ACIN
AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS- & REGELUNGSTECHNIK

A Service-Oriented Domain Specific Language Programming Approach for Batch Processes

DIPLOMARBEIT

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Diplom-Ingenieurs (Dipl.-Ing.)

unter der Leitung von

Univ.-Prof. Dr.sc.techn. Georg Schitter
Dipl.-Ing. Martin Melik-Merkumians

eingereicht an der

Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik
Institut für Automatisierungs- und Regelungstechnik

von

Matthias Baierling
Matrikelnummer: 0826680

Wien, im August 2016

Gruppe für Industrielle Automationstechnik
Gußhausstraße 27-29, A-1040 Wien, Internet: http://www.acin.tuwien.ac.at

Abstract

Nowadays, major challenges of the manufacturing industry are the shortening of product life-cycles and the trend towards personalized productions. This trend is a tremendous turnaround in the industry, since the last decades were coined by mass productions at low costs. Although the demands of current products increased, the price level should stay the same. Therefore, the improvement of flexibility of automation systems is a current issue, in order to enable the implementation of product specific changes with little effort.

This thesis targets this issue by developing and implementing a concept for a flexible process control, based on a domain specific tool, which enables the creation of production recipes by process engineers, without reconfiguring the plant. In order to control the equipment, services in terms of a Service-Oriented Architecture are provided. Based on the recipe, a recipe processing algorithm calls the corresponding services from the equipment. For demonstrating purposes, this concept is evaluated on two linked laboratory tank system plants, which enable process actions for pumping, heating, and mixing. Furthermore, for improving the concept of a flexible process control even more, a dynamic route finding algorithm is integrated, which finds the shortest path for pumping processes based on a model of the plant.

Zusammenfassung

Die Verkürzung von Produktlebenszyklen sowie die kundenspezifische Nachfrage nach individuellen Produkten stellt industrielle Produktionsbetriebe derzeit vor große Herausforderungen. Waren früher noch hohe Produktmengen für niedrige Produktionskosten entscheidend, wird derzeit und in Zukunft eine Individualisierung bei ähnlichem Preisniveau von Kunden gefordert. Dies macht es in weiterer Folge notwendig, die Steuerungssysteme der Produktionsanlagen hinsichtlich Flexibilität zu verbessern um Änderungen mit möglichst wenig Aufwand und in kurzer Zeit implementieren zu können.

Ziel dieser Diplomarbeit ist es, ein Konzept zu entwickeln und implementieren, welches einem Anlagenbetreiber ermöglicht Produktionsrezepte zu erstellen, ohne die Anlage selbst aufwendig umprogrammieren zu müssen. Dafür wird ein Tool entwickelt, welches eine einfache Möglichkeit zur Definition von Produktionsrezepten bereitstellt und diese anschließend ohne weiteren Aufwand abgearbeitet werden können. Das zu steuernde Equipment stellt dafür Services im Sinne einer serviceorientierten Architektur zur Verfügung, die vom Abarbeitungsalgorithmus des Rezepts angesprochen werden. Dieses Konzept wird anhand eines Labortanksystems gezeigt und evaluiert, mit dem Pump-, Heiz- und Mixprozesse durchgeführt werden können. Zur Verbesserung der Flexibilität wird außerdem ein dynamischer Pfadfindungsalgorithmus integriert, der für Pumpprozesse anhand des tatsächlichen Tanksystems den kürzesten Pfad ermittelt.

Contents

List of Figures

List of Tables

Acronyms

4diac Framework for Industrial Automation & Control

API Application Programming Interface

ASN.1 Abstract Syntax Notation One

BNF Backus - Naur Form

CFB Composite Function Block

CM Control Module

CPS Cyber-Physical System

CSS Cascading Style Sheets

DCS Distributed Control System

DDS Data Distribution Service

DPWS Devices Profile for Web Services

DSL Domain Specific Language

EBNF Extended Backus - Naur Form

EM Equipment Module

EMF Eclipse Modeling Framework

ERP Enterprise Resource Planning

FB Function Block

FBD Function Block Diagram

FIFO First-In First-Out

FMS Flexible Manufacturing System

FORTE 4diac Runtime Environment

GDS Global Data Space

GPL General Purpose Language

GUI Graphical User Interface

GVL Global Variable List

HMI Human Machine Interface

HMS Holonic Manufacturing System

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

IEC 61131 IEC 61131 – Programmable controllers

IEC 61499 IEC 61499 – Function Blocks

IEC 61512 IEC 61512 – Batch control

IEC 62264 IEC 62264 – Enterprise-control system integration

IEC 62541 IEC 62541 – OPC Unified Architecture

IL Instruction List

IMC-AESOP ArchitecturE for Service-Oriented Process - Monitoring and Control

IoT Internet of Things

IPC Industrial PC

ISA International Society of Automation

ISO/IEC 14977 ISO/IEC 14977: Information technology – Syntactic metalanguage – Extended BNF

IT Information Technology

LD Ladder Diagram

M2M Machine-to-Machine

MAS Multi-Agent System

MEP Message Exchange Pattern

MES Manufacturing Execution System

MOM Message-Oriented Middleware

MONACO MOdeling Notation for Automation COntrol

MQ Message Queue

MQTT Message Queue Telemetry Transport

OOP Object-Oriented Programming

OPC UA OPC Unified Architecture

P&ID Piping and Instrumentation Diagram

PFC Procedure Function Chart

PLC Programmable Logic Controller

QoS Quality of Service

RPC Remote Procedure Call

SCADA Supervisory Control and Data Acquisition

SFC Sequential Function Chart

SIFB Service Interface Function Block

SIPN Signal Interpreted Petri Net

SIRENA Service Infrastructure for Real-time Embedded Networked Applications

SOA Service-Oriented Architecture

SOCRADES Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded deviceS

SQL Structured Query Language

SSL Secure Sockets Layer

ST Structured Text

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

WSN Wireless Sensor Network

XML eXtensible Markup Language

CHAPTER 1

Introduction

In the last two centuries, production methods have fundamentally changed. Initiated by the migration from the countryside as well as the invention of new machines (e. g. the Flying-Shuttle-Loom by John Kay or the essential efficiency improvement of the steam-engine by James Watt), resulted in the first industrial revolution. These machines became an integral part of craft productions and characterized the production process in the second half of the 19^{th} century. [1]–[4]

At the beginning of the 20^{th} century, electrification revolutionized the equipment in factories and initiated the second industrial revolution. Some important changes were the replacement of steam-engines by electrical machines, the development of a basic automation control based on relay logic and the introduction of assembly lines. These inventions enabled mass-productions with the potential to increase manufactured quantities at low costs. [1]–[4]

The third industrial revolution (also called *digital revolution*) was introduced by an invention of Richard Morley (`Modicon`) and Odo Struger (`Allen Bradley`), in the late 1960's and early 1970's, respectively. In order to enhance flexibility and get rid of the inflexible relay logic, they developed a digital programmable microcontroller, which is used for controlling actions in industrial environments (e. g. exposed to dirt, vibrations, or electromagnetic disturbances). Struger termed this device Programmable Logic Controller (PLC). This novel type of controlling device initiated a shift of the production paradigm from mass-productions to mass-customizations. It allowed an expansion of the product variety based on consumer demands. However, this can only be achieved economically, if the plant is reconfigurable without big effort. [1]–[4]

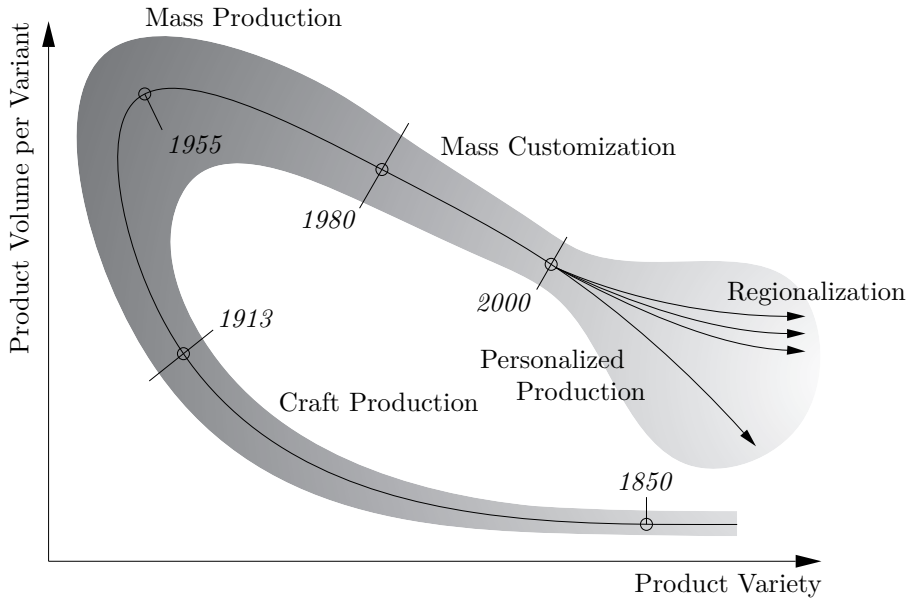


Figure 1.1: In the course of time, the production paradigm shifted due to technical abilities and society needs, as presented by Koren [2]. The year dates describe the shift in the automotive industry in the Western world.

In the year 1995, a next developmental step towards more flexible productions, was made by the International Society of Automation (ISA). They introduced the standard ISA-88, which was adopted as IEC 61512 – Batch control [5], in 1997. The major feature of this standard is the separation of production equipment capabilities from process recipe information. The standard describes a manufacturing method, called *batch processes*, that produces finite output quantities. In contrast to discrete or continuous productions, batch processes have a dedicated beginning and a defined end. However, the design principles, described in the standard, can also be adapted to discrete or continuous productions [6].

Figure 1.1 shows the paradigm shift of manufacturing methods over time, with approximate year dates, that are explicitly valid for the automobile industry in the Western world, but similar for other industries [2]. Customization is still in progress and currently turning into a personalized production, which means that the batch size shifts towards one. Therefore, fast-reconfigurable automation systems are requested. However, current systems have not been designed for personalized production, so an individual reconfiguration would be too laborious and time intensive [7], [8]. Consequentially, one of the most important features of current automation systems is *flexibility*. Systems that are designed to be flexible, are termed Flexible Manufacturing System (FMS). Since this keyword is used in different scopes, Browne [9] classified them concerning their field of usage. Similar concepts, which are designed to handle complex automation systems by a modular composition of intelligent components, are Multi-Agent System (MAS) and Holonic Manufacturing System (HMS) [10]. These concepts are based on distributed systems of autonomous components, that interact with each other to process a specific task.

Currently, there is a trend towards a fourth industrial revolution, also termed *Industrie 4.0*. This term was coined by the German government, in the context of the *High-Tech Strategy 2020* [11]. It describes the vision of industrial technologies of the future. Major components of Industrie 4.0 are autonomously interacting physical devices that are networked to a Internet of Things (IoT). Furthermore, this physical world is fused with the virtual world by Cyber-Physical System (CPS). This is done by the integration of smart devices into all technical environments, like smart factories, smart grids, or smart mobility. Concerning to smart factories, Industrie 4.0 tries to enhance production processes and facilitate personalized productions by applying design principles like modularization, decentralization, or service-orientation. [12]

1.1 Scope of the Thesis

This thesis focuses on a flexible, modular and fast-reconfigurable control system for industrial batch processes, which aims to enhance productivity by reducing planned downtimes. This shall be reached by using design principles of Industrie 4.0 (modularization, decentralization, and service-orientation) in combination with a recipe based controlling approach, similar to IEC 61512.

For this, in a first step, a recipe creation tool for describing batch processes is developed. This tool targets to plant operators as users, and should therefore work without taking a hand into the controlling program of the equipment. This requires a recipe creation tool, which is specific to the domain of the plant and easy to handle. In order to execute the single processing steps of the recipe, a PLC, where all the electrical equipment of the plant is connected to, provides several services in terms of a Service-Oriented Architecture, for reading sensor values and manipulating the actuators of the plant.

For demonstration purposes, this principle is evaluated on two linked laboratory tank system plants with seven tanks connected by partially redundant pipes, illustrated in Figure 1.2. Due to this redundancy, a second step towards a flexible batch control is a dynamic route finding, based on a model of the plant.

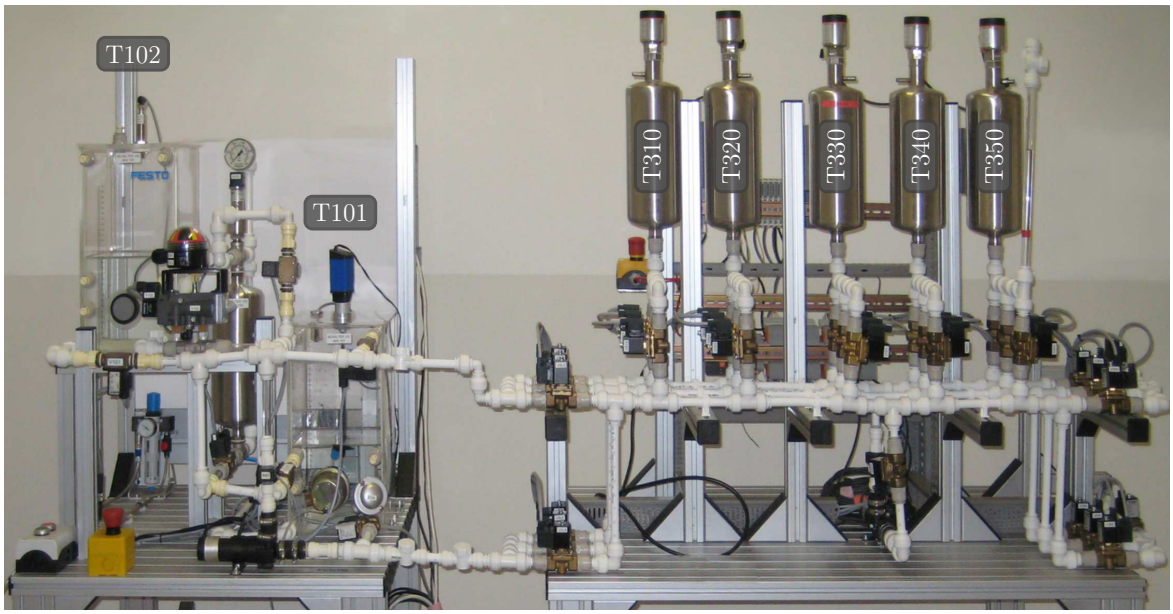


Figure 1.2: The equipment consists of two linked laboratory tank systems, a Festo Didactic process industry demonstrator plant (left) with two tanks and a custom built storage tank system (right) with five tanks. The overlaid labels are referenced later in this work. Each system has a pump, several valves, and all tanks but one have a level sensor. Additionally, the right tank of the left plant (T101) has a heating element and a temperature sensor, as well as an agitator.

1.2 Thesis Outline

In this thesis, a flexible and modular control for batch processes will be developed. In the following chapter, an overview of the current state of research concerning this and related subjects is presented. Afterwards, in Chapter 3, the concept of the system, based on a Service-Oriented Architecture, is described. Basically, this architecture is defined by providing services to other network components. Chapter 3 also introduces an editor for creating batch recipes, which is targeted to be used by process engineers. Since this work especially focuses on controlling tank systems, this concept also introduces an approach for finding paths dynamically in redundant pipe systems, on the basis of the actual tank system. For this, another domain specific editor for modeling tank systems is introduced. The implementation of the developed system is given in Chapter 4. First, this chapter gives an overview of the programming concept and the applied tools. Next, the implementation of domain specific editors for creating batch recipes and modeling tank systems as well as the path finding algorithm is described. At the end of this chapter, the main program with its recipe processing algorithm and the implementation of the services is presented. In Chapter 5 this implementation is demonstrated and evaluated on two linked laboratory tank systems, as depicted in Figure 1.2. Finally, Chapter 6 gives a conclusion of the developed concept and proposes future research topics.

CHAPTER 2

State of the Art

This chapter gives an analysis of currently available literature for concepts and implementations, related to the topic of creating flexible batch process control units. First, an overview of batch processes and batch recipes is presented. Afterwards, the concept of Service-Oriented Architectures (SOAs) is described, that can be utilized to achieve a flexible implementation of batch processes. As some major characteristics of SOA depend on the underlying messaging system, the following section will give a general overview of different communication principles. Subsequently, a method is introduced, that enables a domain specific recipe description and creation, just with the knowledge of a process engineer and without the need of a programmer. In order to improve the flexibility of industrial plants even more, in the last section, the shortest path problem is discussed, which addresses to dynamic route finding algorithms in systems with multiple paths.

2.1 Batch Process Control

Industrial manufacturing processes are classified into three different types: continuous, discrete, and batch productions. In the manufacturing industry, almost fifty percent are batch processes [13]. However, most factories use combinations of them, e. g., a batch production with a discrete packaging [14]. Since batch productions are in the focus of this thesis, they will be explained in more detail.

A batch process is a cyclic manufacturing method. At every production cycle, which

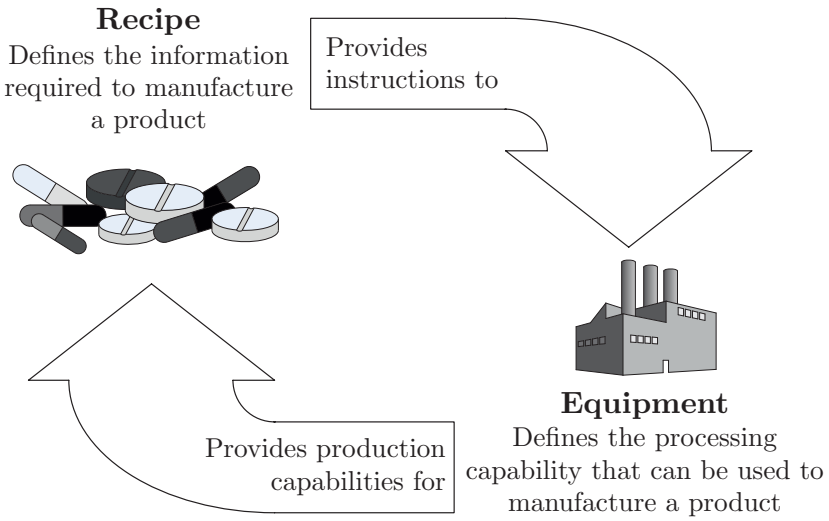


Figure 2.1: The recipe/equipment separation pattern, as presented by Brandl [14].

is called *batch*, a finite amount of output material is produced using one or more equipment components. For this, input material is transported, stored, or transformed. The transformation can be achieved chemically, physically, or biologically. The exact production procedure is specified in the batch *recipe*. It consists of an ordered set of processing actions and is executed in a finite period of time. [5]

2.1.1 IEC 61512 - Batch Control

The standard IEC 61512 – Batch control [5] defines a terminology and reference models for developing batch processes. It gives just an abstract overview, however, there are resources, that give practical implementation details [13], [14], or [15]. A main characterization of this standard is a strict separation of product definition information from production equipment capabilities. This allows an independent development and deployment of physical facilities and recipe creation. Therefore, equipment can be used for different productions, just by changing the recipe. On the other hand, the same recipe can be processed on several process cells or sites. In order to produce the desired product, both, recipe information and physical facilities are linked together, as illustrated in Figure 2.1. The equipment must provide its production capabilities to the recipe and, for this, receives instructions to utilize appropriate equipment, the exact amount of input materials, and the production procedure. This design pattern of a recipe/equipment separation can be scaled up for any complex manufacturing process. Especially, complex systems benefit from this, since changes of the recipe can be performed in a matter of minutes or hours. In contrast to this, conventional systems requires a reprogramming, which requests a programmer. This can take weeks to months and has to be documented. [14]

Another integral part of IEC 61512 is the definition of three hierarchical models, the Physical Model, Procedural Model, and the Process Model. These models provide an

abstract description of the physical facilities of a plant, the production procedure, and the general manufacturing process. The standard proposes four hierarchical levels for each of these three model, however, it also points out, that additional levels can be added for complex processes [16]. An overview of the models and their relationships is given in Figure 2.2.

Physical Model: The *Physical Model* describes the structure of the equipment. The top of the hierarchy is formed by a *Process Cell*. It contains the entire equipment, which is required to execute one or more batches. On the lower levels, the equipment is subdivided into major and minor processing steps, which are called *Units* and *Equipment Modules (EMs)*, respectively. The bottom level of the hierarchy is formed by *Control Modules (CMs)*. They are responsible for basic control of all kinds of actuators and sensors.

Procedural Model: The *Procedural Model* describes the recipe, consisting of the individual recipe steps. The *Recipe Procedure* forms the top of the hierarchy. It provides the instructions of the batch to the *Process Cell*. The *Recipe Procedure* consists of an ordered set of *Unit Procedures*, which, in turn, consist of an ordered set of *Operations*. Equivalent to the structure of the physical model, they provide instructions for major and minor processing stages. The bottom level of the procedural hierarchy is formed by *Phases*. They instruct the equipment to perform a distinct task, like adding a specific amount of a liquid to a specific tank. In IEC 61512, EMs are the lowest level of the *Physical Model*, which can be addressed by a Phase.

Process Model: The *Process Model* gives a conceptual description of the production process. The structure is similar to the *Procedural Model*, however, the *Process Model* is not linked to any equipment and therefore, it does not contain any plant specific information and conditions [17].

The *Physical Model*, described in IEC 61512, can also be generalized to higher hierarchical levels, representing the area, the site, and the whole enterprise on top of the hierarchy. IEC 61512 indicates that structure, however, it is not explained in detail. Since these levels are not specific to batch productions, it is sourced out into IEC 62264 – Enterprise-control system integration [18]. It describes the hierarchy from the viewpoint of operations control technology, by integrating Manufacturing Execution System (MES) and Enterprise Resource Planning (ERP) functionality.

Although the IEC 61512 reference model is defined for whole *Process Cells*, it is not necessary to implement all levels of the hierarchy. Since many processes are small and simple, Case [19] gives an overview of how this principle can be applied to such processes. Depending on the process, it may not be economical to implement all levels, for example, due to fast changing recipes, or an excessive programming effort. However, every implementation level reduces the required interactions of the operator, and increases reusability. The minimum implementation refers to the bottom level of the *Physical Model*. CMs form the fundament for basic control. They enable a certain

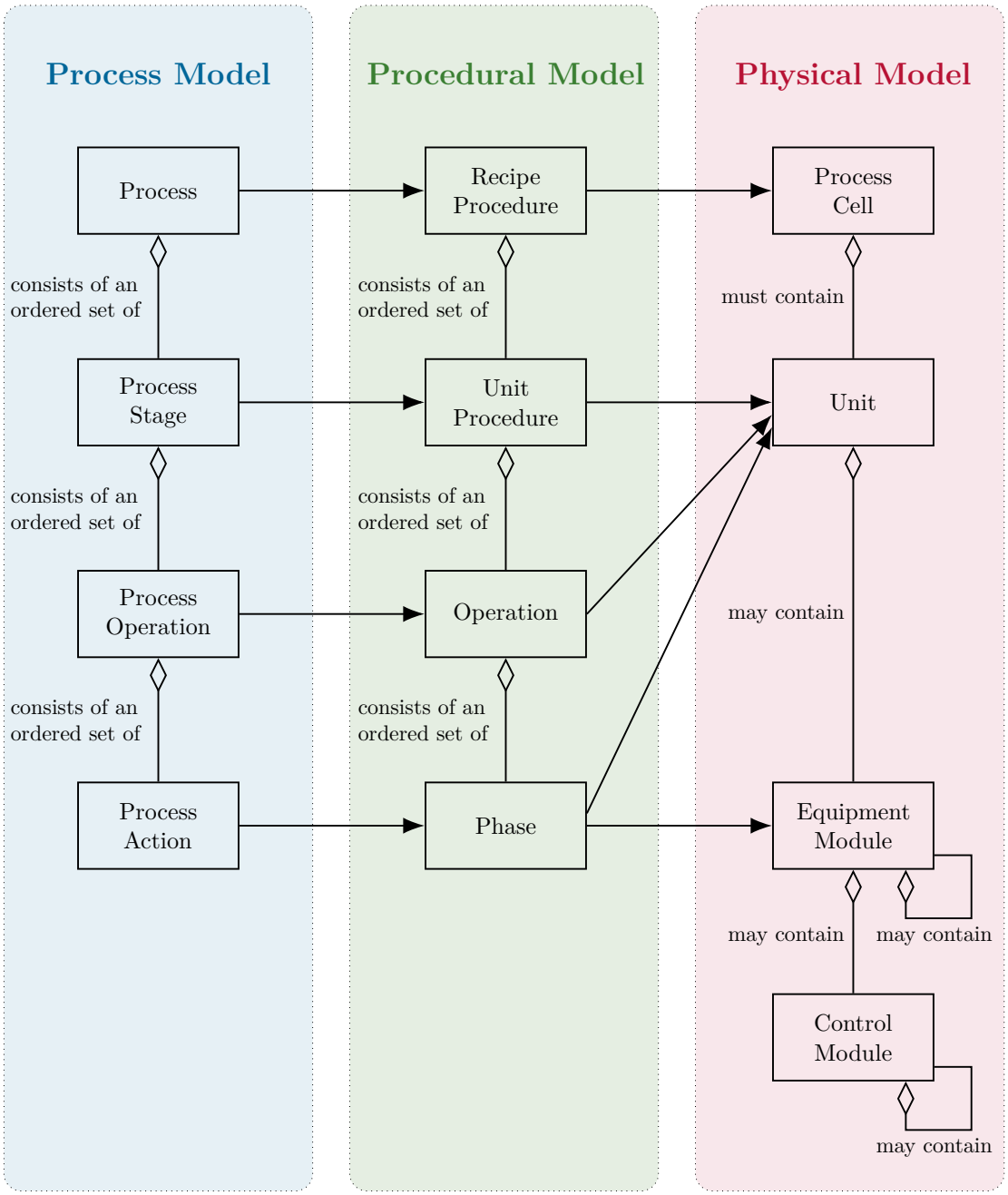


Figure 2.2: The IEC 61512 reference model, consisting of the Process Model, Procedural Model, and Physical Model [17]. Each of these three hierarchical models is composed of four levels. The elements of these levels are related to other elements, which is indicated by black arrows. The relations between Process and Procedural Model describe the mapping from the process elements to procedural elements, which contains additional equipment specific information. The relations between the Procedural and Physical Model describe, where the procedural elements are executed.

degree of process safety, for example, by providing failure notifications of exceeded threshold values. The next step is to implement EMs. They allow a coordination of associated groups of CMs. The third developmental step is to create *Phases* in terms of the *Procedural Model*, which address the EMs and initiate the corresponding actions. Finally, these single *Phases* can be coordinated by formula editors and batch sequencers. This enables the creation of complete batch recipes and automated processing.

2.1.2 Batch Recipes

Usually, batch recipes are illustrated graphically. For this, the standard IEC 61512-2 – Batch control – Part 2: Data structures and guidelines for languages [16] defines guidelines for a uniform visualization, by using *Procedure Function Charts (PFCs)*, which are an adaption of the Programmable Logic Controller (PLC) language Sequential Function Chart (SFC). According to the structure of the Procedural Model with Procedures, Unit Procedures, Operations, and Phases (see Figure 2.2), PFCs are created for each level, however, as mentioned above, not all of them have to be implemented.

An example of a PFC with three levels is illustrated in Figure 2.3. Independent of the hierarchical level, every procedure has to consist of at least one starting symbol and one end symbol, which are indicated by a triangle. The procedural elements of the recipes are illustrated by rectangles. As the example shows, there are identification marks at the corners of the rectangles. These marks describe the belonging to the hierarchical level, as illustrated in Figure 2.4. The *plus* marking (+) at the top right corner indicates an encapsulation of other procedural elements of the next lower level. IEC 61512-2 also defines a *minus* sign (-) at the top right corner, which is used, if the next lower procedure recipe is directly painted inside the rectangle.

The general execution direction of PFCs is from top to bottom and left to right. The left to right direction is necessary for determining the evaluation order of conditional branches. This can be useful, e. g., for cooling or heating procedures, if a certain target temperature has to be reached. In addition to the conditional execution of branches, IEC 61512-2 also allows concurrent execution of parallel procedures. In the PFC, this is indicated with two horizontal lines that connect the procedural elements (see Figure 2.3).

The execution of procedure elements starts, if all preconditions are satisfied. This is immediately the case after a start symbol, or if the previous procedure element is completed. However, IEC 61512-2 also allows defining *explicit transition* conditions, which are indicated by two short bars below a procedure element. These conditions imply, that the equipment is asked, if it is safe to complete the previous procedure element. Just if these conditions are satisfied, the next procedure element can be started.

Godena et al. [20] propose an alternative approach to complete procedure elements. Instead of using explicit transitions, they introduce the term *dominant phase* and

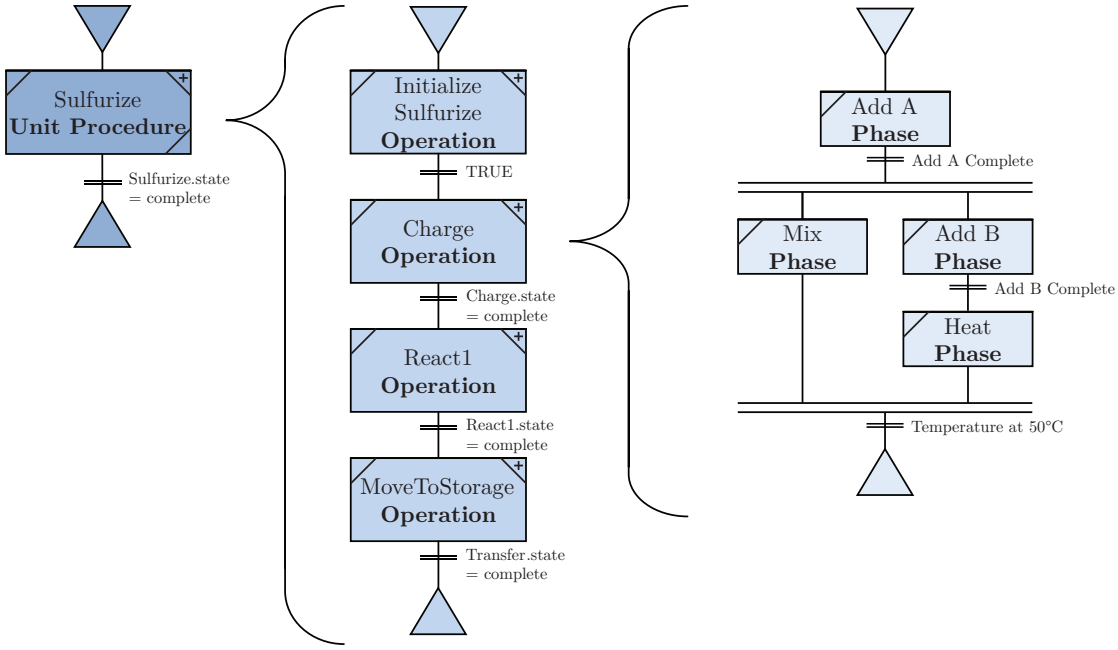


Figure 2.3: Example of a Procedure Function Chart (PFC) for a sulfurize process. The figure is adapted from Brandl [14] and identification marks at the corners, in terms of IEC 61512-2, are added.

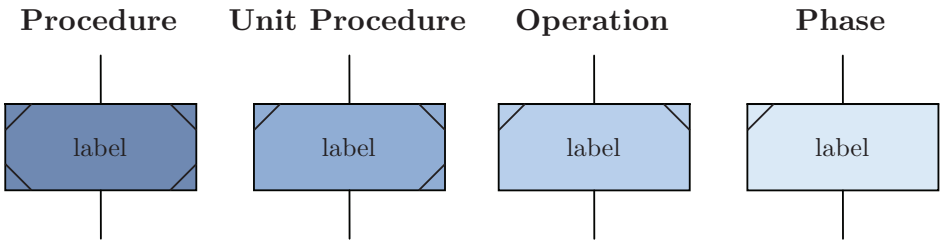


Figure 2.4: In IEC 61512-2, the PFC symbol of a procedural element depends on the level of the hierarchy.

distinguish between dominant and nondominant phases. The difference between them is, that only dominant phases have an end condition. In a parallel execution of a dominant and nondominant phase, both are complete, if the end condition of the dominant phase is fulfilled. This can be useful, for example, if a tank has to be heated up to a specific temperature and concurrently be mixed for an even dispersion of the heat. In this example, both phases are complete, if the desired temperature is reached.

2.2 Service Oriented Architecture

Service-Oriented Architecture (SOA) is a design pattern, which is used to describe distributed systems and tasks. It is a term of the Information Technology (IT) and presently used twofold. On the one hand, it describes *web services* and on the other hand, it used to describe business processes. Due to these different applications, a

major issue of SOA is, that there are too many similar definitions of what SOA really is [21]. However, all of them share the same meaning to the central component of the pattern, which is called *service*. The following listing gives a brief overview of the key features of a service [22].

- A service is *self-contained* and can be modularly composed of other services.
- A service is a *distributed component* and available over a network via a *published interface*.
- Services are *interoperable*, therefore service providers and clients can use different implementation platforms and languages.
- Services are *discoverable*. They are registered in a directory service, which is accessible for the user.
- Services are *dynamically-bounded* to the user application. They are located and bounded at runtime, thus they can be developed temporally independent from the user application.

In theory a service is described by all these principles, but usually the practice shows that not all of them are implemented, especially the principles of a discoverable and dynamically-bounded service are often omitted. [22]

A service is defined by its inputs, outputs, and a service contract. The service contract specifies a set of preconditions, which have to be fulfilled in order to execute the corresponding action. If the preconditions are satisfied, the contract guarantees post-conditions. This principle is called *Hoare triple* [23], [24]. The information, defined in the service contract is sufficient for a user to consume this service, so it is not necessary to know any implementation details of the service itself.

Basically, the communication concept of SOA consists of three different types of participants. A *service provider* provides functionalities, which are executed, when they are requested by the *service consumer*. In order to inform the consumer about the service contract and conditions, the service provider registers the service at a *service broker*. This is a central directory, where all available services are listed. After registration, the service consumer can locate the services and call it directly from the provider. These relations are illustrated in Figure 2.5. However, as mentioned above, the functionality of the discoverability are often omitted, which means, that the service consumer has a priori knowledge of the service contract. In this case, a service broker is not required.

These service properties enable the development of flexible systems, based on a composition of loosely coupled components. Since this is also an important requirement in the automation industry, SOA is handled as a promising candidate for future industrial automation [25]. Several EU projects, like SIRENA [26], SOCRADES [27], or IMC-AESOP [28] proved the feasibility and pointed out the potential of enhancing

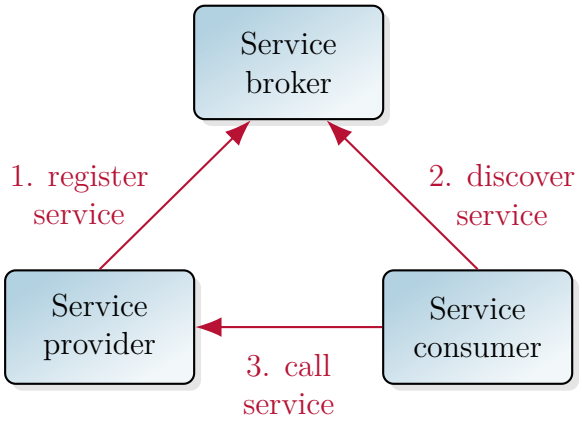
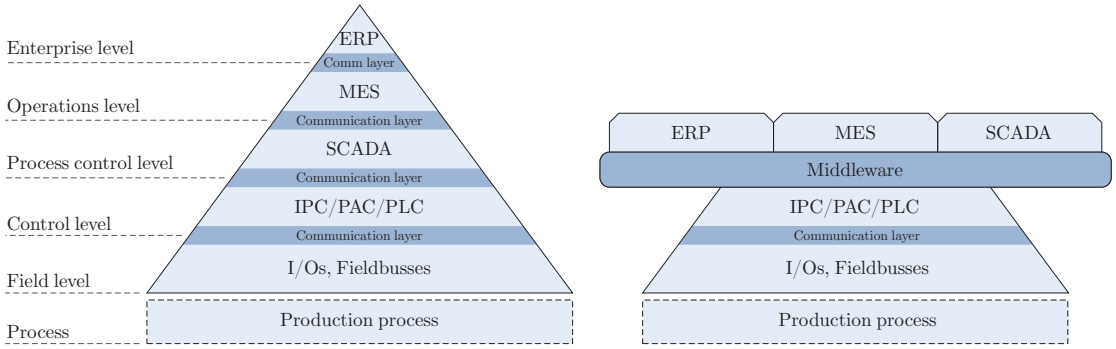


Figure 2.5: The process of a SOA communication is built up of three steps. First, the service provider registers his services at the broker, which can, in the second step, be discovered by the service consumer. Finally, the consumer can call the service from the provider. Figure adapted from Josuttis [21, p. 218].

processes, by implementing SOA in industrial plants. In this context, SOA is used as the communication technology between different layers in the hierarchy of an industrial plant. Figure 2.6a shows the traditional layered automation pyramid. A main characterization of this hierarchy is, that each layer only communicates with its neighboring layers. Virta et al. [29] showed an approach of SOA between the MES and Supervisory Control and Data Acquisition (SCADA) layer. For this, they use OPC Unified Architecture (OPC UA) as communication protocol. Since the communication is requested to be bidirectional, both, the MES and SCADA layers run an OPC UA server, which provides their services. A SOA-based middleware hosts OPC UA clients using the services provided by the OPC UA servers.

In contrast to this hierarchy with a vertical integration, SOA can also be used in a flat hierarchy approach, as shown in Figure 2.6b. Therefore, a monitoring functionality is provided to business layers (MES and ERP), which gives information about current production capabilities, or stock levels. In dependence of current orders, this information can, in turn, be used to send new production instructions to the control level [30].

The concept of a SOA-based middleware (see Figure 2.6b) can also be extended to all layers, as presented by Karnouskos et al. [31], which was also the goal of IMC-AESOP. In this approach, all layers provide their services in a so-called *cloud of services*. However, this leads to several challenges due to very large heterogeneous networks. All layers have individual requirements in network specifications, like transfer rates, timings, reliability, or Quality of Service (QoS), which have to be supported by the SOA-based middleware. Even though this is a tremendous overhead, compared to legacy systems, it enables to manage complex factories of the future and be fit for the era of Internet of Things (IoT), infrastructure virtualization and real-time high performance solutions [31].



(a) traditional layered automation pyramid (b) SOA-based approach

Figure 2.6: The automation pyramid represents the hierarchy of an industrial plant as standardized in IEC 62264 and IEC 61512. The communication path of the traditional layered pyramid, on the left hand side, is, from a layer to a neighboring layer. The graphic on the right hand side illustrates a SOA-based approach [24], where business layers on top of the hierarchy (ERP, MES) can directly access data of the control level. The graphics are adapted from Melik-Merkumians et al. [24].

Cândido et al. [32] used SOA at the device level. They compared OPC UA to Devices Profile for Web Services (DPWS), which are two commonly used communication protocols in the automation industry and analyzed their strengths and weaknesses. They concluded, that none of them can entirely cope with the requirements of SOA at the device level alone. However, a combined approach allows utilizing the benefits of both of them.

2.3 Messaging Systems

The complexity of today's automation system usually demands for distributed systems [33]. Thus, the requirements for reliable communication between a high amount of devices is a major challenge for messaging systems. For this, most messaging protocols specify a QoS, which describes the capabilities of message delivery, e. g., concerning reliability, or in-order delivery. In the automation industry, usually, it is necessary to send messages exactly once, since if, for example, a toggling command is sent twice to a switch, the switch position would immediately be turned back to the original position. However, this can also be handled by *idempotent* messages. This means, that the function does not depend on the current state and an instruction, sent multiple, would not change the result. This can be expressed mathematically as $f(f(x)) = f(x)$. Therefore, the message has to be unambiguous, e. g., turning on or turning off, instead of toggling. By using this, the requirements to QoS can be reduced to an *at least once* delivery, which reduces the amount of messages. Besides to a reliable and secure communication, a modern messaging system must also manage a high throughput of messages and

be independent from hardware platform, operating system and programming language [33].

A fundamental feature of a messaging system is the interaction model, which can either be synchronous or asynchronous. The following section describes messaging patterns, which are based on these two interaction models.

2.3.1 Message Exchange Patterns

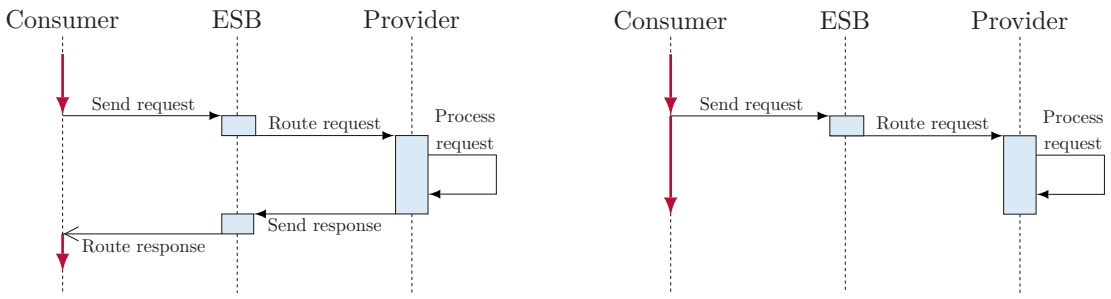
Josuttis [21] uses the idea of synchronous and asynchronous communication to distinguish two basic Message Exchange Patterns (MEPs). These patterns are general concepts to describe the communication between two or more participants. Since they will be explained in the context of SOA, the participants, which are usually called *sender* and *receiver*, will be called *(service) consumer* and *(service) provider*. However, these patterns are universally usable and not limited to SOA. The following sections will give a short overview of two basic and, on the basis of them, two advanced MEPs.

Request/Response

The *Request/Response* (also called *Request/Reply*) pattern is, as Josuttis says [21, p. 124], probably the most important pattern for SOA. For calling a specific service, the consumer sends a request, which is routed to the provider. Subsequently, the provider processes the request, executes the corresponding service, and sends a response. This situation is illustrated in Figure 2.7a. From a programmer's view, this principle is easy to implement, since the service call can be handled as a Remote Procedure Call (RPC). A big benefit of this RPC structure is, that a specific sequence of events can easily be handled. However, as this process is based on a synchronous communication model, program execution of the service consumer is blocked for all other actions, until the response arrives. The response time depends on various parameters, like transmission speed, the load of the provider, and the execution time of the requested service. Therefore, a synchronous communication may decrease the overall performance of the complete system and can, at worst, end up in a mutual deadlock of applications [34], if, for example, the provider is not available or the request gets lost. Consequently, such cases have to be considered and handled by the programmer.

One-Way

The second basic MEP is the *One-Way* MEP. As the name says, the consumer sends a request to the provider, which processes the request and executes the corresponding action. However, the provider does not send any confirm or response message. Therefore, this pattern is also called *fire and forget*. The message flow, as illustrated in Figure 2.7b, equals an asynchronous message. Since no response is expected, a blocking state cannot occur and the problems of performance losses or deadlocks are avoided. However, this



(a) Request-Response MEP (b) One-Way MEP

Figure 2.7: Comparison of the two basic Message Exchange Patterns (MEPs), as presented by Hohpe and Woolf [35]. The vertical red arrow line indicates the program flow of the consumer program. Figure (a) shows the Request-Response pattern with its major flaw of a program blocking, while sending a request, indicated by an interruption of the red line. In the case of the non-blocking One-Way pattern in (b), the program flow continues immediately after sending the request, so the red line is not interrupted, however, there is no response to the consumer. The intermediate station of the Enterprise Service Bus (ESB) represents the SOA infrastructure, which is responsible for data transfer.

causes a more complex message handling, since the messages have to be administrated for a correct assignment and in the correct order to the receiver. A Message Queue (MQ) is a program, which implements the One-Way pattern and manages the exchange of messages with other applications. The queue buffers the messages until the receiver retrieves them.

Request-Callback

The *Request-Callback* MEP is an advanced messaging pattern, composed of two asynchronous One-Way messages. This pattern is usually used, if a confirmation or response message is required, but a blocking state must be avoided. Therefore, it is also called nonblocking Request-Response, or asynchronous Request-Response. Triggered by a One-Way request of the consumer, the provider answers with another One-Way message, which is delivered to the callback function of the calling process. In dependence of the message, the callback function executes the corresponding actions. In order to achieve this, all messages must be identifiable for a correct delivery. It is obvious, that this MEP is much more complex compared to the basic Request-Response pattern, however, this kind of communication leads to a loose coupling of the services, which is a major requirement of SOA.

Publish-Subscribe

A commonality of all previously described MEPs is the one-to-one nature with one sender and one receiver. However, there are scenarios, where one-to-one communica-

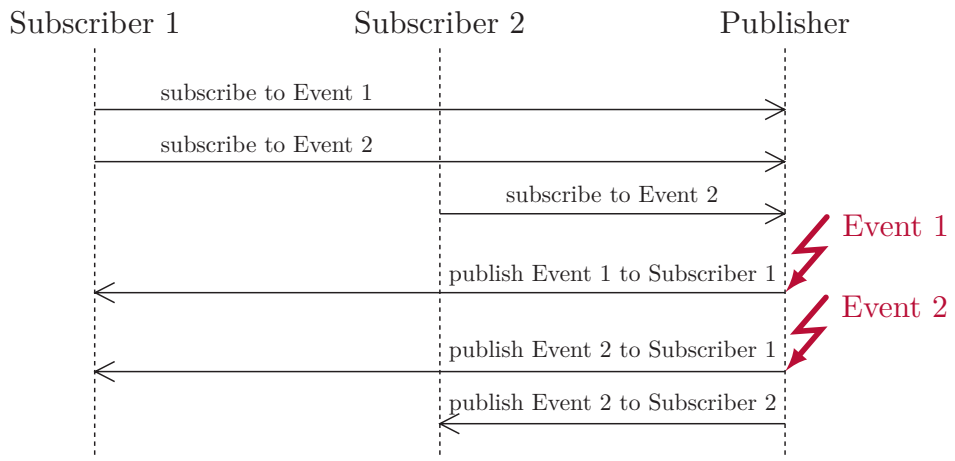


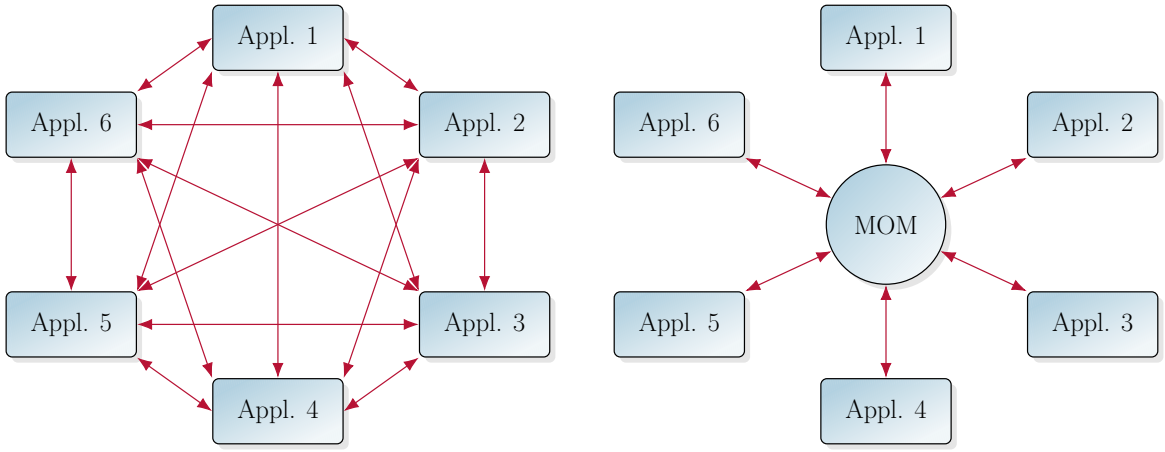
Figure 2.8: Example of a Publish-Subscribe communication with two subscribers and one publisher. Subscriber 1 registers for Event 1 and Event 2, whereas Subscriber 2 just registers for Event 2. Once an event occurs, all subscribers are notified. On the basis of Unified Modeling Language (UML) syntax, the open arrowhead of the messages indicate asynchronous messages. For a better clarity, the Enterprise Service Bus (ESB) is omitted in this graphic.

tions are not well-suited. Such a case could be, if several receivers expect the same messages, usually for notification purposes, or several components send data to the same receiver. Therefore, one-to-many, many-to-one, or many-to-many communications fit much better. The Publish-Subscribe MEP, which is, in the domain of Object-Oriented Programming (OOP), also known as *Observer* pattern [36], addresses primarily to such communications with multiple senders and/or receivers. The pattern is designed to act very dynamically. The publisher does not need any a priori knowledge of the number or addresses of the subscribers. However, as a consequence, an additional step is required for a communication process. In order to receive messages, the subscriber has to be registered at the publisher. After occurrence of the corresponding event, the publisher sends messages to all subscribers, which are registered for this event. Figure 2.8 illustrates an example of this process. Compared to other MEPs, the Publish-Subscribe pattern has a better scalability for larger networks with many devices.

In order to give a summary of all mentioned MEPs, there is no universally usable MEP, since each of these approaches has it benefits and drawbacks. It depends on the application of which should to be chosen.

2.3.2 Message-Oriented Middleware

The significant increase of message transfer in modern systems with distributed devices, necessitates another form of a messaging system to handle the high amount of messages reliably [33]. Therefore, a common approach is to source out all messaging tasks, which are responsible for delivery, to an additional component in the network. This messaging component is called Message-Oriented Middleware (MOM). The entire



(a) Fully connected mesh topology without MOM

(b) Star topology with MOM in the center

Figure 2.9: Comparison between a communication system without (a) and with (b) Message-Oriented Middleware (MOM), adapted from Curry [33]. In a communication system with MOM, each application just communicates with the MOM and there are no point-to-point connections between applications to each other. Furthermore, the MOM is responsible for reliable message delivery.

network transfer is routed from the source application to the MOM, which forwards it to the target application. As a consequence, there are no point-to-point connections between applications, since every application just communicates with the MOM. Figure 2.9 gives a schematic overview of the communication between traditional RPC and MOM deployments. For distributed systems, the MOM approach has several benefits. First, the nature of loosely coupled components simply allows to "tap into" an existing environment, without disturbing it [37]. This leads to a high degree of flexibility and scalability. Secondly, the source needs not care about a successful delivery, since this is a task of the MOM. This can be a crucial factor, if the target application is currently not available or something went wrong at the delivery.

2.3.3 Commonly used Messaging Methods in the Automation Industry

Based on the different MEPs and the opportunity of integrating MOM, there are several messaging technologies and protocols. This section gives an overview of some commonly used protocols in the automation industry.

OPC Unified Architecture

OPC UA is a standard (IEC 62541 – OPC Unified Architecture [38]) for Machine-to-Machine (M2M) communications in industrial applications. However, it is much

more than just a communication protocol, since it provides an entire infrastructure to store data and share it with other participants. The extensive integration of specifications and features, like accessing (historic) data or providing alarm notifications, offers a manifold usage. OPC UA is designed for SOA communications, especially at the process control level (see Figure 2.6), for exchanging data between SCADA systems, Human Machine Interfaces (HMIs), and other gateways [32]. Since OPC UA is an industry driven standard, it copes with the requirements for SOA-based automation systems, as analyzed by Melik-Merkumians et al. [24]. The message exchange bases on an eXtensible Markup Language (XML) structure, which enables a platform independence between participants. Another benefit of OPC UA, is the potential of using a secure communication channel.

OPC UA is based on a Client-Server architecture, that stores all data in a common address space, which is usually maintained by a superior control unit, e. g., an Industrial PC (IPC), or Distributed Control System (DCS). The address space, for example of a sensor, need not just contain the sensor value, but can also contain additional information, like sensor type, engineering unit, or the tolerance of the sensor. In systems with several OPC UA servers, it is possible to implement a discovery server, where all OPC UA servers can be registered and found by OPC UA clients. In order to get data out of the address space, or write into it, there are two options [39]. First, the clients can navigate through the address space of the servers and request the desired data, or second, if the unique NodeID, which is the identification of the desired data, is previously known, the client can directly access these data.

The message transfer of OPC UA is based on the Request-Response MEP with an asynchronous message exchange [40, p. 126]. However, most OPC UA Stack Application Programming Interfaces (APIs) wrap them into synchronous calls. The communication via OPC UA requires several request-response queries for opening a secure channel, finding servers, getting end points, creating a session, activating it, reading or writing the desired data, closing the session, and closing the secure channel [39]. Compared to other protocols, that will be explained later in this section, this is much protocol overhead, which results in a corresponding transmission time. However, if a high amount of data is to be transmitted and the connection is established, OPC UA is very efficient [41].

OPC UA also provides the feature of subscribing to an event. If this event occurs, the subscribed client gets notified. Anyhow, this requires a resource intensive permanent established connection between Client and Server. Due to that reason, in March 2016, the OPC Foundation, which is the maintainer of OPC UA, announced an additional OPC UA specification for Publish-Subscribe functionality in replacement of the Client-Server architecture, which sends data connectionless from the publisher to subscribers. This Publish-Subscribe architecture suits very well for frequent transmissions of small amounts of data, which is often the case between well-known participants like PLCs. [42]

Devices Profile for Web Services

DPWS is another SOA-based M2M communication protocol, which was in the focus of the previously mentioned EU projects SIRENA and SOCRADES. Originally, DPWS comes from the high level IT, focusing on communication with resource limited embedded devices of the lowest device level [32]. Therefore, DPWS uses network protocols, which are well-known in the IT, like Hypertext Transfer Protocol (HTTP) with Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) as underlying data transfer protocol. DPWS is build on top of the synchronous message exchange protocol SOAP, and relies on additional Web Service (WS-*) specifications [27], [43]:

- *WS-Addressing* defines a transport-protocol independent message transfer. For this, different protocols, as the aforementioned ones, can be used. Supplementary to the synchronous message exchange of SOAP, *WS-Addressing* provides a way for asynchronous One-Way messages.

- *WS-Eventing* specifies an event handling, based on a Publish-Subscribe architecture, that allows subscriptions to events.

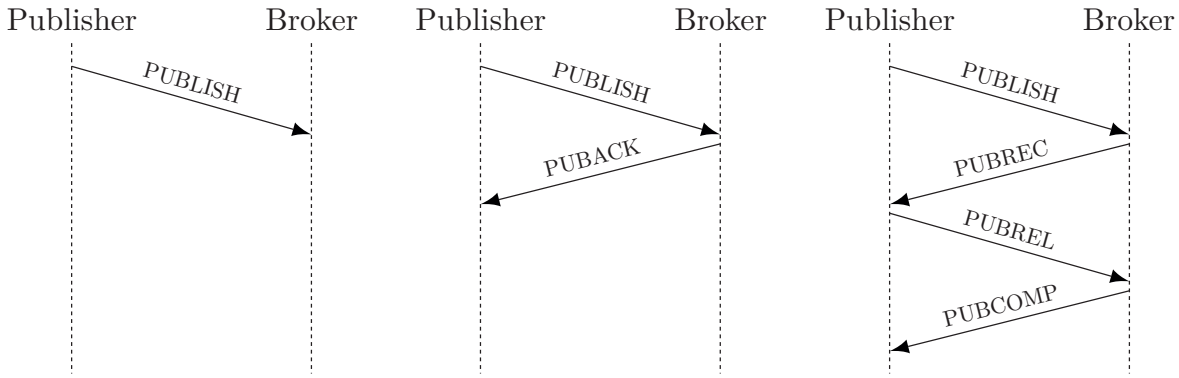
- *WS-Discovery* defines a discovery protocol, based on a network multicast, for searching and locating other devices in the network.

- The *WS-Security* specification describes an optional set of functions for ensuring end-to-end message integrity, message confidentiality, and authentication of the participants. For a secure communication, Transport Layer Security (TLS) can be integrated.

Similar to OPC UA, an interoperability and platform independence can be achieved by an XML-based message exchange. However, DPWS is less efficient as OPC UA, since there is no optimization of the XML data, which results in an even higher data traffic. Furthermore, the data structure of DPWS is not related to real-world objects with all its variables, attributes, and relations, as it is in OPC UA with the address space [32]. However, DPWS benefits of the usage of generic open web standards [32], whereas a drawback of OPC UA is, that there is no open reference implementation available, that can be used in research for free [44].

Message Queue Telemetry Transport

Compared to service-oriented messaging protocols, like OPC UA or DPWS, there is another group of messaging protocols, that is not designed for executing a service, but focuses on the message transfer. A prominent representative of these message-oriented communication protocols, is Message Queue Telemetry Transport (MQTT). It is a simple and lightweight messaging protocol, designed for use in constrained networks with low bandwidth, high latency, data limits and fragile connections [45]. Due to the low requirements and just a little protocol overhead with a fixed header of only 2 bytes,



(a) QoS 0 (b) QoS 1 (c) QoS 2

Figure 2.10: Comparison of different Quality of Service (QoS) levels of the Message Queue Telemetry Transport (MQTT) protocol. These figures illustrate the message transfer from the Publisher to the Broker. In Figure (a) with QoS 0, the message is published without any confirmation. In Figure (b) with QoS 1, an acknowledgment is sent back to the Publisher, in order to confirm the reception of the message. In a failure case, where the message is not confirmed, the Publisher resends the message. In addition to the message transfer of QoS 1, in Figure (c) with QoS 2, the reception of the acknowledgment is confirmed to the Broker with an additional publish release (PUBREL) and publish complete (PUBCOMP) message. This is the only way to guarantee a delivery, which is sent *exactly-once*. The same principle is applied for the Broker to Subscriber communication.

MQTT is well-suited for distributed systems, consisting of many devices with limited resources, like in Wireless Sensor Networks (WSNs) or IoT applications.

MQTT is based on a Publish-Subscribe architecture with a message broker, which is responsible for message delivery. In order to achieve a reliable communication, MQTT specifies three QoS levels. The QoS level describes whether a message must be delivered *at most once* (QoS 0), *at least once* (QoS 1), or *exactly once* (QoS 2). The message transfer of these QoS levels is illustrated in Figure 2.10. However, the higher the QoS level, the higher the necessary amount of transmitted messages, since the arrival of the notification at the sender has to be confirmed to the receiver. Due to this additional confirmation messages and the need of an additional message broker, the latency of message delivery for MQTT is higher, compared to other protocols without an intermediate station.

For MQTT, there are several open-source implementations available, on both sides, for brokers and clients, as well as for different platforms and programming languages. In order to achieve secure communications, TLS, or its predecessor Secure Sockets Layer (SSL) can be used. However, this requires more computing power for encryption/decryption and increases the transmission overhead [45]

Zero MQ

ZeroMQ is a lightweight message-oriented messaging library for various sorts of communications, like in-process, or intra-process communication, as well as between different devices in distributed networks. Compared to other message-oriented protocols, the major difference is, that ZeroMQ is designed to work without a message broker. This is also indicated by the term *Zero* in ZeroMQ, which means, that no message broker is required and therefore latencies can be reduced to (almost) zero [46]. Hence, messages are directly sent to the receiver. For this, several MEPs, like a synchronous Request-Response or asynchronous Publish-Subscribe pattern can be used. Concerning reliable message delivery, ZeroMQ has no features for a guaranteed delivery implemented, however, the official ZeroMQ Guide [47] proposes several patterns to implement reliability. There are also source code examples available for implementing these patterns [46]. Despite of the major characteristic of ZeroMQ, in fact that it works brokerless, most of these pattern require a message broker, which can still be implemented with ZeroMQ. Anyway, it is obvious, that an additional broker influences the latency of message delivery negatively.

Data Distribution Service

Data Distribution Service (DDS) is a M2M protocol designed for real-time communications between embedded devices in distributed systems. A major feature of DDS is the concept of data transfer, which is different, compared to other protocols with a message-oriented or service-oriented approach, since data is not exchanged by sending messages or requesting services. It bases on a data-centric Publish-Subscribe approach, which uses a Global Data Space (GDS) that is accessible for all interested applications [48]. Publishers can post new data to the GDS and send notifications to all subscribing applications, that are registered for this specific topic. Subsequently, these applications can directly access these data from the GDS, which is different from message-oriented protocols, where the data is directly sent. Figure 2.11 illustrates this communication process.

Due to this data-centric approach, that utilizes a Publish-Subscribe architecture, DDS is a very flexible and scalable communication protocol, since the GDS is not restricted to one physical device, but can also utilize several local data spaces of distributed devices and combine them to one logical GDS. DDS is designed for low latency applications with a high throughput, therefore, there are not intermediate servers, as in OPC UA, and DDS also supports multicasts, so that all subscribers are addressed at once. For providing and accessing data, DDS features finely controllable QoS functionality with different modes, ranging from best-effort to reliable in-order delivery [49]. DDS can be used on different platforms independent of the programming language and operating system. For this, there are several proprietary and open-source implementations available covering those platforms.

Similar to message-oriented messaging protocols, like MQTT, DDS is not designed for

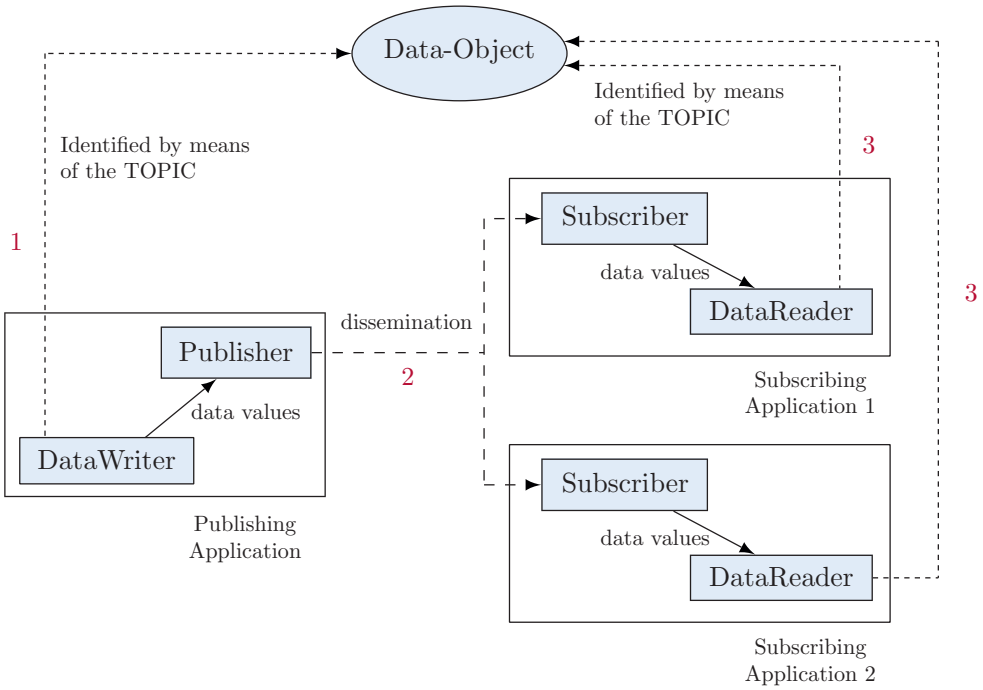


Figure 2.11: Schematic overview of the data transfer in a Data Distribution Service (DDS) configuration, as presented by the Object Management Group [48]. The communication process is as follows: first, data is written via DataWriter methods by the publishing application to a Data-Object. Secondly, the publisher notifies all subscribers, which can, thirdly, access the data via DataReader methods.

executing services in terms of SOA. If these functionalities are required, they have to be implemented additionally.

2.4 Domain Specific Language

As mentioned in Chapter 1, personalized production is a major challenge of current automation systems. A way to handle this challenge is to use fast changeable production recipes, which, in turn, requires flexible recipe creation tools. The major task of such a tool is to represent all possible actions with an easy to handle user interface. At best, the process engineer can create a program just with domain specific process knowledge and process specific keywords. Domain Specific Languages (DSLs) addresses exactly to such problems.

A DSL is a language, that is tailored to one specific application area. It models a problem just at the same level of abstraction and uses common notations of the domain. The opposite of a DSLs are General Purpose Languages (GPLs), which are universally usable across domains. Some commonly used GPLs are *Java*, *C/C++*, and *Python*. The major characterization of GPLs is a clear Turing completeness [50]. This

means simplified, that anything, which is capable to be computed on a computer, can be expressed with the language [51]. Since DSLs are an abstraction of one specific domain, they are usually not Turing complete [52]. This, on the other hand, reduces the program complexity of the DSL and keeps it compact. An example of a textual DSL, that points out the potential of compactness of a DSL, is presented in Listing 2.1. It shows a simple database query in Structured Query Language (SQL) syntax. The code is reduced to the minimum of essential components without unnecessary overhead.

```
1 SELECT *
2 FROM Accountlist
3 WHERE ID = 5;
```

Listing 2.1: SQL Example: The query responses all (*) entries from the database named *Accountlist* for the account with ID = 5

2.4.1 Classification of DSLs

DSLs are classified into two basic types, *internal* and *external* DSLs. The former one depend on a host language, whereas the latter one is independent of any other language. For developing a DSL, nowadays, all necessary tools are packed into Integrated Development Environments (IDEs).

Internal DSLs

Internal DSLs are build on the basis of an existing GPL (also called *host language*) and utilize their infrastructure [53]. Thus, the DSL is bound to the underlying GPL, which brings its benefits and drawbacks. On the one hand, building an internal DSL enables the usage of a working environment and features. This can be benefiting, if the programmer is used to the GPL, but an obstacle if not. On the other hand, internal DSLs are limited to the host language. If something has to be expressed that does not map well with the host language, it is getting complicated to achieve the desired functionality [54]. Usually, internal DSLs are syntactically and structurally oriented on the host language [54].

External DSLs

An external DSL enables the development of a language, which is exactly required with as little as possible but as much as necessary features. In contrast to internal DSLs, external DSLs are build from scratch with their own syntax and semantics [53]. Therefore, they do not require infrastructure of a GPL. However, in order to execute a program, separate infrastructure is required for lexical analysis and parsing as well as code interpretation or code generation. Figure 2.12 shows a standard process of language recognition, by using a lexer and parser. The lexical analysis converts a

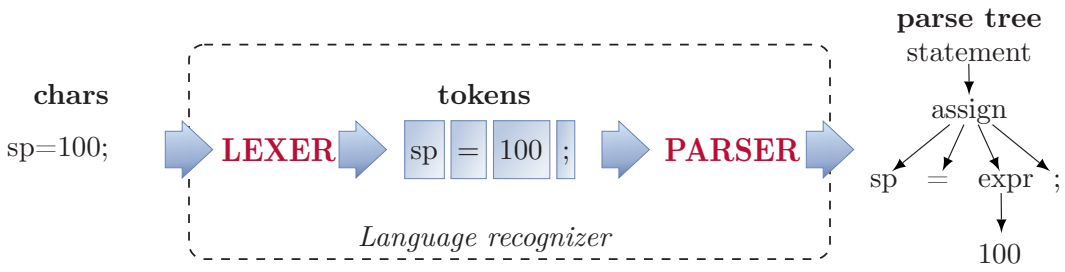


Figure 2.12: Language recognition by a *lexer* and *parser*, adapted from Parr [56]. The lexer analyses the input sentence, that consists of single characters, and combines them into syntactically associated tokens. In the next step, the parser analyses these tokens and put them into a parsing tree, which is also termed Abstract Syntax Tree (AST). The AST is the basis for semantical correctness checks and also the basis for code generation or code interpretation [55].

sequence of characters into a sequence of tokens, which are single atomic elements of the language, like keywords, identifiers, or symbol names [55]. Subsequently, these tokens are syntactically analyzed and put into context by a parser. Finally, there are two ways to execute the desired DSL program, via an *interpreter* or *code generator* [52, p. 29]. An interpreter is running directly on the target platform. It loads the DSL program and executes the desired actions. A code generator requires an additional step, since it just creates the source code for another language, like a GPL. This generated code can now be compiled or interpreted with common tools of the corresponding language.

However, for a comfortable DSL development and usage several additional features are required. For this, Fowler coined the term *Language Workbench* [54], [57]. It integrates all necessary tools into an IDE, starting from the definition of language specific keywords up to running the DSL program. For an easy development, supplementary features for, e. g., testing, debugging, syntax highlighting, code completion, or symbolic integration between all developmental steps should also be integrated.

2.4.2 Defining a Grammar of a Domain Specific Language

A grammar is the basis of every language, independent whether it is a DSL or GPL. It describes all production rules, that define how a valid textual input, which is called *sentence*, should look like [52]. A common notation for describing grammars is the Backus - Naur Form (BNF) [58], named after John Backus and Peter Naur. In order to write sentences in BNF notation, two different types of symbols are required. *Terminal symbols* form the basis and can not be fragmented into other symbols. *Non-terminal symbols*, on the other hand, can consist of terminal and non-terminal symbols. A sentence in BNF looks as follows: $S ::= P_1 \dots P_n$, where S is the non-terminal symbol, which is defined by a series of pattern expressions $P_1 \dots P_n$ [52]. Pattern expressions, in turn, consist of keywords and other terminal or non-terminal symbols. In Listing 2.2,

an example in BNF notation is presented.

```
1 <digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
2 <uint> ::= <digit>|<uint><digit>
3 <int> ::= <uint>|('-'<uint>)
```

Listing 2.2: Definition of an *integer* in BNF-notation. In the first sentence, *digits* are defined. A digit can either be *0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, or *9*. The *or* statement is indicated by a vertical bar "|". The second sentence describes the grammar rule for unsigned intergers (*uint*), consisting, either of only one *digit*, or another *uint* followed by a *digit*. In the third sentence, signed integers (*int*) are defined, which consist either of a *uint* or a *minus sign* and a *uint*.

This example shows, that the definition of a *signed integer* requires three rules. If the explicit definition of an *unsigned integer* is not necessary, the grammar rule of the *signed integer* can also be written in two lines, as illustrated in Listing 2.3

```
1 <digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
2 <int> ::= (<digit>|<int><digit>) | ('-'(<digit>|<int><digit>))
```

Listing 2.3: Another *integer* definition in BNF-notation, with only two grammar rules.

However, this example is quite complicated to read. In order to shorten grammar definitions and even enhance the readability of the grammar, there are several improvements of the BNF [52]. One of these improved notations is the Extended Backus - Naur Form (EBNF), which is standardized in ISO/IEC 14977: Information technology – Syntactic metalanguage – Extended BNF [59]. The advantage of using the EBNF for grammar definition is the additional support for several convenience operators. As illustrated in Listing 2.4, the grammar rule for a *signed integer* in EBNF-notation can be shortened to only one line.

```
1 int = ['-'] {'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'};
```

Listing 2.4: Definition of an *integer* in EBNF-notation. The square brackets describe an optional expression. In this example, the minus sign *can*, but need not be used. Furthermore the curly brackets describe a repetition. The bracketed rule can be repeated as often as desired. In EBNF-notation, every sentence is terminated with a semicolon.

A commonly used tool for actually defining DSL grammars is the Xtext framework [60] of the Eclipse Foundation. Xtext is a *language workbench* (see Section 2.4.1) for developing textual DSLs. It bases on a EBNF-like notation, however, some symbols vary. In Xtext, optional expressions are marked with parenthesis followed by a question mark sign "(`optional expression`)?", instead of square brackets "[`optional expression`]" of the standard EBNF-notation. Furthermore, in Xtext, repetitions of rules are marked with a plus sign "(`rule`)+" or an asterisk "(`rule`)*", for describing *one or more* or *zero or more* occurrences of the rule, respectively. Listing 2.5 gives an example of the signed integer definition in Xtext nomenclature.

```
1 grammar org.xtext.example.mydsl.MyDsl hidden(WS)
2 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
3 generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"
4
5 Model: integerList+=int*;
6 int: ('-')? (DIGIT)+;
7
8 terminal DIGIT: ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0');
9 terminal WS: (' '|'\t'|'\r'|'\n')+;
```

Listing 2.5: DSL example based on the Xtext framework. The *Model* contains a list of zero or more integers. The characteristic of a *list* is indicated by the "+=" symbol.

The first three lines of this example are necessary to define the workbench of the language and importing the required *ecore* metamodel. The grammar of the DSL itself starts with the definition of the *Model*, which describes a list of zero or more *integers*. In the next line, these *integers* are actually defined, with an optional minus prefix "('-')?", followed by one or more *DIGIT* symbols. The definition of the *DIGIT* represents a *terminal rule*, as described above, since it cannot be fragmented into other symbols. In order to ignore whitespace characters, another *terminal rule* is required. As defined in the first line (*hidden(WS)*), all these characters will be ignored by the parser.

2.4.3 Application Examples of Domain Specific Languages

In the following sections, typical examples of DSLs are described. Although DSLs are used in various domains, these sections just cover the domains of programming PLCs and creating batch recipes in terms of IEC 61512.

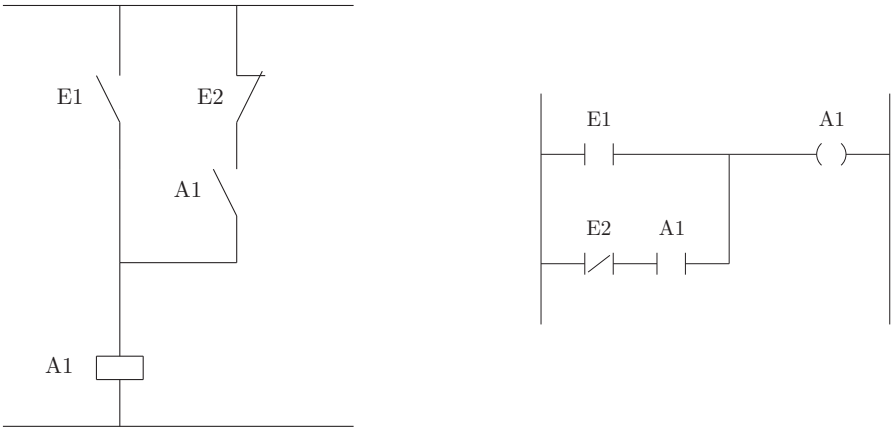
IEC 61131-3

IEC 61131-3 – Programmable controllers - Part 3: Programming languages [61] is a well-established standard, that defines five languages for programming PLCs. Two of them, *Instruction List (IL)* and *Structured Text (ST)*, are textual languages, and the other ones, *Ladder Diagram (LD)*, *Function Block Diagram (FBD)*, and *SFC*, are graphical DSLs. IL and ST cannot exactly be termed DSLs or GPLs, since although they are just used for PLC programming, they are very similar to the assembler language and the GPL PASCAL, respectively. All five IEC 61131-3 languages are used to develop PLC programs, however, every language covers a different environment where it is most suitable (see Table 2.1).

An example of the PLC language LD is illustrated in Figure 2.13. It shows a traditional wiring diagram in Figure 2.13a and the equivalent implementation of a DSL, which is defined in IEC 61131-3 and called Ladder Diagram, in Figure 2.13b. As can

Table 2.1: IEC 61131-3 languages and typical applications, adapted from Vogel-Heuser [62].

Name / Functionality	Application
Ladder Diagram / circuit diagram	On/off, lamps
Instruction List / assembler type	Time-critical modules
Function Block Diagram / Boolean operations and functions	Interlocking, controller, reusable functions, communication
Sequential Function Chart / state diagram	Sequences
Structured Text / higher programming language	Controller, technological functions



(a) Wiring Diagram

(b) Ladder Diagram (IEC 61131-3)

Figure 2.13: Comparison of a traditional wiring diagram with a DSL implementation of IEC 61131-3, called Ladder Diagram. The DSL represents almost a one-to-one mapping of the wiring diagram and therefore it is very easy to create for electrical engineers. The circuit shows a simple latched contactor/relay.

be seen, the Ladder Diagram is almost a one to one representation of the wiring diagram. Therefore, every electrical engineer with its domain specific knowledge can create Ladder Diagrams, which can be executed on PLCs.

IEC 61499

Another example for using a DSLs for programming PLCs and DCSs, is defined in the standard IEC 61499. In the industry, it is regarded as a possible successor of the well-established standard IEC 61131 – Programmable controllers [65] [66]. Compared to IEC 61131, IEC 61499 is distinguished by two major differences, first, a direct support for distributed systems, and second, an event-driven execution process [24]. The support for programming distributed systems addresses exactly to the demands

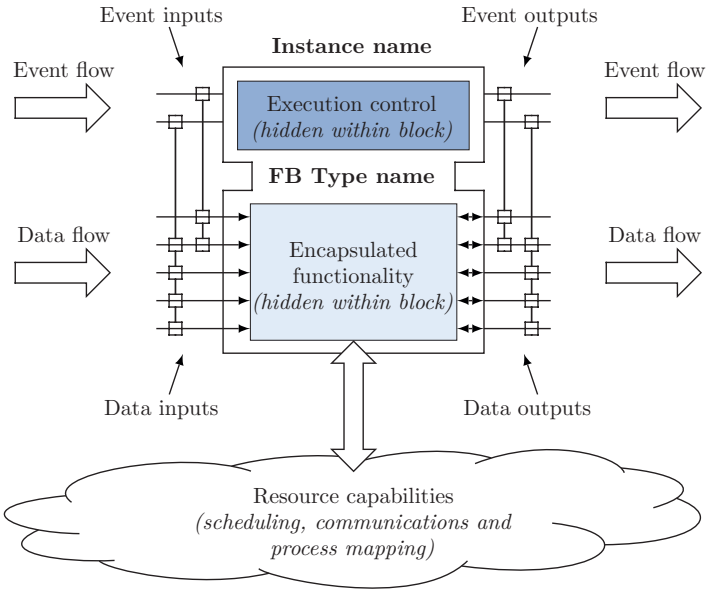


Figure 2.14: Description of an IEC 61499 – Function Blocks [63] Function Block (FB), adapted from Zoitl and Lewis [64]. A Function Block (FB) consists of an event part on the top and data part at the bottom of the block. Inputs are placed on the left side of the block, and outputs on the right side. The small rectangles indicate the relation for each event input/output to the data inputs/outputs. For incoming events: if an event is triggered, the corresponding data inputs are read. For outgoing events: if an event is sent, all corresponding data outputs are valid. The functionality of the block itself, is hidden inside the block and usually programmed in Structured Text or C/C++.

of current and future automation systems, as described in Chapter 1. IEC 61499 is a graphical DSL with FBs as its basic component, that is exemplary illustrated in Figure 2.14. The representation of FBs is similar to the IEC 61131-3 language FBD, however, as mentioned earlier, a major characterization of the standard is its event-driven process execution. Therefore, additional to a typical FBs of IEC 61131-3, an IEC 61499 FB has, on the top of the block, an event execution control with incoming and outgoing event junctions. If an event is triggered, the associated data inputs, which have to be valid at triggering, are read, the corresponding functionality is executed, data outputs are written, and finally, the output event(s) are sent [64]. In IEC 61499, there are four different types of FBs defined. *Basic FBs* are used for simple tasks, *Service Interface Function Blocks (SIFBs)* for communication with other resources, *Composite Function Blocks (CFBs)* for combining several other FBs into a clear new block, and *Adapter FBs* as a structured interface for data and event signals with no other functionality. Due to this modular concept of IEC 61499, a major benefit of the standard is the reusability.

MONACO

Prähofer et al. [67] developed a textual DSL for hierarchical, event-based machine automation programming. They termed it MOdeling Notation for Automation COntrol (MONACO). The syntax is similar to the GPL PASCAL (e. g., it supports subroutines, loops, local variables), however, it's targeted to domain experts. Due to the event-driven structure, procedures can react to specific events, for example, if a certain target temperature is reached, the heater shall be turned off. In addition to the textual editor, they also implemented a visual programming environment, that bases on the same source.

Procedural Function Chart

The PFC is a graphical DSL for creating batch recipes and is standardized in IEC 61512-2. Please see Section 2.1.2 for a detailed description.

Petri nets

Petri nets are another graphical method for describing procedures of batch processes [68]. Initially, they were created as a general purpose mathematical and graphical tool to describe relationships between conditions and events in computer systems. However, due to their event-driven structure, they also comply with the requirements of industrial manufacturing systems, since they are suitable for modeling system states and their transitions into other states, based on external events [69]. With Petri nets, sequential and parallel processes can be modeled. By adding the concept of *time* to Petri nets, it is also possible to derive and evaluate quantitative performance indices, like the cycle time of batches, production rates, or resource utilization [68].

Basically, Petri nets consist of four fundamental symbols. In batch processes, they are depicted and used as follows:

- **place nodes** are depicted as circles and formulate the status of resources or conditions,
- **transitions** are depicted as boxes or vertical bars and are used to model events,
- **directed arcs** connect place nodes with transitions and define specific actions, and
- a discrete number of **tokens**, depicted as small solid dots, are used as indicator for active places. [68]

Figure 2.15 illustrates an example of a batch process, that is modeled as a Petri net. A basic design rule of Petri nets is, that there is always a change of places and transitions.

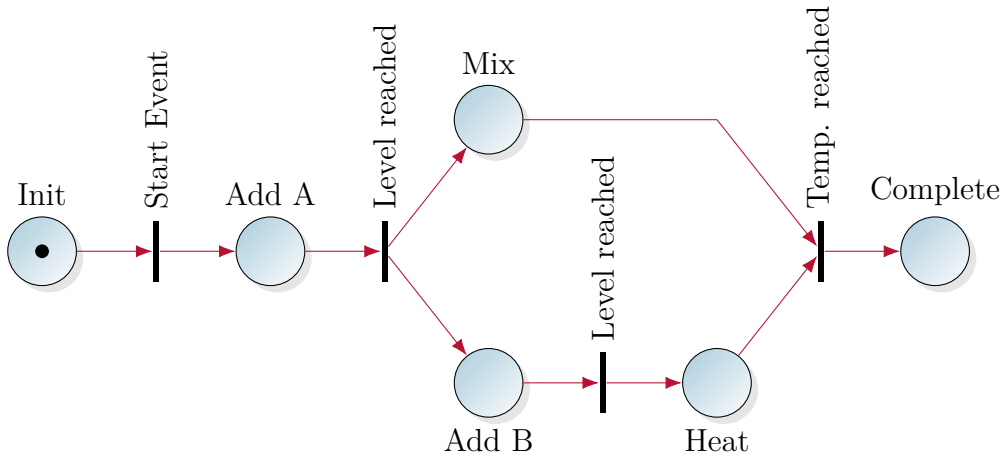


Figure 2.15: Batch recipe defined as a Signal Interpreted Petri Net, applied to the Phase procedure of Figure 2.3. The small solid dot at the *Init* Node indicates the currently active state and is forwarded to the next state if the following transition, which is *Start Event* in this case, is triggered.

This means, that a transition must be followed by a place and vice versa. At the beginning of the example, there is just one token, which indicates the initial state. This state is followed by a start event transition. Generally, if all previous places are marked with a token, all these tokens from previous places are taken and one token is put into all following places, that are directly connected to the transition. Therefore, the number of tokens in a Petri is not necessarily constant. At transitions, additional tokens can be created or destroyed. If Petri nets are used to describe batch processes, the principle of transitions is extended. In addition to the condition, that all previous places must be marked with a token, an external *event-based* condition, like the reaching of a specific fluid level or temperature, can be added. This kind of Petri nets is called Signal Interpreted Petri Net (SIPN).

Tabular Batch Recipe Description

In contrast to these graphical representations of batch recipes, Godena et al. [20] proposes a tabular notation, which is probably easier to implement. As an example, the Phase procedure of Figure 2.3 is mapped to a tabular batch recipe and illustrated in Table 2.2. Phases are represented as cells of the table. An additional asterisk (*) indicates a dominant Phase with an end condition, e. g., a certain filling amount or end temperature, as described in Section 2.1.2. All Phases of the same row are executed simultaneously. If all dominant Phases of the active row are completed, the phases of the next row are started.

Table 2.2: Batch recipe in a tabular notation of Godena et al. [20], applied to the Phase procedure of Figure 2.3. The asterisks indicate dominant Phases with an end condition.

Add A*	
Add B*	Mix
Heat*	Mix

2.5 Shortest Path Problem

Transport systems are widely used in the industry, like for pumping liquids through pipes. It is very beneficial to add redundancy to such systems, since this enables to execute multiple transporting actions. Another benefit of redundant transport paths is, that scheduled and unscheduled events can be handled with less downtimes. However, this requires dynamic path planing algorithms, which are able to react to such events and calculate the shortest path depending on the current situation.

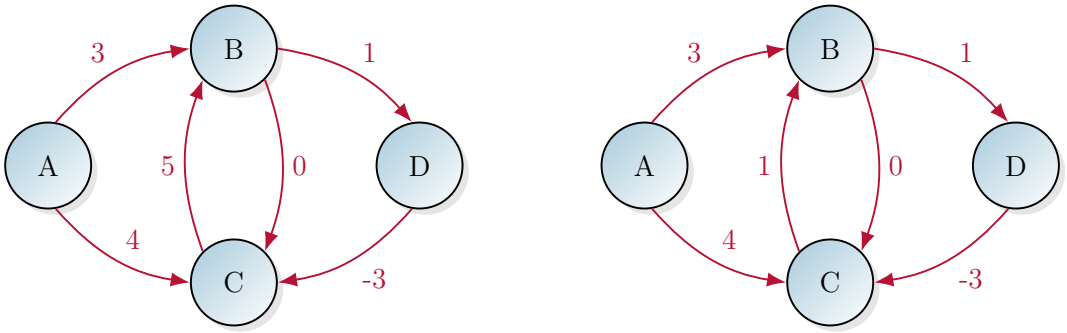
2.5.1 Graph Fundamentals

A graph describes a set of *Nodes* (also called *Vertices*), which are connected via *Edges*. Depending on the application of the graph, it features specific properties. Basically, graphs are classified in directed or undirected and weighted or unweighted graphs. The weight of edges can, for example, be used to describe the distance of the route. This information enables to find the shortest path between two nodes in a graph, which is called *shortest path problem*. Weighted graphs can further be distinguished, whether a negative edge weight is allowed, or not. However, a major problem of negative edges is, that negative cycles can occur, where the total edge weight of a path from a node over one or more edges back to the original node is less than zero. In such a graph, the shortest path is not defined. This problem is illustrated in Figure 2.16. In Figure 2.16a, the possible paths from node A to node C are:

$$\text{weight}(A \to C) = 4 \tag{2.1}$$
$$\text{weight}(A \to B \to C) = 3 \tag{2.2}$$
$$\text{weight}(A \to B \to D \to C) = 1. \tag{2.3}$$

Therefore, $A \to B \to D \to C$ is the shortest path. However, in Figure 2.16b, where just the weight of the edge from node C to node B has changed, it is not possible to determine a shortest path, since this would yield

$$\text{weight}(A \to B \to D \to C \to B \to D \to C \to B \to ...) \to -\infty. \tag{2.4}$$



(a) no negative cycle

(b) negative cycle between nodes $B \rightarrow D \rightarrow C$

Figure 2.16: Sample graph with negative edge weight. Figure (a) has no negative cycle, whereas in Figure (b) there is a negative cycle.

To prevent such negative cycles, it is obvious, that the sum of every cycle must be greater or equal to zero.

2.5.2 Shortest Path Algorithms

In order to find the shortest path in a graph, there are two different kinds of algorithm classes, single-source and all-pair shortest path algorithms [70]. Single-source algorithms, on the one hand, are designed for calculating one shortest path between two nodes. On the other hand, all-pair shortest path algorithms calculate shortest paths between any two nodes in a graph. Since the latter ones are designed for calculating all paths at once, they are much more efficient for such situations. However, in a dynamic scenario, where the graph changes, e. g., due to blocked transport routes, they are not as efficient.

In the following sections, some commonly used shortest path algorithms are described. Due to the target of implementation, only single-source shortest path algorithms for weighted and directed graphs are handled.

Dijkstra's Algorithm

Dijkstra's algorithm [71] is one of the best-known shortest path algorithms in graph theory [70]. It determines the shortest path for directed or undirected graphs with non-negative edge weights. At first, the algorithm starts with an initializing phase. In this phase, the distance of the source node is set to zero and the distance of all other nodes to the source node is set to infinity. The algorithm stores two values for every node, the distance to the source node and the previous node. After initialization, the algorithm starts analyzing all neighboring nodes of the source node and updates their stored distance, based on the weight of the edge, that connects them. Now, the algorithm takes the node with the shortest distance and analyzes all reachable neighboring nodes. If a node can be reached in various ways, of course, the shortest possible distance and

the corresponding previous node is stored. Therefore, while computing, the algorithm always stores *valid* paths of visited nodes, however, they are not necessarily the shortest. The shortest path is just known at the end of the algorithm. This procedure is repeated, until all nodes are analyzed once, and therefore, the shortest path is found.

A* Search Algorithm

The A* (A Star) Search algorithm [72] is another algorithm for determining the shortest path in a non-negative weighted graph. It is a variation of Dijkstra's algorithm, that adds heuristic information for finding the shortest path [70]. This heuristic is used to estimate the distance between source and target node. However, this requires additional information of the graph, like the representation of the graph in a coordinate system, where the distances can be determined from the coordinates. A typical field of application of the A* Search algorithm is route planning in road networks. In such a case, the heuristic information is given by the beeline between start point and target. The initializing phase is similar to Dijkstra. After this phase, the algorithm starts analyzing all neighboring nodes of the source node. A* Search also stores a distance value and the previous node, however, the stored distance value is not just the distance from the source node to the currently analyzed one, but the estimated distance to the target is added. Based on this distance, the node with the shortest value is the starting point of the next step. This procedure is continued until the target node is reached. Since the estimated distance to the target represents the absolute minimal distance, unfavorable nodes with a higher distance need not be analyzed. Therefore, the A* Search algorithm is usually much faster than Dijkstra's algorithm.

Bellman-Ford Algorithm

The shortest path algorithm of *Bellman-Ford* [73] (also known as *Bellman-Ford-Moore* algorithm) can handle directed graphs with negative edge weights, however, not with negative cycles, which can just be detected. But for all that, the running time of Bellman-Ford's algorithm is longer, than for Dijkstra's algorithm [74]. At first, there is, similar to Dijkstra's algorithm, an initializing phase, where the distance of the source node is set to zero and the distance of all other nodes to the source node is set to infinity. After initializing, the algorithm iteratively tries to reduce this distance. This iteration takes as many cycles as nodes exist in the graph. In every iteration cycle, every edge $e(n_{prev} \rightarrow n_{following})$ is analyzed, if the distance to the following node $n_{following}$ can be reduced. The new distance of the following node is defined as:

$$\delta_{new}(n_{following}) = \min\{\delta_{old}(n_{following}), \delta(n_{previous}) + \omega_{edge}\}, \tag{2.5}$$

where ω equates to the weight of the edge. However, the actual distance to the source node is just known at the end of the algorithm. This procedure is repeated in each cycle. Finally, a negative cycle exists, if any distance can be reduced in the last iteration cycle. Otherwise, the shortest path is found.

2.6 Research Questions

An emerging challenge of industrial production systems is the increased customer demand of personalized productions. The best way for meeting this requirement is the development of flexible automation systems, where the production procedure can easily be adapted to the demanded goods.

For this, the current state-of-the-art concerning the issue of improving the flexibility of batch productions was analyzed and several concepts and applications were presented in this chapter. As described, the standard IEC 61512 defines a good basis for batch process control. The major characterization of this standard is a strict separation between physical equipment and product definition information, however, it does not give any guidelines, of how this can be implemented. An eligible candidate for this interface is the usage of services in terms of a Service-Oriented Architecture, as already shown by Melik-Merkumians et al. [24]. They define services for heating, cooling, agitating, and adding liquids. These services correspond to typical Phases, as described in IEC 61512. Therefore, two questions arise:

Research Question 1

Can the flexibility of batch productions be improved by breaking these typical services/phases down to more atomic services?

Research Question 2

Is a flexible and modular approach reasonable for systems with low or no changes in the recipes?

CHAPTER 3

Concept of a Flexible Batch Process Control

In this chapter, a concept for a flexible and modular batch process control is presented. Basically, the concept can be used in any batch process domain, like brewery, oil industry, or pharmaceutical drugs productions, however, the following sections concentrates on the domain of tank systems with redundant piping, since this concept will be demonstrated on such a system in the next chapter.

3.1 General Overview of the Concept

The fundament of this concept is the standard for batch processes, IEC 61512 – Batch control [5], which describes a strict separation of the physical equipment and the production instructions, that are defined in the batch recipe. This separation is the first step of a flexible batch process control. However, the standard just gives guidelines, but no instructions, on how this separation can be applied. As indicated in the last chapter, Service-Oriented Architecture (SOA) fits very well for these requirements, although it is not especially designed for using it as a middleware for batch processes. The service itself represents the action, that the recipe processing algorithm requests from the equipment. Such a service request must contain all parameters, that are required to perform the corresponding action. In order to define a batch recipe, which calls these services, an input method for recipe is required. Conceptual details about a Domain Specific Language (DSL) based recipe creation tool and recipe processing will be explained later, in Section 3.2. Due to the SOA characteristic independence of service provider (the equipment) and service consumer (the recipe processing algorithm), both

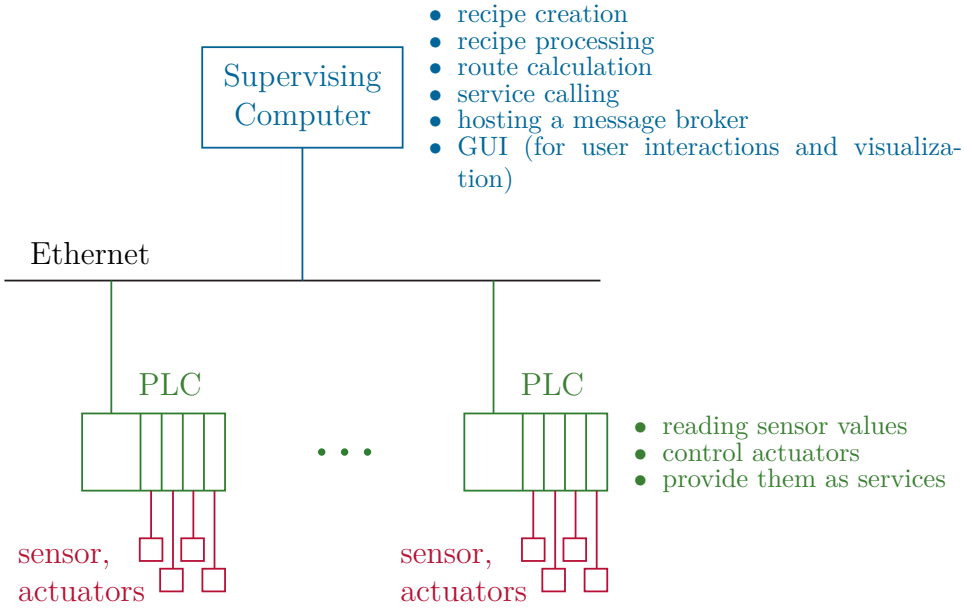


Figure 3.1: Schematic overview of the concept, consisting of a supervising computer and several PLCs with attached sensors and actuators. Furthermore, this figure specifies the tasks of the supervising computer and PLCs.

participants can be developed autonomously, e. g., on different platforms or with different programming languages. The architecture also allows to enhance the flexibility by extending the structure to a distributed system with several service providers, where every provider offers a certain set of services. In order to cope with the requirements of an industrial system, where Programmable Logic Controllers (PLCs) are usually used, the functionality of the service providers are implemented as a program, that is running on a PLCs and controlling the sensors and actuators of the plant. All other tasks, like the recipe processing algorithm, are sourced out into a supervising computer in the same physical network. The tasks of both, the supervising computer and the PLCs are summarized and illustrated in Figure 3.1.

In this concept of a flexible batch process control, it is desirable, that the supervising computer does not need any information of which PLC executes which service. This can just be achieved by using appropriate Message Exchange Patterns (MEPs). First, the structure of the service requests suggests a non-blocking, asynchronous messages with a confirmation notification for service completion. The major benefit of asynchronous messages is, that all service requests can be sent to the providers almost immediately, without waiting for a response. Second, the MEP should handle one-to-many messages, for which the Publish-Subscribe patterns suits perfectly. After subscribing, all service providers receive the service request and just act if they are responsible for this certain action. Additionally, the messaging protocol must also provide the functionality of a reliable message delivery.

As mentioned in the last chapter, in the domain of pipe systems with redundant pipes, dynamic path planing algorithms can also be used for enhancing flexibility. Section 3.3 devotes on this challenge. The usage of a dynamic path planing algorithm has two essential benefits. First, model changes can be implemented quickly without modifying

the program, and second, unavailable equipment (e. g., due to usage or maintenance reasons) will not be requested for a route and alternative routes will be found, if available. However, for this, the path planing algorithm must know a model of the actual plant with all its tanks, pipes, valves, and other components. Section 3.3.1 addresses to this issue and presents a method to model a tank system, based on another DSL.

In order to recap the following sections, Section 3.2 proposes a method for creating batch recipes, based on a Domain Specific Language (DSL). For this, the required grammar of the DSL is derived from a proposed example recipe description. Afterwards, in Section 3.3, another DSL is proposed, for modeling tank systems. This DSL provides the tank system information for the path finding algorithm, as described in Section 3.3.2.

3.2 Batch Recipe Creation with a DSL

The domain of batch processes is analyzed for implementing service actions for batch processes. IEC 61512 names just a few examples of process actions, as there are *add*, *heat*, and *hold*. There are several other references [13], [14], that name other actions, e. g., *cool* or *mix* (also called *agitate*) to cope the typically required actions of batch productions. In this concept, and with respect to the abilities of the demonstration plant that is used the next chapter, three basic processing actions are used. These actions are

- *add*: for pumping a certain amount of liquid from one tank into another,
- *heat*: for heating up a tank content to a specific temperature, and
- *mix*: for mixing the content of a specific tank.

The mix action can either be used to mix two different liquids or for a uniform heat distribution in a heating tank. Each process action expects specific input parameters, which are defined by the process expert in the procedural recipe.

As analyzed in the last chapter, DSLs are a simple way to create batch recipes in a specific domain. Although graphical DSLs probably give a better overview of the entire recipe, textual DSLs are much easier to implement. Due to this reason, this concept uses a textual representation of the batch recipe. In order to describe tank systems as assumed above, three processing actions (add, heat, and agitate) as well as a start and stop action have to be made accessible to the recipe creating process engineer. Additionally, all required parameters as well as the processing sequence must be representable. A proposed description of batch recipes is presented in Listing 3.1.

```
start Start
add AddA amount 2000 ml from T310 to T101 after Start
mix Mix1 tank T101 after AddA nondominant
add AddB amount 2000 ml from T320 to T101 after AddA
heat Heat1 tank T101 up to 50.0 °C after AddB
stop Stop after AddB Mix1
```

Listing 3.1: Proposed description of the batch recipe of Figure 2.3. Each line describes a recipe step (*Phase*). Additionally, as defined in IEC 61512, every recipe needs a dedicated **start** and **stop**. In order to identify a Phase unambiguously, every Phase requires a unique name, e. g., *Start*, *AddA*, or *Mix1* in this example. The recipe processing order is specified with the keyword **after**, followed the ID of the corresponding Phase. The recipe also contains all parameters, that are required to process the Phases. Bold and non-bold words describe keywords and parameters/identifiers, respectively.

With this recipe convention, the *heat* and *add* Phase have an implicit end condition, the target temperature or fluid level, respectively. However, the *mix* Phase does not require an end condition, since it is terminated, when the parallel Phase, which is *Heat1* in this example, is completed. For this, the concept of dominant phases [20], as described in Section 2.1.2, is applied. Thus, the *mix* Phase is marked with the keyword **nondominant**, in order to specify the absence of an end condition. This information is important for recipe execution.

Based on this example recipe, a DSL grammar can be defined, which enables to create such recipes. An appropriate grammar in Xtext notation is presented in Listing 3.2.

```
Model:  steps += Step*;
Step:   Add | Heat | Mix | Stop | Start ;
Start:  'start' name=ID;
Stop:   'stop' name=ID 'after' formerStep+=[Step]*;
Add:    'add' name=ID 'amount' amount=INT 'mL' 'from' source=ID 'to'
    target=ID 'after' formerStep+=[Step]*;
Heat:   'heat' name=ID 'tank' tank=ID 'up to' temp=DOUBLE '°C' '
    after' formerStep+=[Step]*;
Mix:    'mix' name=ID 'tank' tank=ID 'after' formerStep+=[Step]*
    dominant='nondominant';

terminal INT returns ecore::EInt: ('0'..'9')+;
terminal DOUBLE returns ecore::EFloat: INT '.' INT;
terminal WS: (' '|'\t'|'\r'|'\n')+;
terminal ID: ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+;
```

Listing 3.2: Proposed grammar for developing recipes in Xtext notation. The *Model* consists of a list of steps, which can either be an *Add*, *Heat*, or *Mix* Phase or a *Start* or *Stop* element. In order to cross-reference former steps, the square bracket notation of Xtext is applied. For conciseness, the Xtext preamble is omitted in this example (see Section 2.4.2).

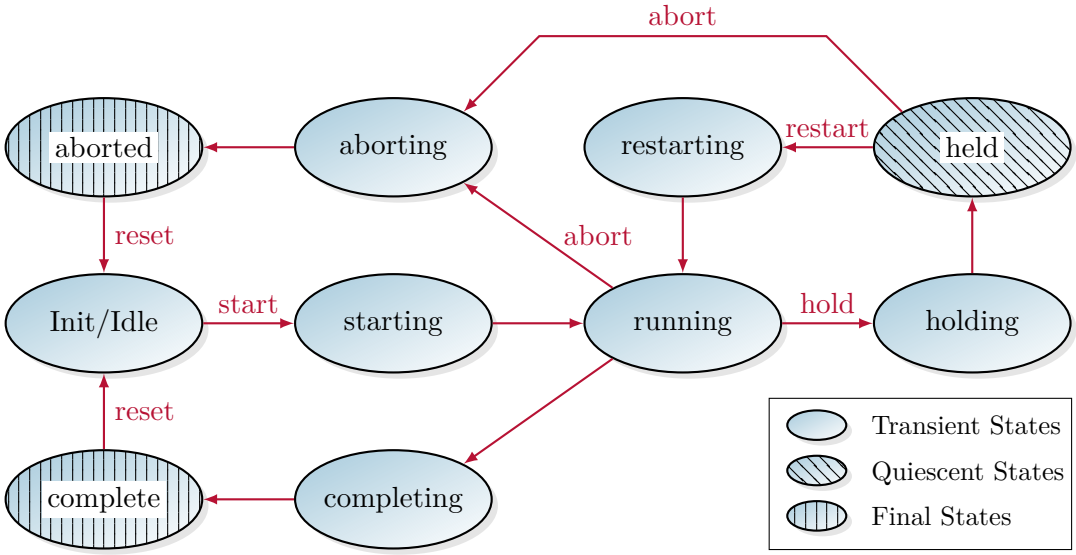


Figure 3.2: Modified version of the procedural state machine of IEC 61512 [5]. User interactions are indicated by arrow labels between the states.

After a recipe is created with the recipe editor, it has to be semantically analyzed and executed. The analysis of the recipe is done by a DSL interpreter. It translates the recipe into a recipe model, which is, in the next step, processed by another algorithm. This algorithm is responsible for a correct execution of the batch recipe. If a Phase has a several preceding Phases, IEC 61512-2 – Batch control – Part 2: Data structures and guidelines for languages [16] says, that all of them must be completed, before this Phase is started. The only exception is a nondominant Phase, which is set to be complete, if all other parallel Phases are completed.

In order to implement user interactions with the process, e. g., for starting, pausing, or stopping the process, IEC 61512 defines a state machine, that describes the current process state and proposes some nonobligatory system states. Brandl [14] recommends implementing at least the states *idle*, *running*, *held*, *complete*, and *aborted*. Based on these states, in this concept, it is useful to add some additional transient states, in order to describe the transitions between the states. This results in a procedural state machine as illustrated in Figure 3.2. Additionally, it shows user initiated and automatic state transitions.

3.3 Path Planning Algorithms in Redundant Pipe Systems

In the last section, a recipe editor was introduced, which enables the creation of batch recipes, based on *Heat*, *Mix* and *Add* Phases. In addition to the parameters of the recipe, the *Add* Phase requires extra information, since, as proposed in Section 3.1, an algorithm for finding a path shall be implemented. However, this requires the information of the actual plant. For this, in the following section another DSL is presented, in order to model tank systems. Afterwards, the shortest path problem is handled with respect to the problem of implementing a required intermediate point into the path.

3.3.1 Modeling a Tank System

In this concept, there are two essential causes for modeling tank systems. It is necessary for the path planing algorithm to find an available path and to provide all additional information to the Phases, in order to translate them into services. If, for example, tank T101 should be heated up to 50.0 °C, the model provides the information about the identifiers of the heating and sensing element in tank T101. An input method for the model can be created in the same way as the batch recipe, with an DSL. In Listing 3.3, a proposed description for tank system models is presented.

```
1  tank T101 {  heater actuator E104 sensor B104
2               mixer E105
3               directedConnectionPoint(T101C01,bi)
4               directedConnectionPoint(T101C02,in) }
5
6  valve V101 { connenctionPoint1 V101C01
7               connenctionPoint2 V101C02
8               flowDirection bi }
9
10 pump P101  { input P101C01 output P101C02 }
11
12 pipe P1001 { length 6 endPoint1 T101C01 endPoint2 V101C01 }
```

Listing 3.3: Proposed description for modeling a tank system. In this example, the required attributes for tanks, valves, pumps, and pipes are defined. A tank requires the information of an optional heater and mixer. Furthermore, a tank requires directed connection points, in order to describe inputs, output, or bidirectional usable connection points. A valve needs the information of two connection points and a flow direction. Pumps are designed with only two connection points, since the flow direction is usually unidirectional. Finally, a pipe is defined by its length and two connection points.

Based on this proposed description of tank systems, a grammar can be developed. For this, again, the Xtext notation is used. The resulting grammar is presented in Listing 3.4.

```
grammar tanksystem.Tanksystem hidden(WS)
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate tanksystem "http://www.Tanksystem.tanksystem"

Model:  entities += Entity*;
Entity: Tank | Valve | Pump | Pipe  ;
Tank:   'tank' name = ID '{'
        ('heater' 'actuator' heaterActuator=ID 'sensor'
           heaterSensor=ID)?
        ('mixer' mixer=ID)?
        directedConnectionPoint += DirectedConnectionPoint+ '}';
Valve:  'valve' name=ID '{'
        'connenctionPoint1' connectionPoint1 = ID
        'connenctionPoint2' connectionPoint2 = ID
        'flowDirection' flowDirection = FlowDirection '}';
Pump:   'pump' name=ID '{'
        'input' input = ID
        'output' output = ID '}';
Pipe:   'pipe' name = ID '{'
        'length' length=INT
        'endPoint1' point1 = ID
        'endPoint2' point2 = ID '}';
Direction:  typeName=('in' | 'out' | 'bi');
FlowDirection:  flowDirection = ('1to2' | '2to1' | 'bi');
DirectedConnectionPoint:    'directedConnectionPoint' '('
   connectionPoint = ID ',' direction=Direction ')';

terminal INT returns ecore::EInt: ('0'..'9')+;
terminal WS          : (' '|'\t'|'\r'|'\n')+;
terminal ID          : ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Listing 3.4: Proposed Xtext grammar for describing tank system models. The *Model* consists of a list of *Entities*, which can either be a *Tank*, *Valve*, *Pump*, or *Pipe*.

After analyzing the developed tank system model, a directed weighted graph consisting of nodes and edges can be created. The keyword *directed* relates to a predefined flow direction of valves, pumps, and connection points of tanks, since, for example, a connection point at the top of a tank can just be an input connection. In order to be able to calculate the shortest available path, it is necessary to describe the system with a *weighted* graph. The weight itself represents the length of the pipes. By manipulating this length, it is also easily possible to set a pipe *in use*, so that it will not be used for other parallel processes.

3.3.2 Finding the Shortest Path

Based on a graph with nodes and edges in a directed weighted graph, there are several algorithms to find the shortest path between two nodes, as described in Section 2.5.2. However, in the domain of tank systems, an additional constraint is, that *add* processes usually require a pump, which is not necessarily part of the *shortest* path between source and target tank. Therefore, the algorithm must be able to handle an intermediate point. Additionally, it has to be considered, that edges must not be taken twice. An easy way to deal with that problem is to set the weight of the pump to a relatively big negative number, compared to the weight of the other edges in the graph. With this, it is ensured, that the pump is used. This requires an algorithm, like *Bellmand-Ford*, which can handle such a negative weight. However, as pointed out in Section 2.5.1, this would probably lead to negative cycles, in which the shortest path is not defined.

An alternative way to find the shortest path is to solve the problem in a directed graph with just positive weights and calculate two subpaths. For this, Dijkstra's algorithm can be used, since it is more efficient than *Bellman-Ford* [74]. The first subpath ranges from the source tank to the input connection point of the pump, and the second ranges from the output connection point of the pump to the target tank. In this case, the weight of the pump is irrelevant. As an example, three similar graphs are illustrated in Figure 3.3 with two tanks T_1 and T_2, a pump P_{ump1}, and some pipes P_x with their weight. For conciseness, valves are omitted in this example. In Figure 3.3a, the shortest path from tank 1 to tank 2, with P_{ump1} as intermediate point, utilizes the subpaths

$$\text{Subpath 1:} \quad T_1 \rightarrow P_1 \rightarrow P_4 \rightarrow P_7 \rightarrow P_{ump1.1} \quad \text{and} \tag{3.1}$$

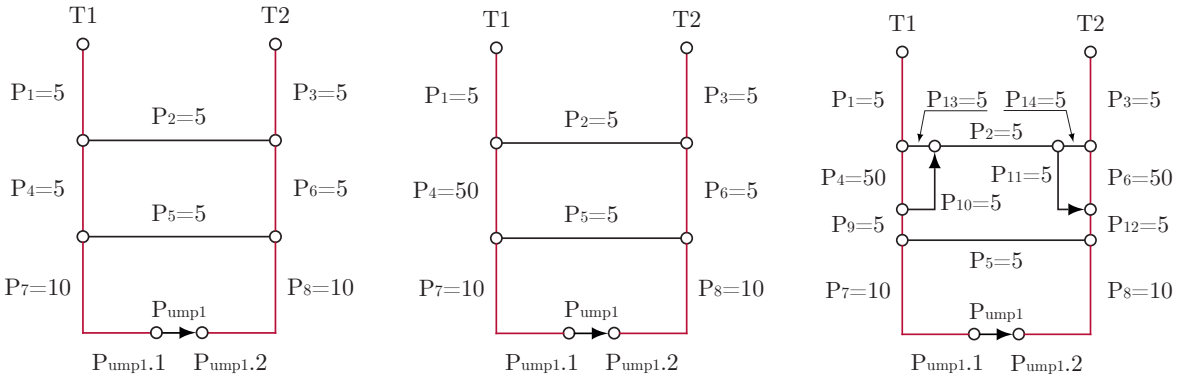
$$\text{Subpath 2:} \quad P_{ump1.2} \rightarrow P_8 \rightarrow P_6 \rightarrow P_3 \rightarrow T_2. \tag{3.2}$$

Based on this graph, a path planning algorithm can easily find the shortest path, since the subpaths are not in trouble with each other. However, this is not the case in Figure 3.3b, where the weight of P_4 is modified to a much higher number. The first subpath from tank 1 to the pump would now use the route

$$\text{Subpath 1:} \quad T_1 \rightarrow P_1 \rightarrow P_2 \rightarrow P_6 \rightarrow P_5 \rightarrow P_7 \rightarrow P_{ump1.1} \tag{3.3}$$

and the second subpath could not find a route, since pipes cannot be used twice. However, this can be bypassed, if, in a second step, the second subpath (from the pump to the target) is evaluated primarily and the first subpath (from source tank to the pump) afterwards. If two valid paths with different total weights are found, the shorter one is used. In graphs with more than one pump, this must be evaluated for every pump.

This path planning concept can find the shortest path in almost every graph, however, there are some special cases, like in Figure 3.3c, where this algorithm fails, although a path from T_1 to T_2 would be possible. Anyway, this algorithm is sufficient in the gross of typical cases.



(a) Graph 1 (b) Graph 2 (c) Graph 3

Figure 3.3: Path planing for different graphs. The shortest path is always highlighted in red. In Figure (a), the presented algorithm can easily find the shortest path. In Figure (b), where the weight of P_4 is modified, the first subpath from the source to the pump would utilize P_6 and therefore, the second subpath from the pump to the target is not possible without using pipes of the first subpath. However, if the order is inverted and the second subpath is evaluated first, the algorithm finds the shortest path. Anyway, there are pathological graphs, like in (c), that are specially constructed, so that they cannot be handled by this algorithm, since for both orders of evaluation, the first subpath would claim pipes of the second subpath.

3.3.3 Processing of the Add Phase

With the knowledge of the shortest path and the current tank system model, all required components, which are part of the path are now specified. The *Add* Phase is broken down into its single valves, a pump, and a control unit, which is responsible for monitoring the filling state. For this, the Procedural Model of IEC 61512 (see Figure 2.2) is extended with an additional level for these elementary components beneath the Phase level, as it is proposed by Lepuschitz and Zoitl [66]. In order to call the physical components, the PLCs must offer them as services. Therefore, for the *Add* Phase, three services are required, for switching a valve, controlling a pump, and monitor the fluid level of a tank. The valve and pump service requires only the name of the target component and the desired switching state. The service for monitoring the fluid level requires the name of the level sensor and the target filling amount.

Summarizing, an overview of the entire mapping from the Phases to the services is presented in Figure 3.4.

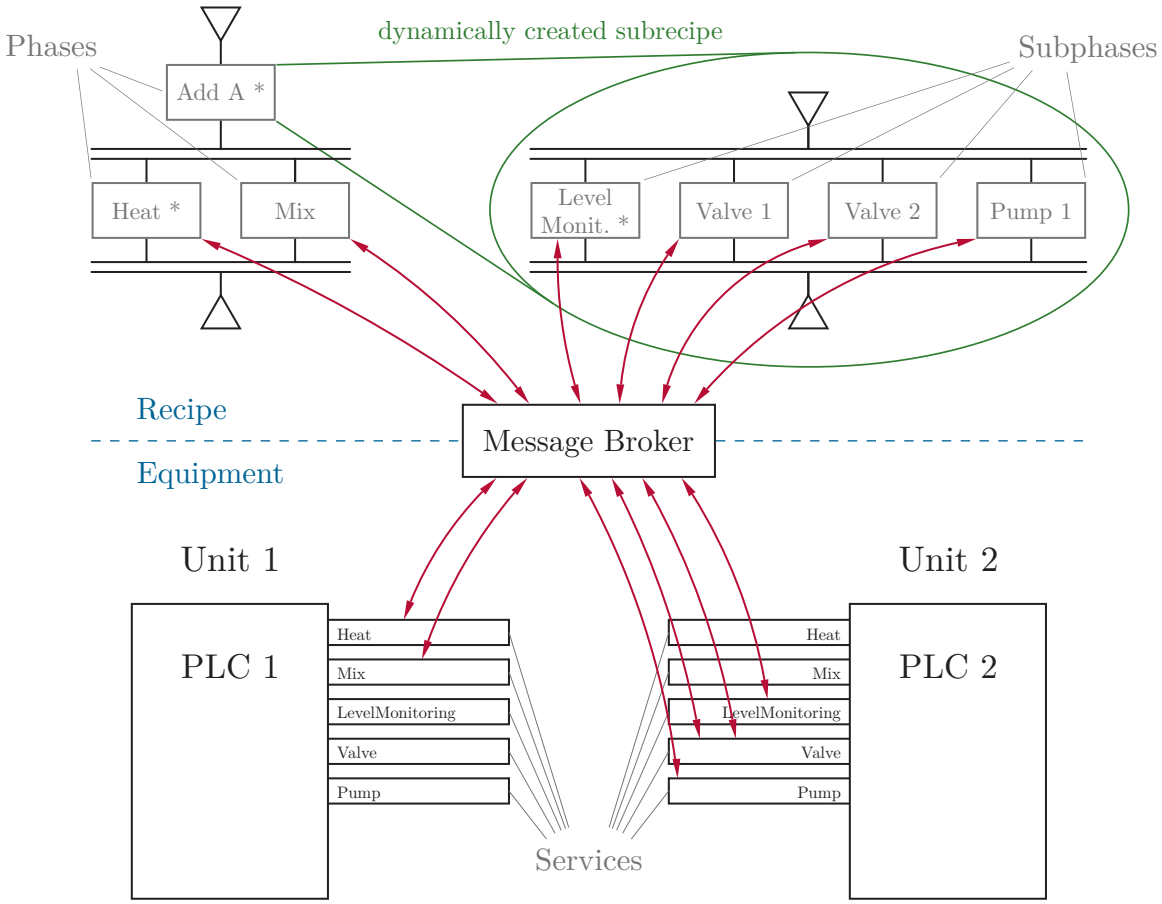


Figure 3.4: Schema of the bridge between the Phases of the procedural recipe and the services, that are executed on the PLCs. It also shows the break down of the *Add* process to a parallel execution of the required valves and pump services as well as the dominant SubPhase *Level Monitoring* for checking if the desired filling amount is reached. The internal structure of the subrecipe of the add process is dynamically created, based on the shortest path from the source to the target tank. An asterisk inside a Phase/SubPhase indicates a dominant Phase/SubPhase with an end condition, as defined by Godena [20].

CHAPTER 4

Implementation of the Flexible Batch Control Concept

In this chapter, the implementation of the previously described concept for a flexible batch process control is described in detail. For demonstrating purposes, the implementation is shown on two linked laboratory tank system plants, a Festo Didactic process industry demonstrator plant, and a custom built storage tank system, as depicted in Figure 1.2.

4.1 Program Overview and Basic Design Decisions

In the previous chapter in Figure 3.1, a schematic overview of the hardware of this concept was already illustrated. Based on this concept, in this section, an overview of the implementation and its software components is presented and illustrated in Figure 4.1.

The concept consists of two different types of computing components, PLCs and a supervising computer. Typically, PLCs are programmed with programming languages of IEC 61131-3 – Programmable controllers - Part 3: Programming languages [61] or IEC 61499 – Function Blocks [63]. Due to this concept of distributed units, IEC 61499 is the first choice. The event-driven execution model of this standard is also very beneficial, since if the computer requests a service from a PLC, the corresponding event is directly triggered and need not be polled all the time, as in the cycle execution model IEC 61131-3. A convenient IEC 61499 programming tool is Framework for Industrial Automation & Control (4diac) [75], which is an Integrated Development Environment

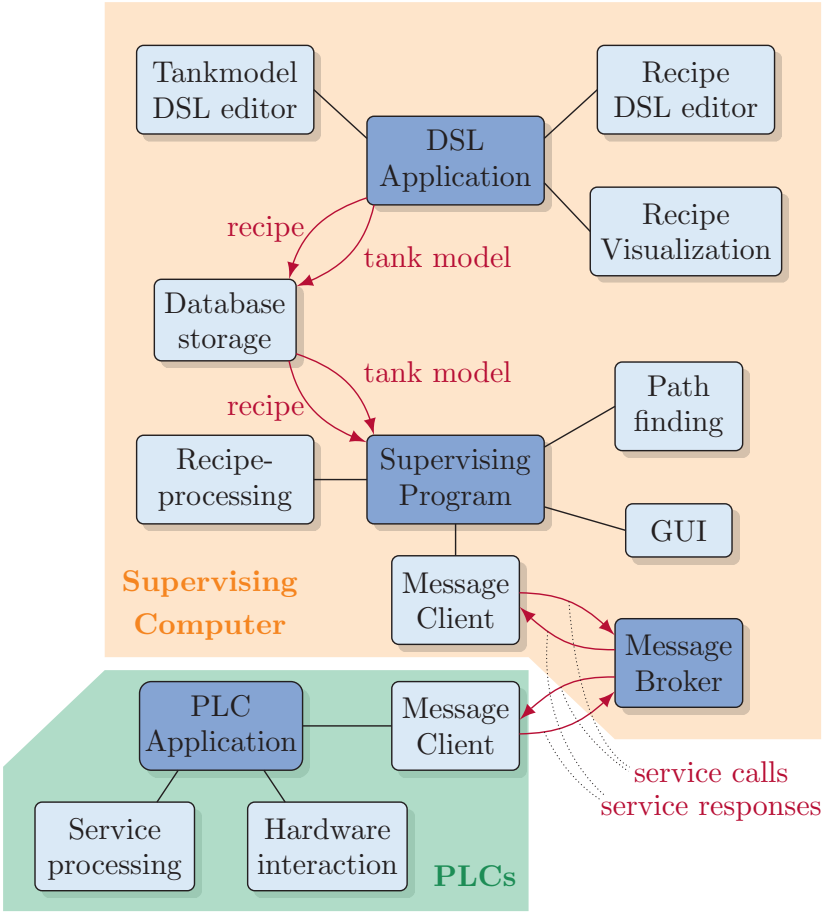


Figure 4.1: Schematic overview of the required tools (highlighted dark-blue), their tasks (highlighted light-blue) and communication interfaces (depicted as red paths).

(IDE) for creating IEC 61499 based distributed control applications. These programs are executed on the 4diac Runtime Environment (FORTE), which is a lightweight C++ program, that can be compiled for almost any platform. By embedding vendor specific Application Programming Interfaces (APIs), like Beckhoff's ADS protocol, the access to hardware components, as sensors and actuators, can be integrated into IEC 61499 Function Blocks (FBs).

The communication between supervising computer and PLCs requires a reliable messaging system. As analyzed in Section 3.1, for this, a messaging system based on the Publish-Subscribe pattern is preferred, since messages can be sent to all PLCs, that are subscribed to a specific topic. In Section 2.3.3, some commonly used examples of messaging methods are described. Although the OPC Foundation announced support for this pattern, currently, there is no implementation available. Devices Profile for Web Services (DPWS), could be used, however, due to the expected high amount of small data packages for all service requests, it is not very suitable, since the message overhead is too high. The major characteristic of ZeroMQ is, that typically no message broker is integrated. However, since a reliable communication is necessary, this functionality

must be implemented additionally. Finally, both, Message Queue Telemetry Transport (MQTT) and Data Distribution Service (DDS) are based on a Publish-Subscribe architecture and meet the demands of a reliable communication protocol. Since 4diac provides native support for MQTT, this protocol is used.

On the supervising computer, there are three programs running. The first program is the Recipe and Tank System *DSL Application*, which is used as front-end for creating batch recipes and tank system models. Both DSLs are created with the open-source framework *Xtext* [55], which is a powerful language workbench (see Section 2.4.1) for developing textual DSLs. However, since textual DSLs are not as illustrative as graphical DSLs, the visualization tool *EuGENia* [76] is used. It is another Eclipse tool and works well with Xtext. This tool enables to visualize the recipe and highlight currently active processing steps. Section 4.2 describes the process for creating and visualizing DSLs, utilizing these tools.

The second and central program is a supervising Java application, which acts as the controlling component. It handles the user interactions of the Graphical User Interface (GUI), the recipe processing algorithm, and the path finding algorithm based on the current tank system model. For accessing the recipe and tank system model, a common database is used as communication interface.

Since MQTT is used as platform for communication between the supervising computer and PLCs, the third task of the supervising computer is to host an MQTT broker. For this purpose, the open-source message broker *Mosquitto* [77] was used. This broker can be configured individually, e. g., for a reliable in-order message delivery. By configuring the Quality of Service (QoS) level of the communication, the broker can guarantee, that messages are delivered exactly once.

4.2 Domain Specific Language Editors and Visualization

In this section, the implementation of the aforementioned DSLs for creating procedural batch recipes and modeling tank systems in Xtext is described. The development of a new language with Xtext consists of two steps. The first one is to define the grammar of the language in a Extended Backus - Naur Form (EBNF)-like notation. Afterwards, the language has to be created, based on this grammar. When this is done, an Eclipse Application with the new language can be started. Within this application, a user can write sentences based on the grammatical rules, as defined before. These sentences are stored in an Eclipse Modeling Framework (EMF)-metamodel. However, actually, there is no functionality assigned, so, in the second step of language creation, a *code generator* or *interpreter* has to be developed, in order to analyze the EMF-metamodel and create or execute the desired program. Code generators just create code for a specific target language, whereas interpreters directly execute a program, with the base data

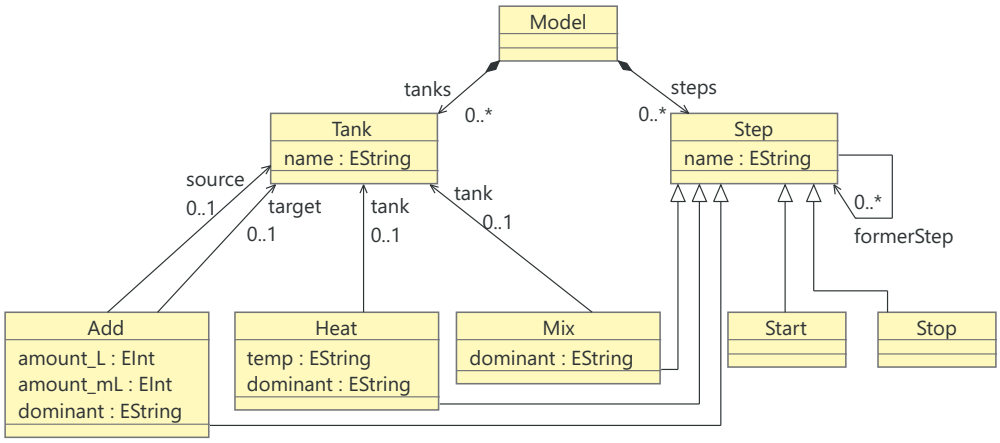


Figure 4.2: Class diagram of the Procedural Recipe Language.

of the EMF-metamodel. In this implementation, both, the procedural recipe language and the tank system language, use an interpreter. Xtext directly executes the code generator/interpreter, when the user-created content is stored. The recommended [60] programming language for code generation/interpretation is *Xtend* [78], which is based on Java. In addition to the Java main functionality, it provides some useful features for language creation, like, multi-line template expressions, extension methods, type-based switch statements, or lambda expressions [55], [79]. By utilizing these features, Xtend programs are more compact than Java programs and the code is easier to read.

4.2.1 Procedural Recipe Language

A major benefit of using Xtext is the integration of several features for an easy language development, like, a syntactic and semantic coloring, cross referencing to existing objects, error checking, auto-completion, formatting, or hover information with quick fix proposals [60]. By leveraging these features, batch process experts can easily develop batch recipes.

Grammar Definition of the Procedural Recipe Language

Basically, as illustrated in Listing 4.1 and visualized in Figure 4.2, the grammar of the Procedural Recipe Language corresponds with the recipe design of Section 3.2. However, in order to simplify the recipe creation and prevent typing errors, the Xtext feature of cross-referencing to existing object is integrated. For this, all tanks are defined at the beginning of the recipe, and can afterwards be used in the actual recipe. When the editor expects a tank, all available tanks are proposed in a list, and the user can easily choose the desired one. Furthermore, the transfer amount for *Add* processes can be specified in liter or milliliter. Based on these grammatical rules, an example of a batch recipe is illustrated in Listing 4.2.

```
grammar tanksystem.recipe.Recipe with org.eclipse.xtext.common.
    Terminals
generate recipe "http://www.recipe.tanksystem/Recipe"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model:   {Model}
         'Tanks'
         tanks+=Tank*
         'Recipe'
         steps += Step*;
Tank:    name=ID;
Step:    Add | Heat | Mix | Stop | Start  (';')?;
Start:   'start' name=ID ('after' formerStep+=[Step]* )?;
Stop:    'stop' name=ID 'after' formerStep+=[Step]*;
Add:     'add' name=ID 'amount' ((amount_l=INT 'l') | (amount_ml=INT
     'ml')) 'from' source=[Tank] 'to' target=[Tank] 'after'
    formerStep+=[Step]* (dominant='nondominant')?;
Heat:    'heat' name=ID 'tank' tank=[Tank] 'up to' temp=DOUBLE '°C'
    'after' formerStep+=[Step]* (dominant='nondominant')?;
Mix:     'mix' name=ID 'tank' tank=[Tank] 'after' formerStep+=[Step
    ]* (dominant='nondominant')?;

terminal DOUBLE returns ecore::EFloat: INT '.' INT;
```

Listing 4.1: Xtext grammar definition of the Procedural Recipe Language.

```
Tanks
  T101 T102 T310 T320 T330 T340 T350
Recipe
  start Start
  add Add1 amount 2000 ml from T310 to T101 after Start
  mix Mix1 tank T101 after Add1 nondominant
  heat Heat1 tank T101 up to 50.0 °C after Add1
  add Add2 amount 2000 ml from T101 to T102 after Heat1 Mix1
  stop Stop after Add2
```

Listing 4.2: Example of the Procedural Recipe Language.

Interpretation of the Procedural Recipe Language

While creating a DSL file, like the example of Listing 4.2, Xtext automatically puts the content into an EMF-model, based on the grammatical rules. Now, an interpreter, which is immediately started when the recipe is saved, analyzes this EMF-model and, in this case, stores the content into a database. As database system, the lightweight SQLite library is used. Another task of the interpreter is to analyze the processing order of the single procedural steps of the recipe. This is necessary, since in the editor, there are just the previous elements referenced, but the recipe processing algorithm requires the linkage to the following elements.

4.2.2 Visualizing a Textual Domain Specific Language with EuGENia

In a textual editor, the current processing step and already processed steps cannot be easily indicated. Due to this reason, the Eclipse tool *EuGENia* is used, which addresses to this issue. It extends the automatically generated EMF-metamodel of the Xtext grammar. The visualization can be developed, just with some annotations of this model. Listing 4.3 illustrates the EMF-metamodel of the grammar with the annotations for EuGENia, which are indicated with an At-symbol (@). Note, that an annotation always concerns to the next line, or block. The visualization can also be illustrated in the same Eclipse Application, as the recipe and tank system editor. Based on the annotations of Listing 4.3 and the recipe of Listing 4.2, the output looks as in Figure 4.3. The boxes, Start, Add1, Mix1, and Heat1 are colored by the recipe processing algorithm, which is described in Section 4.4.3. Boxes, that are colored green, indicate a completed processing step and orange boxes indicates a currently active step. Additionally, gray and red boxes indicate steps in a hold and abort state, respectively.

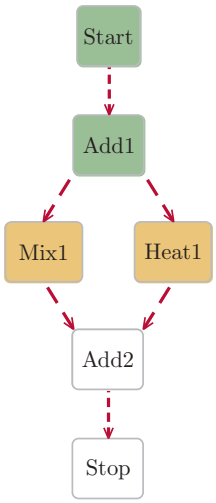


Figure 4.3: Visualization of the recipe of Listing 4.2, created with EuGENia. The colors indicate the actual state of the phases.

```
@namespace(uri="http://www.recipe.tanksystem/Recipe", prefix="
    recipe")
package recipe;

@gmf.diagram
class Model {
  val Tank[*] tanks;
  val Step[*] steps;
}

class Step {
  attr String name;
  @gmf.link(width="2",color="183,18,52",source.decoration="arrow",
      style="dash")
  ref Step[*] formerStep;
}

@gmf.node(label="name", label.icon="false")
class Start extends Step {}

@gmf.node(label="name", label.icon="false")
class Stop extends Step {}

@gmf.node(label="name", label.icon="false")
class Add extends Step {
  attr int amount_L;
  attr int amount_mL;
  ref Tank source;
  ref Tank target;
  attr String dominant;
}

@gmf.node(label="name", label.icon="false")
class Heat extends Step {
  ref Tank tank;
  attr String temp;
  attr String dominant;
}

@gmf.node(label="name", label.icon="false")
class Mix extends Step {
  ref Tank tank;
  attr String dominant;
}

class Tank {
  attr String name;
}
```

Listing 4.3: Annotated EMF model for visualization with EuGENia. The EMF-metamodel is automatically generated by Xtext, however, just the red highlighted annotations have to be added.

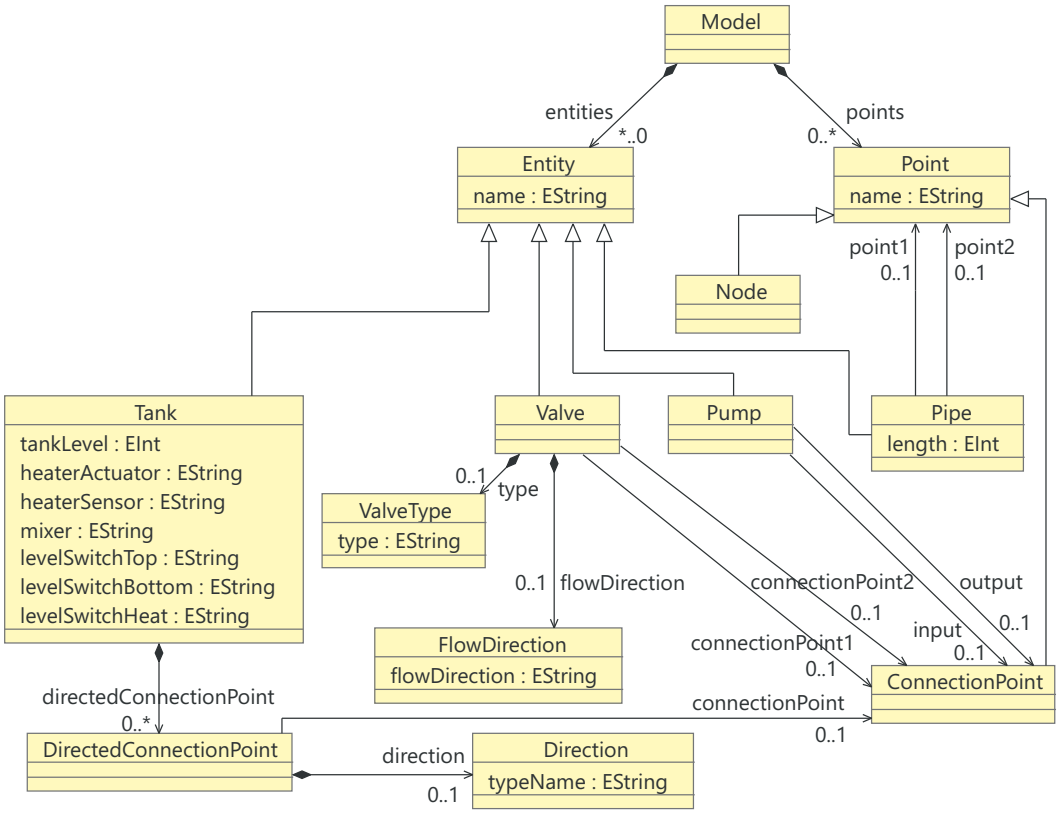


Figure 4.4: Class diagram of the Tank System Language.

4.2.3 Tank System Language

Similar to the editor for creating a procedural batch recipes, the editor for modeling tank systems is also implemented in Xtext. As described in Section 3.3.1, the path planing algorithm and the transition from recipe Phases to services require the actual model of the tank system. For this, a DSL is developed, that can model all these required information.

Grammar Definition of the Tank System Language

In order to model tank systems, a general overview of required components with their properties are specified in Section 3.3.1. The proposed grammar is used as a draft for actually implementing the language. Additionally, the Xtext feature of cross-referencing is implemented for *Points*. A *Point*, can either be a *Node* or *ConnectionPoint*. Furthermore, the implemented grammar, which is illustrated in Listing 4.4, also enables to add level switches to tanks. Based on this grammar, a class diagram, which illustrates all components, their attributes, and links to each other, is depicted in Figure 4.4.

Based on this grammar file, the demonstration plant of Figure 1.2 can be modeled. For conciseness, just a short excerpt of this created tank system file is illustrated in Listing 4.5. In the next section, the evaluation of this file is described.

```
grammar tanksystem.Tanksystem with org.eclipse.xtext.common.
    Terminals
generate tanksystem "http://www.Tanksystem.tanksystem"

Model:   entities += Entity*
         points += Point*
Entity: Tank | Pipe | Valve | Pump  ;
Tank:    'tank' name = ID '{'
         ('heater' 'actuator' heaterActuator=ID 'sensor'
            heaterSensor=ID)?
         ('mixer' mixer=ID)?
         ('levelSwitchTop' levelSwitchTop=ID)?
         ('levelSwitchBottom' levelSwitchBottom=ID)?
         ('levelSwitchHeat' levelSwitchHeat=ID)?
         directedConnectionPoint += DirectedConnectionPoint+ '}';
DirectedConnectionPoint:    'directedConnectionPoint' '('
    connectionPoint = [ConnectionPoint] ',' direction=Direction ')';
ConnectionPoint:    'connectionPoint' name=ID;
Pump:    'pump' name=ID '{'
         'input' input = [ConnectionPoint]
         'output' output = [ConnectionPoint] '}';
Pipe:    'pipe' name = ID '{'
         'length' length=INT
         'endPoint1' point1 = [Point]
         'endPoint2' point2 = [Point] '}';
Direction:  typeName=('in' | 'out' | 'bi');
Valve:   'valve' name=ID '{'
         'type' type = ValveType
         'connenctionPoint1' connectionPoint1 = [ConnectionPoint]
         'connenctionPoint2' connectionPoint2 = [ConnectionPoint]
         'flowDirection' flowDirection = FlowDirection '}';
ValveType:  type = ('continuous' | 'OnOff');
FlowDirection:  flowDirection = ('1to2' | '2to1' | 'bi');
Point:  Node | ConnectionPoint;
Node:    'node' name = ID;
```

Listing 4.4: Xtext grammar definition, used to define the process domain plant system DSL.

```
1  tank T101 {
2    heater actuator E104 sensor B104
3    mixer E105
4    levelSwitchTop S111
5    levelSwitchBottom B113
6    levelSwitchHeat B114
7    directedConnectionPoint(T101C01,bi)
8    directedConnectionPoint(T101C02,in)
9    directedConnectionPoint(T101C03,in) }
10
11 tank T102 {
12   levelSwitchBottom S112
13   directedConnectionPoint(T102C01, bi)
14   directedConnectionPoint(T102C02, bi) }
15                                    ⋮
16 tank T350 {
17   directedConnectionPoint(T350C01, bi) }
18
19 valve V101 { type OnOff
20              connenctionPoint1 V101C01
21              connenctionPoint2 V101C02
22              flowDirection bi }
23                                    ⋮
24 valve V3R8{  type OnOff
25              connenctionPoint1 V3R8C01
26              connenctionPoint2 V3R8C02
27              flowDirection bi }
28
29 pump P101 { input P101C01  output P101C02 }
30 pump P301 { input P301C01  output P301C02 }
31
32 pipe P1001 { length 6 endPoint1 T101C01 endPoint2 V103C01 }
33                                    ⋮
34 pipe P3P04 { length 8 endPoint1 N342 endPoint2 N334 }
35
36 connectionPoint P101C01
37                                    ⋮
38 connectionPoint T350C01
39
40 node N101
41                                    ⋮
42 node N342
```

Listing 4.5: Excerpt of the Tank System Language of the demonstration plant.

Interpretation of the Tank System Language

Similar to the interpretation of the Procedural Recipe Language, also the Tank System Language is interpreted and put into several lists of the same SQLite database. The database is organized with lists for every kind of component. Therefore, there is a list for tanks, valves, pumps, pipes, and nodes. Every list has its type specific parameters, that are stored in columns.

In particular, the process of creating list entries goes in three steps. When the interpreter is started, first, all five lists are deleted if they already exist. Even if the model has just changed slightly, this is procedure is much more efficient, than iterating through all tables and search for differences. In a second step, new lists are created. For four of these five lists, the quantity of columns are predefined, in dependence of the parameters, however, just for the tanklist, the quantity of columns is dynamically adapted, depending on the highest number of connection points of all tanks, which is a priori not known, since it is defined by the user. At last, the EMF-model is analyzed and put into these lists.

4.3 Path Planning for Add Processes

In this section, the implementation of the shortest path algorithm is described. This implementation utilizes the powerful Java library *GraphStream* [80], which is able to model dynamic graphs, find paths, and visualize them.

4.3.1 Requirements for Finding the Shortest Path with GraphStream

In order to use GraphStream, first of all, it requires the graph, which represents the tank system plants, in a special description format, stored as a `.dgs` file. For this, the tank model, which is stored in the SQLite database, must be read out and translated into this .dgs structure. This procedure is executed at the start of the Java main program. The .dgs file is structured in an introducing header, where the file format version and the name of the graph are defined, and a body, where all edges and nodes of the graph are described. Nodes are used for mixing tee nodes, connection points, and tanks. Edges, on the other hand, are used for valves, pumps, pipes, and directed connections in order to model input and output connections of tanks. By adding specific attributes, like the length of a pipe, nodes and edges can be specified. There are no limitations in naming as well as quantities and they can easily be accessed by their name. Additionally, there are also some common attributes, like, `ui.class`, which can be used to classify a node or edge. In the visualization, all elements of a class, e. g., all tanks or valves, can be put into the same look, by specifying it in a standard Cascading Style Sheets (CSS) notation. Later, by changing the `ui.class`

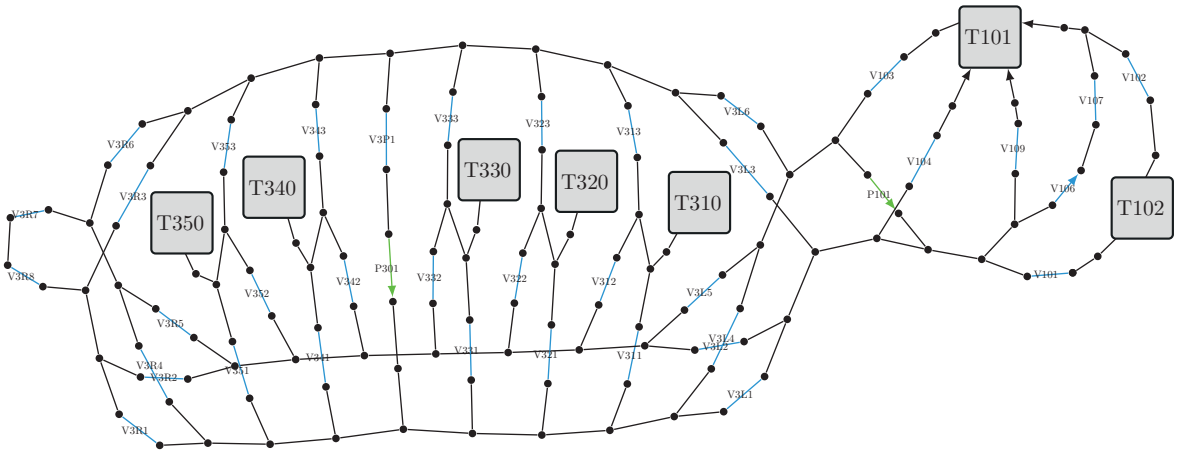


Figure 4.5: Visualization of the demonstrator plant in GraphStream. This graph shows all components, that are required to model the graph, with all seven tanks, that are painted as gray boxes, valves as blue connections, pumps as green connections, and pipes as black connections. The arrangement of the components is automatically created by GraphStream.

this functionality is also used for highlighting active routes in the visualization. The visualization of demonstrator plant via GraphStream is depicted in Figure 4.5.

4.3.2 Implementation of the Shortest Path Algorithm

As described in Section 3.3.2, the demand of a pump necessitates a separation of the shortest path between source and target tank into two subpaths with the pump as intermediate station. Based on the `length` attribute of each edge, GraphStream can be used for find the shortest path between two nodes. For this, GraphStream supports several well-known algorithms, like Dijkstra, Ford-Bellman, or A-Star search. As analyzed in Sections 2.5.2 and 3.3.2, Dijkstra's algorithm is the most suitable, and therefore it was chosen.

The easiest way for finding the shortest path occurs, if the source tank is mounted on a higher level, than the target tank, where the gravity is sufficient to transfer liquids. In this case there are no problems with Dijkstra's algorithm. In order to describe the mounting level of the tanks, the grammar of the Tank System Language is extended with an additional attribute `tankLevel` for every tank, which contains this information.

On the other hand, if the source tank is not higher mounted, the algorithm consists of two parts, first, finding the shortest subpath from the source node to the intermediate node plus the subpath from the intermediate node to the target, and second, the intermediate node to the target node plus the source node to the intermediate node, as described in Section 3.3.2. Since edges cannot be used twice, the length is manipulated and set to a high number. Therefore, the actual length of the pipe must be stored in the `defaultLength` attribute. Afterwards, the total length of both parts is compared

and the shorter one is buffered. This procedure is repeated for each pump and finally, the shortest path is chosen. If any problem occurs, e. g., if the algorithm cannot find a route, an error is saved in a variable.

Since there are pathological graphs, where this algorithm fails (see Section 3.3.2), the entire algorithm is implemented as a self-contained module with the shortest path and error message as required getter methods. Just these two methods are called from the main program. Therefore, the algorithm itself can easily be replaced, without affecting the surrounding program.

4.4 Supervising Program

As introduced in Section 4.1 and Figure 4.1, the supervising program has several tasks:

- offering a path planning algorithm for finding the shortest path for *Add* processes, as already described in the last section,
- providing a GUI for user interactions,
- enabling communication with the MQTT broker, and
- processing the recipe.

Furthermore, a central program component is required, which combines these tasks into a program. In the following sections, the functionality of these tasks is described in more detail.

4.4.1 Graphical User Interface

The GUI, as depicted in Figure 4.6, is created with the Java library *Swing*. For this, the class `GUIview.java` provides two basic functions for user interaction with the program. First, it visualizes tank levels and one temperature of the demonstrator plant, and second, it allows to take action into the recipe processing. For this, all user interactions, that are defined in the concept (see Figure 3.2) are implemented: start, hold, restart, abort, and reset. Since the `restart` action depends on the `hold` action, both share the same button and the functionality is switched, depending on the current state. As Figure 3.2 shows, the state machine does not allow all state transitions, hence, all buttons of illegal state transitions are disabled and just valid transitions are enabled. Additionally, there is a button for updating all sensor values. All of these button actions are forwarded to the central class `Controller.java`, which initiates all further actions.

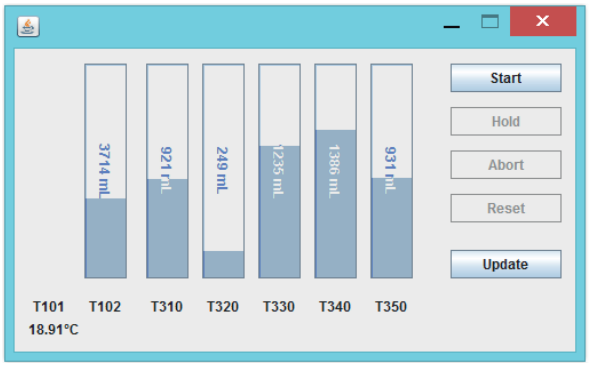


Figure 4.6: The Graphical User Interface indicates all available tank levels and tank temperatures. Furthermore, it provides buttons for user interactions.

4.4.2 Communication with the MQTT Broker

In order to send MQTT messages between Java and other applications, an MQTT-Client implementation is necessary. For this, the open-source tool *Paho* was integrated into the Java program. The implemented program itself consists of two Java classes, `Executable.java` for establishing a connection with the message broker as well as sending messages and `Callback.java` for receiving messages. The Callback class implements a Paho interface with the corresponding method `messageArrived(Topic, Message)`, that is called, when messages are received. Additionally, there is another simple Java class `MqttMsg.java`, which combines message and topic into a structure.

All messages, that are sent or received, have to be encoded or decoded, respectively, with Abstract Syntax Notation One (ASN.1) [81], since the MQTT-Client of FORTE uses this format. This notation allows to send messages over heterogeneous systems and still distinguish different data types uniformly. In the Java program, there is just the *String* data type implemented, since other data types are not required. For Strings, a message looks as follows: the first two bytes of the message `80 00` define the data type String, followed by the length of the String, and the actual String itself. When messages arrive, they are decoded and forwarded to `Controller.java`, which interprets them and perform, depending on the topic, the corresponding actions.

MQTT is able to send several messages almost simultaneously. However, this could not be realized, since there is a malfunction in the MQTT-Client implementation of FORTE. If too many messages are sent in a short interval, every message is immediately confirmed to the message broker, even though, they are not just fully forwarded to the next FB. Hence, messages can overwrite other messages, and therefore they are lost. Since reliability is a strict requirement of automation systems, this issue was bypassed in `Executable.java` by storing messages locally in a queue and just send the next message, when the queue is empty or the previous message is confirmed by an additional MQTT message of the receiver.

4.4.3 Recipe Processing

The recipe processing algorithm consists of several Java classes. In Figure 4.7 an overview of them is illustrated.

`Recipe.java`: is the base class of the recipe processing algorithm. First of all, the recipe is read out from the SQLite database. For every Phase a new object of `Phase.java` is created and all attributes, like the following Phases and Phase specific parameters, are stored into it. Additionally, `Recipe.java` is also responsible for a correct processing of the recipe. If there are just sequential Phases, the next Phase is started, when the previous one is complete. However, if a Phase has several dominant former Phases, all of them have to be completed, until the following Phase/Phases is/are started. If there is a dominant and one or more non-dominant Phases, all non-dominant Phases are set complete.

`Phase.java` is the abstract structure, which is handled by `Recipe.java`. Therefore, it has a uniform appearance and common interfaces to the actual classes with the specific actions, like methods for state transitions, or callback actions. Depending on the current state, this class calls a method of `Colorize.java` in order to indicate the actual state of the recipe visualization of EuGENia by changing the background color of the Phase (see Figure 4.3).

`Steps_Phase.java` is an abstract class, which is just responsible for generalizing all specific classes, that perform the actual actions. This class helps to simplify method calls, since objects of the `Phase.java` need not check, which kind of subclass is required.

`Steps_Phase_Mix`, `Steps_Phase_Heat`, `Steps_Phase_Add` define the actual actions, that are performed if, for example, the Phase is started. For every action, these classes generate the MQTT message strings, which are, subsequently, put into the sending queue of the `Executable.java` object.

`Steps_Phase_Add.java` has an additional functionality implemented, since this class is responsible for transferring liquids from one tank into another and the required components are unknown at this point. When the corresponding Phase is started, this class requests the path from the path finding algorithm and creates a subrecipe with all valves, an optional pump and the monitoring of the tank level. If available, the level sensor of the target tank is chosen, if not, the level sensor of the source tank. The structure of the subrecipe corresponds exactly to the structure of the recipe and so they are not described again. These corresponding classes are `SubPhaseRecipe`, `SubPhase`, and `Steps_SubPhase`. The actual actions are defined in `Steps_SubPhase_TankFillingWatch`, `Steps_SubPhase_Valve`, and `Steps_SubPhase_Pump`.

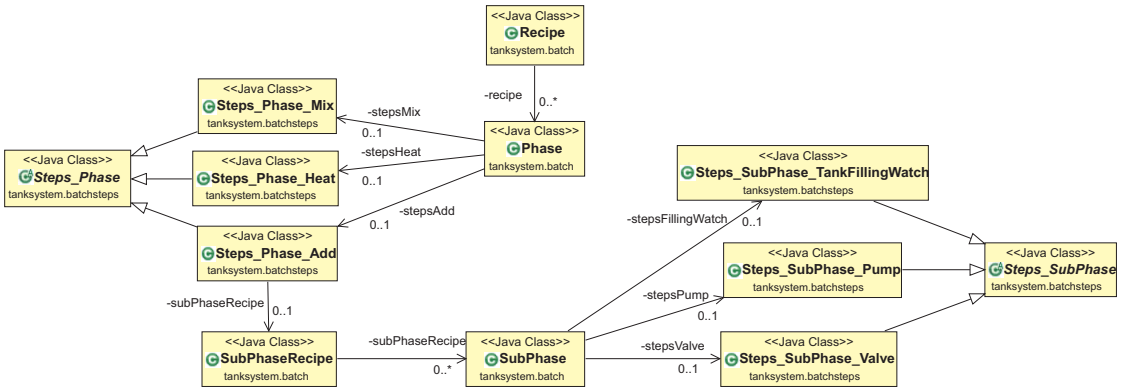


Figure 4.7: Class diagram of the recipe processing algorithm. Due to clarity, just the most important references are illustrated.

4.4.4 Controller Class

Primarily, `Controller.java` is responsible for administrative functionality of the Java program, like instantiation of objects, subscribing to MQTT topics, or starting the GUI. It also implements the `main` method, which is called at program start. Since, as described in Section 4.4.2, MQTT callbacks are forwarded to this class, there is a list (`procedureCallbackList`) of all Phases and SubPhases, which are registered for callbacks, in order to forward receiving messages to the corresponding object. Depending on the topic of the MQTT message, the `Controller.java` class initiates corresponding actions.

Another task of this class is an emergency handling, if tanks are almost empty or full. In both cases, the PLC sends messages to the `emergency` topic, in order to prevent the pump to run idle if the source tank is empty, or a flooding, if the target tank is full. Another emergency state occurs, if the liquid level of a tank is lower than the mounting level of the heating element. If such an undesired state occurs, the recipe is aborted, and therefore, all components are turned off. However, this is just implemented for the level switches, and not for the level sensors, since if a tank has a level sensor integrated, the level is displayed in the GUI and the user can react in case of emergency.

4.5 Implementation of the Services

As announced in Section 4.1, the services are implemented as IEC 61499 FB networks in 4diac, which are afterwards executed in FORTE on the PLCs. Figure 4.8 shows the system configuration of the 4diac system. It describes the distributed system with all devices and their inter-device communication links [64]. In IEC 61499, this representation of the devices is termed *System model*. Since the demonstrator plant contains of two PLCs, there are also two devices defined in the configuration, which are integrated into the same Ethernet as the MQTT broker and supervising computer. The naming of the devices, PLC1 and PLC3, is justified in view of the fact, that the

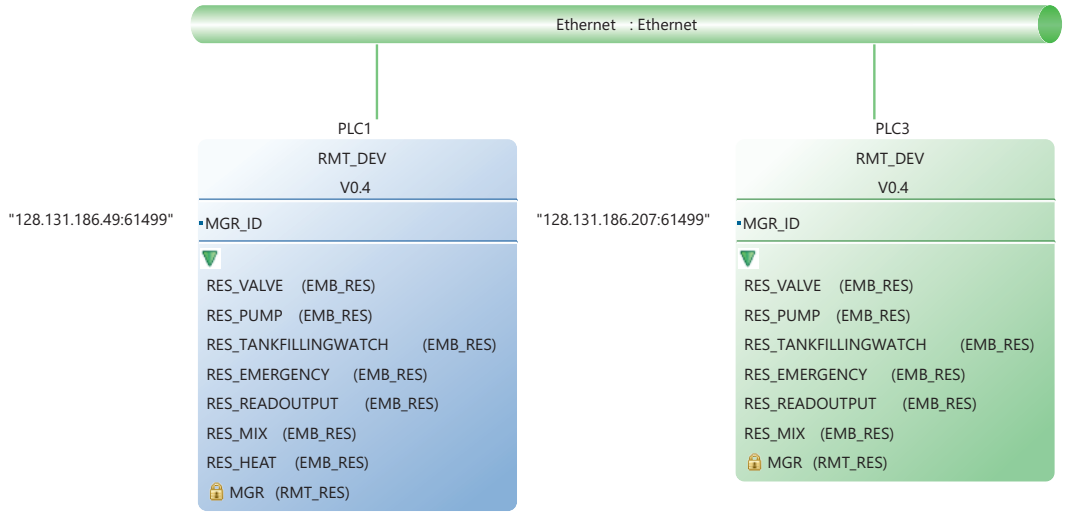


Figure 4.8: 4diac System Configuration consisting of two, via Ethernet connected, devices.

naming of the components of the demonstrator plants also start with one and three (see Figure 5.1).

As defined in the *Device model* of IEC 61499, a device consists of several resources. In this implementation, all services are implemented as applications into separate resources. Summarizing the concept of Chapter 3, five services are specified, for heating, mixing, monitoring a tank level, and switching a valve, or pump. However, due to additional features for emergency handling, and the graphical visualization of the levels and temperatures in the GUI, two more are implemented. Since there are no heating and mixing elements integrated into the subsystem with PLC3, these services and furthermore the corresponding resources are omitted on these devices. All other services are provided on both devices. The applications of these equal services is also equal, however, just the client identification, that is required for the communication with the MQTT broker, is obviously different, in order to address the correct device. In the following sections, the implementation of the single applications is described in more detail.

In order to access the hardware components, as sensors and actuators, Beckhoff provides the ADS protocol. However, this can just be used, when the inputs and outputs of the PLCs are mapped to variables, which are defined in a Global Variable List (GVL). This was done in TwinCAT, which is the official programming tool of Beckhoff PLCs, and uploaded to the PLCs. Additionally, in the main program of the PLCs, the sensor values are converted from a digitization value between 0 and 32767 into a physically interpretable value in milliliters and degree Celsius for the tank levels and temperature, respectively.

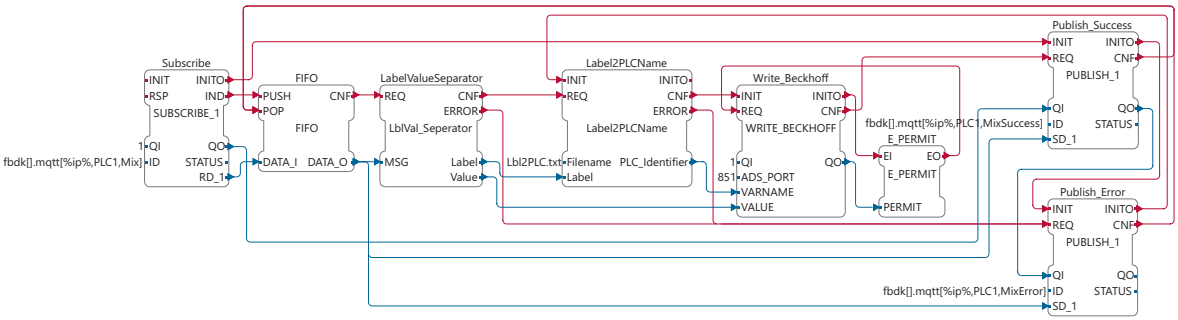


Figure 4.9: Function Block network of the Mix service in 4diac. Disregarding the MQTT topics, this network is equal to the service implementation for the valve and pump service. In the ID of the MQTT topic, %ip% is placeholder for the IP address of the MQTT broker.

4.5.1 Mix, Valve, Pump Services

Disregarding the MQTT topics, the FB network of the services for mixing and switching valves as well as pumps are equal. As an example, the application for the mixing service is illustrated in Figure 4.9. On the left side of the FB network, there is a *Subscribe* FB, which is the receiving station of MQTT messages for the corresponding topic. If a message is published to the corresponding topic, this FB sends an output event on the `IND` channel with the actual message in `RD_1`. Subsequently, the event with the message data is forwarded to a First-In First-Out (FIFO) queue, that buffers incoming messages, if there are more messages, than the remaining program can handle. The message for all these three topics, mix, valve, and pump, consists of the component label, and the desired value, separated by a semicolon, e. g., `E105;true` for turning on the mixer. Now, the *LabelValueSeparator* FB splits this message into the component name and the value. Since the component name need not necessarily be equal to the, in the GVL of *TwinCAT* defined, name, the *Label2PLCName* FB translates it into this addressable name. The translation table is stored in the file, as specified in the `Filename` data input with a syntax: `label;PLC-identifier`, e. g., `E105;GVL.E105`. Afterwards, the PLC specific FB *Write_Beckhoff* initializes the variable and, in a second step, writes the value. For this, Beckhoff's ADS protocol was used at port 851. If the operation is successful, a message is published to the topic *MixSuccess* (*ValveSuccess*, *PumpSuccess*), otherwise, in case of a failure, a message is published to the topic *MixError* (*ValveError*, *PumpError*).

4.5.2 Tank Level Monitoring Service

As Figure 4.10 illustrates, the implementation of the Tank Level Monitoring service contains several FBs, that were already described in the previous section. However, this service has a central controlling component implemented, which is responsible for reading the liquid level of the desired tank, and publishes a message, if it is reached. The service is initialized with a MQTT message, that follows the pattern

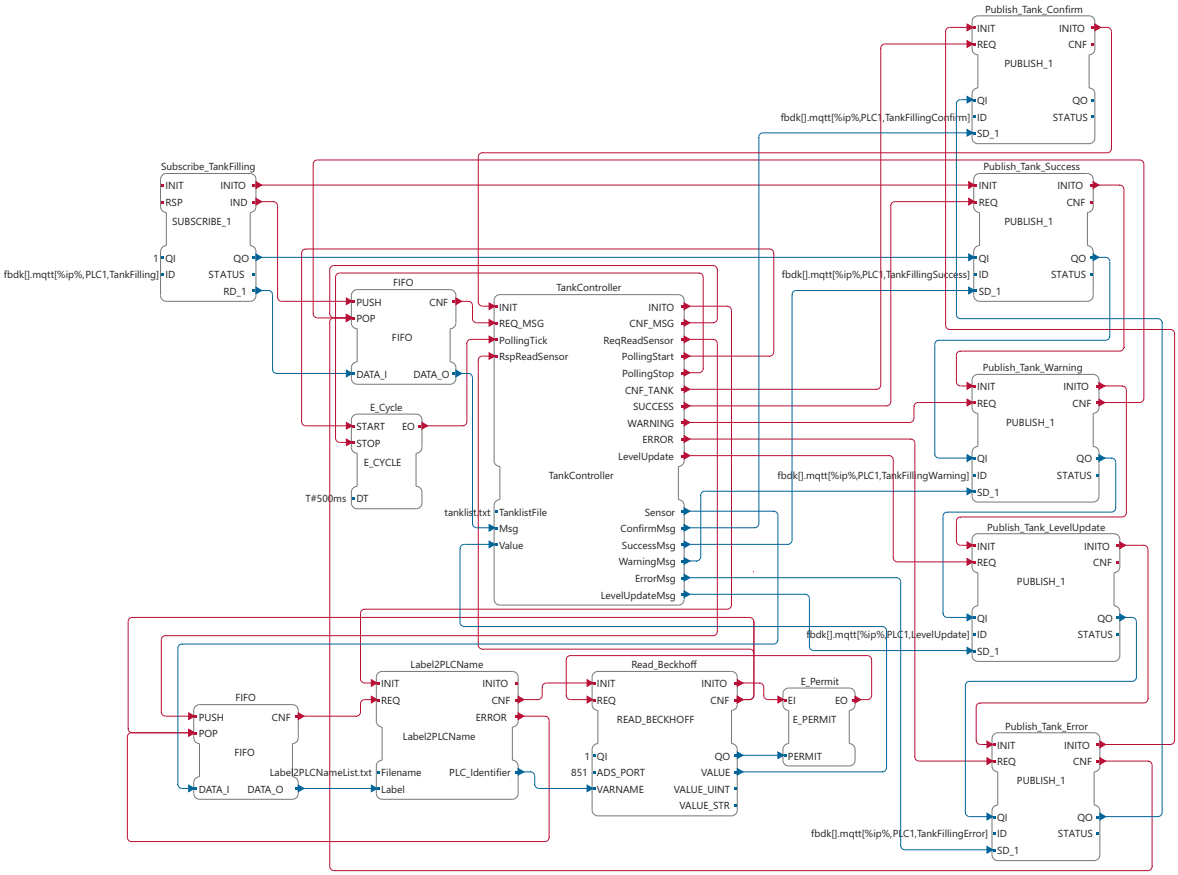


Figure 4.10: Implementation of the Tank Level Monitoring service in 4diac.

`phaseID;tankname;filling;amount`, where `phaseID` is a unique identification of the recipe Phase, `tankname` defines the desired tank, which is to be monitored, `filling` is a boolean value, that describes, whether the tank is to be filled (true), or drained (false), and `amount` specifies the transfer amount in milliliters. Via the `TanklistFile` data input, a file can be specified, where the corresponding level sensor and the conversion factor are stored. When a valid message is received, either, in case of the tank has a level sensor integrated, a confirmation message is published to the *TankFillingConfirm* topic, or, if not, a message is published to the *TankFillingWarning* topic. Afterwards, a cyclic event trigger *E_Cycle* is started, which generates output events, for polling the level sensor. With the first polling event, the controller stores the current level and calculates the target level. If this is reached, a success message is published. Additionally, at every polling tick, the current level is published to the *LevelUpdate* topic, which is used for updating the GUI. In case of an error, an error message is published to the *TankFillingError* topic.

Since there is not just one service per level sensor, but one service per device, it is also possible, that several Tank Level Monitoring services are requested concurrently. Therefore, the controller contains a list, that stores all transfer jobs and polls each level sensor alternatively. If all transfer processes are finished, the cyclic event generator for polling the level sensors is stopped.

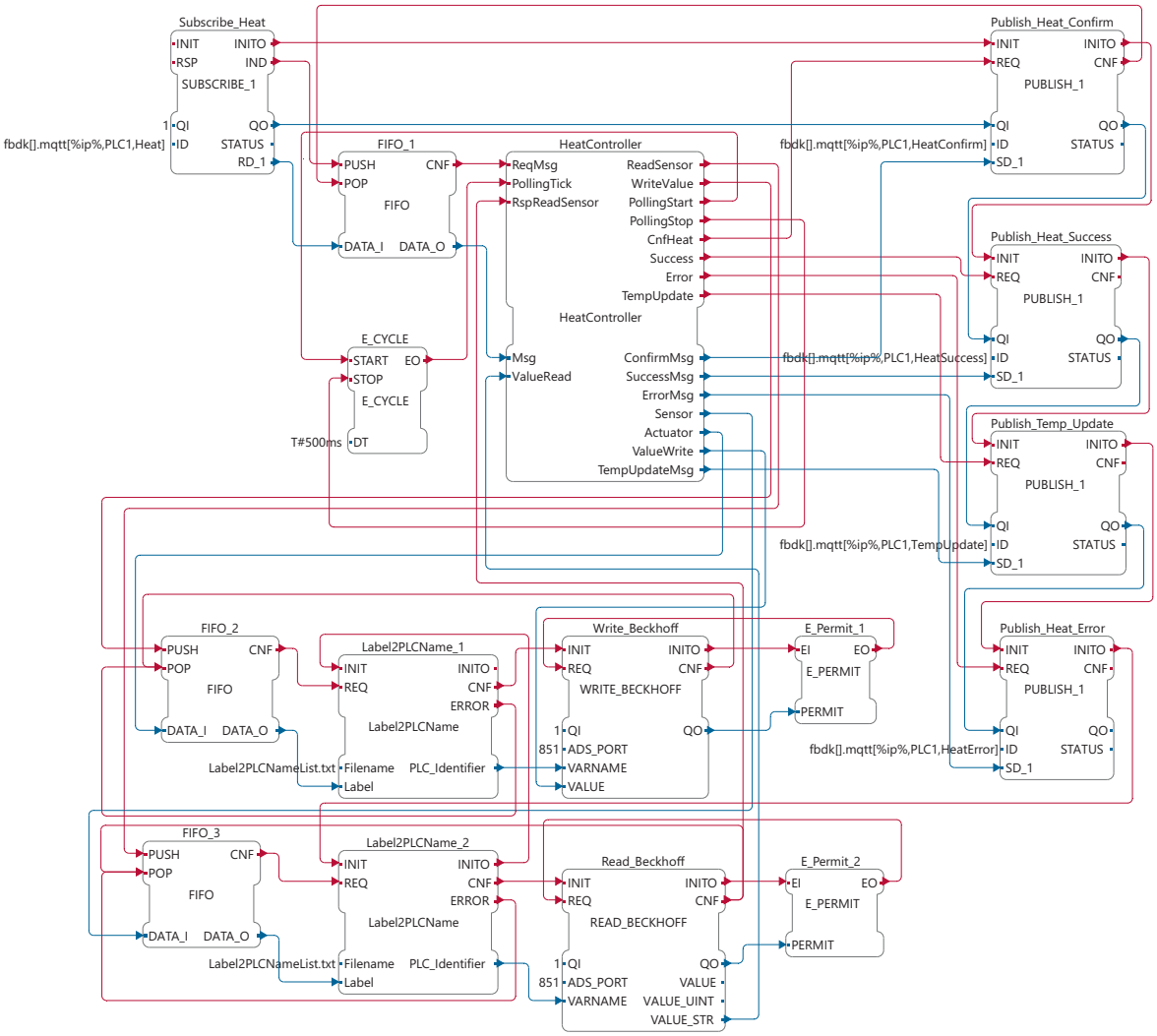


Figure 4.11: Implementation of the Heat service in 4diac.

4.5.3 Heat Service

As Figure 4.11 shows, the FB network of this service implementation consists of almost the same FBs, as the previously explained services. The *Subscribe* FB of this service listens to the MQTT topic *Heat*, which expects a message, based on the template `phaseID;tankname;actuatorname;sensorname;targetTemp`. All of these parameters are either defined in the recipe or tank model. Similar to the Tank Level Monitoring application, there is also a central controlling component, termed *HeatController*, which controls the polling of the temperature sensors and sends events to turn the heating element on/off. There is just a simple control algorithm implemented, which turns on the heating element at service start and turns it off, when the desired temperature is reached. When a special command, like `hold`, `restart`, or `abort`, is triggered by the user, the supervising computer forwards the command to the, in this case, *Heat* topic, with the message `abort;tankname` and this application is now responsible for turning on/off the corresponding heating element.

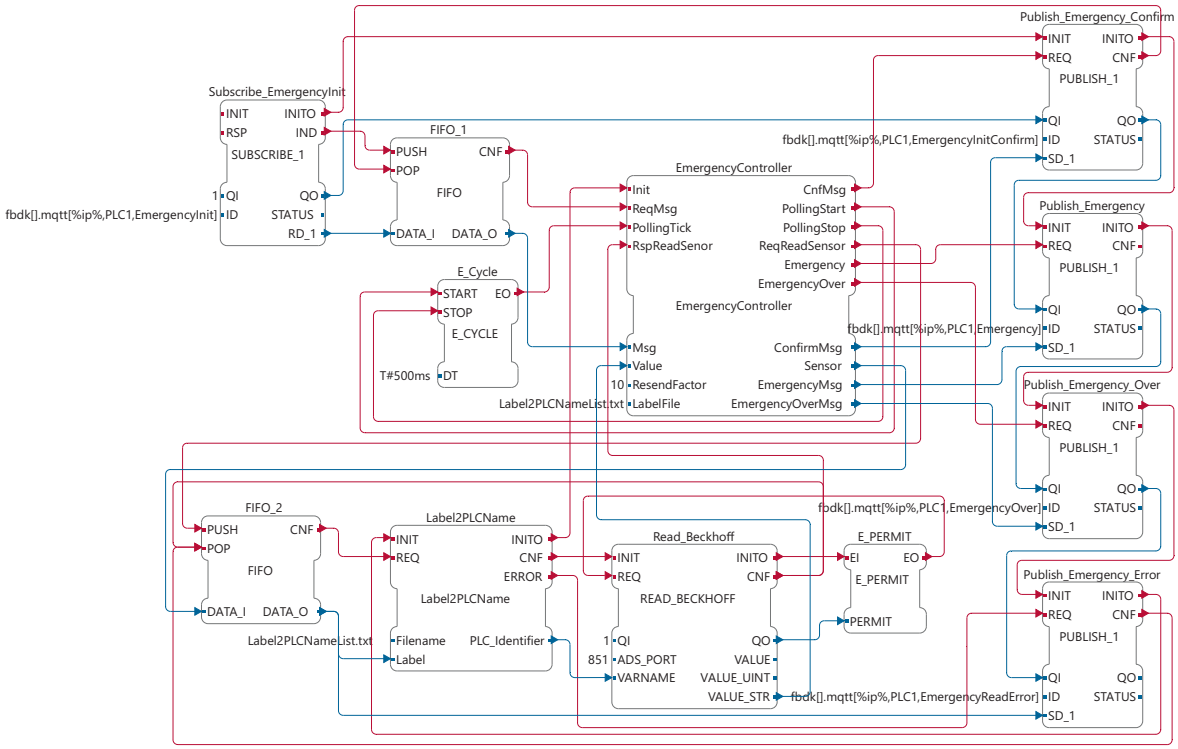


Figure 4.12: Implementation of the Emergency service in 4diac.

4.5.4 Emergency Service

As described in Section 4.4.4, an emergency function is implemented, in order to prevent the pump to run idle, overfilling of the tanks, and heating if there is too little liquid inside a tank. These precaution mechanisms were put into an emergency service, which is depicted in Figure 4.12. It looks similar to the Tank Level Monitoring and Heat service, with a central controlling unit. This service is initialized with the *EmergencyInit* topic, based on a template `sensorname;emergency_condition`, where the emergency condition can either be a boolean condition, like, true/false, or a numeric condition, like greater/lower than a numerical value. As an example, the EmergencyInit message of a level switch at the top of tank T101 is `S111;true`, so, if sensor S111 becomes true, an emergency message is published to the topic *Emergency*. This message is repeated, until the emergency state is over. However, in order to prevent a flooding of emergency messages, the `ResendFactor` defines the ratio between polling ticks to published messages. If, for example, the `ResendFactor` is ten, just every tenth polling tick is published. When the undesired system state is over, a message is published to the *EmergencyOver* topic.

4.5.5 Read Service

In order to illustrate the sensor values of the plant, the read service was implemented. The only objective of this service is to read a sensor value from the PLC and publish it

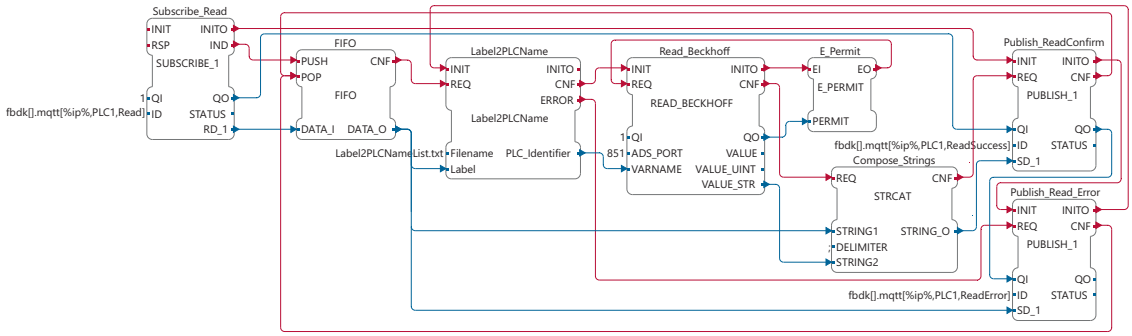


Figure 4.13: Implementation of the Read service in 4diac.

as a composition of the label and the actual value. In Figure 4.13, the FB network of this service is depicted. The subscribe FB of this service listens to the topic *Read*. The message only contains the name of the component. Compared to the other applications, this FB network is similarly constructed, there is just an additional FB, which composes the component name with the value, separated by a semicolon.

CHAPTER 5

Evaluation of the Flexible Batch Process Control

In this chapter, the SOA-based flexible batch process control for two linked Festo demonstrator plants is evaluated. For this, first, the demonstrator plants and their hardware setup is described. Afterwards, the functioning of the add process with its components for finding the shortest path and creation of the subrecipe is presented. At last, the functioning of the parallel processing of the heating and mixing Phases is demonstrated.

5.1 Hardware Setup of the Demonstrator Plant

The demonstrator plant of Figure 1.2 consists of a Festo Didactic process industry plant with two tanks and a custom built storage tank system with five tanks. These subsystems have two paths connecting them, which can be used bidirectionally, in order to enable several ways for transfer processes. The Piping and Instrumentation Diagram (P&ID) of the demonstrator plant is depicted in Figure 5.1. Both subsystems are controlled via a separate off-the-shelf Beckhoff CX5010 PLC with several input and output terminal modules. A full list of all modules and their attached components is given in Table 5.1.

All but one tank have level sensors integrated for measuring the liquid level of the tanks. Tank T101 of the Festo Didactic demonstrator subsystem has no level sensor, however, there are level switches at the bottom and top of the tank in order to recognize a full or empty tank. Additionally, tank T101 has a heating element with a temperature sensor

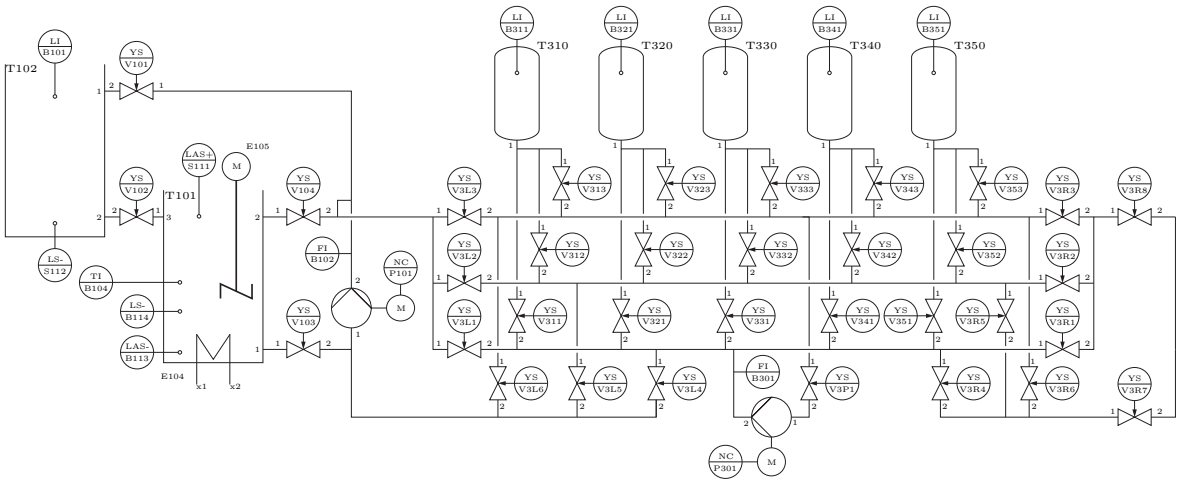


Figure 5.1: P&ID of the demonstrator plant.

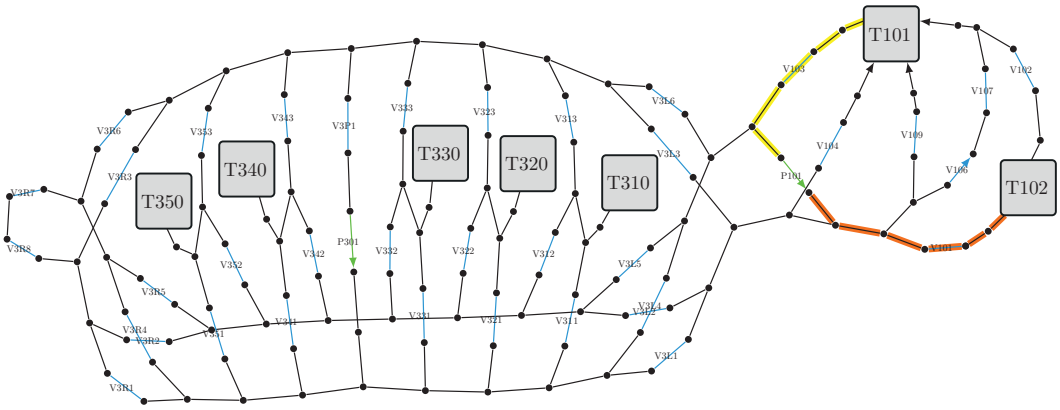
Table 5.1: Components connected to the Beckhoff CX5010 PLCs.

Plant	**Module**	**Specification**	**Attached components**
Festo Didactic	EL2008	DO 24V	valves, pump, mixer, heating element
	EL4004	AO 0 V to 10 V	pump speed
	EL3064	AI 0 V to 10 V	ultrasonic level sensor, temperature sensor, flow sensor
	EL1018	DI 24V	level switches, capacitive proximity sensors
Custom Build	EL2008	DO 24V	valves, pump
	EL4004	AO 0 V to 10 V	pump speed
	EL3064	AI 0 V to 10 V	guided wave radar level sensor, flow sensor

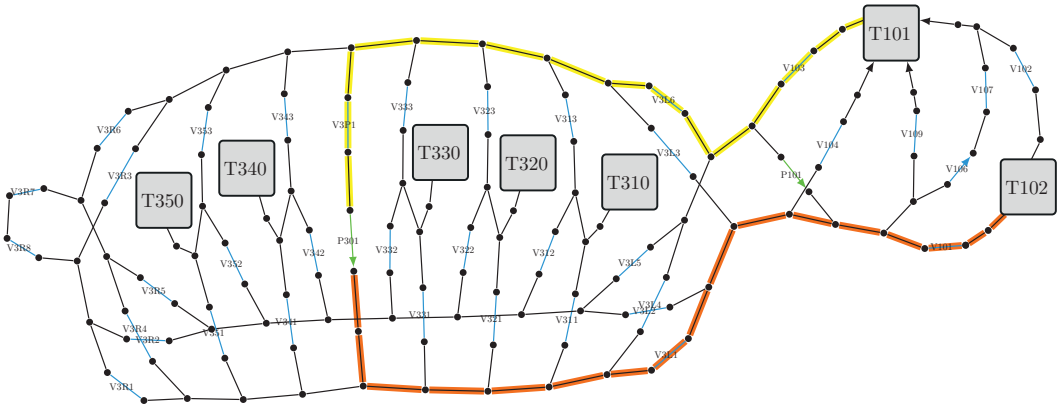
and a mixer integrated. Since the heating element is mounted about ten centimeters above the base of the tank and below the heater, there is a capacitive proximity sensor, which is mounted a few centimeters higher, in order to prevent heating, when there is too little liquid inside the tank. Furthermore, in both subsystem, a unidirectional centrifugal pump and several 2/2 way solenoid valves are integrated. Since the custom built subsystem consists of a redundant pipe system, these valves are required to define a path between two nodes.

5.2 Evaluation of the Route Finding Algorithm

Just before the transfer processes are evaluated, in this section, the evaluation of the shortest path problem is presented. An evaluation example of the route finding algorithm is depicted in Figure 5.2, where the shortest path from tank T101 to T102 is calculated. Since both tanks are mounted on the same level, the algorithm recognizes,



(a) Route 1 with P101: length = 112cm



(b) Route 2 with P301: length = 296cm

Figure 5.2: Route finding example for a transfer process from T101 to T102 using the pumps P101 or P301. The first subpath from the source tank to the pump and the second subpath from the pump to the target tank are highlighted, in yellow and orange, respectively. Since the path of figure (a) is shorter, the algorithm chooses this one.

that a pump is required for the transfer operation. Therefore, the graph is analyzed and all pumps are filtered. Now, the algorithm calculates the shortest path via each pump consecutively. In Figures 5.2a and 5.2b the shortest path and its length is evaluated for pumps P101 and P301, respectively. The comparison of the lengths of both paths yields, that the route via P101 is shorter and therefore this route is chosen for the transfer operation.

5.3 Evaluation of an Add process

When the route is calculated, the corresponding services for the valves, the pump, and the tank level monitoring are requested. The MQTT message correspondence between supervising computer and PLC is depicted at the bottom of Figure 5.3. It shows a zoomed-in view of the starting and end of the transfer process. As the diagram shows, the entire process for turning on all valves, the pump, and the level monitoring takes about half a second. In Section 4.4.2, it was mentioned, that there is a malfunction in the MQTT-Client implementation of FORTE. Due to this problem, a reliable message transfer requires, that the next message is not sent until the last message is confirmed. Basically, MQTT would also allow to send them immediately, which would also increase the turning on/off process, however, since a drop of one or more messages can not be ruled out, the reliable, but more time consuming, method was chosen. Figure 5.3 also shows the sensor values of the level sensor B101 of tank T102, and the flow sensor B102, that is mounted at the output of the pump. After turning off the pump and valves, the flow sensor, that bases on an axial paddle wheel turbine, still measures a flow. This is caused by the sensor principle itself, since, for a moment, the turbine continues turning and therefore the sensor outputs a signal different from zero.

5.4 Evaluation of the Heat and Mix Phase

The Heat Phase was evaluated in combination with the Mix Phase, in order to get a uniform heat distribution inside a tank. Since the recipe creation editor and the recipe processing algorithm allows the execution parallel Phases, this can easily be achieved by specifying the same previous Phase. Each of these two Phases correspond to a single service, which is requested from the PLC. Figure 5.4 shows the progression of the temperature of tank T101 while heating and mixing. The Heat service itself is just implemented with a simple control algorithm, which turns on the heating element at service start and turns it off, when the desired temperature is reached. At the desired target temperature of 35 °C, it is noticeable, that the temperature still rises, although the heater is turned off. This phenomenon is caused by the residual heat of the heating element. In order to minimize the error between desired and actual temperature, for example a PI controller can be implemented, which requires a linear controllable heater, or a pulsing of the control signal.

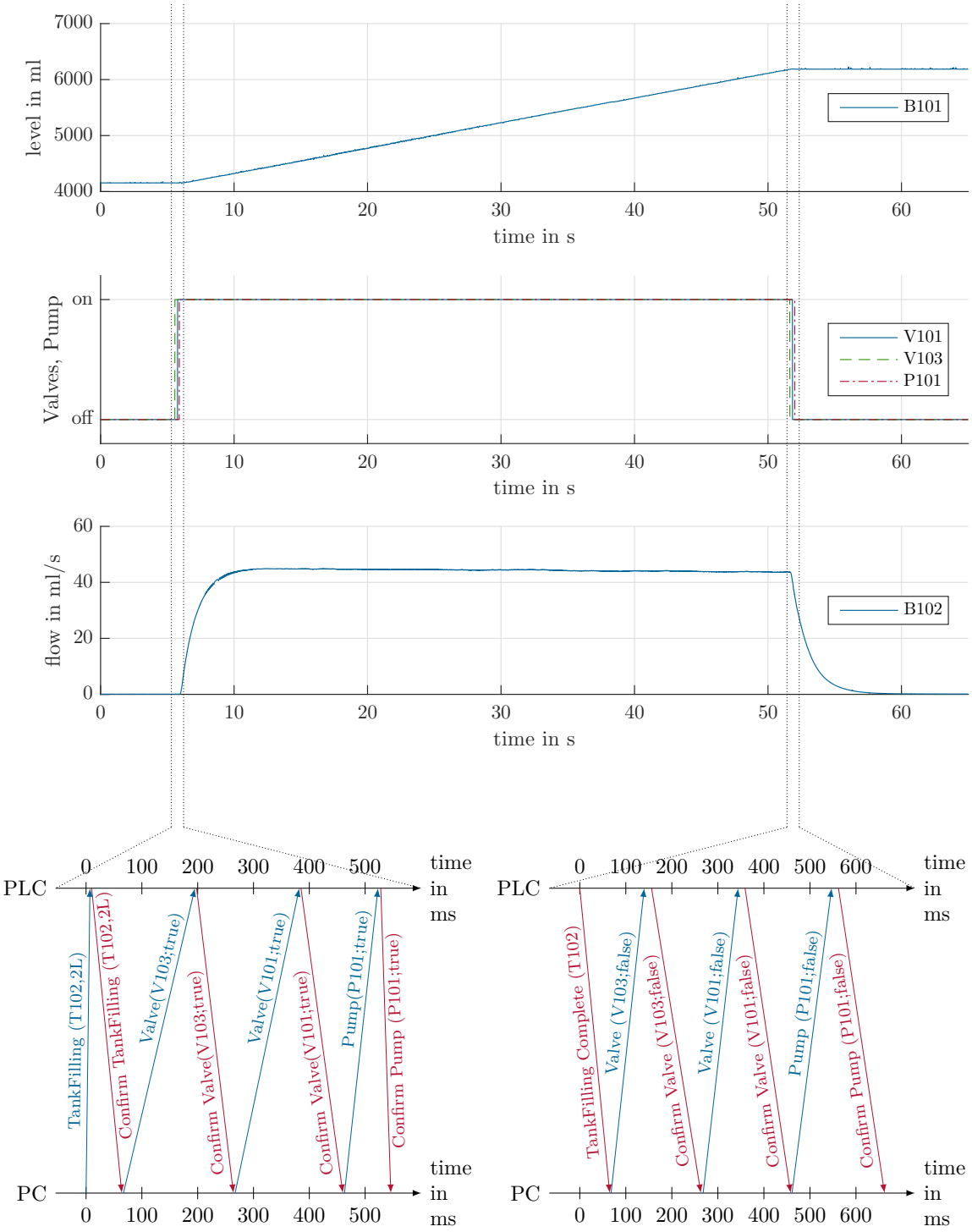


Figure 5.3: Add Process from T101 to T102. The diagrams show the level of tank T102, the state of the required valves and pump, as well as the flow. At the bottom of the figure, the message correspondence between PLC and supervising computer are depicted in a zoomed-in view.

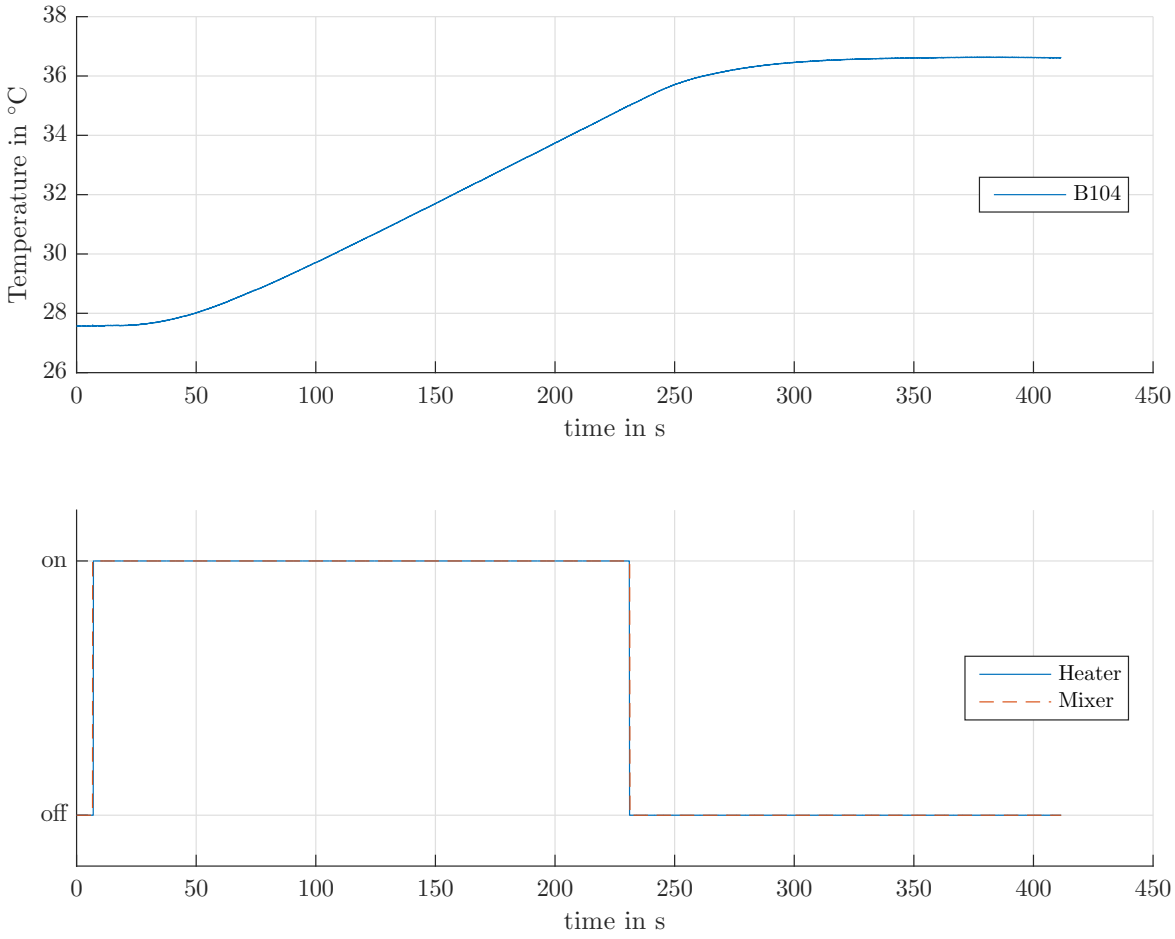


Figure 5.4: Evaluation of a parallel execution of the Heat and Mix Phase. Tank T101 is heated up to 35 °C. However, although the heating element is turned off when the desired temperature is reached, the actual temperature is still rising, which is caused by the residual heat of the heating element.

CHAPTER 6

Conclusion and Outlook

In this thesis, a concept for a flexible batch process control based on a Service-Oriented Architecture (SOA) as well as a domain specific recipe editor for batch processes was developed, implemented, and successfully demonstrated on two linked laboratory tank system plants. The concept enables an easy method for creating and executing production recipes without reconfiguring the actual controlling system, which is usually required in traditional monolithic control systems. The tool for creating production recipes is tailored to the domain of the manufacturing plant, hence, the plant operator with his process specific knowledge is able to handle this tool and create or modify production recipes. Since the controlling system does not change, when the recipe changes, an automation engineer is not required. This reduces a plenty of possible error sources, like, misunderstandings between plant operator and automation engineer, or program inconsistencies due to the intervention into the control program. Furthermore, testing phases can also be reduced, which also minimizes the modification time.

The system design bases on a separation of plant specific implementation details from the actual batch recipes, so that both parts can be developed and modified independently from each other. Therefore, the same batch recipe can be executed on several plants without knowing of how the implementation of the actual processing actions looks like and even where these actions are processed. The concept also allows to distribute these processing actions over several control units, which is very beneficial for larger plants. The independence between the physical equipment and the production procedure is achieved by utilizing services in terms of a SOA. One participant provides a service, which is called by another participant. All services are accessed via public interface. For calling a service, it is not necessary to know any implemen-

tation details.All controlling units, which are usually Programmable Logic Controllers, act as service providers, that offer their controllable actions as services, like switching actuators or reading sensor values.

In addition to the PLCs, there is also a supervising computer, which has several functions. First, in order to create production recipes, it provides an easy to handle editor, which is developed exactly for the domain of the process plant. Thus, process experts of this domain can easily create or modify production recipes. A second task of the supervising computer is the processing of the recipe and service calling from the control system (PLCs). Furthermore, a visualization of the recipe enables the process engineer to determine the current state of the recipe, including already completed and currently active operations.

Research Questions

Since the implementation of this concept for a flexible batch process control was demonstrated on a tank system plant, Research Question 1 asked, if the flexibility of batch productions can be improved by breaking down the typical processing actions, like *Add* or *Heat*, into more atomic services. This question can be answered with *yes*. In tank systems, the *Add* process typically consists of switching valves as well as pumps, and monitor the fluid level. Usually, these single tasks are combined into one operation, where the specific components are pre-defined. However, by breaking them down into single components, that are all ible from outside and with the knowledge of the current plant, a dynamic route finding algorithm can be implemented, which accesses these components separately. By using such an algorithm, the flexibility of batch productions can be improved, since in the case of changes of the plant, the model can easily be adapted and new paths are calculated automatically. In order to be able to implement a dynamic route finding algorithm, the plant was modeled with another editor, where all components of the plant and the piping are described. A major challenge of the route finding algorithm itself was the integration of a pump, which is required for pumping liquids from one tank into another. Typical shortest-path algorithms, like Dijkstra's algorithm, can only find the shortest path from one node to another, without intermediate point. Therefore, the route was split into two subroutes with the pump as intermediate node, in consideration of not using a component twice.

In Research Question 2, it was asked, if this concept is also reasonable for systems with low or no changes in the recipe. This can also be answered with *yes*. Even if the recipe does not change, this concept enables to reduce plant downtimes due to planned or unplanned events, e. g., for maintenance or due to malfunctions, respectively. If, for example, a pump breaks down and there is another pump available in the system, the dynamic route finding algorithm can find an alternative path with a functioning pump without modifying the production recipe or reconfiguring the control program. Furthermore, if similar plants are available, the execution of the entire recipe can also be executed on this plant, without modifying any controlling units.

Outlook

This thesis offers a good basis for a flexible batch process control, however, there are still some issues left for improvements in future projects. If the functionality of by-passing unavailable equipment is required, an interface must be provided to the user, in order to define, which components must not be used. Furthermore, the work can also be extended, by improving the structuring of the recipe based on the standard for batch process control IEC 61512. It defines hierarchical models in order to structure the physical equipment and the production information as described in the recipe. Concerning the recipe editor, this hierarchy could be integrated, which would be beneficial for larger projects.

Bibliography

[1] T. Bauernhansl, M. Ten Hompel, and B. Vogel-Heuser, *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung - Technologien - Migration*. Springer-Verlag, 2014.

[2] Y. Koren, *The Global Manufacturing Revolution: Product-Process-Business Integration and Reconfigurable Systems*. John Wiley & Sons, 2010.

[3] S. Y. Nof, „Automation: What It Means to Us Around the World“, in *Springer Handbook of Automation*, Springer, 2009, pp. 13–52.

[4] C. Bissell, „A History of Automatic Control“, in *Springer Handbook of Automation*, Springer, 2009, pp. 53–69.

[5] *IEC 61512-1 Batch Control - Part 1: Models and Terminology*, International Electrotechnical Commission (IEC), 1997.

[6] W. Hawkins, D. Brandl, and W. Boyes, *Applying ISA-88 in Discrete and Continuous Manufacturing*. Momentum Press, 2011, vol. 2.

[7] K. Thramboulidis, „Model Integrated Mechatronics: An Architecture for the Model Driven Development of Manufacturing Systems“, in *ICM '04. Proceedings of the IEEE International Conference on Mechatronics*, Jun. 2004, pp. 497–502.

[8] M. Wenger, M. Melik-Merkumians, I. Hegny, R. Hametner, and A. Zoitl, „Utilizing IEC 61499 in an MDA Control Application Development Approach“, in *IEEE Conference on Automation Science and Engineering (CASE)*, Aug. 2011, pp. 495–500.

[9] J. Browne, D. Dubois, K. Rathmill, S. P. Sethi, and K. E. Stecke, „Classification of Flexible Manufacturing Systems“, *The FMS magazine*, vol. 2, no. 2, pp. 114–117, 1984.

[10] V. Botti and A. Giret, *ANEMONA: A Mulit-agent Methodology for Holonic Manufacturing Systems*, D. Pham, Ed. Springer London, 2008.

[11] Bundesministerium für Bildung und Forschung, Deutschland. (2012). Zukunftsprojekte der Hightech-Strategie, [Online]. Available: `https://www.bmbf.de/pub/HTS-Aktionsplan.pdf` (visited on 07/13/2016).

[12] H. Kagermann, W. Wahlster, and J. Helbig, *Recommendations for implementing the strategic initiative INDUSTRIE 4.0*, 2013.

[13] M. Barker and J. Rawtani, *Practical Batch Process Management*. Elsevier, 2005.

[14] D. Brandl, *Design Patterns for Flexible Manufacturing*. ISA, 2006.

[15] J. Parshall and L. Lamb, *Applying S88: Batch Control from a User's Perspective*. ISA, 1999.

[16] *IEC 61512-2 Batch Control Part 2: Data Structures and Guidelines for Languages*, International Electrotechnical Commission (IEC), 2001.

[17] M. De Minicis, F. Giordano, F. Poli, and M. M. Schiraldi, „Recipe Development Process Re-Design with ANSI/ISA-88 Batch Control Standard in the Pharmaceutical Industry“, *International Journal of Engineering Business Management*, vol. 6, 2014.

[18] *IEC 62264-1: Enterprise-Control System Integration Part 1: Models and Terminology*, International Electrotechnical Commission (IEC), 2003.

[19] C. L. Case, „Applying ISA 88.01 to Small, Simple Processes“, in *The WBF BOOK SERIES–ISA 88 Implementation Experiences*, W. Hawkins, WBF, D. Brandl, and W. Boyes, Eds., Momentum Press, 2010, ch. 3, pp. 21–33.

[20] G. Godena, I. Steiner, J. Tancek, and M. Svetina, „Design of a Batch Process Control Tool on the Programmable Logic Controller Platform“, in *The WBF BOOK SERIES–ISA 88 Implementation Experiences*, ser. WBF book series, W. Hawkins, WBF, D. Brandl, and W. Boyes, Eds., Momentum Press, 2010, ch. 14, pp. 157–173.

[21] N. M. Josuttis, *SOA in Practice*, 1. ed. O'Reilly Media, Inc., 2007.

[22] P. Bianco, R. Kotermanski, and P. Merson, „Evaluating a Service-Oriented Architecture“, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Tech. Rep. CMU/SEI-2007-TR-015, 2007.

[23] C. A. R. Hoare, „An Axiomatic Basis for Computer Programming“, *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[24] M. Melik-Merkumians, T. Baier, M. Steinegger, W. Lepuschitz, I. Hegny, and A. Zoitl, „Towards OPC UA as portable SOA Middleware between Control Software and External Added Value Applications“, in *IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2012, pp. 1–8.

[25] W. Dai, J. Peltola, V. Vyatkin, and C. Pang, „Service-Oriented Distributed Control Software Design for Process Automation Systems“, in *IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Oct. 2014, pp. 3637–3642.

[26] H. Bohn, A. Bobek, and F. Golatowski, „SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains“, in *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*, Apr. 2006, pp. 43–43.

[27] A. Cannata, M. Gerosa, and M. Taisch, „SOCRADES: A Framework for Developing Intelligent Systems in Manufacturing“, in *2008 IEEE International Conference on Industrial Engineering and Engineering Management*, Dec. 2008, pp. 1904–1908.

[28] S. Karnouskos, A. W. Colombo, F. Jammes, J. Delsing, and T. Bangemann, „Towards an Architecture for Service-Oriented Process Monitoring and Control“, in *36th Annual Conference on IEEE Industrial Electronics Society (IECON)*, 2010, pp. 1385–1391.

[29] J. Virta, I. Seilonen, A. Tuomi, and K. Koskinen, „SOA-Based Integration for Batch Process Management with OPC UA and ISA-88/95“, in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2010, pp. 1–8.

[30] A.-W. Colombo, S. Karnouskos, and J.-M. Mendes, „Artificial Intelligence Techniques for Networked Manufacturing Enterprises Management“, in, L. Benyoucef and B. Grabot, Eds. London: Springer London, 2010, ch. Factory of the Future: A Service-oriented System of Modular, Dynamic Reconfigurable and Collaborative Systems, pp. 459–481.

[31] S. Karnouskos, A. W. Colombo, T. Bangemann, K. Manninen, R. Camp, M. Tilly, P. Stluka, F. Jammes, J. Delsing, and J. Eliasson, „A SOA-based architecture for empowering future collaborative cloud-based industrial automation“, in *38th Annual Conference on IEEE Industrial Electronics Society (IECON)*, IEEE, 2012, pp. 5766–5772.

[32] G. Cândido, F. Jammes, J. B. de Oliveira, and A. W. Colombo, „SOA at Device level in the Industrial domain: Assessment of OPC UA and DPWS specifications“, in *8th IEEE International Conference on Industrial Informatics (INDIN)*, Jul. 2010, pp. 598–603.

[33] E. Curry, „Message-Oriented Middleware“, in *Middleware for Communications*, John Wiley & Sons, Ltd, 2004, pp. 1–28.

[34] D. R. Ferreira, „Messaging Systems“, in *Enterprise Systems Integration*, Springer Berlin Heidelberg, 2013, pp. 32–92.

[35] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[37] G. Banavar, T. Chandra, R. Strom, and D. Sturman, „Distributed Computing: 13th International Symposium (DISC'99)“, in, P. Jayanti, Ed. Springer Berlin Heidelberg, 1999, ch. A Case for Message Oriented Middleware, pp. 1–17.

[38] *IEC 62541: OPC Unified Architecture*, International Electrotechnical Commission (IEC), 2010.

[39] J. Imtiaz and J. Jasperneite, „Scalability of OPC-UA Down to the Chip Level Enables “Internet of Things”“, in *11th IEEE International Conference on Industrial Informatics (INDIN)*, Jul. 2013, pp. 500–505.

[40] W. Mahnke and S.-H. Leitner, „OPC Unified Architecture“, in. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ch. Services, pp. 125–190.

[41] L. Dürkop, B. Czybik, and J. Jasperneite, „Performance Evaluation of M2M Protocols Over Cellular Networks in a Lab Environment“, in *18th International Conference on Intelligence in Next Generation Networks (ICIN)*, Feb. 2015, pp. 70–75.

[42] OPC Foundation. (2016). OPC UA is Enhanced for Publish-Subscribe, [Online]. Available: `https://opcfoundation.org/opc-connect/2016/03/opc-ua-is-enhanced-for-publish-subscribe-pubsub/` (visited on 04/11/2016).

[43] F. Jammes, A. Mensch, and H. Smit, „Service-Oriented Device Communications using the Devices Profile for Web Services“, in *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC)*, ACM, 2005, pp. 1–8.

[44] F. Palm, S. Grüner, J. Pfrommer, M. Graube, and L. Urbas, „Open Source as Enabler for OPC UA in Industrial Automation“, in *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, Sep. 2015, pp. 1–6.

[45] V. Lampkin, W. Leong, L. Olivera, S. Rawat, N. Subrahmanyam, R. Xiang, G. Kallas, N. Krishna, S. Fassmann, M. Keen, *et al.*, *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*, ser. IBM redbooks. IBM Redbooks, 2012.

[46] P. Hintjens, *ZeroMQ: Messaging for Many Applications*, A. Oram and M. Gulick, Eds. O’Reilly Media, Inc., 2013.

[47] iMatix Corporation. (2014). ZeroMQ - The Guide, [Online]. Available: `http://zguide.zeromq.org/` (visited on 04/15/2016).

[48] Object Management Group. (Apr. 2015). Data Distribution Service. Version 1.4, [Online]. Available: `http://www.omg.org/spec/DDS/1.4` (visited on 07/12/2016).

[49] S. Schneider and B. Farabaugh, „Is DDS for You?“, *A Whitepaper by Real-Time Innovations*, 2009.

[50] A. Møller and M. I. Schwartzbach, „10th International Conference on Database Theory (ICDT)“, in, T. Eiter and L. Libkin, Eds. Springer Berlin Heidelberg, 2005, ch. The Design Space of Type Checkers for XML Transformation Languages, pp. 17–36.

[51] S. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach*, 3rd revised edition. Prentice Hall, 2010.

[52] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[53] D. Ghosh, *DSLs in Action*. Manning Publications Co., 2010.

[54] M. Fowler, „Language Workbenches: The Killer-App for Domain Specific Languages?“, 2005.

[55] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[56] T. Parr, *The Definitive ANTLR 4 Reference*, S. Pfalzer, Ed. Pragmatic Bookshelf, 2013.

[57] M. Fowler, *Domain-Specific Languages*, ser. Addison-Wesley Signature Series. Pearson Education, 2010.

[58] J. W. Backus, F. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, *et al.*, „Revised Report on the Algorithmic Language ALGOL 60“, *The Computer Journal*, vol. 5, no. 4, pp. 349–367, 1963.

[59] *ISO/IEC 14977: Information technology - Syntactic metalanguage - Extended BNF*, International Organization for Standardization, 1996.

[60] Eclipse Foundation. (2016). Xtext - Framework for Development of Programming Languages and Domain-Specific Languages, [Online]. Available: `http://www.eclipse.org/Xtext/` (visited on 06/06/2016).

[61] *IEC 61131-3: Programmable controllers - Part 3: Programming languages*, International Electrotechnical Commission (IEC), 2013.

[62] B. Vogel-Heuser, „Springer Handbook of Automation“, in, Y. S. Nof, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ch. Automation in the Wood and Paper Industry, pp. 1015–1026.

[63] *IEC 61499: Function Blocks*, International Electrotechnical Commission (IEC), 2012-2013.

[64] A. Zoitl and R. W. Lewis, *Modelling Control Systems Using IEC 61499*, 2. ed., ser. IET Control engineering series ; 95. London: IET, 2014.

[65] *IEC 61131: Programmable controllers*, International Electrotechnical Commission (IEC), 2001-2014.

[66] W. Lepuschitz and A. Zoitl, „An Engineering Method for Batch Process Automation using a Component Oriented Design based on IEC 61499“, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2008, pp. 207–214.

[67] H. Prähofer, D. Hurnaus, C. Wirth, and H. Mössenböck, „The Domain-Specific Language Monaco and its Visual Interactive Programming Environment“, in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2007, pp. 104–110.

[68] T. Gu and P. A. Bahri, „A survey of Petri net applications in batch processes“, *Computers in Industry*, vol. 47, no. 1, pp. 99–111, 2002.

[69] B. Favre-Bulle, *Automatisierung komplexer Industrieprozesse: Systeme, Verfahren und Informationsmanagement*. Springer-Verlag, 2013.

[70] G. T. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell*. O'Reilly Media, Inc., 2008.

[71] E. W. Dijkstra, „A Note on Two Problems in Connexion with Graphs“, *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[72] P. E. Hart, N. J. Nilsson, and B. Raphael, „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“, *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[73] J. Bang-Jensen and G. Z. Gutin, *Digraphs: Theory, Algorithms and Applications*, 2nd. Springer Publishing Company, Incorporated, 2008.

[74] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd Edition. MIT Press, 2009.

[75] Eclipse Foundation. (2016). 4diac - Framework for Industrial Automation & Control, [Online]. Available: `http://www.eclipse.org/4diac` (visited on 06/17/2016).

[76] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck, „Taming EMF and GMF Using Model Transformation“, in *Model Driven Engineering Languages and Systems (MODELS): 13th International Conference, Part I*, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., Springer Berlin Heidelberg, 2010, pp. 211–225.

[77] Eclipse Foundation. (). Mosquitto - Open-Source MQTT Message Broker, [Online]. Available: `http://mosquitto.org/` (visited on 06/23/2016).

[78] ——, (2016). Xtend - Programming Language, [Online]. Available: `http://www.eclipse.org/xtend/` (visited on 06/21/2016).

[79] K. Birken, „Building Code Generators for DSLs Using a Partial Evaluator for the Xtend Language“, in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014*, T. Margaria and B. Steffen, Eds., Springer Berlin Heidelberg, 2014, pp. 407–424.

[80] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné, „GraphStream: A Tool for bridging the gap between Complex Systems and Dynamic Graphs“, in *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS)*, Oct. 2007.

[81] O. Dubuisson, *ASN. 1: Communication between heterogeneous systems*. Morgan Kaufmann, 2001.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, im August 2016 Matthias Baierling