FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integration des Functional Mockup Interfaces in IEC 61499-basierte Komponenten

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Michael H. Spiegel
Matrikelnummer 1125727

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Wolfgang Kastner
Mitwirkung: Univ. Ass. Dipl.-Ing. Günther Gridling

Wien, 5. Oktober 2015

Michael H. Spiegel	Wolfgang Kastner

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integrating the Functional Mockup Interface into IEC 61499-based components

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Michael H. Spiegel
Registration Number 1125727

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Wolfgang Kastner
Assistance: Univ. Ass. Dipl.-Ing. Günther Gridling

Vienna, 5th October, 2015

Michael H. Spiegel	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Michael H. Spiegel
Wurzbachtalgasse 25, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. Oktober 2015

Michael H. Spiegel

Acknowledgements

I would like to thank the Austrian Institute of Technology, which founded the presented work and Prof. Wolfgang Kastner and Dipl.-Ing. Günther Gridling representing the Automation Systems Group who supervised this thesis and enabled a fruitful cooperation. Furthermore, I would like to thank Fabian Leimgruber, who supervised my work at the Austrian Institute of Technology and who always provided valuable and supporting feedback. I would also like to express my gratitude to my former colleagues at the Austrian Institute of Technology, in particular Johannes Kathan and Dr. Edmund Widl. Thank you for supporting my work and for many inspiring and fruitful discussions we have had during the last months! Last but not least I want to thank Stefanie Berger, who spent hours in proof reading this thesis.

Kurzfassung

Zahlreiche technische Prozesse werden mittels industrieller Steuerungen kontrolliert. Der IEC Standard 61499 spezifiziert eine Beschreibungssprache zur Konfiguration von verteilter Automatisierungsinfrastruktur auf der Systemebene. Sein Anwendungsfeld erstreckt sich von der Steuerung produktiver Systeme bis zur Modellierung von Steuerungsaspekten in gekoppelten Simulationen. In diesen Simulationen werden derzeit IEC 61499-basierte Steuerungen meist mittels programmspezifischer Schnittstellen mit Simulationsmodellen gekoppelt. Um den Simulationsprozess auf domänenspezifische Aspekte zu fokussieren und die arbeitsintensive Kopplung der Programme zu erleichtern, ist die Verwendung von standardisierten Schnittstellen wie dem Functional Mockup Interface anzudenken.

In dieser Bachelorarbeit wird eine neuartige vorhersagebasierte Methode zur Kopplung von ereignisbasierten Komponenten mittels Functional Mockup Interface vorgestellt und auf IEC 61499-basierte Regelungen beziehungsweise Steuerungen angewandt. Die Arbeit dokumentiert die dafür notwendigen Anpassungen der Methode und erweitert diese für den Einsatz in Echtzeitsystemen. Weiters wurde der vorgeschlagene Ansatz zur Übertragung der Ausgaben des Simulationsmodells an die eingesetzten Regelungen beziehungsweise Steuerungen implementiert und evaluiert. Dieser Ansatz zeichnet sich durch eine Reduktion der ausgelösten Ereignisse sowie durch eine zeitnahe Kommunikation aus. Neben algorithmischen Details und der eingesetzten Softwarearchitektur werden die Testanordnungen sowie die experimentell ermittelten Ergebnisse der Evaluierung vorgestellt. In einem solchen Experiment werden die Ausgaben eines Simulationsmodells an einen IEC 61499-basierten Wechselrichter-Teststand zeitgerecht übertragen und die aufgenommene Leistung gemäß der Vorgaben angepasst.

Abstract

Numerous technical processes are controlled by an automation infrastructure. The IEC 61499 standard provides a system level design language for complex distributed automation systems. It may be deployed in a productive environment or used to model control related aspects in a co-simulation setup. State of the art tool coupling approaches link IEC 61499-based controllers via simulation tool-specific interfaces. Labor intensive tool coupling may be avoided by using standardized interfaces such as the Functional Mockup Interface. This thesis presents a novel predictive approach which may be used to couple event-based components via the Functional Mockup Interface. It applies this approach to IEC 61499-based controllers. Additional modifications were made in order to allow a soft real-time operation. The implementation and evaluation of the presented approach are also discussed in this thesis. Furthermore, this thesis deals with algorithmic details and the implemented software architecture. The prediction-based approach reduces the number of triggered events and precisely adapts to the timing of the model. First experiments which include an inverter test-stand show promising results. The outputs of a test model were successfully transferred to the test-stand hardware.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

A broad variety of technological domains require a flexible, scalable and dependable automation infrastructure. It has to efficiently control the technical processes of the domain which range from chemical plants to electricity grids. For instance, future manufacturing processes are expected to require advanced Information and Communication Technology (ICT) and automation systems [18, 61]. They support upcoming trends such as mass customization in manufacturing. First experimental factories show the benefits of a flexible automation infrastructure and produce customized products at the cost of mass goods [42].

The electric energy domain, likewise, has to perform fundamental transitions in order to face grand societal challenges such as combatting climate change [22, 61]. Fossil fuel-based electricity generation has a significant share in total man-made greenhouse gas emissions. The energy supply sector globally accounts for approximately 26% of the total greenhouse gas emissions [37]. In order to reduce the anthropogenic greenhouse gas emissions, renewable energy sources need to be integrated into the energy systems [22, 68, 69]. Such energy sources are, for example, solar Photovoltaic (PV), wind generators, and biomass. The variability of many renewable energy sources is a major challenge [68]. It requires the development of flexibility options such as demand-side response and energy storage. Traditional electricity transmission and distribution networks are not designed to handle a high number of renewable and distributed electricity producers [22]. Smart grids are developed in order to face that challenge. They enable demand response and the large scale integration of variable energy sources [22, 70].

The International Electrotechnical Commission (IEC) standard 61499 specifies a system level design language for distributed automation systems [6, 41, 42]. It has shown its benefits in many technical domains such as manufacturing and power systems [10, 23, 42, 44, 51, 53]. The IEC 61499 provides the required flexibility and reusability and may be used to implement advanced control strategies. Some authors have already shown that it is beneficial to use the same IEC 61499-based controls both during design as well as in the field [22, 44, 51]. Due to the flexibility of the standard the same control

applications can be used in a virtual testing setup and in a productive environment. The actual implementation of the control logic does not have to be changed.

Note that some parts of this thesis have already been published at the proceedings of the workshop on modeling and simulation of cyber-physical energy systems [43]. The conference paper was created under the kind supervision of Fabian Leimgruber, Edmund Widl and Günther Gridling.

1.1 Motivation

The development of cyber-physical energy systems or sophisticated manufacturing systems, for instance, requires advanced simulation and testing capabilities [22, 61]. Frequently, different specialized tools and models have to be coupled in order to accurately model the behavior of a system [10, 44, 53]. For example, a smart grid model may include one tool which models the behavior of a grid. Another independent tool may implement the control logic of a grid component. Both tools have to be coupled in order to gain a holistic view.

In the energy domain, development and testing is rarely constrained to pure virtual simulations [22, 64]. Controllers which run on their target hardware and real power hardware may be coupled with (real-time) simulations to verify their operation. Depending on the Hardware under Test (HuT), different interfaces are needed to couple various kinds of hardware. Whilst a controller only needs low energy interfaces such as a network connection to transmit the control information, power Hardware-in-the-Loop (HIL) experiments require hardware interfaces which emulate high power components such as PV arrays or distribution grids. For instance, the inverter test-stand presented by Andrén et al. features a Direct Current (DC) supply which mimics a PV array, a connection to the local power grid, and an adjustable local load [22].

1.2 Problem Statement

A professional and seamless tool coupling work-flow reduces the overhead of coupling different simulation tools. It allows to focus on domain related aspects instead of interface implementations. Existing model and tool coupling approaches in the IEC 61499 domain often require tool or application-specific interfaces [10, 44, 51, 53]. Most tools aiming at simulation do not implement communication standards which are used in the automation domain [71]. Hence, tool-specific interfaces are deployed.

The FMI standard provides tool-independent interfaces to couple simulation tools and models. Several simulation tools support the FMI interface but no FMI enabled IEC 61499-based Runtime Infrastructure (RTI) is known [7]. It is expected that interaction between simulations and IEC 61499-based controllers via an FMI enhances the development of many energy related cyber-physical systems.

Since many automation infrastructures also include a Supervisory Control and Data Acquisition (SCADA) system which provides a user interface, coupling may also be done at the SCADA level. A SCADA level integration may not only couple the automation

infrastructure with FMI-based models but may also generate new prospects in usability, flexibility and the need for additional infrastructure. Such a SCADA system which interfaces a plant model may execute without any control-level implementation at all. Similar to IEC 61499-based RTIs, no FMI enabled SCADA system is known.

Time management and time synchronization is a crucial part in coupling simulation tools and models [10]. The notion of time of a tool needs to be synchronized in order to get meaningful results. Many RTIs, SCADA systems and most HuT execute in real-time only [9, 31, 46]. Hence, the implemented model or tool coupling setup has to have real-time capabilities.

One possible use-case is the automation of the test-stand presented by Andrén et al. [22]. In their setup, the local load is controlled by an IEC 61499-based controller and a SCADA system. Currently, the load set-points are entered statically and mostly in-time. A model or tool interaction may automate the experiments and may provide dynamically generated set-points. Other use-cases include controller HIL and pure virtual co-simulation experiments.

1.3 Main Goals

First, this thesis shall briefly summarize work which is related to tool and model coupling, focusing in particular on smart-grid related literature and use-cases. Then, it will describe the implementation and evaluation of an FMI-based model coupling approach. In order to couple IEC 61499-based controllers or SCADA systems with FMI-based models, a novel model coupling approach which was introduced by Müller et al. and Widl et al. is adopted [13, 27]. The approach uses a prediction-based methodology which computes future events. The adoptions should target a soft real-time operation of the interfaced components. Furthermore, the algorithm may be altered in order to couple IEC 61499-based controllers. To couple IEC 61499-based controllers as well as SCADA systems, an intermediate protocol may be used.

The implemented software design must follow the modified prediction-based approach. It targets maintainability, usability, and portability aspects. The intended software life-time is set beyond the scope of this thesis. Future modifications have to result in adequate implementation effort. Every developed interface software provides unit test-cases which test its functionality. The unit test-cases shall target the boundary values of the program instead of documenting the intended operating conditions. The implementation shall be evaluated in terms of its timing capabilities and its expected accuracy. The evaluation shall include the test-stand hardware and simple test models. Additional experiments may be conducted to further evaluate the implementation.

1.4 Methodology

First, the chosen approach which includes necessary modifications of an existing approach is shortly described. Based on the approach, an interface software is designed. The design fulfills the main goals described in section 1.3. Following the software design, the

interface software is implemented. The software components are tested by automated unit test-cases which check the program functionality.

Based on the prototypical implementation, the capabilities of the modified approach are evaluated. Therefore, a dedicated test infrastructure is used. The evaluation is based on two simple test models which are accessed via FMI, a local IEC 61499-based test application, and the test-stand infrastructure. The timely behavior of every setup is evaluated based on different sets of timing records. Power measurements which are captured by the test-stand and the outputs of the local test application are used to evaluate the simulation results.

Due to the limited scope of this thesis, only a one directional information flow is implemented and evaluated. The outputs of a model are sent to the IEC 61499-based controllers but the model currently does not receive information from the IEC 61499-based infrastructure. Nevertheless, the implementation tries to prepare future extensions which allow a closed loop operation.

1.5 Structure

Chapter 2 summarizes the state of the art. It discusses work related to IEC 61499, co-simulation and model exchange interfaces as well as work related to HIL setups. Additionally, work concerning IEC 61499-based controllers in co-simulation setups and system models in IEC 61499-based controls is described. Chapter 3.3 shortly states the theoretical background and introduces the implemented approach. The implementation of an interface component which enables IEC 61499-based controllers to utilize simulation models is described in chapter 4. Chapter 4 also describes the user interface, the software architecture and lists deployed third party software. The implemented test-cases and the evaluation of the approach are described in chapter 5. Chapter 6 concludes the findings and lists open questions which exceed the scope of this thesis.

CHAPTER 2

Related Work

2.1 Discussion of the IEC 61499

The IEC standard 61499 defines a system level design language for distributed automation systems [6, 41]. It is available in the second edition and has been developed for distributed, modular, (re)configurable, and flexible control systems [6, 52]. The IEC 61499 extends the Function Block (FB) concept of the well established IEC 61131-3 by an event handling mechanism [52]. Additionally, FBs are extended to provide a finite automaton named Execution Control Chart (ECC) which controls the execution of the algorithms of a FB [6, 41]. The FB concept provides the ability to encapsulate algorithms and data into robust and reusable components [6, 52] resulting in an event-based and object-oriented engineering approach. To gain interoperability between different devices and vendors, an Extensible Markup Language (XML) format which stores system-configurations is also defined by the IEC 61499 [41].

First professional IEC 61499-based solutions are now available and various publications discuss different implementations, expected advantages and the integration of concepts from computer science [42]. Special attention is put to several ambiguities, mostly in the first edition of the standard. These ambiguities lead to different interpretations of the defined execution models and often to incompatible system designs [41, 42, 52]. One of the spotted ambiguities refers to the lifetime of event-input variables [41]. When clearing event-variables immediately after their first use, transitions within the ECC might get lost, while keeping them after the first positive transition might trigger undesired transmissions.

Another issue related to the IEC 61499 is the execution order within compound FBs which consist of other FBs. The execution order is not entirely defined by the abstract event-flow model [41, 52]. Different scheduling schemes, including sequential scheduling using global or local queues, cyclic and parallel executions are feasible. Miscellaneous scheduling schemes could lead to different results, especially if an input variable content is latched more than once. Also, real-time parameters like the responsiveness vary

significantly for different execution models [52]. Attempts have been made to address the ambiguities by developing compliance profiles which further specify the execution model [29, 41].

2.2 Using IEC 61499-Based Controllers in Co-Simulation

IEC 61499-based controls have been successfully coupled with simulation tools to model control-specific behavior. Yang et al. presented a proxy-based approach simulating the execution time of a controller [10]. The proxy-based approach does not require any real-time simulation to run the model. Instead, the control reaction is delayed according to the simulation time and the estimated reaction time. On each variation of a simulated value in the physical domain, a control event is generated. This event triggers the execution of the IEC 61499-based control and the result is fed back using the time-proxy.

Stifter et al. presented a co-simulation framework for electrical power systems, which relies on open-source software [44]. In the presented setup, GridLAB-D takes control over the simulation. Some battery models are added using the FMI export feature of OpenModelica. The PSAT toolbox running on GNU Octave was integrated using a thin wrapper accessing the data of GNU Octave. Additionally, the IEC 61499 controller 4DIAC FORTE was accessed using TCP/IP-Sockets and ASN.1. Strasser et al. presented a similar approach simulating an under-load tap changer and its controller using GNU Octave and 4DIAC FORTE [53]. The simulated voltage of the secondary winding of the transformer is directly passed on from GNU Octave to the controller using a TCP/IP-connection too. The controller returns the resulting tap position and the next simulation step is performed.

Another application of co-simulation was presented by Strasser et al., who introduced a co-simulation training platform tailored to smart-grid applications [51]. In contrast to other work, the platform is used to educate students and power systems professionals in increasingly complex smart-grid applications. The software tool PowerFactory implements the grid model of the platform and acts as the simulation master. PowerFactory provides several interfaces like Dynamic-Link Library (DLL), OPC or a MATLAB/Simulink interface used to include other simulation tools and models. A control logic of a test-stand and a Human Machine Interface (HMI) was also implemented by 4DIAC and the SCADA system ScadaBR.

Although the reviewed papers presented approaches to couple separate IEC 61499-based controls with co-simulation environments, additonal HuT was not included. Also, the application of an automation infrastructure which emulates power grids was not further discussed in this context. The communication between the simulation software and the controller was implemented using TCP/IP- or UDP/IP-connections in conjunction with application-specific interfaces. Because of the high level of necessary adoptions, these interfaces cannot be easily used to couple additional software tools with the deployed controllers. The application of co-simulation specific interfaces, like the FMI [65] to couple IEC 61499-based infrastructure was not demonstrated.

2.3 Using System Models in IEC 61499-Based Controllers

Though some work has been published on using IEC 61499-based controls within co-simulation environments, there is not much focusing on integrating plant simulations within IEC 61499-based architectures. Hegny et al. presented an approach which runs production plant simulation models on IEC 61499-based controls [20]. They proposed a common automation component model to facilitate an integrated model-driven engineering approach for industrial automation. The automation component model uses timed state charts, a subset of Unified Modeling Language (UML) state chart diagrams, as a general plant model. To run a hierarchically structured plant model, it is translated into an IEC 61499 application using the Eclipse Modeling Project tools. Each automation component is mapped to FBs utilizing the ECC and delay FBs to implement timed behavior.

Integration of physical plant models and the automation control logic is proposed based on a modified version of the layered model-view-control design [19]. The layered model-view-control design pattern utilizes a plant model and a view component providing a graphical display of the data [45]. After the validation of the controller component, the model and the view components will be replaced by interface components. The interface components access the physical system instead of the model data. Replacing the model should be eased by specifying the interfaces of the plant before starting the controller design [19]. Based on the FB-type, the actually instantiated implementation is chosen by the runtime environment.

Advantages such as further using existing designs are seen in integrating domain specific tools [18, p.59ff]. The author identified three different ways of extracting information from legacy applications. The first solution, directly implementing the export functionality within the application, can be considered to be optimal. Other ways include external model-to-model transformations which optionally use an intermediate vendor-neutral data format. Transformation of automation component models back to legacy models is discussed as model-to-model or model-to-code transformation which is also facilitated by the proposed architecture. The usage of external models without transforming them into the automation component model is not discussed in detail [18, 19, 20].

Some literature related to co-simulation which is presented in section 2.2 uses independent components to run the control logic and the simulation [10, 44, 51, 53]. Although the focus is put on the co-simulation point of view, some results are applicable to the automation point of view as well. These results include the communication paradigms used between the co-simulation environment and the IEC 61499-based controller. It is important to note that the schedule and control of the simulations are up to the co-simulation frameworks which limits the applicability to time-unaware cases.

2.4 Co-Simulation and Model-Exchange Interfaces

Coupling one or more simulation models and simulation tools to gain a more comprehensive simulation has already been successfully demonstrated [2, 10, 21, 44, 51, 53].

Coupling is done by tool-specific interfaces on the one hand and open standards like the FMI or High Level Architecture (HLA) on the other hand. The FMI standard defines an interface for coupling two or more simulation tools [65] and for sharing models between different simulation tools [66]. The HLA not only defines the interfaces but also a common integrated architecture and a general framework which addresses interoperability and reuse of different types of simulations [15].

The FMI was initiated by Daimler AG, first released in 2010 and recently updated in 2014 by the MODELISAR consortium [60, 65, 66, 67]. The first version defines two main parts, the FMI for model exchange and the FMI for co-simulation. The former part defines interfaces to couple single simulation models without a dedicated solver. The latter part defines interfaces for coupling two or more simulation tools which include their own solvers. The second version of the standard combines the two main parts into a single standard document and defines several new features [67].

Single simulation components using the FMI standard are encapsulated into Functional Mockup Units (FMUs), a zip-compressed archive containing all files needed [60, 65, 66]. An XML file which is included into the FMU describes all variables exposed to the environment, co-simulation tool capabilities and other model information required. The actual interface calculating the model equations or communicating with the co-simulation tool is expressed by a small set of C-functions which could also be provided as binary files. For model exchange, ordinary differential equations in state space form including discontinuities are supported [60, 66]. Discontinuities are represented by events indicating a sudden state change. Exported models usually do not include their own solvers. Numerical integration has to be provided by the simulation tool which imports the model.

In contrast, the FMI for co-simulation requires the FMU to integrate its own solver and restricts communication to discrete communication points only [60, 65]. A master algorithm which is not specified in the standard controls the data exchange between single FMUs and manages the simulation time. The FMI for co-simulation allows several different levels of tool integration. Such levels include one process, one machine, and fully-distributed co-simulations. In each case only the C-function-based interface without any communication protocol is specified by the FMI.

The other aforementioned standard, the HLA, was defined in the Institute of Electrical and Electronics Engineers (IEEE) standard 1516. It was first released in 2000 and revised in 2010 [15, 21]. The standard refers to a collection of interacting simulation tools as federation and to each single tool as federate. Federates specify the information they could provide by a simulation object model. The information exchanged within a federation has to be specified using a federation object model. Additionally, a management object model is provided to monitor and control the execution of the federation. Objects within the context of HLA are collections of attributes which could be defined by XML-based object model templates.

Similar to the FMI, the HMI also supports hybrid simulations combining time-discrete and event-discrete approaches. A dedicated RTI controls the execution of federations, handles the communication between federates and manages the simulation time [15, 21]. The HLA not only specifies the object model declaration but also the services to

be implemented by the RTI and rules for proper interaction between federates. Beside programming language independent service definitions, the HLA also provides C++ and Java interfaces as well as Web Services Description Language (WSDL) bindings.

Müller et al. describe a HLA-based co-simulation framework tailored to the needs of smart-grid applications [21]. The presented framework called INSPIRE integrated an ICT simulation tool and a power system simulation tool. The scenario description which is given as an IEC 61970 compatible structure is parsed and translated to provide HLA object models. INSPIRE also logs system states during the execution of the simulation and provides tools to examine the simulated system. The paper also presents a use-case of INSPIRE which demonstrates the dependencies between the ICT and the physical domain.

A combination of the FMI and the HLA was first presented by Awais et al. [28]. They proposed two algorithms integrating exported FMUs into a HLA and evaluated them empirically. A naive approach simply increasing the simulation time by a fixed lookahead-value and integrating the model provided by the FMU will fail if events are generated too fast. Instead, both algorithms are based on a zero-lookahead simulation-based approach. The first one assumes a fixed step size and the second one is capable of using variable step sizes. Both algorithms were able to operate without losing any event; however, the algorithm using a variable step size generated significantly less events which improved the simulation performance.

Strasser et al. discussed the applicability of open standards in smart-grid applications including different domains [53]. They stated that IEC 61850 only specifies an information model and the transmission of that information. The device functionality needs to be described using other standards such as IEC 61499.

Andrén et al. identified several open issues within smart-grid model and design approaches and envisioned a comprehensive smart-grid information model including physical, communication, control, and application domains [23]. They argued that closer cooperation of simulation models and real power systems resulting in a multi-directional information flow between planning, simulation/validation and operation is needed to face upcoming challenges. A comprehensive smart-grid information model supporting the process is not available so far. The authors proposed a holistic smart-grid model addressing this issue. The model should be based on existing domain models such as the ones defined in IEC 61499 and IEC 61850. When integrating multiple domain models into a single holistic model, information shared between different domains needs to be synchronized. It is proposed to access single domain models via domain specific views, which display only domain relevant information. The authors concluded that the approach should be beneficial for long operational power systems as well as rapidly changing smart-grids.

Another approach which provides tighter tool integration was presented by Biffl et al. They introduced the concept of an automation service bus [25]. The automation service bus should bridge existing gaps between models and tools in the engineering process. It is based on the enterprise service bus and uses message oriented middleware to provide a communication infrastructure between several involved tools. The authors demonstrated

the concept based on an enhanced engineering work-flow. It includes automatic model updates, build and test processes as well as automated issue tracking.

2.5 Hardware-in-the-Loop and Real-Time Testing

In HIL setups, stability of the system is a necessary, accuracy a sufficient condition [64]. The theoretical groundwork of testing and analyzing stability and accuracy is given by means of system theory [16, 17]. Many works which summarize the theoretical basis in a general manner already exist [16, 17, 36, 59]. Viehweider et al. applied system theory to stabilize Power-HIL setups [64]. The power amplifier driving the HuT and the HuT itself are approximated by linear models. To couple the simulation with the HuT, a current/voltage-source pair is used. The current source is virtually connected to the simulation model while the amplifier acts as the voltage source. Each source interacts with the other forming a closed loop control system. The effects of hardware inductance addition, feedback current filtering and multi-rate partitioning in stabilizing the system were analyzed and evaluated. All methods are able to stabilize the system but adding a hardware inductance decreases accuracy significantly. Multi-rate partitioning as well as increasing the overall sampling rate showed the best results.

Guo et al. demonstrated the capabilities of real-time simulators in simulating a small community microgrid [2]. They put special emphasis on the detailed emulation of the communication network by using a dedicated network simulator. It was shown that the four deployed real-time simulators are able to simulate numerous switched devices with over 100 single switches at 10 kHz switching frequency. Each simulator in the presented setup is interconnected via analogue and digital I/O-lines. Additionally, they are connected via a network simulator. The authors concluded that the communication network enables new protection schemes but adequate network simulation is necessary to predict the components' behavior accurately. The integration of real HIL was not part of the presented work.

2.6 Contribution

The integration of IEC 61499-based controls in co-simulation has been successfully shown [10, 44, 51, 53]. However, the integration of simulation models which do not provide their own solver into IEC 61499-based controllers is still rare. Although Hegny et al. presented an approach of modeling automation infrastructure within IEC 61499, the usage of continuous state space models was not covered [18, 20]. Some standards for co-simulation and model exchange are already available [15, 65, 66, 67] but the cooperation of such a standard with IEC 61499-based controllers has not been shown. As a consequence, previous work regarding the integration of models and simulation tools into IEC 61499 applications relies on tool-dependent interfaces.

HIL and Power-HIL setups have already been studied from many points of view [50, 64] but IEC 61499-based controllers could not be found as an intermediate link between the HuT and the simulated environment. The absence of IEC 61499-based controllers

in HIL setups might be based on the lack of feasible interfaces to common simulation environments. However, IEC 61499-based controllers are still in use to control HuT in a human-operated test-stand [51].

This bachelor thesis addresses these issues by adopting an FMI-based predictive model coupling approach [13, 27]. It uses the FMI standard to represent a broad variety of different simulation tools without the need of dealing with tool specifics. Special emphasis is put on HIL setups and necessary features. In addition, the results are demonstrated by implementing and evaluating the adopted approach.

CHAPTER 3

Theoretical Background

3.1 IEC 61499 Component and Execution Model

The main objective of the IEC standard 61499-1 is to provide a reference model for distributed Industrial-Process Measurement and Control Systems (IPMCSs) [6]. It defines several different entities forming an IPMCS. The function of a system is modelled as a possibly distributed application consisting of subapplications and a hierarchy of FBs.

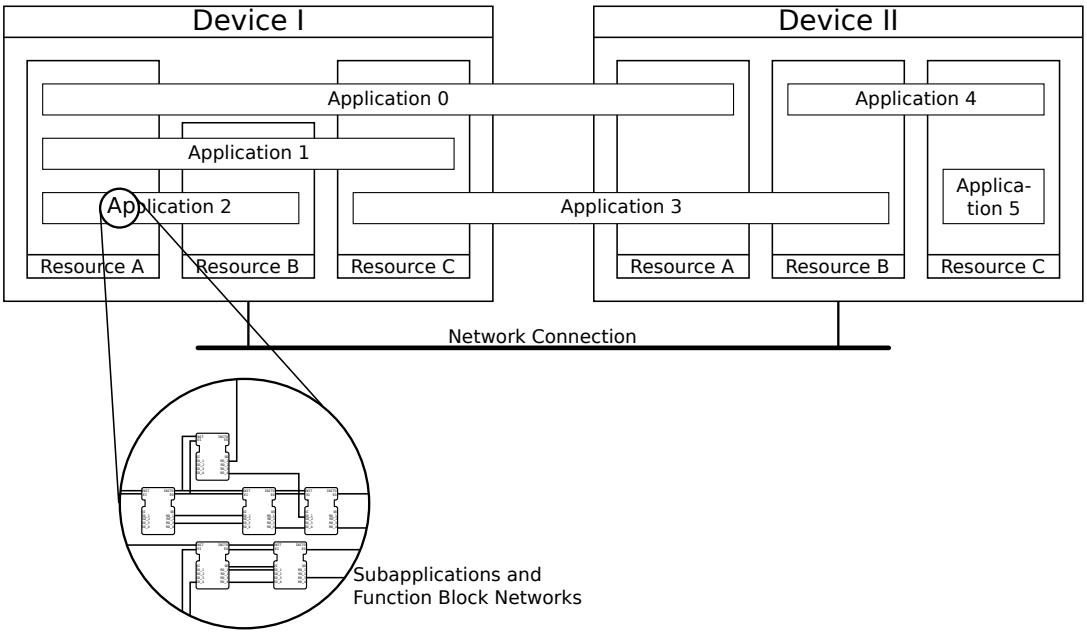


Figure 3.1: IEC 61499 Device Model Example

Applications may be distributed along multiple resources contained within a specific device. A resource is described as "a functional unit, which has independent control of its operation" [6, p.21] containing one or more local applications or local parts of applications. Figure 3.1 provides an example configuration illustrating the application, resource and device mapping.

The execution of function blocks is scheduled on the resource level based on the resource scheduling function. Scheduling is determined by the occurrence of events, FB interconnections and additional scheduling information such as priorities. Each FB instance is a named copy of a specific FB type. FBs may be interconnected within FB networks by connecting FB in- and outputs. Two types of in- and outputs exist, one passing events on and one transferring data between FBs. If an event is passed on to the event input of a FB the FB will be scheduled for execution. It is possible to associate data in- and outputs with event in- and outputs respectively. If an event is triggered, data will be provided at the associated in- or outputs.

The IEC 61499 defines three major classes of FB types: basic-, composite- and service interface FB types. Basic FBs encapsulate a set of algorithms and local variables and provide a set of in- and outputs [6, p.30ff]. The execution of algorithms and the emittance of events is controlled by an ECC, a finite state automaton. Each state may be associated with a set of actions specifying the algorithm executed on entering the state. An associated action may also include an event to be triggered on completing the algorithm. Transitions between states have a transition condition containing an event reference, a guard condition or both. If the transition condition is fulfilled, i.e. the guard condition is true and the event occurred, the corresponding state will be entered. Such a state transition is called crossing a transition. The FB execution stops if no transition can be crossed. The specification of rules necessary to describe the algorithm is beyond the specification provided by IEC 61499 but FBs as defined in IEC 61131-3 may be used [6, p.84].

Composite Function Block (CFB) types contain a set of interconnected component FB types, called FB network, which defined the behaviour of the CFB. The declaration of CFB types follows the rules of basic FB types, except that the event in- and outputs of the CFB may be connected to the out- and inputs of the component FB [6, p.35ff]. The event connections represent the sequence of FB invocations inside the CFB.

The third major class of FB types defined are Service Interface Function Block (SIFB) types, whose instances provide service to the application [6, p.44ff]. SIFBs are based on abstract interactions between applications and resources which are independent of the actual implementation. These interactions are called service primitives. The behaviour of SIFBs is described by sequences of linked service primitives called Service Sequence Diagrams (SSDs). SIFBs are commonly used to access functionality such as process and communication interfaces provided by the underlying infrastructure. The IEC 61499 also defines two special kinds of SIFBs, communication FBs and management FBs. Communication FBs provide interfaces between applications and communication mapping functions of a resource. They can be used to exchange information between different applications or between different parts of an application. Management FBs may

be used to perform application management functions such as creating, starting and stopping specific entities.

An application or subapplication may be distributed among different devices by allocating the used function block instances to different resources [6, p.27]. FBs are specified to be the basic unit of distribution. As a consequence FBs contained within a CFB reside in the same associated resource. It is not allowed to use subapplications within CFBs.

3.2 Functional Mockup Interface Model

The first version of the FMI protocol suite contains two separate interfaces, the FMI for model exchange [66] and the FMI for co-simulation [65]. The first interface is used to exchange dynamic system models between simulation tools and the second interface couples independent simulations. In addition, a second version of the FMI standard exists, combining the two former standards in one document [67]. The second version introduces several improvements and new features; however, due to the fact that it has been released in June 2014, many tools do not yet support it. As a consequence the theoretical discussion provided will focus on the first version of the standard. Additionally, remarks pointing out relevant differences between the FMI versions may be provided.

Some parts such as the definition of input and output variables are commonly used in the two interfaces of the FMI standard but some parts such as the model representations differ significantly. In contrast to the discrete event-based execution model of IEC 61499, FMI supports the simulation of piecewise continuous systems. Time is expressed in the first version of the FMI by a dense time base. The second version uses a super dense time base additionally enumerating events at a given instant in time [67, p.69]. In FMI terms points of discontinuities are also called event but to explicitly distinguish them from IEC 61499-events, they will be called FMI-events. Different types of FMI-events including state and time events exist but every FMI-event indicates a possible discontinuity in the output or state derivative function.

Both FMI interfaces call a single component encapsulating a model or a co-simulation tool an FMU instance. FMUs are exchanged as zip-compressed archives containing all relevant information such as input and output descriptions as well as executable interface code. Static information such as the number and type of in- and outputs as well as model parameters are represented in an XML file within the FMU. Functions calculating the model equations may be provided as C-code or by platform dependent executable binary files. Since this thesis focuses on FMI for model exchange, FMI for co-simulation is not described in detail.

The FMI for model exchange represents systems as Ordinary Differential Equations (ODEs) in state space form [66, p.9ff]. Each FMU provides a set of C-functions to access input and output variables, states and their derivatives as well as setting the current simulation time. A dedicated solver, provided by the software including the FMI, has to solve the ODE numerically. FMI for model exchange defines three types of events: time, state and step events. The instant of time when the next time event occurs has to

be reported by the FMU in advance at each occurrence of an event. Step events may be triggered after the solver finishes an integration step. After issuing a step event it is not possible to set time instants prior to the step event. If, for example, a solver triggers a step event after an integration step is taken, then it must not recalculate the last integration step. State events use special indicator variables to determine the time instant when the event occurs. It will be triggered if an event indicator variable changes its domain. In this case, the solver has to perform an iteration over time to determine the precise event time.

The FMI for model exchange is designed to enable efficient caching of calculated values. Only changed inputs will be set and the FMU may evaluate only the equations necessary to calculate a certain output [66, p.18ff]. Both FMI versions specify a state machine defining admissible function calls. The second version of the FMI explicitly marks state transitions by function calls [67, p.83]. If the FMU is in an event mode it is generally not allowed to set a new time instant. In- and outputs are assumed to be continuous in a continuous mode of operation.

It is optionally possible to declare direct dependencies of input and output variables within the descriptive XML file. If no dependency information is given it is assumed that an output directly depends on every input [66, p.37f]. On connecting FMUs with direct dependencies the solver has to deal with possibly occurring algebraic loops [66, p.48ff]. Within an algebraic loop an output of an FMU directly depends on the output of another FMU and vice versa. The resulting system of algebraic equations has to be solved in order to obtain correct outputs.

3.3 Predictive Model Coupling

Müller et al. and Widl et al. introduced a novel predictive approach which couples FMUs and discrete event simulations [13, 27], which uses the FMU to pre-calculate future FMI-events. These events can be relayed accurately to the coupled tool and model. Hence, no artificial delay has to be introduced. In contrast, conventional approaches may deploy a periodic synchronization and may delay events until a common synchronization point is reached. The prediction first assumes that no external event is triggered. If an event is issued to the FMU, the predictions are still valid until the issued event time. After the event time, all predictions have to be invalidated. Invalidated predictions have to be re-calculated and new events may be scheduled.

The approach is applied to the IEC 61499 by translating IEC 61499-events and FMI-events. Every FMI-event is directly mapped to an IEC 61499-event which is associated with the output variables of the model. If no FMI-event is predicted within a certain time-span, an IEC 61499-event is also scheduled at the end of the prediction period. The period is called lookahead horizon. It assures that continuous model outputs which do not directly trigger an FMI-event will also be transferred. The lookahead horizon has to be chosen with respect to the properties of the model [13, 27].

Any incoming IEC 61499-event first has to be time-stamped. Secondly, associated data needs to be fetched and set in the FMU at the appropriate time. Since the IEC 61499 does

not provide a time-stamping mechanism [6], each event time has to be tracked externally. It may be recorded by the interface logic while the IEC 61499-event is first processed. In order to mark the discontinuity, every IEC 61499-event shall be directly translated to an FMI-event. The implementation of the IEC 61499 to FMI-event translation is beyond the scope of this thesis. However, the design also tackles future extensions such as bidirectional data flow.

IEC 61499-based controllers will process incoming events as soon as possible. FMI-events, however, are predicted beforehand. In order to apply the predictive approach in real-time environments, FMI-events need to be delayed appropriately. The system clock of the workstation which executes the FMU is used as a reference. Any event will be delayed according to that clock. Furthermore, the processing and queuing time of the IEC 61499-event is considered to be part of the system. It will not be compensated by additional means. Further details of the implemented approach and required modifications can be found in [43].

The IEC 61499 also specifies an Abstract Syntax Notation One (ASN.1)-based network protocol [6]. The protocol is capable of transferring IEC 61499-events and its associated variables. Additionally, it may be supported by SCADA systems. The interface logic which encapsulates the FMU is connected to one or more controllers by using the specified protocol. The protocol enables a vendor- and platform-independent implementation. It allows a separation of the hardware, which executes the controller, and the machine, which executes the model.

CHAPTER 4

Implementation

4.1 Use-Cases

Targeted use-cases which require a model-controller interaction include the development and testing of smart-grid components in an IEC standard 61499-based test-stand and the development of advanced control strategies in smart-grid related systems. The test-stand addressed in the first use-case features resistive, inductive and capacitive loads which are controlled by an IEC 61499-based controller. The time until new set-points are applied is up to several seconds [22, p.46]. Typical set-point adoption intervals range from few minutes up to several hours. Due to the large update time-spans, timing deviations of a few hundred milliseconds will not noticably affect the test outcome. Additionally, the test-stand is designed to tolerate usage failures and timing deviations without causing serious damage. The second targeted use-case only includes control equipment which is not capable of causing any damage. In this case large timing deviations may affect the outcome but do not cause any damage. Hence, both targeted use-cases can be met with a best-effort approach which records every timing deviation.

The test-stand features a static control infrastructure which is not designed to be changed frequently. A fully functional operation and high availability is required to provide an uninterrupted laboratory service. Both constraints imply that changes of the controllers should be kept at a minimum. It is expected that the included model will be changed much more often than the control infrastructure. A standard communication-based approach is used to decouple the execution of the controller and the model. The model is wrapped by an external program called FMITerminalBlock which executes it on a separate workstation. On changing the model only the wrapping program has to be reconfigured. The control infrastructure may remain unaffected.

At a first implementation step it is feasible to support FMI for model-exchange only. As a consequence, the software has to provide its own solver. As stated in section 1.3 a predictive approach is implemented. Events are triggered in soft real-time using the mechanisms of the operating system. Beyond that best-effort no real-time analysis of

the code was performed. Any real-time analysis would require sound assumptions of the timing of the model [43] which is beyond the scope of this thesis.

4.2 Program Flow

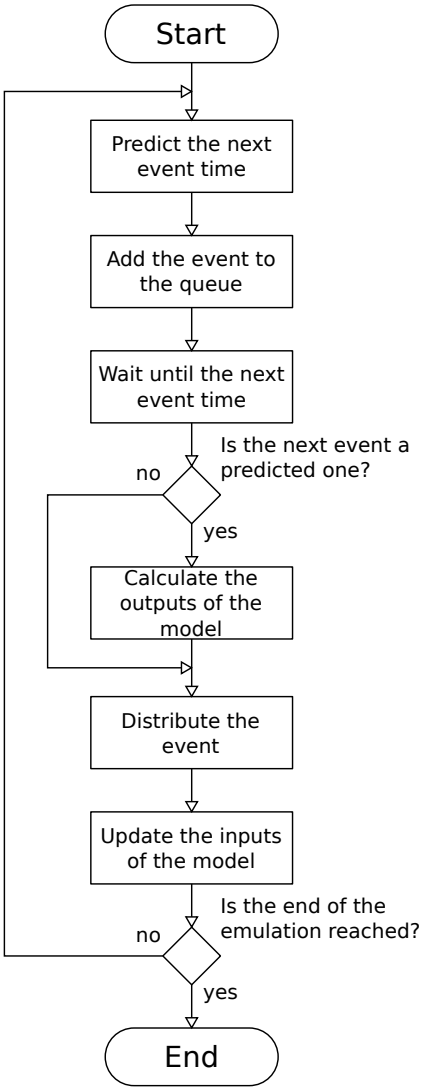
The intended program flow which follows the algorithmic concept was modeled as follows: It assumes that an abstract predictor is able to calculate future events which are triggered by the model [13, 27]. If the next event time exceeds a specified prediction horizon, an event at the end of the prediction horizon is returned. Every event is stored in a central event queue. The queue is used to determine the time of the next event and merges different event sources. In addition to predicted events, external events triggered by the IEC 61499-based controllers may be generated and have to be managed. Events are not limited to certain points in time. Hence, different event sources have to be operated in parallel. The central queue which merges different event sources has to be capable of receiving events concurrently. Although a hard real-time operation does not require an event queue which is capable of holding multiple events, it avoids losing late events in a best-effort approach [43].

Figure 4.1 visualizes the program flow of the interface software. It consists of different parts executed in parallel. The main thread shown in Figure 4.1a executes the model and distributes the events in time. One or more auxiliary threads detect incoming events. These threads also queue detected events in the central storage but return to a listening state immediately after reporting the detected event. Figure 4.1b briefly summarizes the operation of the event receiver.

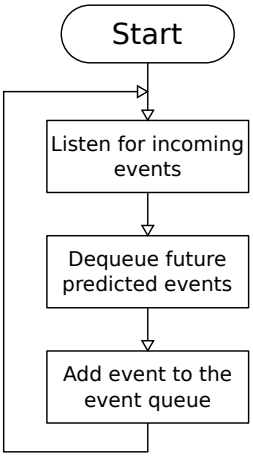
First, the program which includes the model has to be initialized. Initialization includes setting up the user interface such as parsing the user input and contains the initialization procedure of the model which is specified by the FMI standard [66, 67]. After the initialization step, every part of the program is started and able to perform its regular operation.

At the beginning of each event cycle, the next event time has to be computed. The predicted event is based on the assumption that no other event generated by an external event source is emitted. If the assumption fails, the predicted event will get invalid and the program has to repeat the prediction. In order to repeat a prediction, the discrete and continuous states of the FMU will have to be reset to the state at the external event time. The FMI for model exchange in its first version, however, does not expose the discrete states and hence does not allow to reset them to a previously calculated value [13, 27, 66]. Note that after predicting the next event time by observing event indicator changes or by stopping at previously known event times, the prediction logic must not execute the event update function. Calling the event update function potentially changes the discrete state of the FMU [66] which prevents a proper reset. The event update functions will have to be called after the FMI-event is actually taken.

As soon as the next event time is predicted, the event will be added to the sorted event queue and the main thread waits until the next event time equals the system time. At the waiting stage, any incoming event which is issued by an auxiliary thread will modify



(a) Main program flow



(b) Event receiver program flow

Figure 4.1: Interface Program Flow

the targeted wake-up time. The event may also release the main thread immediately. If, for example, the current real time instant is $t_0 = 3\,\mathrm{s}$ and the next event time is predicted at $t(e_{\mathrm{pred}}) = 5\,\mathrm{s}$, the main thread will initially try to sleep until $t' = t(e_{\mathrm{pred}}) = 5\,\mathrm{s}$. Let e_{ext} be an interrupting external event at $t(e_{\mathrm{ext}}) = 4\,\mathrm{s}$. On receiving the interrupting event e_{ext}, the wake-up time of the main thread will be altered to $t' = t(e_{\mathrm{ext}}) = 4\,\mathrm{s}$ and the predicted event will be invalidated.

If the next event returned after waiting is a predicted event, the output data associated with that event still needs to be calculated. It is important to notice that the data associated with an event corresponds to the limit from the right of the outputs of the model. Let $\mathbf{y}(t)$ be the output at time t and $\mathbb{V}(e)$ the associated variables of event e. Discontinuous outputs may only change its value $\mathbf{y}(t)$ if an event occurs [66, 67]. If and only if the FMI-event e_{pred} changes the outputs at the event time $t(e_{\mathrm{pred}})$, (4.1) holds.

$$\lim_{t \to t(e_{\mathrm{pred}})^-} (\mathbf{y}(t)) \neq \lim_{t \to t(e_{\mathrm{pred}})^+} (\mathbf{y}(t)) \tag{4.1}$$

As a consequence of (4.1), two potential sets of values may be associated with the event. The first one, $\mathbb{V}^-(e_{\mathrm{pred}}) = \lim_{t \to t(e_{\mathrm{pred}})^-} (\mathbf{y}(t))$ contains the limit from the left of the model output values and the second one, $\mathbb{V}^+(e_{\mathrm{pred}}) = \lim_{t \to t(e_{\mathrm{pred}})^+} (\mathbf{y}(t))$ contains the limit from the right. Since the values of an event are intended to trigger control actions, previously valid values are only of limited use. The program has to calculate and distribute $\mathbb{V}(e_{\mathrm{pred}}) = \mathbb{V}^+(e_{\mathrm{pred}})$ in order to obtain expected results.

Since the solver cannot trigger the event update function immediately after predicting the next event time, the discrete state of the model which is calculated during the prediction step is equal to the state just before the event is triggered. Requesting outputs at this point in time will approximately result in a variable set equal to $\mathbb{V}^-(e_{\mathrm{pred}})$. At the output calculation step, first the event handling functions are invoked. After the event handling terminates, the state reflects the changed discrete state and the outputs at $\mathbb{V}(e_{\mathrm{pred}}) = \mathbb{V}^+(e_{\mathrm{pred}})$ can be calculated.

On receiving an external event, the model has to be reset to the external event time $t(e_{\mathrm{ext}})$. During prediction, intermediate continuous states will be stored which allows to quickly reset the state of the FMU at the external event time [13, 27]. After resetting the state, new inputs received by the external event will be applied. Subsequently, the state is updated and further predictions based on the altered inputs may be done. Note that the current program flow assumes that external inputs will remain constant between two consecutive external events. Any non constant interpolation of external values received from the IEC 61499-based control or the SCADA application is omitted.

Every triggered event will be distributed using the configured distribution channels. The distribution may be done concurrently and parallel to the model update operation but in order to simplify the implementation it is modelled sequentially. The program may be shut down after a predefined stopping criterion is fulfilled. Such a stopping criterion may be a predefined stopping time or a stopping condition based on the outputs of the model. To demonstrate the feasibility of the predictive approach only the first stopping criterion is actually implemented.

The program flow of the event receiver as shown in figure 4.1b first listens for incoming events. The generic listening step includes handling various communication protocols such as the ASN.1-based protocol specified in the IEC standard 61499 [6]. After the event is received and its data is parsed, the central event queue has to be managed. Each predicted event that is triggered after the received event time is based on the assumption that no external event is triggered and will therefore become invalid. Hence, the auxiliary thread will remove these events from the event queue before the received event is inserted. Multiple external event sources such as IEC 61499-based controllers or a SCADA system may generate events independently. These events may arrive out of their temporal order. The thread which accesses the queue must reorder incoming events based on their time-stamp to establish a consistent order. After the auxiliary thread has added the received event, it immediately starts listening again.

4.3 User Interface Design

The user interface was designed to generate a trade-off between an intuitive and an efficient implementation. As described in section 4.2, the execution parameters such as the executed model, the featured lookahead horizon and the network address of the controller will be fixed during initialization. After the initialization no user input is required and FMITerminalBlock executes autonomously. The targeted use-cases require the user to have detailed knowledge of the simulation and the underlying ICT infrastructure. Hence, a command-line interface which takes the execution parameters as command-line arguments is justifiable.

During initialization, the given command-line arguments will be validated. If an error is detected, an appropriate error message will be displayed and FMITerminalBlock will prematurely terminate. In case of a detected error, the exit code of the program will be set to a non-zero value. Additional debug information including debug messages, timestamps and log levels is printed via the standard output of the operating system as well.

4.3.1 Command-Line Arguments

The command-line interface uses a property oriented syntax. Each property is passed on as a single command-line argument and consists of a key and a value part. The textual encoding follows the scheme `<key>=<value>`. The `<key>` fragment corresponds to the property key and `<value>` fragment to the configured value. Table 4.1 and 4.2 summarize supported properties. The property key must not contain any equals sign and the first equals sign is used to separate the key from the value fragment. In general, both property parts allow to store any other printable American Standard Code for Information Interchange (ASCII) character. Note that many command-line interpreters require some characters to be properly escaped or quoted [3, 47].

The output of FMITerminalBlock does not depend on the order of the properties. Each property may be passed on at an arbitrary position in the list of input arguments. If

Property Key	Type	Description
fmu.path	URL	The URL which specified the directory of the model
fmu.name	String	The model identifier
app.lookAheadTime	Decimal	The total lookahead horizon in seconds
out.<chn-nr>.protocol	String	The protocol identifier of the channel
out.<chn-nr>.addr	String	The destination address and port of the channel
out.<chn-nr>.<port-nr>	String	The associated model variable name
out.<chn-nr>.<port-nr> .type	Integer	The FMI type number of the port

Table 4.1: Mandatory Properties

Property Key	Type	Default Value	Description
fmu.instanceName	String	fmu.name	The instance name of the model
app.lookAheadStepSize	Decimal	$\frac{\text{app.lookAheadTime}}{10}$	Size of a single prediction step
app.integratorStepSize	Decimal	$\frac{\text{app.lookAheadStepSize}}{10}$	Size of a single integrator step
app.startTime	Decimal	0.0	The initial time of the simulation
app.stopTime	Decimal	∞	The final time of the simulation
app.timingFile	String	-	The name of the timing log file
out.<chn-nr> .<port-nr> .encoding	String	Sensitive	The ASN.1 port value encoding

Table 4.2: Optional Properties

a key is specified more than once, an error message will be printed and the program exits prematurely. The strict unambiguousness requirement avoids misconfigured simulations which are caused by contradicting property definitions.

FMITerminalBlock requires two parameters to load the FMU. The first parameter, `fmu.path` is the Unified Resource Locator (URL) to the extracted base directory of the FMU. It will be used as a base path to determine the descriptive XML file and to load the binaries of the model. The second parameter, `fmu.name` corresponds to the model identifier as specified in the descriptive XML file [66]. It is required by the current version of FMI++ to load the binaries. The current version of FMITerminalBlock avoids parsing the descriptive XML file outside the FMI++ library. Hence, it requires the user

to pass on the model identifier. Future versions of the software may automatically query the name from the descriptive XML file. Each FMU instance which shares a common binary may have a unique instance name which possibly differs from the model identifier. The instance name can be passed on by the optional `fmu.instanceName` property. If no instance name is specified, the model identifier will be used.

The mandatory `app.lookAheadTime` property specifies the lookahead horizon. After the specified time in seconds, the prediction logic will generate an output event independent from any FMI-event triggered by the FMU. The lookahead horizon is divided into a number of equidistantly spaced steps. At the end of each step the continuous state of the FMU is saved. The saved state will be used to efficiently interpolate the state if an external event occurs. The size of these lookahead steps can be controlled via the optional `app.lookAheadStepSize` parameter. By default, it is chosen to feature ten steps per lookahead horizon. The integrator step-size is controlled by the optional `app.integratorStepSize` parameter. Each lookahead step may be divided into several integrator steps. As for the other prediction parameters, the integrator step-size is expected to have the unit of seconds. The unit is implicitly assumed and the user must not append an additional unit postfix. By default, the integrator step-size is chosen to feature ten integrator steps per lookahead step as well.

The start time will be initially set to the `app.startTime` property value. If no start time is specified, a start time of zero seconds will be taken. The duration of a simulation may be limited by specifying the `app.stopTime` property. If the latest distributed event exceeds or equals the stop time, the simulation run will be terminated. To gain a more intuitive program flow, termination will be delayed at the end of the distribution cycle. Hence, no artificially generated event at the end of the simulation has to be distributed. Let, for example, $t_{\text{stop}} = 3\,\text{s}$ be the specified stop time and e_{end} the latest event which is scheduled at $t(e_{\text{end}}) = 3.5\,\text{s}$ and let the event before e_{end} be scheduled at $t(e_{\text{end}-1}) = 2.9\,\text{s}$. As soon as $e_{\text{end}-1}$ is triggered, the next cycle will be executed and e_{end} will be predicted. After the prediction step, FMITerminalBlock waits until the next external event is triggered or $t = 3.5\,\text{s}$ is reached and distributes the next event. According to Figure 4.1a, the end condition will be checked after the last completed cycle and the program will terminate at about $t_{\text{stop}} \leq t = 3.5\,\text{s}$.

FMITerminalBlock supports multiple output channels which feature a variable number of output ports and different protocol implementations. An output port is considered as a model variable which is transmitted to a data sink. Data sinks may be arbitrary network devices which implement the communication protocol. In the intended use-cases IEC 61499-based controllers or SCADA systems may act as data sinks. Different output ports must be grouped into one or more output channels. A single channel uses common communication parameters like communication protocols and communication endpoints. Many channels such as the implemented ASN.1-based channels ideally transmit the data of an event consisting of every configured port in one User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) packet. These packets will have to be decoded by the data sink in order to extract the transmitted information of the output port.

Each output channel property has an `out.<chn-nr>.` prefix where `<chn-nr>`

FMI Type	Type Number
fmiReal	0
fmiInteger	1
fmiBoolean	2
fmiString	3

Table 4.3: FMI Type Numbers

corresponds to the number of the output channel. The channel numbers are consecutive integer values starting at zero. If the next consecutive channel number is not present, it is expected that no more channels are configured. The assumption on consecutive channel number avoids an additional parameter which defines the total number of output channels. Each output channel must have an `out.<chn-nr>.protocol` property which specifies the used protocol.

Currently the protocol identifiers `CompactASN.1-TCP` and `CompactASN.1-UDP` are supported. Both protocols implement the ASN.1-based compact encoding as specified in IEC 61499-1 [6]. They differ by the used transport protocol. The first one, `CompactASN.1-TCP`, uses a TCP connection and the second one encapsulates configured ports into one UDP packet per event. Every ASN.1-based output channel requires an `out.<chn-nr>.addr` property which specifies the remote address of the channel. The first part of the address names the destination host or Internet Protocol (IP) address and the second part specifies the destination port. Both address parts are separated by a single colon.

The output ports of a channel are configured by the property `out.<chn-nr>.<port-nr>`. As for channel numbers, the port number represented by the generic `<port-nr>` part is a consecutive integer starting at zero. On using an ASN.1-based encoding, the output port number specifies the transmission order of the configured ports. Each `out.<chn-nr>.<port-nr>` property expects the output variable name of the model. For example, the argument `out.0.0=x` configures the first port of the first channel to transmit the output variable `x` of the model. Currently, the configuration must contain the FMI type number of each port. The FMI type number is specified in FMI++ and names the FMI type of an output port. The FMI type is used to correctly query model variables and may be different to the network encoding of the variable. Table 4.3 lists the corresponding type numbers. Later versions of the software may automatically extract the type number from the model description. To correctly specify the real typed model variable `x` of the previous example, the property `out.0.0.type=0` has to be added.

ASN.1-based channels may encode specific FMI types in different ways. A real FMI value, for example, may be encoded as `LREAL` or `REAL` value [6, 14]. In this case, the 64-bit `LREAL` features full precision and the 32-bit `REAL` may encode the same number with a reduced precision. By default, the ASN.1 encoder chooses the most appropriate

Encoding	Name	Description
0	Real Time Generation	After submission by a real-time event source
1	Prediction	After prediction by the included FMU
2	End of Distribution	Before the event is destroyed
3	Begin of Distribution	Before the event data is distributed

Table 4.4: Event Stage Encoding

encoding but some use-cases may require to alter the behavior. The encoding may be specified by the `out.<chn-nr>.<port-nr>.encoding` property which sets the output encoding. It takes a string value which has to be equal to the data type names defined in IEC 61499. If the conversion from a model variable type to the type of the encoding is not supported, an error will be triggered during initialization. For example, the property `out.0.0.encoding=REAL` reduces the precision of the model variable and transmits the value of the variable as a 32-bit floating point number.

4.3.2 Timing Data Interface

A timing data interface allows to track certain timing information of the simulation process. It is used to validate the best-effort approach and to track timing violations. The timing data interface is based on the assumption that during its lifetime an event crosses several program parts, called stages. At each stage, the timing according to the system clock and the event time is recorded. The collected information is written to a timing file which follows a syntax based on Comma Separated Values (CSV) files. A single semicolon is used as field separator.

Each line of the written timing file contains a single record made at a defined stage. To ease further processing, the event stage is encoded as a single integer value. Table 4.4 shortly summarizes the stage encoding. The current time-stamp is divided into several CSV fields including the current weekday, hour, minute and second. The field which stores the seconds value of the wall clock also includes fractions of seconds. Table 4.5 describes the fields of the CSV file in the order of their appearance. The last field may only contain some debug information and may not be properly quoted. It is advised to ignore every information after the last valid field separator.

The software will write the timing information to the file specified by the property `app.timingFile`. If the property is not present, no timing file will be written. After starting FMITerminalBlock, any existing content will be cleared and the first event is logged in the first line of the output file.

Field Index	Range	Field Description
0	$[0, 6]$	Number of the current day of the week starting at Sunday
1	$[0, 23]$	The hour field of the log record
2	$[0, 59]$	The minute field of the log record
3	$[0, 60)$	The seconds field of the log record
4	$[0, \infty)$	The logged event time-stamp in seconds
5	$[0, 3]$	The encoded stage
6	-	Some debug information

Table 4.5: Timing File Record Description

4.4 Software Design

The following section describes the deployed software design which includes the software partitioning and internal interface design. The software was designed with flexibility in mind. In addition to the operation specified in sections 4.2 and 4.3, the following modifications shall be possible without re-factoring unrelated parts of the source code.

- Adding additional properties to refine the behavior of the program
- Using another predictive event source
- Adding new transmission protocol implementations
- Transferring the real-time management to a connected device
- Adding various external event sources

The usage of dedicated interface classes simplifies implementation, testing, and maintenance. The following sections describe the software partitioning as well as the developed software modules.

Some of the Sections 4.4.2 to 4.4.5 show class diagrams. The syntax of these diagrams is based on the UML specification in its version 2.3 [8, 26]. To gain a better overview, some minor modifications according to the rules of the standard were made. The diagrams show public and protected class members only. To reduce the overhead, the diagrams contain neither parameter lists nor return values and show function names followed by two parenthesis only. Final class members will be written in upper case letters only. By convention, final class members starting with `PROP_` correspond to property keys of the global configuration. Following the C++ standard, the class diagrams mark each object destructor with a leading swung dash [58]. For instance, the function `~Event()` corresponds to the destructor of the `Event` class.

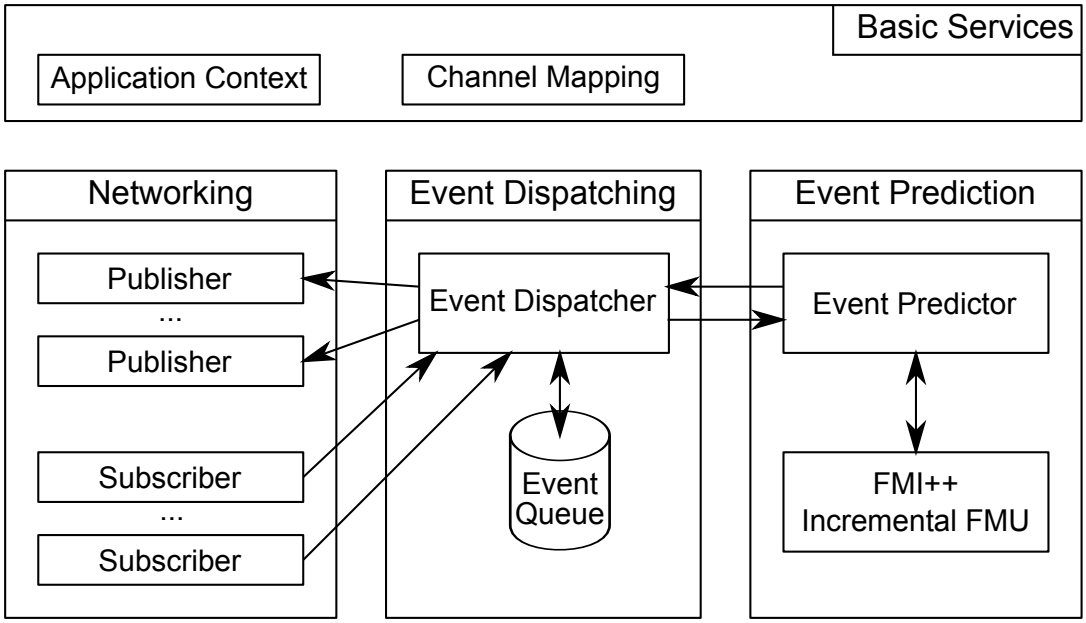


Figure 4.2: Basic Program Design

4.4.1 Basic Design

The solver and event mapping components which are required to integrate an FMU are mainly implemented by FMI++. Hence, the basic system design focuses on the remaining parts of FMITerminalBlock. The design includes the user interface, the networking facility, the real-time scheduler which properly delays events, and the event prediction logic which subsumes several components described by the abstract model. Figure 4.2 illustrates the basic system architecture. FMITerminalBlock is partitioned into four modules. Each module has its associated name-space and consists of several classes.

The basic service module provides helper classes and functions which do not implement the core program logic but provide commonly used features. These classes and functions include global configuration management. Each program module must have access to the user's configuration in order to parametrize its operation. The configuration is distributed by a globally unique application context object which encapsulates the configuration information. The application context object also provides access to other basic services like the channel mapping. The main purpose of the channel mapping is to provide a common representation of the data flow between the event prediction logic and available output channels. It only encodes the routing information of the data transport and does not provide any transport or data encoding functionality itself. For example, the channel mapping encodes that the output variable x of the model will be sent to the first publisher. It does not redirect the output to the given publisher or specify any representation of the value of x.

The event prediction module implements the simulation logic. It predicts the next

event based on the included FMU and updates the model according to external events. The event prediction module has to implement the event mapping logic which maps the event-based execution of IEC 61499-based systems to continuously operating FMUs.

Communication between the IEC 61499-based controllers or a SCADA system and FMITerminalBlock is handled by the networking module. It encodes predicted events and generates events by parsing incoming network traffic. Since the received traffic is potentially independent from the generated one, the decoding and encoding logic is modelled in separate program parts. A publisher encodes events and the subscriber parses incoming network traffic and emits new events if necessary. Several independent subscribers and publishers may coexist. Each publisher and subscriber may also implement a different communication protocol.

The event dispatching module conducts the main program flow described in section 4.2 and delays events according to their associated time. It uses the event prediction module to forecast events and redirects the events to the networking module. The event dispatching module also provides some interfaces to register externally triggered events. The event timing and emission will be managed in a central event queue which is accessed by the event dispatcher.

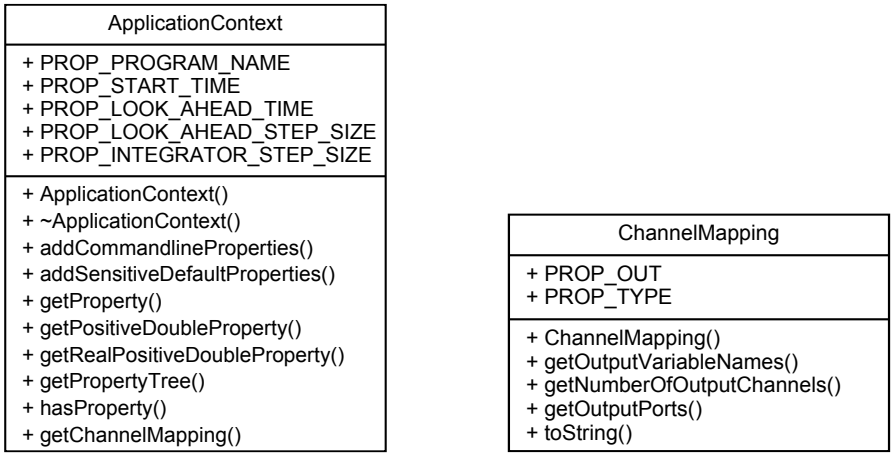
Each module is located in a separate C++ namespace. The basic services share the namespace `FMITerminalBlock::Base`, the event prediction module is located at `FMITerminalBlock::Model`, the event dispatching module is located at `FMITerminalBlock::Timing` and the C++ namespace `FMITerminalBlock::Network` covers the network facilities.

The basic program design was checked against the design goals listed in section 4.4. The usage of a dynamic application context allows to easily add new properties. The generic networking model enables various external event sources. In contrast, the implementation of different real-time management and prediction strategies is not covered at this design stage. Proper abstract interfaces which allow the displacement of single components have to be specified by the specific software design.

During further development, the basic design was reduced to its essentials and is not fully implemented. Currently the event prediction, dispatching, and publishing logic is present. Program parts which receive and process external events are contemplated but not coded and may be added in future versions. The following sections will focus on the implemented features and do not cover future extensions in detail.

4.4.2 Basic Services

The class `ApplicationContext` is the center of the basic services. It provides access to information such as the configuration and the channel mapping which has to be available globally. Figure 4.3a gives a first overview of the application context members. The configuration is stored in terms of configuration properties in a property tree. Each property is addressed by a unique hierarchical identifier, called path. A path of a property may be encoded by a single string. The different hierarchic levels within the path string are separated by a single dot character.



(a) Application Context (b) Event receiver program flow

Figure 4.3: Main Basic Service Classes

The property tree is first populated by parsing the command-line arguments. During initialization, it is enhanced by global sensitive default values. By adding calculated values, other program parts gain a consistent view of the configuration values. In contrast to global default values, local default values will not be stored in the property tree. They can be easily managed by the requesting program module and will not have to be available for other modules. The central configuration is also used to add static information like the name of the command at the start-up phase. The added information may be accessed consistently and does not have to be maintained separately.

The `ApplicationContext` class does not only provide access to property subtree objects itself but also defines several façade functions which directly allow to retain configuration values [62, p.73f]. Some of the access functions already check the range of the configuration value and may throw an appropriate exception. An accessing program part does not need to perform these checks manually and can reduce its code complexity.

Although the application context object is globally unique, it was not designed as singleton instance [62, p.35ff]. The singleton design pattern restricts inheritance and decreases the extensibility of the program. Future versions of FMITerminalBlock may have to maintain multiple instances of the event processing framework described in section 4.4.1. Hence, they may need separate application context objects. In contrast to the singelton approach, the application context is managed by the basic framework functions which start the application. Each program module maintains a reference to the application context object which is passed on to the constructor of the module.

The application context object also implements the `getChannelMapping(void)` function which creates and returns a pointer to a `ChannelMapping` object. The channel mapping object represents the association of model variables and output channels in an easily processable format. Figure 4.3b shows an overview of the channel mapping members. Each piece of accessible information is called port. Each port is identified

by its data type and a numerical identifier. For example the real typed output variable x may be internally addressed by its type `fmiTypeReal` and a numeric identifier, let us say 42. Both parts of the port identifier are encapsulated into a `std::pair` called `PortID` [32, 56].

The port identifier allows to efficiently address differently typed variables. The channel mapping object manages the port identifier. The numerical identifier of each port is always unique for a particular port type. It will be assigned consecutively starting at zero. The assignment policy allows to efficiently buffer the content of the model variable into arrays. In contrast, numerical variable identifier of the FMI may not be consecutively assigned and their usage does not allow a direct addressing of array elements [66]. In addition, `IncrementalFMU` which is implemented by FMI++ hides these FMI identifiers and only provides name-based variable access.

The channel mapping object also exposes the number of output channels and a list of port identifiers per output channel. A publisher may utilize the list of output ports to send associated information. A model variable may be associated to multiple output channels. In this case, the same port identifier is returned for multiple channels. A single port identifier may also be associated to an output channel multiple times. In case of equal port identifiers, the value may also be sent multiple times.

The channel mapping focuses on the output port assignment only. It does not store any additional information beyond the mapping information. Any additional information such as output addresses or encoding information must be parsed individually by each publisher. Future versions may alter this behavior and may also provide additional configuration information via associated property subtrees.

4.4.3 Event Dispatching

The event dispatching module defines an abstract event class which encapsulates the event time and its associated values. Figure 4.4 shows the functions of the event classes as well as two derived classes inheriting from the event base class. Each value is called variable and consists of its port identifier which is further described in section 4.4.2 and its actual value. The value is stored in a `boost::any` object [35]. The universal value representation allows a unified processing of differently typed variables. Both elements of a variable are encapsulated in a `std::pair` named `Variable` [32, 56].

The abstract event class called `Event` defines two main functions. The implemented function `getTime(void)` returns the previously set time and the virtual function `getVariables(void)` returns a vector of event variables. Some event sources like the model may return the whole set of available output variables but future implementations may only emit a subset of variables. Additionally, a virtual function `toString(void)` is defined which returns a human readable representation of the event data. The function is mainly used for debugging purposes and may not always be fully implemented.

Each life-cycle of an event object begins with the event detection or prediction and ends after it has been finally distributed. In general, the event producer object and the object which determines that the life-time of the event has expired are not equal. First, the event will be produced by an event predictor or a subscriber. It will reach the end of

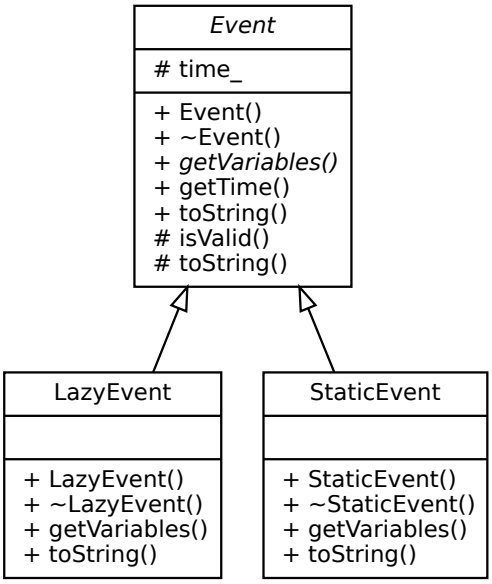


Figure 4.4: Abstract Event Class and Sub-Classes

its life-cycle after the event dispatcher has finally distributed the event. At the end of the life-cycle of an event, it has to be deleted and the memory has to be freed. To free allocated memory, each event source may also register as an event sink which deletes the object. In this case, the dispatcher has to call the event source after all other sinks are notified. Otherwise, event sinks may receive an invalid reference. To reduce the code complexity an exception to the commonly enforced memory policy was made. In case of event objects, freeing of allocated memory is not up to the object which allocates the memory of the event. Instead, the producer is liberated from its obligation and the caller must ensure deallocation. If, for example, the event predictor produces an event, the receiving event dispatcher will have to delete it.

The event dispatcher which is introduced in section 4.4.1 controls the main program flow. It implements an observer-like pattern to distribute emitted events [62, p.68]. Each event observer which needs to receive emitted events implements the `EventListener` interface. The listener interface defines a single virtual function, `void eventTriggered(Event *)` which receives the triggered event. The event dispatcher acts as observable object and provides functions which add an observer to the list of notified observers. Figure 4.5 shows an UML-like class diagram which contains the event dispatcher and its major relations.

Additionally, the event dispatcher defines a `void run(void)` function which executes the main program flow. On calling the `void run(void)` function, first the event dispatcher uses the event predictor to calculate the next event. As soon as the next predicted event is known, the dispatcher puts the event into its event queue and waits until the queue returns the next event. Lastly, it distributes the gathered event by calling every listening observer and then deletes it. The event dispatcher may run multiple event

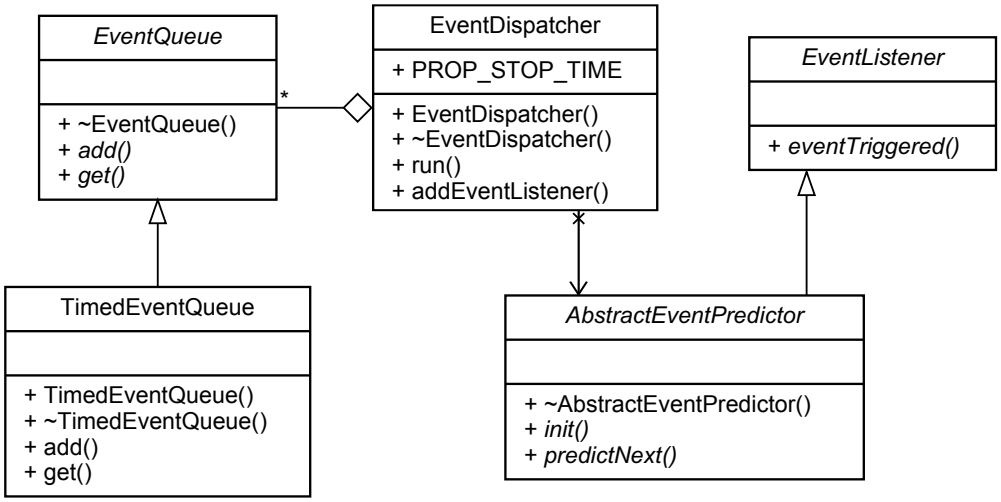


Figure 4.5: Event Dispatcher Class

cycles until the exit condition is reached.

Delaying certain events and scheduling recent events is delegated to a dedicated event queue object. The event queue stores and manages events and returns them according to the event management strategy. An abstract interface called `EventQueue` is introduced to ease the implementation of alternative scheduling strategies. Each event queue provides two main interface functions, an adder function and a getter function. The adder function, `void add(Event *, bool)`, takes a pointer to the event and a flag which indicates the source of the event. The flag has to be set if the event is a predicted one which may get invalid before it is emitted. The getter function, `Event * get(void)`, returns the next scheduled event according to the implemented scheduling strategy. It may delay the calling thread until the event time is reached.

The event queue has to be able to process concurrent accesses. These accesses won't be synchronized by the event dispatcher and have to be managed by the queue. Most notably, the queue has to process incoming events while the getter function blocks a calling thread.

An event queue implementation is provided by the timed event queue class `TimedEventQueue`. It implements the queue management functionality described in section 4.2 and blocks the fetching thread until the next event time. Internally it uses a `boost::mutex` object to synchronize concurrent accesses. A condition variable is utilized to notify waiting threads about incoming events [54].

The `void add(Event *, bool)` function of the event queue first locks the internal event pointer storage and checks whether the new event is an external one. If the event is an external one, every predicted event which is to be scheduled after the new one is deleted. After the new event is inserted to the internal event storage, the waiting thread gets notified about the queue changes using the condition variable.

Snippet 1 shows the get function of the timed event queue. The get function delays an

```
Event *
TimedEventQueue::get(void)
{
    boost::unique_lock<boost::mutex> lock(queueMut_);
    Event* ret = NULL;

    while(ret == NULL)
    {

        if(queue_.empty())
        {
            newEventCondition_.wait(lock);
        }else if(isFutureEvent(queue_.front().first)){
            // Wait until the event time is reached
            (void) newEventCondition_.timed_wait(lock,
                getSystemTime(queue_.front().first));
        }else{
            // Process event immediately
            ret = queue_.front().first;
            queue_.pop_front();
        }

    }

    return ret;
}
```

Snippet 1: Event Retrieval and Blocking Function

event according to its associated time-stamp. If the internal element storage `queue_` is empty, the requesting thread is blocked until the adder function notifies a waiting thread via the condition variable `newEventCondition_`. If the element storage is populated, the `timed_wait` function of the condition variable is utilized to delay the event until it is scheduled. The function either times out if the event time is reached or returns if the adder queues another event. As a consequence, the timing highly depends on the implementation details of the synchronization primitive. After the return of the waiting function, the content of the event storage is evaluated again and immediate events will be dequeued and returned.

4.4.4 Event Prediction

The event predictor described in section 4.4.1 was specified using the interface `AbstractEventPredictor`. Figure 4.6 visualizes the abstract event predictor interface and its implementation. It defines a virtual initialization function, `void init(void)`, which initializes the model as soon as the global configuration is stable and every property is known. The next event will be predicted by calling the virtual function `Event * predictNext(void)`. It creates a new event instance and returns a pointer to it.

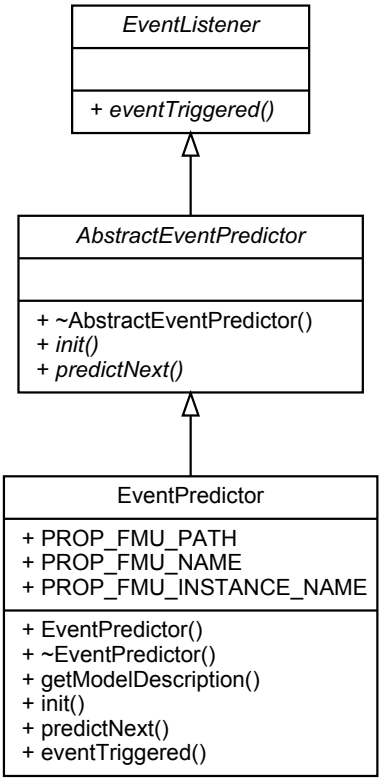


Figure 4.6: Event Predictor Classes

Following the approach described in section 4.4.3, the memory of the event instance must be managed by the caller instead of the callee.

A model may also receive incoming events to update its state accordingly. The abstract event predictor interface inherits from the event listener interface `EventListener` which specifies an event notification function. When an external event occurred, the event notification function has to be called and the model may be updated accordingly. As a result of the FMI specification, some restrictions are imposed on the calling sequence. Therefore, the sequence must be specially designed on the abstract level to ensure a proper operation of the implemented predictor. The outputs of the model may be calculated on request only. According to section 4.2, the discrete state of the FMU cannot be reset after the calculation of outputs. Hence, the predictor cannot process any external event after the outputs of the event are read.

The event predictor implementation accesses the `IncrementalFMU` implementation which is provided by FMI++. Any event prediction is done by the FMI++ library. The event predictor conducts the single calculation steps and queries the results. After the prediction, the next event time may be immediately queried but its output calculations have to be delayed until the event is actually taken. An event class friended to the event predictor, `LazyEvent`, implements the delayed output query. It calls the event predictor

and queries the model data only upon request. Instead of an event instance which already contains the predicted data, the `Event * predictNext(void)` function of the event predictor returns a new `LazyEvent` instance. By utilizing the returned event object, the state of the FMU may be queried later on.

A predicted event may get outdated if another event is predicted by the same event predictor instance. An outdated event must be destroyed and cannot return previously calculated outputs. Figure 4.1 shows that only one predicted event is needed at a time. Each new prediction step will invalidate any previously calculated event. To detect program errors `LazyEvent` checks whether the event is already outdated and will throw an appropriate exception if needed.

4.4.5 Networking Module

The networking module is split up into several publisher instances and a managing object called `NetworkManager`. The network manager instantiates and deletes the publisher object instances. It also registers these instances at the event dispatcher instance for event notification. Figure 4.7 visualizes the relationships between the classes of the networking module.

An abstract publisher interface class called `Publisher` defines the interface functions that are used to manage each publisher instance. The interface allows to add new publisher instances without modifying major parts of the network manager object. As soon as a publisher is instantiated, an initialization function is invoked to configure the publisher. The initialization function expects the port identifier of the publisher and the configuration subtree which configures the specific publisher. The split initialization was chosen to ease the publishers instantiation. Each publisher should provide a default constructor. The function which dynamically instantiates the specified publisher only has to call the default constructor and does not have to deal with different parameters.

Event notification is done by the event listener interface of the event dispatcher. The publisher interface inherits the listener and provides a common event notification function. On receiving an event notification, each publisher instance transmits the event data via the configured output ports.

The ASN.1-based communication protocol is supported via UDP and TCP. For each transport protocol, a dedicated publisher class is implemented. Class `CompactASN1TCP ClientPublisher` connects to a TCP server and transmits any ASN.1-based message via the opened channel. In contrast, `CompactASN1UDPPublisher` does not maintain a connection and publishes each message via UDP. Both classes share a common base class, `CompactASN1Publisher`, which manages the output variables and encodes the messages.

Each ASN.1-based publisher maintains a copy of its configured output variables. On receiving new events, the publisher updates its output variable copies according to the event variables. If the event does not contain the full set of output variables, missing variables remain unchanged. An encoding function encodes the buffered output variables and adds their encoded values to a message buffer.

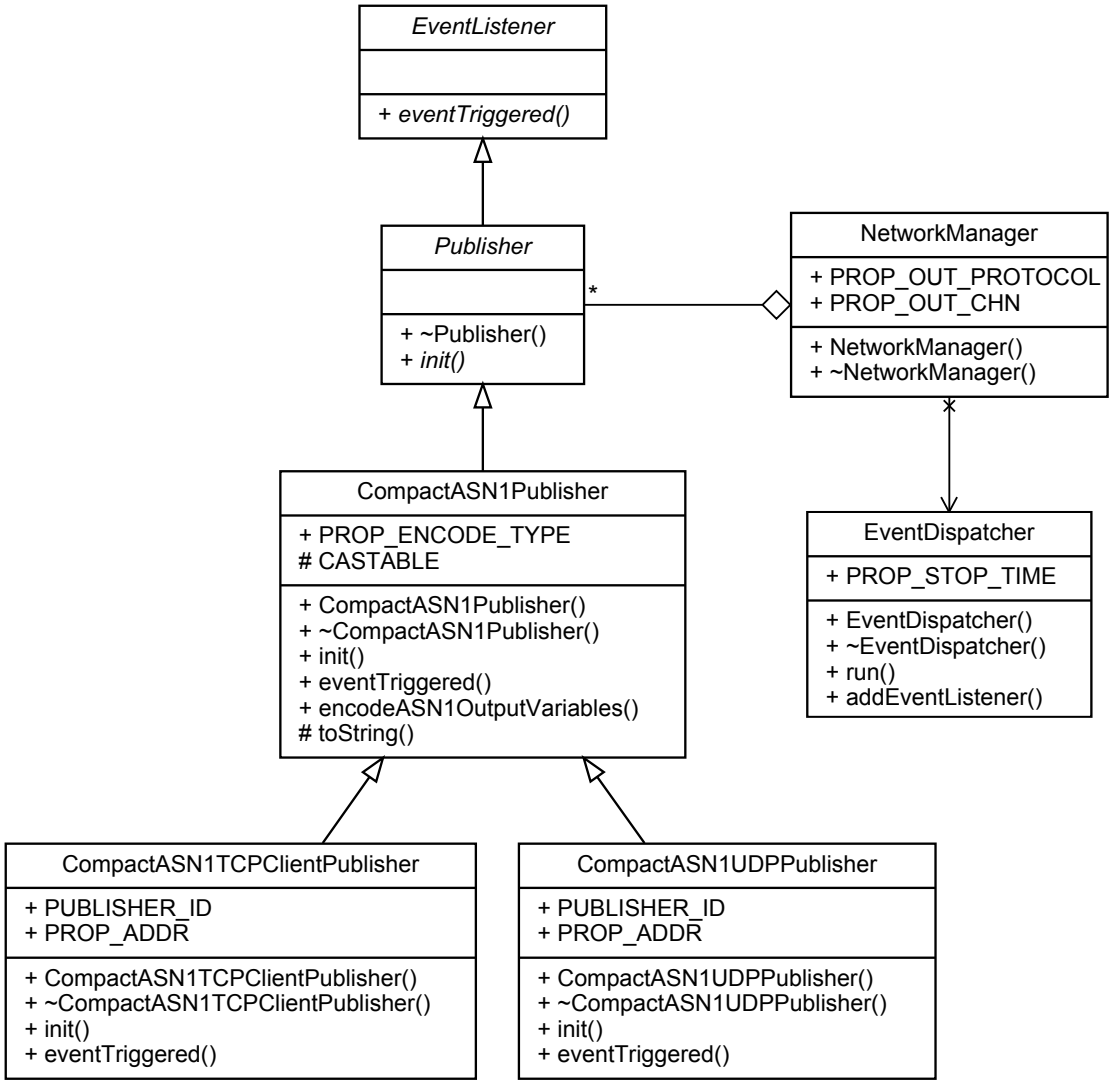


Figure 4.7: Networking Module Overview

The encoding scheme follows the compact encoding rules defined by the IEC standard 61499 [6, p.98ff] and by the ASN.1 standard itself [4, 12]. The implemented ASN.1-based publisher partly accepts different encoding schemes per FMI type. For example, an `fmiReal` typed value may be encoded as 32-bit or 64-bit floating point number. By casting the `fmiReal` typed 64-bit value to a 32-bit floating point number, precision might get lost [14, p.6] but the integration of existing IEC 61499-based applications is eased. Per default, a lossless encoding is chosen which avoids numerical errors introduced by an inexact casting.

The implemented encoding functions have to assure that the byte order of a value is converted to the big endian network byte order [6, 12, 48]. The byte order is guaranteed by populating the message buffer on a per-byte basis. To transmit floating point numbers which follow the IEEE standard 754, FMITerminalBlock assumes that the host processor and the used compiler internally use this encoding. The assumption slightly restricts targeted platforms but increases the program code maintainability due to a simplified program flow.

4.5 External Software

Some external programs and program libraries are used to ease development and maintenance of FMITerminalBlock. The following sections shortly describe the external software and explains the reason for their usage.

4.5.1 FMI++ Import Utility

The FMI++ library includes various import utility classes which handle the instantiation, communication, and integration of included FMUs [11, 13, 27]. FMI++ also provides a class named `IncrementalFMU` which already implements the event prediction. It is distributed by a BSD-like license which allows modifications of the included source code. At the time of writing, an updated version is not yet published but can be expected soon.

The inclusion of external model exchange FMUs which follow the proposed approach requires the implementation of the low-level C interface. Additionally, a numerical integration mechanism which is able to deal with intermediate FMI events has to be provided. The FMI++ library already implements both parts. Since the source code of FMI++ is available, it can be adopted to the needs of FMITerminalBlock. These facts lead to the design decision of building on the import functionality of FMI++.

The predictive FMU implementation, `IncrementalFMU`, implements a function which changes the current state of the FMU to a predicted one [11, 13, 27]. Since the update function called `fmiTime updateState(fmiTime)` only uses predicted values, it may only be set to a value before the event handling function is executed. As mentioned before, any predictive value issued after an event is being handled may potentially change the internal discrete state of an FMU [66] and prohibits a proper reset.

As described in section 4.2, the program requires to calculate the output value directly after an event has happened. To overcome the minor limitation, a function

`fmiTime updateStateFromTheRight(fmiTime)` was added to the FMI++ library. The newly added function checks whether an event has happened and performs a very small integrator step. During this step, the FMI event handling functions get called and the state updates to its value right after the event.

4.5.2 CMake Build System

To ease cross platform development, the build system generator CMake has been chosen [30]. In contrast to GNU autoconf, CMake does not depend on platforms which maintain a shell compatible interpreter [30, 39]. It is also capable of generating Microsoft Visual Studio-based build environments. CMake generates the actual build system such as a Microsoft Visual Studio Project Folder or a GNU Make file based on a list of high level targets. To configure the build process, several CMake input files were written.

FMI++ also uses CMake to generate its build system. To increase the compatibility, most of the compiler settings of FMI++ are also used in the CMake input files of FMITerminalBlock. The newly written CMake configuration requires a path pointing to the development directory of FMI++. It automatically compiles FMI++ into shared library files which were needed by FMITerminalBlock. Optionally, the CMake configuration also compiles the test-cases into test programs. These test programs also require the FMI++ library object and execute a collection of test-cases.

4.5.3 Boost

Several functionalities such as multi-threading or networking support are required to efficiently develop a program which interfaces FMUs and IEC 61499-based controllers. Ideally, these functionalities can be abstracted from the operating system Application Programming Interfaces (APIs), which leads to an almost platform independent program code. FMI++ depends on several free libraries called Boost [11, 24]. The Boost libraries provide the required basic functionalities and most often support a broad variety of platforms [1, 33, 34, 35, 49, 54, 55, 63]. Using the Boost libraries does not introduce another dependency because they are needed by FMI++. They provide several functionalities which are not covered by the C++ standard library yet [32, 56]. Hence, FMITerminalBlock extensively uses some Boost libraries.

The property tree implementation of Boost provides the basis for the global configuration of FMITerminalBlock. The application context class maintains a property tree object which stores the configuration [63]. Several other parts such as the event publisher also access the configuration via a property subtree. By using a library, the complex tree management and path resolving is delegated.

The networking and multi-threading facilities of the Boost libraries are used to implement the event scheduling and event transmission [1, 54]. To increase portability, only functionality available on both Linux and Windows was used from these libraries. Both libraries support various platforms and avoid the usage of operating system dependent APIs. Additionally, several other libraries such as the Boost any type library [35] and the format library of Boost [34] are used to avoid re-implementation of available functionality.

CHAPTER 5

Evaluation

5.1 Unit Testing

To test the program, an automatic testing framework was deployed. The testing framework includes several unit tests which cover every program module described in section 4.4. Each test collection which covers a coherent part is compiled into a different test executable. On executing a test executable, every corresponding test-case is executed and the result is returned. The implemented test-cases use the Boost Test library extensively [55]. The Boost test library provides the basic testing framework and assembles single test-cases into a common executable. Additionally, it provides several helper functions which check return values and expected conditions.

Generally, every test collection covers valid use cases of each tested class and some faulty usages which are expected to throw an exception. Every test-case which handles an exception targets exactly one error condition. Let, for example, $f(x, y), x > 0 \wedge y > 0$ be a function which expects two real positive parameters. A test-case shall never set both arguments to zero at the same time in order to efficiently test the exception handling. All test-cases specially focus on boundary values and may not reflect expected input values.

To test the ASN.1-based network facilities, some test fixtures which contain commonly used functionalities were written. Each test fixture follows the test fixture approach which is implemented by Boost [55]. A test fixture encapsulates its variables and functions in a C++ structure. Figure 5.1 visualizes the concept of the implemented test-cases which test ASN.1-based publishers. The `ASN1Fixture` structure contains a minimal configuration which is necessary to instantiate relevant publishers. Additionally, it provides a function which checks the content of a message buffer. The checking function is used as a callback function. A UDP or TCP server which is instantiated for testing purpose only executes the callback function and passes the received data on. Moreover, the status of the server is checked to be successful.

Two derived test fixtures implement the actual servers which listen for incoming packets. Most network related test-cases use these servers to check the network communication.

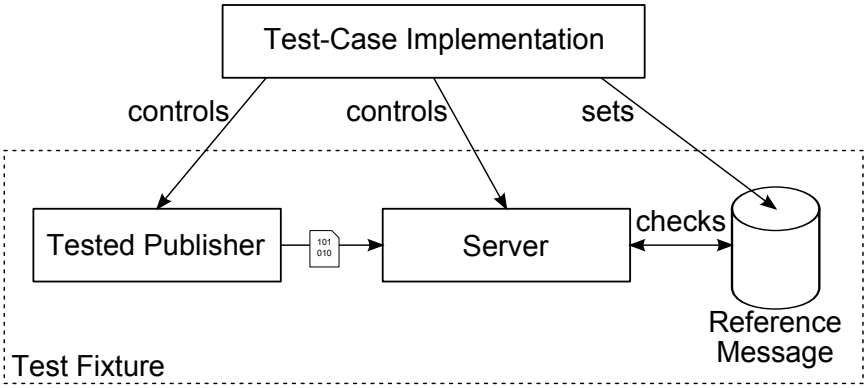


Figure 5.1: Publisher Test Setup

Both servers rely on the Boost Asio library and implement the asynchronous approach [1]. To control the operation of a test fixture, the callback functions are only called if the test-case runs an Input/Output (IO) service execution function `ioService.run_one()`. If the execution function is not called, no result will be checked. Hence, no synchronization mechanism is needed and the test-case complexity is reduced. At the end of each test-case the number of successfully received messages is tested. Testing the message count also detects failures which do not call the checking function at all.

The ASN.1-based encoding of transmitted values is compared to a statically coded reference. The static reference which mainly consists of manually derived values avoids false positive results by failures which are present in the reference implementation as well. The reference values were further compared to the values sent by the IEC 61499-based controller 4DIAC FORTE [31]. Both encodings turned out to be identical.

Snippet 2 shows a test-case implementation which uses the TCP fixture `ASN1TCP Fixture`. Each test-case in the test set is started by an appropriate macro which is defined by the Boost Test library [55]. The `BOOST_FIXTURE_TEST_CASE` macro takes exactly two arguments, a unique name of the test-case and the name of the included test fixture.

Obviously, several variables like `config`, `receiveReference` and `publisher` are defined outside the test-case (see snippet 2). These variables are defined as members of the test fixture and are initialized during its instantiation. The test-case does not have to instantiate them manually and may be reduced to a minimum amount of code. Although the amount of code is shortened and the maintainability of several similarly written test-cases is increased, the test fixture approach introduces a significant drawback. A reader must know the capabilities and implementation details of the test fixture to fully understand the test-case. In future versions, the structure-based approach proposed by the Boost Test library [55] may be improved by test fixtures which reduce the number of exposed variables by proper encapsulation.

Less complex test-cases like those testing the event predictor or the application context only access a very simple test fixture or do not use a test fixture at all. For example,

```
BOOST_FIXTURE_TEST_CASE( test_publish_ASN1_TCP_manual_enc_0, ASN1TCPFixture )
{

    // Setup configuration
    config.put("0.encoding", "LREAL");

    // Setup event variables and their values
    std::vector<Timing::Event::Variable> vars;

    vars.push_back(std::make_pair(
            std::make_pair(fmiTypeReal, 666),
            (fmiReal) DBL_EPSILON));

    // Setup reference
    uint8_t ref[] = {0x4b, 0x3c,0xb0,0x00,0x00,0x00,0x00,0x00,0x00};
    receiveReference.assign(ref, ref+sizeof(ref));

    // Init publisher
    publisher.init(config, ports);

    // Let the client connect
    ioService.run_one();

    // Trigger event
    Timing::Event * ev = new Timing::StaticEvent(0.0, vars);
    publisher.eventTriggered(ev);

    // Fetch and check the IO result
    ioService.run_one();

    delete ev;
    BOOST_CHECK_EQUAL(validMessages, 1);
}
```

Snippet 2: Example of an ASN.1 Unit Test

the fixture which supports event predictor test-cases only exposes a single variable. The variable holds a minimal configuration inside an application context object. A person who reads the code still has to know the state of the application context, but the small test fixture is much easier to review.

The test-cases which test the event handling functionalities do not only demonstrate the event operation of the dispatcher but also test the exchangeability of the event predictor. An alternative event predictor class was written which simply emits equidistantly timed events. The event dispatcher test-cases successfully use the alternative event predictor instead of the complex FMI++-based one. The newly written predictor fits into few lines of code and produces a very predictable output. Hence, failures which result from an inexact FMI implementation can be eliminated. Aspects related to FMI and FMI++ are tested separately.

The event predictor test-cases utilize an FMU which was primarily written to test the FMI++ implementation. Most related test-cases instantiate a predictor which accesses the test FMU. The outputs of the predictor are checked against a manually calculated reference. Each significant deviation will be reported as an error. For floating point comparisons, the Boost macro `BOOST_CHECK_CLOSE` is utilized. The macro will report a failure if the gathered result deviates too much from the expected one. Due to unavoidable numerical errors, an exact floating point comparison may result in false negatives.

The unit tests focus on the correct functionality of single program modules. They supported the initial development and raised several bugs which might not have been found otherwise. Especially, the ability of Boost to find memory leaks supported the detection of bugs which are usually hard to find [55]. The automatic test-cases also showed their potential for regression tests. After adding new features, existing test-cases were executed which avoided a time consuming and error prone manual testing. At the end of the development cycle, every test-case was executed successfully.

5.2 Proof of Concept Setup

This thesis mostly focuses on the interface logic between FMI for model exchange and IEC 61499-based controllers. Several aspects such as the detailed software development cycle and extended use-cases are beyond the scope of this document. However, two simple test setups and their experimental evaluations are presented. Both test setups aim at demonstrating the potential and limitations of the prediction-based approach. They do not claim full applicability in complex energy systems.

Both setups include models which were generated in OpenModelica [71] and exported by its built-in FMI export functionality. Sections 5.2.1 and 5.2.2 describe these models. One model was executed at a hardware test-stand and the other model was executed at a local test setup. Both setups are described in section 5.2.3.

5.2.1 Test Model

Testing and evaluation of the implemented prediction-based approach requires a test FMU which triggers events. To limit the probability of errors which are introduced by the model, a simple model was chosen. The simple model outputs a time-dependent function consisting of single steps and a sinusoidal function. Since IEC 61499-based controllers communicate data at discrete points in time only [6], a sampling component was added. The sampling component of the model generates FMI-events and maintains constant outputs between two consecutive events. It could use a constant sampling rate but equidistantly triggered events do not fully utilize the capabilities of the prediction-based approach. Hence, a dynamic sampling component which adjusts the sample rate based on the inputs of the sampler is implemented.

Figure 5.2 shows the block diagram of the sampling component as displayed by Open-Modelica. The functionalities of the model are implemented by connecting predefined Modelica library function blocks. The function blocks provide an adequate visualization

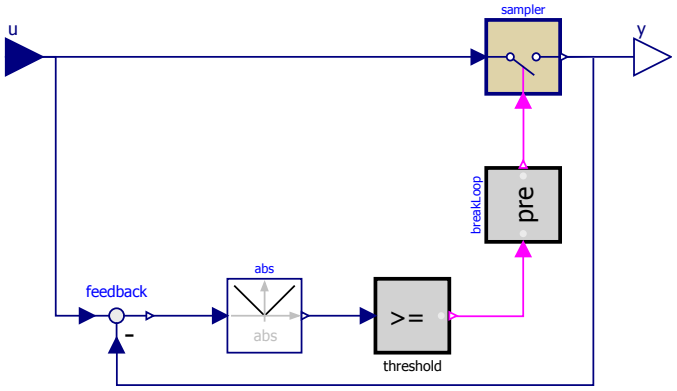


Figure 5.2: Dynamic Sampling Component

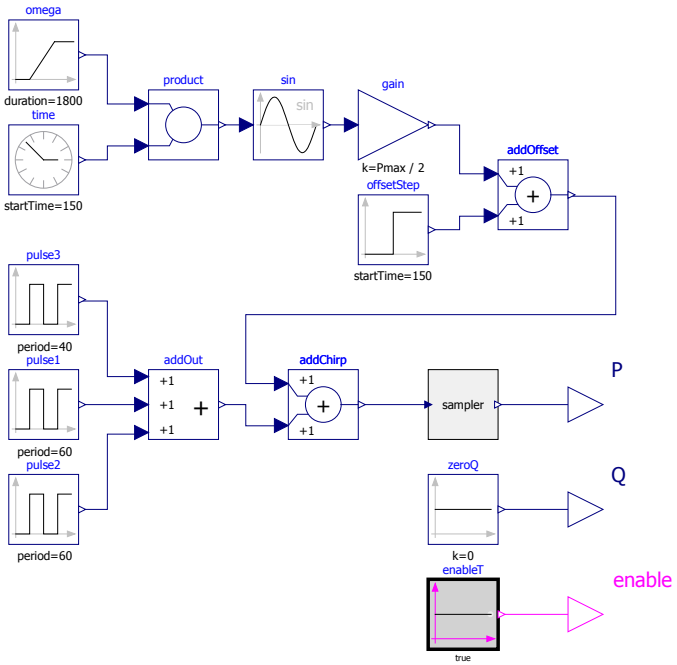


Figure 5.3: Test Model

without the need of exposing the text-based Modelica language. The dynamic sampling component continuously compares the input $u(t)$ with its output $y(t)$ via the feedback block. If and only if the absolute deviation of the in- and output exceeds a given threshold, the input $u(t)$ will be sampled. Depending on the input function, sampling will be done in non constant intervals. Note that the pre block delays its input by an infinitesimal small time step [57]. It breaks the algebraic loop which arises by the feedback loop and allows a proper simulation.

The test signal generation is shown in figure 5.3. The implemented model also uses Modelica function blocks which generate the output signal. The outputs `P`, `Q` and `enable`

Parameter	Configuration Fast	Slow
$P_{\max}$	12 kW	12 kW
ω_0	$2\pi\frac{1}{15}$	$2\pi\frac{1}{360}$
$\Delta\omega$	$2\pi\frac{1}{60}$	0
t_0	5 s	5 s
t_1	25 s	45 s
t_2	60 s	120 s
t_3	90 s	180 s
t_4	120 s	240 s
t_5	150 s	300 s
t_6	1950 s	1950 s

Table 5.1: Test Model Parameters

of the model are sent to the test-stand controller. Only the resistive load set-point `P` is currently adjusted during the experiments. Other outputs are present to meet the interface requirements of the test-stand.

To test the stepwise adjustment, the function blocks `pulse1`, `pulse2` and `pulse3` first generate a single pulse and a stepped pulse. Figures 5.13 and 5.14 show the expected outputs of the model. Let $y_{\text{pulse}}(t)$ be the output of the pulse function block at time t and $\sigma(t)$ be the Heaviside function [17, 59]. The output $y_{\text{pulse}}(t)$ is given by (5.1) which assumes the parameters $P_{\max}$ and $t_0 < t_1 < t_2 < t_3 < t_4$ to be known beforehand.

$$\begin{aligned} y_{\text{pulse}}(t) &= \frac{3P_{\max}}{4}\left(\sigma(t-t_0)-\sigma(t-t_1)\right) \\ &\quad + \frac{P_{\max}}{3}\left(\sigma(t-t_2)+\sigma(t-t_3)-2\cdot\sigma(t-t_4)\right) \end{aligned} \tag{5.1}$$

After the step functions return to zero, a chirp-like test sequence is started. The function block `omega` constantly increases the frequency of the sine component until the maximum frequency is reached. Since the resistive load does not feature negative power, an offset was added to the sine signal. Hence, all load set-points will be positive. The output $y_{\text{chirp}}(t)$ of the test sequence is given by (5.2).

$$\begin{aligned} y_{\text{chirp}}(t) &= \sigma(t-t_5)\left(\frac{P_{\max}}{2}\sin\left(\left(\Delta\omega(t-t_5)-\Delta\omega(t-t_6)\sigma(t-t_6)+\omega_0\right)\cdot t\right)\right) \\ &\quad + \sigma(t-t_5)\cdot\frac{P_{\max}}{2} \end{aligned} \tag{5.2}$$

The chirp-like test sequence is parametrized by the parameters $P_{\max}$, ω_0, $\Delta\omega$ and $t_5 < t_6$. When testing, two different configurations of the model were applied. Table 5.1 shows the deployed parameter values of both configurations.

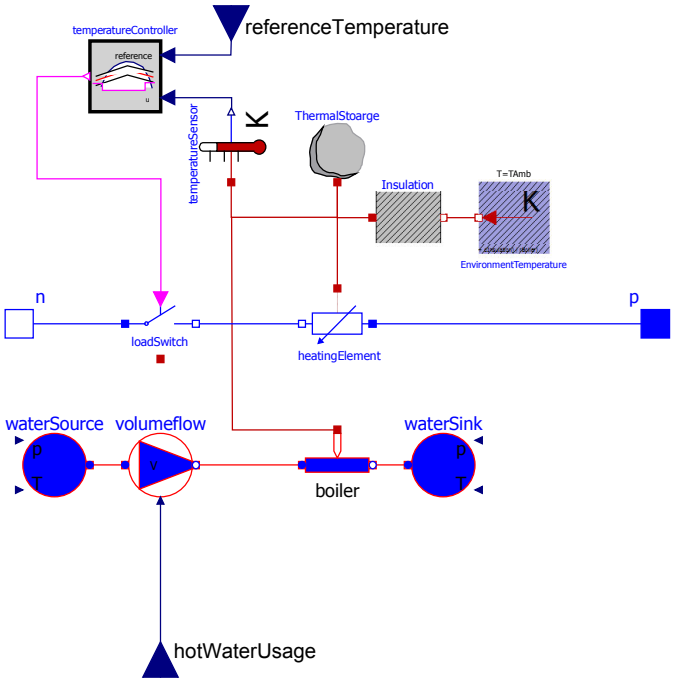


Figure 5.4: Simple Boiler Model

First, the FMU which was exported by OpenModelica failed to execute. An error message which indicated that it is not allowed to set an empty state vector was displayed. Since the test model does not contain any continuous state, the state vector originally remained empty and only algebraic equations had to be solved. To overcome the limitation, a constant dummy state was manually added to the model. Snippet 3 shows the applied workaround.

```
model TestSignal
    // [...]
    Modelica.SIunits.Temperature T(start = 0) "Some dummy temperature";
equation
    der(T) = 0;
    // [...]
end TestSignal;
```

Snippet 3: Missing State Workaround

5.2.2 Household Model

To give a first impression of the capabilities of the approach, the energy consumption of an electric hot water boiler was modelled. The model focuses on the event generation introduced by the temperature controller of the boiler and does not claim full physical

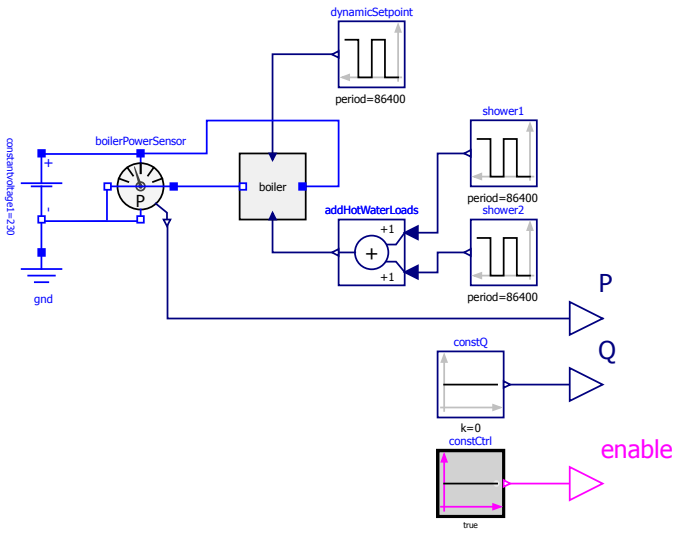


Figure 5.5: Household Model

accuracy. The construction of a detailed system model is beyond the scope of this thesis. Figure 5.4 shows the model of the hot water boiler. It maintains two ports connecting the electrical load, one port connecting the hot water usage and one port connecting the temperature set-point. The thermal mass of the fluid is modelled as a concentrated thermal mass. Any temperature gradient within the fluid is neglected. The actual heat transfer to the heated media is modelled by a `HeatedPipe` library object [57]. An `OnOffController` library block controls the temperature and switches the resistive load which heats the boiler. Additionally, heat losses via the boilers insulation are modelled.

The hot water boiler model was embedded in a simple household model which is shown in figure 5.5. The household model applies static set-points and outputs the calculated energy consumption. A transient electrical analysis is avoided by using an equivalent DC source. The temperature set-point of the boiler and the hot water consumption are approximated by step functions. The simple usage profile consists of two pulses which represent a ten minute shower each. The temperature set-point mimics an intelligent home automation system which rises the set-points in time of high photovoltaic gains.

The model is parametrized via a set of model parameters. These parameters are the physical dimensions of the boiler (radius r, height h), ambient and fluid temperatures, the heat conductivity λ_{ins} of the insulation as well as the fluid properties. It is assumed that the boiler has a cylindrical shape which is covered by the insulation. Additionally, it is assumed that the insulation material consists of $d = 10\,\text{cm}$ polyurethane rigid foam with a thermal conductivity of $\lambda_{\text{ins}} = 0.03\,\text{W}\,\text{m}^{-1}\,\text{K}^{-1}$ [40]. The heated fluid is assumed to be water. Its thermal properties are stated in the Modelica library class `Modelica.Thermal.FluidHeatFlow.Media.Water`. The heat capacity C_{storage} of the thermal storage is calculated by (5.3) where ρ corresponds to the fluid density and c_p

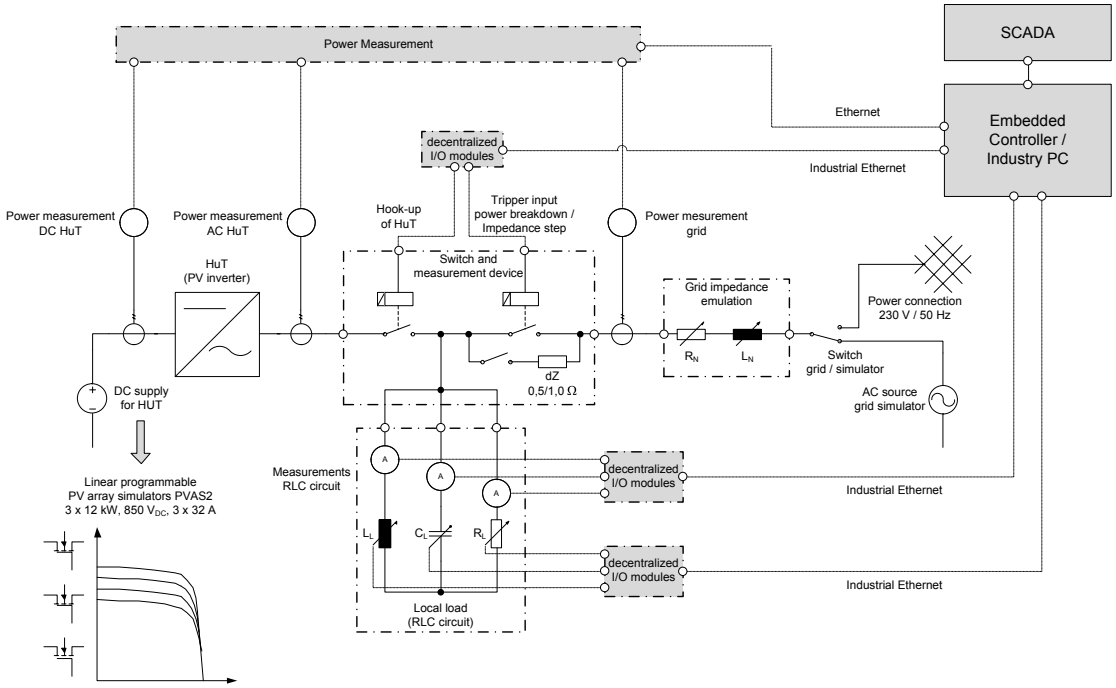


Figure 5.6: Hardware Test-Stand (Taken from [22])

corresponds to the specific heat capacity of the fluid at constant pressure.

$$C_{\text{storage}} = r^2 \cdot \pi \cdot h\rho c_p \tag{5.3}$$

According to the OpenModelica documentation, the thermal conductance G_{ins} for a cylindrical geometry of the insulation is given by (5.4) [57].

$$G_{\text{ins}} = 2\pi \cdot \lambda_{\text{ins}} h \cdot \log\left(\frac{r+d}{r}\right) \tag{5.4}$$

Figure 5.12 shows the outcome of the model in a 26.5 hour period which starts with the first hot water consumption. The model is mainly used to demonstrate the long-term stability of FMITerminalBlock and its ability to reduce triggered events. Since the set-point of the boiler is increased after 4.5 hours, the energy consumption rises at that time. The set-point is reset after three hours. The second hot water consumption period after nine hours consumes the same amount of water as the first period. Due to the increased storage temperature the consumption results in a decreased heating-up time.

5.2.3 IEC 61499-Based Infrastructure

The test-stand is described by Andrén et al. who implemented an automatic tuning of the deployed oscillatory circuit [22]. Figure 5.6 is provided by Andrén et al. and shows the test-stand setup. The HuT is driven by linear programmable PV array simulators on

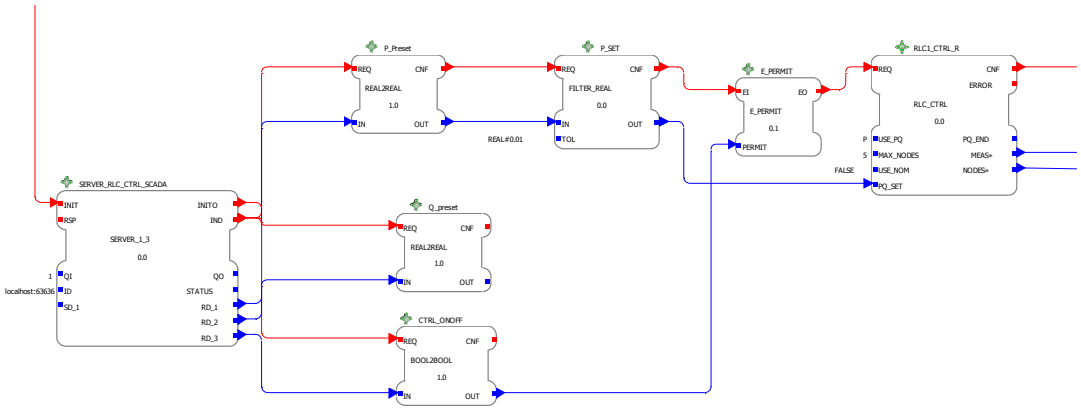


Figure 5.7: IEC 61499-Based Interface

its DC side. On the Alternating Current (AC) side of the HuT it is possible to apply several configurations. The HuT may be connected to an AC grid simulator or directly to the local grid. The oscillatory circuit may be used to apply local loads and the grid impedance emulation may emulate the properties of the power grid. An IEC 61499-based 4DIAC FORTE controller operates the oscillatory circuit and the deployed switching devices [22, 31]. It is executed on a dedicated industrial PC. The controller implements several safety measures and adjusts the oscillatory circuit of the test-stand based on given power set-points. A SCADA system which is based on ScadaBR [46] provides the user interface of the test-stand and logs recorded states. All subsystems are interconnected via ethernet and industrial ethernet respectively.

During the performed tests, the local load was connected to the local power grid and the resistive load was adjusted by applying the outputs of the test model. An HuT or any artificial grid impedance was not present. The power consumption of the load was recorded by a DEWETRON measurement station. It operates independently from the measurement equipment which is used to control the local load. The local resistive load consists of a single continuous load and several discrete loads which can be switched individually. The controller adjusts the discrete and the continuous loads in order to follow the given set-point.

Figure 5.7 shows the first few components of the load adjustment control logic [22]. It was exported by the 4DIAC IDE. Set-points are received via the server FB (which is drawn on the left hand side of figure 5.7). They are passed on for further processing such as filtering new set-points. Note that the third variable of the server contains a flag which enables the adjustment logic. Currently, only the resistive adjustment is implemented but the interface is prepared to adjust the reactive loads as well.

To test FMITerminalBlock, a local IEC 61499 application was created. The application was modeled using the 4DIAC IDE (version 1.5.3) and executed on an FBRT and a 4DIAC FORTE RTI [9, 31]. Both RTIs were distributed with the 4DIAC IDE installation. The local application receives the outcome of the simulation and displays the data of the model at an event log component. Figure 5.8 shows the FBs of the application. The

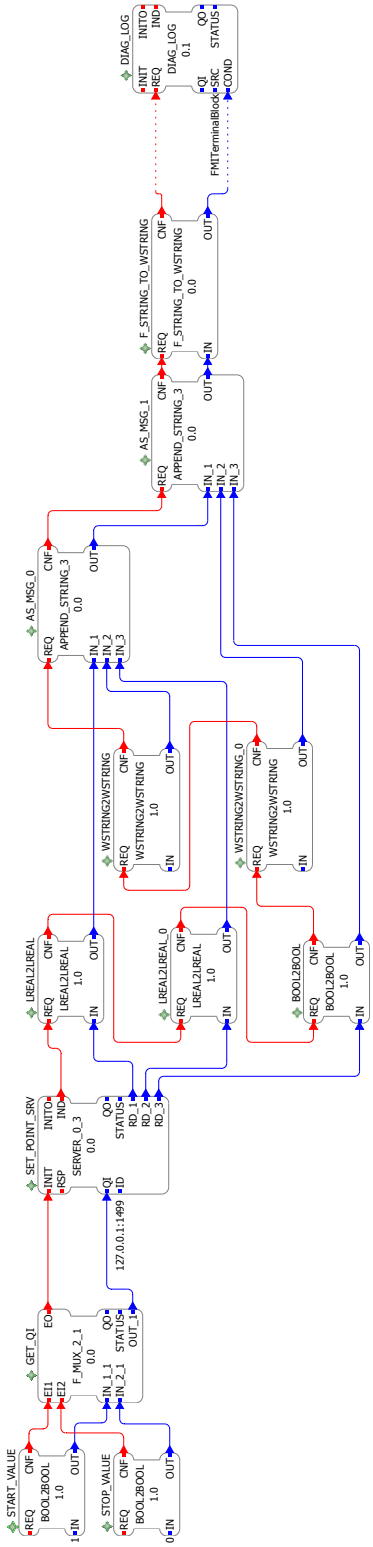


Figure 5.8: IEC 61499-Based Test Application

DIAG_LOG FB of FBRT currently does not support arbitrarily typed inputs. Hence, a data type conversion from the LREAL typed outputs of the model to the WSTRING typed inputs of the display is needed. Additionally, different variables are concatenated to a single human-readable string.

The application is implemented in a distributed way because FBRT and 4DIAC support different sets of FBs. 4DIAC provides convenient string concatenation FBs while FBRT provides HMI FBs. At first, the application was modelled from a holistic perspective. Afterwards, it was distributed to the deployed controllers. The server component and the conversion logic resides on the 4DIAC FORTE-based controller and the HMI FB (DIAG_LOG) is mapped to the FBRT-based controller. The data between both controllers is transmitted via a publisher and a subscriber FB. Since both FBs are modeled at the device level, they are not shown in figure 5.8 which displays the general application.

5.3 Timing Evaluation

When using FMITerminalBlock productively, the timing of the experiment has to be evaluated on a regular basis. Only if the observed timing deviations do not exceed application-specific limits the output of the experiment is reasonable. To simplify the timing analysis, a set of MATLAB scripts and functions was written [38]. The functions read FMITerminalBlock timing files, evaluate and display the results. To test and verify the logging facilities of FMITerminalBlock, external timing sources were used. Two experimental setups were used to evaluate the capabilities of FMITerminalBlock.

The first setup consists of the local IEC 61499 application which is described in section 5.2.3. It was executed on an office PC under Windows 7 Enterprise. In the local setup, the household model was executed and the timing was recorded by FMITerminal-Block, the FBRT controller, and Wireshark (version 1.12.3), a network protocol analyzer. To test the long term stability and to fully utilize the capabilities of the predictive approach, a prediction horizon of five minutes was chosen. Additionally, the settings of FMITerminalBlock were altered to internally store the results of the integrator in a five second interval. The current energy consumption of the boiler, a constant boolean flag, and the current temperature of the boiler were transferred to the local application. No other model variable was recorded.

By evaluating the timing records provided by FMITerminalBlock, the delay of each single event is estimated. The estimation is done by subtracting the event time-stamp from the time-stamp of the timing record at the end of the distribution stage. Every protocol implementation currently uses the synchronous `send(...)` functions of Boost to transmit data [1]. The record is added after all publisher instances are notified and the model variables are updated. Therefore, a network packet delay may be less than the recorded delay. In most use-cases, the time of the model does not directly correspond to the absolute time which is recorded in the timing file. To compare the absolute timing and the calculated time of each event, the relative time of the event is converted to an absolute time-stamp. As a reference time, the absolute time-stamp of the first event is

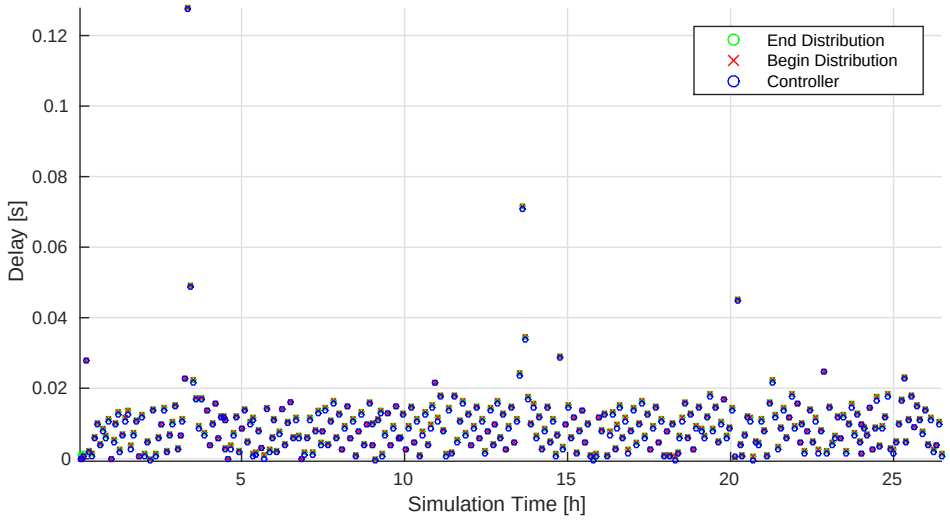


Figure 5.9: Event Delays of the Household Model

taken. In particular, the time-stamp which is recorded after the event is emitted is used as an absolute reference. Following this method, constant timing offsets introduced by the event scheduling functions can not be detected. However, since most experiments do not require an absolute timing anyway, constant timing offsets can be neglected.

Figure 5.9 shows the delay of each recorded event at different stages of the program flow. For each stage, the calculated nominal time of the event is used as the reference time. Due to the large time-span of at least 242 ms between two consecutive events, no event prediction had to be delayed, and they all finished before the scheduled time for the event. The time stamps at the beginning and the end of the event distribution were recorded by FMITerminalBlock. Since the outputs are calculated on request only, the time which is required by the distribution also includes the calculation of the outputs of the model. It turned out that most of the accumulated delay arises by late scheduling. Only 0.09% of the accumulated delay at the end of the distribution stage arose during distribution.

The delay of the controller was calculated from the event log timestamps of the controller. It includes delays which were introduced by FMITerminalBlock as well as the execution time of each involved controller. The controller records the event timing with a precision of only three decimal digits. Hence, some controller records show less delay than the corresponding distribution records. No additional delay which was added by the controllers beyond the precision limits was recorded.

In addition to each event delay, the MATLAB functions output some statistics of the recorded timing. The statistical information includes the average nominal time between two events and their absolute boundaries. The average and Root Mean Square (RMS) error as well as the absolute boundary delay values are also printed. In addition, the uppermost 1.5% of the observed delays are considered as outliers. The maximum delay

Description	Value
Nominal time between events	[242 ms, 300 s]
Average nominal time between events	294 s
Delay range	[−0.2 ms, 127.7 ms]
Mean delay	9.7 ms
RMS delay	13.8 ms
Upper 1.5% delay threshold	29.0 ms

Table 5.2: Timing Statistic of the Household Model Experiment

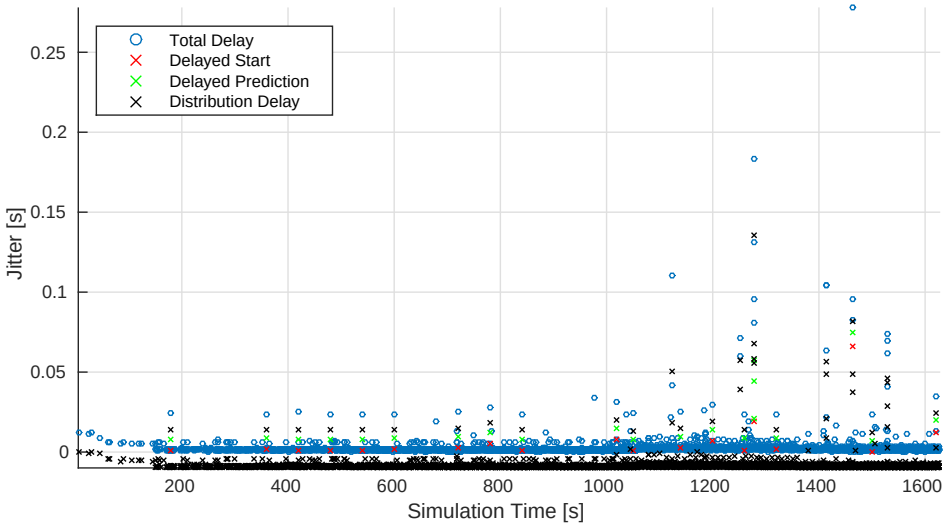


Figure 5.10: Event Delays of the Test Model (Fast Configuration)

of the remaining samples is displayed, as well. Table 5.2 shows the calculated statistics of the first experiment.

The network timing files which were recorded and exported by Wireshark show a significant clock drift. After capturing the network traffic, the files were exported via the CSV export functionality of Wireshark. The packet timing is represented as an absolute date string. After approximately 26 hours of operation, the clock of Wireshark was 1.862 s behind the clock of FMITerminalBlock. Since the timings of the controller are reasonable, it is assumed that the deviation is introduced by Wireshark.

The second experimental setup consists of the hardware test-stand and a workstation which was connected to the test-stand infrastructure. Both configurations of the test model which is described in section 5.2.1 were executed. The events' timing was recorded by FMITerminalBlock and by a Wireshark instance which was executed on the workstation. The timing records of the experiment which were generated by FMITerminalBlock were

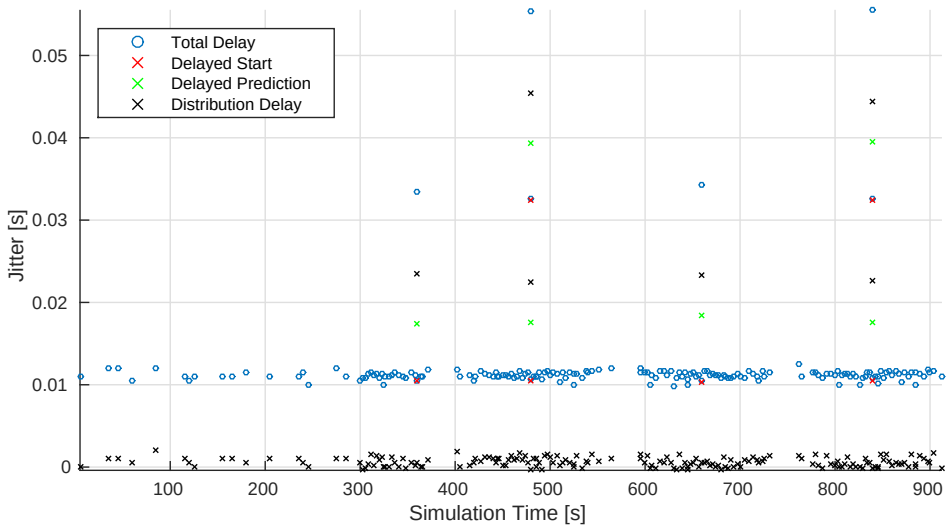


Figure 5.11: Event Delays of the Test Model (Slow Configuration)

Description	Configuration Fast	Slow
Nominal time between events	[0.05 ms, 20 s]	[0.1 ms, 30 s]
Average nominal time between events	396 ms	5.0 s
Delay range	[0.1 ms, 278 ms]	[9.8 ms, 55 ms]
Mean delay	2.7 ms	12.1 ms
RMS delay	8.1 ms	13.4 ms
Upper 1.5% delay threshold	11.1 ms	33.4 ms

Table 5.3: Timing Statistic of the Test Model Experiments

first evaluated independently from any other external timing information. The applied method follows the method which was used to evaluate the local timing of the household model. Figures 5.10 and 5.11 show the recorded event delays of the fast and the slow configuration respectively. Table 5.3 additionally lists the calculated timing statistics of both experiments.

Figures 5.10 and 5.11 plot each event delay at different processing stages. The distribution delay corresponds to the delay at the beginning of the distribution stage. The total delay corresponds to an event delay at the end of the distribution stage. If an event is already delayed before its prediction starts or after it is predicted, the delayed start and the delayed prediction marks respectively show the event delay.

Figures 5.10 and 5.11 indicate that most event delays fall inside a narrow time interval. For instance, 90% of the observed event delays from the slow experiment

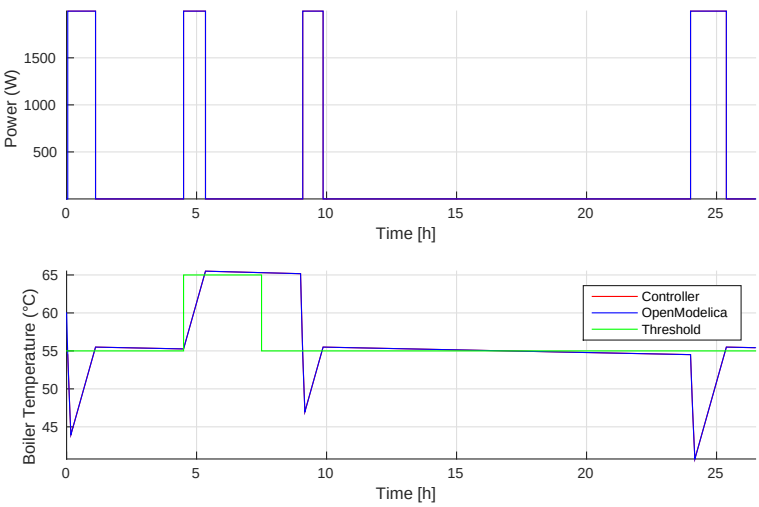


Figure 5.12: Comparison of the Outputs of the Household Model

are in $[10.1\,\text{ms}, 12.0\,\text{ms}]$. Especially large total timing delays show positive delays at every possible processing stage. Some events first got delayed at the beginning of their distribution, some events were not predicted in time and some event cycles even start delayed. Hence, no single source which caused the observed outliers can be determined.

Since the reference time is chosen by the scheduling of the first event, figure 5.10 also shows negative distribution delays. It has to be noted that negative distribution delays slightly bias the statistics presented in table 5.3. Hence, the fast configuration shows smaller absolute delay values than the slow configuration. From the perspective of a controller, actual delays may range from the distribution to the total delay values.

The Wireshark instance collected the network traffic which was sent via the TCP-based protocol to adjust the set-points. Since TCP returns an acknowledgment packet, the network delay including the processing time of the network stack can be estimated. For every event two packets have been captured, one containing the set-points and one acknowledging the first one. Additional packets which are used to establish and reset a TCP connection are neglected [5]. Like in the first experiment, the exported timing information of Wireshark showed a clock drift of 1.3 s within the first 27 minutes of the fast configuration. According to that timing information, the packets of an event were received before the corresponding event was actually triggered. Since the local setup provides strong indicators that the invalid timing is caused by Wireshark, further research on that issue is beyond the scope of this thesis.

5.4 Measurements

The errors of the local test setup are not only estimated in the time domain but also output value deviations are considered. The IEC 61499-based logging application records two

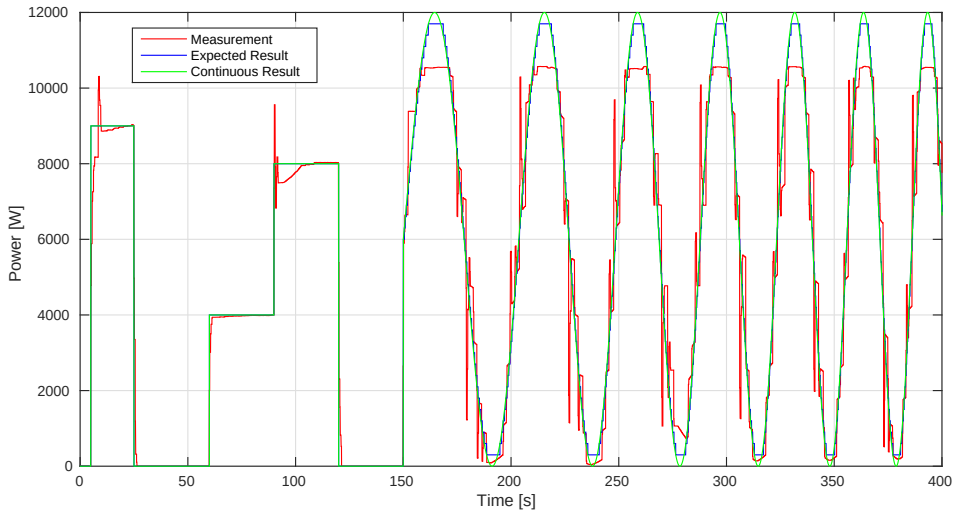


Figure 5.13: Measurement and Simulation Results (Test Model, Fast Configuration)

output values, the boiler temperature and the current power consumption. Figure 5.12 compares the notion of the output values of the controller to a reference output which was created by OpenModelica. The OpenModelica configuration uses a DASSL integrator and saves the outputs of the model in five second intervals. The results of the timing evaluation which are stated in section 5.3 lead to the expectation of accurate measurement results. Measured outputs confirm the expectations and do not tend to be erroneous. Hence, most outputs in figure 5.12 nearly overlap. After aligning the output time-stamps, an RMS deviation of 15 W for the power value and an RMS deviation of 1.5×10^{-4} K for the boiler temperature was calculated. The mean deviations of both outputs are 1.2 W and -4.2×10^{-5} K, respectively.

To evaluate the whole test-stand setup which includes the test-stand hardware, the resistive load was measured and evaluated. The deployed measurement station periodically samples the current and voltage levels of the load. Based on the measurements, it calculates the resistive and reactive power. For both experiments a sampling frequency of 10 kHz was chosen. After five voltage periods, the data was aggregated and the actual power was calculated. Hence, the measurement station sampled the power in 100 ms intervals. The measurement station and the workstation which executed the test model currently do not share a common time base, and they are not synchronized. Therefore, it is not possible to directly compare the timing of the result with the output of the model. Instead, a manually adjusted offset is used to align the first measured edge.

Figure 5.13 shows the first 400 s of the results of the fast configuration. Additionally, the expected outcome and the continuous output are presented. The expected outcome was generated by OpenModelica which features DASSL integration. The step function at the beginning shows that the controller first adjusts the discrete loads [22]. As soon as the discrete loads are adjusted, the continuous load is used to fine-tune the power

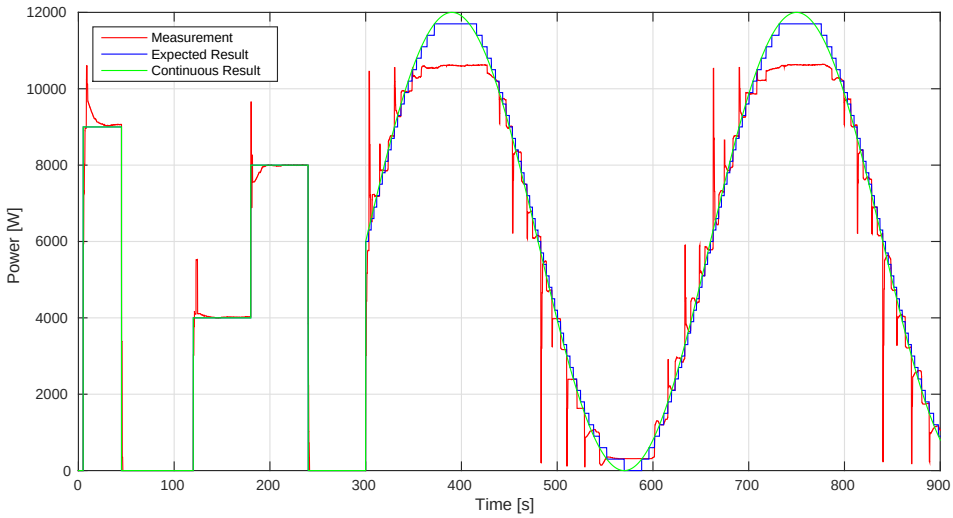


Figure 5.14: Measurement and Simulation Results (Test Model, Slow Configuration)

consumption. The actual power is often temporarily higher than the intended set-point. The transient overshoot is a result of the implemented control algorithm. Since both configurations intend to challenge the presented setup and frequently adjust the set-points, transient overshoots are observed regularly. During the measurements it turned out that the controller restricts the loads' power consumption to $\approx 10.5\,\mathrm{kW}$. Hence, the actual maximum power is below the assumed maximum power of $P_{\max} = 12\,\mathrm{kW}$.

In the first experiment which uses the fast configuration, the first 1630 s of the model execution were recorded. After aligning expected results and recorded measurements, the RMS and the average error were calculated. The measurement shows an RMS deviation of 1.7 kW and an average deviation of −49 W. Since the actual maximum power consumption biases the result, the reference was adjusted. The new reference features a maximum power of 10.5 kW. Adjusted statistics were calculated based on the newly calculated values. The adjusted RMS of the experiment is 1.6 kW and the adjusted mean value is 227 W.

The outcome of the slow configuration was also recorded and evaluated. Figure 5.14 shows the first 900 s of the calculated output as well as corresponding measurements. The increased time constants allow a better adoption to given set-points. Hence, the unadjusted RMS error is only 589 W and the unadjusted average error is −174 W. The adjusted reference shows an RMS error of 506 W and an average error of 83 W. The glitches which result from an adoption of the set-points still exist. Nevertheless, in particular for the step functions, set-point changes are triggered less frequently.

Figures 5.13 and 5.14 indicate that the average set-point adoption time of the test-stand is much larger than every observed event delay. Hence, a precise measurement of the timing of FMITerminalBlock which is based on the measured power consumption is

not feasible. As a direct consequence, it is expected that the timing of FMITerminalBlock will allow reasonable experiments at the presented test-stand.

CHAPTER 6

Conclusion and Outlook

Many use-cases require the coupling of different simulation tools or models. By combining IEC 61499-based control infrastructure with plant models new prospects arise. Control infrastructure can be developed and tested with a minimal effort in porting the control logic to a productive environment. The integration of simulation models in IEC 61499-based infrastructure enables dynamically generated set-points of an IEC 61499-based inverted test-stand or advanced controller HIL experiments. Current tool coupling approaches which include IEC 61499-based controller most often require labor intensive configurations and interface implementations. The FMI standard which specifies model exchange or co-simulation interfaces does not impose the need of implementing tool-specific interfaces. Hence, it enables a fluent development work-flow.

A predictive approach which integrates FMI-based models into IEC 61499-based infrastructure is implemented and evaluated. It uses an ASN.1-based communication protocol to transmit the outputs of the model to the control infrastructure. This thesis describes the intended program flow and the user interface. It also states the chosen software architecture. A third party library called FMI++ is used to interface FMI-based models. Hence, the software design focuses on the user interface, timing and data transmission. To evaluate the best-effort approach, a timing interface is implemented. The interface outputs the program timing information and records several time-stamps during the operation. It turned out that the usage of external software and software libraries such as Boost, FMI++ or CMake enhanced the development. The software already provided several basic functions on which the interface software is based.

The implemented interface program is tested by a set of unit test-cases to improve the program code quality. The automated tests enable frequent regression tests. Furthermore, the program and the implemented approach were evaluated in different experiments. Two test setups in different configurations were deployed. One includes the test-stand hardware and one includes a local IEC 61499 test application which records the output of the program. Additionally, a network protocol analyzer was used to trace generated network traffic. It turned out that the network protocol analyzer suffered from a large clock drift

which biases the results. Independent timing records were made by the interface program and the local test application. They strongly indicate that the deviation is based on the timings of the network protocol analyzer. Hence, the timing source was not used to evaluate the timing of the interface program.

Two different models were used to evaluate the program. One outputs a stateless time-dependent function and one contains a simple model of a hot water boiler. The boiler model was executed by the local application. Most delays in that setup originated from the operating system scheduling. At the local experiment, an RMS delay of 13.8 ms was achieved. Delays at the test-stand which outputs the results of the test model were dominated by the network delay and the delay caused by output calculations. In this setup, the timing of the interface program exceeds the precision of the adaptive load by far.

The output of the local test application showed only little error compared to a reference simulation. The deviation is mostly caused by numerical errors. The error in the measured power curve is dominated by the adjustment time of the test-stand. Each deployed electromechanical component requires a notable time-span to shift its state. In both setups, the implemented approach performs well and outputs predicted events in time. In rare circumstances, the best-effort implementation showed event delays up to 278 ms which are far beyond the average delays. Nevertheless, the observed maximum delays are far below the adjustment time of the test-stand.

Although this thesis presents first promising results, future research and engineering is still necessary. For instance, a systematic discussion on combining the FMI and IEC 61499-based controllers is still missing. The interface program may also be extended by an updating functionality which changes the inputs of a model based on the outputs of a controller. The resulting closed loop system would require separate evaluation. Additionally, exhaustive tests which include large models and more realistic use-cases are still missing. To further improve the interface program code quality, test-cases from an independent tester are desirable.

Bibliography

[1] Christopher M. Kohlhoff. *Boost.Asio*. 2014. URL: http://www.boost.org/doc/libs/1_56_0/doc/html/boost_asio.html (visited on 10/29/2014).

[2] Feng Guo, Luis Herrera, Robert Murawski, Ernesto Inoa, Chih-Lun Wang, Philippe Beauchamp, Eylem Ekici, and Jin Wang. „Comprehensive Real-Time Simulation of the Smart Grid“. In: *Industry Applications, IEEE Transactions on* 49.2 (Mar. 2013), pp. 899–908. ISSN: 0093-9994. DOI: 10.1109/TIA.2013.2240642.

[3] Brian Fox and Chet Ramey. *GNU Bourne-Again SHell. Linux man page*. Free Software Foundation, Inc. URL: http://linux.die.net/man/1/bash (visited on 12/17/2014).

[4] *ITU-T X.680: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. International Telecommunication Union, ITU, Nov. 2008. DOI: 11.1002/1000/9604.

[5] *RFC 793: TRANSMISSION CONTROL PROTOCOL*. Information Sciences Institute. Sept. 1981. URL: https://tools.ietf.org/html/rfc793 (visited on 02/05/2015).

[6] *IEC 61499-1/Ed.2: Function blocks - Part 1: Architecture*. International Electrotechnical Commission, IEC, Nov. 2012. URL: http://www.iec.ch/.

[7] *FMI Support in Tools*. Modelica Association Project. 2014. URL: https://www.fmi-standard.org/tools (visited on 09/29/2014).

[8] *OMG Unified Modeling Language^{TM} (OMG UML), Superstructure*. Version 2.3. Object Management Group, Inc. 2010. URL: http://www.omg.org/spec/UML/2.3/Superstructure/PDF (visited on 02/19/2015).

[9] *FBDK - The Function Block Development Kit*. Holobloc Inc. Mar. 2011. URL: http://www.holobloc.com/doc/fbdk/index.htm (visited on 02/04/2015).

[10] Chia-Han Yang, Gulnara Zhabelova, Chen-Wei Yang, and Valeriy Vyatkin. „Cosimulation Environment for Event-Driven Distributed Controls of Smart Grid“. In: *Industrial Informatics, IEEE Transactions on* 9.3 (Aug. 2013), pp. 1423–1435. ISSN: 1551-3203. DOI: 10.1109/TII.2013.2256791.

[11] Edmund Widl. *FMI++. A High-level Utility Package for FMI for Model Exchange*. 2013. URL: http://sourceforge.net/projects/fmipp/?source=navbar (visited on 01/07/2015).

[12] Walter Gora. *ASN.1. abstract syntax notation one.* 3., aktual. Aufl. DATACOM-Fachbuchreihe. Bergheim: DATACOM-Buchverl., 1992. ISBN: 3-89238-062-7.

[13] Edmund Widl, Wolfgang Müller, Atiyah Elsheikh, Matthias Hörtenhuber, and Peter Palensky. „The FMI++ library: A high-level utility package for FMI for model exchange“. In: *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on.* 2013, pp. 1–6. DOI: 10.1109/MSCPES.2013.6623316.

[14] *IEEE Standard for Binary Floating-Point Arithmetic.* 1985. DOI: 10.1109/IEEESTD.1985.82928.

[15] *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules.* 2010, pp. 1–38. DOI: 10.1109/IEEESTD.2010.5553440.

[16] Jan Lunze. *Regelungstechnik 2; Mehrgrößensysteme, Digitale Regelung.* Springer-Lehrbuch. Berlin, Heidelberg: Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN: 978-3-642-10197-7. DOI: 10.1007/978-3-642-10198-4.

[17] Jan Lunze. *Regelungstechnik 1; Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen.* 9., überarbeitete Aufl. 2013. Springer-Lehrbuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-29533-1.

[18] Ingo Hegny. „Development and simulation framework for industrial production systems“. Technische Universität Wien, 2014.

[19] Ingo Hegny, Alois Zoitl, and Wilfried Lepuschitz. „Integration of simulation in the development process of distributed IEC 61499 control applications“. In: *Industrial Technology, 2009. ICIT 2009. IEEE International Conference on.* Feb. 2009, pp. 1–6. DOI: 10.1109/ICIT.2009.4939681.

[20] Ingo Hegny, Monika Wenger, and Alois Zoitl. „IEC 61499 based simulation framework for model-driven production systems development“. In: *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on.* Sept. 2010, pp. 1–8. DOI: 10.1109/ETFA.2010.5641364.

[21] Sven Christian Müller Müller, Hanno Georg, Markus Küch, and Christian Wietfeld. „INSPIRE - Co-Simulation of Power and ICT Systems for Evaluation of Smart Grid Applications“. In: *At-Automatisierungstechnik* 62(5) (Apr. 2014), 315–324. ISSN: 0178-2312. DOI: 10.1515/auto-2014-1086.

[22] Filip Andrén, Felix Lehfuß, and Thomas Strasser. „A DEVELOPMENT AND VALIDATION ENVIRONMENT FOR REAL-TIME CONTROLLER-HARDWARE-IN-THE-LOOP EXPERIMENTS IN SMART GRIDS“. In: *International Journal of Distributed Energy Resources and Smart Grids* 9.1 (Hardware-in-the-loop Testing July 2013), pp. 27–50. ISSN: 1614-7138.

[23] Filip Andrén, Matthias Stifter, and Thomas Strasser. „Towards a Semantic Driven Framework for Smart Grid Applications: Model-Driven Development Using CIM, IEC 61850 and IEC 61499“. English. In: *Informatik-Spektrum* 36.1 (2013), pp. 58–68. ISSN: 0170-6012. DOI: 10.1007/s00287-012-0663-y.

[24] Beman Dawes and David Abrahams. *Boost C++ Libraries*. 2005. URL: http://www.boost.org/ (visited on 01/07/2015).

[25] Stefan Biffl, Alexander Schatten, and Alois Zoitl. „Integration of heterogeneous engineering environments for the automation systems lifecycle“. In: *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*. 2009, pp. 576–581. DOI: 10.1109/INDIN.2009.5195867.

[26] Bernhard Rumpe. *Modellierung mit UML. Sprache, Konzepte und Methodik*. 2nd ed. Xpert.press. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2011. ISBN: 9783642224133. DOI: 10.1007/978-3-642-22413-3.

[27] Wolfgang Müller and Edmund Widl. „Linking FMI-based components with discrete event systems“. In: *Systems Conference (SysCon), 2013 IEEE International*. 2013, pp. 676–680. DOI: 10.1109/SysCon.2013.6549955.

[28] Muhammad Usman Awais, Peter Palensky, Wolfgang Mueller, Edmund Widl, and Atiyah Elsheikh. „Distributed hybrid simulation using the HLA and the Functional Mock-up Interface“. In: *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*. 2013, pp. 7564–7569. DOI: 10.1109/IECON.2013.6700393.

[29] Christoph Sünder and Valeriy Vyatkin. *O^{3}neida Workgroup on Execution Models of IEC 61499 Function Block Applications*. O^{3}neida. Oct. 2011. URL: http://www.oooneida.org/standards_development_Compliance_Profile.html (visited on 08/13/2014).

[30] *CMake 3.1.0 Documentation*. Kitware, Inc. URL: http://www.cmake.org/cmake/help/v3.1/ (visited on 01/07/2015).

[31] *4DIAC: FORTE*. PROFACTOR GmbH. URL: http://www.fordiac.org/8.0.html (visited on 09/18/2014).

[32] Pete Becker. *The C++ standard library extensions. a tutorial and reference*. Upper Saddle River, NJ, July 2006.

[33] Rene Rivera. *Boost Predef library*. June 2014. URL: http://www.boost.org/doc/libs/1_56_0/libs/predef/doc/html/index.html (visited on 11/24/2014).

[34] Samuel Krempp. *Boost Format library*. Dec. 2006. URL: http://www.boost.org/doc/libs/1_56_0/libs/format/ (visited on 11/24/2014).

[35] Kevlin Henney. *Boost.Any*. 2001. URL: http://www.boost.org/doc/libs/1_56_0/doc/html/any.html (visited on 11/24/2014).

[36] Hermann Kopetz. *Real-Time Systems; Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Boston, MA: Springer Science+Business Media, LLC, 2011. ISBN: 9781441982377. DOI: 10.1007/978-1-4419-8237-7.

[37] *Global Greenhouse Gas Emissions Data*. EPA, United States Environmental Protection Agency. 2013. URL: http://www.epa.gov/climatechange/ghgemissions/global.html (visited on 02/13/2015).

[38] *MATLAB Documentation*. MathWorks, Inc. 2014. URL: http://de.mathworks.com/help/matlab/index.html (visited on 12/01/2014).

[39] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. „GNU Autoconf, Automake, and Libtool. expert insight into porting software and building large projects using GNU Autotools“. In: 1st ed. Indianapolis, Ind.: New Riders, 2001. Chap. Writing 'configure.in'. ISBN: 978-1-57870-190-2. URL: https://www.sourceware.org/autobook/autobook/autobook_toc.html (visited on 01/07/2015).

[40] Alfons Oebbeke. *Dämmstoff Magazin: Wärmeleitzahlen / λ-Werte*. ARCHmatic. 2015. URL: http://www.baulinks.de/baumaterial/lambda-werte-waermeleitzahl-waermeleitfaehigkeit-waermedaemmung.php (visited on 02/03/2015).

[41] Valeriy Vyatkin. „The IEC 61499 standard and its semantics“. In: *Industrial Electronics Magazine, IEEE* 3.4 (Dec. 2009), pp. 40–48. ISSN: 1932-4529. DOI: 10.1109/MIE.2009.934796.

[42] Valeriy Vyatkin. „IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review“. In: *Industrial Informatics, IEEE Transactions on* 7.4 (Nov. 2011), pp. 768–781. ISSN: 1551-3203. DOI: 10.1109/TII.2011.2166785.

[43] Michael H. Spiegel, Fabian Leimgruber, Edmund Widl, and Günther Gridling. „On using FMI-based models in IEC 61499 control applications“. In: *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2015 Workshop on*. 2015, pp. 1–6. DOI: 10.1109/MSCPES.2015.7115407.

[44] Matthias Stifter, Edmund Widl, Filip Andrén, Atiyah Elsheikh, Thomas Strasser, and Peter Palensky. „Co-simulation of components, controls and power systems based on open source software“. In: *Power and Energy Society General Meeting (PES), 2013 IEEE*. July 2013, pp. 1–5. DOI: 10.1109/PESMG.2013.6672388.

[45] *Layered Model-View-Control Design Pattern*. Holobloc Inc. Jan. 2011. URL: http://www.holobloc.com/doc/despats/mvc/ (visited on 08/12/2014).

[46] *ScadaBR English Summary*. URL: http://sourceforge.net/p/scadabr/wiki/Manual%20ScadaBR%20English%200%20Summary/.

[47] *TechNet about_Quoting_Rules*. Microsoft. May 2014. URL: http://technet.microsoft.com/en-us/library/hh847740.aspx (visited on 12/17/2014).

[48] Greg Ippolito. *Endianness: Big and Little Endian Byte Order*. 2013. URL: http://www.yolinux.com/TUTORIALS/Endian-Byte-Order.html (visited on 11/24/2014).

[49] Andrey Semashev. *Chapter 1. Boost.Log v2*. 2014. URL: http://www.boost.org/doc/libs/1_56_0/libs/log/doc/html/index.html (visited on 01/19/2015).

[50] Martin Schlager. „Interface Design for Hardware-in-the-Loop Simulation of Real-Time Systems“. Disertation. Technischen Universität Wien, Fakultät für Informatik, Sept. 2007.

[51] Thomas Strasser, Matthias Stifter, Filip Andrén, and Peter Palensky. „Co-Simulation Training Platform for Smart Grids“. In: *Power Systems, IEEE Transactions on* 29.4 (July 2014), pp. 1989–1997. ISSN: 0885-8950. DOI: 10.1109/TPWRS.2014.2305740.

[52] Thomas Strasser, Alois Zoitl, James H. Christensen, and Christoph Sünder. „Design and Execution Issues in IEC 61499 Distributed Automation and Control Systems“. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 41.1 (Jan. 2011), pp. 41–51. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2010.2067210.

[53] Thomas Strasser, Matthias Stifter, Filip Andrén, Daniel Burnier de Castro, and Wolfgang Hribernik. „Applying open standards and open source software for smart grid applications: Simulation of distributed intelligent control of power systems“. In: *Power and Energy Society General Meeting, 2011 IEEE*. July 2011, pp. 1–8. DOI: 10.1109/PES.2011.6039314.

[54] Anthony Williams and Vicente J. Botet Escriba. *Boost Thread 4.3.0*. 2014. URL: http://www.boost.org/doc/libs/1_56_0/doc/html/thread.html (visited on 01/07/2015).

[55] Gennadiy Rozental. *Boost Test Library*. 2007. URL: http://www.boost.org/doc/libs/1_56_0/libs/test/doc/html/index.html (visited on 01/07/2015).

[56] *Standard C++ Library reference*. cplusplus.com. 2014. URL: http://www.cplusplus.com/reference/ (visited on 01/07/2015).

[57] *Modelica Documentation*. OpenModelica. 2015. URL: https://build.openmodelica.org/Documentation/ (visited on 01/27/2015).

[58] Ralf Schneeweiß. *Moderne C++ Programmierung. Klassen, Templates, Design Patterns*. Xpert.press. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2006. ISBN: 9783540459545. DOI: 10.1007/3-540-45954-5.

[59] Gerhard Doblinger. *Zeitdiskrete Signale und Systeme; eine Einführung in die grundlegenden Methoden der digitalen Signalverarbeitung*. 2., überarb. Aufl. Wilburgstetten: Schlembach, 2010. ISBN: 978-3-935340-66-3.

[60] Torsten Blochwitz, Martin Otter, Martin Arnold, C Bausch, Christoph Clauß, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, M Monteiro, T Neidhold, et al. „The functional mockup interface for tool independent exchange of simulation models“. In: *8th International Modelica Conference, Dresden*. 2011, pp. 20–22. URL: https://www.modelica.org/events/modelica2011/Proceedings/pages/papers/05_1_ID_173_a_fv.pdf.

[61] *FACTORIES OF THE FUTURE. Multi-annual roadmap for the contractual PPP under Horizon 2020*. European Commission. 2014. URL: http://www.effra.eu/attachments/article/129/Factories\%20of\%20the\%20Future\%202020\%20Roadmap.pdf (visited on 02/12/2015).

[62] Karl Eilebrecht and Gernot Starke. *Patterns kompakt. Entwurfsmuster für effektive Software-Entwicklung*. Heidelberg: Spektrum Akademischer Verlag, 2010. ISBN: 3827425255. DOI: 10.1007/978-3-8274-2526-3.

[63] Marcin Kalicinski. *Boost.PropertyTree*. 2008. URL: http://www.boost.org/doc/libs/1_56_0/doc/html/property_tree.html (visited on 01/07/2015).

[64] Alexander Viehweider, Georg Lauss, and Felix Lehfuss. „Stabilization of Power Hardware-in-the-Loop simulations of electric energy systems“. In: *Simulation Modelling Practice and Theory* 19.7 (2011), pp. 1699 –1708. ISSN: 1569-190X. DOI: 10.1016/j.simpat.2011.04.001.

[65] *Functional Mock-up Interface for Co-Simulation*. Version 1.0. Modelica Association Project. MODELISAR consortium, Oct. 2010. URL: https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_CoSimulation_v1.0.pdf (visited on 07/07/2014).

[66] *Functional Mock-up Interface for Model Exchange*. Version 1.0. Modelica Association Project. MODELISAR consortium, Jan. 2010. URL: https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_ModelExchange_v1.0.pdf (visited on 08/15/2014).

[67] *Functional Mock-up Interface for Model Exchange and Co-Simulation*. Version 2.0. Modelica Association Project. MODELISAR consortium, July 2014. URL: https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf (visited on 08/15/2014).

[68] *Technology Roadmap. Solar Photovoltaic Energy*. International Energy Agency. 2014. URL: http://www.iea.org/media/freepublications/technologyroadmaps/solar/TechnologyRoadmapSolarPhotovoltaicEnergy_2014edition.pdf (visited on 02/13/2015).

[69] *Technology Roadmap. Wind Energy*. International Energy Agency. 2013. URL: http://www.iea.org/publications/freepublications/publication/Wind_2013_Roadmap.pdf (visited on 02/13/2015).

[70] *Technology Roadmap. Smart Grids*. International Energy Agency. 2011. URL: http://www.iea.org/publications/freepublications/publication/smartgrids_roadmap.pdf (visited on 02/13/2015).

[71] *https://openmodelica.org/*. Modelica Association. 2015. URL: http://sourceforge.net/projects/fmipp/?source=navbar (visited on 01/19/2015).