
    * entity meter:CommercialConnectionPoint
      static name:name
    end entity
    * entity  triplex_meter:ResidentialConnectionPoint
      static name:name
      port accepting * ResidentialLoad
        in load : parentChildRelation
      end
      port accepting * Recorder
        in recorder : parentChildRelation
      end
    end entity
  end model

  * model house allowing different parameters
    latitude    :string
    longitude :string
    floor_area  :float in sf [0.0, MAXFLOAT]
    envelope_UA :float
    window_wall_ratio :float [0.0, 1.0]
    heating_setpoint  :float [0.0, MAXFLOAT]
    cooling_setpoint  :float [0.0, MAXFLOAT]
    //more parameters here...

    1 entity house:ResidentialLoad
      port accepting * Appliance
        in appliances : parentChildRelation
      end

      port accepting 1 ResidentialConnectionPoint
        out load : parentChildRelation
      end
    end entity
  end model

  * model recorder allowing different parameters
    interval:integer in s default = 3600
    property:string
    1 entity recorder:Recorder
      port accepting *
        out recordee:parentChildRelation
      end
    end entity
  end model

  * model waterheater allowing different parameters
    tank_volume :integer
    tank_UA :float
    water_demand :float //gpm
    heating_element_capacity :integer in W
    location : string {"INSIDE"}
    tank_setpoint :integer in degF

    1 entity waterheater:Appliance
      port accepting *
        out load : parentChildRelation
      end
    end entity
  end model
end sim
```

K.3 residentialModel.mosl

```
package gridlabd
import gridlabd.*

composition residentialModel
  sim config gridlabd:GridLabD with stepsize 1
    model config house:house
      latitude      = "48N"
      longitude       = "125W"
      floor_area      = 1500.0
      envelope_UA     = 450.0
      window_wall_ratio = 0.25
      heating_setpoint  = 72.0
      cooling_setpoint  = 76.0
    end
    model config waterheater:waterheater
      tank_volume   = 60
      tank_UA     = 2.0
      water_demand  = 0.25
      heating_element_capacity = 4500
      location     = "INSIDE"
      tank_setpoint = 120
    end
  end sim config

  physical topology
    houses = 1 gridlabd.house //with house as ResidentialModel
    heater = 1 gridlabd.waterheater
      connect 1 heater.waterheater to houses.house
  end physical topology
end
```

K.4 residentialScenario.mosl

```
package gridlabd
import gridlabd.*

composition residentialScenario
  start:datetime
  stop:datetime

  sim config gridlabd:GridLabD with stepsize 1
    model config grid:grid //based on residential_loads.glm of GridLab-D
        samples folder
      file = "C:/Users/sschuette/Dissertation_StS/Evaluation/mosaikLab-D/
          mosaik/test/data/4nodefeeder.glm"
      yields 1 instances of triplex_meter
    end
    model config rec:recorder
      property = "measured_current_1"
    end
  end sim config

  composition residential:residentialModel
  end

  physical topology
    grid = 1 gridlabd.grid
    loads = 2 residential
    rec = 1 gridlabd.rec

    connect 0..2 loads->houses.house to grid.triplex_meter
    connect 1 rec.recorder to grid.triplex_meter
  end physical topology
end
```

K.5 residentialScenario.glm

```
module tape;
module powerflow;
module climate;
module residential {
    implicit_enduses LIGHTS|PLUGS;
};

clock {
    starttime '2000-01-01 00:00:00';
    stoptime '2000-01-07 23:59:59';
    timezone GMT0;
}
```

```
#include "C:/Users/sschuette/Dissertation_StS/Evaluation/mosaikLab-D/mosaik/
    test/data/4nodefeeder.glm";

object recorder{
    name MeterCorder;
    parent Meter;
    property measured_current_1;
    file plot:residential_meter.plot;
    interval 3600;
}

object house {
    name house_1;
    parent Meter;
    latitude "48N";
    longitude "125W";
    floor_area 1500.0 sf;
    envelope_UA 450.0;
    window_wall_ratio 0.25;
    heating_setpoint 72.0;
    cooling_setpoint 76.0;
}

object waterheater {
    name waterheater_1;
    parent house_1;
    tank_volume 60;
    tank_UA 2.0;
    water_demand 0.25 gpm;
    heating_element_capacity 4500 W;
    location INSIDE;
    tank_setpoint 120 degF;
}

object house {
    name house_0;
    parent Meter;
    latitude "48N";
    longitude "125W";
    floor_area 1500.0 sf;
    envelope_UA 450.0;
    window_wall_ratio 0.25;
    heating_setpoint 72.0;
    cooling_setpoint 76.0;
}

object waterheater {
    name waterheater_0;
    parent house_0;
    tank_volume 60;
    tank_UA 2.0;
    water_demand 0.25 gpm;
    heating_element_capacity 4500 W;
    location INSIDE;
    tank_setpoint 120 degF;
}
```

Glossary

In the following the essential terms used in this thesis are elucidated. This glossary assumes that the reader has background knowledge in the field of information technology. Related terms, especially concepts of object oriented programming, are therefore not listed in this glossary. The provided definitions rely on the IEC 60050 Electropedia[1], professional literature (references to which are given when the respective term is introduced in the thesis) as well as on own definitions of concepts developed in this thesis. The used symbol ~ refers to the term that is currently explained and the symbol ↾ refers to another term being part of this glossary.

ActivePower Under periodic conditions, mean value, taken over one period T, of the instantaneous power p (the product of the voltage U between the terminals and the electric current I in the element or circuit).

ApparentPower Product of the root-mean-square (rms) voltage U between the terminals of a two-terminal element or two-terminal circuit and the root-mean-square (rms) electric current I in the element or circuit.

Appliance An ~ is an apparatus intended for household or similar use.

Branch Subset of an electrical network, considered as a two-terminal circuit, consisting of a circuit element or a combination of circuit elements.

CircuitBreaker A mechanical switching device, capable of making, carrying and breaking currents under normal circuit conditions and also making, carrying for a specified time and breaking currents under specified abnormal circuit conditions such as those of short circuit.

Combined Heat and Power (CHP) ~ plants are systems which transform the used primary energy (i.e. natural gas, oil, wood, etc.) into heat and electricity, usually resulting in optimized efficiency in comparison to separate transformations. Typical examples for ~ plants are gas turbines, combustion engines and fuel cells.

CompositeEntityType A ~ is a logical ↾EntityType which does not actually occur in a ↾simulation model but which is defined by a ↾CompositeModel as a logical group of ↾EntityTypes.

CompositeModel A ~ allows to specify ↾ParameterSets, ↾EntitySets as well as ↾ConnectionRules. Hence, it defines how entities from different ↾simulation models form a ↾composition.

CompositeModelParameterSet A ~ is a ↾ParameterSet containing values for the parameters of a specific ↾CompositeModel.

Composition In the context of this thesis, the term ~ refers to a number of interconnected ↾simulation models.

[1] IEC 60050 - International Electrotechnical Vocabulary, `http://www.electropedia.org` (accessed 21 June 2013)

ConnectionRule A ~ allows to specify two ↑EntityTypes representing entities from different ↑simulation models and number of constraints that govern how the specified ↑EntityTypes are to be connected to form a composition.

Consumer A user of electricity provided by an electrical ↑power system.

Control strategy The term ~ refers to any algorithm which is intended to observe and manipulate the state of the objects (simulated or real) of a power system (or those that are somehow physically connected to the power system) and that is implemented in such a way that it can be executed by a computer.

ControlAPI The ~ is an interface that allows a ↑control strategy to interact with the simulated ↑entities.

Displacement Angle Under sinusoidal conditions, the ~ describes the phase difference between the voltage applied to a linear two-terminal element or two-terminal circuit and the electric current in the element or circuit.

Distributed Energy Resource (DER) The term ~ includes distributed generation, the storage of electrical and thermal energy, and/or flexible ↑consumer. DER units are operated either independently of the electrical system or connected to its low or medium distribution networks and located close to the point of consumption, irrespective of the technology..

DistributionLine A ↑line which is used for the distribution of electricity.

Domestic Appliance See ↑appliance.

Dynamic Data The term ~ refers to those attributes of an ↑entity which do change over time (are time variant).

Entity An ~ is any object or component in the modeled system that requires explicit representation.

EntitySet An ~ represents the entities that will at run-time result from the instantiation of one or more ↑simulation models or ↑CompositeModels for a given ↑ModelParameterSet or ↑CompositeModelParameterSet.

EntityType An ~ is a formal description of a specific type of entities that occur in a ↑simulation model.

Institute of Electrical and Electronics Engineers (IEEE) The ~ describes itself as the world's largest professional association dedicated to advancing technological innovation and excellence for the benefit of humanity. ~ and its members inspire a global community through its highly cited publications, conferences, technology standards, and professional and educational activities. ~ publishes nearly a third of the world's technical literature in electrical engineering, computer science, and electronics.

International Electrotechnical Commission (IEC) The ~ describes itself as the world's leading organization for the preparation and publication of International Standards for all electrical, electronic and related technologies. IEC provides a platform to companies, industries and governments for meeting, discussing and developing the International Standards they require.

Interoperability ~ is the ability of two or more systems or components to exchange information and to use the information that has been exchanged.

Inverter An ~ is an electric energy converter that changes direct electric current to single-phase or polyphase alternating currents.

Line A device connecting two points for the purpose of conveying electromagnetic energy between them.

Mealy-type System A ↑system is called a ~ if the output is a function of its current state and the current input.

Memoryless System A ↑system is called an ~ if potential state transitions are time-triggered and the output depends on the state only (↑Moore-type system).

Memoryless System A ↑system is called a ~ if the output is directly dependent on the input and no state transitions occur.

Metamodel A metamodel is a model of models. Instances of a metamodel are models.

Model Used as a synonym of ↑Simulation Model.

ModelParameterSet A ~ is a ↑ParameterSet containing values for the parameters of a specific, formally described ↑simulation model.

Moore-type System A ↑system is called a ~ if the output is computed solely on its current state.

Node End-point of a ↑branch connected or not to one or more other branches.

Object Constraint Language (OCL) The ~ is a textual semi-formal general-purpose language standardized by the ↑Object Management Group (OMG) used to define different expressions that complement the information of UML models.

Object Management Group (OMG) The ~ is an international, open membership, not-for-profit computer industry standards consortium. Founded in 1989, OMG standards are driven by vendors, end-users, academic institutions and government agencies.

ParameterSet A ~ is a set of non-redundant, mandatory parameter values for a specific, formally defined parameterizable ↑Simulator, ↑SimulationModel or ↑CompositeModel.

Photovoltaics (PV) ~ is a decentralized electricity generation technology based on the direct conversion of light into electricity.

Physical Time The term ∼ refers to the time in the physical system that is being simulated.

Power Electronics The term ∼ describes a field of electronics which deals with the conversion or switching of electric power with or without control of that power.

Power System A ∼ includes all installations and plants provided for the purpose of generating, transmitting and distributing electricity.

PQ Bus A ∼ (also called load bus) is a ↑node with predetermined ↑active power and ↑reactive power input.

Protection The term ∼ describes provisions for detecting faults or other abnormal conditions in a ↑power system, for enabling fault clearance, for terminating abnormal conditions, and for initiating signals or indications.

Pumped (Hydroelectrical) Storage In a ∼ water is raised by means of pumps and stored for later use in one or more hydroelectric installations for the generation of electricity.

PV Bus See ↑Voltage Controlled Bus.

ReactivePower For a linear two-terminal element or two-terminal circuit, under sinusoidal conditions, quantity equal to the product of the apparent power S and the sine of the ↑displacement angle.

Scenario The term ∼ refers to the system or situation being modeled.

Scenario Metamodel The ↑metamodel allows to define ↑scenario models by referring to formally described simulators of the semantic layer.

Scenario Model A ∼ is a description of an actual ↑scenario.

Semantic Metamodel The ↑metamodel allows to create a domain-specific reference data model and formal descriptions of ↑simulators and their ↑simulation models and ↑entities that are based on this reference data model.

Semantics ∼ is the relationship of symbols or groups of symbols to their meaning in a given language.

SimAPI The ∼ is an interface that a ↑simulator has to implement for being able to get integrated into the mosaik platform.

Simulation The process of executing a (computer-based) ↑simulation model.

Simulation Model A simulation model is an abstract representation of a system and is executed by a simulation environment. The term SimulationModel, written in medial capitals (camel case), refers to the corresponding class of the ↑semantic metamodel which represents a formal description of a ∼ .

Simulation Time ∼ is the representation of the ↑physical time within a ↑simulation model.

Simulator A ~ is the execution environment for one or more ↑simulation models.

SimulatorParameterSet A ↑ParameterSet for a specific, formally described ↑simulator.

SlackBus An infinite ↑bus where the voltage magnitude is predetermined and which is at the same time reference node and balancing bus.

Solar Irradiance (Direct) ~ is a measure of the rate of solar energy arriving at the Earth's surface from the Sun's direct beam, on a plane perpendicular to the beam.

Static Data The term ~ refers to those attributes of an ↑entity which do not change over time (are time invariant).

Substation A part of an electrical system, confined to a given area, mainly including ends of ↑transmission lines or ↑distribution lines, electrical switchgear and controlgear, buildings and ↑transformers. A ~ generally includes safety or control devices (for example ↑protection).

SwingBus See ↑slack bus.

System A ~ is set of interrelated elements considered in a defined context as a whole and separated from their environment.

Transformer A ~ is an electric energy converter without moving parts that changes voltages and currents associated with electric energy without change of frequency.

TransmissionLine A manufactured transmission medium used to convey electromagnetic energy between two points with a minimum of radiation.

Voltage Controlled Bus A ~ is a ↑node with predetermined active input power and node voltage magnitude.

Wall-clock Time The term ~ refers to the processing time of the ↑simulation.

Wind Energy Converter (WEC) A system which converts the kinetic wind energy into electric energy.

Abbreviations

API	Application Programming Interface
BOM	Base Object Model
CHP	Combined Heat and Power
CIM	Common Information Model (IEC 61970)
COTS	Commercial Off-The-Shelf
CSP	Constraint Satisfaction Problem
CSV	Comma-Separated Values
DER	Distributed Energy Resource
DEVS	Discrete Event System Specification
DFG	Data Flow Graph
DSL	Domain-specific Language
DSM	Demand-Side-Management
EV	Electric Vehicle
EBNF	Extended Backus-Naur Form
FMI	Functional Mock-Up Interface
FMU	Functional Mock-Up Unit
FOM	Federate Object Model
FNSS	Function Specified System
GLM	GridLab-D Model (file)
GUI	Graphical User Interface
HLA	High Level Architecture
HV	High Voltage
IEC	International Electrotechnical Commission
IED	Intelligent Electronic Device
IEEE	Institute of Electrical and Electronics Engineers
IMPP	Maximum Peak Power Current
IO	Input/Output
JSON	JavaScript Object Notation
LCIM	Levels of Conceptual Interoperability Model
LV	Low Voltage
MAS	Multi-Agent system
MIC	Model-Integrated Computing
MoSL	Mosaik Specification Language
M&S	Modeling and Simulation
M&V	Medium Voltage
MRM	Multi-Resolution Modeling
OCL	Object Constraint Language
OMT	Object Model Template
OMG	Object Management Group
OMG	Object Management Group
OPC UA	OPC Unified Architecture
PV	Photovoltaics
RTI	Run-Time Infrastructure
SG	Schedule Graph

SOA	Service-Oriented Architecture
SOM	Simulation Object Model
SoC	State-of-Charge (of a battery/storage)
SQL	Structured Query Language
UML	Unified Modeling Language
UMPP	Maximum Peak Power Voltage
V2G	Vehicle-To-Grid
XML	Extensible Markup Language
ZMQ	ZeroMQ (a messaging library)

Figures

Tables

Listings

References

[AAV99] Paul M. Anderson, B. L. Agrawal, and J. E. Van Ness. *Subsynchronous Resonance in Power Systems*. Wiley-IEEE Press, 1999.

[AGK06] Colin Atkinson, Matthias Gutheil, and Kilian Kiko. On the Relationship of Ontologies and Models. In *Proceedings of the 2nd International Workshop on Meta-Modelling, WoMM 2006*, pages 47–60, Bonn, 2006.

[Agr06] Poonum Agrawal. *Overview of DOE Microgrid Activities*. Montreal Symposium on Microgrids, 2006.

[Ant03] Jiju Antony. *Design of Experiments for Engineers and Scientists*. Butterworth-Heinemann, 2003.

[Ass11] Association for Standardisation of Automation and Measuring Systems (ASAM). ASAM MCD-2 NET V 4.0.0, 2011.

[BÖ3] Lutz Bölöni. The Bond Agent System. `http://www.cs.ucf.edu/~lboloni/Teaching/EEL5937_MultiAgentSystems_Spring2003/slides/lecture18.ppt`, 2003.

[Ban10] Colin John Bankier. *GridIQ - A Test Bed for Smart Grid Agents*. Bachelor thesis, University of Queensland, 2010.

[Bas08] Hauke Basse. *Spannungshaltung in Verteilnetzen bei Stützung durch dezentrale Erzeugungsanlagen mit lokaler Blindleistungsregelung*. Diploma thesis, Universität Karlsruhe, 2008.

[BB11] Houda Benali and Narjès Bellamine Ben Saoud. Towards a component-based framework for interoperability and composability in Modeling and Simulation. *Simulation*, 87(1-2):133–148, July 2011.

[BBD+13] Marita Blank, Timo Breithaupt, Arne Dammasch, Steffen Garske, and Astrid Nieße. Evaluationsszenarien für die Teilprojekte 1 - 4. `http://smartnord.de/static/downloads/Technischer_Bericht_Evalutionsszenarien_130510.pdf`, 2013.

[BBFF12] Philippe Beaucage, Michael C. Brower, Jaclyn D. Frank, and Jeffrey D. Freedman. Development of a StochasticKinematic Cloud Model to Generate High-Frequency Solar Irradiance and Power Data. In *Proceedings of the 41st World Renewable Energy Forum*. American Solar Energy Society, 2012.

[BCN05] Jerry Banks, John Carson, and Barry L. Nelson. *Discrete-event system simulation*. Prentice Hall International, 4th edition, 2005.

[BCWS11] Jens Bastian, Christoph Clauss, Susann Wolf, and Peter Schneider. Master for Co-Simulation Using FMI. In *8th International Modelica Conference*, Dresden, 2011.

[(BD13] Bundesverband der Energie- und Wasserwirtschaft (BDEW). Standardlastprofile |BDEW| Bundesverband der Energie- und Wasserwirtschaft. `http://www.bdew.de/internet.nsf/id/de_standartlastprofile`, 2013. Accessed: 12 Feb 2013.

[BdBV06a] Csaba Attila Boer, Arie de Bruin, and Alexander Verbraeck. Distributed Simulation in Industry - A Survey Part 1 - The COTS Vendors. In *Proceedings of the 2006 Winter Simulation Conference*, 2006.

[BdBV06b] Csaba Attila Boer, Arie de Bruin, and Alexander Verbraeck. Distributed Simulation in Industry - A Survey Part 2 - Experts on Distributed Simulation. In *Proceedings of the 2006 Winter Simulation Conference*, pages 1061–1068, 2006.

[BEDM98] Perakath Benjamin, Madhav Erraguntla, Dursun Delen, and Richard Mayer. Simulation Modelling at multiple levels of abstraction. In *Proceedings of the 1998 Winter Simulation Conference*, pages 391–398, 1998.

[BGG+10] Johannes Bergmann, Christian Glomb, Jürgen Götz, Jörg Heuer, Richard Kuntschke, and Martin Winter. Scalability of Smart Grid Protocols. In *First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 131–136, Gaithersburg, 2010. IEEE.

[BGL+04] Henrik Bindner, Oliver Gehrke, Per Lundsager, Jens Carsten Hansen, and Tom Cronin. IPSYS - A tool for performance assessment and supervisory controller development of integrated power systems with distributed renewable energy. In *Solar 2004*, Perth, 2004.

[BKS02] Bernhard Beckert, Uwe Keller, and Peter H Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *VERIFY, Workshop at Federated Logic Conferences (FLoC)*, pages 113–123, 2002.

[Boe05] Casba Attila Boer. *Distributed Simulation in Industry*. PhD thesis, Erasmus University Rotterdam, 2005.

[BPA11] Masssimo Bongiorno, Andreas Petersson, and Evert Agneholm. Windforsk - Research area 4, Wind power in the power system - The impact of Wind Farms on Subsynchronous Resonance in Power Systems. Technical Report April, ELFORSK, 2011.

[Bro97] Jan F. Broenink. Bond-graph modeling in modelica. In *Proceedings of ESS'97 - European Simulation Symposium*, 1997.

[Bro02] Alec N. Brooks. Vehicle-to-Grid Demonstration Project: Grid Regulation Ancillary Service with a Battery Electric Vehicle. Technical report, 2002.

[BS13] Jörg Bremer and Michael Sonnenschein. Sampling the Search Space of Energy Resources for Self-organized , Agent-based Planning of Active Power Provision. In *EnviroInfo*, Hamburg, 2013.

[BSA11] Sebastian Beer, Michael Sonnenschein, and Hans-Jürgen Appelrath. Towards a Self-Organization Mechanism for Agent Associations in Electricity Spot Markets. In *Informatik 2011*, 2011.

[Cal05] California State University. Main Effects and Interactions. `http://psych.csufresno.edu/price/psych144/mainint.html`, 2005. Accessed: 10 Jul 2013.

[CASB12] Christoph Clauß, Martin Arnold, Tom Schierz, and Jens Bastian. Master zur Simulatorkopplung via FMI. In *ASIM STS/GMMS Workshop*, 2012.

[CBA95] Gary Cook, Lynn Billman, and Rick Adcock. *Photovoltaic Fundamentals*. National Renewable Energy Laboratory, 1995.

[CBS+11] Joel Chinnow, Karsten Bsufka, Aubrey-Derrick Schmidt, Rainer Bye, Ahmet Camtepe, and Sahin Albayrak. A simulation framework for smart meter security evaluation. In *2011 IEEE International Conference on Smart Measurements for Future Grids (SMFG)*, Bologna, November 2011. IEEE.

[CG12] Jordi Cabot and Martin Gogolla. Object Constraint Language (OCL): A Definitive Guide. In Marco Bernado, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, pages 58–90. Springer, Berlin, 2012.

[Cha09] David P. Chassin. GridLAB-D Overview. `http://mek.sitecore.dtu.dk/upload/centre/cet/det_sker/09/powerevent_200910091500.pdf`, 2009.

[CHD09] Kristien Clement, Edwin Haesen, and Johan Driesen. Coordinated Charging of Multiple Plug-In Hybrid Electric Vehicles in Residential Distribution Grids. *Power Systems Conference and Exposition*, 2009.

[CRE12] CREALYTICS. crealytics/orderly_json_xtext - GitHub. `https://github.com/crealytics/orderly_json_xtext`, 2012. Accessed: 08 May 2012.

[CS99] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, June 1999.

[CSG08] David P. Chassin, K. Schneider, and C. Gerkensmeyer. GridLAB-D: An Open-source Power Systems Modeling and Simulation Environment. In *IEEE 2008 PES Transmission and Distribution Conference and Exposition*. IEEE, 2008.

[CSLS00] Christoph Clauß, A. Schneider, T. Leitner, and P. Schwarz. Modelling of Electrical Circuits with Modelica. In *Proceedings of the Modelica Workshop 2000*, pages 3–12, Lund, 2000.

[CT10] Loredana Carradore and Roberto Turri. Electric Vehicles Participation in Distribution Network Voltage Regulation. In *45th Universities' Power Engineering Conference (UPEC)*, Cardiff, 2010.

[CW09] David P. Chassin and S. E. Widergren. Simulating Demand Participation in Market Operations. In *Power & Energy Society General Meeting, PES'09*, pages 1–5. IEEE, 2009.

[DA06] Turhan Demiray and Göran Andersson. Simulation of Power Systems Dynamics using Dynamic Phasor Models. In *X Symposium of Specialists in Electric Operational and Expansion Planning (SEPOPE)*, Florianópolis, Brazil, 2006.

[DB98] Paul K. Davis and James H. Bigelow. Experiments in Multiresolution Modeling (MRM). `http://www.dtic.mil/dtic/tr/fulltext/u2/a355041.pdf`, 1998.

[DBK12] Volker Diedrichs, Alfred Beekmann, and Marcel Kruse. Reactive Power Capability and Voltage Control with Wind Turbines. In Thomas Ackermann, editor, *Wind Power in Power Systems*, chapter 43, pages 975–997. John Wiley & Sons, 2nd edition, 2012.

[DH10] Larry Dickerman and Jessica Harrison. A New Car, a New Grid. *IEEE Power & Energy Magazine*, (april):55–61, 2010.

[Dig11] Wouter Van Diggelen. *Changing Face-to-Face Communication: Collaborative Tools to Support Small-group Discussions in the Classroom*. Phd thesis, University of Groningen, 2011.

[Dyn05] DynamicDemand. A dynamically-controlled refrigerator. Technical report, 2005.

[Ecl12] Eclipse Foundation. Xtext 2.3 Documentation. `http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf`, 2012.

[ED12] EU-DEEP. EUDEEP: Distributed Energy Resources. `http://www.eu-deep.com/index.php?id=460`, 2012. Accessed: 24 Feb 2013.

[EPR10] (Electric Power Research Institute EPRI). Harmonizing the International Electrotechnical Commission Common Information Model (CIM) and 61850 Standards via a Unified Model: Key to Achieve Smart Grid Interoperability Objectives. Technical report, Palo Alto, CA, 2010.

[Eur09] European Network of Transmission System Operators for Electricity (ENTSO-E). P1-Policy 1: Load-Frequency Control and Performance. `https://www.entsoe.eu/fileadmin/user_upload/_library/publications/entsoe/Operation_Handbook/Policy_1_final.pdf`, 2009.

[Eur10a] European Committee for Electrotechnical Standardization (CENELEC). Voltage characteristics of electricity supplied by public electricity networks, EN 50160, 2010.

[Eur10b] European Technology Platform SmartGrids. Strategic deployment document for europe's electricity networks of the future. Technical report, European Technology Platform SmartGrids, 2010.

[EWP12] Atiyah Elsheikh, Edmund Widl, and Peter Palensky. Simulating Complex Energy Systems With Modelica: A Primary Evaluation. In *6th IEEE International Conference on Digital Ecosystems Technologies (DEST)*, pages 1–6, Campione d'Italia, 2012.

[Fav04] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, pages 1–28, Dagstuhl, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[FF91] Brian Falkenhainer and Kenneth D. Forbus. Compositional modeling: finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.

[FH] James Freeman-Hargi. Rule-Based Systems and Identification Trees. `http://ai-depot.com/Tutorial/RuleBased.html`. Accessed: 15 Jul 2013.

[Fis95] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1995.

[FP10] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison-Wesley, 2010.

[Fuj90] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM - Special issue on simulation*, 33(10):30–53, 1990.

[Fuj00] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000.

[GBB+05] Peter A. J. Gijsbers, E. Brakkee, R. Brinkman, J. B. Gregersen, S. Hummel, and S. J. P. Westen. Part C - the org.OpenMI.Standard interface specification. `http://www.aquo.nl/aspx/download.aspx?File=/publish/pages/2385/c_org.openmi.standard_specification.pdf`, 2005.

[Gen13] General Electric. PSLF. `http://site.ge-energy.com/prod_serv/products/utility_software/en/ge_pslf/index.htm`, 2013. Accessed: 13 Feb 2013.

[Ger07] German Solar Energy Society (DGS). *Planning and Installing Photovoltaic Systems: A Guide for Installers, Architects and Engineers*. Earthscan, 2007.

[Ger08] Mark Germagian. Three-Phase Electric Power Distribution for Computer Data Centers. `http://www.opengatedata.com/pdf/ep901-three-phase-power-distribution-wp2.pdf`, 2008.

[GHJ94] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns*. Addison-Wesley Longman, Amsterdam, 1994.

[GKR+07] Hans Grönninger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Text-based Modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007)*, Nashville, 2007.

[GMD+10] Tim Godfrey, Sara Mullen, Roger C Dugan, Craig Rodine, David W Griffith, and Nada Golmie. Modeling Smart Grid Applications with Co-Simulation. In *First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 291–296, Gaithersburg, 2010.

[Gri08] GridWise Architecture Council. GridWise Interoperability ContextSetting Framework. Technical report, GridWise Architecture Council, 2008.

[GS12] Dennis Grunewald and Stephan Schmidt. NeSSi2 Manual Version 2.0.0-beta.6. Technical report, TU Berlin, DAI Labor, 2012.

[Gua98] Nicola Guarino. *Formal Ontology in Information Systems*. IOS Press, 1998.

[GW10] Giancarlo Guizzardi and Gerd Wagner. Towards an ontological foundation of discrete event simulation. In B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, editors, *Proceedings of the 2010 Winter Simulation Conference*, pages 652–664, 2010.

[Han12] Holly A. H. Handley. The role of architecture frameworks in simulation models: The human view approach. In *Engineering Principles of Combat Modeling and Distributed Simulation*, pages 811–824. John Wiley & Sons, Inc., 2012.

[HDS07] Klaus Heuck, Klaus-Dieter Dettmann, and Detlef Schulz. *Elektrische Energieversorgung - Erzeugung, Übertragung und Verteilung elektrischer Energie für Studium und Praxiy*. Vieweg & Sohn Verlag, Wiesbaden, 7th edition, 2007.

[Hel94] Joseph M. Hellerstein. Practical predicate placement. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 325–335, Minneapolis, 1994. ACM.

[Hen95] Jan Hensen. Modelling Coupled Heat and Air Flow: Ping-Pong vs Onions. In *Proc. 16th AIVC Conference*, Palm Springs, 1995.

[HH12a] Tim Hoerstebrock and Axel Hahn. A Toolbased Approach to Assess Technology Introduction in Transportation Systems Demonstrated by the LNG Introduction for Ship Propulsion. In Hans Otto. Günther, Kap-Hwan Kim, and Herbert Kopfer, editors, *The 2012 International Conference on Logistics and Maritime Systems (LOGMS): Proceedings*, Bremen, 2012.

[HH12b] Tim Hoerstebrock and Axel Hahn. An Approach Towards Service Infrastructure Optimization for Electromobility. In Michael Hülsmann and Dirk Fornahl, editors, *Evolutionary Paths Towards the Mobility Patterns of the Future*. Springer, Heidelberg, 2012.

[HHS13] Tim Hoerstebrock, Axel Hahn, and Jürgen Sauer. Tool based Assessment of Electromobility in Urban Logistics. In *Soft Computing for Business Intelligence*. Springer, 2013.

[HJ08] Kenneth A. Hawick and Heath A. James. Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *Proceedings of the 2008 International Conference on Foundations of Computer Science, FCS 2008*, pages 14–20, Las Vegas, 2008.

[HK12] Su-Youn Hong and Tag Gon Kim. Specification of multi-resolution modeling space for multi-resolution system simulation. *Simulation*, 89(1):28–40, September 2012.

[HL05] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(04):281, November 2005.

[HLR11] Ulf Häger, Sebastian Lehnhoff, and Christian Rehtanz. Distributed coordination of power flow controlling devices in transmission systems. *Automatisierungstechnik*, 59(3):153–160, 2011.

[HNN$^+$12] Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits, and Gabor Karsai. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *Simulation*, 88(2):217–232, March 2012.

[HRU10] J. Himmelspach, M. Röhl, and a. M. Uhrmacher. Component-Based Models and Simulations for Supporting Valid Multi-Agent System Simulations. *Applied Artificial Intelligence*, 24(5):414–442, May 2010.

[HS12] Andreas Heckmann and Sebastian Streit. The Modeling of Energy Flows in Railway Networks using XML-Infrastructure Data. In *Proceedings of the 9th International Modelica Conference*, pages 125–132, Munich, November 2012.

[HSB12] Tim Hoerstebrock, Steffen Schütte, and Marius Buchmann. Integrating fleets of EVs into rural low-voltage grids with a high share of photovoltaic energy. In *VDE KONGRESS 2012 Smart Grid*, Berlin/Offenbach, 2012. VDE Verlag GmbH.

[HSL13] Christian Hinrichs, Michael Sonnenschein, and Sebastian Lehnhoff. Evaluation of a Self-Organizing Heuristic for Interdependent Distributed Search Spaces. In Filipe Joaquim and Ana Fred, editors, *International Conference on Agents and Artificial Intelligence (ICAART 2013)*, pages 25–34, Barcelona, 2013.

[HWG+06] Kenneth Hopkinson, Xiaoru Wang, Renan Giovanini, James Thorp, Kenneth Birman, and Denis Coury. EPOCHS: A Platform for Agent-Based Electric Power and Communication Simulation Built From Commercial Off-the-Shelf Components. *IEEE Transactions on Power Systems*, 21(2):548–558, 2006.

[iMa12] iMatix Corporation. Ømq language bindings. `http://www.zeromq.org/bindings:_start`, 2012. Accessed: 23 Jul 2012.

[Ins90] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries: 610*. IEEE Press, New York, 1990.

[Ins00] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1471-2000 - IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. 2000.

[Ins10a] Institut für angewandte Sozialwissenschaft GmbH (infas). Mobilität in Deutschland 2008 - Tabellenband. Technical report, 2010.

[Ins10b] Institute of Electrical and Electronics Engineers (IEEE). IEEE SA - 1516.1-2012 - IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification. `http://standards.ieee.org/findstds/standard/1516.1-2010.html`, 2010.

[Int07] International Electrotechnical Commission (IEC). OPC Unified Architecture Specification - Part 3: Address Space Model, 2007.

[Int11] International Organization for Standardization (ISO). ISO/IEC 9075-2:2011 - Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation). `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53682`, 2011. Accessed: 18 Jun 2012.

[JB03] Nicholas R. Jennings and Stefan Bussmann. Agent-Based Control Systems - Why Are They Suited to Engineering Complex Systems. *IEEE Control Systems Magazine*, (June):61–73, 2003.

[JK02] Frank Joswig and Stefan Kulig. Subsynchronous resonance in turbine generators caused by faulty induction machines. In *IFToMM*, Sydney, 2002.

[Joh75] Donald B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[JRMRM12] Javier Juárez, Carlos Rodríguez-Morcillo, and José Antonio Rodríguez-Mondéjar. Simulation of iec 61850-based substations under omnet++. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, pages 319–326, ICST, Brussels, Belgium, Belgium, 2012. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[JSO10] JSON Schema. `http://www.json-schema.org/`, 2010. Accessed: 08 May 2012.

[Kam10] Andreas Kamper. *Dezentrales Lastmanagement zum Ausgleich kurzfristiger Abweichungen im Stromnetz*. Phd thesis, Karlsruher Institut für Technologie, 2010.

[Kap08] Marko Kapitza. Design of Experiments. `http://www.scai.fraunhofer.de/fileadmin/ArbeitsgruppeTrottenberg/WS0809/seminar/Kapitza.pdf`, 2008.

[KD06] Willett Kempton and Amardeep Dhanju. Electric Vehicles with V2G: Storage for Large-Scale Wind Power. *Windtech International*, 2(2):18–21, 2006.

[KEJ08] Clement Kristien, Haesen Edwin, and Driesen Johan. The Impact of Uncontrolled and Controlled Charging of Plug-in Hybrid Electric Vehicles on the Distribution Grid. In *EET-2008 European Ele-Drive Conference*, 2008.

[KGC$^+$12] Thomas Konnerth, Dennis Grunewald, Joel Chinnow, Silvan Kaiser, Karsten Bsufka, and Sahin Albayrak. Integration of Simulations and MAS for Smart Grid Management Systems. In *Thrid International Workshop on Agent Technologies of Energy Systems (ATES)*, 2012.

[KHH$^+$10] Gunnar Kaestle, Thomas Hesse, Ralf Hesse, Hans Peter Beck, and Martin Kleimaier. Vorrichtung und Verfahren zum Steuern des Austausches von elektrischer Energie. `http://depatisnet.dpma.de/DepatisNet/depatisnet?action=pdf&docid=DE102010030093A1`, 2010.

[KKM97] Alfred Katzenbach, Matthias Kreutz, and Frank Müller. Digital Mock-up in der PKW-Entwicklung. In *Veranstaltungsdokumentation des 3. CAD/CAM Forum der Daimler-Benz AG, Fellbach*, 1997.

[KKOM04] John D. Kueck, Brendan J. Kirby, Philip N. Overholt, and Lawrence C. Markel. Measurement Practices for Reliability and Power Quality. `http://www.globalregulatorynetwork.org/Resources/measurementpractices0604.pdf`, 2004.

[KN09] Stamatis Karnouskos and Thiago Nass De Holanda. Simulation of a Smart Grid City with Software Agents. *2009 Third UKSim European Symposium on Computer Modeling and Simulation*, pages 424–429, November 2009.

[KN12] Hans Knudsen and Jorgen Nygard Nielse. Introduction to the Modelling of Wind Turbines. In Thomas Ackermann, editor, *Wind Power in Power Systems*, pages 769–797. John Wiley & Sons, 2nd edition, 2012.

[Kni02] Gil Knier. How do Photovoltaics Work? `http://science.nasa.gov/science-news/science-at-nasa/2002/solarcells/`, 2002. Accessed: 15 Dec 2011.

[KPV86] C. Koutsougeras, C. A. Papachristou, and R. R. Vemuri. Data flow graph partitioning to reduce communication cost. In *Proceedings of the 19th annual workshop on Microprogramming*, MICRO 19, pages 82–91, New York, NY, USA, 1986. ACM.

[KRB$^+$10] René Kamphuis, Bart Roossien, Frits Bliek, Jorgen can den Noort, Albert van de Velde, Johan de Wit, and Marcel Eijgelaar. Architectural Design and First Results Evaluation of the PowerMatching City Field Test. In *4th International Conference on Integration of Renewable and Distributed Energy Resources*, Albuquerque, 2010.

[KSW06] Martin Kaltschmitt, Wolfgang Streicher, and Andreas Wiese. *Erneuerbare Energien: Systemtechnik, Wirtschaftlichkeit, Umweltaspekte*. Springer Verlag, Berlin, 4th edition, 2006.

[KT05] Willett Kempton and Jasna Tomic. Vehicle to grid implementation: From stabilizing the grid to supporting large-scale renewable energy. *Journal of Power Sources*, 144(1):280–294, 2005.

[KT07] Willett Kempton and Jasna Tomic. Using fleets of electric-drive vehicles for grid support. *Journal of Power Sources*, 168(2):459–468, 2007.

[Kup08] Friederich Kupzog. *Frequency-responsive load management in electric power grids*. PhD thesis, Vienna University of Technology, 2008.

[KvD11] James Keirstead and Koen H. van Dam. A survey on the application of conceptualisations in energy systems modelling. In Pieter E. Vermaas and Virginia Dignum, editors, *Formal Ontologies Meet Industry (FOMI2011)*, pages 50–62. IOS Press, Delft, the Netherlands, 7–8 July 2011.

[KWK05] Ken Kok, Cor Warmer, and René Kamphuis. PowerMatcher: Multiagent Control in the Electricity Infrastructure. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (AAMAS'05)*, pages 75–82, 2005.

[KWN$^+$10] Koen Kok, Cor Warmer, David Nestle, Patrick Selzam, Anke Weidlich, Stamatis Karnouskos, Aris Dimeas, Stefan Drenkard, Sotiris Hatzimichael, and Anita Buchholz. SmartHouse/SmartGrid - Deliverable D1.1 High-Level System Requirements. `http://www.smartgrid.gov/document/smarthousesmartgrid_d11_high_level_system_requirements`, 2010.

[LAh11] Vincenzo Liberatore and Ahmad Al-hammouri. Smart Grid Communication and Co-Simulation. In *IEEE EnergyTech 2011*, Cleveland, OH, USA, 2011.

[Lar00] Mats Larsson. ObjectStab - A Modelica Library for Power System Stability Studies. In *Proceedings of the Modelica Workshop 2000*, pages 13–22, Lund, 2000.

[Lar12] Ake Larsson. Practical Experience with Power Quality and Wind Power. In Thomas Ackermann, editor, *Wind Power in Power Systems*, pages 195–208. John Wiley & Sons, 2nd edition, 2012.

[Law06] Averill Law. *Simulation Modeling and Analysis*. McGraw Hill Higher Education, 4th edition, 2006.

[LC08] Yingyi Liang and Roy H. Campbell. Understanding and simulating the iec 61850 standard *. `https://www.ideals.illinois.edu/handle/2142/11457`, 2008.

[Lin12] Hua Lin. *Communication Infrastructure for the Smart Grid: A Co-Simulation Based Study on Techniques to Improve the Power Transmission System Functions with Efficient Data Networks*. Phd thesis, Virginia Polytechnic Institute and State University, 2012.

[LKS+04] Björn Löfstrand, Ville Kärkkäinen, Johan Strand, Michael Ericsson, Maj Johansson, and Harald Lepp. Scenario Management - Common Design Principles and Data Interchange Formats. In *Proceedings of 2004 Euro Simulation Interoperability Workshop, 04E-SIW-70*, 2004.

[LMLD11] Weilin Li, Antonello Monti, Min Luo, and Roger A. Dougal. VPNET: A co-simulation framework for analyzing communication channel effects on power systems. In *2011 IEEE Electric Ship Technologies Symposium (ESTS)*, pages 143–149, April 2011.

[LSS+11] Hua Lin, Santhoshkumar Sambamoorthy, Sandeep Shukla, James Thorp, and Lamine Mili. Power System and Communication Network Co-Simulation for Smart Grid Applications. In *Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES*, pages 1–6, 2011.

[Lue12] Ontje Luensdorf. *Selbstorganisation virtueller Geräte für das Lastmanagement von Kleinverbrauchern*. Phd thesis, Carl von Ossietzky Universität Oldenburg, 2012.

[LW00] Klaus Lamberg and Peter Wältermann. Using HIL Simulation to Test Mechatronic Components in Automotive Engineering, dSPACE GmbH. `http://www.dspaceinc.com/ftp/papers/hdt00_e.pdf`, 2000.

[LWSF11] Tobias Linnenberg, Ireneus Wior, Sebastian Schreiber, and Alexander Fay. A market-based multi-agent-system for decentralized power and grid control. In *16th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, 2011.

[MA12] Poul Erik Morthorst and Thomas Ackermann. Economic Aspects of Wind Power in Power Systems. In Thomas Ackermann, editor, *Wind Power in Power Systems*, pages 489–516. John Wiley & Sons, 2nd edition, 2012.

[MAE+04] Alan W. McMorran, Graham W. Ault, Ian M. Elders, Colin E.T. Foote, Graeme M. Burt, and James R. McDonald. Translating CIM XML Power System Data to a Proprietary Format for System Simulation. *IEEE Transactions on Power Systems*, 19(1):229–235, February 2004.

[MB04] Ruth Malan and Dana Bredemeyer. Conceptual Architecture. `http://www.bredemeyer.com/ArchitectingProcess/ConceptualArchitecture.htm`, 2004. Accessed: 03 May 2012.

[MDC+07] Stephen D. J. McArthur, Euan M. Davidson, Victoria M. Catterson, Aris L. Dimeas, Nikos D. Hatziargyriou, Ferdinanda Ponci, and Toshihisa Funabashi. Multi-Agent Systems for Power Engineering Applications - Part I: Concepts, Approaches, and Technical Challenges. *IEEE Transactions on Power Systems*, 22(4):1743–1752, November 2007.

[MH02] Tino Miegel and Richard Holzmeier. High Level Architecture. `http://ifgi.uni-muenster.de/~bernard/public/InteropArchitekturen2003/HLA_Ausarbeitung.pdf`, 2002.

[MHC07] John A. Miller, Congzhou He, and Julia I. Couto. Impact of Semantic Web on Modeling and Simulation. In *Handbook of Dynamic System Modeling*, chapter 3. 2007.

[MHG+12] Sven C. Müller, Ulf Häger, Hanno Georg, Sebastian Lehnhoff, Christian Rehtanz, Christian Wietfeld, Horst F. Wedde, and Thomas Zimmermann. Einbindung von intelligenten Entscheidungsverfahren in die dynamische Simulation von elektrischen Energiesystemen. *Informatik-Spektrum*, 36(1):6–16, December 2012.

[Mil10] Federico Milano. *Power System Modelling and Scripting*. Springer, London, 2010.

[Mod] Modelica Association. Modelica and the Modelica Association. `https://www.modelica.org/`. Accessed: 01 Oct 2012.

[Mod12a] Modelica Association. 9th International Modelica Conference - Modelica Association. `https://modelica.org/events/modelica2012`, 2012. Accessed: 15 Feb 2013.

[MOD12b] MODELISAR. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_ModelExchange_and_CoSimulation_v2.0_Beta4.pdf`, 2012. Accessed: 24 July 2013.

[Moo10] Roger Moore. OpenMI Standard 2 Specification. `http://openmi.org/reloaded/standard/specification.php`, 2010.

[Mor08] Farshad Moradi. *A Framework for Component Based Modelling and Simulation using BOMs and Semantic Web Technology*. Phd thesis, KTH Royal Institute of Technology, 2008.

[MT00] Todd McDonald and Michael L Talbert. Agent-based Architecture for Modeling and Simulation Integration. In *Proceedings of the National Aerospace & Electronics Conference (NAECON 2000)*, pages 375–382, 2000.

[Nat05] National Institute of Standards and Technology (NIST). Engineering Statistics Handbook - 5.2.1.2. One variable at a time. `http://atomic.phys.uni-sofia.bg/local/nist-e-handbook/e-handbook/pri/section2/pri212.htm`, 2005. Accessed: 10 Jul 2013.

[Nat10] National Institute of Standards and Technology (NIST). NIST Framework and Roadmap for Smart Grid Interoperability Standards, Release 1.0. Technical report, U.S. Department of Commerce, 2010.

[Nat11] National Renewable Energy Laboratory (NREL). Getting Started Guide for HOMER Version 2.68. `http://homerenergy.com/pdf/homergettingstarted268.pdf`, 2011.

[NDWB06] Scott Neumann, Arnold DeVos, Steve Widergren, and Jay Britton. Use of the CIM Ontology. `http://cimug.ucaiug.org/KB/KnowledgeBase/Use_of_the_CIM_Ontology_DistribuTech_2006.pdf`, 2006.

[NeM11] Aktuelle Projekte: Modellregion Elektromobilität Bremen/Oldenburg. `http://www.modellregion-bremen-oldenburg.de/de/projekte-und-aktivitaeten/aktuelle-projekte/nemoland.html`, 2011. Accessed: 04 Jun 2013.

[NKM+07] James Nutaro, Phani Teja Kuruganti, Laurie Miller, Sara Mullen, and Mallikarjun Shankar. Integrated Hybrid-Simulation of Electric Power and Communications Systems. *2007 IEEE Power Engineering Society General Meeting*, June 2007.

[NM02] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 77–88, New York, NY, USA, 2002. ACM.

[NNL98] Hyacinth S. Nwana, Divine T. Ndumu, and Lyndon C. Lee. ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.526&rep=rep1&type=pdf`, 1998.

[NPS+10] Joachim Nitsch, Thomas Pregger, Yvonne Scholz, Tobias Naegler, Michael Sterner, Norman Gerhardt, Amany von Oehsen, Carsten Pape, Yves-Marie Saint-Drenan, and Bernd Wenzel. Langfristszenarien und Strategien für den Ausbau der erneuerbaren Energien in Deutschland bei

Berücksichtigung der Entwicklung in Europa und global - "Leitstudie 2010". Technical report, 2010.

[NQX^{+}12] Dao Viet Nga, Do Nguyet Quang, Chee Yung Xuen, Lai Lee Chee, and Ong Hang See. Visualization Techniques in Smart Grid. In *2012 International Conference on Smart Grid Systems*, pages 22–26, 2012.

[NS13] Astrid Nieße and Michael Sonnenschein. Using Grid Related Cluster Schedule Resemblance for Energy Rescheduling Goals and Concepts for Rescheduling of Clusters in Decentralized Energy Systems. In *Proceedings of SMARTGREENS 2013 - International Conference on Smart Grids and Green IT Systems*, Aachen, 2013.

[NTS^{+}11] Astrid Nieße, Martin Tröschel, Stefan Scherfke, Steffen Schütte, and Michael Sonnenschein. Using Electric Vehicle Charging Strategies to Maximize PV integration in the Low Voltage Grid. In *6th International Renewable Energy Storage Conference and Exhibition (IRES 2011)*, 2011.

[Nut11a] James Nutaro. Designing power system simulators for the smart grid: Combining controls, communications, and electro-mechanical dynamics. In *2011 IEEE Power and Energy Society General Meeting*, San Diego, July 2011. IEEE.

[Nut11b] James J. Nutaro. *Building Soaftware for Simulation: theory and algorithms with applications in C++*. John Wiley & Sons, Hoboken, 2011.

[Obj07] Object Management Group. UML 2.1.2. `http://www.omg.org/spec/UML/2.1.2/`, 2007. Accessed: 15 Jun 2012.

[Obj12] Object Management Group. OMG Object Constraint Language (OCL). `http://www.omg.org/spec/OCL/2.3.1`, 2012.

[Ord10] ORDERLY JSON. `http://orderly-json.org/`, 2010. Accessed: 08 May 2012.

[Ott12] Malte Otten. *Organisationsparadigmen bei der kontinuierlichen Bedarfsermittlung für die reaktive Einsatzplanung in dezentralen Energiesystemen*. Bachelor thesis, Universität Oldenburg, 2012.

[Pac] Pacific Northwest National Laboratory (PNNL). Grid Friendly Appliance Controller - Available Technologies - PNNL. `http://availabletechnologies.pnnl.gov/technology.asp?id=61`. Accessed: 24 May 2012.

[Pac09] Pacific Northwest National Laboratory (PNNL). GridLAB-D Course - Runtime Classes. `http://sourceforge.net/projects/gridlab-d/files/gridlabd-class/`, 2009.

[Pac12] Pacific Northwest National Laboratory (PNNL). A Unique Tool to Design the Smart Grid. Technical report, Pacific Northwest National Laboratory (PNNL), 2012.

[Pac13a] Pacific Northwest National Laboratory (PNNL). GridLAB-D Simulation Software. `http://www.gridlabd.org/`, 2013. Accessed: 14 Feb 2013.

[Pac13b] Pacific Northwest National Laboratory (PNNL). GridLAB-D: User's Manual. `http://www.gridlabd.org/documents/doxygen/1.1/user_manual.html`, 2013. Accessed: 14 Feb 2013.

[Pag07] Ernest H Page. Theory and Practice for Simulation Interconnection: Interoperability and Composability in Defense Simulation. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modeling*, chapter 16. Chapman & Hall, 2007.

[PBT04] Ernest H. Page, Richard Briggs, and John A. Tufarolo. Toward a Family of Maturity Models for the Simulation Interconnection Problem. In *Proceedings of the Spring 2004 Simulation Interoperability Workshop*. IEEE CS Press, 2004.

[PFR09] Manisa Pipattanasomporn, Hassan Feroze, and Saifur Rahman. Multi-agent systems in a distributed smart grid: Design and implementation. In *2009 IEEE/PES Power Systems Conference and Exposition*. IEEE, March 2009.

[Pid02] Michael Pidd. Simulation Software and Model Reuse: A Polemic. In E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 772–775. USA: Association for Computing Machinery Press, 2002.

[PITY12] Joel Petersson, Pär Isaksson, Hubertus Tummescheit, and Johan Ylikiiskilä. Modeling and Simulation of a Vertical Wind Power Plant in Dymola/Modelica. In *Proceedings of the 9th International Modelica Conference*, pages 631–640, Munich, November 2012.

[PO99] Ernest H. Page and Jeffrey M. Opper. Observations on the complexity of composable simulation. In Phillip A. Farrington, Harriet Black Nembhard, David T. Sturrock, and Gerald W. Evans, editors, *Proceedings of the 1999 Winter Simulation Conference*, pages 553–560, ACM, New York, NY, USA, 1999.

[PQ 13] PQ Soft. PSCAD. `http://www.pqsoft.com/pscad/index.htm`, 2013. Accessed: 13 Feb 2013.

[PRvdL99] David Pratt, Charles Ragusa, and Sonia von der Lippe. Composability as an architecture driver. In *Proceedings of the 1999 I/ITSEC Conference*, 1999.

[PT06] Dirk Pawlaszczyk and Ingo J. Timm. A Hybrid Time Management Approach to Agent-based Simulation. In *29th Annual German Conference on Artificial Intelligence (KI 2006)*, volume 4314, pages 374–388, 2006.

[PW03a] Mikel D. Petty and Eric W. Weisel. A Composability Lexicon. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop, 03S-SIW-023*, 2003.

[PW03b] Mikel D. Petty and Eric W. Weisel. A Formal Basis For a Theory of Semantic Composability. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop, 03S-SIW-054*, 2003.

[PWM03] Mikel D. Petty, Eric W. Weisel, and Roland R. Mielke. Computational Complexity of Selecting Components for Composition. In *Proceedings of the Fall 2003 Simulation Interoperability Workshop, 03F-SIW-072*, 2003.

[PWM05] Mikel D. Petty, Eric W. Weisel, and Roland R. Mielke. Composability theory overview and update. In *Proceedings of the Spring 2005 Simulation Interoperability Workshop, 05S-SIW-063*, 2005.

[Qua12] Gang Quan. Data Flow Graphs Intro. `http://web.cecs.pdx.edu/~mperkows/temp/JULY/data-flow-graph.pdf`, 2012.

[RÖ8] Mathias Röhl. *Definition und Realisierung einer Plattform zur modellbasierten Komposition von Simulationsmodellen.* Phd thesis, Universität Rostock, 2008.

[RAF+00] George F. Riley, Mostafa H. Ammar, Richard Fujimoto, Kalyan Perumalla, and Donghua Xu. Distributed Network Simulations using the Dynamic Simulation Backplane. In *Proceedings of the 21st Annual Conference on Distributed Computing Systems*, 2000.

[RDA+08] A. Rizzoli, M. Donatelli, I. Athanasiadis, F. Villa, and D. Huber. Semantic links in integrated modelling frameworks. *Mathematics and Computers in Simulation*, 78(2-3):412–423, July 2008.

[RN03] Stuard Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson, 2nd edition, 2003.

[RNP+04] Stewart Robinson, Richard E. Nance, Ray J. Paul, Michael Pidd, and Simon J. E. Taylor. Simulation model reuse: definitions, benefits and obstacles. *Simulation Modelling Practice and Theory*, 12(7-8):479–494, November 2004.

[Roh13] Henning Rohlfs. *Entwicklung eines Konzeptes zur Integration agentenbasierter Koordinationsverfahren in das Smart Grid-Simulationsframework mosaik.* Diploma thesis, Universität Oldenburg, 2013.

[ROSS06] Krzysztof Rudion, Antje Orths, Zbigniew A. Styczynski, and Kai Strunz. Design of benchmark of medium voltage distribution network for investigation of DG integration. In *Power Engineering Society General Meeting, 2006.* IEEE, 2006.

[RTI00] RTI 1.3-Next Generation Programmer's Guide. `http://sslab.cs.nthu.edu.tw/~fppai/HLA/RTI1.3/RTI_NG13_ProgramerGuide.pdf`, 2000.

[RUA10] Sebastian Rohjans, Mathias Uslar, and Hans-Jürgen Appelrath. OPC UA and CIM: Semantics for the smart grid. In *IEEE PES Transmission and Distribution Conference and Exposition 2010*. IEEE, April 2010.

[RVRJ12] Sarvapali D. Ramchurn, Perukrishnen Vytelingum, Alex Rogers, and Nicholas R. Jennings. Putting the "Smarts” into the Smart Grid: A Grand Challenge for Artificial Intelligence. *Communications of the ACM*, 55(4):86–97, 2012.

[RWE] RWE Power AG. Pumpspeicherkraftwerk Herdecke. `http://www.rwe.com/web/cms/mediablob/de/346008/data/183748/5/rwe/innovation/projekte-technologien/energiespeicher/pumpspeicher-herdecke.pdf`.

[SBD] Eckhard Scholz, Christian Burkhardt, and Sascha Dietrich. Digital Mock-Up in der Produktentwicklung. `http://fbme.htwk-leipzig.de/fileadmin/fbme/informationen/TVorstellung/DMU.pdf`.

[SBO+06] Ramakrishna Soma, Amol Bakshi, Abdollah Orangi, Viktor K. Prasanna, and Will Da Sie. A Service Oriented Data Composition Architecture for Integrated Asset Management. In *SPE Intelligent Energy Conference and Exhibition*, Amsterdam, 2006.

[SBP07] Ramakrishna Soma, Amol Bakshi, and Viktor K. Prasanna. A Semantic Framework for Integrated Asset Management in Smart Oilfields. *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid ’07)*, pages 119–126, May 2007.

[SBP+08] Rainer Stamminger, Gereon Broil, Christiane Pakula, Heiko Jungbecker, Maria Braun, Ina Rüdenauer, and Christoph Wendker. Synergy Potential of Smart Appliances. Technical report, Rheinische Friedrich-Wilhelms-Universität Bonn, Bonn, 2008.

[Sch09] Kai Schulte. *Ein Werkzeug für die Erstellung und kennzahlbasierte Bewertung von Siedlungsszenarien mit unterschiedlicher Durchdringung und Betriebsweise dezentraler Energieumwandlungsanlagen*. PhD thesis, Universität Oldenburg, 2009.

[SG310] SG3 - SMB Smart Grid Strategic Group. IEC Smart Grid Standardization Roadmap. Technical Report June, 2010.

[SGW08] Arne Schuldt, Jan D. Gehrke, and Sven Werner. Designing a Simulation Middleware for FIPA Multiagent Systems. In Lakhmi Jain, Pawan Lingras, Matthias Klusch, Jie Lu, Chengqi Zhang, Nick Cercone, and Longbin Cao, editors, *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 109–113. IEEE Computer Society, December 2008.

[Sha02] Mary Shaw. What Makes Good Research in Software Engineering? *International Journal of Software Tools for Technology Transfer*, 4(1):1–7, 2002.

[Sim06] Simulation Interoperability Standards Organization. Base Object Model (BOM) Template Specification. `http://www.sisostds.org/ProductsPublications/Standards/SISOStandards.aspx`, 2006.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *Computer*, 30(4):110–111, 1997.

[SKdJdK11] Oliver Schmitz, Derek Karssenberg, Kor de Jong, and Jean-Luc de Kok. Constructing integrated models: a scheduler to execute coupled components. In *European Geosciences Union, EGU General Assembly*, pages 1–10, Wien, 2011.

[SL10] Gunnar Schulte-Loh. *Entwicklung eines Simulationsmodells zur Nachbildung von Fahrzeugbewegungen auf Basis statistischer Daten*. Diploma thesis, Universität Oldenburg, 2010.

[SMH+10] Gregory A. Silver, John A. Miller, M. Hybinette, G. Baramidze, and William S. York. DeMO: An Ontology for Discrete-event Modeling and Simulation. *Simulation*, 87(9):747–773, December 2010.

[SMJ02] Peter Spyns, Robert Meersman, and Mustafa Jarrar. Data modelling versus ontology engineering. *ACM SIGMOD Record*, 31(4):12, December 2002.

[SNo12] Smart Nord - Wilkommen. `http://www.smartnord.de/`, 2012. Accessed: 09 May 2012.

[Sol82] Henk G. Sol. *Simulation in Information Systems Development*. Phd thesis, University of Groningen, 1982.

[Sou13] SourceForge. GridLAB-D - Browse /gridlab-d/Last stable release at SourceForge.net. `http://sourceforge.net/projects/gridlab-d/files/gridlab-d/Laststablerelease/`, 2013. Accessed: 14 Feb 2013.

[SRK+11] Marcelo G. Simoes, Robin Roche, Elias Kyriakides, Abdellatif Miraoui, Benjamin Blunier, Kerry D. McBee, Siddharth Suryanarayanan, Phuong Nguyen, and Paulo Ribeiro. Smart-grid technologies and progress in Europe and the USA. *2011 IEEE Energy Conversion Congress and Exposition*, pages 383–390, September 2011.

[SS12] Stefan Scherfke and Steffen Schütte. mosaik - Architecture Whitepaper. `http://mosaik.offis.de/downloads/mosaik_architecture_2012.pdf`, 2012.

[SSW+10] Stefan Scherfke, Steffen Schütte, Carsten Wissing, Astrid Nieße, and Martin Tröschel. Simulationsbasierte Untersuchungen zur Integration von Elektrofahrzeugen in das Stromnetz. In *VDE|ETG-Fachtagung "Smart Cities" zu Elektromobilität und intelligent vernetztem Heim*, Leipzig, 2010.

[ST07] Claudia Szabo and Yong Meng Teo. On Syntactic Composability and Model Reuse. *First Asia International Conference on Modelling & Simulation (AMS'07)*, (March):230–237, March 2007.

[Str96] David. L. Streiner. Maintaining standards: differences between the standard deviation and standard error, and when to use each. *Canadian journal of psychiatry. Revue canadienne de psychiatrie*, 41(8):498–502, October 1996.

[Str99] Steffen Straßburger. On the HLA-based Coupling of Simulation Tools. `http://www.strassburger-online.de/papers/S1_01.pdf`, 1999.

[Syr05] Elisabeth Syrjakow. *Eine Komponentenarchitektur zur Integration heterogener Modellierungswerkzeuge*. Phd thesis, Universität Fridericiana zu Karlsruhe, 2005.

[Tel12] Telecom Italia SpA. Jade - Java Agent DEvelopment Framework. `http://jade.tilab.com/`, 2012. Accessed: 09 May 2012.

[The97] The Open Group. crontab. `http://pubs.opengroup.org/onlinepubs/007908799/xcu/crontab.html`, 1997. Accessed: 28 Feb 2013.

[Tho11] Karsten Thoms. Language Workbench Competition 2011 Xtext Submission. `http://lwc11-xtext.eclipselabs.org.codespot.com/files/LWC11-XtextSubmission-v1.2.pdf`, 2011.

[THR+06] Martin Törngren, Dan Henriksson, Ola Redell, Christoph Kirsch, Jad El-khoury, Daniel Simon, Yves Sorel, Hanzalek Zdenek, and Karl-Erik Årzén. Co-design of Control Systems and their real-time implementation - A Tool Survey. Technical report, Royal Institute of Technology, KTH, Stockholm, 2006.

[TM03] Andreas Tolk and James A Muguira. The Levels of Conceptual Interoperability Model (LCIM). In *Proceedings of the Simulation Interoperability Workshop, 03F-SIW-007*, September 2003.

[TM04] Andreas Tolk and James A. Muguira. M & S within the Model Driven Architecture. In *Interservice/Industry Training, Simulation and Education Conference*, 2004.

[TO10] Okan Topçu and Halit Oguztüzün. Scenario Management Practices in HLA-based Distributed Simulation. *Journal of Naval Science and Engineering*, 6(2):1–33, 2010.

[Tol06] Andreas Tolk. What comes after the Semantic Web? PADS implications for the Dynamic Web. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, number 1, pages 55–62, 2006.

[Tol10] Andreas Tolk. Interoperability and Composability. In John A. Sokolowski and Catherine M. Banks, editors, *Modeling and Simulation Fundamentals*

- *Theoretical Underpinnings and Practical Domains*, chapter 12, pages 403–433. John Wiley & Sons, 2010.

[Trö10] Martin Tröschel. *Aktive Einsatzplanung in holonischen Virtuellen Kraftwerken*. Phd thesis, Universität Oldenburg, 2010.

[TS07] Yong Meng Teo and Claudia Szabo. CODES: An Integrated Approach to Composable Modeling and Simulation. Technical report, Asia Pacific Science and Technology Center, Singapore, 2007.

[TSS$^+$11] Martin Tröschel, Stefan Scherfke, Steffen Schütte, Hans-Jürgen Appelrath, and Michael Sonnenschein. Maximierte PV-Integration in Niederspannungsnetzen durch intelligente Nutzung von Elektrofahrzeugen. In *Internationaler ETG-Kongress*, 2011.

[TT08] Charles Turnitsa and Andreas Tolk. Knowledge representation and the dimensions of a multi-model relationship. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pages 1148–1156, 2008.

[TTD08] Andreas Tolk, Charles Turnitsa, and Saikou Diallo. Implied ontological representation within the levels of conceptual interoperability model. *Simulation*, 2:3–19, 2008.

[TTDW06] Andreas Tolk, Charles D. Turnitsa, Saikou Y. Diallo, and Leslie S. Winters. Composable m&s web services for netcentric applications. *Journal for Defense Modeling and Simulation*, 3(1):27–44, 2006.

[TU 13] TU Wien. EA: ADRES-Concept. `http://www.ea.tuwien.ac.at/projekte/adres_concept/`, 2013. Accessed: 07 Feb 2013.

[Tur05] Charles D. Turnitsa. Extending the Levels of Conceptual Interoperability Model. In *IEEE Summer Computer Simulation Conference*. IEEE CS Press, 2005.

[TV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 3rd edition, 2011.

[TWH$^+$03] James S. Thorp, Xiaoru Wang, Kenneth M. Hopkinson, Denis Coury, and Renan Giovanini. Agent Technology Applied to the Protection of Power Systems. In *Autonomous Systems and Intelligent Agents in Power System Control and Operation*, chapter 7, pages 115–154. Springer Verlag, 2003.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, 1988.

[Uni13a] University of South Carolina. VTB Overview. `http://vtb.engr.sc.edu/vtbwebsite/#/Overview`, 2013. Accessed: 13 Feb 2013.

[Uni13b] University of Southern California. The Network Simulator - ns-2. `http://www.isi.edu/nsnam/ns/`, 2013. Accessed: 13 Feb 2013.

[URS+09] Mathias Uslar, Sebastian Rohjans, Tanja Schmedes, José M. Gonzales, Petra Beenken, Tobias Weidelt, Michael Specht, Christoph Mayer, Astrid Nieße, Jens Kamenik, Claas Busemann, Karlheinz Schwarz, and Franz Hein. Untersuchung des Normungsumfeldes zum BMWi-Förderschwerpunkt E-Energy - IKT-basiertes Energiesystem der Zukunft. Technical report, 2009.

[U.S09] U.S. Department of Energy. Smart Grid System Report. Technical report, 2009.

[USR+13] Mathias Uslar, Michael Specht, Sebastian Rohjans, Jörn Trefke, Christian Dänekas, José M. Gonzáles, Christine Rosinger, and Robert Bleiker. *Standardization in Smart Grids*. Springer, 2013.

[Van] Vanderbilt University. GME: Generic Modeling Environment. `http://www.isis.vanderbilt.edu/Projects/gme/`. Accessed: 07 Jun 2012.

[VP02] Kristofer Vorwerk and Glenn Norman Paulley. On Implicate Discovery and Query Optimization. In *Proceedings of the 2002 International Symposium on Database Engineering & Applications*, pages 2–11. IEEE Computer Society, 2002.

[vR12] Jeffery von Ronne. Simulation: Overview and Taxonomy. `http://www.cs.cmu.edu/~tcortina/15110sp12/Unit12PtB.pdf`, 2012. Accessed: 01 August 2013.

[WJ95] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[WKK+09] Cor Warmer, Koen Kok, Stamatis Karnouskos, Anke Weidlich, David Nestle, Patrick Selzam, Jan Ringelstein, Aris Dimeas, and Stefan Drenkard. Web services for integration of smart houses in the smart grid. In *Grid-Interop - The road to an interoperable grid*, Denver, Colorado, USA, 2009.

[Woo09] Michael Wooldridge. *An Introduction To MultiAgent Systems*. John Wiley & Sons, 2nd edition, 2009.

[WTW09] Wenguang Wang, Andreas Tolk, and Weiping Wang. The Levels of Conceptual Interoperability Model: Applying Systems Engineering Principles to M & S. In *Spring Simulation Multiconference (SpringSim'09)*, San Diego, 2009.

[ZH07] Bernard P. Zeigler and Philip E. Hammonds. *Model and Simulation-Based Data Engineering*. Academic Press, New York, 2007.

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.

Index