Master's thesis:

Development of model-based control applications compliant with IEC 61499 for building energy systems with a focus on photovoltaics

Marc Jakobi
6th October 2017
FB1, Master Regenerative Energien, HTW Berlin

Supervision:
Prof. Dr.-Ing. Volker Quaschning
M.Sc. Tjarko Tjaden

Abstract

Political marginal conditions for PV systems, such as feed-in limitations have resulted in the need for intelligent operation strategies. Proprietary solutions available on the market today are costly and intelligent controllers for building energy systems can thus be classified as luxury products. There is a need for generic software based on standards for the control of multi-generator systems. This thesis aims to provide an open source solution that can be used on a variety of inexpensive hardware. Communication libraries that enable co-simulations between IEC 61499 control systems and simulation software (Matlab® and Polysun®) are developed. Using the libraries and simulation tools, models and algorithms are transferred to IEC 61499 control applications with the industry compatible, open source environment 4diac. The applications are then deployed and prepared for use in the field.

Acknowledgements

Preface

Typing conventions of this document

Since this thesis in part provides a detailed description of source code, the following special text formatting is used extensively:

- Source code, variables, objects and properties are formatted in `fixed-width` font and colour coded according to the respective programming language. The same applies to IEC 61499 function block names, inputs, outputs algorithms and states.

- Functions (methods) are formatted in `fixed-width` font with brackets added at the end, e.g., `helloWorld()`.

- Formulas, mathematical symbols, physical units and constants are formatted according to the standards DIN 1338, DIN 1304, DIN 1301 and DIN 1313.

- If a formula, symbol, unit or constant is used as a variable within source code, the `fixed-width` font is used instead of the above.

Source code and terminology

Object oriented programming (OOP) and design pattern terminology is used frequently for the source code documentation. It is assumed that the reader has an understanding of OOP and the basic design patterns. Less commonly used design patterns are described briefly. For a deeper understanding of the code documentation, it is recommended that the reader learn how to interpret design pattern catalogues.

The source code of the libraries developed within the scope of this thesis can be downloaded from the following URLs:

- tcpip4diac: Matlab - IEC 61499 communication library
  `github.com/MrcJkb/tcpip4diac/`

- Polysun4diac: Polysun - IEC 61499 communication plugin
  `github.com/MrcJkb/Polysun-4diac-ControllerPlugin/`

- IEC 61499 function block library and a selection of control applications
  `github.com/MrcJkb/PVTControllerLib/`

- HTTP communication layer for 4diac-RTE
  `github.com/MrcJkb/forte_http_comm/`

- EEBus "SPINE" and "SHIP" communication layers for 4diac-RTE
  `github.com/MrcJkb/forte_spine_comm/`

Contents

List of Figures

List of Tables

Acronyms

API   application programming interface

ASCII   American Standard Code for Information Interchange

BFB   basic function block

CLI   command-line user interface

CSIFB   communication service interface function block

CSV   comma separated values

CFB   composite function block

DOY   day of the year

DSM   demand side management

ECC   excecution control chart

FB   function block

FBD   function block diagram

GUI   graphical user interface

HMI   human machine interface

HTTP   hypertext transfer protocol

IDE   integrated development environement

IL   instruction list

I/O   Input/Output

IP   Internet protocol

JSON   JavaScript object notation

LD   ladder diagram

LSB   least significant byte

mDNS   multicast Domain Name System

MPP   maximum power point

MQTT   Message Queue Telemetry Transport

MSB most significant byte

MVC Model View Controller

NTP network time protocol

OO object oriented

OOP object oriented programming

OPC DA OPC Data Access

OPC UA OPC Unified Architecture

OSI Open Systems Interconnection

P proportional

PID proportional-integral-derivative

PLC programmable logic controller

POSIX Portable Operating System Interface

PV photovoltaic

REST representational state transfer

RTE runtime environment

RTU remote terminal unit

SFC sequential function chart

SG smart grid

SHIP smart home IP

SIFB service interface function block

SPINE Smart Premises Interoperable Neutral-message Exchange

SSH secure shell

ST structured text

TCP Transmission Control Protocol

TD time of day

UDP User Datagram Protocol

UPnP   universal plug and play

UML   Unified Modeling Language

XML   Extensible Markup Language

List of symbols

Symbol	Unit	Description
a	%	Degree of self-sufficiency (autarchy)
C_{bu}	Wh	Usable battery capacity
C_{bn}	Wh	Nominal battery capacity
df	%	Derating (curtailment) factor
E	Wh	Energy
$e(t)$	-	Control error
E_{bat}	Wh	Energy stored within the battery
E_{gf}	kWh	Grid feed-in energy
E_{gs}	kWh	Grid supply energy
$E_{load,f}$	Wh	Load forecast
$E_{pv,f}$	Wh	PV forecast
$f_{on/off}$	%	On/Off threshold of the SG Ready heat pump controller
G	W/m²	Solar irradiance
K_i	-	Coefficient for the derivative term of a PID weighted sum
K_i	-	Coefficient for the integral term of a PID weighted sum
K_p	-	Coefficient for the proportional term of a PID weighted sum
k_T	-	Clearness index
k_{Tf}	-	Forecast clearness index
$\overline{k}_{Tf,15}$	-	15 min mean of k_{Tf}
l	%, kWh	Curtailment losses
P	W	Power
$P_{bat,f}$	W	Battery charging power (optimized value)
P_{bat}	W	Battery charging/discharging power
$P_{bat,c}$	W	Battery charging power (set value after error control)
$P_{bat,d}$	W	Battery discharging power (set value after error control)
$P_{d,f}$	W	Power difference forecast
P_f	W	Power forecast
P_{gf}	W	Feed-in power
P_{gfl}	W	Feed-in limit
p_{gfl}	kW/kWp	Specific feed-in limit
P_{gsl}	W	Grid supply limit
$P_{HP,nom}$	kW	Nominal power of a heat pump (electrical)
P_{load}	W	Load
$\overline{P}_{load,15}$	W	15 min mean of the load

... continued

Symbol	Unit	Description
$P_{\text{load,fP}}$	W	Daily persistence forecast of the load
P_{pv}	W	PV power
$P_{\text{pv,f}}$	W	PV power forecast
$P_{\text{pv,max}}$	W	Maximum PV power of the last 10 days (clear sky approximation)
$\overline{P}_{\text{pv,max15}}$	W	15 min mean of the clear sky approximation $P_{\text{pv,max}}$
$P_{\text{pv,cs}}$	W	Clear sky PV generation
P_{STC}	kWp	Nominal PV power
s	%	Self-consumption ratio
SoC	%	State of charge
T	°C	Temperature
t	s	Time
t_0	s	Beginning of a time frame
t_{b}	s	Beginning of an averaging interval
t_{e}	s	End of an averaging interval
t_{l}	s	Point in time of the last data sample arrival
t_{opt}	s	Time of optimization
$u(t)$	-	Controller output
w	-	Weighting function
w_{lfP}	-	Weighting function for $P_{\text{load,fP}}$
w_{lc}	-	Weighting function for $\overline{P}_{\text{load,15}}$
$\overline{x}$	-	Average
η_{bat}	-	Battery efficiency
η_{binv}	-	Battery inverter efficiency
η_{c}	-	Efficiency when charging
η_{d}	-	Efficiency when discharging
τ_i	-	Point in time at which an output is issued by a function block

1. Introduction

1.1. Motivation

Decreasing feed-in tariffs and the combination of declining PV energy costs and increasing grid electricity prices are driving the trend for PV systems further in the direction of self-consumption. As a result, home-owners are incorporating batteries and heat pumps into their systems in order to increase the use of excess energy, rather than sell it to the grid. However, political marginal conditions such as feed-in limitations have resulted in the need for more complex, intelligent operational strategies. Today, there is a broad selection of smart energy managers and controllers available on the market. Their main tasks are to shift the energy consumption to times of high PV production by means of forecast based battery operation and/or demand side management (DSM). The development of such control applications can be a long and tedious process. Often, the control algorithms are first developed and simulated in a prototyping phase, using tools such as Matlab®. Then, the applications must be ported to a programming language that can be understood by the controller hardware and validated again in field tests. This can be a time consuming and challenging process.

The proprietary solutions available on the market today are expensive; thus intelligent controllers for building energy systems can be classified as luxury products. An open source solution, the "PVprog" algorithm developed at HTW Berlin [1], exists in the form of Matlab® simulation source code. It needs to be ported to other programming languages before being applied in the field. There is currently a lack of generic software based on standards for the control of multi-generator systems.

1.2. Objectives

This thesis aims to facilitate the development of intelligent control systems by developing open source libraries that allow the co-simulation of IEC 61499 control applications and simulation tools. The intention is to reduce the prototyping phase to a bare minimum, eliminating the need to port software from one language to another.

Using the co-simulation tools, a set of open source control applications that can readily be run on a large variety of low cost PLC hardware are developed and validated. The applications are focused on grid-connected PV energy system configurations as illustrated in figure 1.1. Each system includes a PV generator, an electrical load, an optional battery and an optional heat pump. The controllers act upon the PV inverter, the battery and the heat pump. Additionally, they take measurements from each of the components. Other household appliances that could potentially be used for DSM, such as washing machines or dish washers, are not regarded within the scope of this thesis. However, a modular design is intended for the control applications, so that new components can be added at a later time.
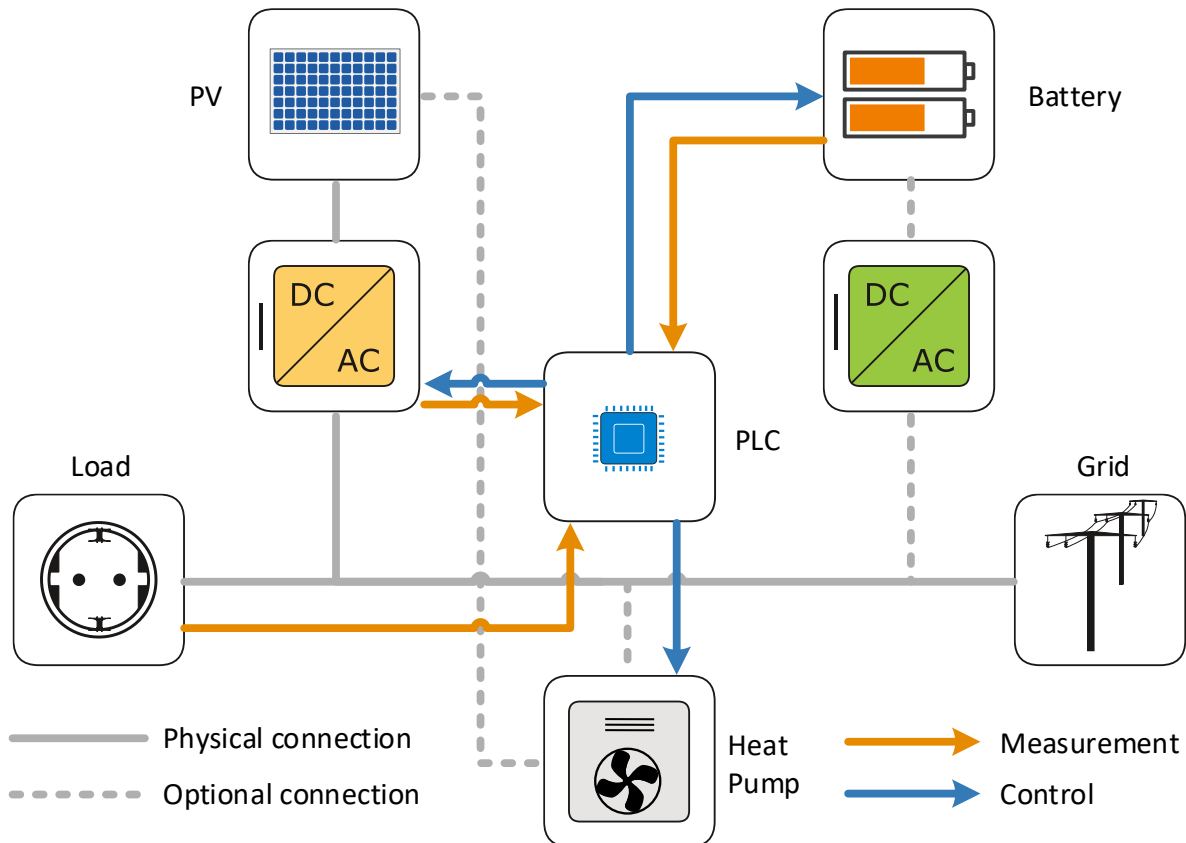
Figure 1.1: *Overview of the system configurations and integration of the controller.*

The objectives can be summarized as follows:

- Facilitation of the development process of control applications by reducing the prototyping phase.

- Elimination of the need to deploy prototypes to different programming languages.

- Design and validation of generic intelligent control applications based on standards for systems as depicted in figure 1.1.

- Establishment of an open source community in the field of energy management.

1.3. Approach

First, the criteria for the control applications are defined and accordingly, appropriate technologies are selected. Benefits and disadvantages of the two PLC standards IEC 61131-3 and IEC 61499 are weighted and according to the comparison, a choice in software and hardware is made. The subsequent section briefly introduces the younger IEC 61499 standard, which defines event-based PLC programming using so-called function blocks (FBs). It aims to provide the reader with the necessary information to understand the IEC 61499 function block library that was created for the development of

intelligent PV system controllers. After providing a detailed description of the underlying algorithms of the forecast based PVprog operation strategy for PV battery systems, the PV system function block libraries are documented.

Prior to the development of the first PVprog and curtailment applications, an open source Matlab® library that enables the `co-simulation` between Matlab® and IEC 61499 applications was devised. The library's use and functionality are documented in detail and it is used for the validation of the first set of control applications. For the creation of more complex applications that also control SG Ready heat pumps, another open source communication library is developed in JAVA™ and used within a plugin that enables the co-simulation of IEC 61499 applications with Polysun®. Various control applications are designed and validated based on Polysun® co-simulations. After augmenting the PLC runtime environment (RTE) "4diac-RTE" (FORTE) with additional communication abilities required for use in many building energy systems, one of the applications is deployed to hardware and tested in the field. Finally, the results of this thesis are summarized, discussed and concluded.

2. Technology selection

The criteria for the control application are discussed in the following subsections. Various technologies are compared and those that best fit the set criteria are selected for use in this project.

2.1. Criteria

At the time of beginning this thesis, it was unclear whether deployment and field testing of a control application would be possible within the time constraints or not. Therefore, one of the main criteria is flexibility. The control applications need to be designed in such a way that their functionality can be validated using the tools at hand. Even with the possibility of field tests, one of the goals is to speed up the development process by reducing the prototyping phase. The control applications designed with simulation tools should be deployable to real systems with as few changes made to them as possible. As mentioned in section 1.2, the software should be generic and based on standards. This ensures a high portability and an easy understandability for potential users and developers. Finally, the project is intended to be fully open source. As such, the final products' usability should not be bound to any proprietary software or hardware limitations.

2.2. Model View Controller

An approach commonly used for high flexibility in object oriented programming (OOP) is the MVC design pattern [2]. It was devised to decouple the model (i.e. the application logic), the view (e.g., a graphical user interface (GUI)) and the controller (that performs operations on the model). This results in a high code re-usability. A visualization of the MVC design pattern is shown in figure 2.1. The user interacts with the view, which in turn delegates the user's requests to the controller's interface. The controller updates the view with the actions being performed and manipulates the model. When the model's state changes, the view is notified and requests the current state from the model. Each element can be exchanged for other components with equivalent interfaces. For example, a GUI can be replaced by a command-line user interface (CLI). The MVC design pattern can be regarded as a good solution for the uncertainty regarding the possibility of field tests, which would be necessary owing to the complexity of some of the control algorithms. In this particular case, the model can either be a simulated energy system (e.g., by Polysun®) or a real system. The view can be a human machine interface (HMI), a GUI or a CLI. Finally, the controller is one of the control applications developed within the scope of this thesis.
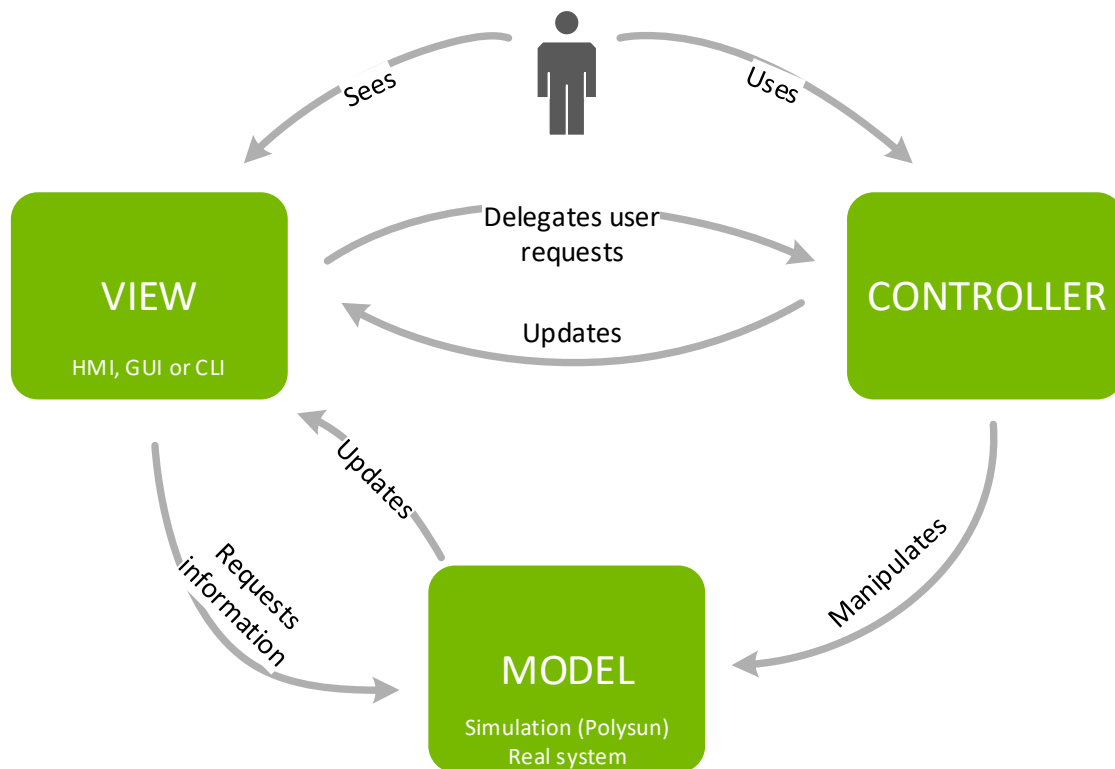
Figure 2.1: *Visualization of the MVC design pattern.*

2.3. PLC standards

There currently exist two standards for PLCs: IEC IEC 61131-3 [3] (originally published in 1993) and IEC 61499 [4] (published in 2005 and revised in 2012), whereby IEC 61131-3 has been widely adopted in the automation industry [5]. IEC 61131-3 defines the following five procedural programming languages:

- Function block diagram (FBD)

- Instruction list (IL)

- Ladder diagram (LD)

- Sequential function chart (SFC)

- Structured text (ST)

FBD, LD and SFC are graphical, while ST and IL are textual programming languages. PLCs can be programmed using combinations of these languages in software environments such as CODESYS. Due to the fact that the standard has been established in the industry, it brings with it the advantage of many years of experience. Thus, the community is large and there is a wide range of software available. In the long run, however, the procedural approach cannot live up to the rising requirements for PLCs [6].

Furthermore, IEC 61131-3 does not make any specifications about cross-compatibility between different platforms. As a result, a control application written for one piece of hardware may have to be rewritten from scratch for another controller. To tackle these drawbacks, IEC 61499 was devised. This standard defines FBs similar to those in FBD, whose interfaces and behaviour are each programmed using a respective graphical syntax, a textual syntax and/or Extensible Markup Language (XML). Rather than being procedural, the graphical applications are object- and event-oriented. The FBs are not activated in a sequential order, but rather by events, which trigger their algorithms. This results in a higher flexibility and in the possibility of more complex, distributed control systems [7]. Moreover, an MVC implementation is possible owing to the event-oriented nature of the standard. Thanks to the "IEC 61499 Compliance Profile for Feasibility Demonstrations" [8], applications developed in compliance with IEC 61499 can be easily transferred between any IEC 61499 compliant software tools and can thus be used to configure any IEC 61499 compliant hardware. Additionally, IEC 61499 compliant devices are interoperable and can communicate with each other. A disadvantage of the standard is that it has yet to be fully adopted by the industry [6]. As a consequence, the community is comparatively small and the range of available software/hardware is still limited compared to IEC 61131-3. There is less experience, which may result in a steeper learning curve.

The advantages and disadvantages of the two PLC standards are summarized in table 2.1. A "+" represents an advantage, a "-" symbolizes a drawback and an "o" stands for "neutral". The benefits of IEC 61499 far outweigh the drawbacks. Besides, it is envisaged that IEC 61131-3 will be revised in the future and made compatible with IEC 61499 [7]. This fact further substantiates the decision to choose IEC 61499 as the standard on which to base the control applications developed within the scope of this thesis.

Table 2.1: *Advantages and disadvantages of the PLC standards.*

IEC 61131-3		IEC 61499	
++	Experience	-	Not yet adopted by the industry
+	Large community	-	Small community
+	Wide range of software/hardware	o	Growing range of software/hardware
- -	Low flexibility	++	High flexibility
- -	Platform dependent	++	Platform independent
- -	Little to no interoperability	++	High interoperability
- -	Little to no portability	++	High portability
-	Designed for centralized systems	+	Distributed systems possible
- -	Limited to simple control systems	++	Complex control systems possible
-	Low reliability	+	High reliability

2.4. Software selection

There is a variety of software available for the design and deployment of IEC 61499 applications. The tools can be categorized as (i) open source tools, (ii) commercial (proprietary) tools and (iii) academic/research developments [9]. The latter may be either open or closed source. Current commercial tools consist of ISaGRAF and NxtONE, the open-source tools include 4diac, FBench, ICARU_FB and GASR-FBE and the academic/research developments are comprised of the tools, FBDK/FBRT, BLockIDE and Corfu ESS / Archimedes. The benefits and drawbacks regarding technical support, costs, usability and transparency for the three software categories are listed in table 2.2. Proprietary solutions come with the advantages of professional technical support and releases are often-times more stable than their open-source counterparts. However, these advantages are not free, while the open-source and academic variants can be downloaded and used at no cost. Research developments are often poorly documented, resulting in difficulty learning how to use them. With respect to support, the open-source software lies between its proprietary and academic counterparts. It is usually well-documented, but one must rely on user-forums for technical assistance. Nevertheless, due to the small community in the particular case of IEC 61499, personal experience has shown that topics posted in the 4diac user forum are answered quickly by the developers in person. Therefore, the open-source tool 4diac was chosen for this thesis. 4diac provides an integrated development environement (IDE) for the development of function blocks and applications and a runtime environment called FORTE that can be run on a large variety of Windows and Linux based hardware.

2.5. Available hardware

Owing to the high portability of IEC 61499 applications and the MVC design approach, a hardware selection is of minor significance within the scope of this thesis. For the sake of completeness, a list of exemplary hardware that can be configured with the devised control application is listed in table 2.3. Vendors that offer IEC 61499 compliant hardware that can be configured by porting the applications to other IDEs include Beckhoff, Siemens, Advantech, Mitsubishi, Bosch, Loytech and Schneider Electric, among others.

Table 2.2: *Advantages and disadvantages of the software categories.*

	Proprietary	Open-source	Academic/research
Technical support	+	-	- -
Costs	- -	++	++
Learning curve / usability	+	-	- -
Transparency	-	+	+

Table 2.3: *A selection of hardware that can be configured with control applications developed in 4diac.*

Hardware	Deployment	Limitations
Windows/Linux PC	Direct	None
Lego Mindstorms (EV3)	Direct	None
RaspberryPi (SPS)	Direct	None
Odroid	Direct	None
Wago PFCs SPS	Direct	None
ICnova	Direct	None
µMic 200	Direct	None
IndraControl XM22 PLC	Direct	None
nxtDSCmini	Port to nxtONE STUDIO	Re-configuration necessary in some cases
Any JAVA™ based hardware	Port to FBRT	Re-configuration necessary in some cases

It must be noted, however, that so-called service interface function blocks, whose algorithms are programmed in C++[i], cannot be transferred directly to other software tools. If such function blocks are used, they must be ported over and the affected applications must be re-configured [10].

---

[i]In the case of 4diac. SIFBs in other IDEs may be programmed in other languages.

3. The IEC 61499 PLC standard

The following section provides a brief exposition into PLC programming using IEC 61499. It presents an overview of the standard's aspects that are relevant to this thesis. Although most of the aspects are standardized, some implementations may vary slightly between different vendors' software solutions. Because the open-source program 4diac is used for development in this thesis, the following subsections lay their focus on the 4diac implementation.

3.1. Function blocks

Function blocks (FBs) are the key elements of the IEC 61499 architecture. They wrap the underlying algorithms with interfaces that can be combined to develop the applications. IEC 61499 defines three different types of FBs: Basic function blocks (BFBs), composite function blocks (CFBs) and service interface function blocks (SIFBs). All three types share the same interface. The standard furthermore defines a graphical syntax, a textual syntax and an XML syntax for FBs (and FB applications) [4]. The definitions of the textual and XML syntaxes, respectively, exceed the scope of this thesis and are omitted from the following subsections. How to compile the syntax is left up to the software vendors. For example, 4diac-IDE compiles FBs to C++ classes, which are used by FORTE.

3.1.1. Function block interface

An example for a FB interface is depicted in figure 3.1. Inputs are on the left side and outputs on the right. Event inputs and outputs are displayed on the top half of the FB and data inputs and outputs on the bottom half. The IEC 61499 data type of an input or output is depicted on the outside of the FB. Every input or output comes with an optional description. Events and data are linked together with so-called `WITH` qualifiers (depicted graphically as square connectors between the event and data inputs

Event Input 1	-	Event	EI1	EO1	Event	-	Event Output 1
Event Input 2	-	Event	EI2	EO2	Event	-	Event Output 2
			FB_BFB_EX 1.0				
Data Input 1	-	BOOL	QI1	QO1	BOOL	-	Data Output 1
Data Output 2	-	ANY	QI2	QO2	LREAL	-	Data Output 2
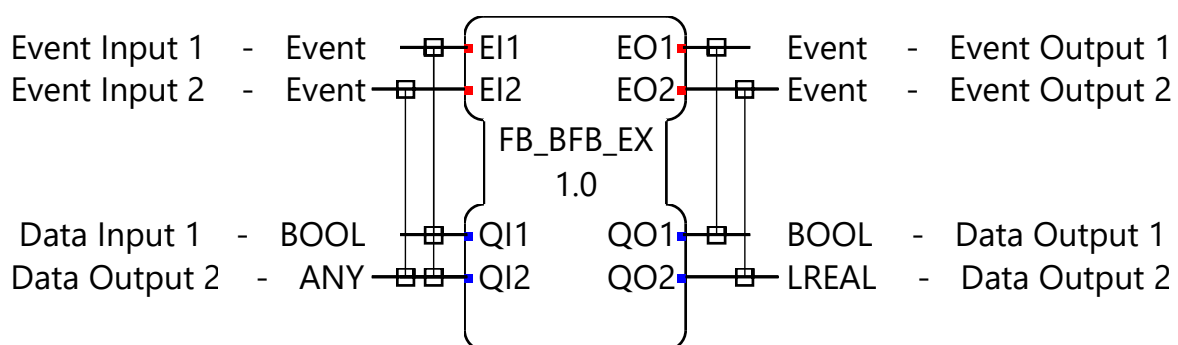
Figure 3.1: *Example of a FB interface as displayed in 4diac-IDE.*

or outputs). This indicates that the linked data must be sampled upon every occurrence of the respective event. Each data input or output can be linked `WITH` one or multiple event inputs or outputs, respectively. In figure 3.1, the data input `QI2` is initialized every time an `EI1` or `EI2` event arrives. All other data sockets are linked `WITH` only one event socket each. Finally, the FB's type name and version number are depicted in the middle of the graphical representation.

3.1.2. Basic function blocks

As the name implies, BFBs are the most simple form of FBs. The internal behaviour of a BFBs is described with states and algorithms, which are programmed graphically using an excecution control chart (ECC), as depicted in figure 3.2. If a certain condition is met (i.e. the arrival of an input event), the BFB switches its state (indicated graphically with an arrow connector), and if that state is linked to an action, it runs an algorithm and/or triggers an output event upon completion. Algorithms, in which the data are manipulated, can be programmed in ST, JAVA™ or C++ [7], whereby ST appears to be the most widely supported language. Apart from the input and output data, a BFB can also hold internal variables, which are equivalent to private properties in OOP classes. Unlike the other states, the initial state the function block starts in is framed with a double outline.

In the example depicted in figure 3.2, the BFB switches to the `Init` state upon arrival of an `EI1` input event that is linked to the `QI1` data input - if `QI1` is set to `true`. The initialization algorithm is run, and upon completion, the output event `EO1` is triggered to notify other function blocks. Because the condition for switching to the `Initialized` state is set to 1 (equivalent to a boolean `true`), it switches immediately after issuing the event. If an `EI2` input event occurs, the FB switches to the `NormalOp` state and runs the `normalOperation` algorithm, followed by an `EO2` output event, before switching back to the `Initialized` state. Lastly, the input event `EI1` together with `QI1` set to `false` triggers the de-initialization process.
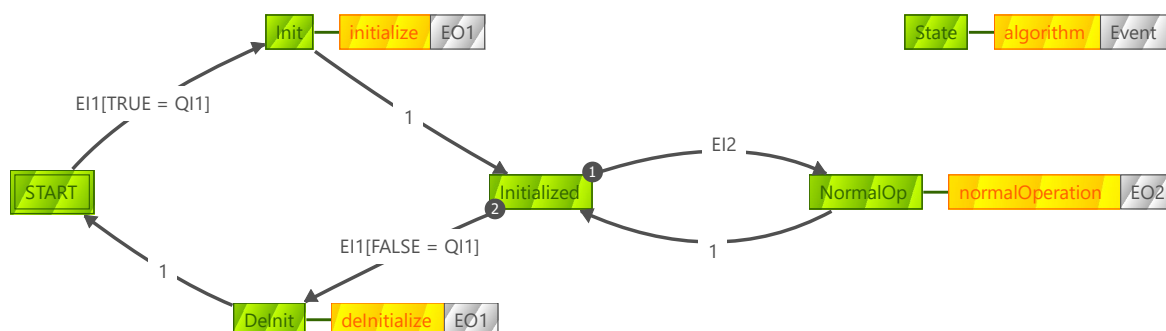
Figure 3.2: *Example of an ECC as displayed in 4diac-IDE.*

3.1.3. Composite function blocks

To create more complex components, multiple FBs can be combined into a CFB. An example of the internal composite network of a CFB is illustrated in figure 3.3[i]. The external interface is exactly the same as that of a BFB. In fact, one cannot tell whether a function block is a BFB or a CFB just by looking at the interface. A CFB can be composed of BFBs, lower level CFBs and SIFBs [7]. The inputs of the external interface are routed to various inputs of the internal FBs (depicted graphically with red arrows for events and blue arrows for data in 4diac-IDE). Per definition, each CFB data or event input must be routed either to one or many internal FBs' inputs or "through-routed" to one or many of the CFB's outputs [7]. Respectively, each output must either be connected to one of the CFB's inputs or an internal FB's input. An internal FB's output does not have to be connected to anything.

3.1.4. Service interface function blocks

The links between external devices and function blocks on a control resource[ii] are established using SIFBs. As suggested by their name, they provide the application with an interface to a service. This could be to an Input/Output (I/O) device, such as an HMI or a CSV (comma separated values) parser or to an external device that is being measured or controlled on a network. Unlike BFBs and CFBs, SIFBs can usually be recognized via the interface, due to the fact that the inputs, outputs and their `WITH` qualifiers are standardized [7]. However, an SIFB is not forced to implement all of the inputs and outputs, and can also add additional ones. A complete overview of the events, data and the `WITH` qualifiers is provided in figure 3.4.

---

[i]The examples depicted in this section provide no relevant functionality with respect to this thesis, and were chosen solely for the purpose of demonstration.

[ii]A resource is what executes function block networks. A device may have multiple resources, e.g., a PC (device) can run FORTE and FBRT (resources) at the same time.
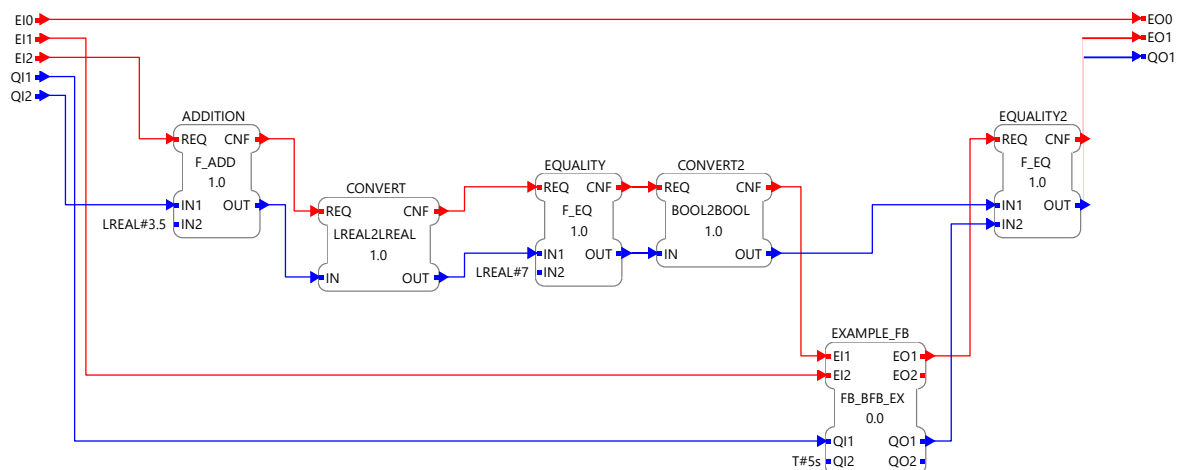
Figure 3.3: *Exemplary composite network of a CFB as displayed in 4diac-IDE.*
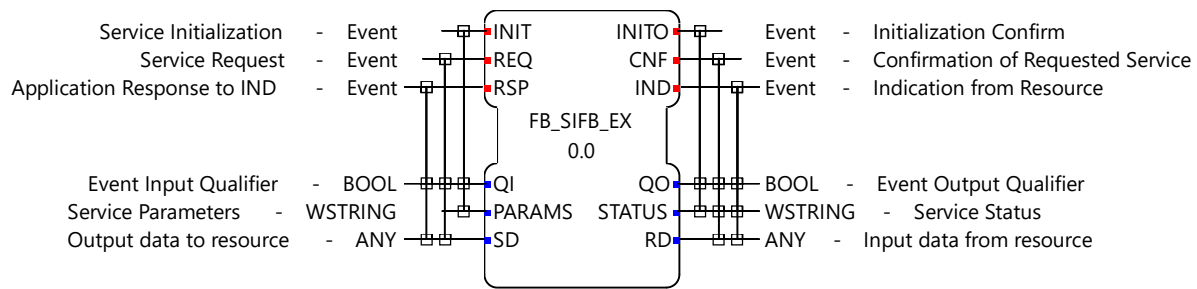
Figure 3.4: *Inputs and outputs of a SIFB as displayed in 4diac-IDE.*

The boolean data input `QI` and its equivalent output `QO` are linked to every input and output, respectively. They act as so-called event qualifiers, indicating which of two actions to perform upon arrival of an event in the case of `QI`; and success or failure of a request or service in the case of `QO`.

An overview of the input and output events depending on their qualifiers is listed in table 3.1. The data input `PARAMS` holds the service parameters that are used to initialize the service, and the output `STATUS` provides a message that accompanies `QO`. It can be used to determine the reason a service or initialization failed, for instance. Lastly, every SIFB can have an arbitrary amount of data inputs `SD` and outputs `RD`. For example, in the case of `RD`, they could be the values read from a CSV file or received from a communication service. In the case of `SD`, they could be the the data that are to be sent to a communication service or written to a CSV file.

Unlike the internals of BFBs and CFBs, those of an SIFB are not defined in IEC 61499. They are implemented in the language of the RTE on which they run (C++ in the case of FORTE and JAVA™ in the case of FBRT, for example). This means that an SIFB created for FORTE must be re-designed for FBRT, and vice versa. For this reason, HMI SIFBs that run on FBRT do not run natively on FORTE [11]. To design an SIFB for FORTE, the interface can be graphically outlined in 4diac-IDE and exported to C++

Table 3.1: *Overview of the event inputs and outputs of a SIFB, depending on the qualifier. `Q` represents the data input `QI` and the data output `QO`, respectively.*

Event	Q = true	Q = false
INIT	Service initialization request	Service deinitialization request
REQ	Request for service	Disable request for service
RSP	Response to service request (success)	Response to service request (failure)
INITO	Confirmation of successful initialization	Confirmation of failed initialization
CNF	Confirmation of successful service request	Confirmation of failed service request
IND	Indication of service arrival (success)	Indication of service arrival (failure)

(a) *Application* **(b)** *Subapplication*
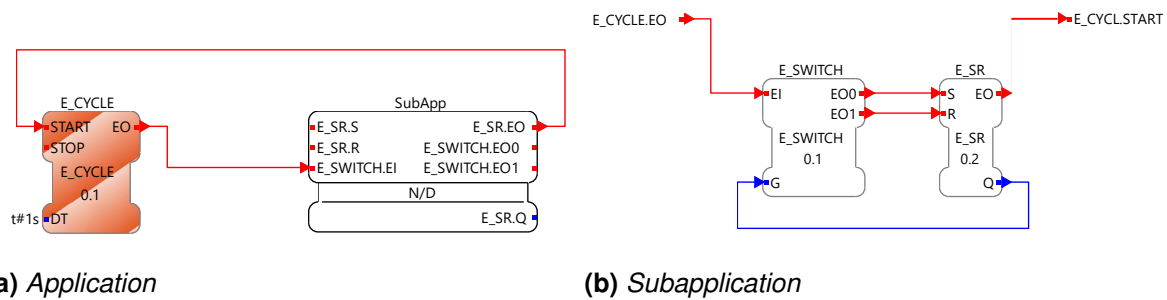
Figure 3.5: *Exemplary function block application (left) and subapplication (right).*

header and source files. The respective algorithms can then be programmed in C++.

3.2. Applications and subapplications

To create functioning PLC control systems using IEC 61499, function blocks must be connected to networks in applications, which can be distributed across devices and resources. Figure 3.5a shows an exemplary application of a set/reset flip-flop connected to a clock. The E_SWITCH and E_SR FBs are combined in a subapplication (figure 3.5b). Subapplications are very similar to CFBs in that they are composed of multiple FBs. The main differences are that they can be run on multiple resources and that their data and events cannot be linked using a WITH qualifier. This is due to the fact that the input and output data are not stored at the subapplication's inputs and outputs, respectively [7].

3.3. Adapters

In some cases, function blocks interact with each other, accessing each others inputs and outputs, as shown in figure 3.6a. This can quickly lead to cluttering of the workspace if many FBs interact with each other in this way. To reduce clutter, IEC 61499 provides a so-called "adapter concept", which reduces such interactions to a reusable interface element on each FB and only one connection between them [7]. The exemplary connection illustrated in figure 3.6a is depicted in figure 3.6b using the adapter concept. The inputs are reduced to a plug on the output side of the first FB that is connected to a socket on the input side of the second FB. In 4diac-IDE, the socket interface must be designed by hand, and the corresponding plug, which simply mirrors the inputs and outputs of the socket, is generated automatically [11]. The socket interface, Adapter_ex, is marked green in figure 3.6b.

(a) *Without adapter*   **(b)** *With adapter*
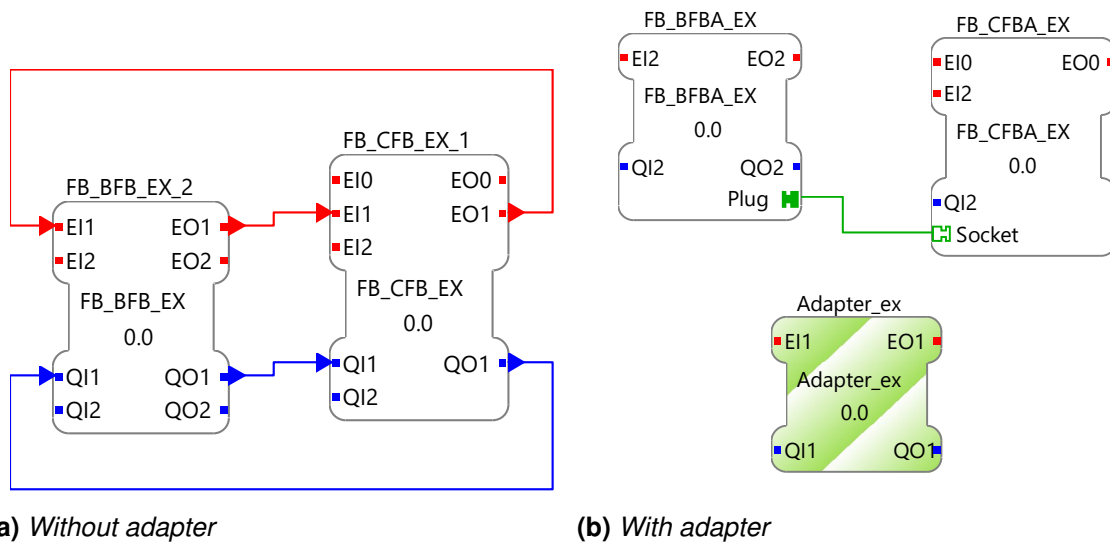
Figure 3.6: *Exemplary FB connection without an adapter (left) and using an adapter (right).*

3.4. Communication protocols

In order for devices to communicate with each other and with external services, communication service interface function blocks (CSIFBs) are used. They are defined in the IEC 61499 Compliance Profile for Feasibility Demonstrations [8]. CSIFBs are SIFBs that provide the interface for various communication protocols. It is exactly the same as that of a regular SIFB, however, the number of inputs `SD1..SDn` on the sending end must match the amount of outputs `RD1..RDn` on the receiving end, and vice versa [11]. Following is a brief description of the basic protocols and the corresponding CSIFB interfaces.

3.4.1. User Datagram/Internet Protocol

The User Datagram Protocol (UDP) implements the Observer design pattern. One or multiple subscribers (recipients) listen in on a publisher, connecting via an Internet protocol (IP) address and a port, whereby the IP address identifies the location of the device, and the port specifies which subscribers are connected to which publishers. One publisher can be linked with many subscribers, but each subscriber can only be connected to one publisher. The subscribers "listen in" on the publisher, which notifies them whenever data were sent. In the case of IEC 61499 `SUBSCRIBER` CSIFBs, this triggers an `IND` output event (see section 3.1.4). Similarly, a `PUBLISHER`'s `REQ` input can be used to send data to subscribers on another device. An example of an IEC 61499 device communicating with two other IEC 61499 devices is presented in figure 3.7. The function blocks are colour coded according to the devices they are mapped to. The `EO1` output event of the `FB_BFB_EX` function block on device 1 (green) triggers the `REQ` event of the `PUBLISH` FB, which sends the data to the `SUBSCRIBE_A` and `SUBSCRIBE_B` FBs
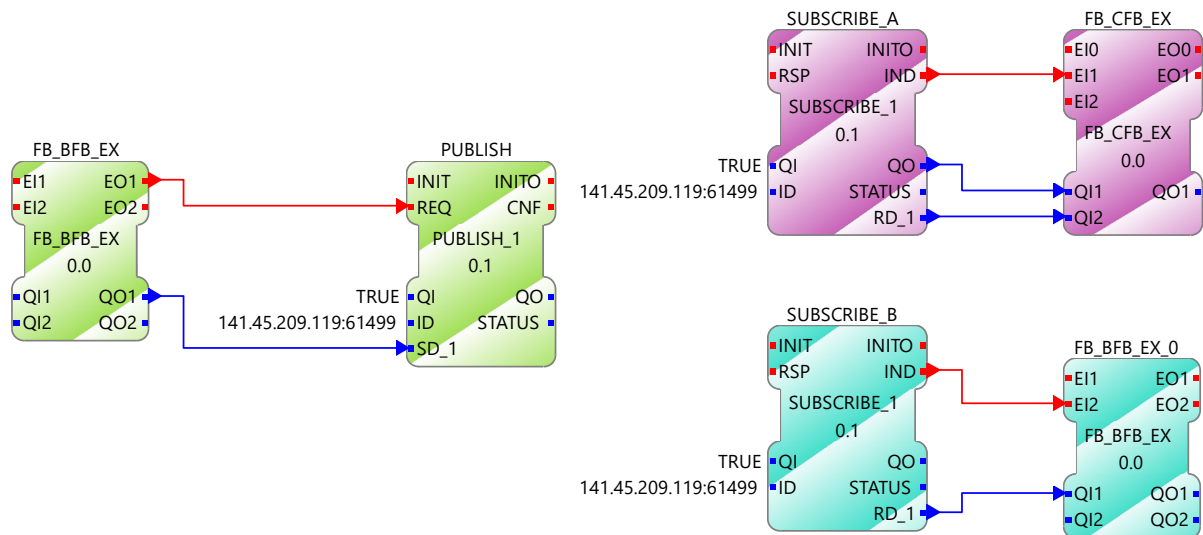
Figure 3.7: *Example of UDP/IP communication between three IEC 61499 devices using* `SUBSCRIBE`/`PUBLISH` *function blocks.*

on devices 2 (purple) and 3 (teal). Each of the subscribers on the receiving end passes an `IND` event along with the received data to its respective function block network. The advantages of UDP/IP are its simplicity and speed, however, it comes with the caveat of low reliability and the possibility of packet losses due to the lack of acknowledgement features [6]. Another drawback is that requesting data from an external device requires multiple sets of publish/subscribe elements, and can quickly become messy.

3.4.2. Transmission Control/Internet Protocol

For higher reliability, the Transmission Control Protocol (TCP), in which every request must be responded to, is used. Two devices or resources communicate using one or multiple client and server pairs, which are linked via the IP address of the device hosting the server and a port number. Each client can be linked to only one server, and vice versa. For a successful connection, the server must be initialized first. An example of the communication between two IEC 61499 devices is depicted in figure 3.8. In this example, the client on device 1 (purple) sends a request to the server every second. The server on device 2 (teal) receives the integer 3 from the client, and passes it to the `FB_BFB_EX` function block, which performs an action on the data. The `FB_BFB_EX` FB then passes its outputs back to the server, which sends a response event, along with its linked data, to the client on device 1.

3.4.3. Other communication protocols

Many more advanced communication protocols are based on either UDP, TCP or combinations thereof. For example, the widely utilized Modbus protocol can be implemented by using `CLIENT` and `SERVER` function blocks or alternatively by using a `CLIENT` FB
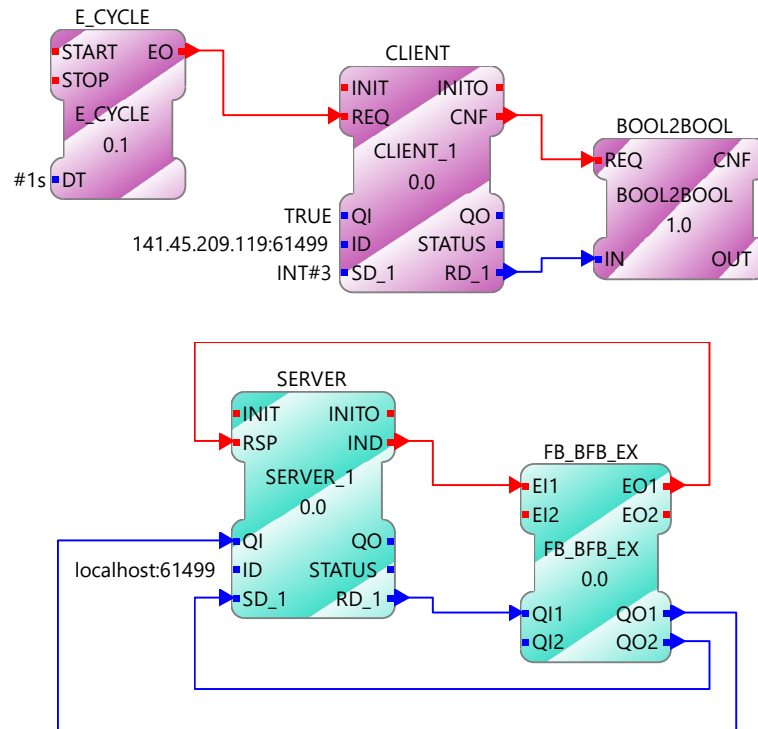
Figure 3.8: *Example of TCP/IP communication between two IEC 61499 devices using* `CLIENT/SERVER` *function blocks.*

combined with `PUBLISH`/`SUBSCRIBE` FBs [6]. State-of-the-art communication protocols supported by 4diac at the time of writing this thesis include OPC Data Access (OPC DA), OPC Unified Architecture (OPC UA), Modbus TCP, Modbus remote terminal unit (RTU), openPOWERLINK, and Message Queue Telemetry Transport (MQTT). A detailed description of the supported protocols exceeds the scope of this thesis. For the use of one of the protocols with FORTE, the appropriate libraries must be downloaded from external sources. The function blocks can then be configured to use the protocol via the ID data inputs.

4. PVprog - Forecast based charging of PV battery systems

Following is a description of the PVprog algorithm[i] [1], a forecast-based charging strategy for PV battery systems developed at HTW Berlin that has been implemented in the form of an IEC 61499 function block library as part of this thesis. Its foundation are location-specific forecasts of the PV generation and the load. The first subsections compare the operational strategy to that of charging the battery as soon as PV surpluses occur, revealing the benefits of forecast-based battery charging. Thereafter, the algorithm is explained mathematically and through graphical visualizations.

4.1. Early battery charging with a feed-in limitation through curtailment

The main objective of this strategy is to utilize the battery primarily for covering the load. The battery is charged with PV surpluses that exceed the load as soon as possible (see figure 4.1, left). If the battery reaches its maximum capacity, the grid feed-in power can increase abruptly, especially on a sunny day. All PV surpluses that exceed the feed-in limitation are curtailed.

---

[i] The original Matlab® implementation is available for download as open source at `https://pvspeicher.htw-berlin.de/veroeffentlichungen/daten/pvprog/`
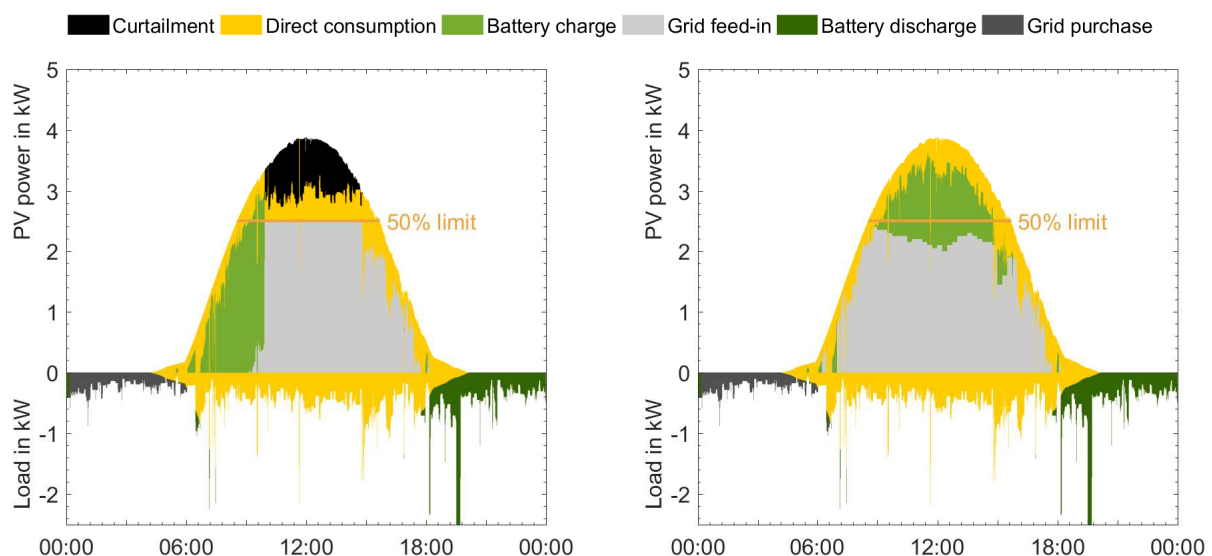
Figure 4.1: *Power flows in a household in which the battery is charged as soon as possible (left) vs. power flows in a household with forecast-based battery charging (right). The feed-in power is limited to* 50 *%. Installed PV power:* 5 kWp, *usable battery capacity:* 5 kWh.

4.2. Forecast-based battery operation with a dynamic feed-in limitation

This strategy ensures that the battery is only charged with PV surpluses that exceed a virtual feed-in limit which is determined by simulating battery charging over a forecast horizon taking PV generation and load forecasts into account. The power flows of a household with forecast-based battery charging are depicted in figure 4.1 (right) for the same day as in figure 4.1 (left). The temporal shift of the battery charging from morning to noon allows a reduction of the curtailment losses while maintaining a high self-consumption rate of the system. Additionally, the battery spends less time in a high state of charge SoC, and therefore, in the case of a lithium-ion battery, has an extended calendar life in comparison with one that is charged as soon as possible [12].

A substantial benefit of forecast-based operational strategies lies within the grid relief [13], [14]. The simulated cumulative power flows of PV battery systems that are distributed across Germany are depicted in figure 4.2. Early charging (figure 4.2, left) results in the phenomenon that most of the batteries have reached maximum capacity by noon. As a result, the peak grid feed-in power is barely reduced. Compared to systems without batteries, the gradient of the feed-in power is in fact increased [14]. Due to the forecast-based dynamic feed-in limitation, battery charging is mostly moved to noontime, resulting in far lower ramp rates [13], [14].

4.3. PV power forecasts

To achieve sufficiently accurate forecasts of the PV generation without having to resort to a costly external communication infrastructure, the PVprog algorithm implements an
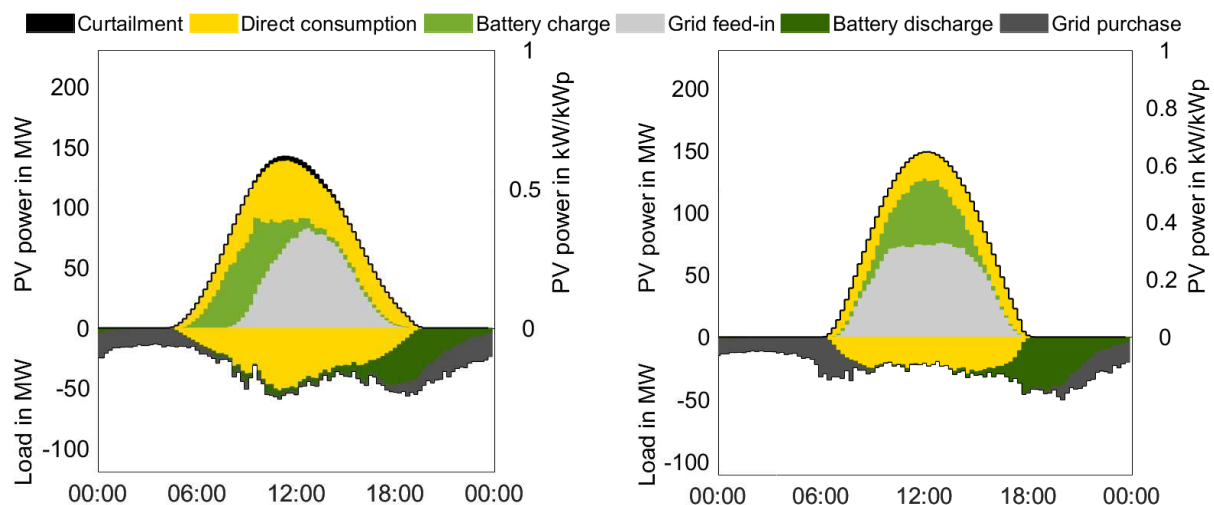
Figure 4.2: *Cumulative power flows of households in which the battery is charged as soon as possible (left) vs. those of households with forecast-based battery charging (right). Simulation of $46{,}126$ households distributed across Germany with a mean installed PV capacity of 5 kWp and a mean usable battery capacity of 5 kWh.*
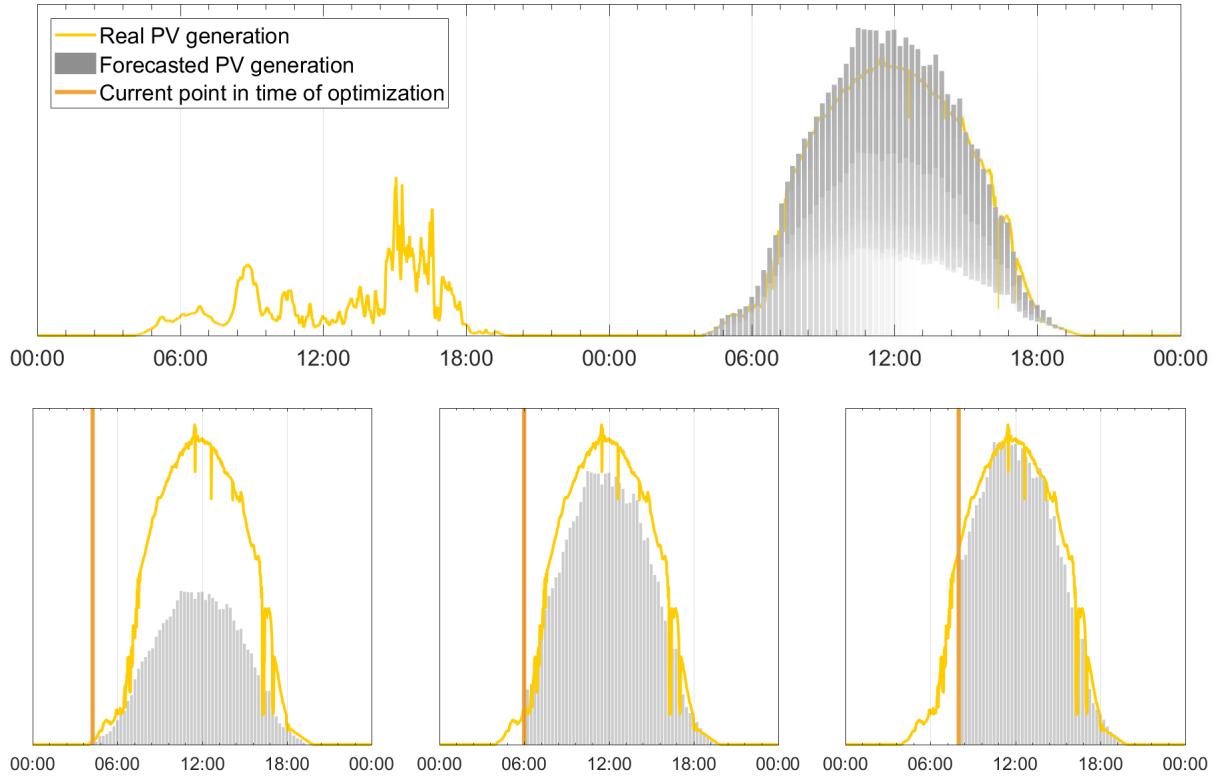
Figure 4.3: *Dynamic adjustment of the PV forecasts during the course of the day.*

adaptive day persistence. Every 15 min, the forecasts are re-calculated with updated measurements. The dynamic adjustment of the PV forecasts during the course of the day is visualized in figure 4.3 using an exemplary sunny day that follows after a very cloudy day. On top, the real PV generation and the forecasts are superimposed with different shades of grey. The bottom subplots show the forecasts at three different points in time. In the morning, a low PV generation is predicted, however, by 6 AM, the algorithm recognizes that the predicted power was under-estimated. The forecast is adjusted to an increased power, and after only a few hours it is almost equal to the real PV generation.

To generate forecasts, the historical PV power P_{pv} and the maximum PV power of the last 10 days $P_{pv,max}$, which is an approximation of the clear-sky generation $P_{pv,cs}$, are required. $P_{pv,max}$ at a given time of day is the maximum of the PV power at the exact same time of day within the last 10 days. After removal of the night time values, the energy sums of P_{pv} and $P_{pv,max}$, respectively, are determined from a persistence time frame. Starting at the point in time t_0, the persistence time frame is the interval $[t_0, t - 1 \text{ min}]$. The fraction of both energy sums is the forecast clearness index k_{Tf}.

$$k_{Tf}(t) = \frac{\int_{t_0}^{t-1 \text{ min}} P_{pv} \, dt|_{P_{pv}>0}}{\int_{t_0}^{t-1 \text{ min}} P_{pv,max} \, dt|_{P_{pv,max}>0}} \tag{4.1}$$

At every optimization time t_{opt} (every 15 min), the current 15 min mean $\overline{k}_{Tf,15}$ of k_{Tf} is

used as a scaling factor for the 15 min mean of the clear sky approximation $P_{\mathsf{pv,max15}}$ for the current day. This results in the PV forecast $P_{\mathsf{pv,f}}$ [15].

$$\overline{k}_{\mathsf{Tf,15}}(t_{\mathsf{opt}}) = \frac{\int_{t_{\mathsf{opt}}-15\ \mathsf{min}}^{t_{\mathsf{opt}}} k_{\mathsf{Tf}}(t)\ dt}{15\ \mathsf{min}} \tag{4.2}$$

$$\overline{P}_{\mathsf{pv,max15}}(t) = \frac{\int_{t-15\ \mathsf{min}}^{t} P_{\mathsf{pv,max}}(t)\ dt}{15\ \mathsf{min}} \tag{4.3}$$

$$P_{\mathsf{pv,f}}(t) = \overline{k}_{\mathsf{Tf,15}}(t_{\mathsf{opt}}) \cdot \overline{P}_{\mathsf{pv,max15}}(t) \tag{4.4}$$

4.4. Load forecasts

The load forecasts are updated dynamically in the same intervals as the PV power forecasts. A day persistence $P_{\mathsf{load,fP}}(t)$ is taken from the interval $[t_{\mathsf{opt}} - 24\ \mathsf{h},\ t_{\mathsf{opt}} - 9\ \mathsf{h}]$, a time frame of $15\ \mathsf{h}$, which is equivalent to the forecast horizon on the day before. It is weighted with the mean load $\overline{P}_{\mathsf{load,15}}$ of the last 15 min before t_{opt} using an exponential weighting function (see figure 4.4, top) [15].

$$\overline{P}_{\mathsf{load,15}}(t_{\mathsf{opt}}) = \frac{\int_{t_{\mathsf{opt}}-15\ \mathsf{min}}^{t_{\mathsf{opt}}} P_{\mathsf{load}}(t)\ dt}{15\ \mathsf{min}} \tag{4.5}$$
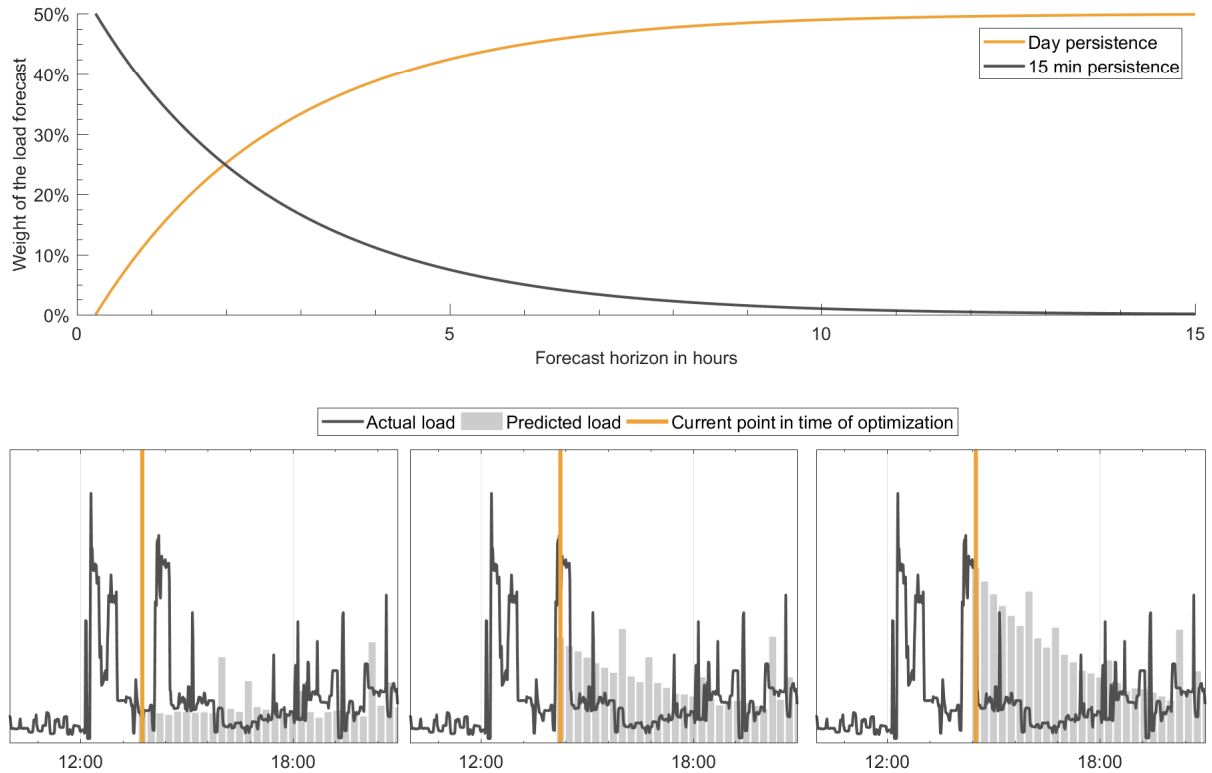
Figure 4.4: *Weighting functions of the load forecast components over the forecast horizon (top) and dynamic adjustment of the load forecast (bottom).*

The function $w_{\mathsf{lfP}}(t)$ for the time-variable weighting of $P_{\mathsf{load,fP}}$ over the forecast horizon $[t_{\mathsf{opt}} + 1\text{ min}, t_{\mathsf{opt}} + 15\text{ h}]$ is given by equation 4.6.

$$w_{\mathsf{lfP}}(t) = \exp(0{,}1) \cdot \exp\left(-0{,}1 \cdot \frac{t - t_{\mathsf{opt}} + 1\text{ min}}{1\text{ min}} \right) \tag{4.6}$$

Using the corresponding function $w_{\mathsf{lc}}(t) = 1 - w_{\mathsf{lfP}}(t)$ for the weighting of $\overline{P}_{\mathsf{load,15}}(t_{\mathsf{opt}})$, the load forecast $P_{\mathsf{load,f}}$ is calculated as

$$P_{\mathsf{load,f}}(t) = w_{\mathsf{lfP}}(t) \cdot P_{\mathsf{lfP}}(t) + w_{\mathsf{lc}}(t) \cdot \overline{P}_{\mathsf{load,15}}(t_{\mathsf{opt}}) \tag{4.7}$$

Figure 4.4 (bottom) visualizes the dynamic adjustment of the load for part of a selected day. At times of low consumption (leftmost), the short-term forecast predicts an ongoing low consumption. When load peaks occur (middle, rightmost), the algorithm increases the load prediction of the short-term future. Due to the course of the weighting curve $w_{\mathsf{lc}}(t)$, the predicted load gradually decreases over the forecast horizon.

4.5. Battery charge and discharge optimization

The battery charge and discharge optimizations are performed by iterating through various virtual feed-in limits between 0 and the set feed-in limit with a subsequent battery pre-simulation. With respect to the feed-in limitation, an optimum is found when the battery simulation indicates that further increasing the dynamic feed-in limit would result in a lower SoC at the end of the forecast horizon. In the case of load limitation, which utilizes the same approach "in reverse", an optimum is found when the simulation suggests that further incrementing the load limit would result in a higher SoC at the end of the forecast horizon. Intermediate discharging and charging are neglected for the charge and discharge simulations, respectively. The slight negative effect this has on the system's degree of self-sufficiency compared to a linear optimization algorithm is negligible [15].
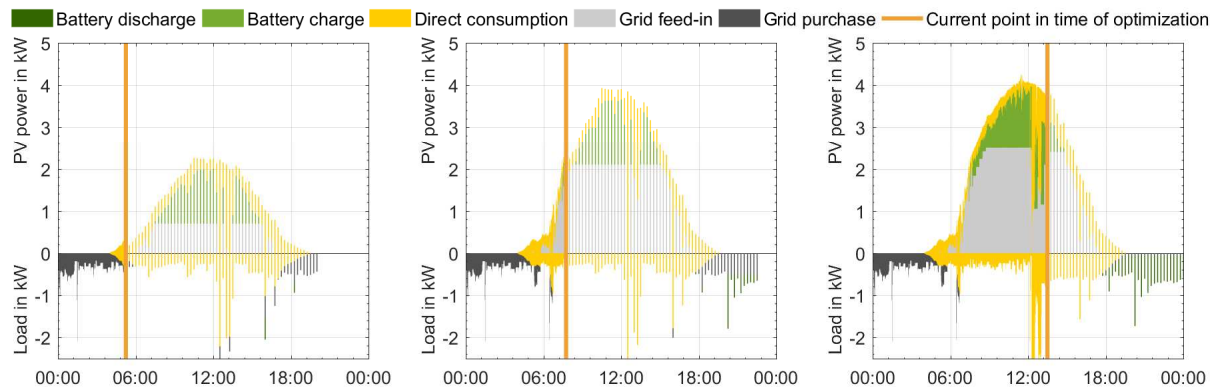
Figure 4.5: *Dynamic adjustment of the battery charge/discharge roadmap.*

The result of the iteration is the dynamic charge/discharge roadmap shown in figure 4.5. $P_{bat,f}$, the power with which the battery must be charged in order to realize the determined optimum, is the optimization algorithm's output. This approach is slightly different than that of the original Matlab® implementation [1], designed with a looser coupling between the battery model and the optimizer in mind (see section 5.1.5). Nevertheless, the end results are almost identical, as the Matlab® validation in section 6.3.1 proves. On a side note, load peak shaving is not implemented in the original algorithm.

4.6. Battery pre-simulation

To be able to estimate the battery's SoC at the end of the forecast horizon, it is necessary to perform a pre-simulation. It is advisable that any vendor who uses the PVprog algorithm for forecast-based charging implement the battery model themselves. A simple model is included in the original PVprog simulation model [1], and a slightly altered version is included as the default in this implementation. The alterations were made to make the model easier to parametrize from data sheets. For example, the original model regards the SoC as the fraction between the battery's current capacity and its *usable* capacity C_{bu}. This model has been adapted to fit the more common interpretation of the SoC as the division of the current capacity by the battery's *nominal* capacity C_{bn}. The model used in this thesis is described as follows:
The amount of energy stored within the battery is determined from the SoC and the nominal capacity,

$$E_{bat} = SoC \cdot C_{bn} \tag{4.8}$$

whereby the minimum and maximum amounts E_{bmin} and E_{bmin} are limited by the minimum and maximum SoC, respectively.

$$E_{bat,max} = SoC_{max} \cdot C_{bn} \tag{4.9}$$

$$E_{bat,min} = SoC_{min} \cdot C_{bn} \tag{4.10}$$

For charging with a given amount of energy E, the battery's and inverter's charging efficiencies $\eta_{bat,c}$ and $\eta_{binv,c}$ are taken into account. The energy contents at the end of the forecast horizon are estimated as

$$E_{bat}(t_{opt} + 15 \text{ h}) = \min(E_{bat,max}, E_{bat}(t_{opt}) + E \cdot \eta_{bat,c} \cdot \eta_{binv,c}) \tag{4.11}$$

Likewise, for discharging, the battery's and inverter's discharging efficiencies $\eta_{bat,d}$ and $\eta_{binv,d}$ are considered.

The remaining energy contents at the end of the forecast horizon are given by

$$E_{\text{bat}}(t_{\text{opt}} + 15\,\text{h}) = \max(E_{\text{bat,min}}, E_{\text{bat}}(t_{\text{opt}}) - E \cdot \eta_{\text{bat,d}} \cdot \eta_{\text{binv,d}}) \tag{4.12}$$

4.7. Error control

To ensure that the system adheres to the set feed-in and/or load limitations, the previously optimized charging/discharging power is continuously adjusted by the difference between the last forecast and the current measurement. Because the battery roadmap is refreshed routinely (every 15 min), updated PV and load forecasts can be taken into account and deviations of the SoC between the corrected and original roadmaps can be compensated. The error control occurs under one of the following conditions:

- The current set value of the battery charge/discharge is not zero.

- The current prediction of the PV surplus or load deficit (absolute value[i]) is greater than the maximum of the predicted feed-in power or grid purchase, respectively, within the forecast horizon.

- The current PV surplus is greater than the set feed-in limit or the current PV deficit is lower than the set load limit.

The corrected set power $P_{\text{bat,c/d}}$ is given by equations 4.13 and 4.14 for charging and discharging, respectively.

$$P_{\text{bat,c}} = \max(0, P_{\text{bat,cf}} + (P_{\text{pv}} - P_{\text{load}}) - (P_{\text{pv,f}} - P_{\text{load,f}})) \tag{4.13}$$

$$P_{\text{bat,d}} = \min(0, P_{\text{bat,df}} + (P_{\text{pv}} - P_{\text{load}}) - (P_{\text{pv,f}} - P_{\text{load,f}})) \tag{4.14}$$

In the original simulation model [1], the error control algorithm limits the battery charge and discharge set value by the maximum and minimum power permitted by the battery inverter. This is not necessary in a practical application due to the fact that battery manufacturers implement this limitation in their batteries for reasons of security. The charge/discharge power restriction step can safely be omitted from the error control algorithm, thus enabling a looser coupling of the battery model and the control algorithm (see section 5.1.6). If, however, a battery were not to feature power limitations, the restriction can be added without much effort.

4.8. Overview

An overview of the PVprog algorithm and the composition of its components with the energy and information flow directions is depicted in figure 4.6.

---

[i]In this context, PV deficits, grid supply and battery discharge are normally represented by negative values.
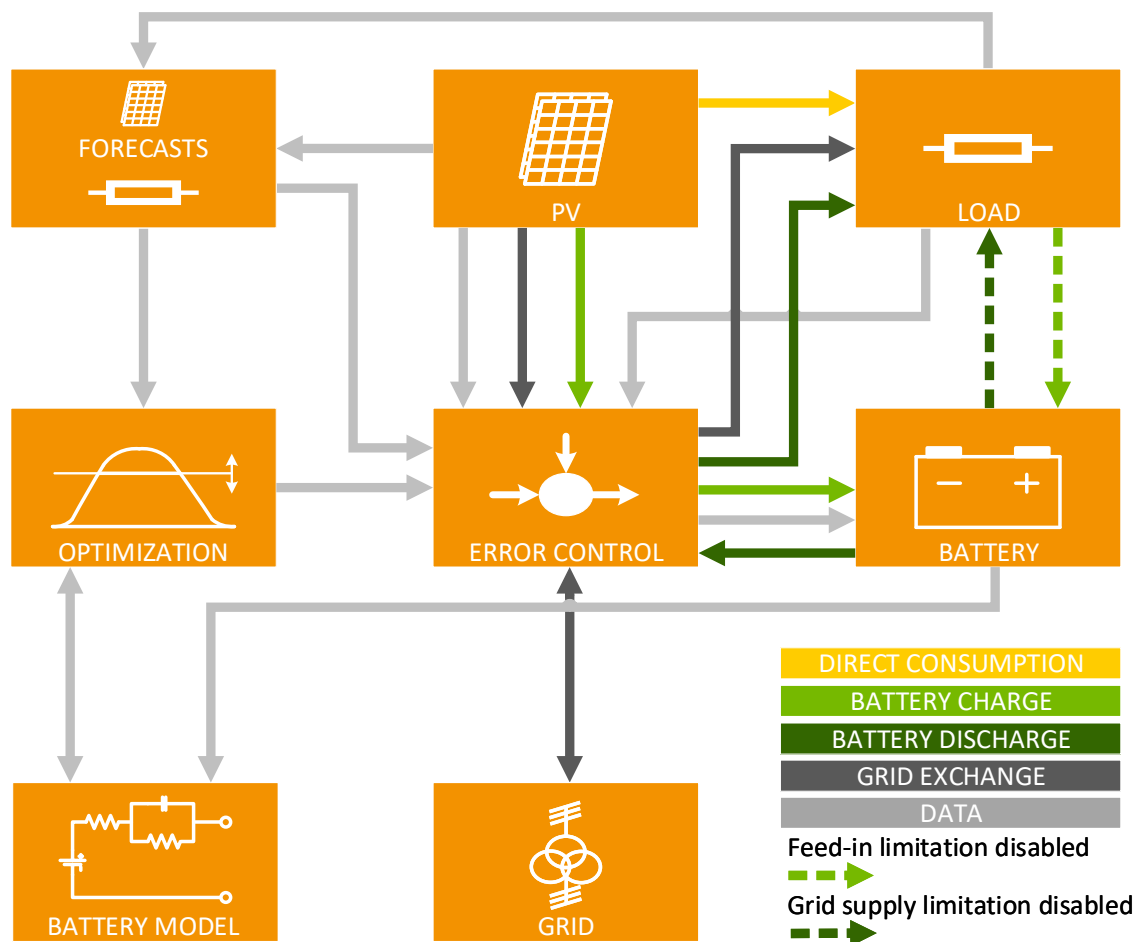
Figure 4.6: *Schematic representation of the PVprog control loop with all energy and data flows.*

The PV and load forecasts are passed on to the optimization iteration, and the battery's current SoC is sent to the battery model. The optimizer delegates the forecasts to the model, and in return receives the capacity at the end of the forecast horizon. It then passes the results to the error control, which compares the forecasts with the current measurements, and adjusts the optimizer's output accordingly. Finally, the adjusted charging/discharging power is sent to the battery as the set value. As shown by the colour-coded energy flows, the PV power is primarily used to cover the load (direct consumption). The paths of surplus PV power and PV deficits depend on the error control and on whether a feed-in or grid supply limitation is enabled, or both. If the feed-in limitation is enabled, the surpluses are either fed into the grid or used to charge the battery. If not, they are used for charging until the battery is full, and then fed into the grid. With load peak shaving enabled, PV deficits are covered by the grid or by battery discharge. Otherwise, the battery is discharged first, and the remaining deficits are pulled from the grid when full capacity has been reached.

5. PV system function block libraries

As part of this thesis, a set of IEC 61499 function block libraries[i] were created for the control of PV energy systems as described in section 1.2. Documented in this section are the FB interfaces and their internals. The underlying ST algorithms and the function blocks' internal variables can be examined in the published source code.

5.1. PVprog function block library

To make it possible to run PVprog optimization on an IEC 61499 PLC application, the components described in section 4 were implemented as function blocks in 4diac-IDE.

5.1.1. PowerForecaster function block

The `PowerForecaster` was created as an abstract template BFB for the `PVForecaster` and `LoadForecaster` BFBs (see sections 5.1.2 and 5.1.3, respectively). Potential new function blocks can be based on the `PowerForecaster` template to save time when designing the interface and ECC. The `PVForecaster` and `LoadForecaster` both share the `PowerForecaster`'s interface[ii] and implement the ECC with some slight extensions. The interface is depicted in figure 5.1. Initialization occurs with the `INIT` event, and the `QI` input set to `true`[iii]. Additionally, the `INIT` event requires the data inputs `TLB` and `TLF`, the time frame to look back for forecast generation and the forecast horizon, respectively. In an application, these inputs can either be set as constants (recommended) or as variables that can be set by the user. The default values are 3 h for `TLB` and 15 h for `TLF`. For normal operation, the FB has two event inputs. The input `UD` is linked to the data inputs `P` (The power measurement), `DOY` the day of the year and `TD` (the time of day, given in minutes since 12 AM).

---

[i]The libraries have been made available as open source at: `https://github.com/MrcJkb/PVTControllerLib`.

[ii]The `LoadForecaster` FB omits the `TLB` data input because it must be the same as the `TLF` input.

[iii]This is the case for most function blocks that need to be initialized. In all further descriptions, the explanation of the `QI` data input is omitted for brevity.

Initialization Request	-	Event	INIT	INITO	Event	-	Initialization Confirm
Forecast Update Request	-	Event	UD	UDCNF	Event	-	Update Algorithm Execution Confirmation
Forecast Request	-	Event	FREQ	FCNF	Event	-	Forecast Confirmation

PowerForecaster
1.0

Input event qualifier	-	BOOL	QI	QO	BOOL	-	Output event qualifier
Time frame to look back when generating forecasts in h	-	LREAL	TLB	PF	LREAL[96]	-	Forecasted power in W
Time frame to forecast ahead (look forward) in h	-	LREAL	TLF			
Measured power in W	-	LREAL	P			
Day of the year [1..365]	-	UINT	DOY			
Time of day [min] [0..1439]	-	UINT	TD			

Figure 5.1: *Interface of the* `PowerForecaster`*,* `PVForecaster` *and* `LoadForecaster` *function blocks.*
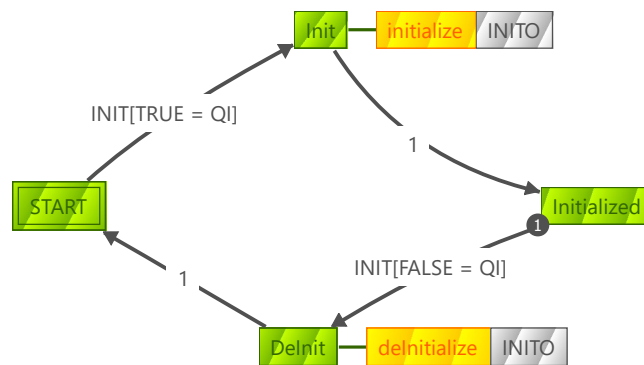
Figure 5.2: *ECC of the* `PowerForecaster` *function block: Initialization.*

It requests an update of the stored data and issues an UDCNF event after completion of the internal algorithms to confirm a successful update. The FREQ event requests a new forecast, resulting in the FCNF output event along with the predicted power PF, an array holding up to 96 values. This represents a forecast horizon of up to one day with a value for every 15 min interval. Forecast horizons above 24 h are invalid, due to the limited array size. If the forecast horizon is set below 24 h, any values of PF that exceed the set time frame should be ignored by the FBs that interpret the data. It must be noted that the TD data input is an unsigned integer. As a result, only one minute running means of the power measurements should be passed to the P input. Sending a UD request more frequently than once per minute would result in the stored data from the previous requests being overwritten until TD is incremented.

In order to provide a more comprehensive overview of the FB's internal functionality, the ECC is split into multiple figures. Figure 5.2 shows the initialization process. It is exactly the same as the example described with figure 3.2, and thus does not require further explanation. In the `initialize` algorithm, the internal variables are set according to the TLB and TLF data inputs. The `deInitialize` algorithm resets the internal memory, and thus the forecast array PF to zeros. The rest of the ECC is illustrated in figure 5.3. A UD event puts the FB into the `UpdateOp` state, in which the `analyzeInputs` algorithm is run. This determines the time since the last update and saves it to the internal variable, TIMESINCELASTUPDATE, among other things. With normal timing, i.e. one minute intervals between the updates, the `initNormalUpdate` algorithm is run, immediately followed by the `updateMemory` algorithm, which caches the data into the appropriate memory locations for later forecast generation. If for some reason one or more minutes have been skipped since the last update operation, the data for the missed updates is interpolated linearly. The `updateMemory` algorithm does not store the data inputs directly, but works with internal variables that are set by the `initNormalUpdate`, `initInterpolation` and `interpolate` algorithms. While `initNormalUpdate` simply sets the internal variables equal to the corresponding data

inputs, `initInterpolation` sets the power equal to that of the last `UD` request. Then, `interpolate` increments the power in a linear manner every time the algorithm is called, until the interpolated data for one minute before the current time is stored. Thereupon, the FB switches to the `NormalUpdateInit` and then to the `NormalUpdate` state. In the last state of the update operation, the `prepareForNextUpdate` algorithm stores the `P` and `TOD` data inputs in case an interpolation is necessary in the next `UD` request. Finally, the FB returns to the `Initialized` state and waits for additional event inputs.

Upon arrival of an `FREQ` event, the FB switches to the `Forecasting` state and runs the `getForecast` algorithm, before outputting a `UDCNF` event. This generates the `PF` data output. In the case of the `PowerForecaster` function block, the `updateMemory` and `getForecast` algorithm implementations are actually abstract, i.e. undefined, and must be implemented by the actual forecaster FBs. For correct operation, it is critical that the sequence of states that occurs after a `UD` event is finished and that the FB returns to the `Initialized` state *before* the arrival of an `FREQ` event. Otherwise, the `FREQ` event is ignored and forecasts are not generated. The correct timing of events in ensured by the `FB_FORECAST_TIMER` function block, which is described in section 5.1.9.

5.1.2. PVForecaster function block

An excerpt of the `PVForecaster`'s ECC that illustrates the extensions to the template is depicted in figure 5.4. Since night time values are disregarded for the computation of k_{Tf} (see equation 4.1), at powers equal to zero, the FB jumps straight to the `EndUpdateOp` state, storing the inputs and issuing a `UDCNF` event. Additionally, for internal array indexing reasons, the update sequence is skipped on the first and last minutes of the day to avoid runtime errors. Since the PV power should be zero at these times, this
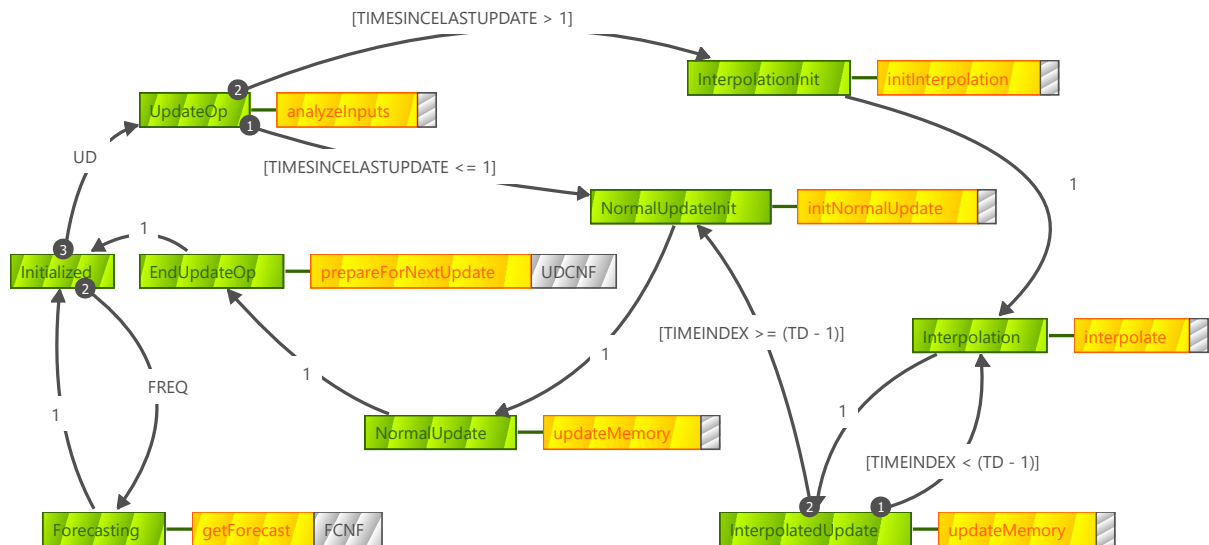
Figure 5.3: *ECC of the `PowerForecaster` function block: Normal operation.*
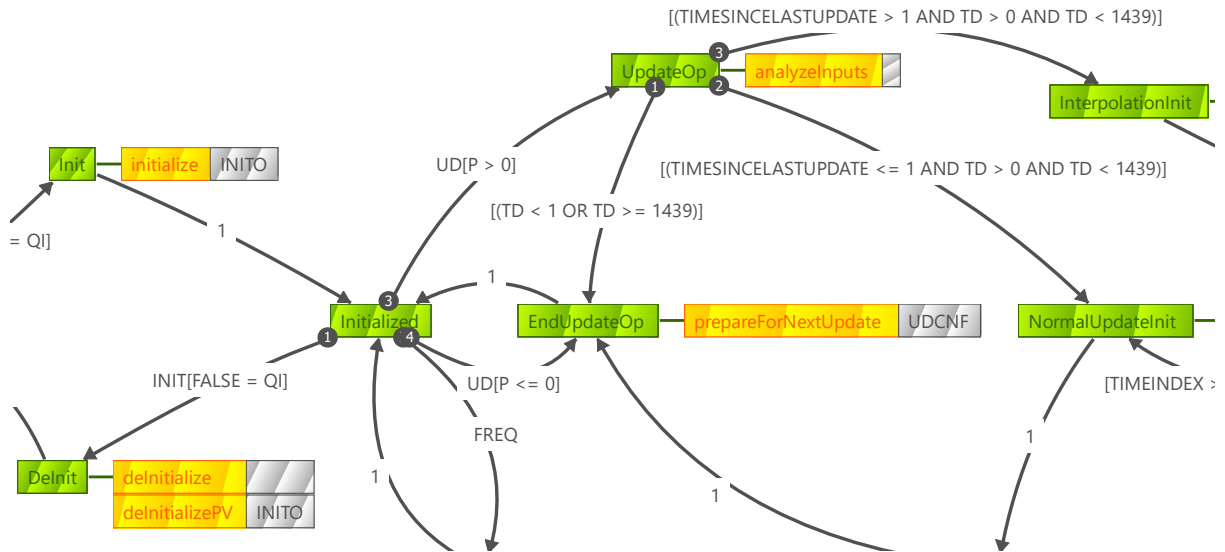
Figure 5.4: *Excerpt of the `PVForecaster` function block's ECC.*

addition may seem unnecessary. It was added; nonetheless, for increased stability. The last extension to the ECC is the addition of a `deIntitializePV` algorithm, which is run after `deInitialize` and resets `PVForecaster`-specific internal variables to zero.

5.1.3. LoadForecaster function block

The `LoadForecaster`'s ECC is exactly the same as that of the `PowerForecaster` template, with the addition of two initialization algorithms and one de-initialization algorithm. The corresponding portion of the `ECC` is depicted in figure 5.5. The `initializeLoad` and `deInitializeLoad` algorithms initialize and de-initialize `LoadForecaster`-specific internal variables. Finally, the weighting functions w_{lfP} and w_{lc} are initialized according to equation 4.6 within the `initializeWeights` algorithm.
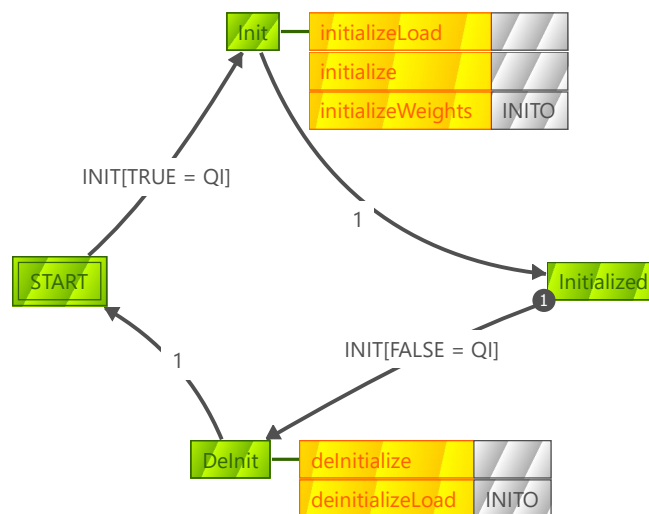
Figure 5.5: *Excerpt of the `LoadForecaster` function block's ECC.*

The only other alteration made from the template is the omission of the `TLB` data input. This was done due to the fact that the load forecasting algorithms require the day persistence time frame to be exactly the same size as the forecast horizon (see section 4.4). As a result, only one input, `TLF` is required. The `PVForecaster` and `LoadForecaster` implementations of the `updateMemory` and `getForecast` algorithms operate according to the corresponding equations in sections 4.3 and 4.4, respectively.

5.1.4. Battery model function blocks

The battery model described in section 4.6 is provided by the `SimpleBatteryModel` FB. Its interface is depicted in figure 5.6. To simplify the interconnection with the `BatteryOptimizer` FB (see section 5.1.5), the adapter concept (see section 3.3) is used. The adapter's outputs depicted in figure 5.6b represent inputs of the FB, and vice versa. Most of the function block's inputs are constant parameters that can be set according to battery data sheets. There are two input events. `SIM` requests a battery simulation with the data input `ECD` representing the requested energy over the simulated time frame in kWh. The `SIM` event is meant to be triggered by the `BatteryOptimizer` FB. After a completed simulation, a `SIMIN` event is sent out along with the estimated capacity at the end of the forecast horizon. `UD` is used to update the battery model with a new SoC, and should be triggered by the physical battery that the model represents. A successful update issues a `UDCNF` confirmation event. The ECC is illustrated in figure 5.7. It does not require any initialization, and starts in the `Idle` state, in which it awaits either a `SIM` or `UD` event.
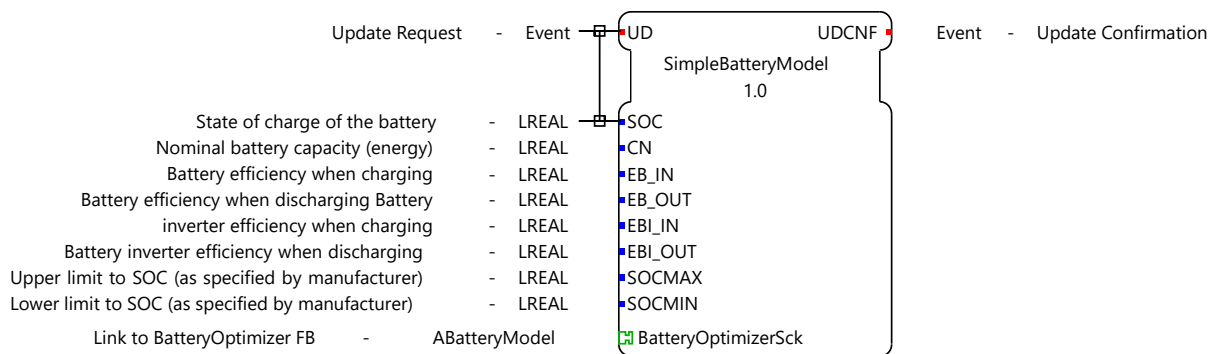
(a) `SimpleBatteryModel` *FB interface*
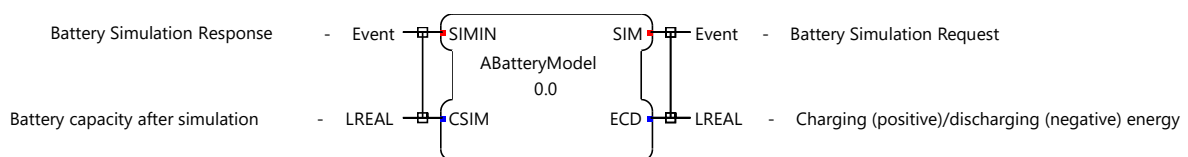
(b) `ABatteryModel` *socket interface*

Figure 5.6: *Interfaces of the* `SimpleBatteryModel` *function block (top) and the* `ABatteryModel` *adapter socket (bottom).*
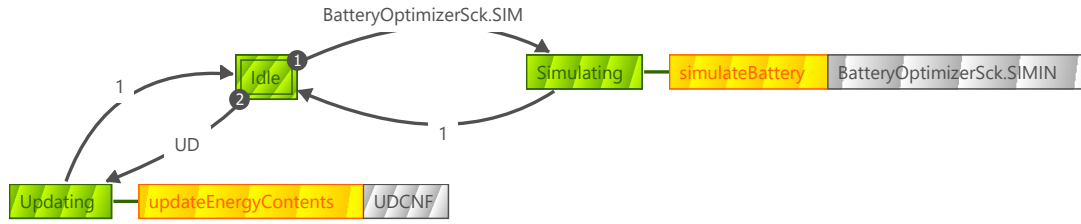
Figure 5.7: *The `SimpleBatteryModel` function block's ECC.*

In addition to updating the SoC, the `updateEnergyContents` algorithm also re-computes the maximum and minimum battery capacities, in case for some reason the SoC limits are changed during operation. It implements equations 4.8 through 4.10. The `simulateBattery` algorithm works according to equations 4.11 and 4.12.

It is envisaged that vendors who make use of the library developed within the scope of this thesis implement their own battery model to replace the `SimpleBatteryModel`. In such a case, the interface provided by the `ABatteryModel` adapter should be used as a basis. Another function block that was added to the library and can be used to represent the battery model is the `BatteryModelClient` CFB shown in figure 5.8. It acts as a wrapper for a `CLIENT_1` CSIFB (see section 3.4.2) and can be used to delegate the battery modelling to an external server[i]. This may be practical for the use with a device or simulation program that is capable of pre-simulating the battery.

5.1.5. BatteryOptimizer function block

The battery charge optimization is implemented in the `BatteryOptimizer` function block. Its interface is depicted in figure 5.9. To interconnect with a battery model, the FB utilizes an `ABatteryModel` plug, which mirrors the socket shown in figure 5.6b. The `REQ` event (along with the `CNF` output) are used to trigger the optimization iteration

---

[i]An equivalent function block, `BatteryModelServer`, was also created.

(a) *`BatteryModelClient`* **(b)** *`BatteryModelClient` composite network*
FB interface
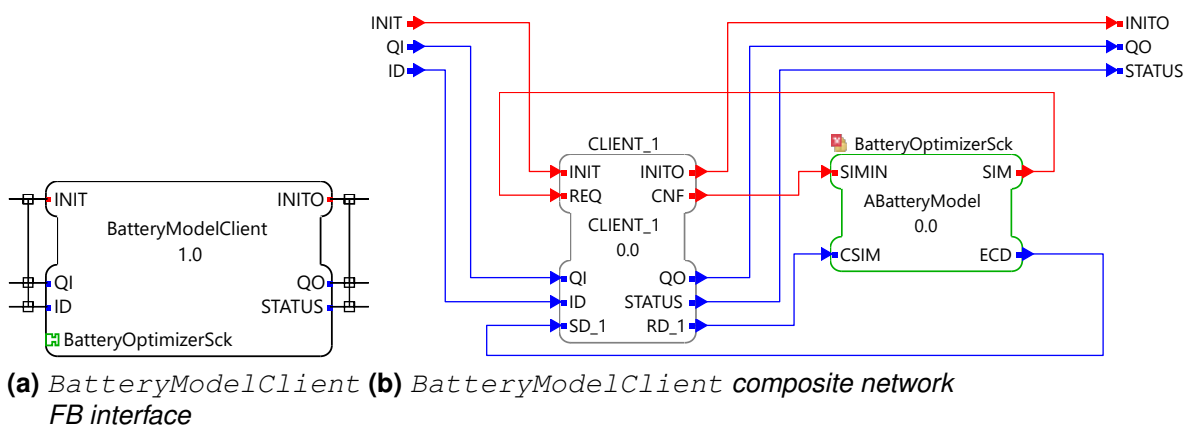
Figure 5.8: *Interface (left) and composite network (right) of the `BatteryModelClient` function block.*
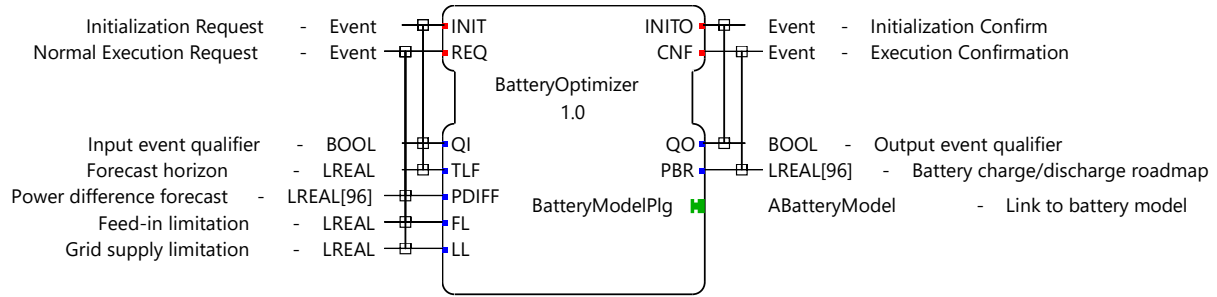
Figure 5.9: *Interface of the* `BatteryOptimizer` *function block.*

described in section 4.5. It requires the predicted power difference $P_{d,f}$, stored within the input array `PDIFF`, which is equal to $P_{pvf} - P_{ldf}$. In addition, the event is linked to the set load limit and the PV feed-in limit. To disable a limitation, the respective data input must be set to zero. Normally, the feed-in limitation is given in kW/kWp, but in this case, W was chosen as a unit for congruency with the load limit, and so as to simplify the interface. Taking the value in kW/kWp would require the nominal PV power as an additional input. Resulting from the optimization iteration sequence is the `PBR` output, which holds the battery charge roadmap (power in W) for the forecast horizon. The usage of the `BatteryModelPlg` adapter plug's `SIMIN` and `SIM` events, which are intended for coupling with the battery model, shall be disclosed later in this section.

An excerpt of the ECC (excluding the initialization, which is the same as that of the `PowerForecaster` FB) is pictured in figure 5.10. Regarding the feed-in (`FL`) and load (`LL`) limitations, there are four possibilities:

   i) Feed-in limitation

  ii) Load peak shaving

 iii) Both (i) and (ii)

 iv) Neither (i) nor (ii)
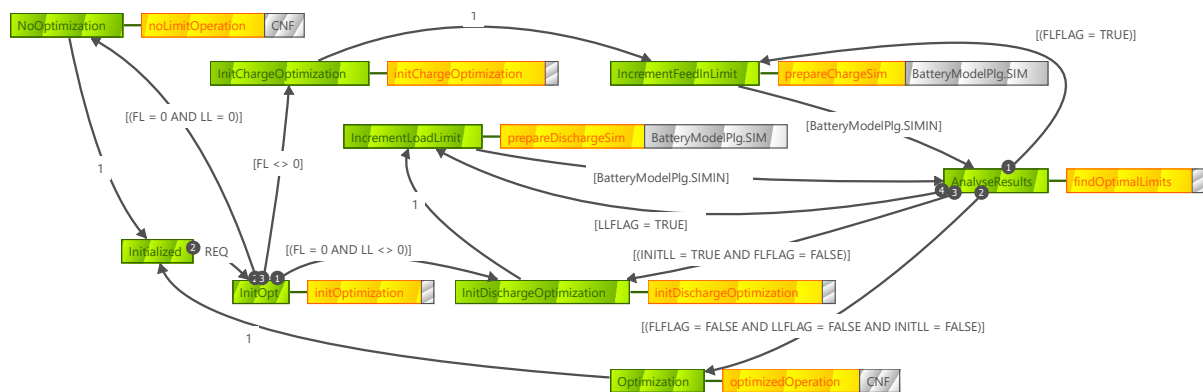
Figure 5.10: *Excerpt of the* `BatteryOptimizer` *function block's ECC. Note: In ST,* $<>$ *is the symbol for "not equal to".*

Option (iv) is technically redundant, since running the controller in such a mode would defeat the purpose of the PVprog algorithm. Nevertheless, it was included in the ECC for stability reasons. Rather than creating a separate initialization sequence for each of the options (i) through (iv), two sequences that merge together were created for the ECC. In the `InitOpt` state, this reduces the option checks upon arrival of the `REQ` event to:

i) Feed-in limitation

ii) Load peak shaving without a feed-in limitation

iii) Neither (i) nor (ii)

In both iterations, a factor is incremented between 0 % and 100 % of the respective set limits. If case (i) occurs, the sequence beginning with the `InitChargeOptimization` state is triggered. The boolean flags, `FLFLAG`, `LLFLAG` and `INITLL` are initialized to indicate which limitations are enabled. Then, the feed-in limit factor is initialized to -1 %, and subsequently incremented by 1 %, thus starting the iteration with a feed-in limit of 0 W. This takes place in the `prepareChargeSim` algorithm, which determines the PV surpluses above the virtual feed-in limit and sends a `SIM` event along with the charging energy over the forecast horizon to the battery model. The function block remains in the `IncrementFeedInLimit` state until a `SIMIN` response arrives. The `BatteryOptimizer` and battery model FBs are separated in this way to enable loose coupling. This makes it possible to replace the `SimpleBatteryModel` FB with a custom model without having to make any changes to the `BatteryOptimizer` FB. An illustration of how the two FBs interact with each other (as it would look without the use of an adapter) is presented in figure 5.11. The `SIM` output event of the `BatteryOptimizer` triggers the `SIM` input event of the battery model.
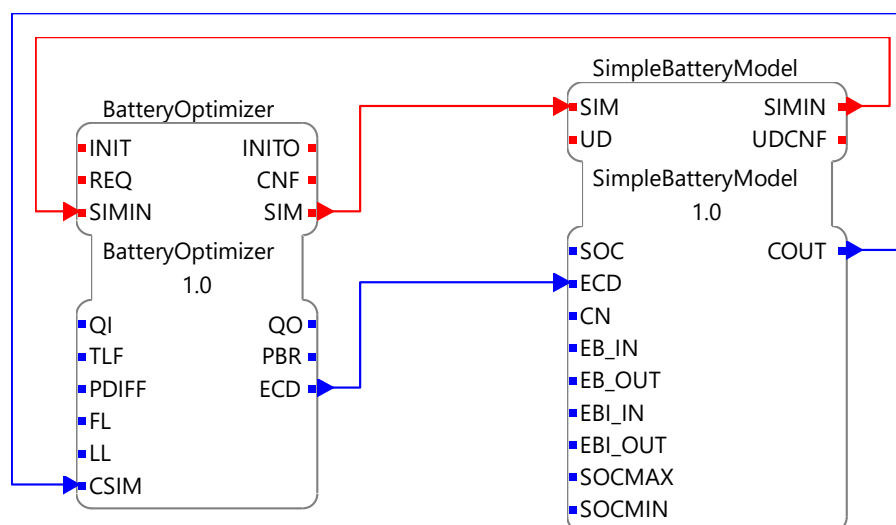
Figure 5.11: *Interaction between the `BatteryOptimizer` and `SimpleBatteryModel` FBs.*

Upon completion of the simulation, a `SIMIN` confirmation event is sent back to the `BatteryOptimizer` with the battery's estimated capacity at the end of the forecast horizon. Back in the ECC (figure 5.10), the `BatteryOptimizer` switches to the `AnalyseResults` state. Here, the results of the current simulation are compared to those of the last one. If an optimum (see section 4.5) is not found, the function block switches back to the `IncrementFeedInLimit` state, and the iteration continues. Once an optimum is found, the FB either performs the same optimization sequence for the load limit (if the `LL` data input was set to something other than zero and no load limit optimization has been completed) or it continues to the `Optimization` state. Here, the `optimizedOperation` algorithm computes the battery charge roadmap using the determined optima, before outputting a `CNF` event. Finally, the FB returns to its `Initialized` state, in which it awaits new requests.

5.1.6. ProgErrCtrl function block

Error controlling (see section 4.7) in the IEC 61499 PVprog implementation is handled by the `ProgErrCtrl` FB. Its interface is depicted in figure 5.12. The required data inputs are the forecast horizon (for initialization), the battery charge roadmap (generated by the `BatteryOptimizer` function block), the current PV power, the load and the feed-in and load limitations. With `CNF`, the FB outputs `PB`, the set value for the battery charging power in W. Figure 5.13 illustrates the ECC (excluding the initialization sequence). The function block utilizes two boolean flags to determine which states to switch to. `CCFLAG` indicates whether all of the control conditions listed in section 4.7 are met (`true`) or not (`false`). `BCDFLAG` specifies charging (`true`) and discharging (`false`). Starting from the `Initialized` state, a `REQ` event triggers the `initErrControl` algorithm, which determines whether the battery needs to be charged or discharged. If either is the case, `BCDFLAG` is set accordingly and `CCFLAG` is set to `true`. This results in the aforementioned control conditions being checked, after which `CCFLAG` is set again. If the conditions are met, the forecast error is corrected according to equations 4.13 or 4.14 for charging and discharging, respectively.
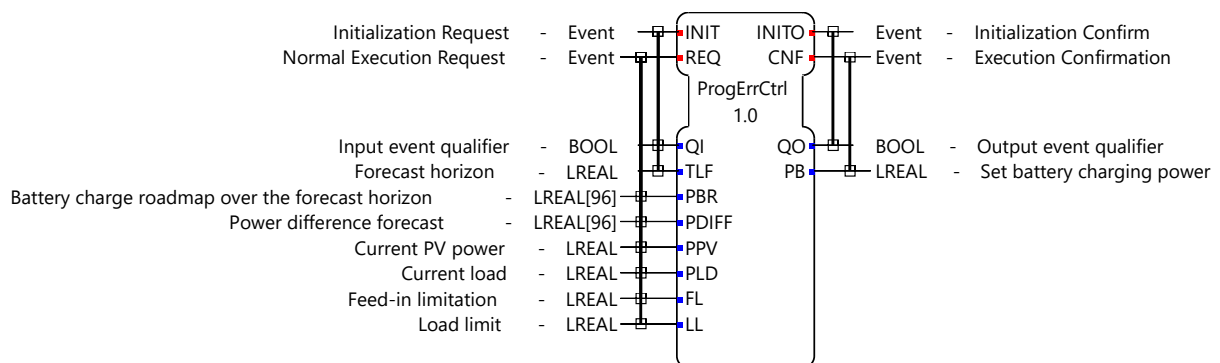
Figure 5.12: *Interface of the `ProgErrCtrl` function block.*
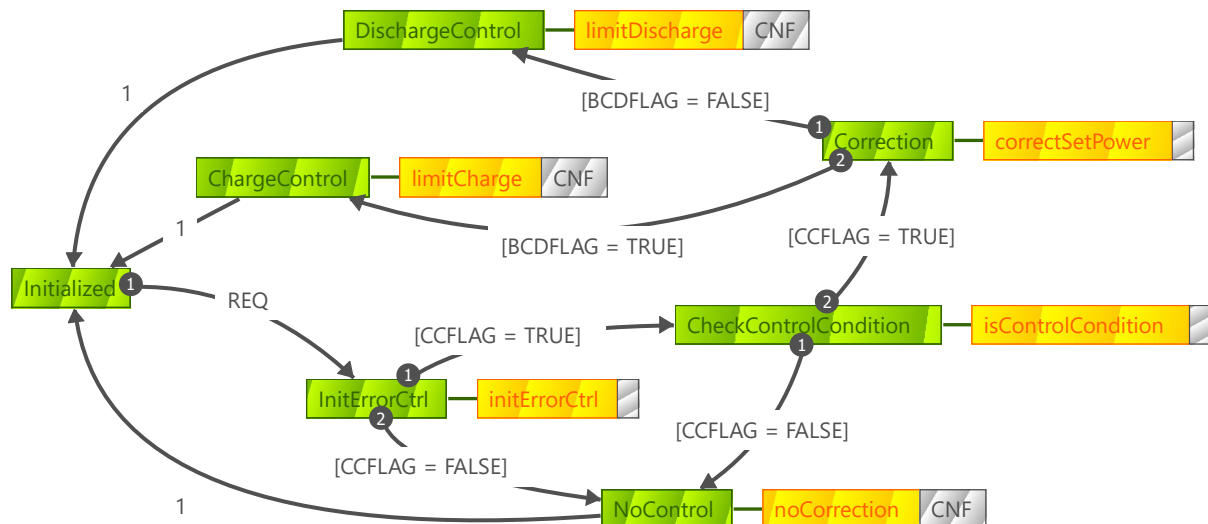
Figure 5.13: *Excerpt of the `ProErrCtrl` function block's ECC.*

Otherwise, the power difference is set to zero to prevent a stepped adjustment of the dynamic feed-in and load limits, before switching to the `NoControl` state, in which `PB` is simply set equal to the difference between `PPV` and `PLD`. As mentioned in section 4.7, a limitation of the battery's charging and discharging power to the battery inverter's limits was omitted from this implementation. Including it would require appropriate inputs to the interface, resulting in a tighter coupling between the `ProgErrCtrl` FB and the battery model, which would be undesirable. Here, the controlled battery must take over the job of limiting its set power, which is usually the case. If a battery inverter were not to limit the requested power for some reason, the IEC 61131-3 function block `F_LIMIT` could be appended to the `PB` output of the `ProgErrCtrl` FB.

5.1.7. Event synchronization

When handling multiple events that are issued in parallel in an IEC 61499 application, timing and synchronization can become a challenge. The following subsections deal with event synchronization used within the PVprog function block library.

5.1.8. Splitting and Rendezvous

In the case of the forecasters, two events, `UD` and `FREQ`, which each must be split and merged, are handled. It must be ensured that both the `PVForecaster` and `LoadForecaster` FBs have finished their operation before the optimization is triggered. For this purpose, two function blocks from the 4diac library are used: `E_SPLIT` and `E_REND`. The former is straightforward, and simply splits an incoming event into two output events. The latter merges two input events in such a way that the FB only outputs an event when both input events have been triggered at least once in a so-called "event rendezvous". An example is illustrated in figure 5.14.
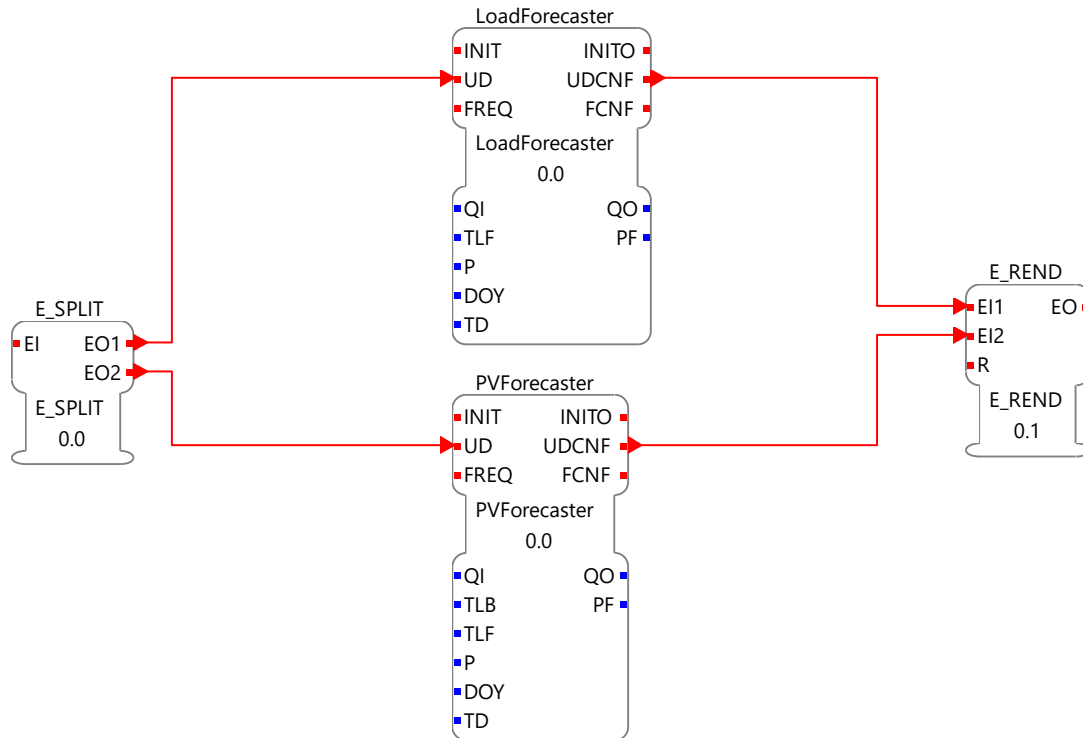
Figure 5.14: *Usage of the E_SPLIT and E_REND function blocks.*

The event rendezvous gets its name because a simple merge would output one event for each input event, thus triggering an output twice after receiving events from the PVForecaster and LoadForecaster FBs. A rendezvous outputs only one event when both input events have arrived.

5.1.9. Event timing

For fast performance without a significant loss of accuracy [15], it may be preferable to compute the forecasts in 15 min intervals instead of every minute, while updating the memory every minute. This can be achieved with clocks; but due to the forecasters' heavy reliance on the time of day, a method that utilizes time stamps is more reliable. At the same time, it must be ensured that the forecasters' memory update sequences (triggered by the UD event) have completed before an FREQ event is received. Otherwise, the FREQ event will be ignored (see section 5.1.1). All of this functionality is combined in the FB_FORECAST_TIMER function block, depicted in figure 5.15. The FB "translates" a series of REQ events into UD and FREQ events. Using the RS data input, the frequency with which the FREQ outputs are issued can be set. The remaining inputs, DOYIN and TDIN (excluding initialization), are required for internal synchronization in case a temporary failure results in the forecast computation being skipped. To ensure that the forecaster function blocks have completed their memory update sequences, an FREQ event is only triggered once they have issued their UD events, which are routed to the UDCNF event input.
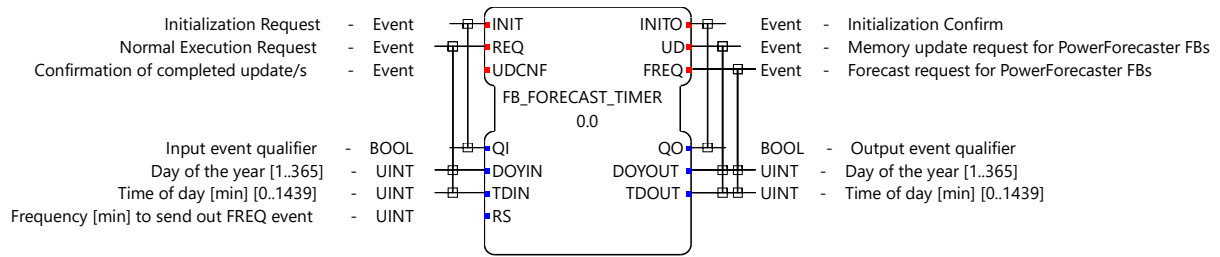
Figure 5.15: *Interface of the* `FB_FORECAST_TIMER` *function block.*

An example of the function block's usage in an application is presented in figure 5.16. The data links have been left out of the illustration for the sake of conciseness. An excerpt of the ECC is depicted in figure 5.17. The initialization and de-initialization algorithms (not depicted) set an internal COUNTER variable to zero. This variable is incremented by the time in minutes since the last request upon every REQ event. Additionally, the time stamp input data are delegated to the outputs, and a UD output event is issued. If the counter is smaller than the frequency set by the RS input, the data are cached for the next request and the FB returns to the Initialized state. Otherwise, the FB is transferred to the WaitForCnf state, in which it remains until a UDCNF event is received from the forecaster FBs. Once the event arrives, the counter is reset, an FREQ event is issued before the data are cached and the function block returns to the Initialized state.
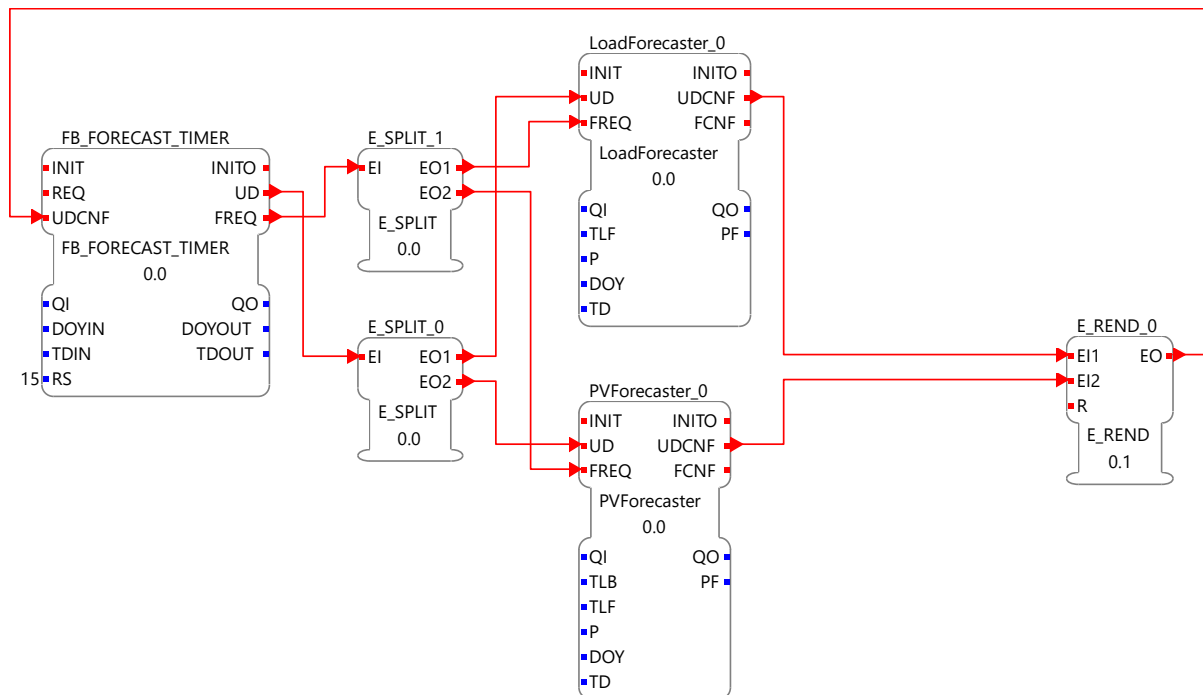
Figure 5.16: *Usage of the* `FB_FORECAST_TIMER` *function block in an application.*
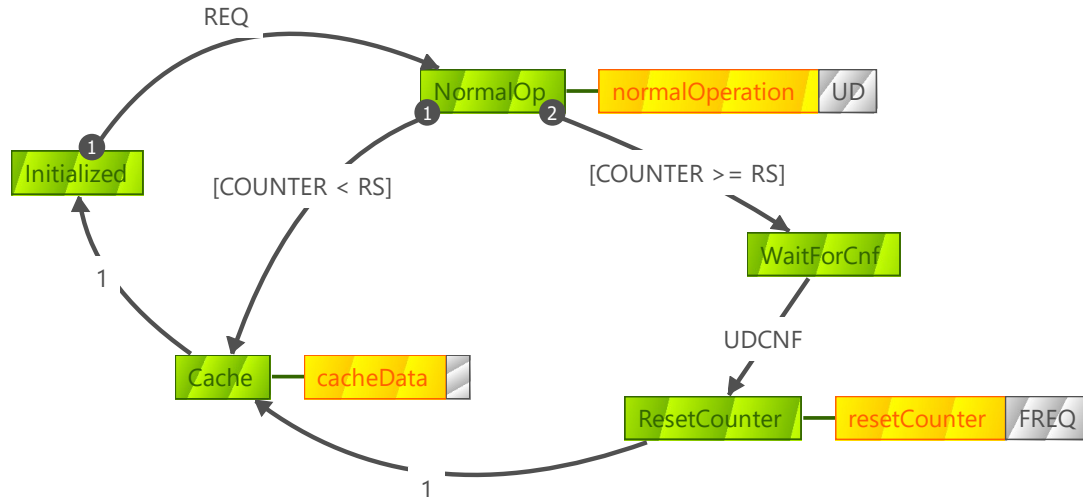
Figure 5.17: *Excerpt of the* `FB_FORECAST_TIMER` *function block's ECC.*

With the `CFB_FORECASTER` composite function block, a wrapper for the `PVForecaster`, `LoadForecaster` the event timing was created. Its interface is depicted in figure 5.18. The FB takes all of the `PVForecaster`'s and `LoadForecaster`'s data inputs along with a single event input for requesting updates and forecasts. The respective outputs are combined into a single `PDIFF` output, which represents $P_{d,f}$ over the forecast horizon. The composite network, with all of the internal event and data connections, is presented in figure 5.19. Unlike events, the data do not need to be synchronized. Thus, splitting data and sending them to two or more FBs does not require `E_SPLIT` function blocks. The `FB_SUB_LREAL_ARR96` FB takes two arrays of size 96 and subtracts each element of the array `IN2` from the respective element in the array `IN1`.

5.1.10. Input locking

If two requests follow in too short succession of one another (e.g., in a simulation in which requests arrive at an accelerated rate), the data sent to the forecaster FBs could theoretically be de-synchronized. Furthermore, problems could arise when requests are sent twice with the same time stamp.
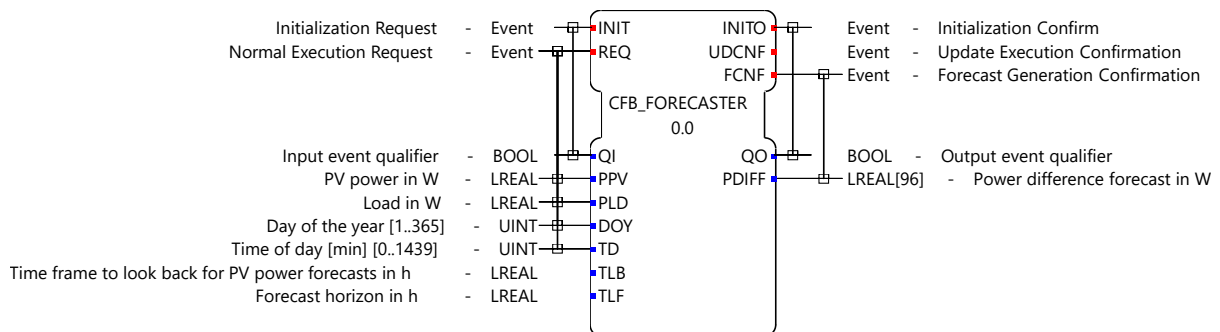
Figure 5.18: *Interface of the* `CFB_FORECASTER` *function block.*
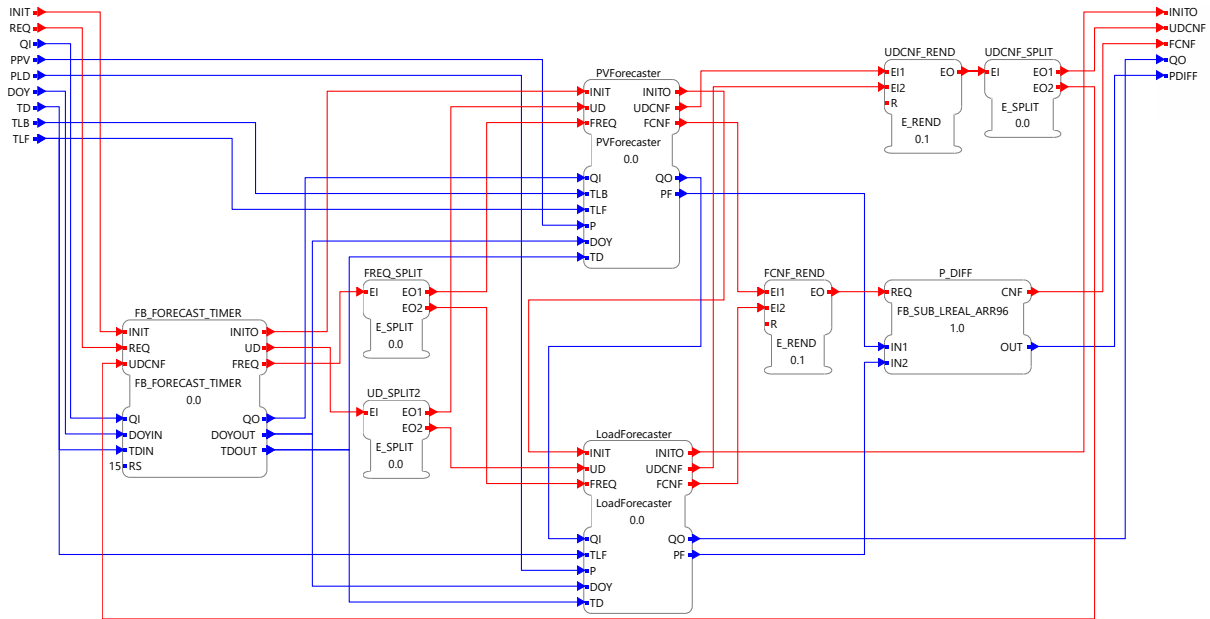
Figure 5.19: *The CFB_FORECASTER function block's composite network.*

To prevent such issues from occurring, a lock mechanism was devised. The PVPROG_LOCK function block that implements the mechanism is depicted in figure 5.20. It takes the input event L, along with the data required for the PVprog implementation, which it delegates to its outputs. Any subsequent requests are ignored until a U event unlocks the FB. In its unlocked state, it lets the next set of data through, as long as the TDI input has incremented by at least the value set by the DT input. On the right hand side, there are confirmation events for the L and U input events, respectively. Additionally, there is a REJ output event that can be used to notify FBs that a request was rejected. The corresponding RO data output returns true if the FB is currently unlocked and the rejection was due to insufficient time passing between requests, or false if the FB is currently locked. An illustration of the function block's usage is provided in figure 5.21. The data connections are omitted for brevity.
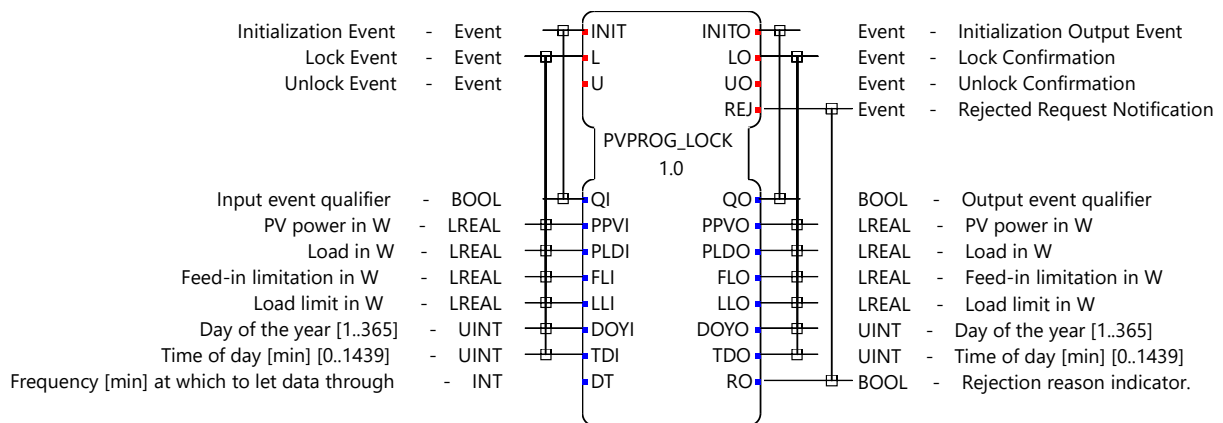
Figure 5.20: *Interface of the PVPROG_LOCK function block.*
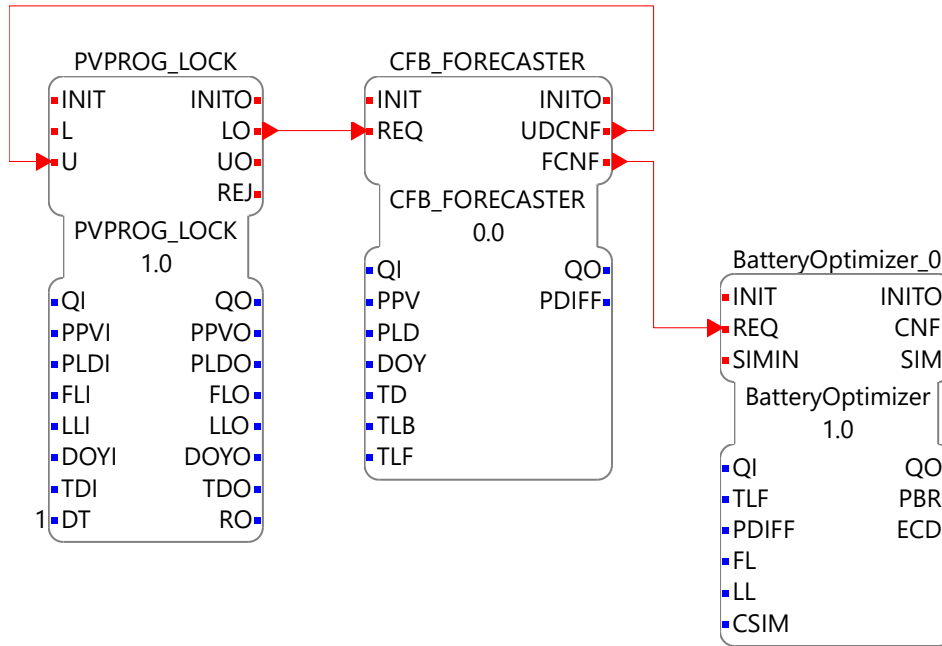
Figure 5.21: *Usage of the PVPROG_LOCK function block in an application.*

Upon triggering the L input event, the LO output event is forwarded to the REQ input of the CFB_FORECASTER FB, bringing the memory update and forecasting algorithms into motion. When the CFB_FORECASTER has completed its operations - instead of sending its output event directly to whichever FB is awaiting confirmation - it reroutes the event to the U input event of the PVPROG_LOCK FB, unlocking it for new inputs. The confirmation event can then be taken from the corresponding UO event output. Figure 5.22 illustrates the function block's ECC. The initialization sequence is slightly different than usual, in that an L event or a de-initialization request is required for it to leave the Init state. The delegateInputs algorithm, which delegates the input data to the FB's outputs, is called from within the LOCKED state. From within the UNLOCKED state, the FB must pass through the CheckTiming state before it can enter the LOCKED state again. If the time since the last request is shorter than the set threshold, it returns to the UNLOCKED state.
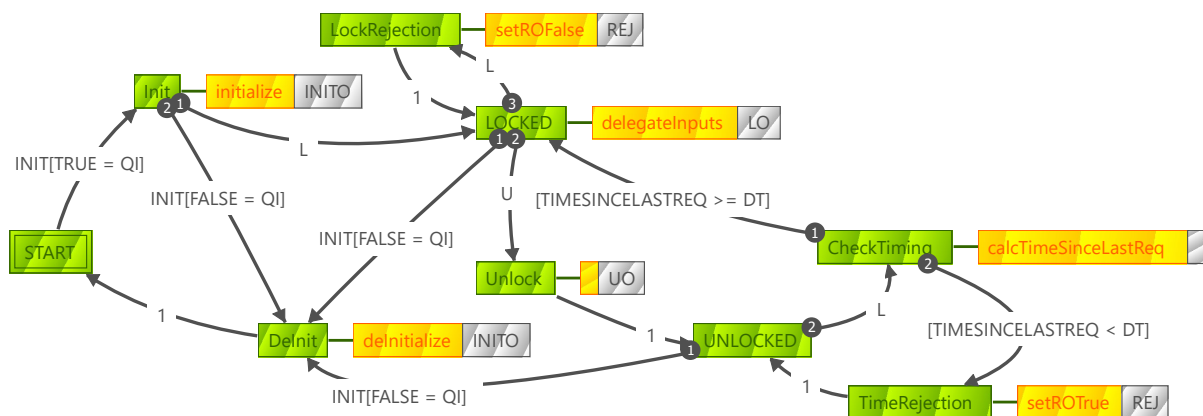
Figure 5.22: *ECC of the PVPROG_LOCK function block.*
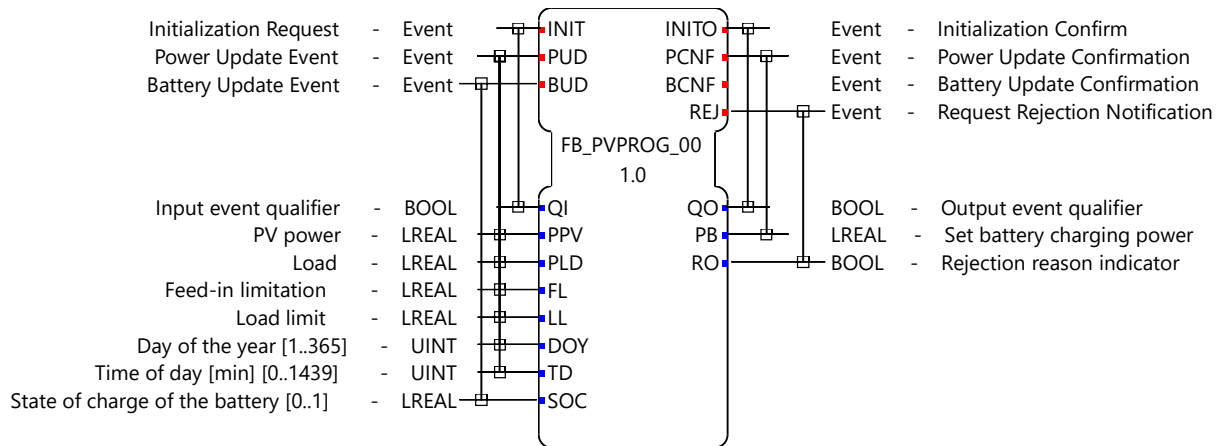
Figure 5.23: *Interface of the* `FB_PVPROG_00` *function block.*

5.1.11. PVprog composite function block

To further facilitate the usage of the PVprog function block library, an adapter was created in the form of a CFB. The interface is depicted in figure 5.23. Apart from the initialization, it has two separate input events: A power update, `PUD` and a battery update, `BUD` event. As shown in figure 5.24, `PUD`, used for forecast generation, is passed through the `PVPROG_LOCK` FB. On the other hand, `BUD`, used for updating the battery model's SoC, goes straight to the `SimpleBatteryModel` FB.
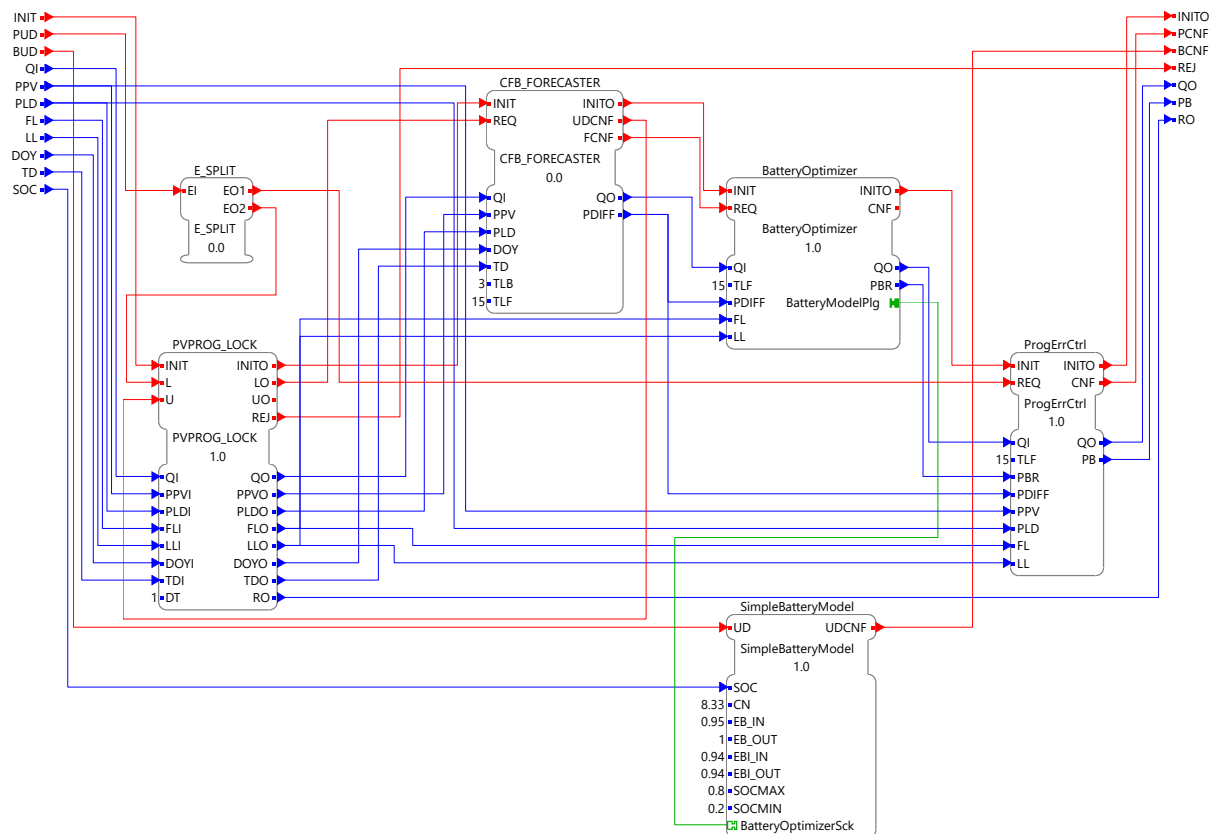
Figure 5.24: *The* `FB_PVPROG_00` *function block's composite network.*

As a result, the battery's SoC can be updated at any time. Likewise, the `ProgErrCtrl` FB is not bound to the locking mechanism, and can be updated with new PV power and load measurement at any time. Frequent updates to the SoC, P_{pv} and P_{load} are recommended for a smooth and reliable optimization. A direct use of the `FB_PVPROG_00` FB is not envisaged. It exists in the library mainly for the purpose of demonstrating what a PVprog subapplication could look like[i], and in order to provide quick access to the algorithm in 4diac for testing purposes.

5.1.12. Additional PVprog utilities

Due to the nature of its implementation, the PVprog function block library comes with a few limitations. To deal with these issues, utility FBs as described briefly in this subsection are provided with the library. Because the forecast generation relies on array indexing according to fixed time steps, the function blocks take day of the year (DOY) and time of day (TD) inputs instead of a time stamp. DOY is given as an unsigned integer between 1 and 366, and TD is an unsigned integer representing the minutes since 12 AM (a value between 0 and 1439). The IEC 61131-3 time stamp data type used in IEC 61499 applications is called `DATE_AND_TIME`. To enable the use of the PVprog FB library with this data type, two conversion FBs are provided: `DT_TO_DOY_UINT` and `DT_TO_TD_UINT`. They take a `DATE_AND_TIME` time stamp as an input and return the corresponding DOY and TD, respectively. The composite networks are depicted in appendix A.

To ensure that the data used for forecasting arrive in fixed intervals, the averaging CFB `F_N_MIN_MEAN_LREAL` can be used. Its interface is depicted in figure 5.25. As inputs, it takes data along with a time stamp and the time interval of which to compute the mean of the arriving values (specified as an IEC 61131-3 `TIME` data type). A `CNF1` event and the average of the input data that have arrived within the current interval is released every time the time stamp `TSI` indicates that the time since the end of the last interval has reached or exceeded the time specified by `N`. If the time since the last interval exceeds `N`, the mean up to `N` is estimated, and the remaining data are included

---

[i]At the time of writing this thesis, the ability to save subapplications for later use is not yet functional in 4diac.
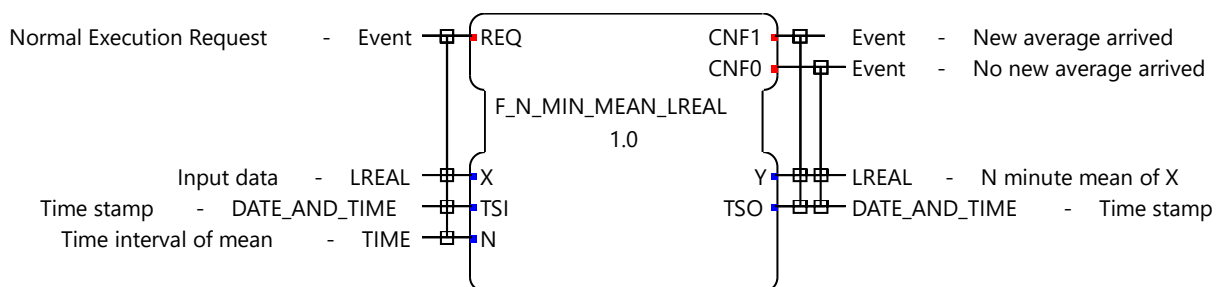
Figure 5.25: *Interface of the `F_N_MIN_MEAN_LREAL` FB.*

in the computation of the subsequent interval's average. To account for the possibility of extreme over-times (e.g., due to blackouts), the data are reset to zero if the time since the end of the last interval exceeds twice the size of the interval set by `N`. If no new average has been calculated from the arriving data yet, a `CNF0` event is issued. The function block's internal network is rather complex and exceeds the scope of this thesis. Thus, it is depicted in appendix A without any further explanation. A brief mathematical description is as follows:

A time interval $[t_{\mathsf{b}}, t_{\mathsf{e}}]$ is defined, where t_{b} is the beginning, t_{e} is the end of the interval, t_{l} is the point in time at which the last data sample x has arrived and t is the point in time of the current data sample's arrival: $t_{\mathsf{b}} <= t_{\mathsf{l}} < t \overset{!}{<=} t_{\mathsf{e}}$. The average $\overline{x}$ of the data within the interval $[t_{\mathsf{b}}, t]$ is determined by multiplying the current data sample with the fraction of time since the last data sample, normalized to the size of the interval $[t_{\mathsf{b}}, t_{\mathsf{e}}]$, and by adding it to the average of the interval $[t_{\mathsf{b}}, t_{\mathsf{l}}]$, which was determined in the last computation step.

$$\overline{x}([t_{\mathsf{b}}, t]) \approx \overline{x}([t_{\mathsf{b}}, t_{\mathsf{l}}]) + \frac{t - t_{\mathsf{l}}}{t_{\mathsf{e}} - t_{\mathsf{b}}} \cdot x(t) \tag{5.1}$$

If a sample arrives at t_{e}, the average is returned along with the time stamp t_{e}, and then reset to 0, resulting in the start of a new interval. If, however, t is greater than t_{e}, the returned average is reduced by the over-time, and returned with the time stamp t_{e}.

$$\overline{x}([t_{\mathsf{b}}, t_{\mathsf{e}}]) \approx \frac{\overline{x}([t_{\mathsf{b}}, t])}{1 + \frac{t - t_{\mathsf{e}}}{t_{\mathsf{e}} - t_{\mathsf{s}}}} \tag{5.2}$$

In this case, the data for the next time frame's average are not initialized to 0, but instead determined as

$$\overline{x}([t_{\mathsf{b}}, t]) \approx \overline{x}([t_{\mathsf{b}}, t_{\mathsf{l}}]) - \overline{x}([t_{\mathsf{b}}, t_{\mathsf{e}}]) \tag{5.3}$$

5.2. PV curtailment function block library

As mentioned in the previous sections, the PV forecasting algorithms rely on historical PV power measurements for generating their predictions. Theoretically, this should be the power before curtailment. In practise, however, curtailment occurs by regulating the voltage of the PV field, and moving a tracker out of its maximum power point (MPP), which results in the measured output being the derated PV power. Consequently, the PV power predictions may at times underestimate the potential output if curtailment occurs. To solve this issue, it makes sense to include the PV power curtailment in the control applications, and to use the output for estimation of the uncurtailed PV power. The function block library that was developed for this purpose is described in the following subsections.
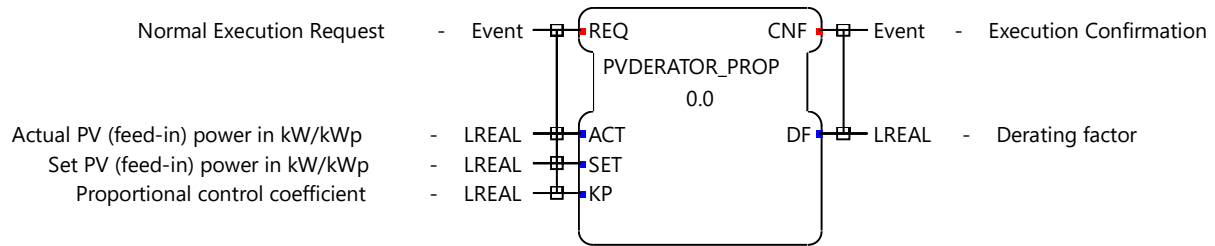
Figure 5.26: *Interface of the* `PVDERATOR_PROP` *function block.*

5.2.1. PV curtailment with regard to the current value

The simplest method for curtailment of PV power is by controlling with regard to the current value of P_{gf}. Due to the highly fluctuating nature a grid feed-in profile can have, a quick reaction capability of the controller is desirable. This is best achieved by implementing a proportional (P) controller, which compares the measured value to a set value, and adjusts the output $u(t)$ according to a proportional coefficient K_{p} to the control error $e(t)$.

$$u(t) = K_{\mathsf{p}} \cdot e(t) \tag{5.4}$$

The interface of the `PVDERATOR_PROP` CFB, which wraps a P controller for PV power curtailment, is depicted in figure 5.26. As data inputs, it takes the measured and set values, which could either be the PV power or the grid feed-in power, both normalized to P_{STC}, respectively. The control output is converted into a derating factor df, a value between 0 and 1 that can be used to specify how strongly the PV power output should be reduced. A df of 1 indicates no curtailment, and a df of 0 specifies zero output. If supported, a PV inverter or MPP tracker could use this value directly for curtailment. Devices that take the absolute power as the set value for power limitation would require a multiplication of df with the measured PV power. As shown in the CFB's composite network (see figure 5.27), the P controller's output is added to df in a recursive loop.
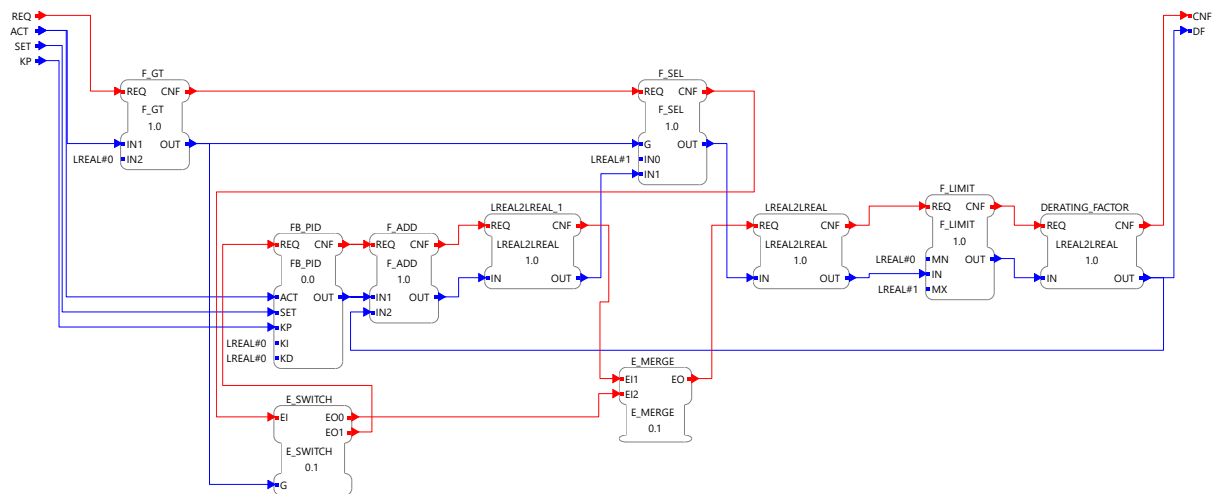
Figure 5.27: *Composite network of the* `PVDERATOR_PROP` *function block.*

The output itself is limited to 0 and 1. Generally, the PV power or grid feed-in will not jump from 0 to above the set value first thing in the morning. To ensure that no curtailment occurs at the beginning of the day, df is reset to 1 as soon as an `ACT` input equal to 0 arrives.

5.2.2. PV curtailment with regard to the running average

In some cases, curtailment with regard to the momentary value may not be necessary. For example, in Germany the 10 min running average of P_{gf} can be used as a reference value for the controller [16]. Doing so may provide the benefit of slightly lessened curtailment losses. Compared to a momentary feed-in profile, the fluctuations of an averaged feed-in profile are significantly reduced, as can be seen in figure 5.28. This changes the criteria from quick reactions to smoother adjustments of the output, for which a proportional-integral-derivative (PID) controller is well-suited. With a PID controller, the output is adjusted according to a proportional, integral and derivative weighted sum, where each term is weighted with a coefficient: K_{p} for the proportional term, K_{i} for the integral term and K_{d} for the derivative term.

$$u(t) = K_{\mathsf{p}} \cdot e(t) + K_{\mathsf{i}} \cdot \int_{t_0}^{t} e(\tau)d\tau + K_{\mathsf{p}} \cdot \frac{\partial e(t)}{\partial t} \tag{5.5}$$

Usually, the integral is computed over all errors since the initialization of the controller. This makes little sense for an intermittent grid feed-in profile; so in this case, the integral time frame is limited. For realization of a PID curtailment control with regard to the n min running average, the PVDERATOR_NMIN_MEAN CFB was created as part of the library. Its interface, which differs slightly from that of the PVDERATOR_PROP function block, is
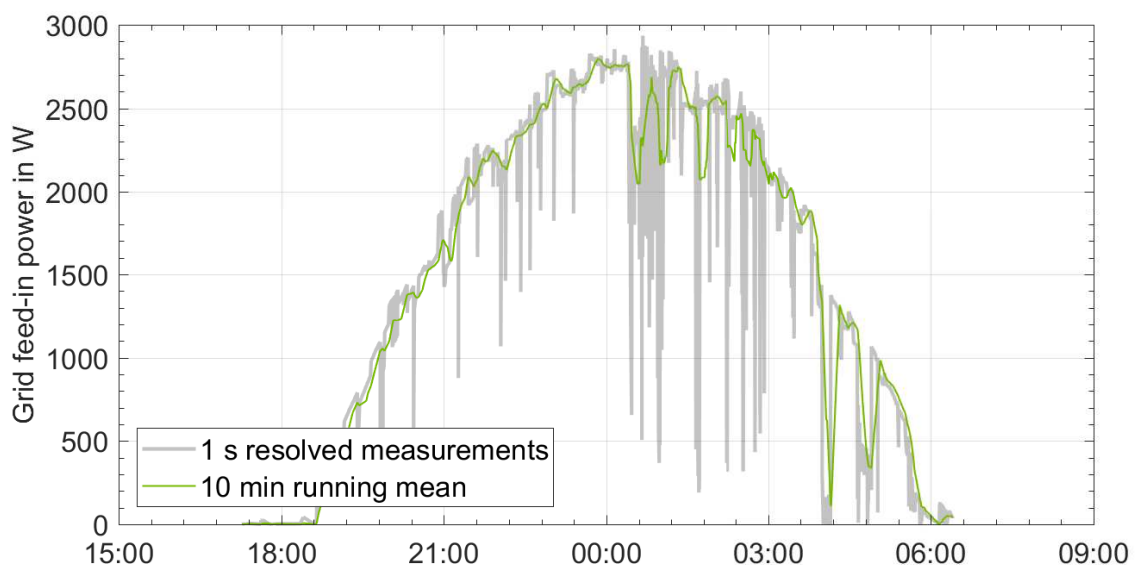
Figure 5.28: *Comparison of a 1 s resolved grid feed-in profile without curtailment with the 10 min running average of the same profile on an exemplary day.*
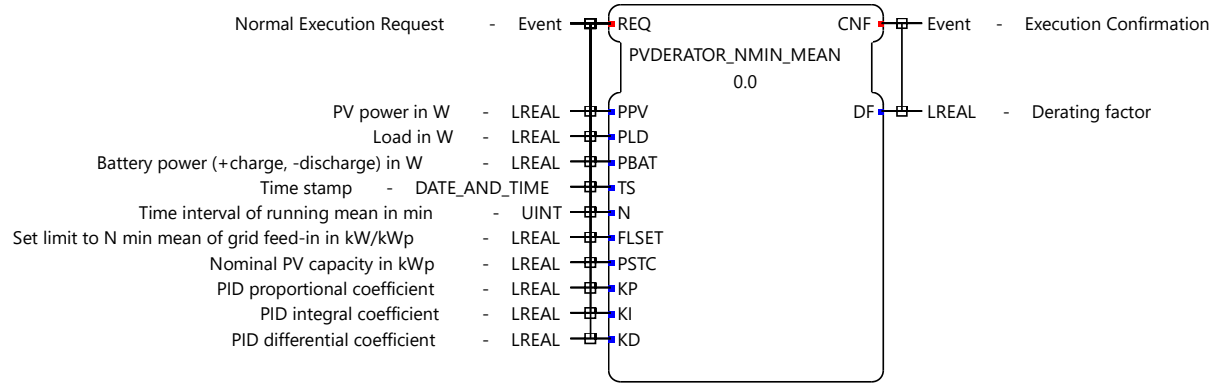
Figure 5.29: *Interface of the* PVDERATOR_NMIN_MEAN *function block.*

illustrated in figure 5.29. As inputs, it takes the PV power, load and battery power, a time stamp that is required for averaging, the interval n, the desired specific limit of the n min average grid feed-in, the nominal PV capacity and the three PID coefficients. Unlike its proportional counterpart, the PVDERATOR_NMIN_MEAN function block takes absolute values, with the exception of the feed-in limit, which is normalized to P_{STC}. The output DF can be used in the same manner as that of the PVDERATOR_PROP CFB. Figure 5.30 provides an illustration of the function block's composite network. First, the grid feed-in power is computed as

$$P_{gf} = \max(0, P_{pv} - P_{ld} - P_{bat}) \tag{5.6}$$

by the FB_PVFEEDIN_CALC CFB, and then normalized to P_{STC}. An F_N_MIN_RUNMEAN function block (detailed in appendix A) estimates the n min running average and passes it to a PID controller, with the integral time frame (limited to n min) as the actuator value. The F_N_MIN_RUNMEAN CFB utilizes an N_MIN_MEAN_LREAL CFB (see section 5.1.12), which issues an updated value in fixed-width time intervals.
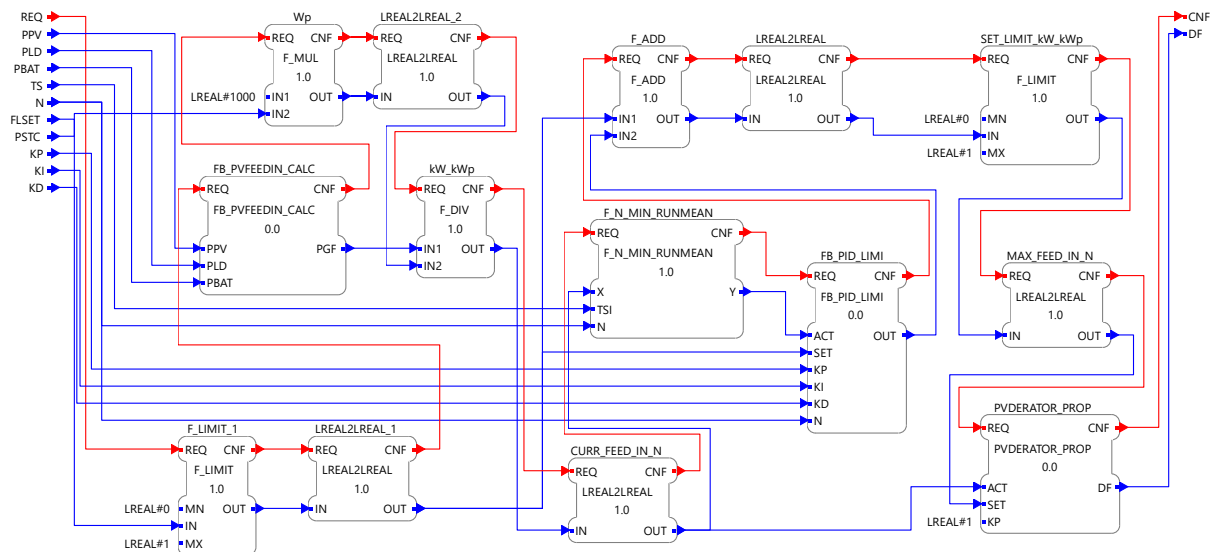
Figure 5.30: *Composite network of the* PVDERATOR_NMIN_MEAN *function block.*

As a result, equation 5.5 can be simplified to

$$u(t) = K_{\mathsf{p}} \cdot e(t) + K_{\mathsf{i}} \cdot \sum_{i=t-n}^{t} e(\tau_i) + K_{\mathsf{p}} \cdot (e(t) - e(t-1)) \tag{5.7}$$

where τ_i is the point in time at which an updated one minute average of P_{gf} is issued by the N_MIN_MEAN_LREAL function block. The PID controller compares the set limit to the running mean, and its output is added to it, resulting in a dynamic feed-in limit with respect to the momentary value. After limiting the intermediate result to $[0, 1]$, it is delegated to a PVDERATOR_PROP function block, which generates the df output as discussed in section 5.2.1.

5.3. SG Ready heat pump controller function block

With over $1{,}000$ heat pumps certified with the "SG Ready[i]" label [17], the specified control interface has been established as a standard in Germany. The standard dictates that heat pumps carrying the label must make it possible to influence their operation using two switch contacts. Each contact can have the state "on" (1) or "off" (0). This results in four operation modes, which are listed and described in table 5.1.
Unfortunately, there is currently no fixed definition for the SG Ready modes 3 and 4. From a user's perspective, it would be desirable to have clearly defined factory settings for the control modes, i.e. fixed set points for the storage tank temperature thresholds or increases thereof in addition to clarifications on whether or not cartridge heaters are used. As part of this thesis, a function block that can be used to control SG Ready heat pumps in a PV system was implemented. It is based on a simple control algorithm by Tjaden et. al, which aims to increase the self supply [19].

---

[i]SG stands for "smart grid".

Table 5.1: *The four states of SG Ready heat pumps. Source: [18]*

#	Contact states	Short description	Long description
1	$1 : 0$	Off	The heat pump is turned off for a maximum of 2 hours.
2	$0 : 0$	Normal	The heat pump runs in normal operation (determined by an internal controller).
3	$0 : 1$	Amplified I	A recommendation for the heat pump to turn on. Whether the heat pump actually is turned on or not, is determined by an internal controller.
4	$1 : 1$	Amplified II	A definitive instruction to turn the heat pump on as long as the internal controller deems it possible.
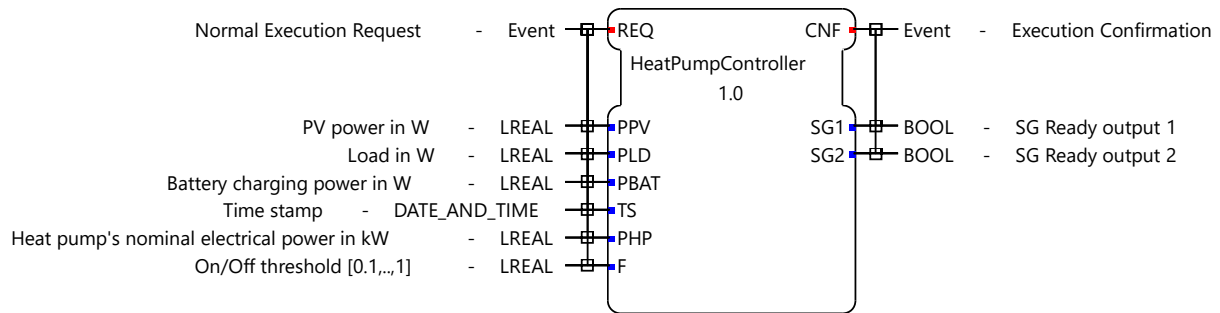
Figure 5.31: *Interface of the* `HeatPumpController` *function block.*

This is done by shifting the times in which the heat pump is powered on to times of excess PV power. The function block sets the SG Ready modes 2 and 3, depending on the conditions in equation 5.8.

$$\text{SG Ready mode} = \begin{cases} 3, & P_{\text{gf}} \geq f_{\text{on/off}} \cdot P_{\text{HP,nom}} \\ 2, & P_{\text{PV}} \leq f_{\text{on/off}} \cdot P_{\text{load}} \end{cases} \tag{5.8}$$

The factor $f_{\text{on/off}}$ is a threshold for when to switch. It should be set to a value between $1\,\%$ and $150\,\%$, depending on the ratio of the installed PV capacity and the heat pump's nominal power $P_{\text{HP,nom}}$ [19]. The function block's interface is depicted in figure 5.31. It takes the inputs required to compute the conditions for switching, whereby they initialize to 0 if not set by the user. So the PBAT input can be ignored if the system does not have a battery. For increased stability, each of the three power inputs is passed through an F_N_MIN_MEAN_LREAL function block, which outputs the 1 min average of the respective value every minute (see figure 5.32). On that account, a time stamp input is necessary. The CFB outputs two boolean signals representing the SG Ready switches. They can be passed directly to an analogue output of a PLC.
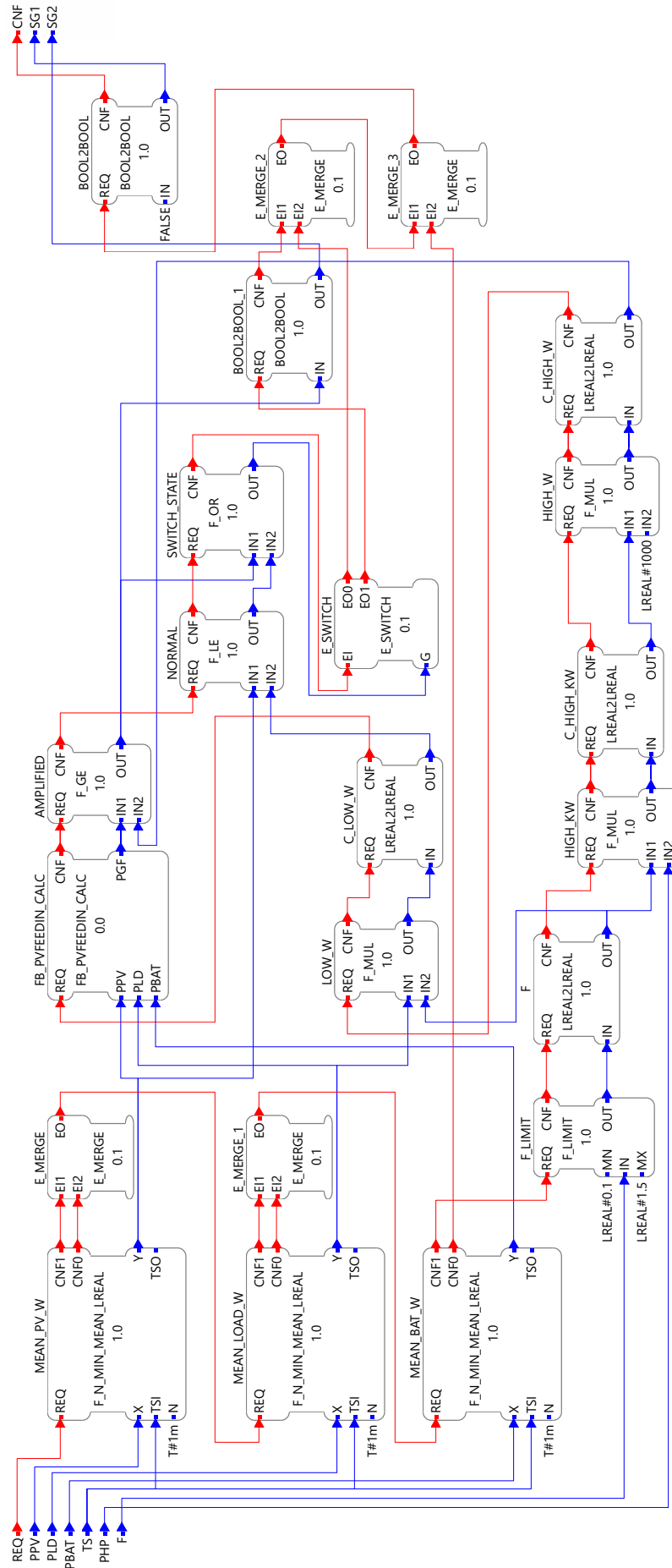
Figure 5.32: *Composite network of the* `HeatPumpController` *function block.*

6. Connection of IEC 61499 applications with Matlab®

Since 4diac is a relatively young application, its function block and application testing/debugging frameworks are still rather limited. To overcome these limitations and to improve the capability of validating 4diac applications through simulation, a communication library was developed in Matlab®, which allows the connection of IEC 61499 control applications with Matlab® simulation models. The library implements the TCP/IP (client/server) communication protocol (see section 3.4.2), and was made available as open source[i]. It should work with any IEC 61499 application, but has only been tested with applications running on FORTE. This section first briefly introduces three testing tools that are currently available with 4diac, and discusses their limitations. Then, the functionality and use of the `tcpip4diac` Matlab® communication library are described. Finally, 4diac/Matlab® co-simulations are performed to validate the IEC 61499 function block libraries and applications developed within the scope of this thesis.

6.1. 4diac testing tools

At the time of writing this thesis, 4diac comes with three validation tools:

- *FBTester*: For testing FBs in 4diac.

- *Live monitoring*: For monitoring running applications (i.e. "watching" inputs and outputs of FBs).

- *Boost Test*: A C++ unit testing library intended for testing the internals of FORTE.

6.1.1. FBTester

The FBTester is part of 4diac's function block development IDE, and is designed for the validation of individual FBs. To use it, an FB is deployed to FORTE and the input data must be entered manually. Events are triggered by clicking a button next to the respective input. This is illustrated in figure 6.1 for the `BatteryOptimizer` FB. When an event is triggered, the resulting output values are displayed next to the data outputs, and the number of output events issued is displayed next to the respective event outputs. On its own, the FBTester is limited by the fact that it can only be used to debug the external behaviour of function blocks, but not the internal implementation. For the internal implementation, it is currently necessary to launch FORTE in debug mode using a C++ IDE and place breakpoints within the underlying C++ source files of the FB under question. Additionally, the FBTester implements a very basic unit testing framework which allows one to run multiple tests in a sequence. Due to the fact that the tester is still in a prototype stage, however, these test sequences currently cannot

---

[i]Available for download at: https://github.com/MrcJkb/tcpip4diac
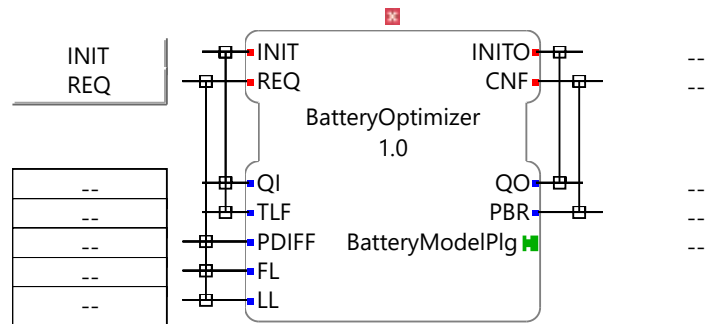
Figure 6.1: *Example of a function block being tested using the FBTester in 4diac.*

be saved. Also, the FBTester currently comes with the limitation that test cases often fail to initialize properly if a data input or output is an array that exceeds a certain size (as is the case in figure 6.1). It is also not possible to test adapter inputs and outputs such as the BatteryModelPlg output in figure 6.1. A partial workaround for this bug is to create a CFB that omits the affected inputs/outputs as a wrapper for the FB that is to be tested. Array outputs can then be analysed in the C++ source code. According to the 4diac contributors, a major rework of function block unit testing is envisaged for the future. In conclusion, the FBTester in combination with a C++ IDE is a very useful tool for the development process of BFBs. However, this requires programming experience that cannot be expected from an average user. On its own, the FBTester suffices only for very simple function blocks.

6.1.2. Live Monitoring

For testing IEC 61499 applications, 4diac offers a live monitoring feature. Here, complete applications are deployed to FORTE. Event and data inputs/outputs of the FBs can be "watched" as the application runs. With the monitoring feature, it is also possible to monitor the internals of subapplications and CFBs. At its current stage, it is further developed than the FBTester, and allows the partial monitoring of array inputs and outputs. To analyse the processes, one can "force" data input values and trigger event inputs. An example is depicted in figure 6.2, where its IN data input of the F_LIMIT FB was forced to a value of 5, and its REQ event input was triggered manually. The monitoring feature is a powerful tool for the development of applications and CFBs. In its current incarnation[i], it lacks the ability to set breakpoints; but this is planned for a future release. There is no built-in way to store and visualize data in graphs, but it is possible to use a CSV_WRITER FB to export the data and visualize them in an external application. The main drawback of the monitoring feature is the fact that debugging by manually triggering events can be an extremely tedious and time consuming process for complicated control applications. In the case of the PVprog algorithm, for example, simulating a single day of PV power with 1 min resolved data would require $1{,}440$ manual

---

[i]4diac version 1.8.4
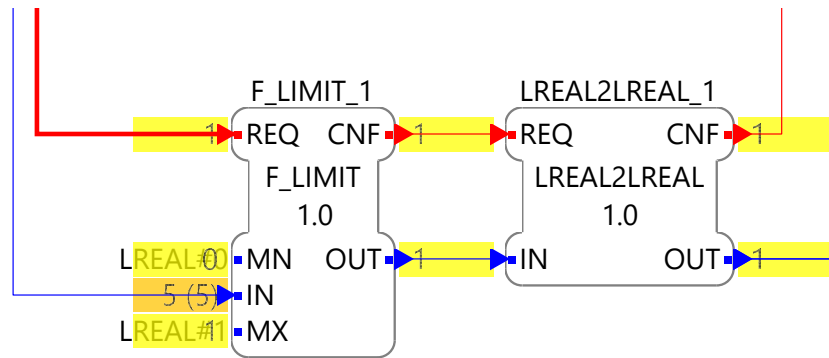
Figure 6.2: *Excerpt of a running IEC 61499 application being monitored in 4diac.*

mouse clicks and data inputs for the PV power, load and battery SoC, respectively; and another $1{,}440$ clicks for triggering events. With 10 days required to fully initialize the `PVForecaster` function block's cache, this would make a validation nearly impossible. A better solution would be to program a specialized CSV reader FB based on the `CSV_WRITER`[i] in C++. Since the CSV format is not standardised, however, a reader that works for one file may not work for another.

6.1.3. Boost Test

Boost Test is an open source C++ unit testing library that is used within the development of FORTE [11]. To enable it, one must download the Boost Test library from an external source, and include it in the FORTE project. Thus, a C++ IDE is required. While it requires knowledge in the C++ programming language, it is also the fastest testing framework, and has the potential to be the most powerful of all. Like the FBTester, one can test single function blocks. Technically, it is possible to set up complex simulations that make use of the function blocks and combine them to applications. As the term implies, however, unit tests are meant for testing isolated units of code, not for running entire simulations [20]. A common use of unit tests is to test already validated software units whenever source code changes are made to ensure their functionality is not impaired. To conclude, the Boost Test library should be used for smaller tests, and is unsuitable for validation of the FB libraries developed with this thesis.

6.2. 4diac/Matlab® TCP/IP communication library

For Ethernet communication using the TCP/IP protocol, Matlab® provides three classes: `tcpclient`, `tcpserver` and `tcpip`. The latter can act as both a client and a server. They communicate by sending and receiving either binary "byte"[ii] or ASCII ("American

---

[i]The source file is called "GEN_CSV_WRITER.cpp", and can be found in the /src/modules/utils directory of FORTE.

[ii]On the machine level, data are represented with bytes, where 1 byte can hold a number between 0 and 255.

Table 6.1: *Selection of IEC 61499 data types, their byte representations and their equivalent Matlab® data types.*

IEC 61499	`typeID`	Number of bytes	Matlab® equivalent
`BOOL` (`1`)	64	0	`logical`
`BOOL` (`0`)	65	0	`logical`
`SINT`	66	1	`int8`
`INT`	67	2	`int16`
`DINT`	68	4	`int32`
`LINT`	69	8	`int64`
`USINT`	70	1	`uint8`
`UINT`	71	2	`uint16`
`UDINT`	72	4	`uint32`
`ULINT`	73	8	`uint64`
`REAL`	74	4	`single`
`LREAL`	75	8	`double`
`STRING`	80	varies	`char`
`WSTRING`	85	varies	-
`DATE_AND_TIME`	79	8	-

Standard Code for Information Interchange") data. For communication between two Matlab® instances, the data do not have to be converted to a byte or ASCII representation. It does, however, for communication between Matlab® and an external socket. Unfortunately, neither of the Matlab® classes provided can be used directly for communication with 4diac `SERVER` or `CLIENT` function blocks running on FORTE. This is due to the fact that the byte-representations of the various data types differ between the two programs. Additionally, some IEC 61499 data types (e.g., `DATE_AND_TIME` and `WSTRING`) do not have direct Matlab® equivalents.

6.2.1. Data type representations

The main difference between the data types' representations is that FORTE adds an additional byte as an indicator for the data type (referred to in this thesis as a `typeID`), while Matlab's casting function, `typecast()` does not. Instead, `typecast()` casts[i] the data type to an 8 bit unsigned integer vector in which each element represents a byte (herein referred to as "raw byte data"). No `typeID` is needed, because the byte vector can be cast back into any data type with one of Matlab's casting functions (e.g., `double()` or `uint32()`). In some cases, FORTE not only adds a `typeID`, but also additional bytes that indicate the length of a string or the size of an array. In the following sections, the bytes that hold the `typeID` and additional information shall be referred to as a "byte header". Table 6.1 provides an overview of some of the IEC 61499 data types, their byte representations and their Matlab® equivalents. The `DATE_AND_TIME` data type

---

[i]Among other possible uses.

is represented as the number of milliseconds since January 1st, 1970 at 01:00:00 (loosely based on the UNIX time stamp format). The least significant byte (LSB)[i] is incremented first, and the most significant byte (MSB) last. The `WSTRING` data type represents wide character arrays (`wchar`), which currently do not exist in Matlab®. Each of the IEC 61499 data types can also exist in the form of an array. In FORTE, arrays' byte-data have a special header with a `typeID` of 118 followed by two bytes that represent the number of elements the arrays hold. This limits the sendable array size to 512 elements. The fourth header byte holds the `typeID` of the IEC 61499 data type the array holds. Appending the header are sequences of raw byte data (without additional headers) for each element of the array.

Connections between Matlab® and FORTE are possible using the three aforementioned classes, but it is a tedious process. To send data from Matlab® to FORTE, the Matlab® data first have to be cast into a byte vector, and then the correct byte header must be added so that the FORTE socket function block can interpret the arriving data correctly. To receive data from FORTE in Matlab®, the byte header must be interpreted, removed from the received vector and optionally cast to the corresponding data type. To automate the casting between Matlab® and IEC 61499 data types, the `tcpip4diac` class, which subclasses `tcpip`, was created. It is capable of "translating" the IEC 61499 data types listed in table 6.1 to their Matlab® equivalents and vice versa. Additionally, `Nx6 double` matrices in the `datevec` format and `DATE_AND_TIME` data can be converted to each other.

6.2.2. Use of the tcpip4diac class

The following subsection provides a brief introduction into the use of the `tcpip4diac` class. It is designed in accordance with the IEC 61499 compliance profile for feasibility demonstrations [8]. Thus, the interface is similar to that of a 4diac `SERVER` or `CLIENT` function block. The syntax to construct a `tcpip4diac` object in Matlab® is one of the following:

```
t = tcpip4diac(networkRole);
t = tcpip4diac(networkRole, remotehost);
t = tcpip4diac(networkRole, remotehost, port);
t = tcpip4diac(.., 'OptionName', OptionValue);
```

The first input argument, `networkRole` must either be `'server'` or `'client'`, and the destination is specified with `remotehost` (`'localhost'` by default for clients, and `'0.0.0.0'` by default for servers) and `port` (61500 by default). Additional parameters can be set using Matlab's `'Option'`/`Value` syntax. The syntax without additional options constructs a TCP/IP object that is capable of communicating with `SERVER_1`

[i]LSB and MSB can also stand for "least significant bit" and "most significant bit", respectively.

or `CLIENT_1` CSIFBs, respectively. To communicate with CSIFBs that have more or less than one data input or output, the `'DataInputs'` and `'DataOutputs'` options are used, as shown in the following examples. The `dataInputs` represent data sent to and the `dataOutputs` assume the role of data received from the socket on FORTE. As per definition in IEC 61499, the amount of the `tcpip4diac` object's inputs must be equal to the number of outputs of the corresponding CSIFB on FORTE, and vice versa for the amount of the `tcpip4diac` object's outputs. All of the IEC 61499 data types listed in table 6.1 are supported. Though possible, it is not recommended to use the `WSTRING` data type, which is cast to a `string` class[i] in Matlab®, and has no true equivalent.

To construct a server with two inputs and one output, the following syntax is used:

```matlab
% Specify the IEC 61499 data input types that are expected in a cell
% array.
dataInputs = {'UINT'; 'LREAL'};
% Call the tcpip4diac constructor
t = tcpip4diac('server', '0.0.0.0', 61500, 'DataInputs', dataInputs);
```

Following is an exemplary construction of a server with three outputs and one input:

```matlab
dataOutputs = {'UINT'; 'LREAL'; 'LREAL'};
dataInputs = {'UINT'; 'LREAL'};
t = tcpip4diac('server', '0.0.0.0', 61500, 'DataInputs', dataInputs, ...
    'DataOutputs', dataOutputs);
```

An empty cell array represents no inputs/outputs.

```matlab
t = tcpip4diac('server', '0.0.0.0', 61500, 'DataInputs', {});
```

Figure 6.3 summarizes the `tcpip4diac` class's essential functionality in a Unified Modeling Language (UML) class diagram. In addition, it depicts two 4diac `CLIENT` and `SERVER` function blocks with colour coded arrows illustrating event - method correspondence. An arrow on the left hand side of an event or method symbolizes it being triggered by an external source (i.e. another function block or line of code). Arrows on the right hand side of an event indicate that it can be triggered by the corresponding methods. In the case of the Matlab® sockets, methods cannot be triggered by events, because `tcpip` is not a `handle` class, and as such cannot implement so-called Observer (or listener) behaviour. However, as a workaround, some methods do not return either until a time-out period is exceeded or until the arrival of a request or response from the connected socket. This is indicated by an arrow to the right hand side of the method. Here, another drawback of the `tcpip` class comes to light.

---

[i]Requires Matlab® version R2016b or higher.
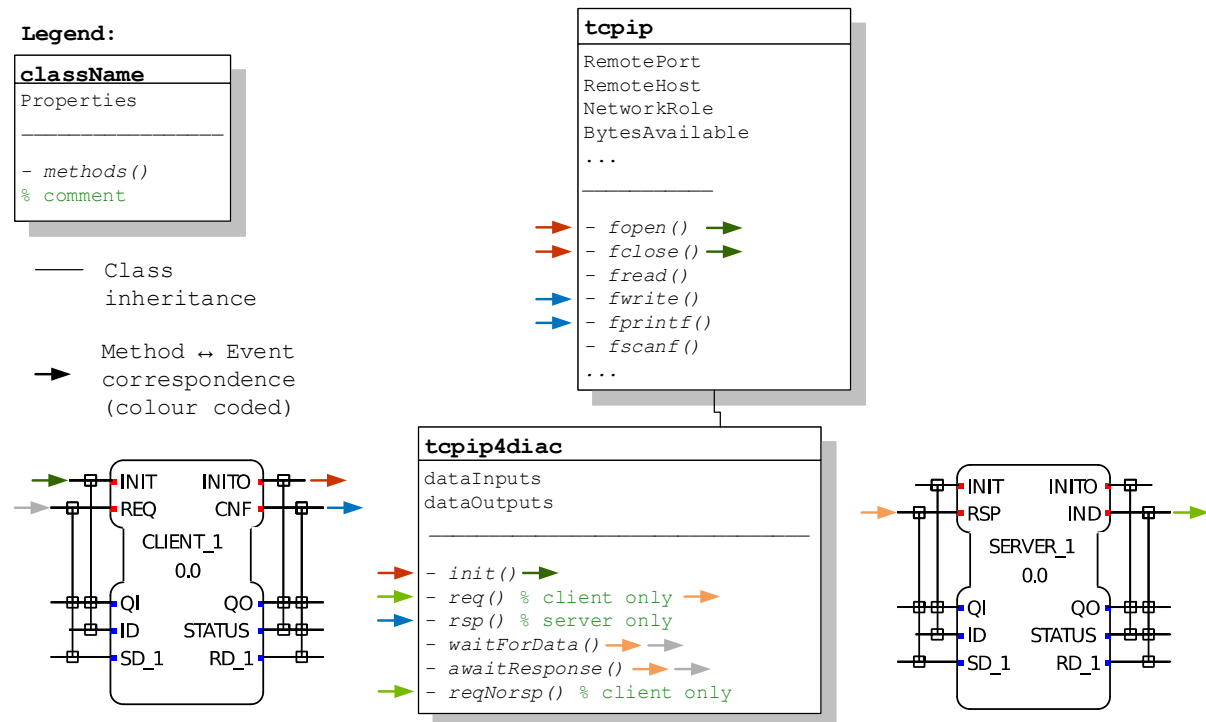
Figure 6.3: *UML class diagram of the `tcpip4diac` class and the correspondence of its methods with the events of 4diac `CLIENT` and `SERVER` FBs.*

The `fread()` and `fscanf()` methods for reading data do not implement this behaviour. For them to return a client request or server response, the data must have been sent already. If the object's time-out is exceeded, an empty `double` vector is returned. It is possible to increase the time-out, but unless the exact number of bytes being received is known beforehand[i], the method will not return until the time-out is exceeded. The `tcpip4diac`'s `req()`, `waitForData()` and `awaitResponse()` methods fix this issue by periodically checking the number of received bytes and returning when it stops increasing and has exceeded the minimum expected number of bytes. Through inheritance, all of the non-private `tcpip` methods (`fread()`, `fwrite()`, etc.) can be performed on a `tcpip4diac` object. However, it is recommended to use the `tcpip4diac` methods instead, wherever possible. To initialize or de-initialize a connection, the following syntax can be used, where `qi` is `true` for initialization, and `false` for de-initialization:

```
init(t, qi) % Omit output arguments
qo = init(t, qi); % Output event qualifier (logical)
[qo, status] = init(t, qi); % Outputs a status message
[qo, status, t] = init(t, qi, remotehost, port); % Change the remote
                                                  % host and port
```

In the `'server'` role, a connection is established once the `CLIENT` function block on FORTE is initialized. With the `tcpip4diac` object's IP set to the default `'0.0.0.0'`,

---

[i]This is often not the case for `STRING`s.

the FORTE `CLIENT` FB's ID must be configured to the PC's local IP address and the `tcpip4diac` server's port. The PC's local IP address can be queried using[i]

```
ip = tcpip4diac.getLocalHostIP
```

To connect to a `SERVER` FB on FORTE, it must be initialized before the `tcpip4diac` client. In this case, the IP on both sockets can be set to the string `'localhost'`.

In the `'client'` role, the `req()` method is used to send requests to a `SERVER` FB. This method does not return until a response is received. An error message is issued if it is called on an object in the `'server'` role. For multiple data inputs, the inputs `in1, ...`
`.., inN` (where `N` is the number of inputs) are automatically cast to the corresponding IEC 61499 data types (see table 6.1) before being sent to the connected function block. The returned output data types `out1, .., outM` (where `M` is the number of outputs) depend on the corresponding IEC 61499 FB's input data types. If the `tcpip4diac` object only has one input, it must be cast to the corresponding Matlab® data type in table 6.1 before being passed to the `req()` function. To send data to FORTE in the form of an array, the data must be passed as a `Nx1` vector.

The syntax for sending a single data input in a request is as follows:

```
[out1, .., outM] = req(t, in);
```

Whereby the Matlab® data type of `in` must correspond with the data type expected by the FORTE application (e.g., a `single` if a `REAL` is expected, see table 6.1). Multiple data inputs can be wrapped in a cell-array.

```
inData = {in1, .., inN}; % Cell-array of inputs
[out1, out2, out3, .., outM] = req(t, inData);
```

If data are sent in Matlab's `datevec` format, the output received by the FORTE `SERVER` will be of the `DATE_AND_TIME` type.

```
in = datevec(now);
[out1, .., outM] = req(t, in);
```

In both the `'client'`[ii] and `'server'` role, the `waitForData()` function can be used to await data from a FORTE socket. It will not return until either a `REQ` or `RSP` event is received or a time-out is reached. The default time-out is `inf` if none other is specified. The data types of the outputs `out1, .., outM` correspond with those of the CSIFB

---

[i]Requires JAVA™.

[ii]With the `req()` method, this function is obsolete for client objects.

inputs `SD1, .., SDN` (see table 6.1).

```
[out1, .., outN] = waitForData(t);
[out1, .., outN] = waitForData(t, timeoutS);
```

In `waitForData`, an error message is issued if the number of function outputs is not equal to the number of data outputs specified in the constructor. To wait for a response (or request), and ignore the received data, the `awaitResponse()` method can be used.

```
awaitResponse(t) % Does not return until a REQ or RSP event is received
```

A response is sent using the `rsp()` method. The syntax for a single data input, which must be cast to the correct data type beforehand, is as follows:

```
rsp(t, in)
```

Multiple inputs are wrapped in a cell-array.

```
inData = {in1, .., inN}; % cell array of inputs
rsp(t, inData)
```

Sometimes it may be necessary to send data from Matlab®, and perform further computations before a response arrives (e.g., send more data to another socket on FORTE using a second `tcpip4diac` socket). For this purpose, the `reqNorsp()` method was added. This method returns regardless of whether a response is received or not. To await a response in this case, the waitForData() or awaitResponse() methods must be called manually. An example of its intended use is as follows:

```
reqNorsp(t, inData); % send data.
% Perform intermediate operations, e.g.,
rew(t2, inData2); % send data to different socket
[out1, ..,outN] = waitForData(t);
% Alternatively: awaitResponse(t);
```

An exemplary communication sequence between a `tcpip4diac` client and a `SERVER_1` function block on FORTE is illustrated in figure 6.4. In the UML representations of the `tcpip4diac` object, only the relevant properties and methods for the respective operations are shown along with their values. Method outputs are grouped with the class's properties. The correspondences of events and methods are depicted with colour coded arrows, as is done in figure 6.3. Inputs and outputs of the `SERVER_1` function block are visualized with 4diac's live monitoring feature (see section 6.1.2). First, a connection is initialized between the two sockets.
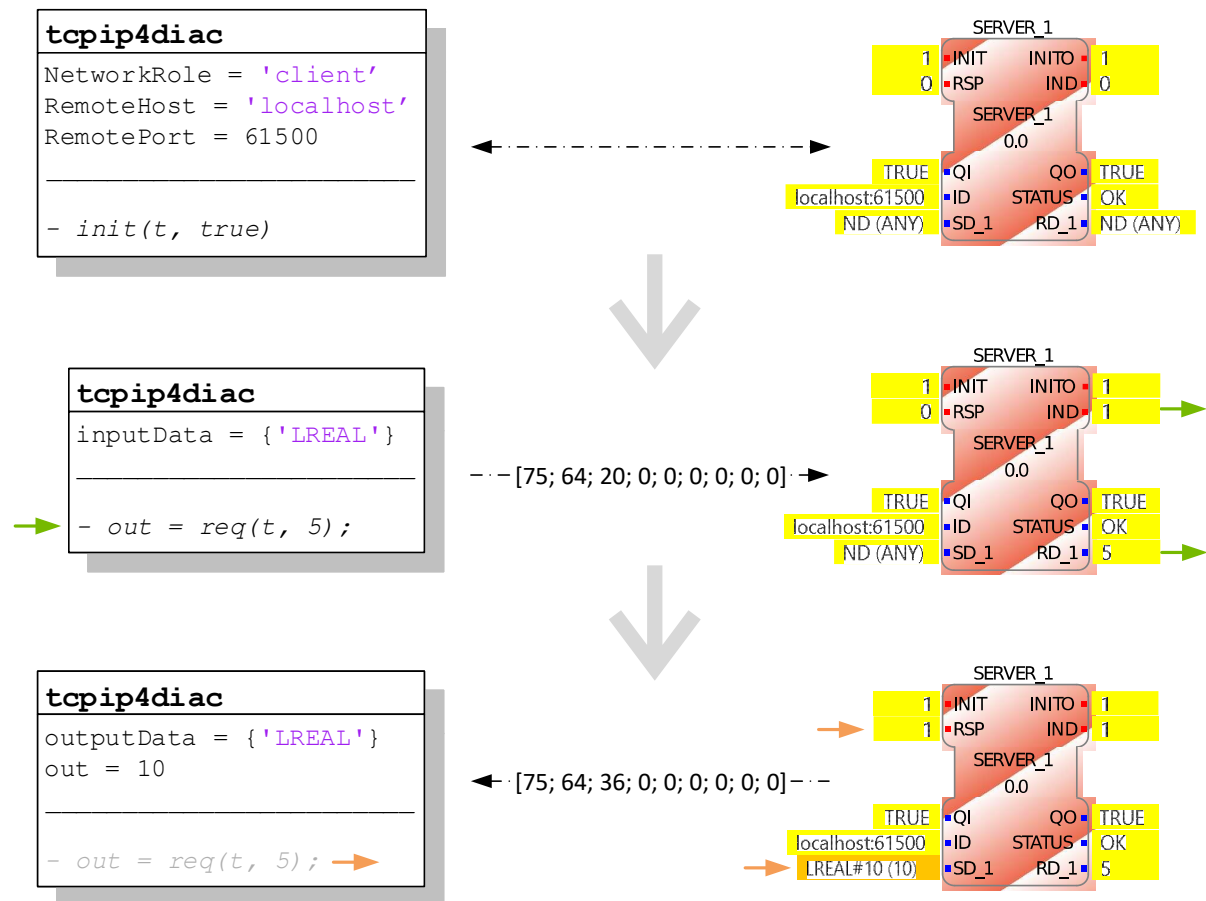
Figure 6.4: *Visualization of a communication process between a `tcpip4diac` client in Matlab®
and a SERVER_1 FB on FORTE.*

The `tcpip4diac` object connects to the already initialized SERVER_1 FB. Next, the `req()` method is called in Matlab®, with a `double` variable of the value 5 passed to it. The `tcpip4diac` instance converts the variable to an appropriate array and appends it to the LREAL typiID, 75. This triggers the IND event on the SERVER_1 FB and outputs an LREAL value of 5. In 4diac, the CSIFB's output is passed to another FB for further use. This was excluded from figure 6.4 for brevity. In the final step, the FB's SD_1 input is "forced" to a value of 10, and a RSP event is triggered using 4diac's live monitoring feature. The binary data are sent to Matlab® and cast to a `double`, allowing the `req()` method that was called in the second step to return.

The same communication sequence with reversed roles is depicted in figure 6.5. This time, the `tcpip4diac` server is initialized first. The initialization of the CLIENT_1 function block establishes a connection and allows the `init()` method to return. This in turn allows the `waitForData()` function to be called. Using the 4diac live monitoring, a REQ event is triggered along with an LREAL value of 5 for the SD_1 data input. The binary data are sent to the `tcpip4diac` instance, whose `waitForData()` method returns it as a `double`. After processing the received data, Matlab® passes the response (a `double` with a value of 10 in this example) to the `tcpip4diac` object via its `rsp()` method.
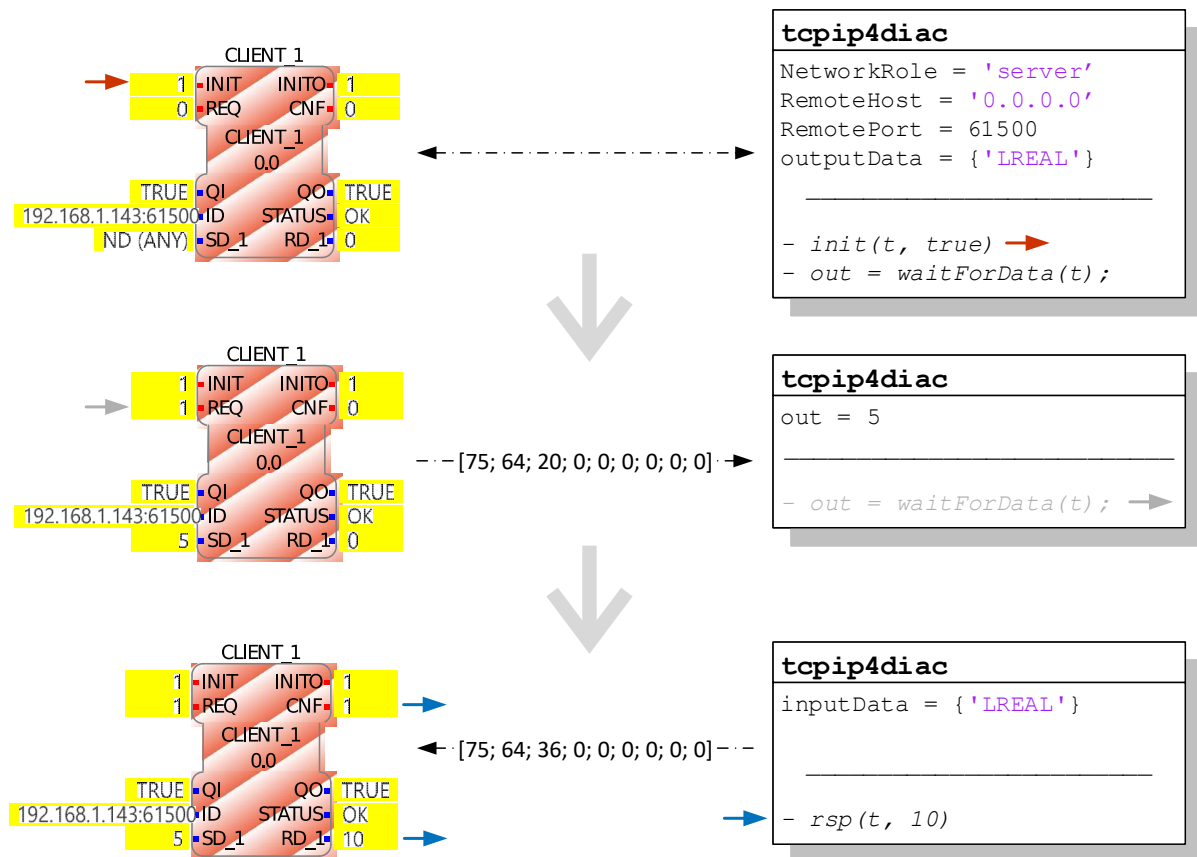
Figure 6.5: *Visualization of a communication process between a* `tcpip4diac` *server in Matlab® and a* `CLIENT_1` *FB on FORTE.*

After converting it to the equivalent FORTE binary format, the object sends the `LREAL` variable to the `CLIENT_1` FB and triggers a `CNF` output event.

As demonstrated with figures 6.4 and 6.5, the `tcpip4diac` class is very flexible. In most co-simulation cases, it makes more sense to use the former connection scenario. It is slightly more event driven, and thus more flexible. Nevertheless, the latter scenario can also be the better choice in some cases. It is also possible to combine both scenarios in one co-simulation, as is shown in section 6.3. In the following graphical representations of the validation co-simulations, only the 4diac side of the communication is shown. With the information provided in this section, the Matlab® side can be deduced[i].

6.3. 4diac/Matlab® co-simulations

For a validation of the function block libraries discussed in sections 5.1 and 5.2, 4diac/Matlab® co-simulations were set up and run using the `tcpip4diac` communication library. They are described in the following subsections.

---

[i]The Matlab® co-simulation source code is also included in the "PVTControllerLib" function block library.
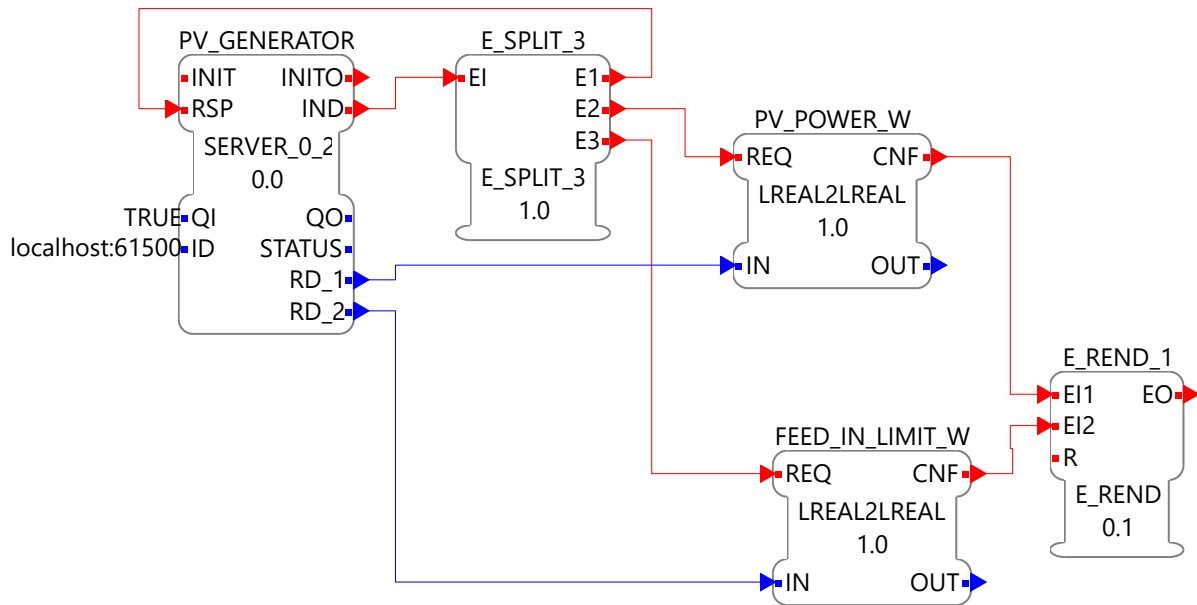
Figure 6.6: *Set-up of the `SERVER` SIFB that receives the PV power and the feed-in limit from Matlab® in the PVprog co-simulation.*

6.3.1. IEC 61499 PVprog co-simulation with Matlab®

On the Matlab® side of the co-simulation, the original Matlab® implementation's source code[i] [1] was used as a basis. A function, `pvprog4diac0`, and a corresponding script, `run_pvprog4diac0`, were created, which share almost the exact same syntax and use precisely the same data as their original counterparts. The difference is that the forecasting and optimization functions are replaced with `tcpip4diac` sockets that communicate with an IEC 61499 application composed of the PVprog function block library. A total of four `tcpip4diac` objects are instantiated: A client that sends the PV data, one for the load, another for the time stamp and a server that accepts the battery's set charging power and sends the updated SoC to the battery model function block. An excerpt of the 4diac application showing the configuration used to receive the PV power from Matlab® is depicted in figure 6.6. To enable the simulation of different feed-in limits, the `PV_GENERATOR` FB has a second output for the absolute feed-in limit in W. Because the only information required by the `tcpip4diac` object in Matlab® concerns the success of data transmission, the `IND` output event of the `PV_GENERATOR` FB is re-routed directly to its `RSP` input. The exact same set-up is used for communicating the load and grid supply limit, with `LOAD_METER SERVER` function block's `ID` set to `localhost:61501`. For receipt of the time stamps, a similar configuration as shown in figure 6.7 is implemented. A `double` vector in the `datevec` format is received as a `DATE_AND_TIME` output, which is converted into a DOY and a TD for the PVprog sequence. (see section 5.1.12). Using `SERVER` function blocks in 4diac, the communication links for the PV

---

[i]Available for download as open source at `https://pvspeicher.htw-berlin.de/veroeffentlichungen/daten/pvprog/`
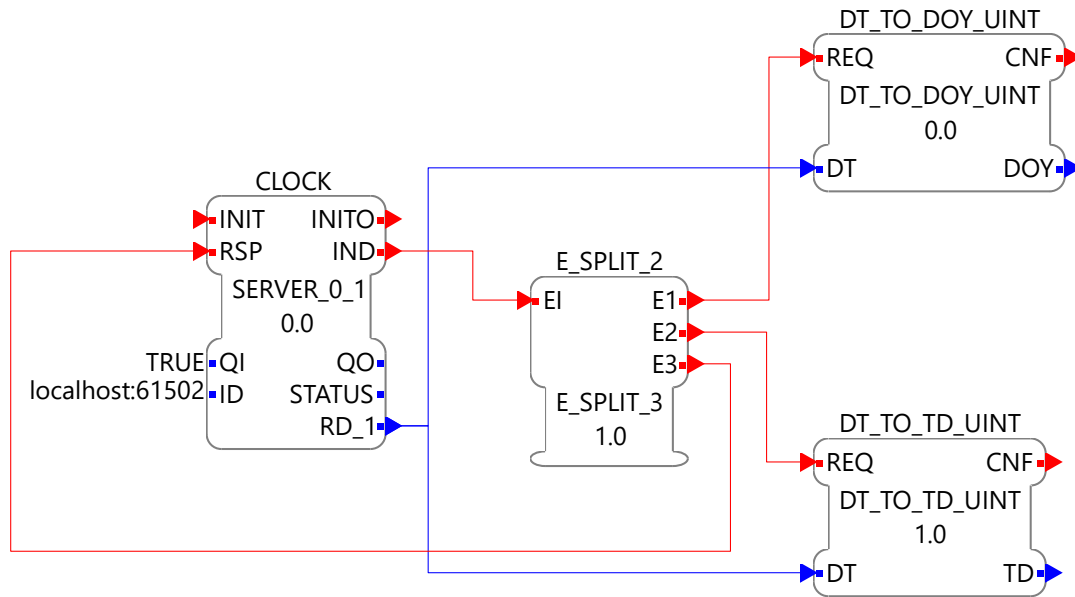
Figure 6.7: *Set-up of the SERVER SIFB that receives the time stamp from Matlab® in the PVprog co-simulation.*

generator, the load and the time stamp implement the sequence illustrated in figure 6.4. An excerpt of the application that illustrates the communication between the battery and the PVprog subapplication is depicted in figure 6.8. Only the event and data connections that are relevant to the communication with the battery are shown. An excluded connection is signified with an arrow head pointing into the input or out of the output, respectively. The internal network of the PVprog_SubApp subapplication is very similar to that of the FB_PVPROG_00 CFB introduced in section 5.1.11 and shall not be elaborated any further (Refer to section 3.2 for details on subapplications). A CLIENT_1 FB represents the battery, thus implementing the communication sequence illustrated in figure 6.5. This is done in order to combine the independent tasks of
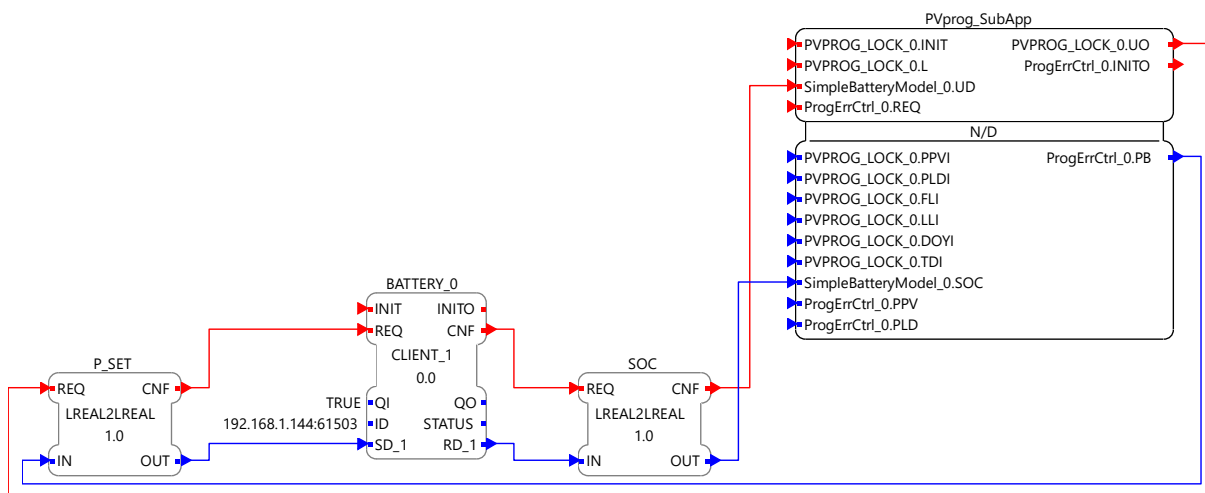
Figure 6.8: *Set-up of the CLIENT SIFB that communicates the battery data between Matlab® and the PVprog subapplication in 4diac within the PVprog co-simulation.*

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences
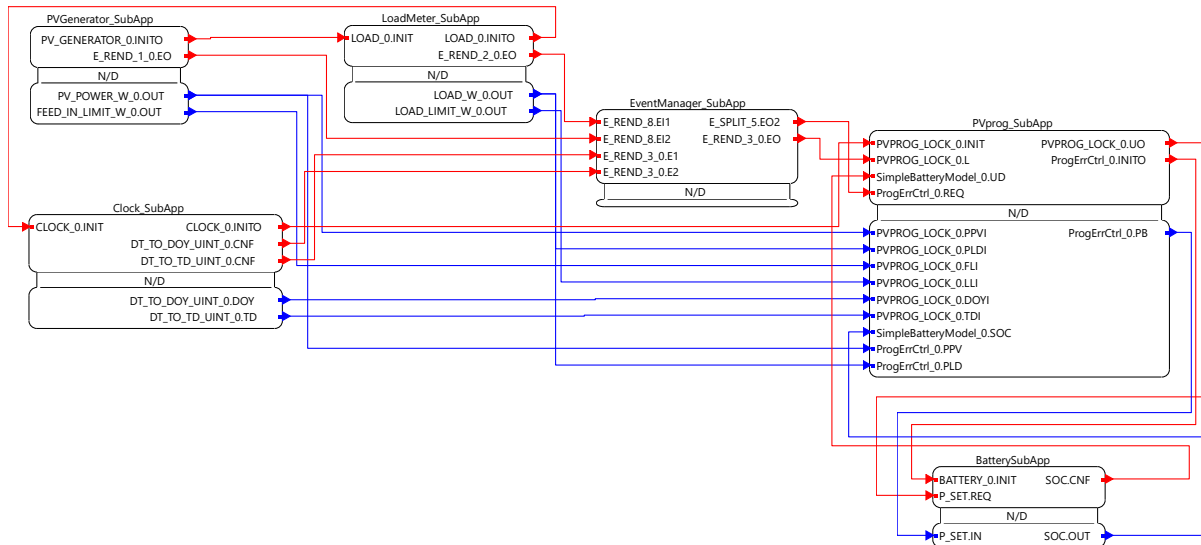
Figure 6.9: *PVprog application in 4diac used in a co-simulation with Matlab®.*

charge optimization and battery model updating into a single CSIFB. Whenever the `ProgErrCtrl_0.REQ` event is triggered, the error control algorithm kicks in, issuing an updated set power to the battery in the form of a request. In the Matlab® model, the `tcpip4diac` server receives the data and passes them on to a battery simulation, which in turn passes its new SoC back to the server socket. This causes an update of the battery model on FORTE. The complete 4diac application is presented in figure 6.9. Each of the communication set-ups illustrated in figures 6.6 to 6.8 is grouped within a subapplication. An additional `EventManager_SubApp` performs rendezvous operations to combine several events arriving from the `SERVER` CSIFBs into two input events for the PVprog optimization and error control, respectively. The power flows resulting from
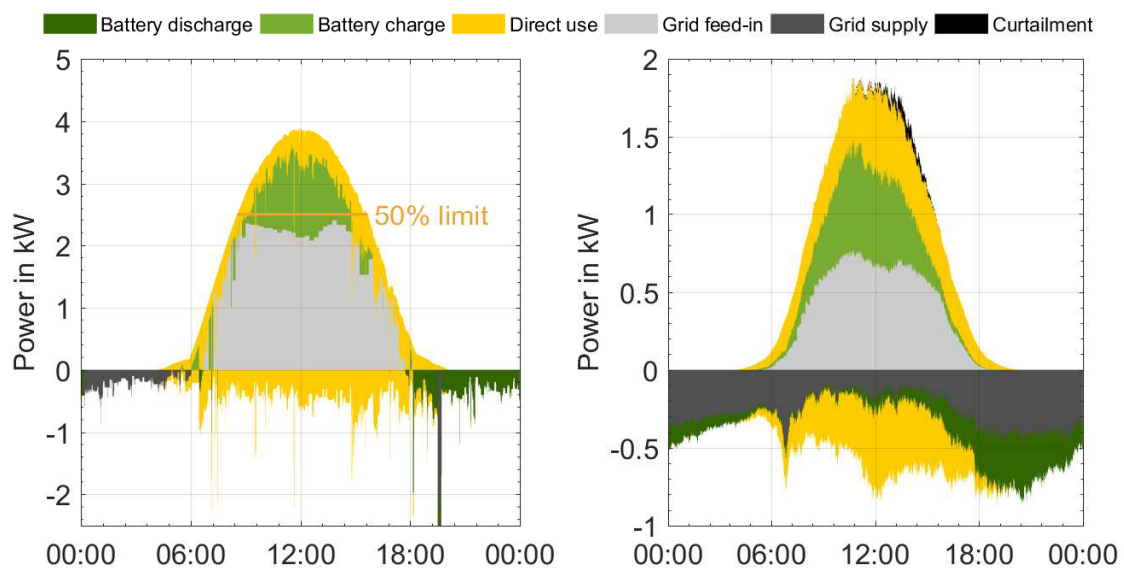
Figure 6.10: *Results of a PVprog 4diac/Matlab® co-simulation: Power flows on a sunny day (left) and average daily power flows of the year (right). Nominal PV power: 5 kWp, usable battery capacity: 5 kWh, annual electricity consumption: $5,000$ kWh.*

Table 6.2: *Comparison of the simulation results between the original Matlab® version [1], the IEC 61499 implementation of the PVprog algorithm and an IEC 61499 implementation with combined PV and load peak shaving co-simulated with Matlab®.*

	Unit	Matlab®	IEC 61499	Difference
Degree of self-sufficiency	%	52.6	52.7	0.1
Curtailment losses	%	0.9	1.4	0.5
PV generation	kWh	5,020.4	5,020.4	0
Electricity consumption	kWh	5,010.1	5,010.1	0
Direct use	kWh	1,497.6	1,497.6	0
Battery charge	kWh	1,353.3	1,363.9	10.6
Battery discharge	kWh	1,136	1,144.9	8.9
Grid feed-in	kWh	2,123.4	2,090.8	−32.6
Grid supply	kWh	2,376.5	2,367.6	−8.9
Curtailment	kWh	46.1	68	21.9

the co-simulation are depicted in figure 6.10. They coincide very well with those of the original Matlab® implementation (see figure 4.1 (right) for a comparison with the sunny day in figure 6.10, left). A more detailed comparison of the simulation results is listed in table 6.2. The discrepancies can be regarded as minor. They are attributable to the fact that the IEC 61499 implementation of the battery optimization iterates through slightly more feed-in limitations as a result of the slightly different optimization approach (see section 4.5). Thus, it can also be assumed that the differences are purely statistical. Figure 6.11 depicts power flows of a co-simulated evening using forecast-based load peak shaving. The grid supply limit was set to 500 W. To ensure that a full charge is
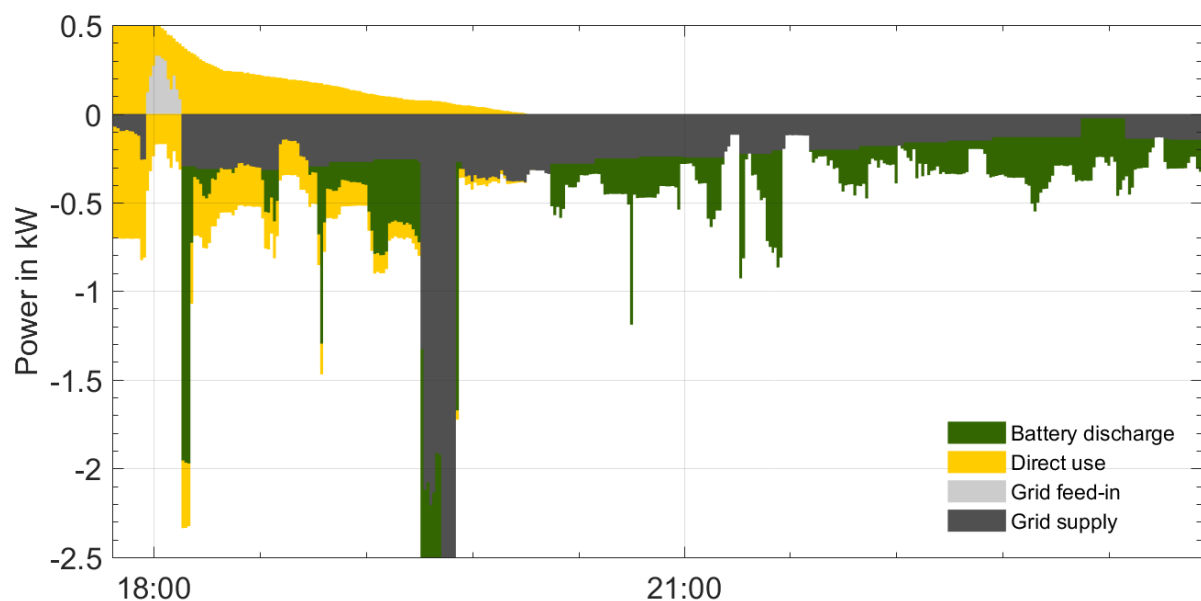
Figure 6.11: *Results of a PVprog 4diac/Matlab® co-simulation: Forecast-based load peak shaving. Nominal PV power:* 5 kWp, *usable battery capacity:* 2.4 kWh, *annual electricity consumption:* 5,000 kWh.

not enough to completely cover the demand at night, the usable battery capacity was reduced to 2.4 kWh. Instead of being emptied immediately, the discharge is spread across the whole evening to ensure that the load peaks purchased from the grid are reduced throughout the entire evening. In this simulation, the dynamic grid supply limit is usually about half of the set value of 500 W. The large load peak shortly after 7 PM temporarily increases it, thus preventing battery discharge entirely. Nevertheless, the high limit is corrected quickly enough to prevent the next load peak that would have exceeded the set limit. The fact that the large load peak exceeds the set limit of 500 W is due to inverter constraints. For a household, such as the one simulated in this thesis, a limitation of the load is unnecessary. However, the operational strategy could be of good use in an industrial application, in which penalties may be incurred for load peaks.

6.3.2. IEC 61499 PV curtailment co-simulation with Matlab®

The 4diac/Matlab® co-simulation of the power curtailment control was performed using 1 s resolved PV power and load data [21]. A 4diac application for the simulation of a proportional controller (henceforth referred to as "Controller A") is illustrated in figure 6.12. In Matlab®, a `tcpip4diac` client communicates the PV power and load to a `SERVER` function block on FORTE. For this simulation, a time stamp is not required, but the data should have a resolution of at least 1 s for adequate results.
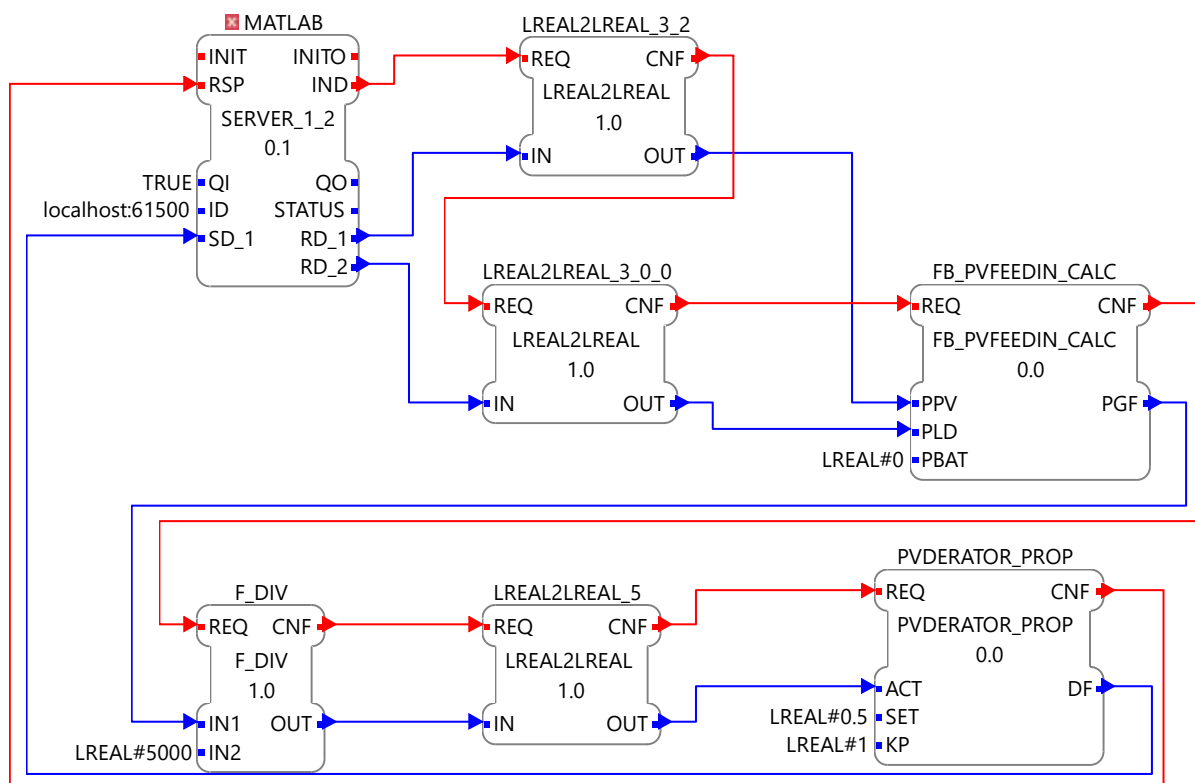
Figure 6.12: *IEC 61499 application set up for co-simulation of PV curtailment using a proportional controller with Matlab®.*
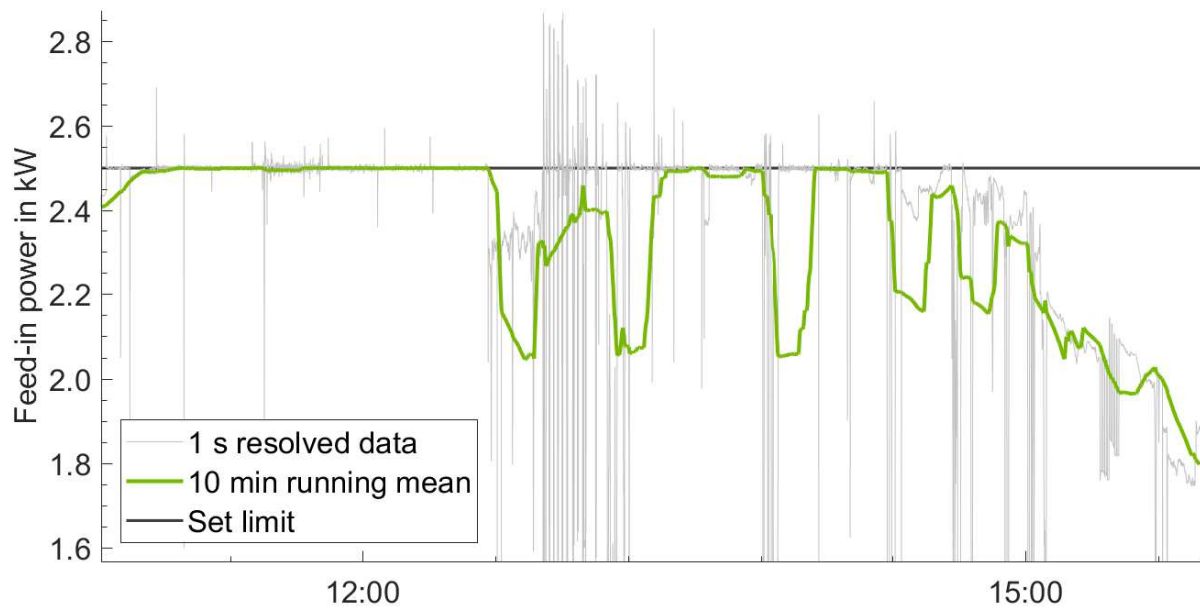
Figure 6.13: *Results of a PVprog 4diac/Matlab® co-simulation: Grid feed-in power after curtailment using a P controller (Controller A) on a sunny day.*

Only one socket on each side is used in order to reduce communication overhead. In a real application, the PV power and load would likely each require their own CSIFB unless the data are transferred via a centralized logger. The grid feed-in power is determined according to equation 5.6 using the FB_PVFEEDIN_CALC FB in the FORTE application. Its output is normalized to P_{STC} and passed on to the PVDERATOR_PROP function block discussed in section 5.2.1, with its coefficient set to 1. The derating factor is delegated back to Matlab®, where the subsequent PV power is curtailed accordingly and sent to FORTE in the next time step.

Figure 6.13 visualizes the simulation results on a sunny day. While it is impossible to perfectly curtail the feed-in power in such a way that the set value is never exceeded using a P controller, the 4diac application achieves good results regarding the 10 min running average, which is the reference value for the German KfW programme [16]. As is illustrated in figure 6.13, the 10 min running average of the feed-in power sometimes falls below the 50 % limit (2,500 kW in this instance) when sudden cloud coverage or load peaks occur. In these cases, temporarily decreasing the derating factor may result in a slight reduction of the curtailment losses. In an attempt to achieve this, the PVDERATOR_NMIN_MEAN was created (see section 5.2.2). Because the function block already implements the computation and normalization of the feed-in power, its implementation in an IEC 61499 application (see figure 6.14) is simpler than that of its proportional counterpart. In this section, it shall be referred to as "Controller B". The three PID coefficients were tuned automatically using the following empirical method:

i) Starting with each coefficient set to zero, K_{p} is increased slightly and a simulation of a few selected days is run after each increment. This is continued until the 10 min average of the simulated feed-in power exceeds the set value by a certain

tolerance. Every time the stopping criteria is met, the last increase is undone and the increment is reduced by half until the reduction by a certain threshold cannot prevent the stopping criteria from being met.

ii) Then, the same is done for K_i (with negative increments). Here, the stopping criteria is a failure to increase the total energy fed into the grid.

iii) Upon completing the incrementing of K_i, the iteration returns to step (i).

iv) When both K_p and K_i can no longer be incremented without improving the simulation results, the K_d iteration begins, returning to step (i) upon completion.

v) The iteration is finished when neither of the three coefficients can be incremented to improve the result.

vi) Finally, the saved results of every iteration step are analysed and the coefficients that resulted in the highest feed-in energy are returned.

For the simulations, an excerpt of the same data that were used for the 4diac/Matlab® co-simulations [21] was used with a PV power generation profile and a corresponding load profile[i]. For increased performance[ii], the `PVDERATOR_NMIN_MEAN` FB was replicated in Matlab®. The result of the curtailment using Controller B in a 4diac/Matlab® co-simulation is compared to that of the previous simulation using Controller A with the same data in figure 6.15. There are only few differences, and a difference between the momentary `feed-in` power flows is barely recognizable most of the time. Most notably, the feed-in power is increased at times when the 10 min running average is below the set limit (e.g., around 1 PM).

---

[i]Due to the many fluctuations of a load profile, using a PV generation profile without a load profile in the tuning process returns completely different coefficients.

[ii]The overhead of TCP/IP communication would result in extremely long durations for the hundreds of simulations performed in the iteration process.
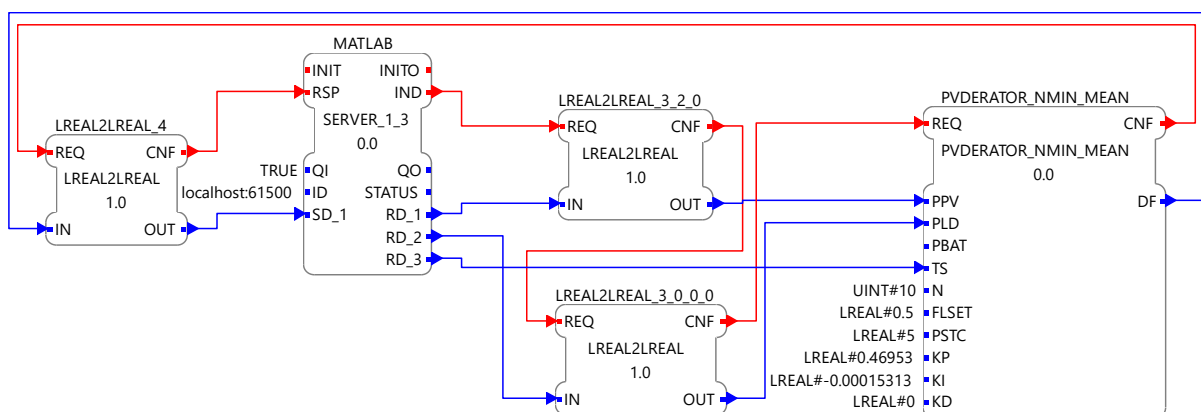
Figure 6.14: *IEC 61499 application set up for co-simulation of PV curtailment using a PID controller (Controller B) with the 10 min mean as the set value with Matlab®.*
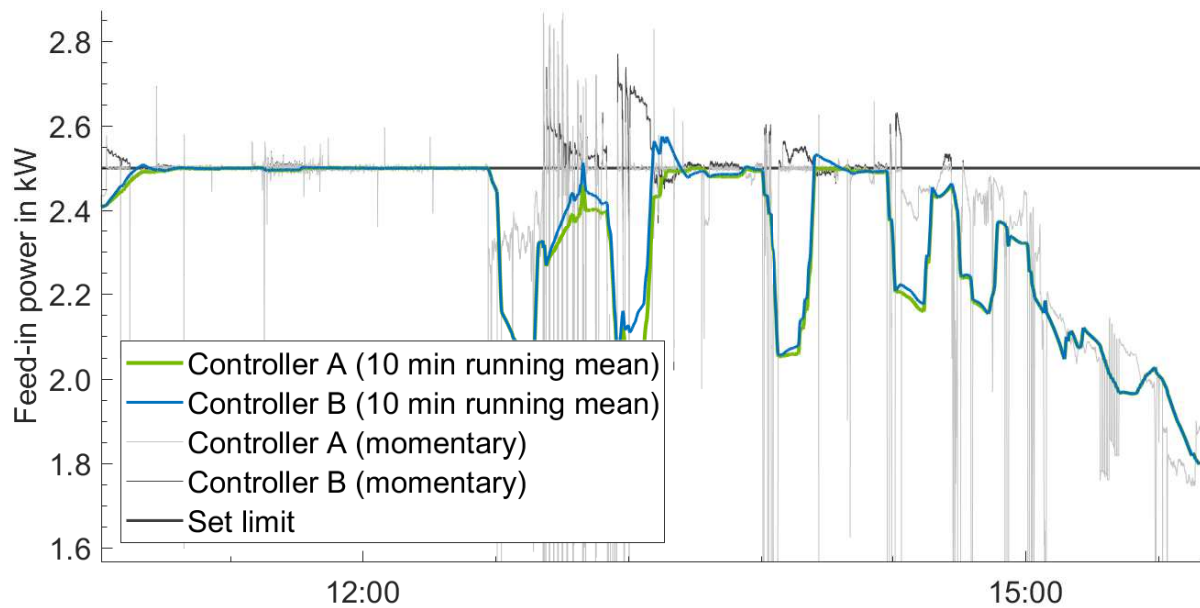
Figure 6.15: *Comparison of the grid feed-in power after curtailment using a PID controller with regard to the 10 min running average (Controller B) on a sunny day with that after curtailment using a P controller with regard to the momentary value (Controller A). Temporal resolution of the input data: 1 s.*

However, with Controller B, slight over-shots occur when the running mean exceeds the set limit. Nevertheless, this appears to be common practise, even without fluctuations caused by clouds or the load [22, p. 106], and overall, the set limit of the running average is held sufficiently. Figure 6.16 compares the two controllers in a co-simulation using 1 min resolved input data.
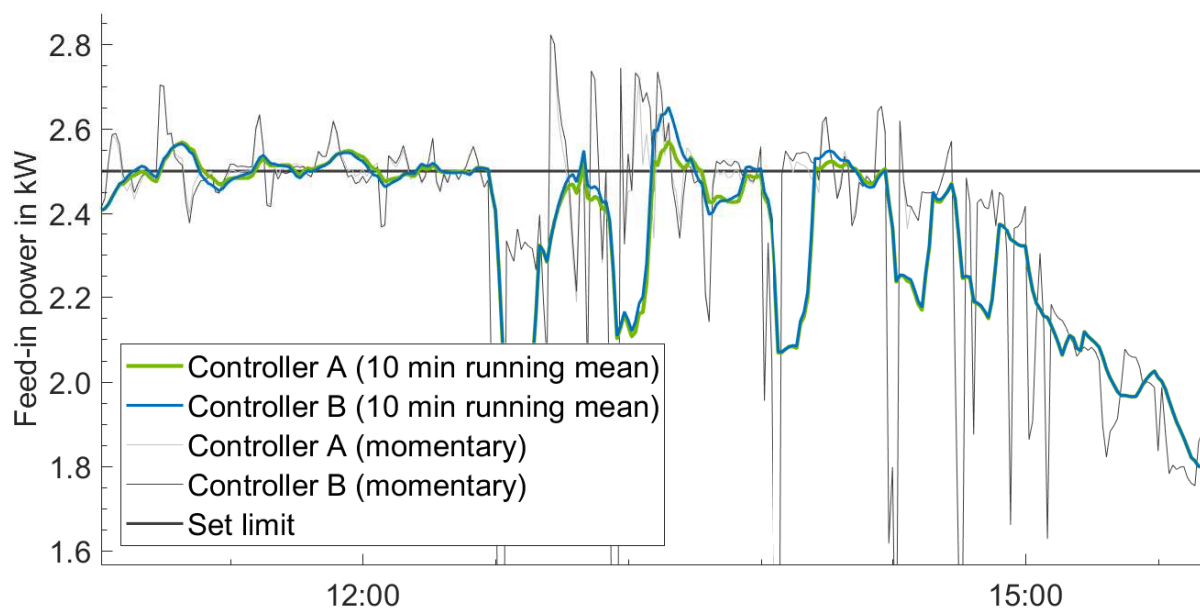
Figure 6.16: *Comparison of the grid feed-in power after curtailment using a PID controller with regard to the 10 min running average (Controller B) on a sunny day with that after curtailment using a P controller with regard to the momentary value (Controller A). Temporal resolution of the input data: 1 min.*

There is barely any difference between the two results, and in both cases, the 10 min running average exceeds the set limit. The result demonstrates that 1 s resolved input data is necessary to validate the controller's serviceability in a real application. It does, however, appear that the 10 min running mean averages around the set limit. Thus, using 1 min resolved input data should suffice for a rough estimate of the total annual curtailment losses in a system simulation.

6.3.3. IEC 61499 combined PVprog and PV curtailment co-simulation with Matlab®

The previous PVprog simulations described in section 6.3.1 assume the use of a perfect controller that curtails with regard to the momentary feed-in power as a set value. Any PV power surpluses that exceed the limit are cut off immediately, with no control error. As demonstrated in section 6.3.2, this is not the case for a real system. Thus, the curtailment and PVprog subapplications were combined into a single IEC 61499 application and simulated using the same 1 s resolved PV power and load data [21] that were used for the previous PV curtailment simulations in section 6.3.2. For deration, the PVDERATOR_NMIN_MEAN function block is used.

The application network shares many similarities with those of the respective previous PVprog and PV curtailment applications. The differences are described as follows: Instead of looping its IND event output straight to the RSP input (cf. figure 6.6), the PV_GENERATOR SERVER CSIFB takes the derating factor as an input (see figure 6.17). The event that triggers the response comes from the PVDERATOR_NMIN_MEAN FB. On the one hand, the PV power is sent directly to the deration function block (cf. figure 6.14). On the other hand, the df from the previous call to the deration FB is used to determine what the PV power would be prior to curtailment. This is done by the FB_PDER_TO_P CFB illustrated in figure 6.18, which includes a selector FB to prevent divisions by zero. Along with the load, for which the set-up has no differences compared with the previous
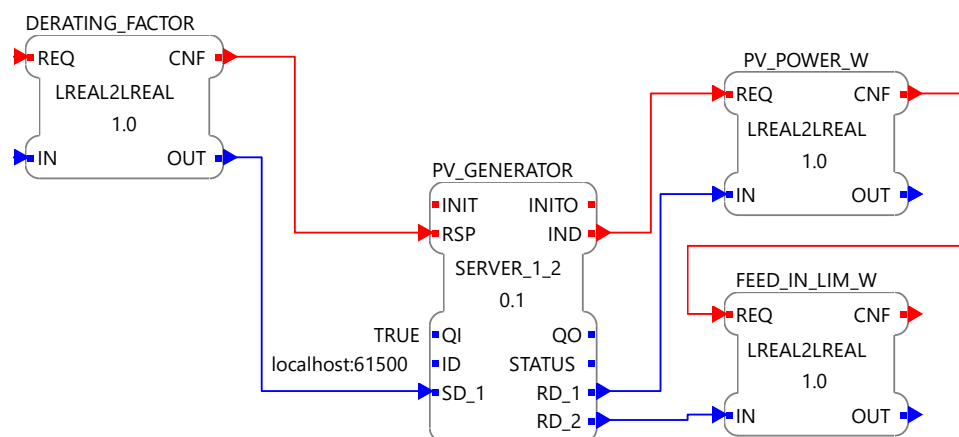
Figure 6.17: *Set-up of the SERVER CSIFB that receives the PV power and the feed-in limit from Matlab® in the combined PVprog and PV curtailment co-simulation.*
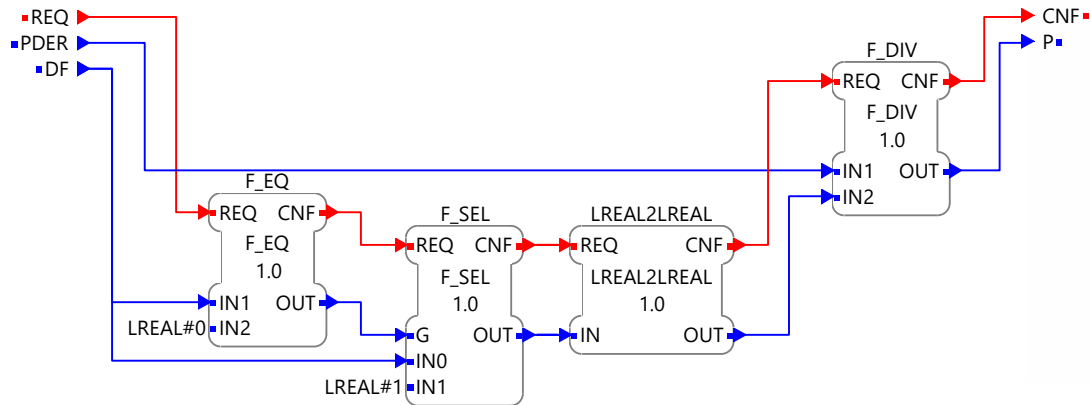
Figure 6.18: *Composite network of the* `FB_PDER_TO_P` *CFB.*

PVprog application, the PV power prior to curtailment is sent to a subapplication that summarizes the data into 1 min averages (see figure 6.19) before being passed to the PVprog subapplication. The function blocks used for this purpose are discussed in section 5.1.12. Additionally, the time stamp to DOY and TD conversion is removed from the `Clock_SubApp` (cf. figure 6.7) and appended to the 1 min mean conversion. For communication with the battery, the set-up is almost exactly the same as shown in figure 6.8. In addition to the SoC, however, the power that was used to charge or discharge the battery P_{bat} is output (see figure 6.20). P_{bat} is sent to the `PVDERATOR_NMIN_MEAN` FB, where it is used to determine the grid feed-in power for the purpose of adjusting df. As there are now two CSIFBs that send a response to Matlab®, the communication set-up on the Matlab® side had to be adjusted accordingly.
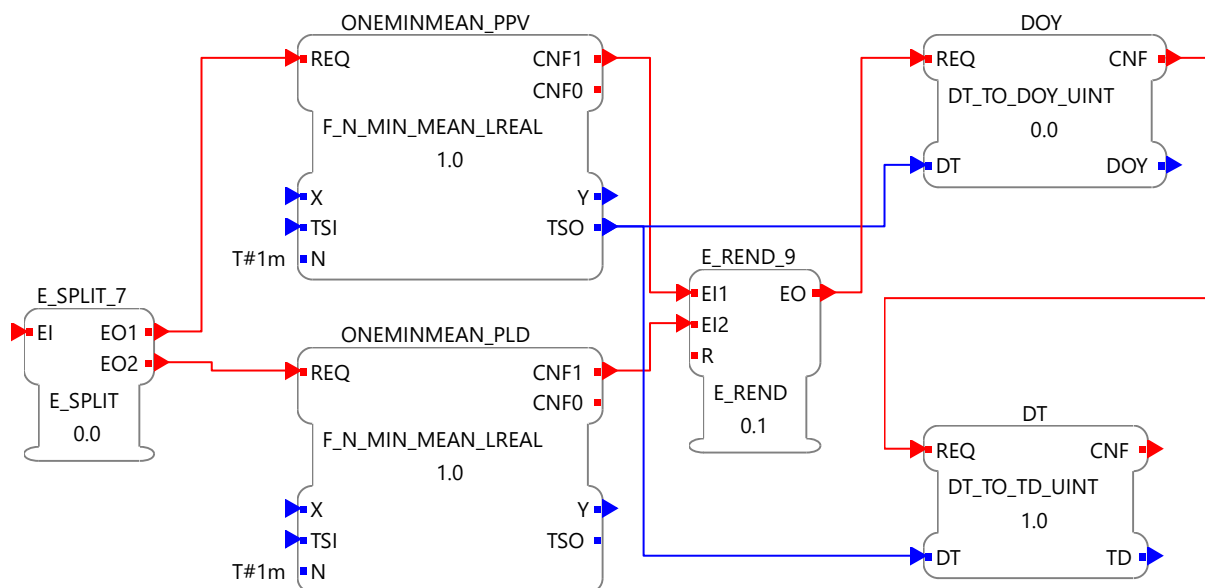
Figure 6.19: *Subapplication used for computing the* 1 min *means of the PV power and load and converting the time stamp to* `DOY` *and* `TD` *integers*.
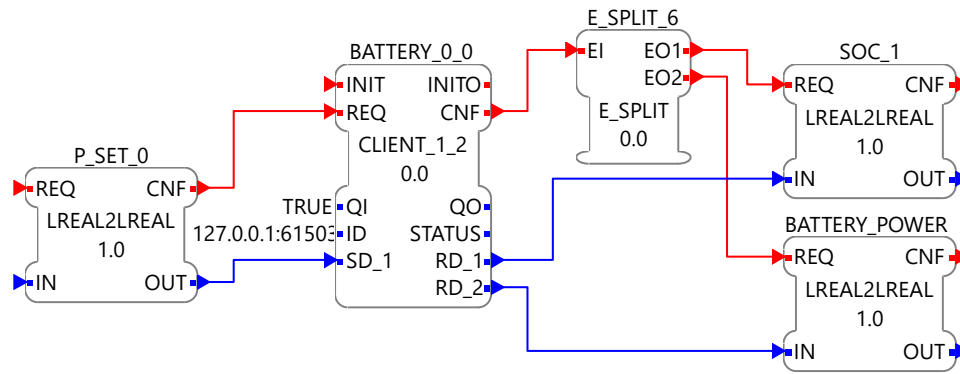
Figure 6.20: *Set-up of the* `CLIENT` *SIFB that communicates the battery data between Matlab®
and the PVprog subapplication in 4diac within the combined PVprog and PV
curtailment co-simulation.*

The sequence of a simulated time step is as follows:

i) *Matlab®*: Compute the curtailed PV power with df received in the previous time
step and send it to 4diac using the `reqNorsp()` method. Like `req()`, a request is
sent, but it allows to execute further lines of code before waiting for a response
from FORTE.

ii) *Matlab®*: Send the load and time stamp using the `req()` method.

iii) *Matlab®*: Call `waitForData()` on the battery server socket and await a response
from FORTE.

iv) *FORTE*: Perform the PVprog algorithm on the received data and send the set
power to Matlab®. The PV power prior to curtailment is computed using df from
the last time step.

v) *FORTE*: Wait for the arrival of P_{bat} before proceeding. This is implemented with
rendezvous FBs.

vi) *Matlab®*: Simulate the battery with the received set power and send the resulting
P_{bat} to FORTE.

vii) *Matlab®*: Call `waitForData()` on the PV generator client socket and await a
response from FORTE.

viii) *FORTE*: Compute the grid feed-in power and adjust df. Send the new df to
Matlab®, triggering the simulation of the next time step.

The co-simulation was performed with two variations of the application. In the first one
(Application A), the derated PV power is passed to the `ProgErrCtrl` function block,
and in the second one (Application B), the PV power before curtailment is used. In both
cases, the non-derated PV power is passed to the forecaster function blocks.
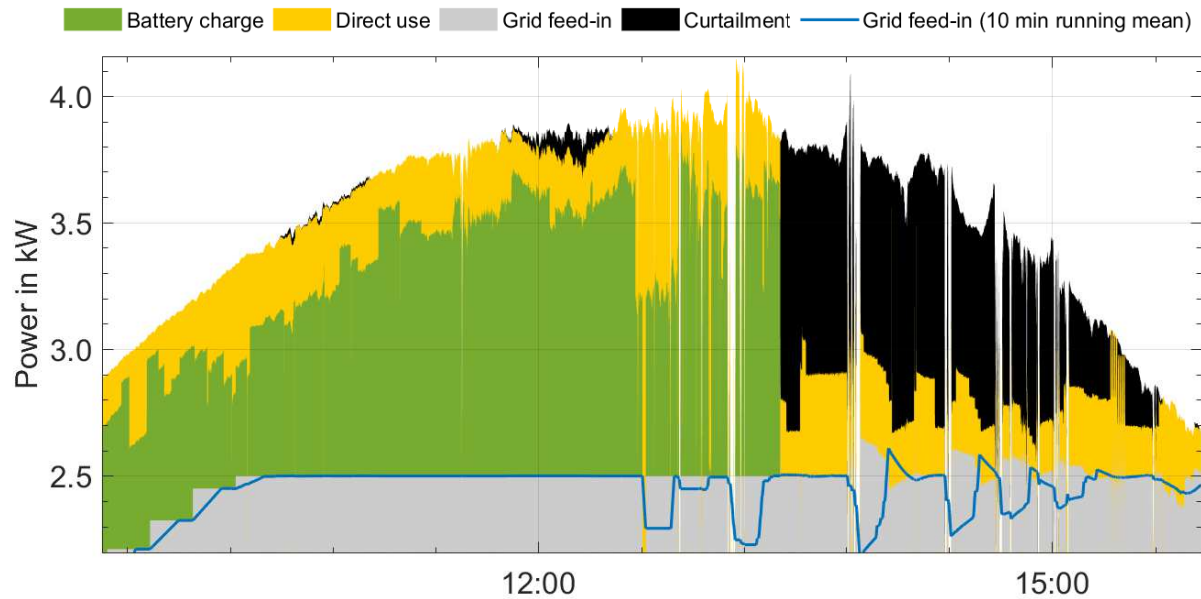
(a) *Derated PV power passed to the* `ProgErrCtrl` *FB (Application A).*
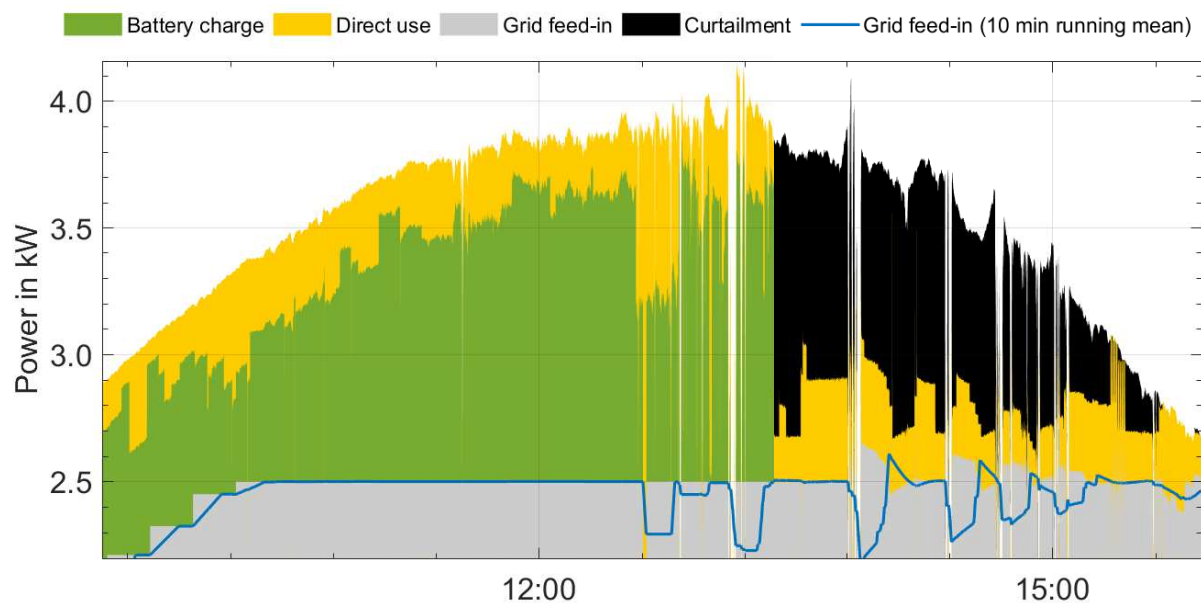
(b) *PV power before curtailment passed to the* `ProgErrCtrl` *FB (Application B).*

Figure 6.21: *Results of the combined PVprog and PV curtailment applications co-simulated with Matlab®. Nominal PV power:* 5 kWp, *usable battery capacity:* 5 kWh, *annual electricity consumption:* 5,010 kWh

An excerpt of the results of the two co-simulations is depicted in figure 6.21. To illustrate both aspects of the controllers, a day in which the battery capacity did not suffice to prevent curtailment throughout the whole day was chosen deliberately. Application A (figure 6.21a) results in the error control at times setting the battery charging power to a too low value, causing unnecessary curtailment. Only an ever so slight increase in the 10 min running average of the feed-in power is enough to get the curtailment started. The error control and deration then amplify each other and eventually level out, as can be seen in figure 6.21a around noon.

Table 6.3: *Results of the combined PVprog and PV deration control application co-simulated with Matlab®over a period of one year. Nominal PV power:* 5 kWp, *usable battery capacity:* 5 kWh, *annual electricity consumption:* 5,010 kWh.

	Unit	Value
Degree of self-sufficiency	%	55.3
Curtailment losses	%	1.2
PV generation	kWh	5,066.8
Electricity consumption	kWh	5,010
Direct use	kWh	1,496.7
Battery charge	kWh	1,518.9
Battery discharge	kWh	1,275
Grid feed-in	kWh	1,191.7
Grid supply	kWh	2,238.3
Curtailment	kWh	59.6

This issue is solved by passing the PV power before curtailment to the error control, as is done in Application B (figure 6.21b). Due to the higher power surplus being reported, the battery charging power is set to a higher value, and unnecessary curtailment is prevented. In both cases, the 10 min running average of the feed-in power rarely exceeds the set limit of 2.5 kW. The over-shots are of similar extent as those observed in figure 6.15. The results of the one-year simulation of Application B are presented in table 6.3. Due to the long simulation time of approx. 3 days, Application A was not simulated for a whole year. Although different data from those of the previous PVprog co-simulation (see section 6.3.1) were used, the results of the similarly sized system share a striking resemblance (cf. table 6.2). Overall, it can be concluded that the control applications perform well and that a conjunction of the PVprog algorithm with PV curtailment is necessary for optimal results.

7. Connection of IEC 61499 applications with Polysun®

Since version 9.0, the simulation software Polysun® can be extended with so-called "controller plugins", which - as the name implies - allow the addition of user-defined controllers called "plugin controllers". These controllers can be programmed in Matlab®, JAVA™ or Python. To enable the use of Polysun's large selection of energy system components for validation of IEC 61499 control applications, a controller plugin was created that allows co-simulations thereof. The easiest approach would be to incorporate the `tcpip4diac` class (see section 6.2) into one or more Matlab® plugin controllers. However, this would induce a significant overhead due to the communication from Polysun® to Matlab®, and then from Matlab® to FORTE. On top of that, a separate instance of Matlab® is started for each plugin controller that is used. Because multiple controllers may be required in many co-simulations, this would quickly become messy. Since Polysun® is written in JAVA™, plugin controllers programmed in JAVA™ incur the least overhead. Thus, a communication library[i] and a controller plugin[i] were written in JAVA™ using Polysun's open source PluginDevelopmentKit. They are discussed in the following subsections.

7.1. Communication Library

To enable a quick and easy extension of the plugin with new communication protocols, the communication library that implements the Open Systems Interconnection (OSI) layer design pattern was created. Aside from its usage in Polysun® plugins, it can also be implemented into any other JAVA™-based programs[ii], and can be extended for the communication with software other than 4diac. The framework is loosely based on FORTE's communication architecture [11]. A visualization of the design pattern as implemented in the library is depicted in figure 7.1.

All of the layers share the same `ICommunicationLayer` interface, which defines methods for opening and closing connections, sending and receiving data and querying the status. The layers used by the plugin controller are grouped together to form a "network stack", which is generated at runtime using a factory class (see section 7.1.5). This stack provides the interface for sending data through the network. The controller sends a request to the top layer, which processes it, and then passes the processed data on to the layer below, which in turn does the same until the network layer is reached and the binary data are sent to FORTE. In the same manner, any data received from FORTE are delegated from the network layer through to the top layer and passed to the controller. Using this design pattern, new communication protocols can easily be implemented by adding new layers. Similarly, the data processing can be adjusted

---

[i]Both the communication library and the plugin have been released as open source at `https://github.com/MrcJkb/Polysun-4diac-ControllerPlugin`.

[ii]JRE 7 or above is required.
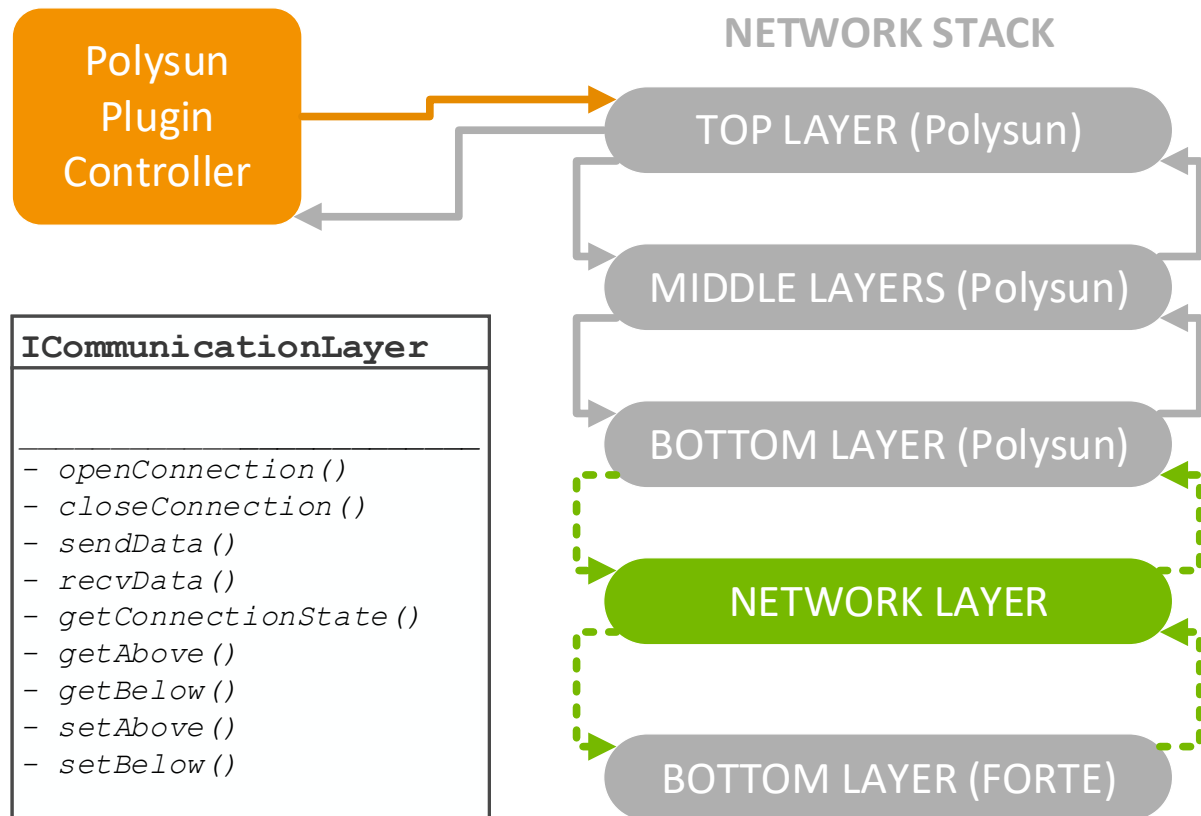
Figure 7.1: *Visualization of the communication architecture used in the Polysun®-4diac Controller Plugin and FORTE.*

for use with other programs by replacing the IEC 61499 data processing layer with a different one. A visual overview of the classes' relationships in the Polysun-4diac communication library is presented in figure 7.2.

All layers implement the `ICommunicationLayer` interface and the `IForteSocket` provides the interface for use by plugin controllers. An implementation of the factory design pattern is applied for network stack generation, which is performed by the `CommLayerParams` class. The following subsections provide an overview of the library to readers who intend to use it for the development of additional plugin controllers for communication[i].

---

[i]A complete `javadoc` documentation is available online at `https://mrcjkb.github.io/Polysun-4diac-ControllerPlugin`, and is also included in the library.
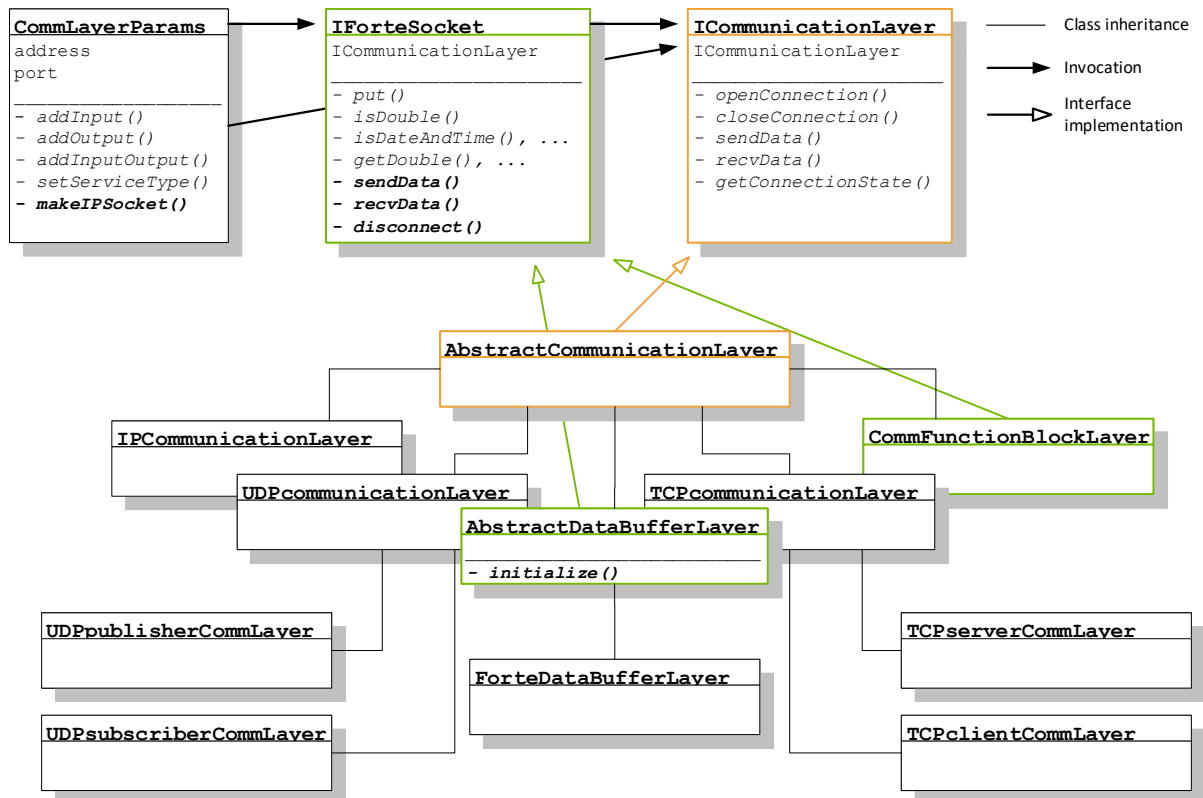
Figure 7.2: *Overview of the Polysun-4diac communication library visualizing the classes' relationships. The interface implementations are colour coded, and methods that invoke methods on another interface are marked with a bold typeset.*

7.1.1. The ICommunicationLayer interface

The most important methods of the ICommunicationLayer interface that is implemented by all OSI layers are listed as follows:

- openConnection() Layers are constructed in an uninitialized state. This method initializes the layer, and if necessary, creates additional layers below, initializing them, too. Its input parameter is a factory class. Since none of the layers are aware of which specific layers are below, the factory class is used to initialize them.

- closeConnection() Calls closeConnection() on the layer below and performs all necessary clean-up operations.

- sendData() Processes data received from the layer above and passes them to the layer below. If there is no layer below, the data are passed to the network.

- recvData() Waits for data to be received from the layer below (or the network if there is no layer below), processes them and passes them to the layer above.

- getConnectionState() Returns true if the layer or the layer below is connected to a network, false otherwise.

7.1.2. The IForteSocket interface

The `IForteSocket` interface is used by the plugin controller. It acts as a façade for the network stack and is implemented by the top layer. The additional tasks of a class that implements the `IForteSocket` interface include buffering data before they are sent to the network stack and storing data that were received. It also includes methods for ensuring that the expected data types have arrived. The most important methods are summarised as follows:

- `disconnect()` Calls `closeConnection()` on the top OSI layer. If necessary, additional clean-up is performed. There is no `connect()` method, because opening the connection is handled by the factory class.

- `put()` Adds data to the internal buffer.

- `isX()` (e.g., `isDouble()`, `isInt()`, ...) Used to determine if the next variable on the stack is of a certain data type.

- `getX()` (e.g., `getDouble()`, `getInt()`) Returns the variable on top of the stack and increments the buffer's position so that it points to the next variable on the stack.

- `sendData()` Calls the top layer's `sendData()` method.

- `recvData()` Calls the top layer's `recvData()` method.

7.1.3. Communication Layers

The OSI layers included in the library are as follows (listed from top to bottom):

- `AbstractCommunicationLayer` This abstract class implements the `ICommunicationLayer` interface and provides the default behaviour. It exists only for the maximisation of code reuse.

- `CommFunctionBlockLayer` If the connected IEC 61499 function block's number of data inputs differs from its number of outputs, this layer is used to separate processing of the data to be sent and of the data to be received. It is an exceptional layer that holds a reference to two `AbstractDataBufferLayer`s below; one for the input data and one for the output data (see figure 7.3, left). Since it is a top layer, it also implements the `IForteSocket` interface.

- `AbstractDataBufferLayer` This class provides an interface for OSI layers that process and buffer the data. Any class that extends this class can either act as the top layer (see figure 7.3, right) if the connected function block's number of data inputs is the same as its number of outputs, or as a middle layer below the

CommFunctionBlockLayer. It combines the `IForteSocket` and `AbstractCommunicationLayer` into one interface and adds an `initialize()` method used for initializing the internal data buffer.

- `ForteDataBufferLayer` An extension of the `AbstractDataBufferLayer` that is specialised on the conversion between JAVA™ objects and IEC 61499 data types. As a subclass of the `AbstractDataBufferLayer`, it can be used as a top layer or be placed as one of the `AbstractDataBufferLayer`s below the `CommFunctionBlockLayer`. This layer can be exchanged with layers that implement new communication protocols or data processing for other applications.

- `IPcommunicationLayer` This middle layer is responsible for setting up an IP connection below, depending on what is specified by the factory class.

- `TCPclientCommLayer` Bottom OSI layer for handling TCP/IP communication of a client socket. Intended for communication with an IEC 61499 `SERVER` function block (see section 3.4.2).

- `TCPserverCommLayer` Bottom OSI layer for handling TCP/IP communication of a server socket. Intended for communication with an IEC 61499 `CLIENT` function block (see section 3.4.2).

- `UDPpublisherCommLayer` Bottom OSI layer for handling UDP/IP communication of a publisher socket. Intended for sending data to an IEC 61499 `SUBSCRIBER` function block (see section 3.4.1). If this layer is at the bottom of the network stack, attempting to call `recvData()` or `readX()` will throw an `IOException`.

- `UDPsubscriberCommLayer` Bottom OSI layer for handling UDP/IP of a subscriber socket. Intended for receiving data from an IEC 61499 `PUBLISHER` function block (see section 3.4.1). If this layer is at the bottom of the network stack, attempting to call `sendData()` will throw an `IOException`.

7.1.4. Data type processing

The conversions performed by the `ForteDataBufferLayer` between JAVA™ objects and IEC 61499 data types are listed in table 7.1. Since JAVA™ only supports two integer types, `int` and `long`, and since it does not support unsigned integers, all of the IEC 61499 integer types can be mapped to the two primitives. To specify which type of variable is to be mapped, a `ForteDataType` enumeration[i] is used. Its value names are the same as the supported IEC 61499 data types, e.g., `ForteDataType.LREAL` or `ForteDataType.DATE_AND_TIME`.

---

[i]A class that represents a set of predefined constants.
[i]`DateAndTime` is part of the Polysun-4diac ControllerPlugin library (see section 7.1.6).
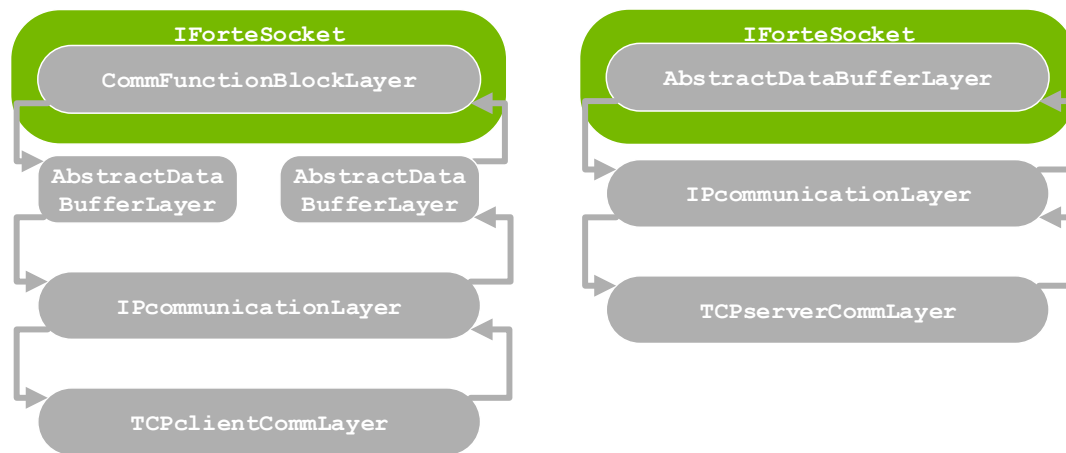
Figure 7.3: *Two exemplary scenarios for the network stack. Left: A stack in which the number of data inputs is not the same as the number of data outputs. Right: A stack in which the number of data inputs and outputs are the same.*

7.1.5. Communication Layer factory

As mentioned earlier, none of the layers know anything about each other. To create the network stack, a factory class is used, which contains all of the parameters required for opening the connection and initializing all of the individual layers. It is implemented in the class `CommLayerParams`, which extends `java.net.InetSocketAddress` and is constructed with a network address and a port number. To set the type of service, the `setServiceType()` method is called with an enumeration of the desired service as an input. The IEC 61499 data types that are to be sent and received are added after construction via the class's `addInput()` and `addOutput()` methods, whereby the order in which the inputs and outputs are added must match the order of the IEC 61499 FB's data outputs and inputs from top to bottom, respectively. To generate an `IForteSocket` object that holds the network stack, the `makeIPSocket()` factory method is used. An exemplary initialization of a connection between a JAVA™ application and a `CLIENT` function block with two data inputs and one data output on FORTE is as follows:

Table 7.1: *JAVA™ object and primitive types supported by the `ForteDataBufferLayer` and their IEC 61499 equivalents.*

JAVA™ data types	IEC 61499 equivalents
`boolean`	`BOOL`
`int`	`SINT`, `INT`, `DINT`, `USINT`, `UINT`, `UDINT`
`long`	`LINT`, `ULINT`
`float`	`REAL`
`double`	`LREAL`
`String`	`STRING`
`DateAndTime`[i], `java.util.Calendar`	`DATE_AND_TIME`

```
CommLayerParams params = new CommLayerParams("localhost", 61499);
params.setServiceType(ForteServiceType.SERVER);
params.addInput(ForteDataType.LREAL); // Adds an LREAL data input
params.addOutput(ForteDataType.LREAL, 5); // Adds an LREAL array
                                          // data output of length 5
params.addOutput(ForteDataType.BOOL); // Adds a BOOL data output
IForteSocket socket = params.makeIPSocket(); // Opens the connection.
```

To send data to FORTE, the variables are stored in the internal buffer via the `put()` method before calling `sendData()`.

```
socket.put(5.0); // Add a double to the byte buffer
socket.sendData(); // Sends the binary data to FORTE
```

Unlike Matlab®, the JAVA™ programming language enforces type safety, preventing programs from accessing memory in unexpected ways. Thus, a method for checking the correct type before attempting to access buffered data was implemented. For example the correct way to receive an `LREAL` array and a `BOOL` variable from FORTE is:

```
socket.recvData(); // Returns when the connected IEC 61499 FB has
                   // sent its data
double dVar;
boolean bVar;
if (socket.isDoubleArray()) {
    dVar = socket.getDoubleArray();
} else {
    // Handle or throw exception
    throw new PluginControllerException("Unexpected data type.");
}
if (socket.isBool()) {
    bVar = socket.getBool();
} // exception handling omitted for brevity
```

7.1.6. The DateAndTime class

Polysun® simulations do not include time stamps. Instead, an integer that represents the number of seconds since the beginning of the simulation is passed to the plugin controller's `control()` method. To add support for the IEC 61499 DATE_AND_TIME data type, the `DateAndTime` class was created. A `DateAndTime` object has the same internal representation as a DATE_AND_TIME variable in FORTE (discussed in section 6.2.1). Upon construction, the date at the beginning of the simulation is specified. This can

either be done with a set of integers or with a `java.util.Calendar` object[i], which can be used to dynamically set the beginning of the simulation according to the operating system's clock.

```
import java.util.Calendar;
// initialization with
// year, month, day, hour, minute, second, millisecond
DateAndTime dt = new DateAndTime(2017, 1, 1, 0, 0, 0, 0);
// initialization with Calendar
DateAndTime dt2 = new DateAndTime(Calendar.getInstance());
```

In the plugin controller's `control()` method, the time can be set according to the `simulationTime` input (see the Polysun® user manual [23]) and passed directly to an `IForteSocket`.

```
dt.setSimulationTimeS(simulationTime);
socket.put(dt);
```

7.2. Polysun-4diac controller plugin

The Polysun-4diac controller plugin (named `ForteActorSensorPlugin`) was developed using the open source PluginDevelopmentKit that ships with Polysun® 9.0 and above. The internal implementations of the individual plugin controllers shall not be discussed any further in this thesis. Readers who intend to use the published project for the further development of Polysun-4diac plugins or other communication plugins based on this project are advised to refer to the Polysun® user manual [23] as well as the PluginDevelopmentKit's and this project's respective `javadoc` documentations.

7.2.1. Use of 4diac plugin controllers in Polysun®

Because Polysun® is GUI based, simulations cannot be programmed freely, i.e. the user has no influence over the sequence in which the components are simulated. The exception are controllers: The order in which the controllers are invoked is equal to the order in which they are placed in the system. This aspect is of great importance for the design of Polysun®/4diac co-simulations. To provide users with as much flexibility as possible when designing co-simulations, the plugin controllers are separated into three types: Sensors for sending data to FORTE, actors for receiving data from FORTE and combinations thereof. A set of component-specific plugin controllers were developed

---

[i]Support for the more flexible `java.time.LocalDateTime` will be added when Polysun® is updated to support JAVA™ SE 8.

for the use in this project. Furthermore, three generic controllers (one of each type) were created for users without knowledge of the JAVA™ programming language who require communication with additional components. All of the plugin controllers, with the exception of the "Generic 4diac Controller" (see section 7.2.5), implement a TCP/IP client service. Therefore, their FORTE counterparts make use of `SERVER` function blocks, and must be initialized before the plugin controllers. The plugin controllers are initialized at the beginning of the simulation.

To enable the use of 4diac plugin controllers in Polysun®, the JAR file "ForteActor-SensorPlugin.jar" must be placed in Polysun's ..\plugins directory (The default for Windows is %userprofile%\Polysun\plugins). After restarting Polysun®, the controllers can be added to a system by selecting the Controller component and clicking anywhere in the system. A list of options pops up in which the "Plugin controller" must be selected. This opens a drop-down list in which the individual plugin controllers can be placed (see figure 7.4).
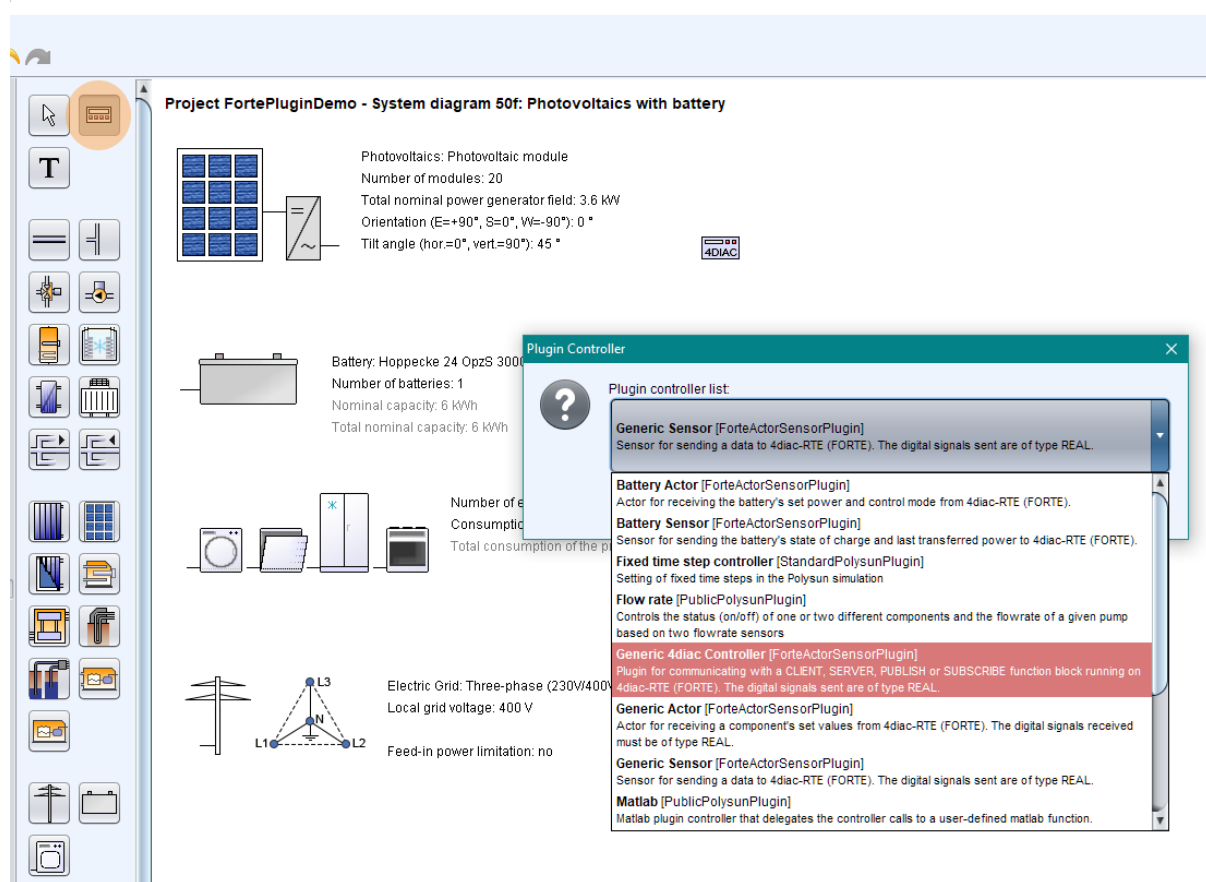
Figure 7.4: *Placement of the 4diac plugin controllers in Polysun®.*

7.2.2. Sensor plugin controllers

The sensor plugin controllers can be used for sending data to FORTE. One each was created for the battery, the photovoltaics and the electrical consumers (load), respectively. Additionally, a generic sensor controller was developed for the use with any arbitrary Polysun® component that has connectible input parameters. The GUI of the "Battery Sensor" controller is displayed in figure 7.5, in which the relevant settings and control inputs are highlighted in orange. Each sensor has the same settings. The host name and port number must be set according to the function block on the receiving end with which the plugin controller is to be connected. Additionally, each sensor is given the option to send a time stamp (the translation from Polysun's simulation time to the IEC 61499 DATE_AND_TIME format is discussed in section 7.1.6). If the option "yes" is selected, users can set the beginning of the simulation by specifying the date and time in the dd.mm.yyyy HH:MM:SS format. Finally, the controller gives the choice of whether to wait for a response or not. To take full advantage of the TCP's stability, it is recommended to wait for a response, if possible. However, this may not be desirable in certain situations (see the 4diac/Matlab® co-simulation in section 6.3.3, for example). All of the sensors and their control inputs are listed in table 7.2.

Figure 7.5: *GUI of the "Battery Sensor" plugin controller in Polysun®.*

Table 7.2: *Forte sensor plugin controllers and their control inputs.*

Name	Control inputs	Input units
Battery Sensor	State of charge	-
	Battery transfer	W
Photovoltaics Sensor	PV power output AC	W
	Maximum grid feed-in	-
Load Sensor	Electricity consumption	W
Generic Sensor	Up to five inputs	Any

The inputs of the component-specific sensors[i] must not forcibly be set to what is indicated with their names. For example, the "Battery transfer" input can be connected to any component's output as long as it has the unit W. The controller merely provides an interface as a guideline. If a sensor has more than one control input, only one must be connected; the others are optional. Only the Generic Sensor can be connected to zero inputs as long as its "Send time stamp" option is configured to "yes". For each of the component-specific plugin controllers, an IEC 61499 CSIFB counterpart was created in 4diac[ii]. The interface of the `BatterySensor` function block that connects to the Battery Sensor plugin is depicted in figure 7.6.

All of the sensor function blocks share a similar interface and act as a façade for two `SERVER` FBs - one with two data outputs coupled with the `IND` event output and one with three. Which one is used depends on the input `TSF`, which indicates whether the connected plugin controller has its "Send time stamp" option set to "yes" (`TSF = true`) or "no" (`TSF = false`). The rest of the interface is the same as that of a `SERVER` FB, except that the `RSP` input event is redundant and type-safety is enforced for the data outputs. Apart from the time stamp, all of the plugin controllers' control inputs must be connected to a component if they are used in conjunction with the provided sensor function blocks.

---

[i]Battery-, Photovoltaics- and Load Sensor.
[ii]The .fbt and C++ source files of the plugin CSIFBs are included as part of the library.
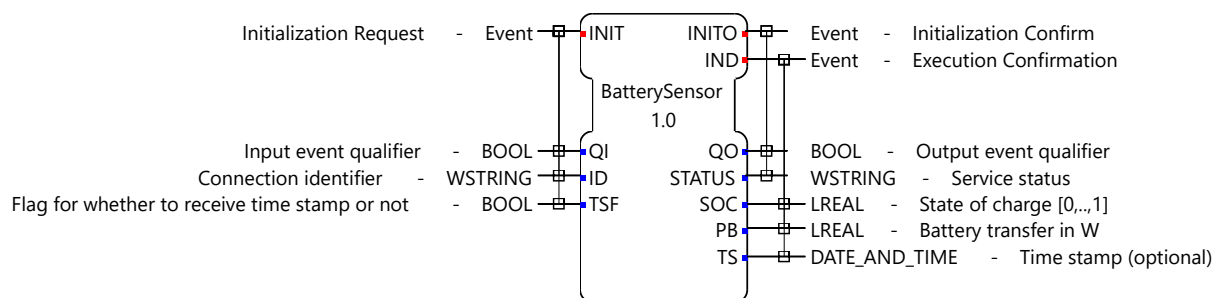
Figure 7.6: *Interface of the `BatterySensor` function block.*

7.2.3. Actor plugin controllers

The actor plugin controllers can be used to receive data from FORTE. Since the load cannot be controlled in Polysun®, actors were only created for the battery and the PV field - with the addition of a generic actor that can be connected to an arbitrary controllable component. The GUI of the "Battery Actor" is presented in figure 7.7 with the most relevant elements highlighted in orange. Since actors only receive data, the only configurations required are the address, the port number and the control outputs. A complete list of the actor plugin controllers and their control outputs can be found in table 7.3. The interface of the corresponding `BatteryActor` CSIFB is depicted in figure 7.8. It acts as a wrapper for a single `SERVER` FB with a similar interface. However, it mirrors the Polysun® sensor function blocks, enforces type-safety for the data inputs and the `IND` output event is omitted due to its redundancy.
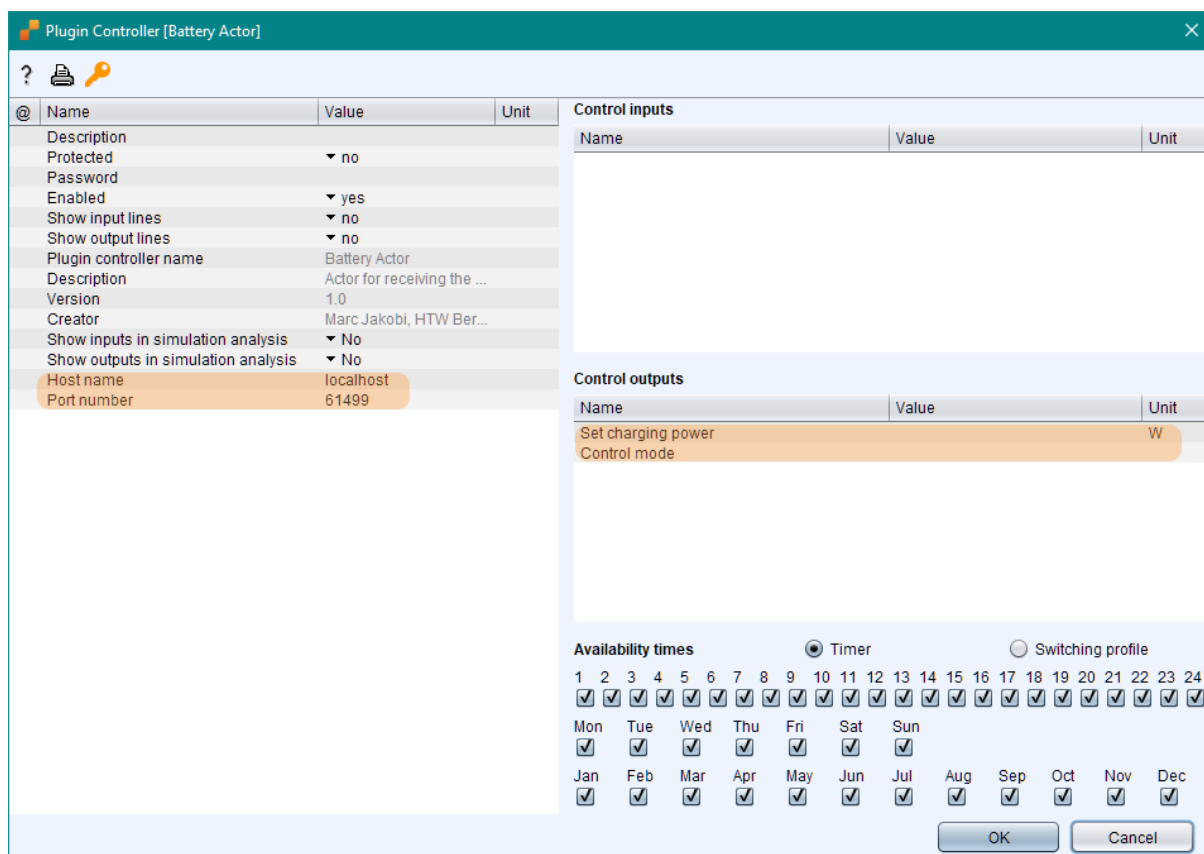
Figure 7.7: *GUI of the "Battery Actor" plugin controller in Polysun®.*

Table 7.3: *Forte actor plugin controllers and their control outputs.*

Name	Control outputs	Output units
Battery Actor	Set charging power	W
	Control mode	-
Photovoltaics Actor	Derating factor	-
Generic Actor	Up to five outputs	Any

Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences
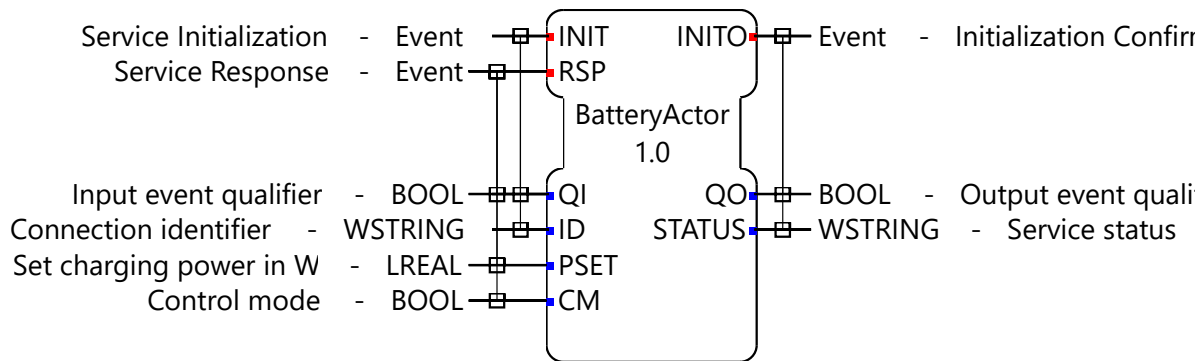
Figure 7.8: *Interface of the* `BatteryActor` *function block.*

7.2.4. SG Ready heat pump plugin controller

An additional plugin controller was created to enable co-simulations with IEC 61499 applications that use the `HeatPumpController` FB (see section 5.3). While the contact states are clearly defined in the SG Ready standard, the heat pumps' internal controllers and operation modes are not. At the time of writing this thesis, heat pumps in Polysun® do not yet support the SG-Ready interface. However, they can be controlled freely using a heating controller component, a programmable controller or a plugin controller. To enable co-simulations with IEC 61499 applications that control heat pumps according to the SG Ready standard, a plugin controller was developed that acts as an adapter to the interface. From an IEC 61499 control application's point of view, the SG Ready heat pump plugin controller is an actor. Unlike the actor plugins described in section 7.2.3, however, this one performs computations to simulate an internal heat pump controller in addition to the receipt of control signals from an IEC 61499 CSIFB.

It is used in combination with one of Polysun's built-in auxiliary heating controllers, which represents the heat pump's normal operation mode (SG Ready mode 2). The auxiliary heating controller's output signal is delegated to the SG Ready adapter, and either left as is or overridden, depending on the operation status. The heating controller is configured as if it were the main controller, but the `String`, "`NORMALMODE`" must be typed into the description box (highlighted in figure 7.9) in order for it to be recognized as the heat pump's "internal" controller and to have its output passed to the SG Ready adapter. To make this possible, an `IOutputOverridable` interface had to be added to the Polysun® PluginDevelopmentKit and incorporated into Polysun's source code. For this reason, the feature cannot be used with Polysun® 10.0. It will be made available in a future release. Effectively, the `IOutputOverridable` interface is an open source façade for Polysun's closed source controllers. Plans have been drafted for a more intuitive way of overriding controller outputs, but the implementation falls outside the scope of this thesis. The SG Ready heat pump controller's GUI is presented in figure 7.10. Since it takes a storage tank's temperature as an input signal and the heat pump's internal auxiliary heater as an output, the controller acts as both an actor and a sensor from
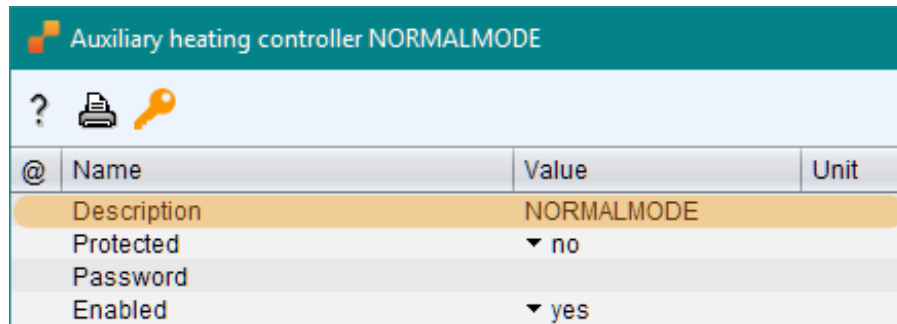
Figure 7.9: *Configuration of the auxiliary heating controller for override by the SG Ready heat pump adapter in Polysun®.*

Polysun's point of view. In addition to the usual connection settings, users can configure the SG Ready control modes 3 and 4. This is done in part by defining the upper temperature thresholds for the buffer storage at which the internal algorithm forces a normal operation no matter what input signal is received from FORTE. Furthermore, to set a hysteresis for the control modes, the amount by which the storage buffer's temperature must drop before allowing the set control mode again must be defined. Finally, the user is free to choose whether each control mode includes the use of the heat pump's internal auxiliary heater, if available. The `SGReadyHeatPumpAdapter` CSIFB's interface is pictured in figure 7.11. Like all other Polysun® plugin actors, it acts
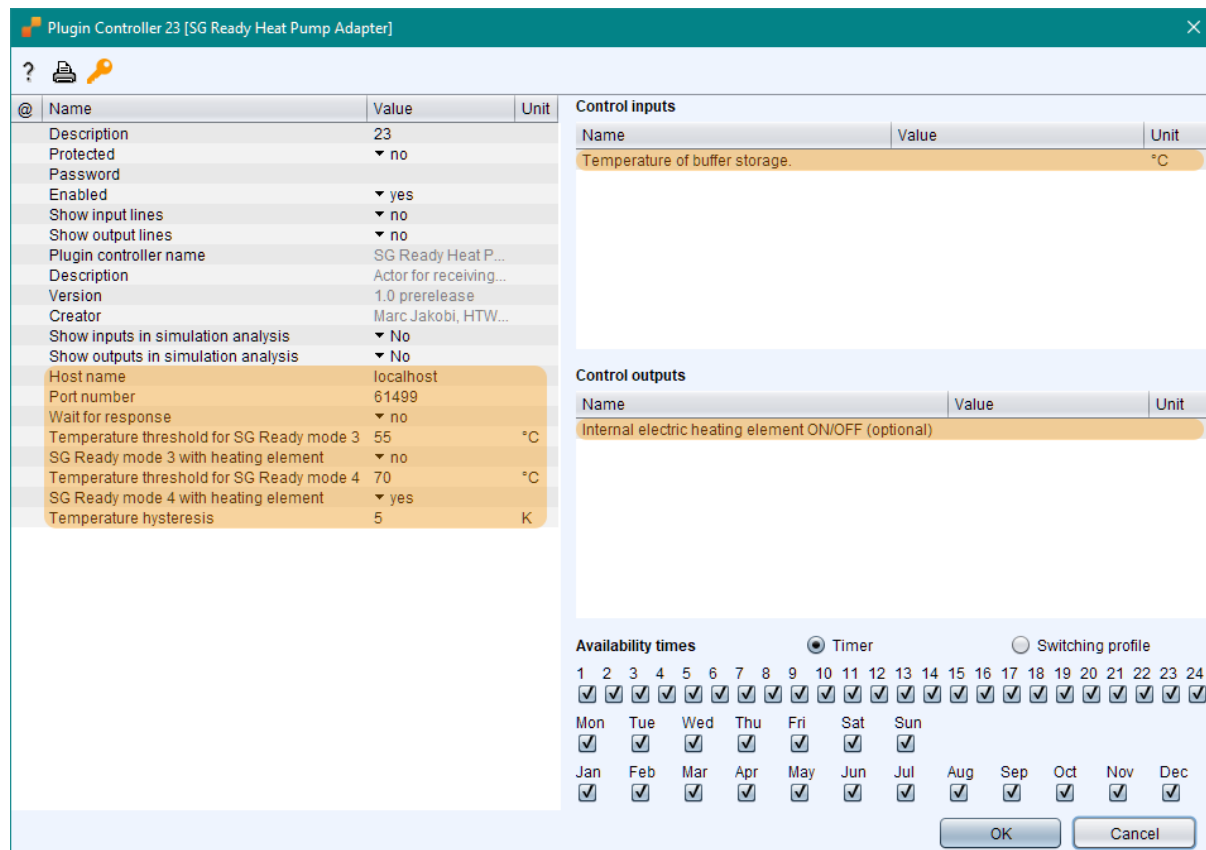
Figure 7.10: *GUI of the "SG Ready Heat Pump Adapter" plugin controller in Polysun®.*
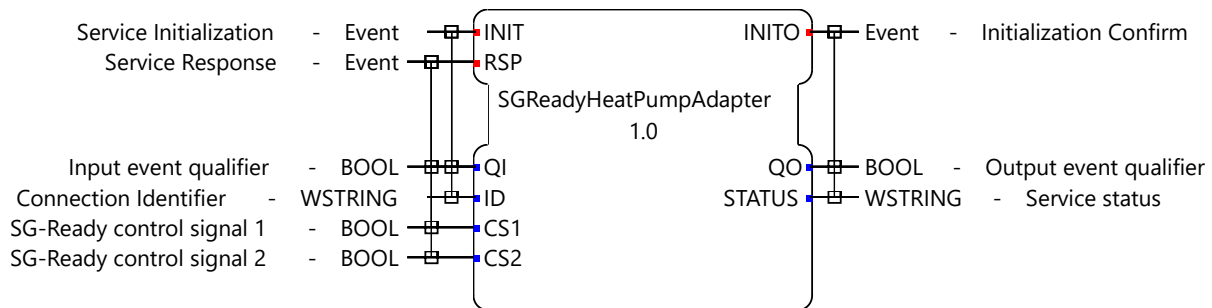
Figure 7.11: *Interface of the `SGReadyHeatPumpAdapter` function block.*

as a wrapper for a `SERVER` function block, hiding all of the unused outputs. It sends two boolean signals, which represent the on/off switches of the SG Ready interface, to the Polysun® plugin.

7.2.5. Generic 4diac plugin controllers

To enable a flexible use of the Polysun-4diac plugin controllers for users with little or no JAVA™ knowledge, three generic plugin controllers were created: An actor, a sensor and a controller that can both send data to and receive data from any IEC 61499 CSIFB. The two-way generic 4diac controller's GUI is displayed in figure 7.12.
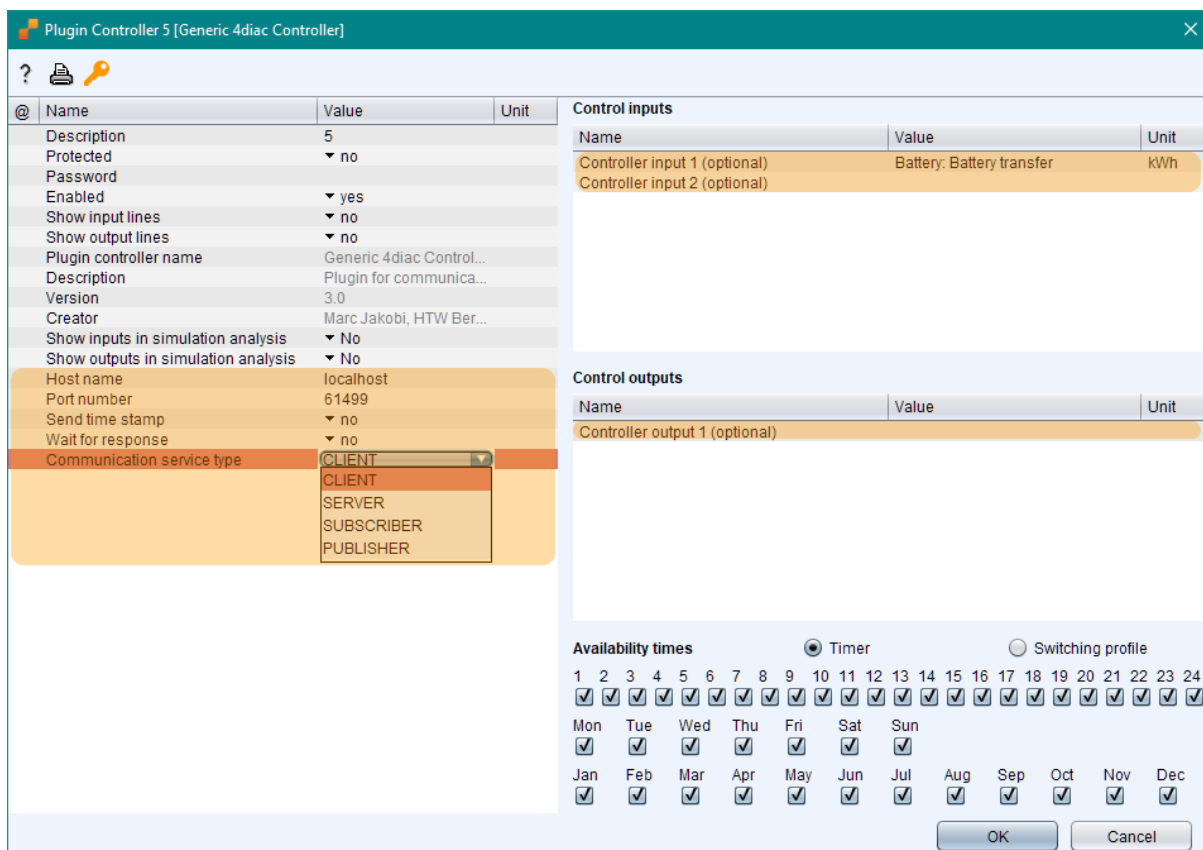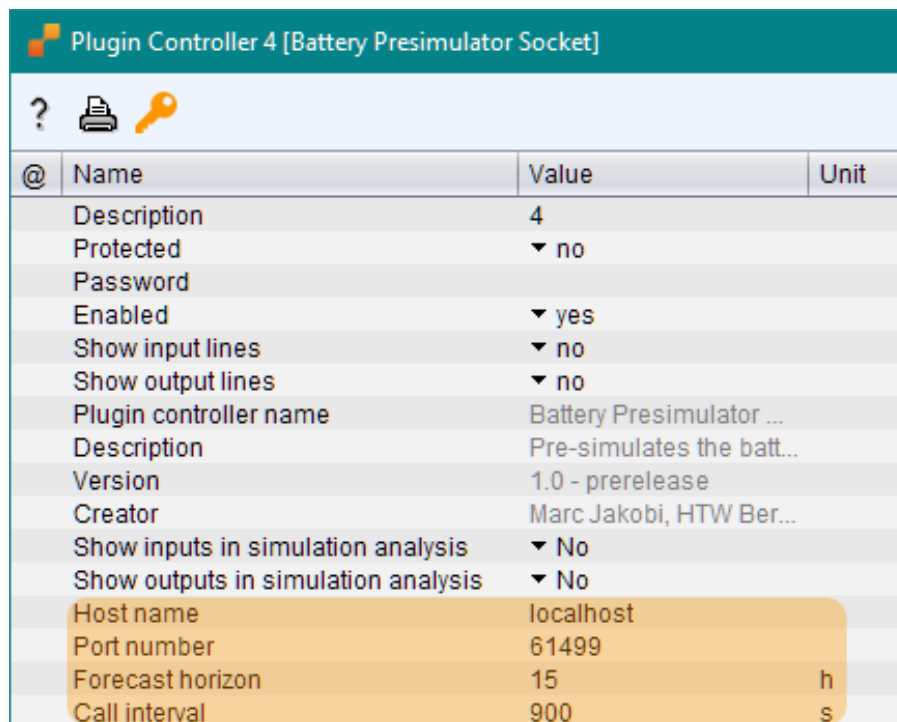
Figure 7.12: *GUI of the generic 4diac plugin controller in Polysun®.*

Up to five inputs and outputs each can be added dynamically as they are connected to the components. The four IEC 61499 basic communication service types are possible, whereby a `CLIENT` plugin controller communicates with a `SERVER` function block, etc.

7.2.6. Battery pre-simulation socket plugin

In order to demonstrate the use of the `ABatteryModel` adapter (see section 5.1.4), an `IPreSimulatable` interface was incorporated into Polysun's source code and the PluginDevelopmentKit. In its current form, the open source façade allows plugin controllers to trigger the pre-simulation of certain closed source Polysun® components (i.e. the battery) and to retrieve the results. Like the `IOutputOverridable` interface (see section 7.2.4), the `IPreSimulatable` interface is not available in Polysun® version 10.0, and will be usable with the next release. An excerpt of the plugin controller's GUI is pictured in figure 7.13. Since the plugin is technically not a controller, it has no inputs or outputs. It is simply placed in a system that has a battery. A reference to the battery (if available) is passed to the plugin at the beginning of the simulation, and the internal implementation transfers the data between the battery and the connected `PolysunBatteryModel` function block on FORTE. The only inputs required from the user are the usual connection parameters, the PVprog forecast horizon in h and the interval in s at which the pre-simulation iteration is triggered on FORTE with a `SIM` event. In the case of the PVprog algorithm, the call interval is equal to the the forecast update frequency: 15 min.

@	Name	Value	Unit
	Description	4	
	Protected	▾ no	
	Password		
	Enabled	▾ yes	
	Show input lines	▾ no	
	Show output lines	▾ no	
	Plugin controller name	Battery Presimulator ...	
	Description	Pre-simulates the batt...	
	Version	1.0 - prerelease	
	Creator	Marc Jakobi, HTW Ber...	
	Show inputs in simulation analysis	▾ No	
	Show outputs in simulation analysis	▾ No	
	Host name	localhost	
	Port number	61499	
	Forecast horizon	15	h
	Call interval	900	s

Figure 7.13: *Excerpt of the "Battery Presimulator Socket" plugin controller's GUI in Polysun®.*

The corresponding `PolysunBatteryModel` function block has a very similar interface to the `BatteryModelServer` FB mentioned in section 5.1.4, with an additional `R` input event used to end the iteration. However, its internal implementation differs. The composite network is presented in figure 7.14. Since the `BatteryOptimizer` issues the `SIM` event many times during the optimization iteration (see section 5.1.5), the Polysun® plugin has to await the data in a `while` loop. When the `BatteryOptimizer` has found an optimum, it must let the plugin know that it has finished its iteration, so that it can break out of the loop and the Polysun® simulation can continue. To do so, the `BatteryOptimizer`'s `CNF` output event can be used to trigger the `PolysunBatteryModel`'s `R` input event. The `E_TF` function block translates the `R` and `SIM` events into a `BOOL` data value, which is sent to Polysun® as an indication for whether to continue waiting or not. Similarly, the `E_PERMIT` FB only delegates the `IND` output event to the `ABatteryModel` adapter if the Polysun® plugin sends `true` back.

7.3. 4diac/Polysun® co-simulations

In the following section, the Polysun-4diac controller plugin is first tested in a PVprog and curtailment co-simulation. Then, a simple PV and heat pump system is co-simulated with the `HeatPumpController` function block and finally, the two control applications were combined and validated in further co-simulations. Pre-existing limitations that were taken into consideration beforehand include:

- In every time step, the controllers are processed first (with the exception of certain components, i.e. the PV field) followed by the components. The controllers are processed in the order in which they were placed in the GUI. It is currently impossible for a user to set the order of the elements as was done in section 6.3.3.

- In its current incarnation, Polysun® does not have the ability to read meteorological
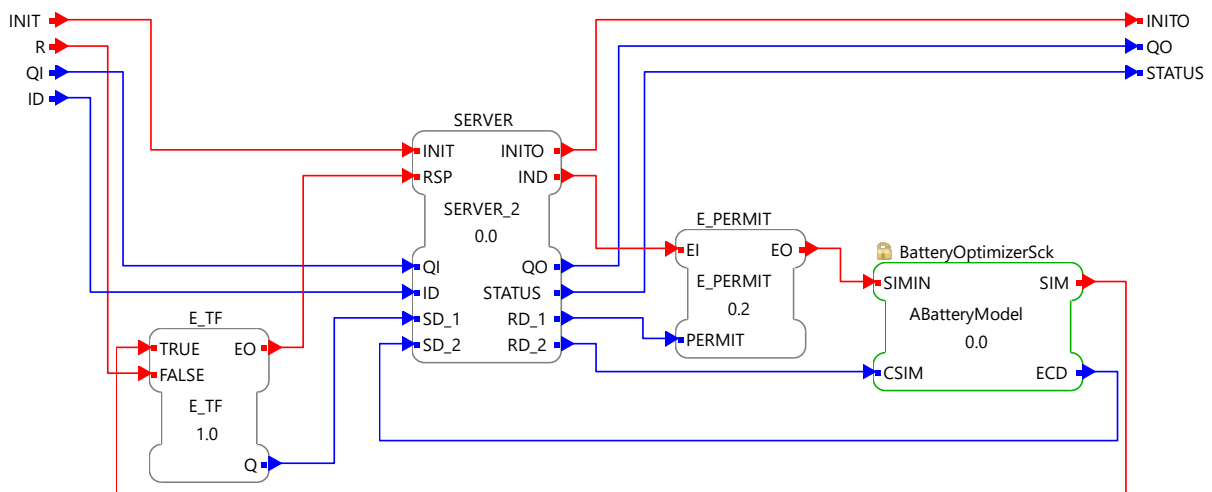
Figure 7.14: *Composite network of the `PolysunBatteryModel` function block.*

data with a temporal resolution other than 1 h. It is, however, possible to activate a linear interpolation of the data in the advanced settings. A re-engineering of Polysun's weather data computations is planned for the future, but exceeds the scope of this thesis.

To minimize the error caused by the inability to define the order in which the components are computed, a high temporal resolution is required. As a matter of fact, however, this may even be a more realistic scenario for real-time controllers, because many system components can have dead times of up to 10 s [24]. Temporal resolutions of between 1 s and 15 min can be enforced in Polysun® using the "Fixed time step controller", a built-in plugin controller. In all of the simulations, the interpolation preference was set to linear. For a visual evaluation of the results, two I/O plugin controllers that export their sensor inputs to Matlab® MAT and CSV files were created[i].

7.3.1. IEC 61499 combined PVprog and PV curtailment co-simulation with Polysun®

To verify the adequacy of Polysun® for co-simulations with real time IEC 61499 control applications, the previously validated PVprog and curtailment application (see section 6.3.3) was used for the test run. The Polysun® system diagram with the plugin controllers used to communicate with the control application is depicted in figure 7.15. It has approximately the same dimensions as the system that was simulated in section 6.3.1 and uses the same load and weather data sources that were used in the

---

[i]They are available for download at `https://github.com/MrcJkb/Polysun-IO-Plugin/releases`.
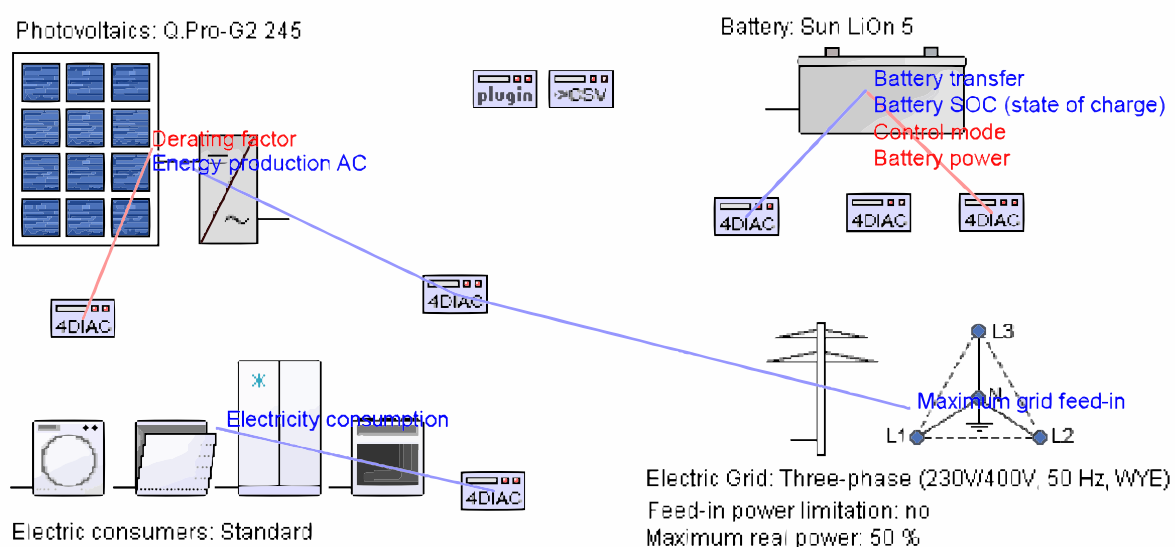
Figure 7.15: *Diagram of the system used in a combined PVprog and PV curtailment co-simulation with Polysun®. Nominal PV power:* 4.9 kWp, *annual electricity consumption:* $5{,}000$ kWh, *usable battery capacity:* 5 kWh.

original PVprog release. The $0°$ south oriented PV modules' tilt angles are set to $30°$. The IEC 61499 application is almost exactly the same as the one used in section 6.3.3, with only the CSIFBs having been switched out for the Polysun® plugin function blocks described in section 7.2 and the `SimpleBatteryModel` having been replaced by the `PolysunBatteryModel`. The results of a co-simulation are presented in figure 7.16 for two selected days. On the left hand side is a sunny day in which the PVprog algorithm is enough to prevent all curtailment by shifting the battery's charge to midday. On the right hand side, the day starts off with a lot of clouds, causing the battery to begin charging at a lower power threshold. When the sun starts shining, the remaining battery capacity is not enough to absorb the entire PV surplus and the curtailment kicks in. Because the controller sets the derating factor according to the 10 min running average using a PID loop, a small overshoot of the feed-in power can be observed just before 12 PM. Altogether, it can be concluded from the visual simulation results that the PVprog algorithm and curtailment controller have excellent coordination and that Polysun® is a more than capable tool for the design and validation of the controller. While its current inability to read temporally high resolved weather data is a definite drawback, the communication and looping overhead in JAVA™ is significantly lower than it is in Matlab®. This resulted in a highly reduced simulation time compared to the same co-simulation with Matlab®. On the machine[i] used for the simulations, Matlab®, using 1 s resolved data, took approximately 3 days, while Polysun® only needed ca. 2.5 h for the same resolution and time frame. That is a performance improvement by a factor of almost 30.

---

[i] Intel i7-6600U processor, 2.6 GHz (3.4 GHz turbo), 16 GB / $1,600$ MHz RAM.
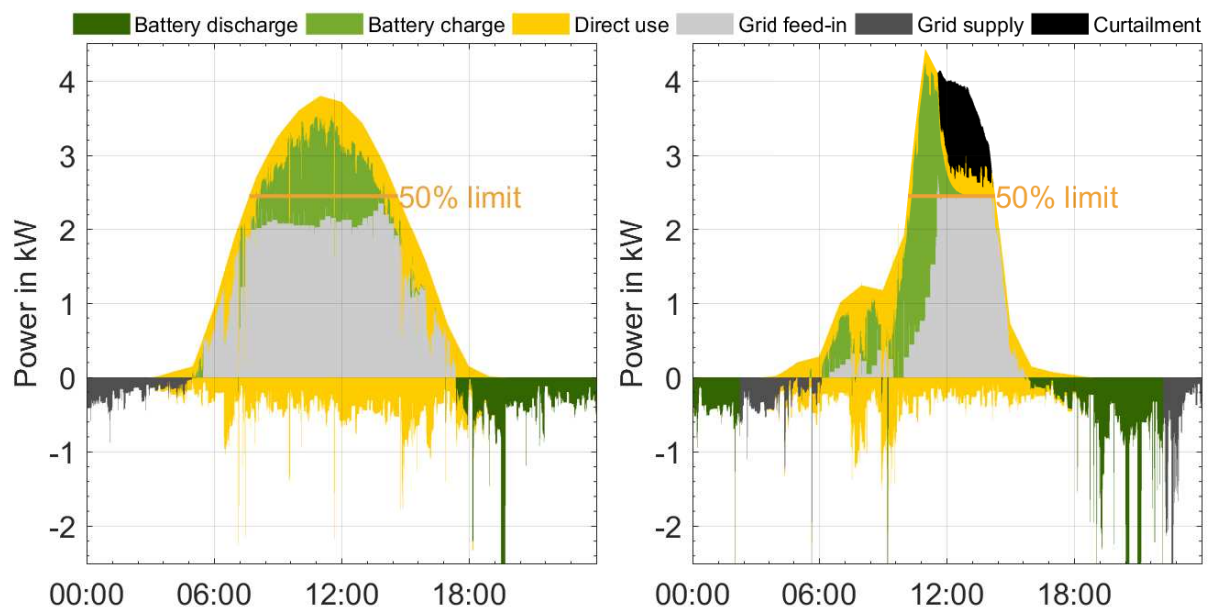
Figure 7.16: *Results of the combined PVprog and PV curtailment application co-simulated with Polysun® on two selected days. Nominal PV power:* 4.9 kWp, *annual electricity consumption:* $5,000$ kWh, *usable battery capacity:* 5 kWh.

7.3.2. IEC 61499 SG Ready heat pump controller co-simulation with Polysun®

In a second step, a simple PV system with a heat pump was simulated. The Polysun® system diagram is pictured in figure 7.17. The system is based on the Polysun® standard template, "16b: Space heating (heat pump, 2 tanks)" with the following adjustments having been made:

- Added a 9.9 kWp PV system.

- Added an electric grid for feed-in.

- Added an electric consumer profile (5 MWh annual consumption).

- Reduced the heat pump's thermal power from 15 kW to 10 kW (and its nominal electrical power from 3.9 kW to 3.3 kW).

- Rotated the three-way switching valve between the heat pump and the storage tanks and adjusted the auxiliary heating controller accordingly. This was done so that water flows from the heat pump into the drinking water tank when the auxiliary heating controller is overridden by the SG Ready heat pump adapter.

For heating, the system has a 600 l buffer tank. This storage tank is ignored by the heat pump controller, since it is mainly used in winter, when there are barely any PV surpluses. The relevant buffer is the 300 l drinking water tank that is active all year, and
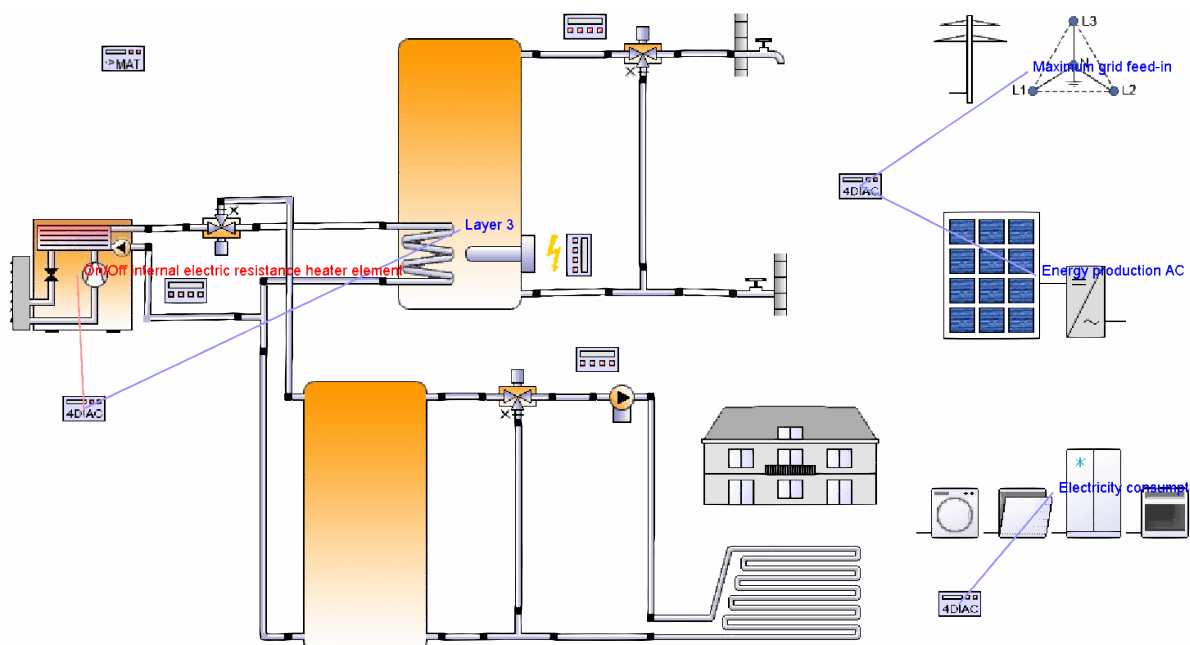
Figure 7.17: *Diagram of the system used in a heat pump controller co-simulation with Polysun®. Nominal PV power:* 9.9 kW *(electrical),* 10.1 kW *(thermal), annual electricity consumption of electrical appliances:* $5,000$ kWh, *annual electricity consumption of thermal appliances:* $5,400$ kWh.

is set to satisfy a hot water demand of $50\,°\text{C}$. It has an internal heater for backup, that theoretically should not be needed and is charged by the heat pump via a heat exchanger. A PV Sensor sends the solar generation power to the FORTE application (see figure 7.18) and a Load Sensor plugin sends the total electrical load. After running the inputs through the `HeatPumpController` FB, the two SG Ready control signals are sent to an SG Ready Heat Pump Adapter, which overrides the control signal of Polysun's built-in auxiliary heating controller. The auxiliary heating controller represents the heat pump's internal controller. It turns the heat pump on and off, and switches the three way valve depending on the temperatures in the storage buffers. The SG Ready heat pump adapter plugin measures the temperature in the third bottommost layer of the 12 layer drinking water tank with a switching threshold set to $55\,°\text{C}$ and a hysteresis of 5 K for the SG Ready control mode 3 (see section 5.3). No internal heating device is used for the amplified operation of the heat pump. The IEC 61499 heat pump controller has its On/Off threshold set to 100 %, as per recommendation for a $P_{\text{STC}}/P_{\text{HP}}$ ratio of $3:1$ by [19]. The simulation results of the system are compared to those of the same system without an override of the heat pump's auxiliary heating controller in figures 7.19 (energy flows) and 7.20 (drinking water storage tank temperatures) for a selected day. The controller adds two load peaks of the heat pump to the day time, thus reducing its use at night and almost completely eliminating the large grid supply peak at 6 PM. Instead of falling steadily during the day time, the temperatures within the controlled system's buffer storage tank are kept at higher levels. By 6 PM, the second layer has a temperature of above 50 ℃, which is more than enough to satisfy the hot water demand. Overall, the use of the controller increases the annual degree of self-sufficiency from 21.2 % to 24.5 % and the self-consumption ratio s from 22.3 % to 26 %. The amounts of energy E_{gf} fed into the grid and E_{gs} purchased from the grid are each reduced by 364 kWh and 298 kWh, respectively. This load shift is approximately equivalent to a month's electricity consumption of the heat pump outside of the heating period.
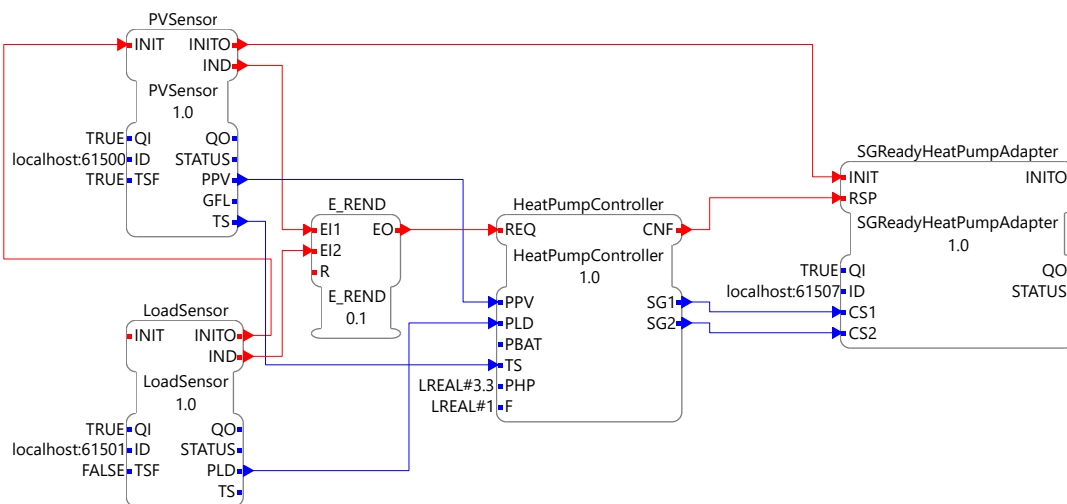
Figure 7.18: *IEC 61499 application for control of a PV heat pump system.*
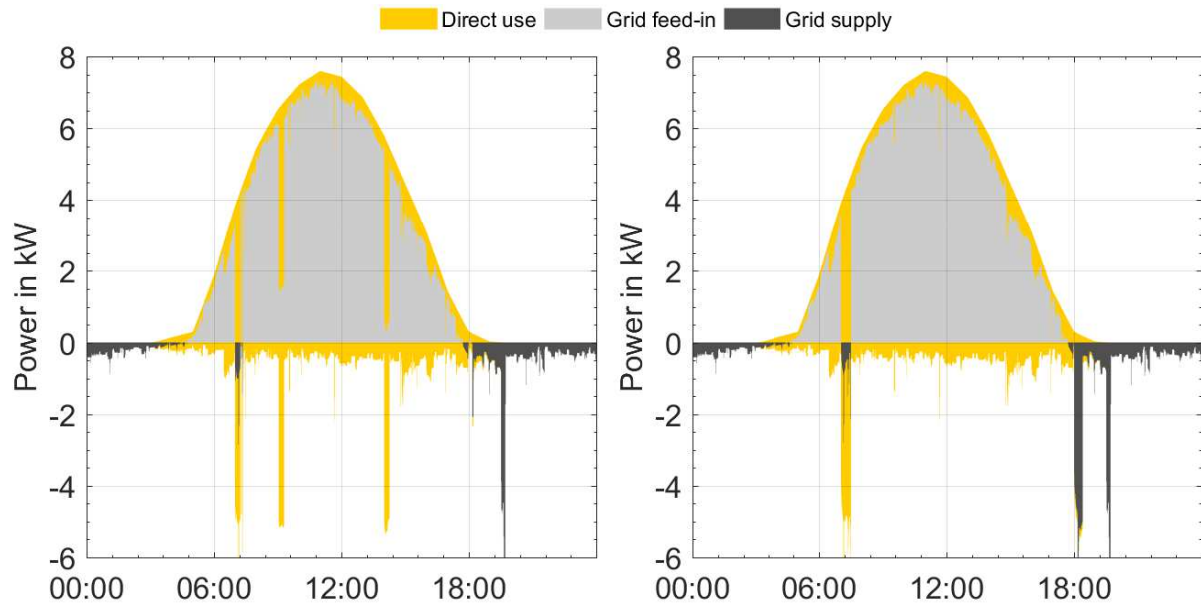
Figure 7.19: *Results of the heat pump controller co-simulation with Polysun®. Energy flows of a system controlled with the IEC 61499 application (left) and a system controlled using only the heat pump's internal controller (right). Nominal PV power: 9.9 kW_p, heat pump's nominal power: 3.3 kW (electrical), 10.1 kW (thermal), annual electricity consumption of electrical appliances: $5{,}000$ kWh, annual electricity consumption of thermal appliances: $5{,}400$ kWh.*
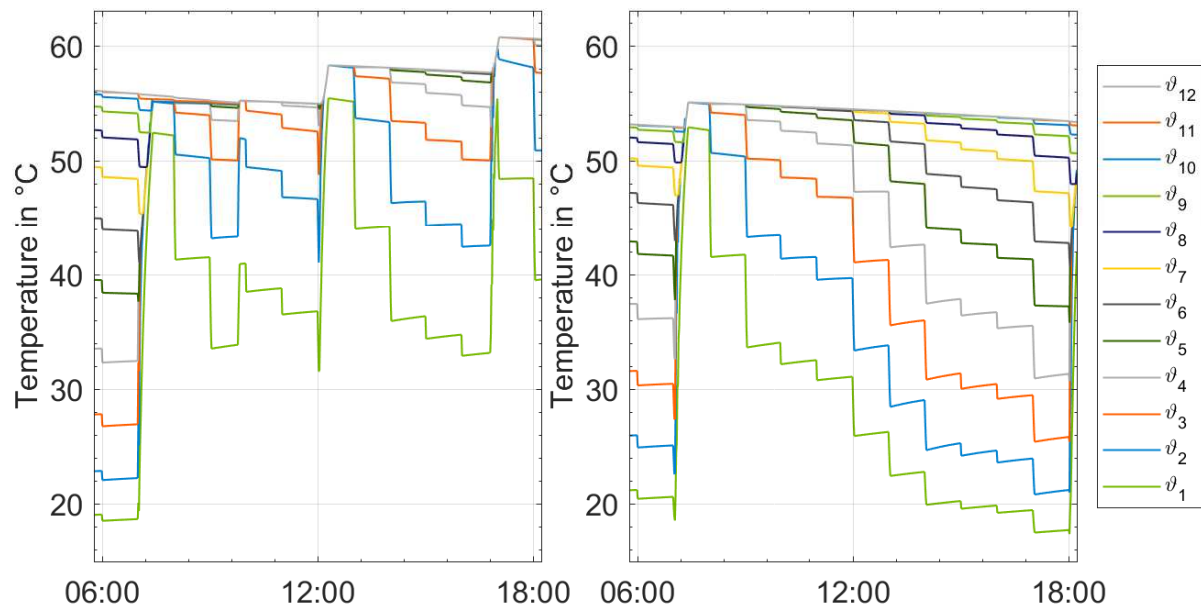
Figure 7.20: *Results of the heat pump controller co-simulation with Polysun®. Storage tank temperatures for a system controlled with the IEC 61499 application (left) and a system controlled using only the heat pump's internal controller (right). Nominal PV power: 9.9 kW_p, heat pump's nominal power: 3.3 kW (electrical), 10.1 kW (thermal), annual electricity consumption of electrical appliances: $5{,}000$ kWh, annual electricity consumption of thermal appliances: $5{,}400$ kWh.*

Further improvements can be achieved for more energy efficient buildings [25] and by fine tuning the controller's On/Off threshold. It can be concluded from the results that

the simple heat pump controller's behaviour is as anticipated and that the way has been paved for combining the heat pump controller, the PVprog controller and the curtailment controller into a single application.

7.3.3. Combined PVprog, SG Ready heat pump and curtailment controller co-simulated with Polysun®

As can be deducted from a brief comparison of the PV heat pump systems' simulation results with those of the PV battery systems, the degrees of self-sufficiency are far lower for the former than they are for the latter. Nevertheless, adding a heat pump to a PV battery system can further increase the overall system performance [25]. For the use in such systems, the applications discussed in sections 7.3.1 and 7.3.2 were combined into a single control application. It is designed in such a way that it can be used with or without a heat pump. In any case, however, a battery is required. In addition to the PV power before curtailment and the load, the battery's power transfer (positive for charging power, and negative for discharging) are sent as a third input to the `HeatPumpController`. Thus, the PV surplus is now equal to the grid feed-in power plus the curtailed PV power. The sequence of the control application is as follows[i]:

- Pass the PV power (before curtailment) and load to the PVprog network to compute the set value of the battery transfer.

- Send the set battery transfer to the battery actor.

- Pass the PV power (before curtailment), load and battery transfer to the heat pump controller function block.

- Pass the PV power (curtailed), load and battery transfer to the curtailment controller function block.

- Send their outputs to the respective actors.

It must be noted that since it is an event oriented application, the order of the sequences is arbitrary in practise. This should have no negative effect on the results. The Polysun® system diagram (not pictured) is a combination of those in figures 7.15 and 7.17. It is essentially the same system as the one simulated in section 7.3.2 with a usable battery capacity of 10 kWh added to it.

To reduce the simulation time, the system was simulated with an increased temporal resolution of 10 s. Figure 7.21 depicts the energy flows on a sunny day resulting from a co-simulation with $f_{\text{On/Off}}$ set to 1. It presents all elements of the control application: Forecast-based battery charging, heat pump optimization and curtailment. Compared

---

[i]The function block network is too large to fit on a single page and can be deduced from the networks of the previously discussed applications.

Figure 7.21: *Results of the combined PVprog, curtailment and heat pump controller (version* 1*) co-simulation with Polysun®. Nominal PV power:* 9.9 kWp, *heat pump's nominal power:* 3.3 kW *(electrical),* 10.1 kW *(thermal), annual electricity consumption of electrical appliances:* 5,000 kWh, *annual electricity consumption of thermal appliances:* 5,400 kWh, *usable battery capacity:* 10 kWh.

to an uncontrolled system, in which the battery is charged as soon as PV surpluses occur and the heat pump controller is never overridden, the degree of self-sufficiency is increased from 39.5 % to 42 %. The self-consumption ratio improves from 44.7 % to 46 % and the curtailment losses are reduced by 443 kWh, while the grid feed-in energy E_{gf} is simultaneously increased by 163 kWh. Additionally, the energy E_{gs} purchased from the grid is reduced by 185 kWh, indicating that the controlled system is more economic than the uncontrolled one in all aspects[i]. It is apparent from figure 7.21 that the deliberate heat pump operation during the day interferes slightly with the PVprog algorithm. The PVprog network has to adjust its output for the heat pump, causing the battery to temporarily charge at lower thresholds. This in turn results in curtailment of the PV generator in the afternoon. To combat this, a variation of the control application was created in which the heat pump's electrical load is measured in addition to the total load. The modified section of the control application is presented in figure 7.22. If the SG Ready control mode 3 is active, the heat pump's load is subtracted from the total to estimate the electrical consumption without the heat pump. This difference load is then passed to the data input of the `LoadForecaster` function block. In normal operation of the heat pump (SG Ready mode 2), the FB receives the total load instead. This way, the deliberately amplified use of the heat pump is treated not as a load, but as what it really is: Energy used for charging a storage buffer. The co-simulation results are depicted in figure 7.23.

---

[i]As with the previous system, the performance improvement depends in part on the energy efficiency of the building.
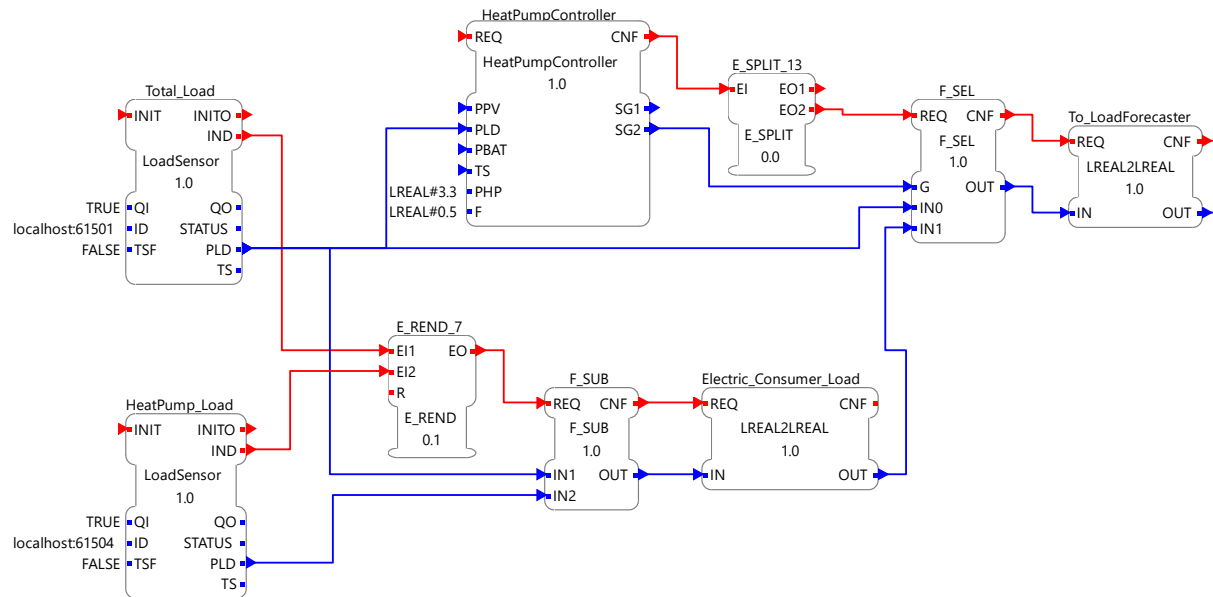
Figure 7.22: *Section of a function block network used to determine which load to pass to the PVprog network for load forecasting depending on the* `HeatPumpController`*'s output. For brevity, event and data flows that are not part of the sequence have been removed from the diagram.*

Overall, the energy flows are more stable than in the previous system. The only adjustment the PVprog network has to make is to reduce the battery's set power when the heat pump is turned on deliberately. It no longer lowers the charging threshold. As a result, the curtailment is reduced even further, by an additional 89 kWh.

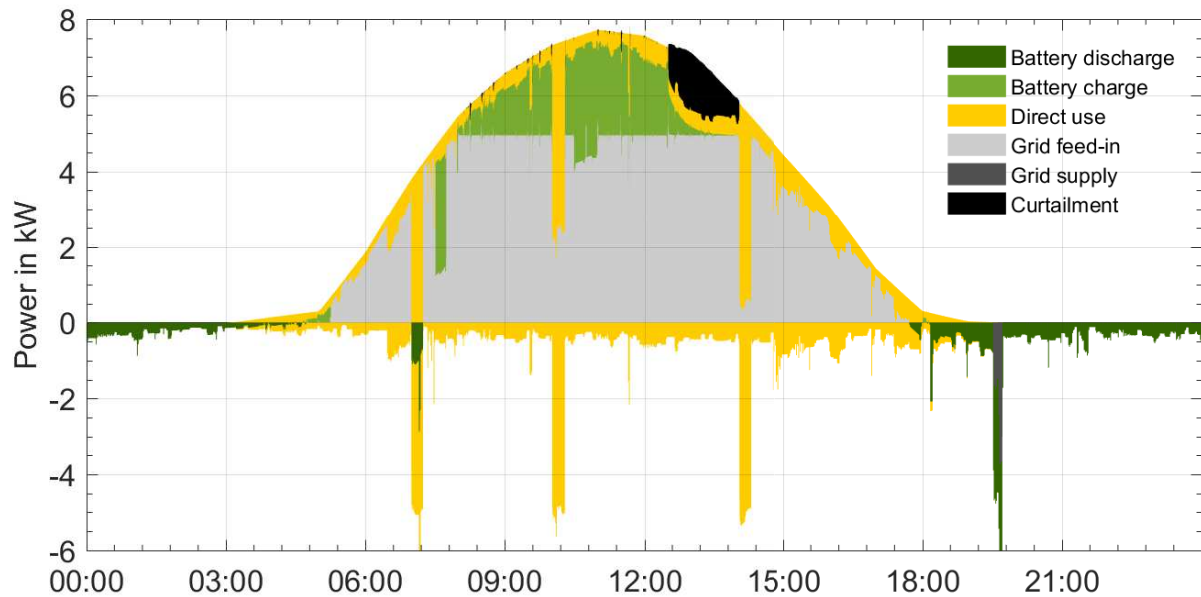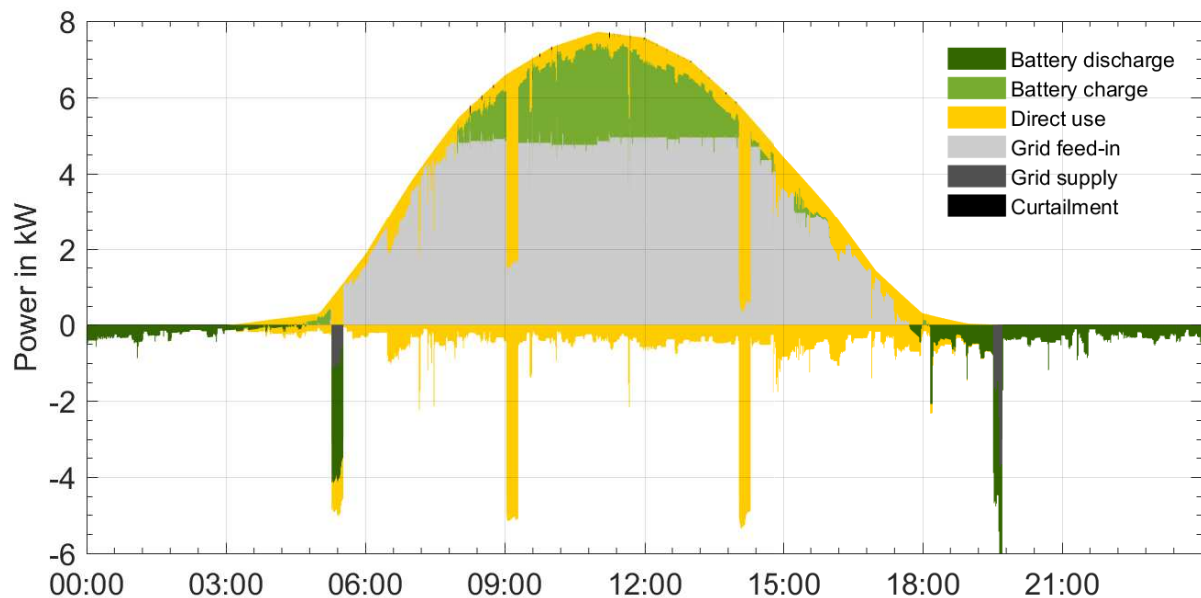**Figure 7.23:** *Results of the combined PVprog, curtailment and heat pump controller (version 2) co-simulation with Polysun®. Nominal PV power: 9.9 kWp, heat pump's nominal power: 3.3 kW (electrical), 10.1 kW (thermal), annual electricity consumption of electrical appliances: $5,000$ kWh, annual electricity consumption of thermal appliances: $5,400$ kWh, usable battery capacity: 10 kWh.*

However, the increased stability comes with the caveat of a reduction in the degree of self-sufficiency by 1.6 % compared to the previous version of the controller. E_{gs} is increased to almost as much as it was for the uncontrolled system. That is, the increase in the energy purchased from the grid is exactly twice as high as the reduction of the curtailment losses. This makes version 2 of the control application with a separate observation of the heat pump's electricity consumption less economical than version 1. In an attempt to further improve the performance, a third and fourth variation of the control applications were created based on versions 1 and 2 (referred to as versions $1b$ and $2b$, respectively). For this, the `HeatPumpController` FB's composite network (see figure 5.32) was altered to activate SG Ready mode 4 if any curtailment occurs. The modified network of the `HeatPumpController2` FB is depicted in figure 7.24. It takes an additional `DF` input for the derating factor. The only change in the application network is that the `PVDERATOR_NMIN_MEAN` FB's `DF` output is passed to the `HeatPumpController2` FB. Figure 7.25 visualizes the function block's conditions for switching between SG Ready modes.

The conditions for switching between modes 2 and 3 remain the same as in equation 5.8. Additionally, the controller switches from any mode it is in to mode 4 if the derating factor is less than 1, i.e. if curtailment has just occurred. From the "Amplified II" mode, it is impossible to switch back to "Amplified I". The same condition is set for switching back to normal operation from either "Amplified I" or "Amplified II". Although it is technically possible for the composite network in figure 7.24 to switch to mode 1 (off), this will never materialize, because curtailment and a load deficit never occur at the same time.

For the co-simulation, it was assumed that the heat pump's SG Ready mode 4 has its upper temperature threshold set to 70 °C and that it activates the internal cartridge heater. Table 7.4 summarizes the results of the four controller variants and compares them to those of the uncontrolled system. Additionally, it lists the results of a system using controller version $2b$ with the drinking water buffer tank's volume increased by 150 l. The additional volume improves the values of all considered performance indicators, except for E_{gs}. For brevity, this analysis was not performed on the other systems. It can be assumed that they would see similar performance boosts. The greatest reduction in curtailment is achieved by version $2b$. From an energetic standpoint, however, the greatest overall system improvement resulting from the control application alone is achieved by version $1b$. The total reduction in electricity purchased from the grid is 229 kWh vs. only 103 kWh for version $2b$. This is a significantly greater difference than the 24 kWh additional reduction in curtailment losses achieved by version $2b$. Nevertheless, a separation of the heat pump's load should be considered for comparison in future applications, since the results may depend on the system dimensions and feed-in limitation. It should be noted that versions 2 and $2b$ of the control application currently cannot be used in a real system unless it is capable of measuring the heat pump's electricity consumption separately.
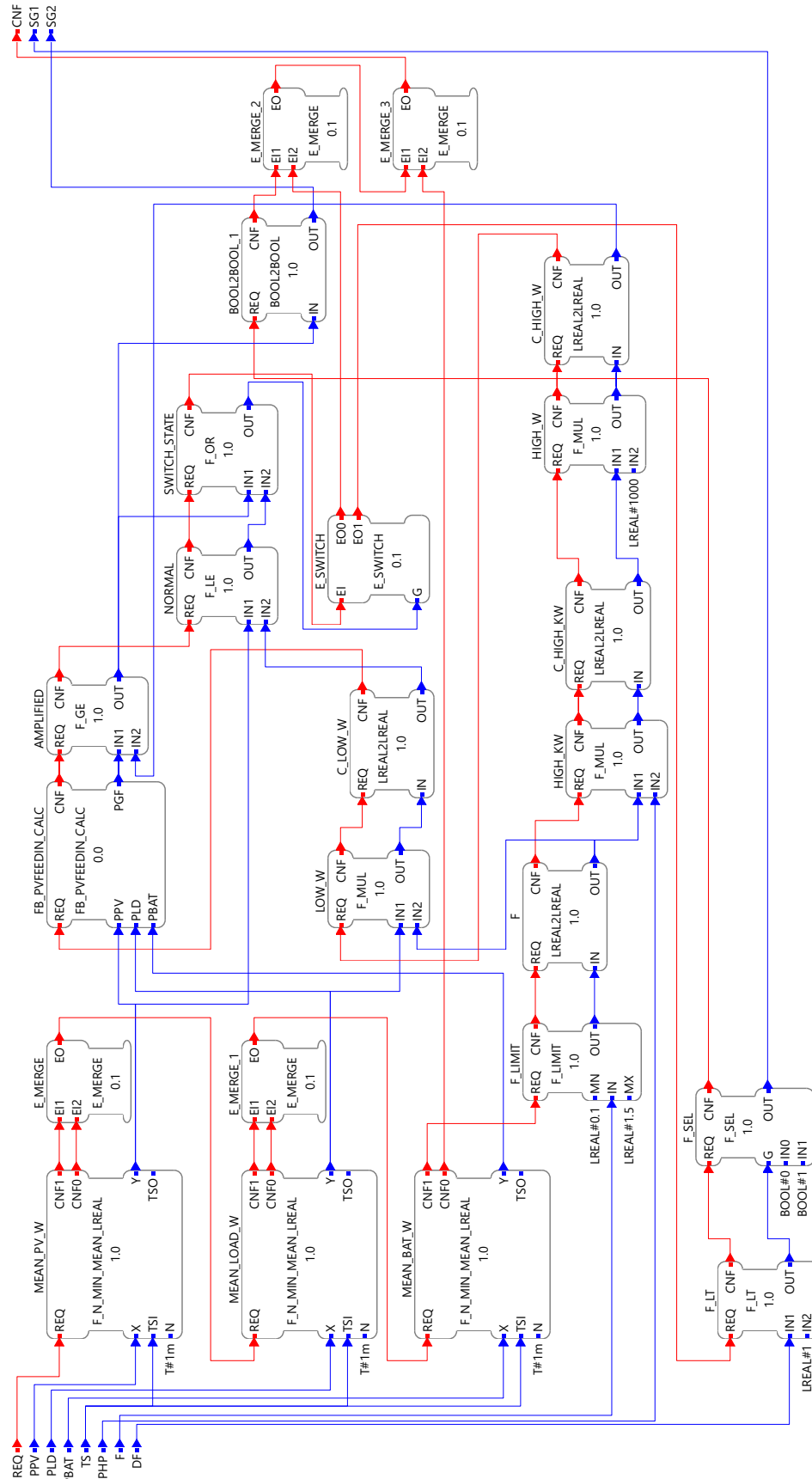
Figure 7.24: *Composite network of the `HeatPumpController2` function block.*
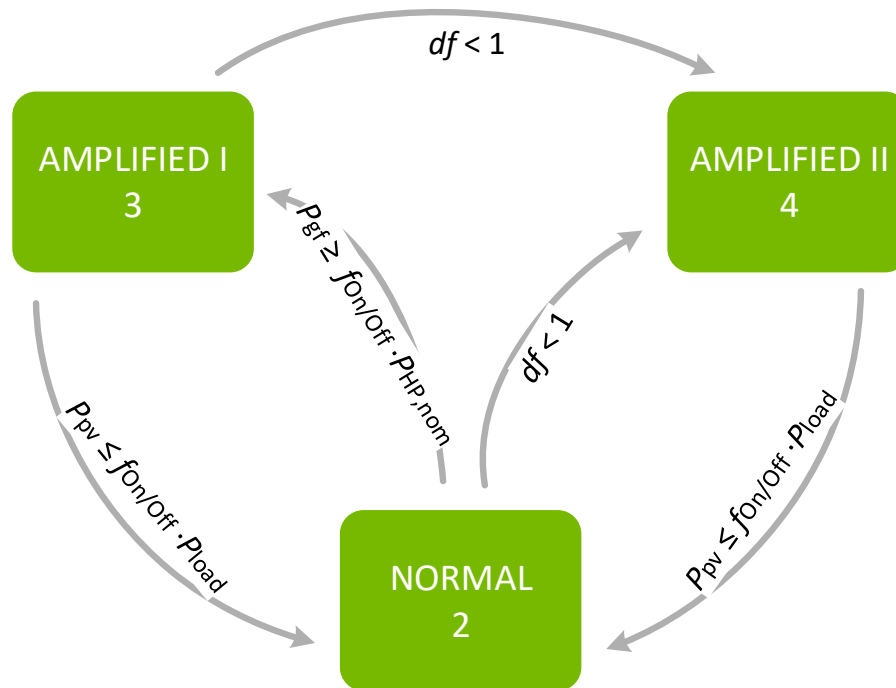
Figure 7.25: *SG Ready mode switching conditions for the* `HeatPumpController2` *function block.*

The potential to further reduce curtailment losses by separating consumption profiles calls for the further development and use of smart devices that are capable of communicating their power flows. To enable more flexibility for the use in DSM, SG Ready heat pumps should allow the configuration of their temperature thresholds in control modes 3 and 4 as well as to define whether or not the internal heating element is used.

---

[i]Version $2b$ with the drinking water tank volume increased from 300 l to 450 l.

Table 7.4: *Comparison of the co-simulated PV heat pump battery systems' results. Version 1: Heat pump load not viewed separately by load forecaster. Version 2: Heat pump load viewed separately by load forecaster. Versions $1b/2b$: Same as versions $1/2$ with SG Ready control mode 4 activated upon curtailment.*

	No controller	Version 1	Version 2	Version $1b$	Version $2b$	$V \uparrow$[i]
a / %	39.5	42	40.4	41.9	42	42.3
s / %	44.7	46	44	45.1	46	46.8
l / kWh	708	265	176	166	142	138
E_{gf} / kWh	4,409	4,572	4,791	4,690	4,568	4,517
E_{gs} / kWh	6,296	6,111	6,289	6,067	6,193	6,194

8. Implementation of new communication protocols in FORTE

The co-simulations in section 7 revealed the potential need for smart system components. Many such components exist already and/or are in development. To maximize their interoperability, the Smart Premises Interoperable Neutral-message Exchange (SPINE) communication protocol was developed by the "EEBus Initiative e.V.". With the aspiration to eventually use the control applications developed within the scope of this thesis with SPINE devices, a project that implements the protocol in FORTE was started[i]. Due to its complexity, a full-blown incorporation of the SPINE protocol into the FORTE communication framework exceeds the scope of this thesis by far, and may even be a worthy topic for another thesis or project work. In addition to the SPINE project, a simple hypertext transfer protocol (HTTP) communication layer was added to FORTE; enabling the interoperability of IEC 61499 applications (assuming client roles) and devices that communicate via a representational state transfer (REST) server[ii]. This section briefly introduces the main aspects of the SPINE protocol, analyses the requirements for an implementation in FORTE and presents an initial design. Finally, the use and implementation of the HTTP communication layer are elucidated.

8.1. The SPINE communication protocol

The specifications of the SPINE protocol define the application (top) layer of the OSI Communication Layer design pattern and are coupled with a smart home IP (SHIP) protocol that defines the transport (bottom) layer [26]. A full technical report containing the protocol and resource specifications for the application layer can be downloaded from the EEBus Initiative's website [27]. The SHIP protocol is still in the standardisation process and its specifications have thus not yet been published. For brevity, a detailed description of the specifications is not provided in this section. Interested readers are advised to review the technical reports. Data are sent and received wrapped in XML structures containing a header and a payload (body). What makes the protocol stand out compared to other communication protocols are:

- The ability of SPINE devices to communicate data about themselves (i.e. electricity consumption).

- The additional control signals accepted by SPINE devices (e.g., a heat pump may be controllable via its 4 SG Ready modes and may additionally allow the electrical or thermal power to be set).

---

[i]The project is hosted as open source at `https://github.com/MrcJkb/forte_spine_comm`.
[ii]Currently, the HTTP communication layer can be downloaded at `https://github.com/MrcJkb/forte_http_comm`. It will later be merged with the official FORTE repository.

- Its "device discovery" and "node management" features. Similarly to the widely used universal plug and play (UPnP), they allow the automatic establishment of connections with various devices on the network once the device in question has been connected. In an IEC 61499 implementation, a single function block would have to be connected to the network and all CSIFBs would connect to any devices they can communicate with without needing to know the address.

A device's node manager handles all connections according to a "device discovery list" it receives from another node manager on the network. The list contains information about every so-called `EntityType`[i] (i.e. a PV inverter), its so-called `FeatureType`s (e.g., `Measurement`), the specific usages (i.e. `Power`) and the connection information. The node manager establishes connections not according to the devices themselves, but according to the `EntityType`s, their `FeatureType`s and specific usages. For example, an `EntityType` that performs curtailment on a PV inverter may denote its `FeatureType` and a specific use as something like: `Setpoint[.DeratingFactor]`. If a PV inverter that supports the specified `FeatureType`s and specific uses exists on the network, the node manager establishes a connection.

Like the IEC 61499 compliance profile for feasibility demonstrations, the SPINE protocol specifies that data can be sent over TCP (client/server) or UDP (publish/subscribe). Publishers and subscribers can also be bound to each other like clients and servers, so that a publisher can only publish to one subscriber, and a subscriber can only listen in on a single publisher.

8.2. Design of a SPINE implementation in FORTE

The main challenge for the implementation of the SPINE communication protocol in FORTE is to abide by both the IEC 61499 standard and the SPINE specifications, since they both define the application layer. A proposed interface of an IEC 61499 SPINE implementation is depicted in figure 8.1. The illustration shows a `SpineNodeManager` function block that initializes the connection to the SPINE network and opens a thread for the node management tasks. Standard IEC 61499 CSIFBs are used for the individual `FeatureType`s, which are specified via the `ID` data inputs with a `spine[]` prefix. An internal `SpineNodeManagementHandler` acts as a mediator between the function blocks and the node manager. To successfully initialize the CSIFBs, the `SpineNodeManager` FB must be initialized first. The Singleton[ii] design pattern could be used for this purpose. Since the `SpineNodeManagementHandler` is always used by a SPINE device, its unique instance can be constructed in a static initializer for thread safety.

---

[i]A device may have multiple `EntityTypes`.

[ii]A singleton is an object of which only a single instance can exist in the entire program. A static method provides a global point of access to it.
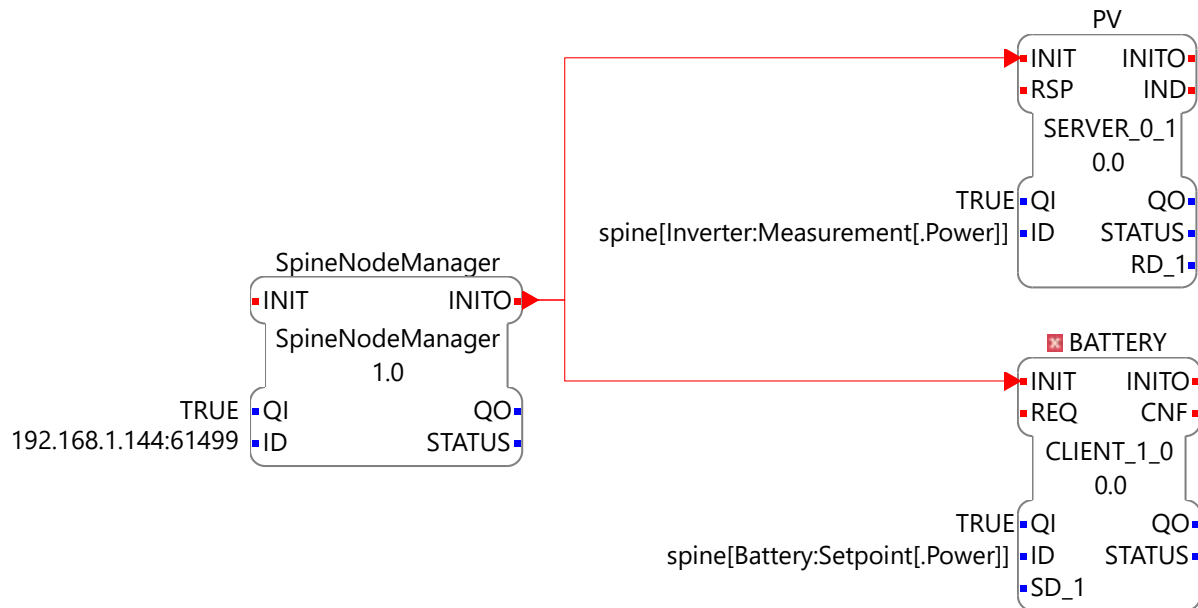
Figure 8.1: *Proposition for the interface of an IEC 61499 SPINE communication protocol implementation. The implementation is still in its design stage, and the interface is subject to change.*

The class dependencies of the protocol implementation in FORTE are presented in figure 8.2. For an understanding of the FORTE communication framework, refer to the 4diac documentation [11] or section 7.1. The `CComFB` is the underlying C++ class that represents a CSIFB in FORTE, and the `CComLayer` is the interface of a communication layer. An additional `CSpineComLayer` implements the application layer defined in [26]. It uses the `SpineNodeManagementHandler`'s `connectEntityType()` method to open the SHIP transport layer's connection, depending on the specific usages specified with the CSIFB's `ID` input. To be able to send and receive XML data, a `SpineXmlParser` object is used. There are many open source and proprietary tools available for generating C++ classes from XML schemas, such as CodeSynthesis, XMLSpy, gSOAP and xsd2cpp. The generated classes represent the XML data structures and are capable of parsing and de/serializing. Since the interface of the CSIFBs is standardized, the `SpineXmlParser` acts as an adapter between the generated XML classes and the function block. It must be initialized at runtime according to the `EntityType` specified in the `openConnection()` method's parameters.

8.3. Implementation of the HTTP communication protocol in FORTE

Today, many building energy components must be accessed via a REST application programming interface (API). For example, storage systems by Sonnen GmbH (previously Sonnenbatterie GmbH) host a REST server, which allows to read measurements of connected components as well as to set the battery's charging or discharging power.

CComLayer // Communication layer
m_poFb // CComFB
m_poTopLayer
m_poBottomLayer
————————————————————————
- openConnection()
- closeConnection()
- sendData()
- recvData()
- processInterrupt()

CComFB // CSIFB
M_poTopOfComStack // CComLayer
————————————————————————
- executeEvent()

SpineNodeManager
mNodeManagementHandler
————————————————————————
- executeEvent()

CSpineComLayer
mNodeManagementHandler
mXmlParser
————————————————————————

SpineNodeManagementHandler
mUniqueInstance
mComLayerContainer
————————————————————————
- getInstance()
- openConnection()
- closeConnection()
- connectEntityType()

SpineXmlParser
————————————————————————
- put()
- get()
- serialize()
- deSerialize()

——— Class inheritance

———▶ Invocation

Figure 8.2: *Class dependencies of the planned SPINE implementation in FORTE. The implementation is still in its design stage, and the dependencies are subject to change.*

This is done via HTTP GET and PUT requests, either as plain text or in the JavaScript object notation (JSON) format [28]. To enable the interoperability of such devices with IEC 61499 applications, a plain text HTTP communication layer was added to FORTE. It uses no external C++ libraries and can thus be used with all of the compatible devices listed in table 2.3. The implementation complies with the FORTE development documentation [11]. Since it currently just provides HTTP client functionality, only CLIENT function blocks can be used with the protocol. An HTTP server implementation would be possible, but appears less likely to be needed. Thus, it has been omitted so far.

A challenge lies in the fact that most HTTP servers deliberately close the clients' connections after a certain time-out period. An implementation that makes use of FORTE's available IP layer is often not fast enough to handle the received data before the connection is closed by the peer. This results in the CSIFB's process being interrupted too soon and in an INITO output event being issued with a false qualifier (ref. table 3.1). Unfortunately, a function block control loop that detects lost connections and attempts to re-open them eventually causes FORTE to crash due to corrupted memory. This is likely the case because FORTE was designed for industrial applications in which socket connections are kept open at all times. To work around this issue, a customized

IP communication layer was added. It handles opening the connection, sending and receiving data and closing the connection all within its `sendData()` method. This occurs in a loop until either the response has been received or a time-out period is exceeded. As a result, the only times an `INITO` event with a `false` qualifier can be issued are either when the function block is de-initialized or if the connection identifier is invalid. Any other problem (such as an unexpected response) is indicated with a `CNF` event and a `false` identifier. In this case, the response header is sent to the data output for debugging purposes.

The type of function block used determines the HTTP method. Since just GET and PUT with plain text are used, only `CLIENT_0_1`[i], `CLIENT_1`[ii] and `CLIENT_1_0`[iii] function blocks are supported at the moment. An example is illustrated in figure 8.3. If a `CLIENT_1` FB is used, the HTTP response header is sent to the `RD_1` output. The `ID` input can be one of the following:

- `http[ip:port/path]`

- `http[ip:port/path;expected_response]`

where `ip` is the server's IP address, `port` is the port number, `path` is the HTTP path. In the top FB in figure 8.3, for example, the `path` is `/rest/devices/battery/M03`. Additional optional arguments are separated with a semicolon.

---

[i]For GET, with no data inputs.
[ii]For PUT, with one data input and output, respectively.
[iii]For PUT, with no data outputs.
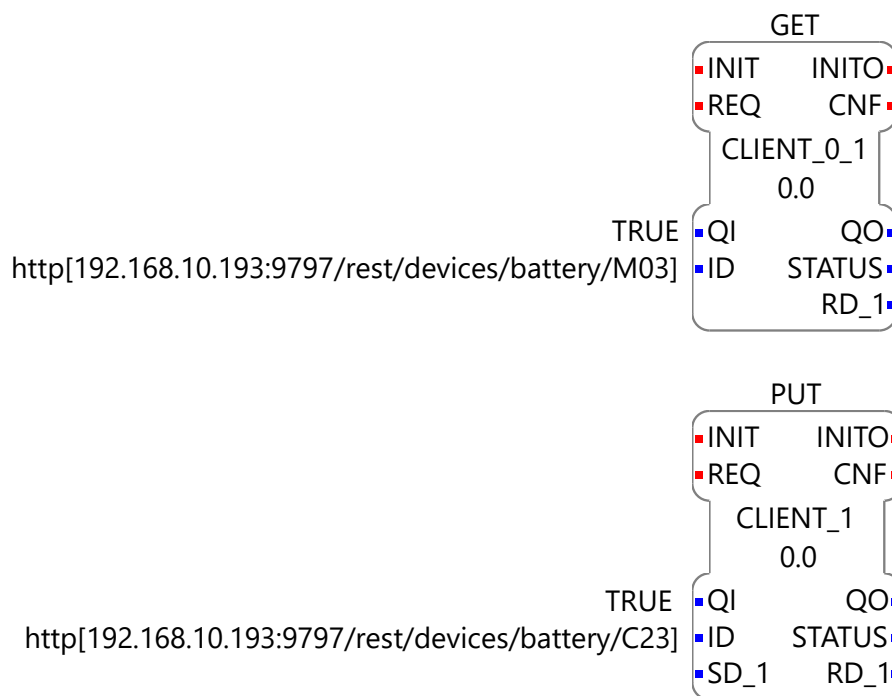
Figure 8.3: *Example for the execution of HTTP PUT and GET requests in 4diac.*
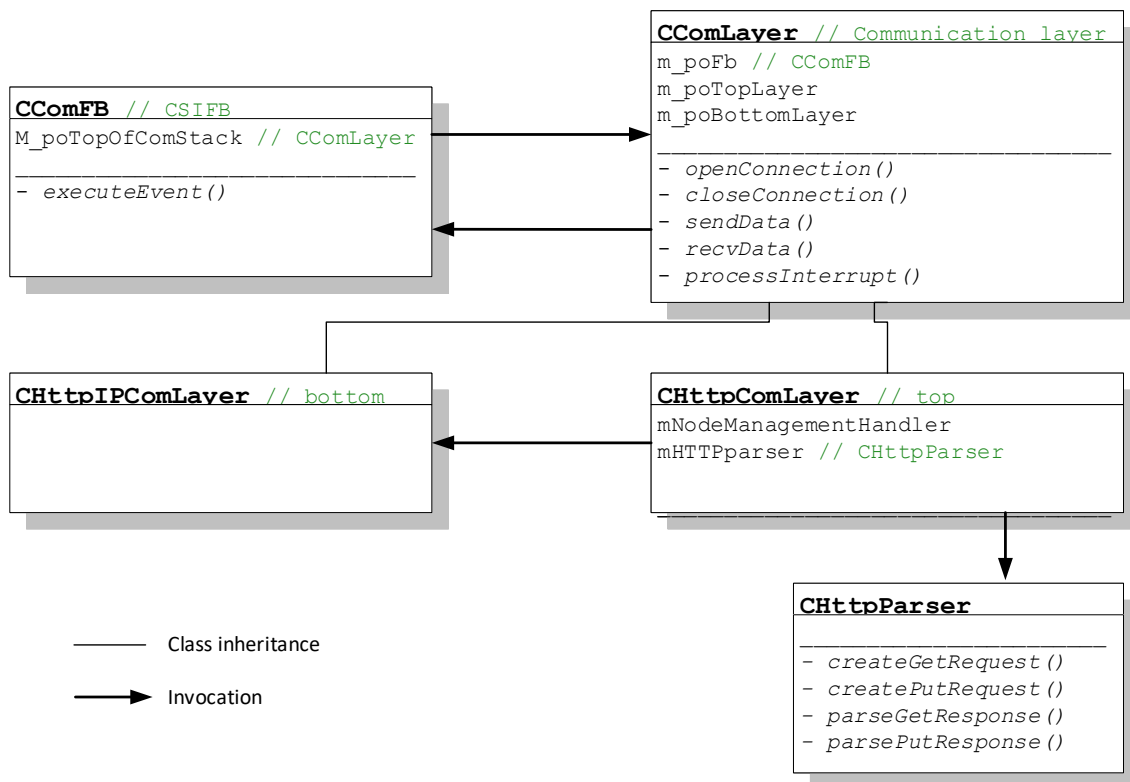
Figure 8.4: *Class dependencies of the HTTP implementation in FORTE.*

Currently, the only optional argument is `expected_response`, which is set to `HTTP/1.1 ... 200 OK` if none is specified. An overview of the class dependencies of the HTTP protocol implementation is presented in figure 8.4. An additional `CHttpParser` class is used for handling the creation of requests and the interpretation of responses. This de-couples the communication layer from the API, increasing the flexibility and making it easier to adapt the code for different HTTP-based APIs.

9. Deployment and field test

Section 2.2 briefly describes the MVC design pattern and how it can be projected to this thesis. So far, the model has been one of the simulation tools; Polysun® and Matlab®. No actual view has been implemented, but the concept has been present in the form of 4diac-IDE's monitoring feature. As mentioned previously, the MVC design pattern decouples the model, the view and the controller from each other by using the Observer and State design patterns. Put simply, one can exchange one model for another without having to make any changes to the controller or the view, and vice versa. This has already been applied to the combined PVprog and curtailment control application co-simulated in section 6.3.3, where the Matlab® simulation model was replaced by Polysun® in section 7.3.1. There were no changes made to the control application apart from the CSIFBs and the fact that the `SimpleBatteryModel` function block was replaced by the `PolysunBatteryModel`. The latter was merely done to prove the flexibility of the controller, and the CSIFBs are technically not part of the control application. They merely provide the interface. Furthermore, commercial tools such as ISaGRAF and nxtSTUDIO handle the communication interface automatically. This feature is also expected for a future version of 4diac [6].

In this section, the advantage of the MVC approach is proven further. The control application "Version $1b$", developed in section 7.3.3, is deployed to a Raspberry Pi 2 and its PVprog operation is used in a real system.

9.1. System overview

The controller was installed in the "Living Equia" house at HTW Berlin [29], an energy-plus house now used primarily for research. An overview of the set-up is depicted in figure 9.1. The building has a rooftop PV system with a nominal capacity of 4.6 kWp and a "Sonnenbatterie" Li-ion battery with a usable capacity of 5.3 kWh, according to the
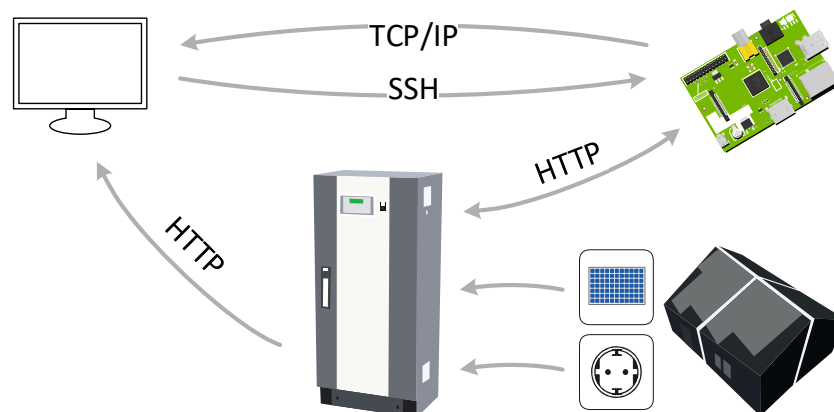

Figure 9.1: *Graphical representation of the controller's field test set-up.*

manufacturer[i]. FORTE, running on the Raspberry Pi, uses the HTTP communication protocol implemented in section 8.3 to communicate with a REST server hosted by the battery. All of the required data - the PV system's power, the building's load, the battery's charging or discharging power and its operation mode (automatic/manual) - are accessed via GET methods. Additionally, the charging/discharging power and operation mode are set by sending PUT requests to the server.

Fortunately, the battery comes with its own view; an online web interface that can be accessed via browser. For additional monitoring capabilities, CSIFBs were added to the control application. Upon request, they send the measurements of the current day and the current forecasts (PV power, load and battery charge roadmap) via TCP/IP sockets. The Raspberry Pi's physical location is within the building. It was connected to the same local network as the battery. A remote configuration is possible via secure shell (SSH).

9.2. Communication interface adjustment

In the co-simulations, the controller's CSIFBs were SERVER function blocks. Since the battery communicates via a REST server, it is not capable of periodically sending updates to the controller. The controller must assume a client role and request the data from the battery. As a result, the CSIFBs had to be rearranged in such a way that they run in a control loop. Additionally, there is no way of requesting a time stamp from the server. In order to generate one, a simple SIFB, FB_SYS_DT was created. Since its internal implementation is written in C++[ii], it cannot be used with any runtime that does not support the programming of SIFBs in C++, other than FORTE. However, since it performs a simple task, porting it to another environment should not be difficult. The function block's interface is easily explained. Upon request, it issues an output event along with the device's network time protocol (NTP) corrected system time in

---

[i]The actual usable capacity may have been slightly reduced due to degradation.
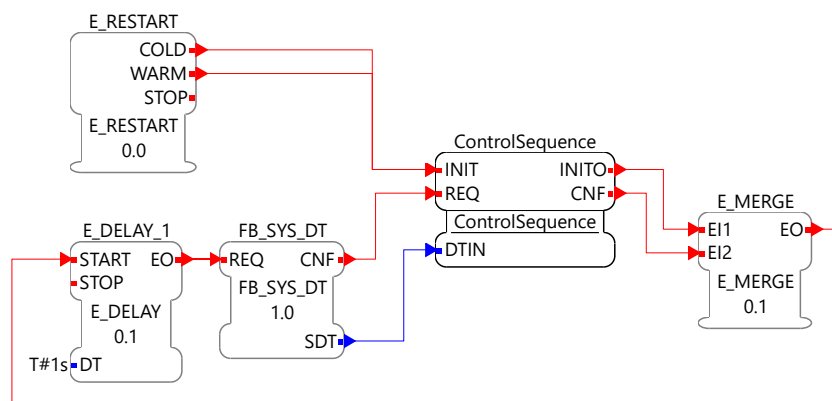[ii]The source code is provided with the function block library.

Figure 9.2: *Qualitative illustration of the control loop implemented for use with a REST server.*

the `DATE_AND_TIME` format. The control loop is illustrated qualitatively in figure 9.2, wherein the unmodified control application and its CSIFBs are represented by the `ControlSequence` subapplication. The loop is triggered after initialization. A delay of 1 s was added to reduce the server load and thus the amount of bad HTTP responses due, for example, to server errors.

9.3. Stability improvements

Unlike simulation models, physical devices are prone to failure. To ensure that the controller runs with as little down time as possible, the following stability improvements were made:

9.3.1. Client CSIFB wrappers

Every now and then, the Sonnenbatterie server encounters issues, causing it to send an `HTTP 1.1 500 Internal Server Error` response. Additionally, it sporadically does not send a response at all, in which case the respective request times out. Neither of these cases interrupt the controller process, as the corresponding `CLIENT` function block still issues a `CNF` event (with a `false` output qualifier, see section 8.3). Nevertheless, if occurring over a longer period, such behaviour may influence the controller's output negatively. A preferable solution is to temporarily pause the controller's operation and wait for the server to recover. To implement this behaviour, a set of wrappers for `CLIENT` function blocks using the HTTP communication protocol were created. They share the same interface as regular `CLIENT` function blocks, except for the fact that their data inputs and outputs are bound to specific data types. As a result, connections do not have to be reconfigured when replacing `CLIENT` FBs with them. Figure 9.3 illustrates the composite network of a `CLIENTRC_0_1` function block. The wrapped `CLIENT_0_1` FB's output events are delegated according to the event qualifier. In the case of success (`QO = true`), the `CNF` event is directly passed on to the CFB's output. Otherwise, it is rerouted to the `CLIENT_0_1` FB's `INIT` input with a `false` qualifier, causing the network stack to be deleted. The deletion is confirmed with an `INITO` output event and a `false` output qualifier, which in turn triggers a reinitialization of the CSIFB. To give the server time to recover, the `INITO` output with a `true` qualifier is passed back to the CSIFB's `REQ` input via a delay of 5 s. This loop continues until a positive response is received. The three top function blocks (`E_TF`, `F_SEL` and `E_SWITCH`) determine whether an `INITO` event was triggered by a previous failed request attempt or by the composite function block's external `INIT` input.
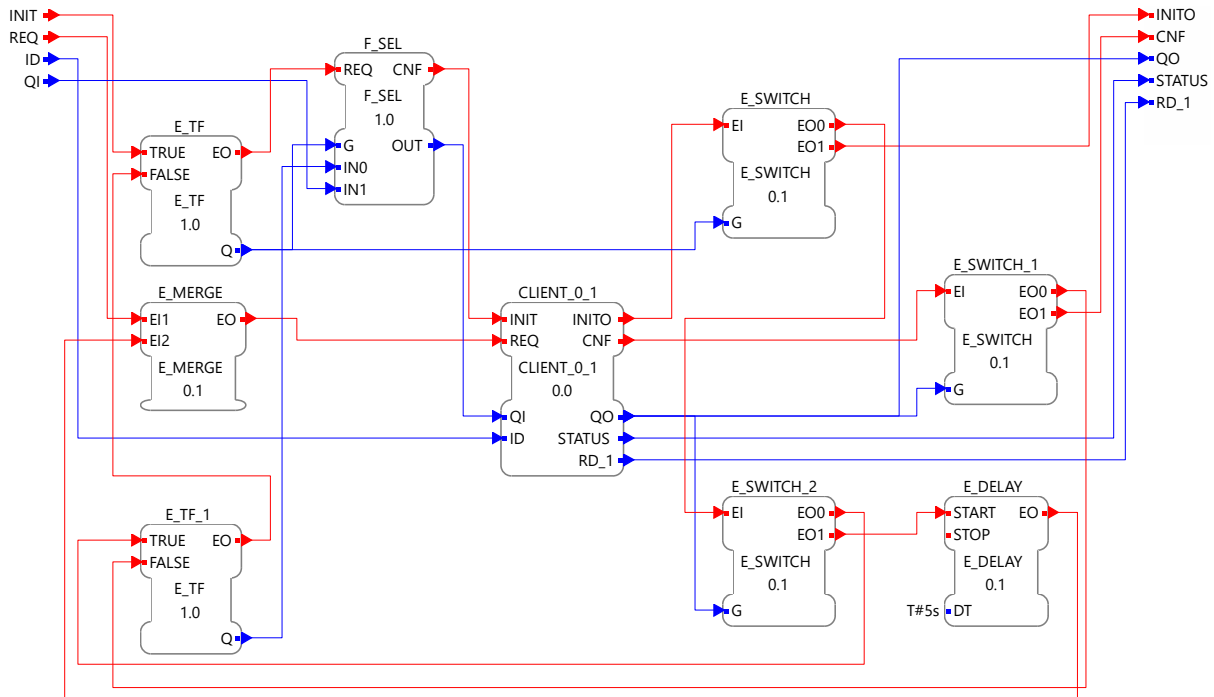
Figure 9.3: *Composite network of the CLIENTRC_0_1 function block.*

9.3.2. Watchdog timer

Another failure possibility for any piece of hardware is freezing. This could occur for various reasons, for example due to overheating, memory leaks, etc. The Linux kernel features a so-called Watchdog API that allows the use of a hardware circuit which can reboot the device if a failure is detected [30]. The circuit runs separately from the processor so that it is unlikely that both would fail at the same time. Such a circuit exists for the Raspberry Pi. Once a watchdog timer is started using the API, it must be pinged regularly (the default interval is usually 15 s), or else it will reboot the device. An SIFB that can be used to access the Watchdog API from IEC 61499 applications was developed and added to the function block library. It can only be used in Portable Operating System Interface (POSIX) systems; attempting to include it in a Win32 version of FORTE will cause the build to fail. The interface of the function block is depicted in figure 9.4.

Figure 9.4: *Interface of the FB_WATCHDOG function block.*

An `INIT` event with a `true` qualifier enables the watchdog timer, and a `false` qualifier disables it, respectively. The desired time-out can be set; however, it must be noted that the device may ignore the set value in some cases. For example, this can occur if another application has already set a shorter time-out or if the set value is outside the supported range.

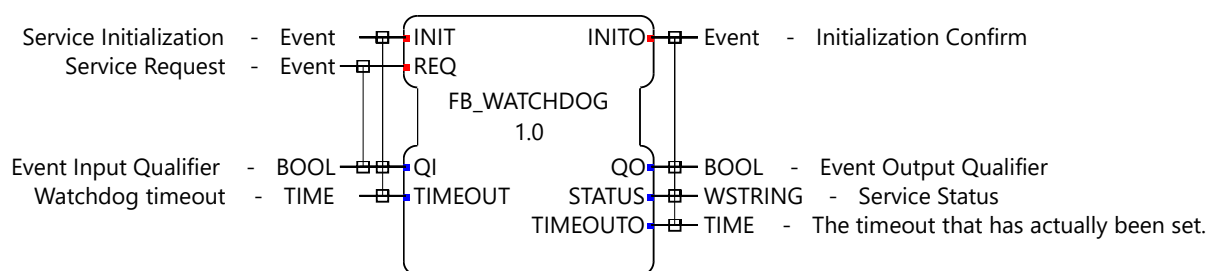For this reason, users are advised to extensively test the function block on the device it is to be used on before using it in an application. The `TIMEOUTO` data output indicates the time-out that was actually set. To ping the watchdog, the `REQ` input is used. It is generally advisable to proceed with caution when using the function block on a device on which FORTE is started automatically. For example, if the control application loads before the device's network interface is fully initialized (see section 9.5 for how to prevent this), the `FB_WATCHDOG` function block could cause a continuous reboot loop. As an SIFB, the `FB_WATCHDOG` cannot be directly used with an RTE other than 4diac. Its source code is included in the library so that it can be transferred to any IEC 61499 environment that supports C or C++ code.

9.3.3. Forecast data backups

The `LoadForecaster` BFB requires 1 day and the `PVForecaster` BFB takes up to 10 days to fully initialize. Because of this, reboots can have a negative impact on the PVprog operation's performance. To minimize the detriment, the two BFBs were replaced with SIFBs, `FortePVForecaster` and `ForteLoadForecaster`, that perform exactly the same operations. In addition, however, they save a backup of their critical data every minute and attempt to load it upon initialization. That way, the initialization phase is shortened if the device reboots unexpectedly. As with the other SIFBs, they are not portable, but their source code is included in the library so that equivalent SIFBs can be created for other environments. If they do not support C++ code, however, the standard BFB variants must be used. A suggestion to the authors of IEC 61499 would be to extend the ST programming language with the ability to save and load variables and arrays.

9.4. Sonnenbatterie CSIFBs

Unfortunately, simply setting the control mode to manual does not suffice to gain control over the Sonnenbatterie. In order to allow the battery to calibrate, it should be put into automatic mode when fully charged. When in trickle charging mode, it should be charged briefly in order to release it from that mode. Finally, switching the battery's operation mode from automatic to manual should be followed by a 10 s delay to prevent it from going into standby. So that users do not have to worry about any of this, the functionality is wrapped in an `SBActorRC` composite function block. Its interface, which is almost identical to that of the `BatteryActor` function block used in co-simulations
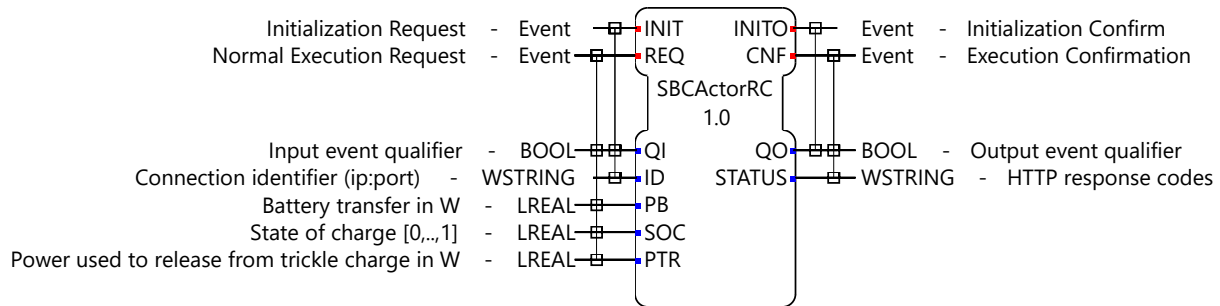
Figure 9.5: *Interface of the `SBActorRC` function block.*

with Polysun® (see section 7.2.3), is presented in figure 9.5. For the aforementioned calibration purposes and trickle charge release, it takes two additional inputs: The SoC and the power used to release the battery from its trickle charging mode. Due to the large amount of function blocks used, the CFB's internal composite network is not discussed in this paper. Interested readers are advised to examine the function block in 4diac. A corresponding `SBSensorRC` (see figure 9.6) wraps around three `CLIENT` function blocks and converts their outputs to the interface of the `BatterySensor` function block used in the Polysun® co-simulations (see section 7.2.2). According to the Sonnenbatterie API, three separate requests must be sent to retrieve positive values for both the charging and discharging power in W, respectively, and the SoC in %. The `SBSensorRC` FB performs all three requests with a single `REQ` event and outputs a positive value for charging, a negative value for discharging power and a normalized SoC, as is expected by the control application. Both function blocks output the HTTP response codes with the `STATUS` data outputs for debugging purposes. To request the load and PV power from the REST server, the use of `CLIENTRC` function blocks is sufficient.

9.5. Deployment to the Raspberry Pi

The deployment of FORTE to the Raspberry Pi (running Raspbian OS) was performed according to the manual found in the 4diac documentation [11].
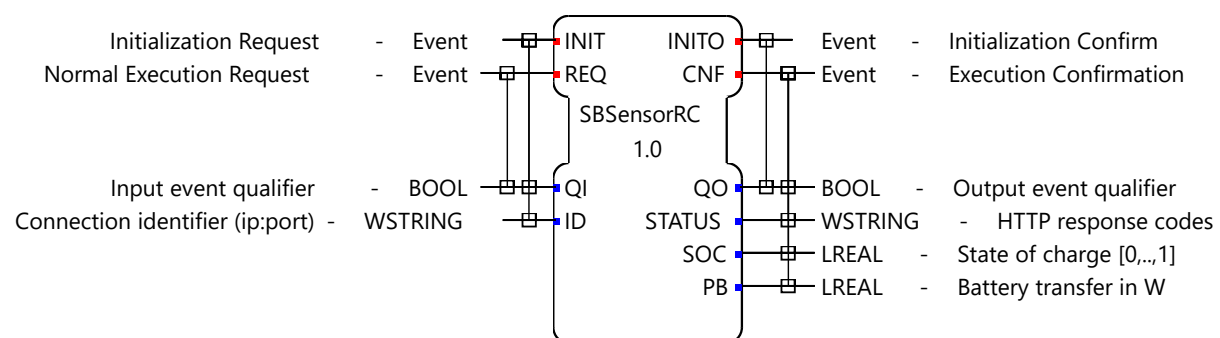

Figure 9.6: *Interface of the `SBSensorRC` function block.*

The following additional configurations were performed in order to start FORTE automatically upon booting the Raspberry Pi and to load the application:

- Exported the application to a FORTE boot file (see 4diac documentation) that is placed in the same directory as FORTE and loaded once FORTE starts up.

- Created a bash script, `startForte.sh`, that waits for a network connection before changing the working directory to the directory containing the boot file and FORTE and then starting up the application. The script also logs all terminal outputs to a text file for debugging purposes.

- Added a line calling `startForte.sh` to the script, `/etc/rc.local`, that runs at boot time.

Assuming that FORTE, the boot files and the `startForte.sh` script are all in the `/home/pi/` directory, the line added to `/etc/rc.local` is as follows:

```
\home\pi\startForte.sh &
```

With the contents of `startForte.sh` being:

```bash
#!/bin/bash
#Checks for a network connection and then starts forte
STATE="error";
while [ $STATE == "error" ]; do
    # Check for an active network connection
    STATE=$(ping -q -w 1 -c 1 `ip r | grep default
    | cut -d ' ' -f 3` > /dev/null &&
    echo ok || echo error)
    # Wait for 2 seconds
    sleep 2
done
# cd to forte dir to enable boot file loading
cd /home/pi/
(
# Start FORTE...
./forte
# ...and log output
) &> /home/pi/forteLog.txt
```

The `startForte.sh` script must be made executable with the following shell command:

```
chmod \home\pi\startForte.sh +x
```

9.6. Monitoring results

The energy flows of the field test are presented in figure 9.7 for a selected day. They clearly prove that the PVprog operation performs as intended in a real system. In the early morning, the dynamic charging threshold is set to a relatively low value and is adjusted for the high PV generation throughout the course of the day. The clouds in the late afternoon result in the threshold being lowered again. Causes for the low threshold in the morning can include:

i) The PV power output in the morning and/or evening of the last day is significantly lower than $P_{\text{pv,max}}$.

ii) The controller is still in its initialization phase. It takes up to 10 days for the PV forecasts to reach their optimal quality.

iii) All of the previous 10 days were cloudy, causing $P_{\text{pv,max}}$ not to be equivalent to $P_{\text{pv,cs}}$.

iv) The usable battery capacity is lower than stated by the manufacturer, e.g., due to degradation.

In this case, the day before the one pictured in figure 9.7 was completely sunny, eliminating (iii) from the possible reasons. Nevertheless, (i) could be a cause due to the fact that the field test was performed in September - a time in which the PV power output is reduced slightly every day due to the sun's decreasing declination - especially in the early mornings, where horizon shading is most prominent.
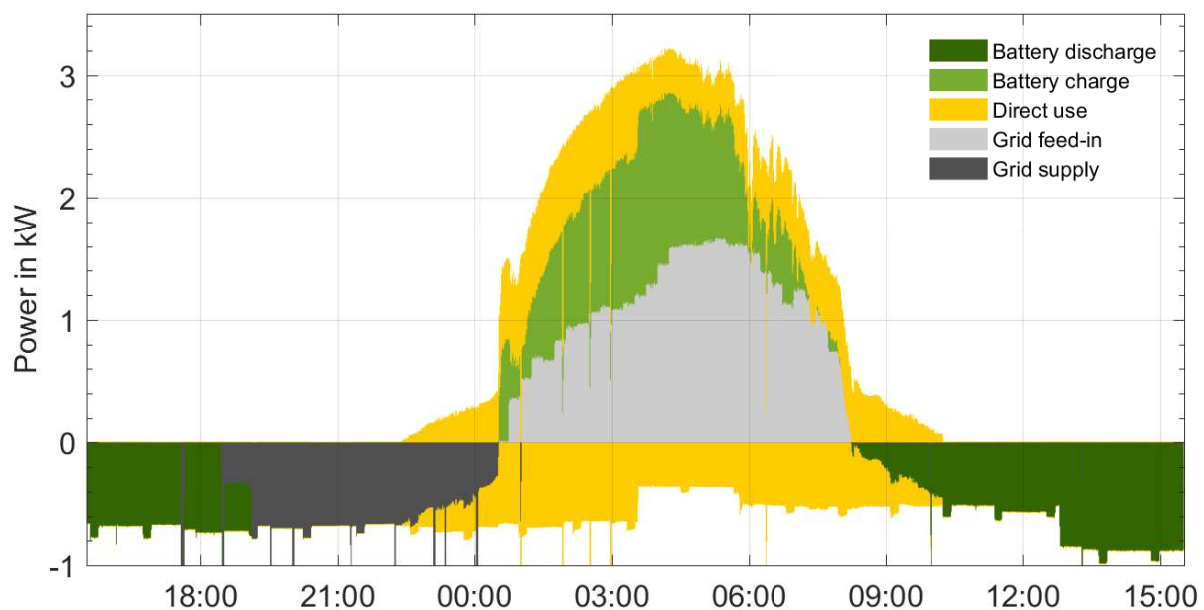
Figure 9.7: *Measured energy flows on a selected day during field testing of the control application running on a Raspberry Pi 2. The measurements are recorded as* 1 *min averages. Nominal PV power:* 4.6 kWp, *usable battery capacity:* 5.3 kWh.

The depicted day in figure 9.7 is exactly 10 days after having installed the controller, so (ii) can be ruled out. This leaves (i) and (iv) as the contributing factors.

The former should be of no concern and the latter could be solved by adding a subapplication to the controller that estimates the usable capacity during discharging at night. It could be triggered when the battery reaches an SoC of 100 %. The capacity in kWh would be estimated by integrating over all measured power flows coming out of and going into the battery until it is empty. An SoC of 0 % could then be used to trigger an update of the usable capacity for the PVprog subapplication. Further analysis is necessary to test the reliability of such an implementation. For the scope of this thesis, however, the field test can be regarded as successful.

10. Summary, outlook and conclusion

After having defined the criteria for the control applications - flexibility, portability, easy understandability and no boundaries by proprietary limitations - a strategy using an MVC approach was devised. The highly flexible and portable IEC 61499 standard was chosen along with the open source development environment, 4diac, as a tool for implementing the strategy. A set of IEC 61499 function block libraries were developed for the use in building energy systems with PV, batteries and/or SG Ready heat pumps. They implement the PVprog algorithm [1] - forecast-based PV feed-in limitation and maximization of self-sufficiency, curtailment and DSM of heat pumps based on [19]. After briefly introducing the reader to the PVprog algorithm and the IEC 61499 standard, the function block libraries were documented in detail. Then, they were used for the development of the actual control applications.

Before proceeding with the development, the tools available in 4diac for the testing of function blocks and IEC 61499 applications were analysed. Though plentiful and useful, they were found to be insufficient for the validation of complex control applications for multi-generator energy systems. Thus, the testing capabilities were extended by developing a 4diac/Matlab® TCP/IP communication library. Its use is documented in detail in this paper. With the `tcpip4diac` class, it was made possible to validate IEC 61499 control applications by using them with Matlab® simulation models. This was done for a PVprog control application, a curtailment application and a combination of the two. The co-simulation of the combined PVprog and curtailment application answered open questions about configuration details. It was shown that each component of the PVprog subapplication needs to know the uncurtailed PV power for optimal operation. Thus, a combination of it with curtailment in the same control application is advisable.

To be able to incorporate heat pump DSM into the control applications and validate them, the simulation software Polysun® was used. For communication, a library that implements the OSI layer design pattern was created and used with Polysun's plugin controller feature. The result was the `Polysun4diac` controller plugin that, together with the communication library it uses, is documented in detail in this thesis. With the actor and sensor plugin controllers, Polysun's adequacy for co-simulations with real time IEC 61499 control applications was first verified by running another co-simulation of the previously validated control application.

Next, an SG Ready heat pump controller, which was later combined with the PVprog/-curtailment application, was devised and co-simulated. A first simulation of the three combined subapplications provided the valuable insight that the SG Ready subapplication interferes slightly with the PVprog operation. Because it has to adjust its output for the heat pump, the PVprog network temporarily lowers the battery charging threshold, which in turn results in curtailment during afternoon hours. This effect was mitigated by creating a variation of the control application that takes separate measurements of

the heat pump and the rest of the load. The heat pump's electricity consumption is treated by the PVprog algorithm as a load when it is operated normally, and as energy storage when in an amplified operation mode. This resulted in a further reduction of the curtailment losses, but also in a half as large reduction of electricity purchased from the grid. Adjusting the SG Ready controller to activate mode 4 when curtailment occurs negated the negative effect on grid purchase slightly, but even more so for the initial version of the control application. Whether or not it is more beneficial to separate the heat pump's electricity consumption most likely depends on the system dimensions and feed-in limit. Comparing the two simulation tools that were used, the development process using the Polysun® plugin was faster than that using the Matlab® library. This is due a 30-fold performance increase in co-simulation time for Polysun®. For 1 s resolved simulations of a year, simulation times of 2.5 h were achieved using the Polysun® plugin. Even shorter simulation times could in theory be achieved by incorporating the ability to run IEC 61499 applications natively in Polysun®.

In a step toward enabling a communication of the developed IEC 61499 applications with SPINE devices, a design was devised for the protocol's implementation in 4diac. Its implementation, however, is planned outside the scope of this thesis. In addition to the initial design of the SPINE communication layer, an HTTP layer was developed and documented to make 4diac compatible with REST APIs used by many building energy components today.

In a final validation step, an application that communicates using the HTTP layer was deployed to a Raspberry Pi 2 and installed in a building with a PV system and a battery. Some minor changes had to be made to the application, because the server/client roles of the system are swapped compared to the simulation. However, those changes are minor, and could easily be automated. Furthermore, using the server role in Polysun® would significantly reduce the need to make changes in the controller. To be able to do this, the Polysun® controllers would have to be run from within parallel threads. Alternatively, a single plugin controller could be created that puts the communication sockets in a multi-threaded environment, but this would decrease the flexibility of the plugin. The need to make changes to the control application could eventually be completely eliminated by implementing the HTTP and SPINE communication protocols into the `Polysun4diac` plugin controller. This would be beneficial not only for the use with IEC 61499 control applications.

In the field test, additional challenges for the real time use of the control applications were identified and overcome. They consisted of the need for a watchdog timer in case of hardware or software failure, the necessity to handle communication issues and the optional addition of the ability to backup internal data that take a long time to initialize. For better portability of the latter feature, an implementation of the ability to save variables in the ST programming language would be desirable. Although the time constraints did not allow for a full year of field testing, enough measurements

could be collected to prove that the PVprog subapplication performs just as intended. It would be interesting to one day be able to compare the results of a PV battery heat pump system co-simulated with Polysun® and 4diac with a Raspberry Pi deployment in the field. Overall, however, the results of this thesis bring us to the conclusion that the project has been a success. The need for generic control software based on standards for multi-generator energy systems has been met with a flexible, fully open source IEC 61499 solution that can be deployed to a large variety of low-cost hardware. Thanks to the communication libraries, its functionality can easily be extended and quickly validated using simulation software. With the rate at which the global renewable industry is growing, the potential for further development of this project may never cease to exist. By providing a first step toward the establishment of an open source community in the so-far luxury field of intelligent energy management, this thesis is an important contribution to a much needed energy revolution.

References

[1]   J. Bergner, J. Weniger and T. Tjaden, *PVprog-Algorithmus - Algorithmus zur Umsetzung der prognosebasierten Batterieladung für PV-Speichersysteme mit messwertbasierten PV- und Lastprognosen*, version 1.1, Berlin, 2016 (cit. on pp. 1, 17, 22, 23, 60, 63, 116).

[2]   E. Freeman, E. Robson, K. Sierra and B. Bates, Eds., *Head First design patterns*, Sebastopol, CA: O'Reilly, 2004, ISBN: 978-0-596-00712-6 (cit. on p. 4).

[3]   I. E. C. (IEC), Ed., *IEC 61131 Programmable controllers*, Geneva: International Electrotechnical Commission (IEC), 1993 (cit. on p. 5).

[4]   I. E. C. (IEC), Ed., *IEC 61499 Function blocks*, 2nd ed., Geneva: International Electrotechnical Commission (IEC), 2012 (cit. on pp. 5, 9).

[5]   W. Bolton, *Programmable logic controllers*. Kidlington, Oxford, UK; Waltham, MA: Newnes, 2015, OCLC: 930872167, ISBN: 978-0-08-100353-4 (cit. on p. 5).

[6]   A. Zoitl and T. Strasser, *Distributed control applications: guidelines, design patterns, and application examples with the IEC 61499*. 2016, OCLC: 932464063, ISBN: 978-1-4822-5906-3 (cit. on pp. 5, 6, 15, 16, 107).

[7]   A. Zoitl, *Modelling control systems using iec 61499*, 2nd edition, ser. IET control engineering series 95. Stevenage, Herts, United Kingdom: The Institution of Engineering and Technology, 2014, 227 pp., ISBN: 978-1-84919-760-1 (cit. on pp. 6, 10, 11, 13).

[8]   (). IEC 61499 compliance profile for feasibility demonstrations, [Online]. Available: `http://www.holobloc.com/doc/ita/index.htm` (visited on 26/04/2017) (cit. on pp. 6, 14, 53).

[9]   I. E. C. (IEC). (). IEC 61499 - international standard for distributed systems, [Online]. Available: `http://www.iec61499.com/` (visited on 27/04/2017) (cit. on p. 7).

[10]   (). Eclipse community forums: 4diac - framework for distributed industrial automation and control - BFB algorithms in java/c++, [Online]. Available: `https://www.eclipse.org/forums/index.php/t/1085751/` (visited on 27/04/2017) (cit. on p. 8).

[11]   (). 4diac documentation, [Online]. Available: `https://www.eclipse.org/4diac/en_help.php` (visited on 29/05/2017) (cit. on pp. 12–14, 51, 73, 103, 104, 112).

[12]   J. Li and M. A. Danzer, 'Optimal charge control strategies for stationary photovoltaic battery systems', *Journal of Power Sources*, vol. 258, pp. 365–373, 15th Jul. 2014, Li14, ISSN: 0378-7753. DOI: `10.1016/j.jpowsour.2014.02.066` (cit. on p. 18).

[13] J. Weniger, J. Bergner, T. Tjaden, J. Kretzer, F. Schnorr and V. Quaschning, 'Einfluss verschiedener Betriebsstrategien auf die Netzeinspeisung räumlich verteilter PV-Speichersysteme', in *30. Symposium Photovoltaische Solarenergie*, Bad Staffelstein, Mar. 2015 (cit. on p. 18).

[14] M. Jakobi, 'Optimierung des Netzeinspeiseverhaltens von deutschlandweit verteilten PV-Speichersystemen mit prognosebasierten Betriebsstrategien', Bachelorthesis, Hochschule für Technik und Wirtschaft HTW Berlin, Berlin, 2015 (cit. on p. 18).

[15] J. Bergner, 'Untersuchungen zu prognosebasierten Betriebsstrategien für PV-Speichersysteme', Bachelorthesis, Hochschule für Technik und Wirtschaft HTW Berlin, Berlin, 2014 (cit. on pp. 20, 21, 35).

[16] Forum Netztechnik/Netzbetrieb im VDE (FNN), 'Anschluss und Betrieb von Speichern am Niederspannungsnetz', Forum Netztechnik/Netzbetrieb im VDE (FNN), Berlin, Jun. 2014, FNN14 (cit. on pp. 44, 65).

[17] (). SG Ready-Datenbank — Bundesverband Wärmepumpe (BWP) e.V., [Online]. Available: `https://www.waermepumpe.de/normen-technik/sg-ready/sg-ready-datenbank/` (visited on 02/08/2017) (cit. on p. 46).

[18] Bundesverband Wärmepumpe e. V. (). Regularien des SG Ready-Labels, SG Reagy-Label - für das Smartgrid geeignete Wärmepumpen, [Online]. Available: `http://www.waermepumpe.de/fileadmin/redakteurdaten/Waermepumpe/Qualitaet/SG_READY/2012-11-01_SG_Ready_Regularien_Version1.1.pdf` (visited on 16/12/2014) (cit. on p. 46).

[19] T. Tjaden, J. Weniger and V. Quaschning, 'Richtige Dimensionierung von Photovoltaik, Wärmepumpen und Pufferspeichern', C.A.R.M.E.N.-Symposium 2017, Straubing, 11th Jul. 2017 (cit. on pp. 46, 47, 93, 116).

[20] C. D. Gladisch, *Verification-based software-fault detection*. Karlsruhe: KIT Scientific Publishing, 2011, 264 pp., OCLC: 837764081, ISBN: 978-3-86644-676-2 (cit. on p. 51).

[21] J. Weniger and T. Tjaden. (Mar. 2017). Performance-Simulationsmodell für AC-gekoppelte PV-Batteriesysteme (PerModAC Version 1.0), Hochschule für Technik und Wirtschaft HTW Berlin, [Online]. Available: `http://pvspeicher.htw-berlin.de/permod` (cit. on pp. 64, 66, 68).

[22] T. Bülo, D. Geibel, S. Sutter and K. Boldt, 'Entwicklung neuer Technologien zur Erhöhung der Aufnahmefähigkeit von Erneuerbaren Energien in Niederspannungsnetzen: Entwicklung einer intelligenten Netzstation', SMA Technology AG, 31st Dec. 2013 (cit. on p. 67).

[23] *Polysun user manual*, version 10.0, Winterthur, Switzerland, 2017 (cit. on p. 80).

[24] J. Weniger, T. Tjaden, J. Bergner and V. Quaschning, 'Auswirkungen von Regelträgheiten auf die Energieflüsse in Wohngebäuden mit netzgekoppelten PV-Batteriesystemen', in *31. Symposium Photovoltaische Solarenergie*, Bad Staffelstein, Mar. 2016 (cit. on p. 90).

[25] T. Tjaden, 'PV-Systeme mit Wärmepumpe ideal betreiben', *pv magazine*, pp. 106–108, Feb. 2015 (cit. on pp. 94, 95).

[26] 'EEBus SPINE technical report', EEBus Initiative e.V., Cologne, Germany, 1.0.0, 29th Apr. 2016 (cit. on pp. 101, 103).

[27] (). Download standard - EEBus Initiative e.V., [Online]. Available: `https://www.eebus.org/download-standard/?lang=en&lang=en` (visited on 18/08/2017) (cit. on p. 101).

[28] *Sonnenbatterie API for battery systems operated in slave mode*, 2013 (cit. on p. 104).

[29] (). Solar Decathlon Europe 2010 - Living Equia - Teilprojekt HTW Berlin (ENOB) (SD Europe (BMWi-Energieoptimiertes Bauen Bundesförderung)) - Hochschule für Technik und Wirtschaft Berlin University of Applied Sciences - HTW Berlin, [Online]. Available: `http://www.htw-berlin.de/forschung/online-forschungskatalog/projekte/projekt/?eid=1161` (visited on 21/09/2017) (cit. on p. 107).

[30] C. Weingel. (2002). The linux watchdog driver API, kernel.org, [Online]. Available: `https://www.kernel.org/doc/Documentation/watchdog/watchdog-api.txt` (visited on 21/09/2017) (cit. on p. 110).
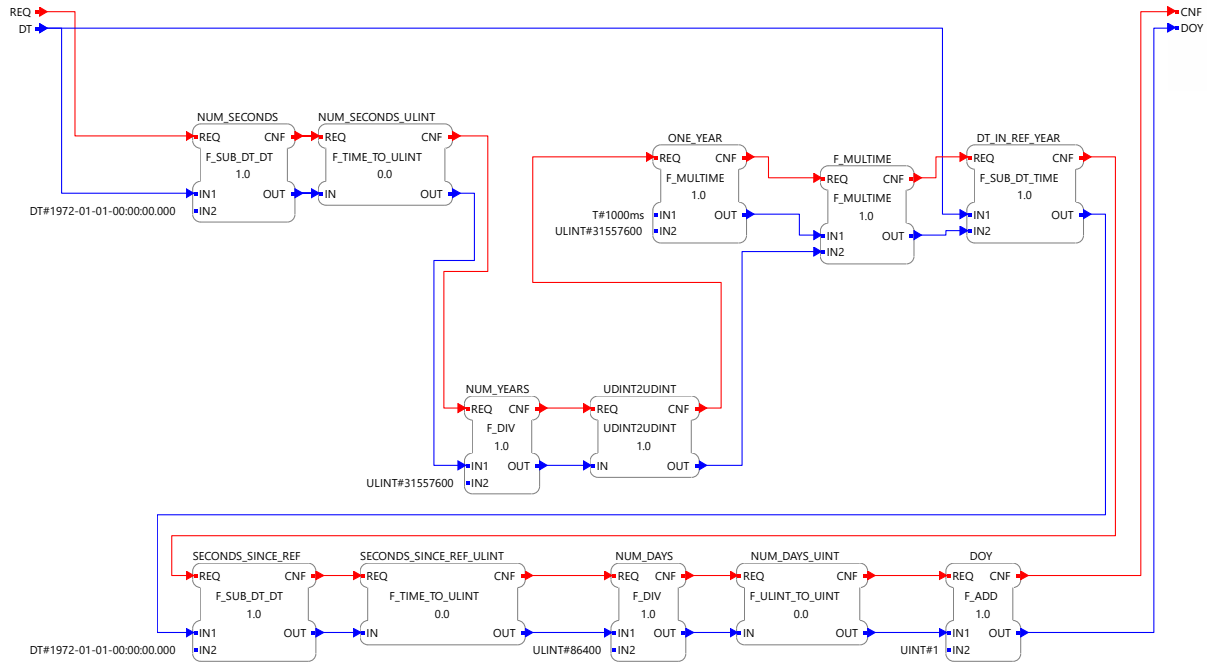
Figure A.1: *The DT_TO_DOY_UINT function block's composite network.*

A. Additional helper function blocks

PVprog utility function blocks

The DT_TO_DOY_UINT FB converts DATE_AND_TIME data types to the day of the year, a UINT between 1 and 366. 1 represents January 1st and 365 or 366 represent December 31st, depending on the occurrence of a leap year. The DT_TO_TD_UINT FB converts DATE_AND_TIME data types to the minute of the day ("time of day"), a UINT between 0 and 1439, representing the number of minutes since 12:00 AM.
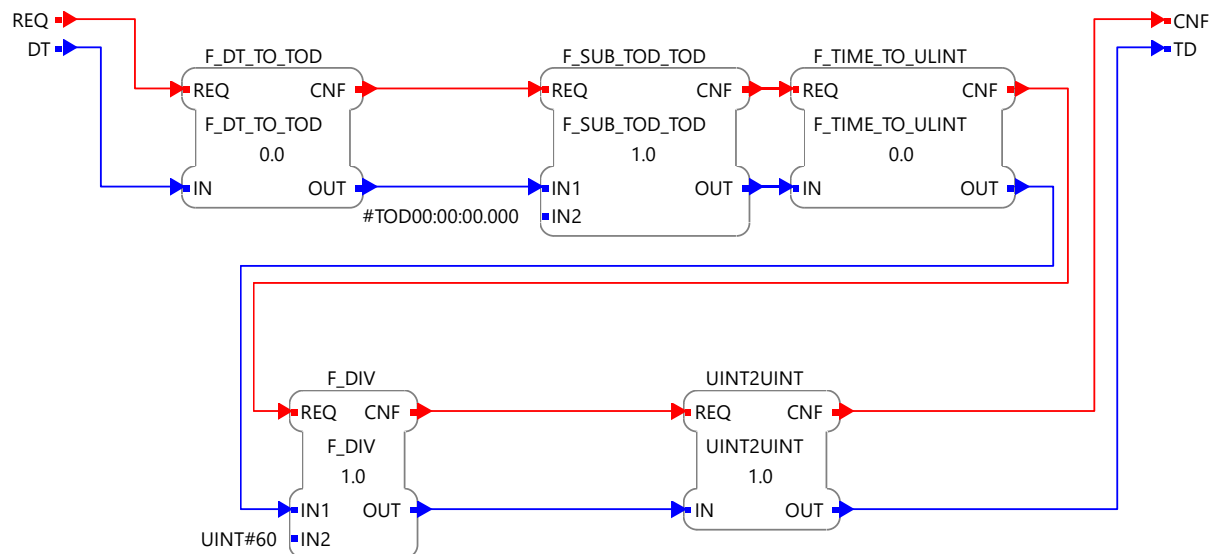

Figure A.2: *The DT_TO_TD_UINT function block's composite network.*

The `F_N_MIN_MEAN_LREAL` FB estimates the mean of data within fixed-sized intervals and returns the output at the end of each interval.
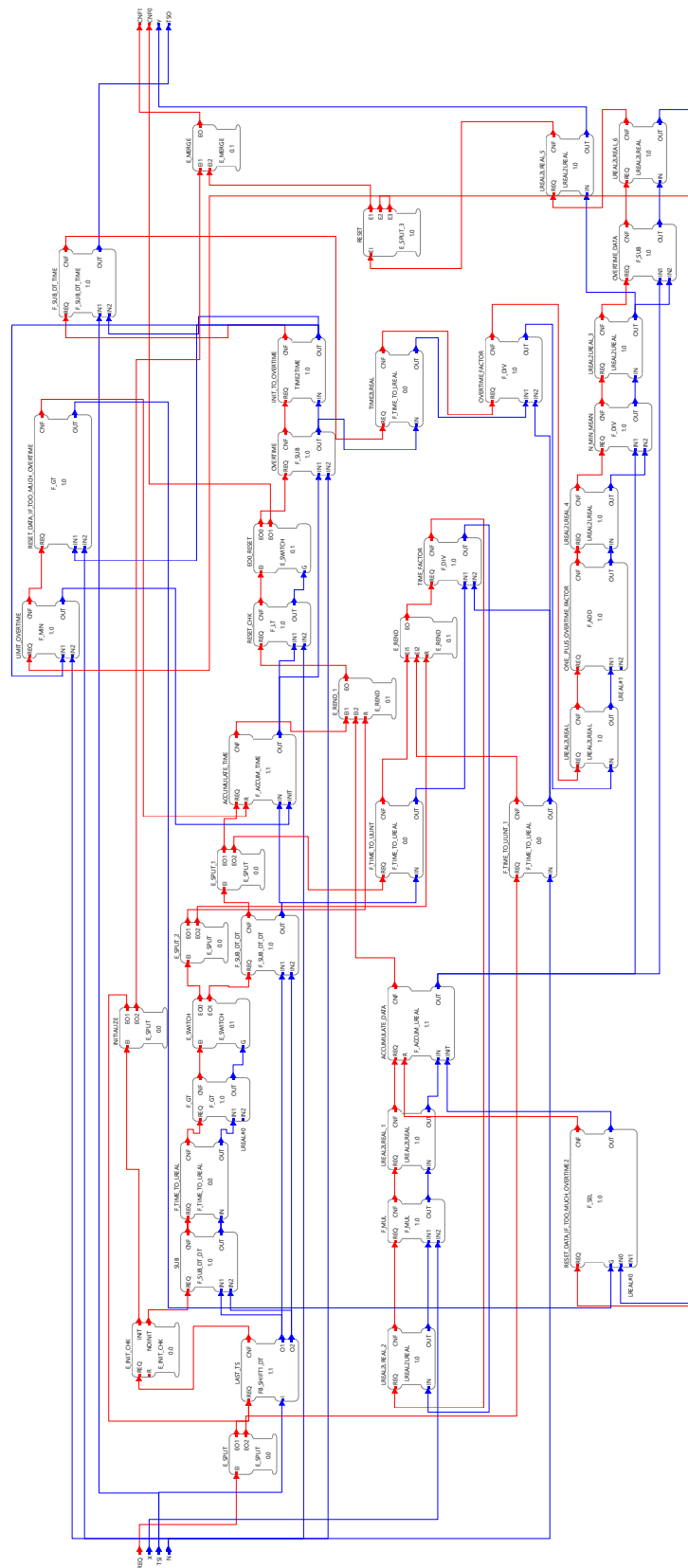

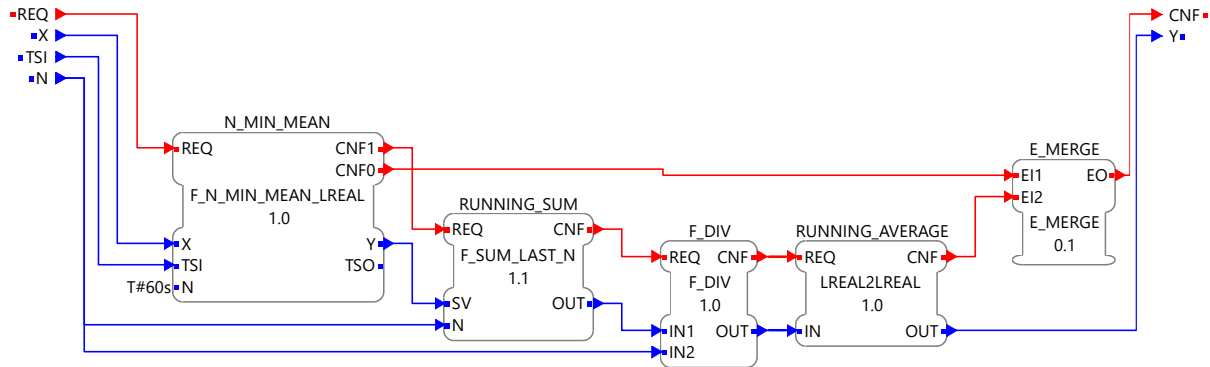**Figure A.3:** *Composite network of the* `F_N_MIN_MEAN_LREAL` *FB.*

Figure A.4: *Composite network of the F_N_MIN_RUNMEAN function block.*

PV curtailment utility function blocks

The F_N_MIN_RUNMEAN CFB (figure A.4) computes an estimation of the n min running average of the FB's input X using the accompanying time stamps. It uses the F_N_MIN_MEAN_LREAL to compute 1 min averages that are issued every minute and caches the last n values in the F_SUM_LAST_N BFB, which outputs the sum thereof. Finally, the intermediate result is divided by the number of minutes n to average over. As the composite network illustrates, an output is generated for every input event. However, the running average is only updated every minute.

Statutory declaration

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. I am aware that the violation of this regulation will lead to failure of the thesis.

Berlin, 6th October 2017