Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Meccanica
A.a. 2022/2023
Sessione di Laurea Dicembre 2023

Development of a Concept for the App-Based Control of a Production Plant in Battery Cell Production

Relatori:
Prof. Luigi Mazza
Prof. Dr.-Ing. Jürgen Fleischer (KIT)
Dott. Imanuel Heider (KIT)

Candidato:
Andrea Albertini s292805

Statement of Originality

I sincerely affirm to have composed this thesis work autonomously, to have indicated completely and accurately all aids and sources used and to have marked anything taken from other works, with or without changes. Furthermore, I affirm to have observed the constitution of the KIT for the safeguarding of good scientific practice, as amended.

Karlsruhe, November 2nd 2023

[illegible]

Acknowledgement

I want to give a big shout-out to the wbk Department at the Karlsruhe Institute of Technology for this amazing opportunity. I especially want to thank my supervisor, M.Sc. Imanuel Heider. He held my hand through this whole journey and helped me grow, both personally and professionally. Working with someone as smart and capable as Imanuel, as well as the other Process Engineers at the Department, was essential for getting the job done. I couldn't have done it without their support. A big thank also goes to Luca Matschinski, your help during this experience has been unquestionably helpful for me to achieve this outcome. During my experience in Germany I've had many problems also outside this project and you always managed to get me through them.

Outside the university environment, my first though goes to Leti. I perfectly know that I'm absolutely not the easiest person to be around, mostly when things get stressful, but you always managed to make everything seems lighter and calmer. Without you everything would have been a lot harder.

Without my parents, on the other hand, it wouldn't have been possible for me to have this experience abroad. They've been supporting me since day one. I know that probably it has been harder for you than for me to live away from home for so long, but I also know that you are proud of the man I have become and I am proud of the parents I have.

At last but not least, I've always been surrounded by many friends, both here and in Germany. Cecco, Mattia, Umbe, Marco, Giulia, Enri, Lili and all the others - I can't mention everyone, but I want you all to know how much I love you.

Thank you all.

Albert

Voglio ringraziare di cuore il Dipartimento wbk del Karlsruhe Institute of Technology per questa straordinaria opportunità. Voglio ringraziare in particolare il mio relatore, il Dottor Imanuel Heider. Mi ha tenuto per mano durante tutto questo percorso e mi ha aiutato a crescere, sia personalmente che professionalmente. Lavorare con qualcuno intelligente e capace come Imanuel, così come con gli altri ingegneri di processo del dipartimento, è stato essenziale per portare a termine il lavoro. Non avrei potuto farcela senza il loro supporto. Un grande ringraziamento va anche a Luca Matschinski, il vostro aiuto durante questa esperienza mi è stato indiscutibilmente utile per raggiungere questo risultato. Durante la mia esperienza in Germania ho avuto molti problemi anche al di fuori di questo progetto e tu mi sei sempre stato di fianco.

Fuori dall'ambiente universitario il mio primo pensiero va a Leti. So perfettamente di non essere assolutamente la persona più facile da sopportare, soprattutto quando le cose si fanno stressanti, ma tu sei sempre riuscita a far sembrare tutto più leggero e tranquillo. Senza di te tutto sarebbe stato molto più difficile.

Senza i miei genitori, invece, non mi sarebbe stato possibile fare questa esperienza in Germania. Mi sostengono dal primo giorno. So che probabilmente è stato più difficile per loro che per me non avermi in casa per così tanto tempo, ma so anche che sono orgogliosi dell'uomo che sono diventato e io sono orgoglioso dei genitori che ho.

Infine, ma non per importanza, sono sempre stato circondato da tanti amici, sia qui che in Germania. Cecco, Mattia, Umbe, Marco, Giulia, Enri, Lili e tutti gli altri - non posso nominare tutti, ma voglio che sappiate quanto vi voglio bene.

Grazie a tutti.

Albert

Abstract

This thesis delves into enhancing agility in battery cell manufacturing, a pivotal aspect of industries like electric vehicles and energy storage systems. It underscores the imperative of swift adaptation to shifting production requirements.

The study explores key technologies including battery cell production, OPC UA communication, Siemens TIA Portal, and the application of ISA-88 standards. By integrating principles of change perception, responsiveness, and adaptability, it aligns production processes with the ever-changing market landscape.

A significant focus is the development of Graphical User Interfaces. This interface provides users with a dynamic view of the production processes. Buttons associated with sub-processes change color to signify different operational states, aiding operators in monitoring and controlling the manufacturing operations.

This thesis stresses the critical role of agility in modern manufacturing and offers a pathway for further refining the control system to cater to various subsystems.

Table of Contents

Table of Abbreviations

Symbol	**Measurement**
LIB	Lithium-Ions-Battery
SSB	Solid-State-Battery
OPC	Open Platform Communications
OPC UA	OPC Unified Architecture
AGV	Automated Guided Vehicle
GUI	Graphical User Interface
HMI	Human Machine Interface

1 Introduction

1.1 Motivation

The world of the electric mobility represents a dynamic market, vital for the decarbonization in road transport, one of the most important sectors regarding urban pollution and toxic emissions. The transport of goods and people accounts for about 20% of the total global primary energy consumed, around 23% of CO_2 emissions and if other greenhouse gases (GHG) such as methane are taken into account, around 14% of the total global GHG emissions (*Is It Really the End of Internal Combustion Engines and Petroleum in Transport?*, 2023). There could be 6–15% improvements in internal combustion fuel efficiency in the coming decade, although filters to meet emission legislation reduce these gains. Using these engines as hybrids with electric motors produces a reduction in energy requirements in the order of 21–28%. (*Science Review of Internal Combustion Engines*, 2023).

Electric mobility denotes, anyway, a key resource for energy transition as well as a steadily growing market which, in recent years, has also undergone a major improvement in efficiency. The utmost development has been seen in China, while, on the other hand, this technology finds complications to spread in some European countries and U.S. Increasing sales pushed the total number of electric cars on the world's roads to 26 million, up 60% relative to 2021, with BEVs (*Battery Electric* Vehicle) accounting for over 70% of total annual growth, as in previous years. As a result, about 70% of the global stock of electric cars in 2022 were BEVs. (IEA, 2023).

In *Figure 1* we can clearly see the growth in electric car sales in recent years. With orange color is represented car sales in China, with blue color Europe and then U.S with light green and other countries with the dark green.

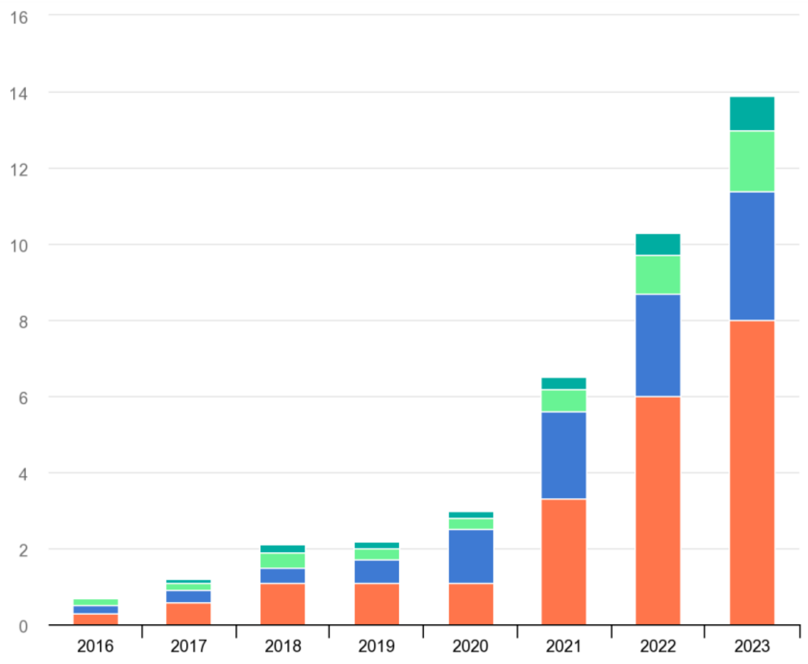


Figure 1 Electric car sales, 2016-2023 (IEA, 2023)

In the face of this increasing market demand the importance of improving efficiencies inside the production plan and becoming more competitive in terms of manufacturing is now clearer than ever. A countless variety of solutions to this problem have been introduced by the Fourth Industrial Revolution. In essence, several new innovations and new technologies have been introduced towards automation and data exchange in manufacturing technologies and processes which include cyber-physical systems (CPS), IoT, industrial internet of things, cloud computing, cognitive computing, and artificial intelligence (*Fourth Industrial Revolution - Wikipedia*, 2023). At present, world is experiencing the revolution created by the Internet and the way it can impact all the industries and common people. The wealth of information and ability to connect multiple things together changes the way industries operate and yield efficiencies.

All these new features have been introduced over the years with the aim of helping to produce goods efficiently and productively across the value chain. Flexibility and data exchange are improved in order to obtain a new concept of manufacturing that can achieve information transparency and better decision. (*What Is Industry 4.0 and How Does It Work? | IBM*, 2023). In today’s fast-moving market a new factory idea was sought with a production method that could draw attention to a quick response in terms of flexibility. In this scenario the concept of Agile Manufacturing has taken hold which can acknowledge the realities of the modern marketplace and transforms them into a competitive advantage.

1.2 Objective

Agile control should emphasize three pivotal characteristics: change perception, responsiveness, and adaptability (Simon Gese, 2021). The initial crucial facet is *change perception*. Alterations in the production process primarily result from decisions made at the business level. Each production process entails a recipe comprising the sequence of operations and their associated parameters. Therefore, any change in production signifies a modification in the recipe. Crafting a recipe necessitates the consideration of technical execution possibilities, which should align with the system. An agile system must be proficient in executing various such recipes. To facilitate perception within the cell, a state system detailing process status will be devised as part of this work.

The second essential requirement is *responsiveness*, signifying the swift implementation of new demands within a defined timeframe to maintain system stability. It is imperative that production segments integrate seamlessly into existing systems, with control systems for new sub-components designed for effortless adaptability.

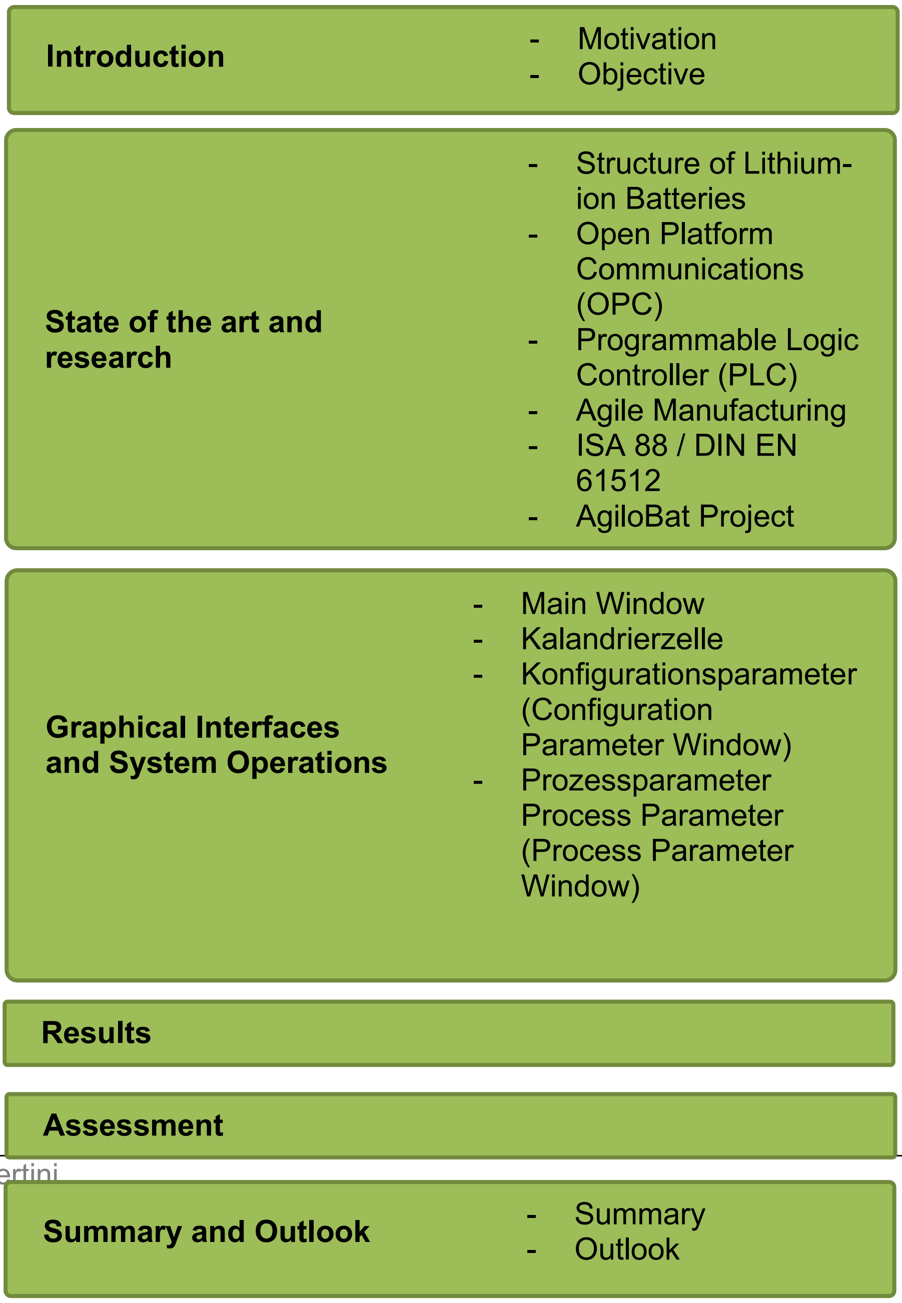
The third key requirement is *adaptability*, encapsulating the need for a system to possess both flexibility and adaptability. This encompasses system expansion and product adaptation. Variables such as quantities and geometry requirements may fluctuate from one recipe to another. Given the current considerable effort required to transition a production line from one battery cell format, as in our case, to another (VDMA, 2023, p. 15), it aligns with sound business strategy to inherently incorporate agile concepts from the outset.

The main purpose of the work was to develop and implement an application suitable for the control and monitoring of an agile manufacturing process designed to produce battery cell. It consists of four different automated microenvironments, one for each main step of a defined process chain. The goal and this Thesis’ intent are to create a system able not only to allow the user to start and stop the production plant, but also to decide to proceed with a process in particular rather than another one. There’s the need to provide the operator with extreme freedom of choice and give him the ability to run only certain processes.

Furthermore, it will also be possible to decide whether to start from the standard parameters stored in a Database or modify them in order to customize the production procedure and, with a specific setting, the user will also be able to enter data such as the URL of the Database HTTP Endpoint and the access to the four distinct Open Platform Communications (OPC) Endpoints corresponding to the four Programmable Logic Controllers (PLCs), each of them associated with a distinct working cell.

1.3 Structure of the thesis

In the following sections, this work provides an overview of the current state of research on the key technologies it encompasses. These technologies include battery cells, OPC UA, and the Siemens TIA Portal. Additionally, it introduces the ISA-88 standard and its most pertinent aspects. The context for this discussion is set within the AgiloBat project, which is briefly introduced.


Introduction
- Motivation
- Objective
State of the art and research
- Structure of Lithium-ion Batteries
- Open Platform Communications (OPC)
- Programmable Logic Controller (PLC)
- Agile Manufacturing
- ISA 88 / DIN EN 61512
- AgiloBat Project
Graphical Interfaces and System Operations
- Main Window
- Kalandrierzelle
- Konfigurationsparameter (Configuration Parameter Window)
- Prozessparameter Process Parameter (Process Parameter Window)
Results
Assessment
Summary and Outlook
- Summary
- Outlook

2 State of the art and research

2.1 Structure of Lithium-ion Batteries

Batteries are utilized as chemical energy storage system and currently the most popular ones are lead-acid batteries components (Noack et al., 2015). This technology is known for its reliability and long-lasting attitude in contrast to other devices such as nickel–cadmium and nickel–metal hydride batteries which have higher energy densities and higher numbers of cycles, but also carry higher costs. Redox-flow batteries are appliances for the storage of electrochemical energy, in which the redox-active constituents are flowing media and the redox reactions take place in an energy converter similar to a fuel cell. To describe how a battery cell work we should start by listing the three main components (Noack et al., 2015, p. 9777):

- **Electrodes**: they should possess a high electrochemical stability, high reaction kinetics of the redox couple and, at the same time, high electrical conductivity and mechanical stability at low cost. Pre-treatment on them can improve kinetics of reaction and their surface quality. The positive electrode is always referred to as the cathode and the negative one is referred to as the anode.
- **Separating membrane**: responsible for dividing the cell into two halves, vital to prevent the mixing between the two electrolyte solutions which could cause an uncontrolled reaction. There are different types depending on the material from which they are made and their purpose
- **Electrolyte**: a conductive substance that is located between the electrodes. It allows ions to flow between the electrodes. There are liquid and solid electrolytes.

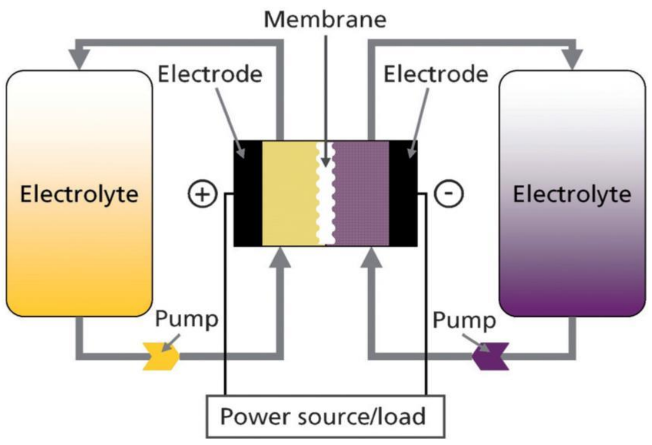


Figure 2: Redox-flow batteries with electrolytes as the media for energy storage. (Noack et al., 2015, p. 9779)

2.1.1 Battery Chemistry

The working principle of redox-flow batteries in the discharge mode can be generally represented as a chemical reaction of two redox couples that results from the combination of two corresponding half-reactions (Noack et al., 2015, p. 9782). While the battery is discharging and providing an electric current, the anode releases lithium ions to the cathode, producing a stream of electrons from one side to the other. When plugging in the device, the reverse happens: Lithium ions are released by the cathode and received by the anode. (Energy.gov, 2023). This kind of batteries require nonaqueous electrolytes since lithium metal reacts spontaneously with water, due to its weak metallic bonding, favorable formation of Li+(aq), favorable bond formation in H2, and highly favorable formation of solvated OH−. The net reaction in the lithium ion battery is (Schmidt-Rohr, 2018, p. 1807):

$$LiC_6 + CoO_2 \rightarrow C_6 + LiCoO_2$$

2.1.2 Manufacturing

The manufacturing of battery cells is a highly intricate procedure composed of multiple sequential stages aimed at establishing the foundation for high-performance and safe batteries. The initial phase involves anode mixing, wherein anode materials are blended together. Subsequently, these anodes are applied to substrates and undergo a drying process to ensure a consistent anode coating. The coated anodes are then subjected to calendaring and singulation, followed by vacuum drying to eliminate moisture and enhance their quality. Concurrently, the cathode mixing process is executed, succeeded by cathode coating, drying, calendaring, and separation steps. Similar to anodes, cathodes also experience vacuum drying to enhance their quality.

The actual battery cell is formed by either winding or stacking the anode and cathode electrodes. This cell is then enclosed within a casing, and electrical connections are established. Electrolyte is introduced into the cell to facilitate ionic conductivity. Throughout the manufacturing process, various other stages such as wetting, forming, degassing, maturation, and End-of-Line (EoL) testing are conducted to ensure both performance and safety.

Moreover, separator films are manufactured to segregate the anode and cathode within the cell. Multiple cells are assembled into modules to attain the required voltage and capacity, and eventually, these modules are combined to create battery packs.

Battery cell production mandates meticulous control, stringent quality assurance, and rigorous safety measures to yield batteries that align with the demands of applications like electric vehicles and energy storage systems. The optimization of this process is of paramount importance in the development of more potent and sustainable battery technologies, addressing the evolving requirements of the contemporary world.

2.1.3 Battery types

2.1.3.1 Lithium-Ion-Battery (LIB)

Lithium-ion batteries (LIBs) stand as the prevailing choice in the commercial battery cell arena. By 2022, the global LIB market was projected to reach a staggering 700 GWh or even higher (VDMA, p. 11). In the preceding year, 2021, the worldwide demand for LIB cells ranged between 460-500 GWh. Electromobility accounted for over 350 GWh of this demand, while stationary applications contributed approximately 50 GWh (VDMA, p. 11).

LIBs encompass a broad category of batteries, all united by the presence of lithium compounds within the cathode. These lithium compounds empower LIBs to achieve remarkable energy densities, earning them the moniker of high-energy batteries. A diverse array of electrolytes can be found within LIBs, including liquid and solid electrolytes, along with polymer electrolytes. Among these, liquid electrolytes find the widest application. However, polymer electrolytes, often paired with lithium metal anodes, tend to exhibit lower conductivity. Solid electrolyte batteries, theoretically capable of delivering higher energy densities, face practical limitations due to interfacial resistance at normal temperatures, making their commercial utilization less common (Lydia Dorrmann et al., p. 4). Nevertheless, comprehensive battery research underscores that the potential of established large-format lithium-ion batteries is far from realization (VDMA, p. 9).

2.1.3.2 Solid State Battery (SSB)

Unlike conventional batteries, solid-state batteries (SSBs) operate without a liquid electrolyte; instead, they employ a solid electrolyte, promising significant enhancements in various critical performance parameters (Fraunhofer ISI, 2022, p. 13). There exist several types of solid electrolytes (SEs) applicable to SSBs, such as oxide RE, sulfide RE, and polymer RE. Each of these solid electrolyte types comes with distinct advantages and drawbacks, although a detailed exploration of these specifics is beyond the scope of this discussion.

In general, SSBs are expected to exhibit long-term stability and service life comparable to, or slightly better than, their liquid electrolyte lithium-ion battery (LIB) counterparts. Currently, SSBs are predominantly in the research and development phase. However, polymer SSBs are already finding application in electric buses, which presently represent the largest market segment for SSBs (Frauenhofer ISI, 2022, p. 14). Potential future applications for SSBs include the automotive and heavy industries, along with sectors that require robust performance in demanding environments (Frauenhofer ISI, 2022, p. 14). Despite their promising attributes, the current market share of SSBs is relatively small, accounting for less than 0.5% of the market share, amounting to 2 GWh. Projections by the Frauenhofer Institute ISI suggest that this proportion could potentially reach 1% by the year 2035 (Frauenhofer ISI, 2022, p. 14).

2.1.4 Cell geometry

Battery cells come in diverse shapes and sizes, and their designs are primarily driven by the need to efficiently fit within available installation space and meet the specific demands of various battery systems (VDMA, p. 34). Three common geometries have been established, each offering distinct attributes:

- *Pouch cells* are characterized by a flexible pouch, typically constructed from a plastic-aluminum composite film. Their lightweight and thin shell contribute to enhanced gravimetric energy density when compared to prismatic cells, which have a thicker shell. Pouch cells excel in heat dissipation, facilitated by current conductors and the cell's sides, giving them the best cooling performance among the three geometries (VDMA, p. 33).
- *Cylindrical cells*, on the other hand, are typically created through winding processes and feature a sturdy, thick shell. These cells boast a hard shell, offering a high energy density and exceptional rigidity. However, their heat dissipation capabilities are comparatively less efficient.
- *Prismatic cells* share the same hard shell attribute but have an intermediate gravimetric energy density. This is due to a substantial portion of their weight being attributed to the shell. Prismatic cells are known for their high rigidity and efficient temperature regulation, owing to their favorable surface-to-volume ratio.

In essence, the various cell geometries are customized to meet specific requirements and applications, each offering a unique blend of advantages and disadvantages related to weight, rigidity, energy density, and heat dissipation.

2.2 Open Platform Communications (OPC)

Industries and businesses rely on products from diverse sellers for automation. In earlier times, these vendors employed distinct communication methods, making challenges in communicating among control system components. As a result, data sharing became complicated, contributing to elevated expenses for users.

In this environment, systems integration requires an enormous effort, especially for large-scale infrastructures. In general, these facilities are complex, vast networked systems that comprise a vast number of devices and applications with different communication protocols. Therefore, data acquisition, exchange, and processing are achieved in a distributed way between heterogeneous data sources and consumers. Cyber–physical systems and IoT are represented by platforms that are integrated through connectivity protocols that permit a wide sharing of information among different devices (González et al., 2019).

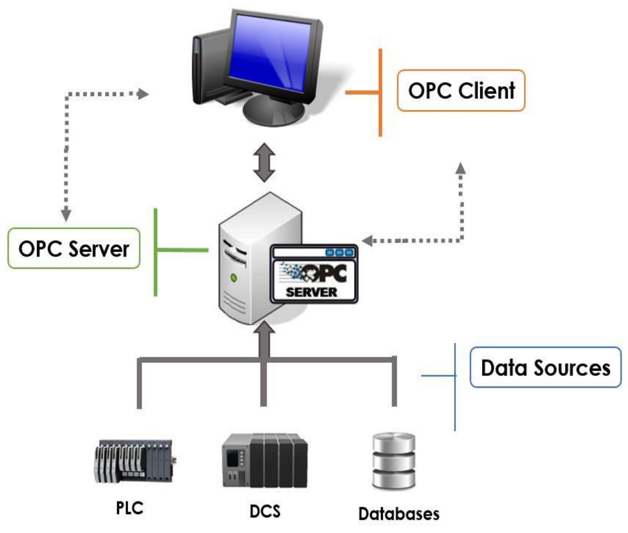


Figure 3: OPC (OPC Blog, 2018)

OPC serves as the universal standard for secure and dependable data exchange, not only within industrial automation but also across diverse industries. This platform-independent standard facilitates the seamless sharing of information among devices manufactured by different companies. The OPC Foundation is responsible for the development and upkeep of this standard. Initially introduced in 1996, its primary purpose was to abstract PLC-specific protocols (such as Modbus, Profibus, and others) into a standardized interface. This interface empowered HMI/SCADA systems to communicate with an intermediary component, which acted as a translator, converting generic OPC read/write requests into device-specific ones and vice versa. Consequently, this gave rise to an entire industry of products that allowed end-users to construct systems harnessing the best available products, all seamlessly interacting via OPC. (OPC Foundation, 2017).

At present, the OPC standard encompasses ten distinct specifications under the stewardship of the OPC Foundation. These specifications are meticulously designed and maintained to serve various essential functions. They include Data Access (DA), Historical Data Access (HDA), Alarms and Events (A&E), XML-Data Access (XML-DA), Data Exchange (DX), Complex Data (CD), Security, Batch, Express Interface (Xi), and Unified Architecture (UA). (González et al., 2019).

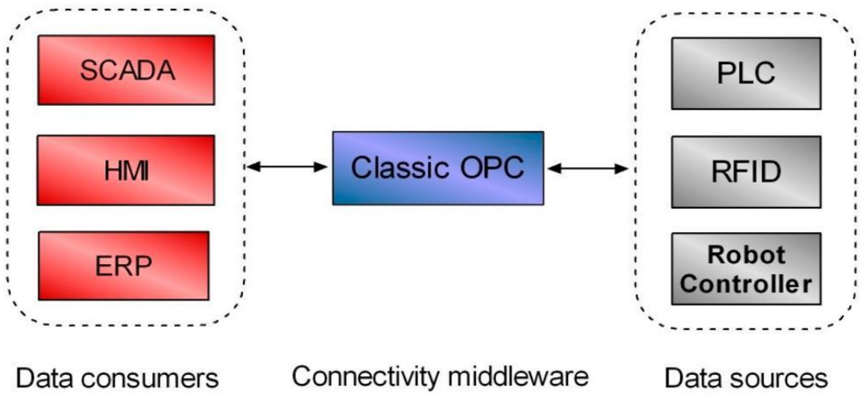


Figure 4: Classic open platform communications (OPC)-based communication scheme in automation system (González et al., 2019).

2.2.1 OPC UA

OPC UA, is gaining ever-increasing attention. Developed as the successor to classic OPC, UA specification was released in 2006 and is an IEC international standard of the international electrotechnical commission (IEC), namely, IEC 62541. (González et al., 2019). The emergence of service-oriented architectures within manufacturing systems brought forth fresh complexities related to security and data structuring. In response to these demands, the OPC Foundation crafted the OPC UA specifications. This development not only catered to these requirements but also introduced an expansive technology framework with an open and adaptable architecture. Notably, it was designed to withstand the test of time, offering scalability and extensibility. (OPC Foundation, 2017).

Four different application scenarios for OPC UA will be described (Schleipen et al., 2016, p. 316):

- Quality defect tracking system
- Visualization of process information (monitoring)
- Management information board (monitoring and control)
- Orchestration of cyber-physical production systems (production cells) (our specific case): This scenario demonstrates the ability of OPC UA to be used as generic interface for orchestration of components in a production cell. This can be used to achieve flexible software deployment in adaptive plants.

In this thesis, the latter feature is of utmost significance. In fact, one of the primary objectives of this project was to develop the coordination across various levels of an agile production plant by utilizing the OPC UA communications standard.

OPC UA is a vehicle for describing status and results of components and software modules and providing access to the services of the software modules and components. If the properties of one component fit to the need of another component, e.g. a camera for a robot, the component is able to connect to the other component and to execute the analyzed code during the production process. The state of the methods and their accessibility depend on the status of the system. For example, the distribution method is only available if the analysis was successful.

The orchestration OPC UA server was designed and developed to provide platform- and system-independent information via OPC UA. It receives input from a user or system which includes the description of the whole production scene to orchestrate.

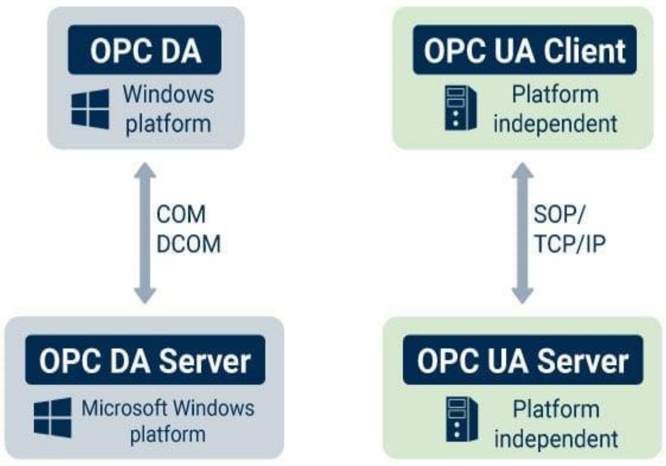


Figure 5: Differences between OPC DA and OPC UA (Rinke, 2022).

In recent years, OPC technology has been adopted by more and more users, in particular it's used to collect several different process data connected to multi-remote OPC server to a specific client application and, with a read-write methodology, this leads to monitor and real-time acquisition of process parameters and other variables.

The OPC standard plays a pivotal role in defining an interface between client applications responsible for data processing and the servers that establish connections with physical industrial devices like PLCs, sensors, and actuators for control. This standard additionally summaries various objects with their respective properties and methods, serving as a standardized means to access data from control devices. All OPC servers working on this model serve as data sources for clients, acquiring the data requested by client devices connected to the industrial process (Diaconescu & Spirleanu, 2012, p. 3).

The image below provides a detailed depiction of how a client application communicates with a specific process. This process is directly linked to the distributed control system, which could be a PLC, SCADA system, or involve actuators and sensors. The communication interface is established through the OPC server, and ultimately connects with the appropriate client.

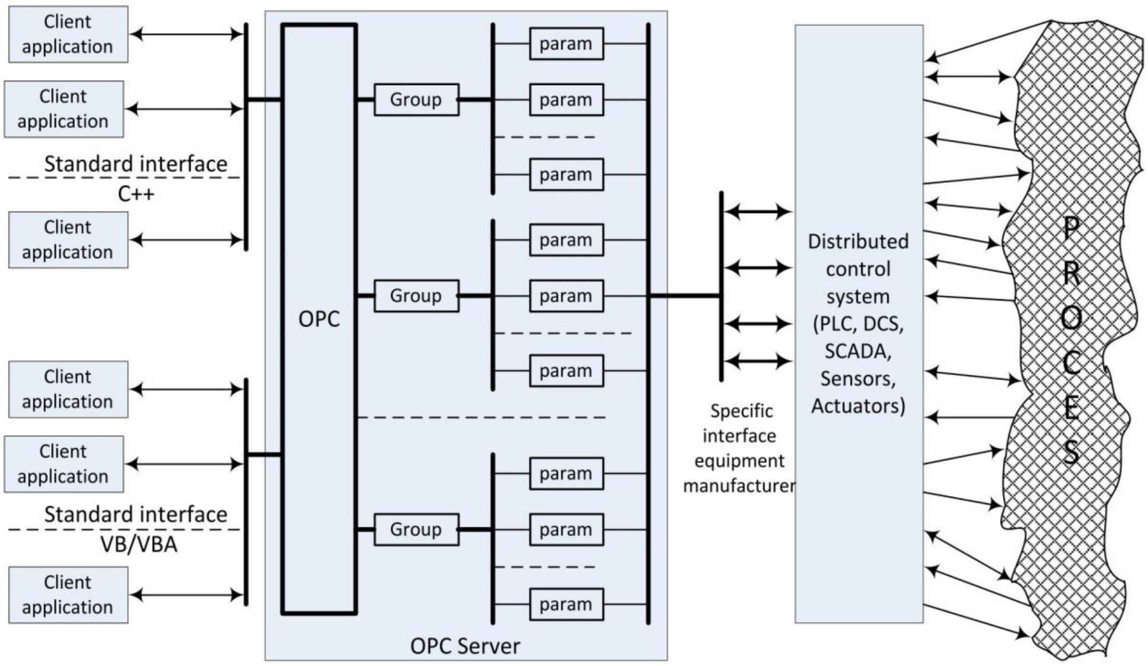


Figure 6: General structure of a control system using OPC server. (Diaconescu & Spirleanu, 2012, p. 3)

Over the course of several decades, fieldbus technology has evolved to meet the demands of the automation environment. It prioritized attributes like user-friendliness for installation employees, resilience in challenging and dirty conditions, as well as cost-effectiveness with simplified wiring.

In recent years, the Transmission Control Protocol (TCP) standard has gained increasing significance due to its ability to simplify network conversations, enabling applications to exchange data. TCP operates as a connection-oriented protocol, signifying that a connection is established and sustained until both ends of the communication have completed their message exchanges (Networking, 2023).

Conversely, Ethernet and TCP/IP have gained wide acceptance within Information Technologies (IT), offering easy access and seamless integration with global internet networks and technology. In clean office environments, Ethernet and TCP/IP stand as the typical communication network, widely recognized and cost-effective, with a substantial pool of IT experts well-versed in their usage. However, to extend this technology for automation purposes, adjustments are needed to adapt to the rugged "field" environment. Transmitting real-time control information imposes precise requirements that necessitate tailored physical interfaces and installation technologies. These requirements are related to determinism, queues, consistence and cyclic control. (Felser, 2001, p. 501).

2.3 Programmable Logic Controller (PLC)

The National Electrical Manufacturers Association (NEMA) defines a Programmable Logic Controller as: “A digitally operating electronic apparatus which uses a programmable memory for the internal storage of instructions for implementing specific functions such as logic, sequencing, timing, counting, and arithmetic to control, through digital or analog input/output modules, various types of machines or processes.” (Netto, 2013)

A PLC is a computer-based device used to control and coordinate various industrial equipment. These are widely utilized in today's industry due to their exceptional efficiency in managing sequential control and process synchronization. Initially designed for digital signal-based switching operations, PLCs have evolved to handle analog signals, making them versatile for a wide range of control processes. Unlike traditional computers, PLCs do not have a monitor; instead, they often incorporate a Human Machine Interface (HMI) flat screen display to depict process or production machine statuses. (Alphonsus & Abdullah, 2016, p. 1187)

In these devices we can clearly identify five different main blocks (Alphonsus & Abdullah, 2016, p. 1187):

- *Rack assembly*: this component is responsible for housing Input/Output modules, processor modules, power supply, and the processor unit. It also simplifies electrical connections between these modules via a printed circuit board at the rear.
- *Power supply*: it provides direct current power to the other modules connected to the rack.
- *Programming device*: used to program the CPU.
- *Input/Output section*: this is where all field devices connect and interface with the CPU. It can be a fixed setup, primarily for small and micro PLCs, or a modular configuration that employs a rack to accommodate varying numbers of I/O modules. Input interface modules obtain signals from machine or process devices, converting them into signals usable by the controller. On the other hand, output interface modules convert controller signals into external signals utilized for machine or process control.
- *Central Processing Unit (CPU)*: the CPU attends as the central coordinator and controller of the entire programmable controller system. Classically positioned at one side of the rack assembly, the processor module comprises integrated circuit chips housing one or more microprocessors, memory chips, and circuits that facilitate data storage and retrieval from memory. The CPU consists of two main components: the Arithmetic Control Unit (ALU) and memory. The ALU is responsible for implementing mathematical calculations and logic functions. On the other hand, the memory component of the processor stores the programs and vital data for the CPU to carry out different operations.

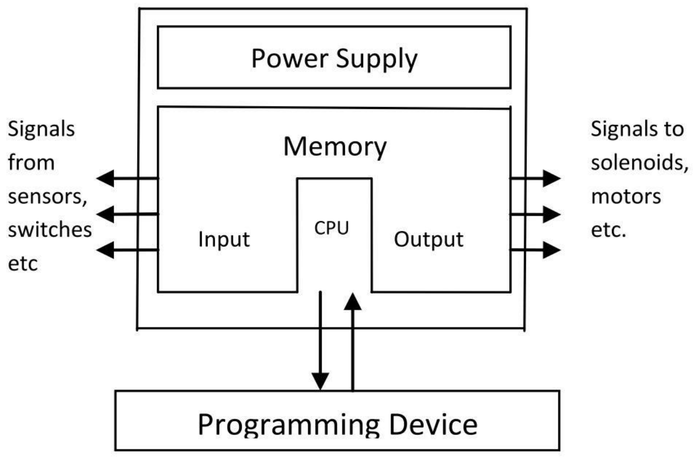

Power Supply
Signals from sensors, switches etc
Memory
Input
CPU
Output
Signals to solenoids, motors etc.
Programming Device

Figure 7: Block diagram of a PLC (Netto)

Industries often have production tasks that involve high levels of repetition. Despite the repetitive and monotonous nature of these tasks, they request the operator's careful attention to ensure effective production. Whenever there's a need for sequential control and automation, PLCs are the best choice to carry out these tasks effectively (Netto).

In recent years, PLCs have seen significant improvements, making them essential to many automation processes, offering users flexibility and efficiency. A common practice is to connect PLCs with other devices such as controllers to perform tasks like supervisory control, data collection, device and parameter monitoring, as well as program uploading and downloading. Modern PLCs are also capable of handling timer and counter functions, memory operations, and mathematical computations (Netto).

2.4 Agile Manufacturing

In recent decades, there has been a growing emphasis on flexible and agile manufacturing systems. The basis for this shift began in the 1980s, encouraged by challenges related to excess inventory, shortened lead times, and the pursuit of higher product quality and customer service. This led to the introduction of the term "Lean Production”. By the 1990s, efforts were made to formulate a new manufacturing paradigm, even though many firms were still grappling with the implementation of Lean Production. While these two concepts may appear similar, they have significant differences. Lean Manufacturing is primarily a response to struggle within constraints and emphases on operational techniques that optimize resource utilization. In contrast, Agile Manufacturing responds to the complexities arising from constant change and adopts a holistic strategy aimed at thriving in an unpredictable environment. In this post-mass-production era, the sharing of resources and technologies among firms becomes essential. An agile enterprise possesses the organizational flexibility to choose the most suitable managerial method for each project, thus achieving the greatest competitive advantage.(Sanchez & Nagi, 2001, p. 2)

2.5 ISA 88 / DIN EN 61512

The ISA-S88 is a standard from the International Society of Automation (ISA) for batch-oriented operation of a system. This standard was published a few years later as the DIN standard DIN EN 61512. Like ISA-S88, DIN EN 61512 consists of four parts that are almost identical. The following work will refer to the German DIN standard. Below these four parts are listed:

- Models and Terminologies (ISA-88.00.01 / DIN EN 61512-1) (2010)
- Data Structures and Language Guide (ISA-88.00.02 / DIN EN 61512-2) (2001)
- Models and representations of process and factory recipes (ISA-88.00.03 / DIN EN 61512-3) (2003)
- Batch production records (ISA-88.00.04 / DIN EN 61512-4) (2006)

DIN EN 61512-1 divides into three different process types. Continuous processes, processes with piece production and batch processes. A batch process is therefore defined as follows:

“A process that leads to the production of finite amounts of substances by subjecting quantities of input materials to an ordered sequence of processing activities using one or more facilities within a finite period of time.” (DIN EN 61512-1, p. 4)

2.5.1 Models for system description

The standard provides various models to relate systems and the processes taking place therein. The standard shows a connection between the following models. “The models described in the standard are […] viewed as complete” (DIN EN 61512-1, p. 3).

2.5.1.1 Process model

The process model, as we see in *Figure* 8, breaks down the batch process into different sections, but remains in a more abstract description. The batch process is defined as above and describes the complete sequence of all process steps that are necessary to produce a batch of a product. The standard uses the example of the production of polyvinyl chloride to clearly describe the model. The batch process is therefore the production of polyvinyl chloride, the top level “process” in *Figure 8*. This batch process consists of an ordered number of process sections. The process sections can run serially, run in parallel or both at the same time. A process section usually runs independently of other process sections and usually causes a chemical or physical transformation of the processed substances. Using the example of polyvinyl chloride production, the following process stages would be present: polymerization, recovery and drying. Each process section in turn consists of process operations. Process operations are “major processing activities” such as “preparing the reactor,” “filling,” and “reacting.” The lowest level of the process model is formed by the process steps. They describe smaller processing steps. In polyvinyl chloride production, the processes would be “adding the catalyst to the reactor”, “adding the vinyl chloride monomers to the reactor”, “heating” and “maintaining temperature”. The process model hierarchizes processes and groups them into operations and sections. (DIN EN 61512-1, p. 7)

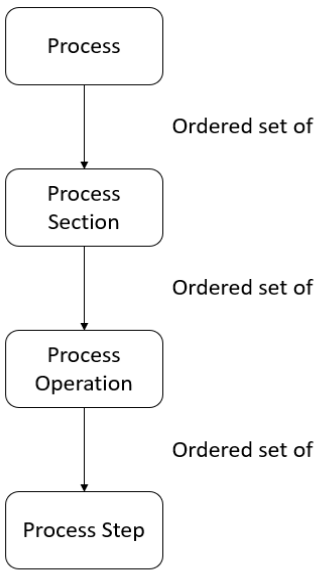


Figure 8: Process model according to DIN EN 61512-1

2.5.1.2 Physical model

In contrast to the process model, the physical model does not describe a process structure, but rather the structure of physical goods such as plants, systems, machines, actuators or control devices. Of the seven levels in *Figure 9*, the top three levels only play a role in a business function and are therefore not considered further. “The four lower levels of facilities (systems, sub-systems, technical equipment and individual control units) are defined by engineering activities” (DIN EN 61512-1, p. 7). A facility is defined as a “logical grouping of facilities that contains the facilities required to produce one or more batches” (DIN EN 61512-1, p. 5). However, the delimitation of a system takes place according to organizational or business criteria. The system level can be divided into different sub-systems. A unit can carry out a variety of larger processing activities such as reactions, crystallization or solution production. It combines all the necessary technical and control components to carry out these activities as an independent unit. In physical terms, a technical device can consist of individual control units and subordinate technical devices. It can function either as part of a sub-system or as an independent group of facilities within a system, with the option of exclusive or parallel use. This technical facility is capable of carrying out a limited number of specific small-scale processing activities, such as dosing or weighing, and it integrates all the necessary procedural and control components to carry out this activity. (DIN EN 61512-1, p. 9)

A single control unit usually represents an integration of measuring instruments, actuators, other individual control units and the associated processing device, which is operated as an independent unit from a control technology perspective. It can also be composed of other individual control units, such as an individual metering control unit, which could be combined from several automatic switching valve individual control units. (DIN EN 61512-1, p. 9)

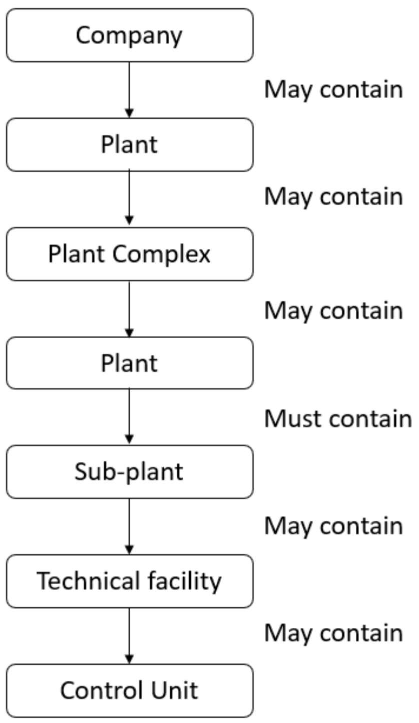


Figure 9: Physical model according to DIN EN 61512-1

2.5.2 Batch control concepts

2.5.2.1 Basic automation

Basic automation is a control concept that is used to maintain a specific operating state. This concept is used in the management of continuous processes, for example in the continuous production of synthetic fuels, in which reactions take place continuously. Since the focus of this work is not based on continuous but rather discrete processes, basic automation is not given any further attention here.

2.5.2.2 Procedural control

Procedure controls are often used in batch-oriented contexts to structure a facility-oriented action so that a process-oriented task is carried out. The spatial physical units/machines are combined with the process chain and united in the procedure control. A procedure is the highest level in the hierarchy and defines the strategy necessary to produce a batch. Example: “Produce PVC”.

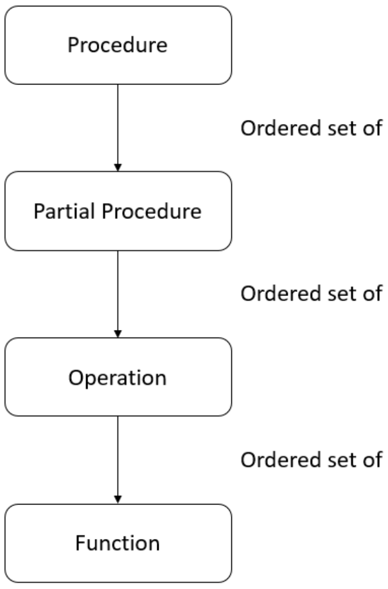


Figure 10: Model of a procedural control according to DIN EN 61512-1

A subprocedure consists of various operations in a fixed order. It is important that it is assumed that only one operation is carried out in a subsystem. However, partial procedures can be executed in parallel. Examples of sub-procedures would be “Polymerize vinyl chloride monomers”, “Recover vinyl chloride residues” or “Dry PVC”. Operations, in turn, contain various functions that may carry out a chemical or physical transformation of a substance. Examples of an operation in PVC production would be preparing the reactor, filling the reactor or the reaction itself. A function, the lowest level, can in turn be composed of other functions. It forms “the smallest element of a procedure control that can carry out a process-oriented task. […] The goal of a function is to cause or define a process-oriented action, whereas the logic or sequence of steps that make up the function is facility-specific.” (DIN EN 61512-1, p. 12).

2.5.2.3 Coordination control

Coordination control is at a level above procedural control. It “directs, triggers and/or changes the execution of procedure controls” (DIN EN 61512-1, p. 12). The availability of capacities, the availability of (partial) systems and the coordination of these fall within the scope of coordination control. Since this is implemented in a separate application, it is also not the focus of this work. (DIN EN 61512-1, p. 12)

2.5.3 Connection of the models with the procedure control

The relationships between the models are visualized again in *Figure 11*. The procedural control model uses the process model as a reference and draws on the physical entities and process variables to define the control logic and control operations. According to DIN EN 61512-1, the interaction between these models ensures efficient and consistent control of the process, including the control of physical components and process variables in accordance with the specified procedures and processes.

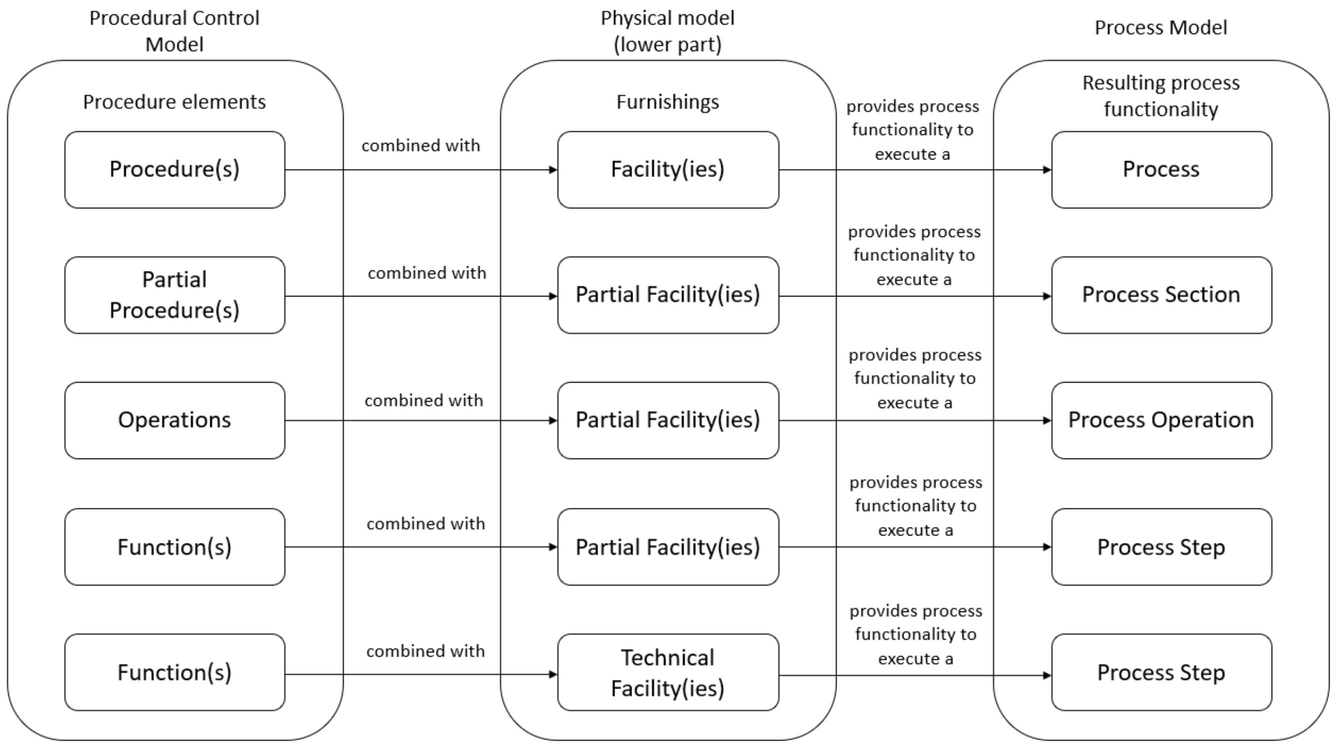


Figure 11: Interrelationships model procedure control, physical model and process model according to DIN EN 61512-1

2.5.4 Operating modes and operating states

The operating modes and operating states of DIN EN 61512 are briefly presented below. Since the standard is from 1999 and no longer includes the most current ISA proposals, the outdated operating modes and operating states are only briefly presented.

2.5.4.1 Operating modes according to DIN EN 61512-1

According to DIN EN 61512, every procedural element can have an operating mode. “An operating mode determines how facility objects and procedural elements react to commands and how they take effect.” The operating mode determines the manner in which transitions between the procedural elements take place. In automatic mode, there is no interruption between elements as long as the conditions are met. The control takes over the switching, so that no external operation is required. In the semi-automatic operating mode, manual switching through an external operation is necessary after the switching conditions have been met. However, the order remains unchanged. In contrast to the manual operating mode, the operator has to determine the order and the element to be carried out himself. “This standard does not exclude other operating modes and does not require the strict use of the operating modes mentioned here.” Relevant operating modes are summarized again in table below. (DIN EN 61512-1, pp. 29–30)

Table 1: Operating modes

Operating mode	Behave	Command
AUTOMATIC	Advances within a procedure are carried out without interruption if the associated conditions are met.	Operators can stop the procedure, but cannot force it to advance.
SEMI-AUTOMATIC	Advances within a procedure are triggered by manual commands when the associated conditions are met.	Operators can stop execution or redirect execution to an appropriate location. Handovers cannot be forced.
MANUAL	The sequential functions within a procedure are executed according to an operator specification.	Operators can stop or force advances.

2.5.4.2 Condition model according to DIN EN 61512-1

As with operating modes, each procedure element can also have one. DIN EN 61512-1 differentiates in its state model between an operating state and commands. An operating state is defined as a “state of a facility object or a procedural element at a specific point in time” (DIN EN 61512-1, p. 6). A command causes a transition from one state to the next state. The states for facility elements differ from the states of procedural elements. In the following we only consider states of procedure elements. Commands and procedural element states are described in a state model such as the DIN EN 61512-1 model in *Figure 12*. There are 3 different types of states: end, rest and transition states. They are shown graphically in *Figure 12*. An important aspect is that the change from one operating state to another operating state can cause changes in levels above or below. This aspect is important with regard to the hierarchy of procedural elements. (DIN EN 61512-1, pp. 30–31)

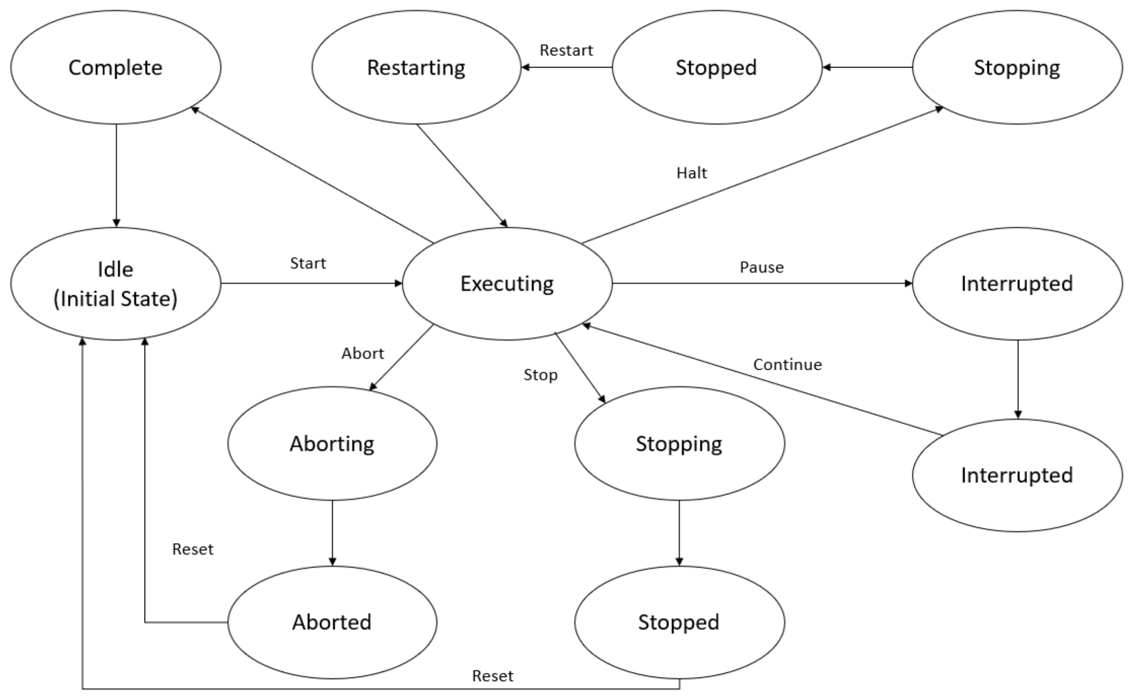


Figure 12: Condition model according to DIN EN 61512-1

2.5.5 PackML Interface State Manager

PackML stands for Packaging Machine Language and is a standard for controlling and communicating packaging machines. It was developed by the Organization for Machine Automation and Control (OMAC) and is based on the ANSI/ISA TR88.00.02-2022 technical report. PackML extends the ISA-88 standard with specific functions and commands for packaging machines. One advantage of PackML is the standardized data model and the common understanding of the operating mode and operating status of a unit. This enables consistent interpretation and use of data. Terms, abbreviations and definitions are taken from ISA 88. (Ph.D. Carsten Nøkleby, p. 0)

2.5.5.1 Physical model according to ISA 88.01

DIN EN 61512 describes a physical model that PackML is based on. The PackML condition model should start at the sub-system level and provide a condition description at this level. An interface should be developed for each sub-system that communicates the required information to the system control. PackML does not provide any information about the design of the underlying levels of the physical model. (Ph.D. Carsten Nøkleby, p. 13)

2.5.5.2 Operating modes according to ISA-TR88.00.02-2022

PackML defines its own operating modes for a unit. In contrast to the original DIN EN 61512, in which each procedural element has an operating state and an operating mode, with PackML both are only defined at the unit level. PackML determines in which operating mode a state model is mandatory and in which it is not. The following main operating modes are defined: Production, Maintenance, and Manual. These correspond to the modes from DIN EN 61512 automatic, semi-automatic and manual

2.5.5.3 Operating states according to ISA-TR88.00.02-2022

Operating states define the state of a subsystem. PackML has expanded the ISA-88/DIN EN 61512 state model by 5 states. The two main elements in PackML are states and commands, which trigger a transition from one state to another state.

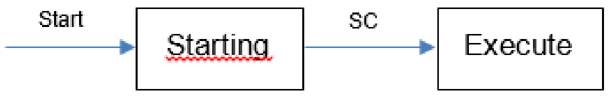


Figure 13: Syntax of the PackML state model

States are divided into two categories: executing states (states in which the unit executes actions) and waiting states (stable states in which a command ensures that a transition to the next state occurs). The only exception that satisfies both is the Execute state. It is both waiting and executing. In *Figure 14*, Start is a command, Starting and Execute are each state. SC means "State Complete" and indicates that an execution has been completed.

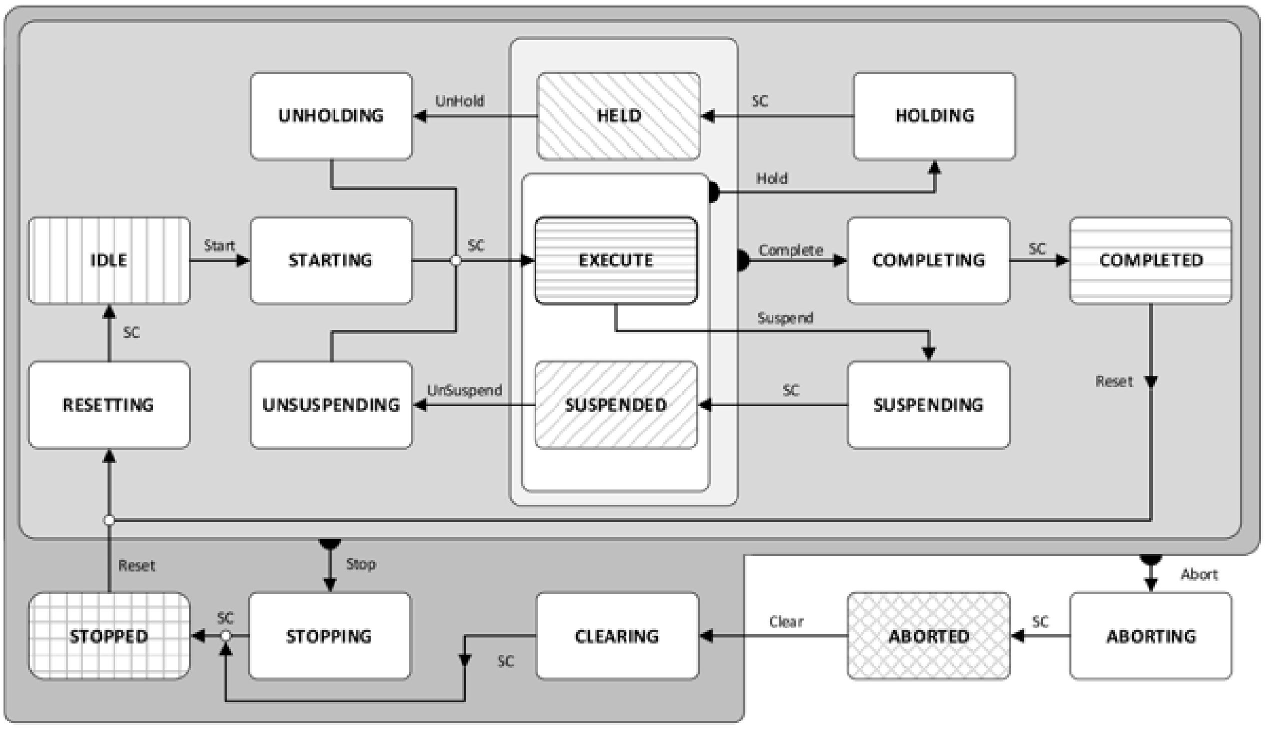


Figure 14: PackML state model DIN EN 61512-1

2.6 Siemens Totally Integrated Automation Portal (TIA Portal)

This project was developed using Siemens TIA Portal V18, a comprehensive software framework designed for programming Siemens hardware. The TIA Portal offers a wide range of functionalities, although not all of them are covered in this context. We focus on the specific functionalities essential for the concept presented here. During the development process, TIA V17 was initially used, and the project was subsequently migrated to the more recent TIA V18 version. The TIA Framework encompasses different software components:

- Siemens SIMETIC STEP 7 enables the programming of a PLC with structuring of the program codes, creation of various variables and communication with the PLC
- Siemens SIMETIC STEP 7 PLCSIM Advanced is a simulation software with which a PLC can be simulated to enable online monitoring via the TIA Portal.

2.6.1 Programming languages

In the functions (FC) and the function blocks (FB) it is possible to implement code using different programming languages. It is possible to use LAD, FBD, STL, SCL, GRAPH and CEM (see abbreviations) as programming languages. LAD, FBD and GRAPH are presented below because they are used in the context of this work.

2.6.1.1 Ladder (LAD)

Modeled after electrical circuits, the ladder diagram (LAD) is a graphical programming language that draws inspiration from circuit diagram schematics. It employs a visual design reminiscent of circuit diagrams, featuring a conductor rail on the left edge from which current paths extend. Binary signal queries, represented as contacts, are positioned along these current paths. The arrangement of elements on a current path dictates whether they are in series or in parallel. More intricate functions are expressed through encapsulated units.

A LAD program comprises elements that can be positioned in rows or in parallel on the network's busbar. These elements frequently require variables. Programming commences at the left edge of the current path branching from the conductor rail. The line rail can be expanded by introducing additional current paths and branches (Siemens, 2021, p. 8179).

2.6.1.2 Function Block Diagram (FBD)

Function Block Diagram (FUP) is a graphical programming language characterized by a visual representation reminiscent of circuit systems. This visual representation closely resembles electronic circuit diagrams, complete with interconnecting paths for binary signals, denoted by boundary boxes. Programs are depicted within networks where elements are linked through binary signal flow, utilizing logical symbols from Boolean algebra. FBD programs are constructed with elements that necessitate variables, and, similar to LAD, programming unfolds within a left-to-right network structure (Siemens, 2021, p. 8241).

2.6.1.3 GRAPH

GRAPH, also known as S7-Graph, is a graphical programming language designed for creating sequence controls. It is based on the GRAFCET design language, documented in DIN EN 60848. This language enables the clear and efficient programming of sequential processes through the use of sequencers. The entire process is broken down into manageable steps, each having a defined set of functions and organized into sequences. Each step outlines the actions to be executed, while the transitions between steps serve as connectors. These transitions include conditions that dictate the circumstances for advancing to the next step, known as transition conditions. These conditions are Boolean expressions, which can evaluate to either True or False. A fundamental principle is that every transition must be followed by a step, and conversely, every step must be followed by a transition (Siemens, 2021, p. 8428).

2.6.2 Data Type

The TIA Portal encompasses various data types, each defining the behavior, structure, and storage space for data. Within the user program, it's feasible to design custom data types that align with the programmer's specific needs. These data types are categorized in different ways, and in the context of this section, we'll provide a brief overview of the most pertinent categories in the following sections.

2.6.2.1 Elementary Data Type

The TIA Portal encompasses various data types, each defining the behavior, structure, and storage space for data. Within the user program, it's feasible to design custom data types that align with the programmer's specific needs. These data types are categorized in different ways, and in the context of this paper, we'll provide a brief overview of the most pertinent categories in the following sections.

2.6.2.2 Composite Data Types

Composite data types encompass data structures composed of various elementary data types. These can range from simple strings to arrays and even to an anonymous data structure known as STRUCT.

- Array: An array is a data structure consisting of a fixed number of elements, all of the same data type. This characteristic distinguishes arrays from STRUCTs.
- STRUCT: A STRUCT, on the other hand, is a data structure that integrates different data types, allowing for nested structures that users can modify as needed. However, they require adjustments if used multiple times and may not be compatible with PLC data types of similar structures. Additionally, they can exhibit poorer performance and increased memory demands."

2.6.2.3 User-defined Data Types (PLC data type)

A PLC data type is a user-defined data type that can incorporate various data types. It is possible to use all available data types, with a limited nesting depth of up to 8. PLC data types are particularly useful for generating data blocks, as they allow the creation of multiple variables of the same PLC data type. This facilitates making changes to all these variables simultaneously by modifying the original blueprint, as they all share the same definition. PLC data types are also well-suited for organizing and storing data in accordance with process control requirements. When an element is replaced, the corresponding data block can be easily updated.

2.6.3 OPC Server

In the context of the TIA Portal, the CPU functions as an OPC-UA server. To enable this functionality, certain settings need to be configured. These settings include defining server addresses and port numbers, which serve as access points for OPC clients. Additionally, critical security-related configurations like authentication and user management are established in this setup. It's also essential to specify any purchased licenses.

After the initial server setup, the next step involves enabling all the variables in the relevant data block for OPC UA access by setting them to 'enabled.' Once all the necessary settings are in place, the server is prepared for operation. Now, clients can access the various nodes on the server, provided they have the requisite permissions.

2.7 AgiloBat Project

The impetus behind the imperative for an adaptable production system geared towards battery cell manufacturing stems from three key driving forces. These forces encompass an escalating demand for battery cells driven by the surge in electrification trends, the adoption of spatially efficient and product-specific cell formats, and heightening uncertainty concerning geopolitical factors (Karlsruher Institut fuer Technologie, 2023).

The term agile comes from Latin term "agilis" and means "to drive, be in motion, do or perform". Consequently, agile production systems should be able to react quickly to changing market requirements. This increases their own competitiveness and enables high profit margins, especially at the beginning of product life-cycles, due to high demand compared to supply (seller's market). (Fleischer et al., 2022, p. 1252).

Figure 15: AgiloBat Logo ("Microsoft Word - Seminararbeit_SimonGese.Docx")

The methodology employed in this project stands in stark contrast to conventional approaches in battery production and design. The primary emphasis is on achieving a comprehensively optimized cell, considering factors such as resources, cost-effectiveness, and performance. The underlying concept revolves around the continuous fine-tuning of battery systems to align precisely with the unique specifications of each application and the available spatial constraints. For instance, the criteria for a battery destined for electric vehicles vastly differ from those of a power tool. In the forthcoming manufacturing process, these distinct requisites will be systematically translated into parameters for battery cells. The outcome will be a versatile array of cell shapes, specifically fine-tuned to accommodate a diverse spectrum of requirements. (Karlsruher Institut fuer Technologie, 2023).

Another important aspect to highlight is the economic benefits that arising from this type of plant. As shown in *Figure 16*, we can clearly observe how the AgiloBat production is thriftily advantageous on the long term respectively to the hand cell production.

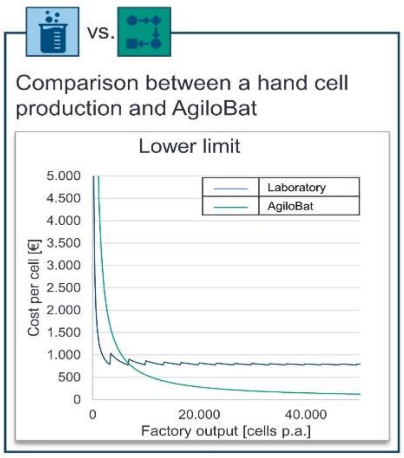


Figure 16: Comparison between a hand cell production and AgiloBat (luetgering)

2.7.1 Production

The agile production of battery cells occurs within compact local drying rooms. Consequently, the dew point temperature within these spaces can be tailored according to specific needs. These rooms, often referred to as microenvironments, are isolated from external surroundings and, housing internal machinery, serve as functional components in accordance with the established terminology. In *Figure 17* we can observe functional units dedicated to crucial process stages. These stages include wet *Coating* and subsequent *Electrode Drying*, *Calendering*, *Separation* or *Singulation*, and *Cell Assembly*.

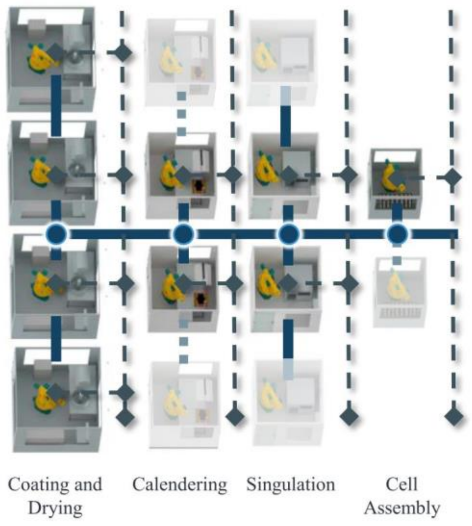


Figure 17: Functional principle of agile battery cell production based on microenvironments equipped with machine modules. The capacity utilization can be kept high despite different processing times thanks to redundant functional units (Fleischer et al., 2022, p. 1254)

Each microenvironment contains a 6-axis industrial robot of type KUKA KR22 R1610-2 as handling module to automate the material flow. In combination with highly automated process modules, this necessitates that no people are present in the microenvironments during production. This reduces employee exposure to hazardous substances and very dry air. There is no need for humans in the microenvironment, so less energy is needed to dehumidify the air. The material flow is realized automatically via material locks, whereby an infeed process takes approx. 2 minutes. (Fleischer et al., 2022, p. 1254). In *Figure 18* we can observe a detailed overview of the AgiloBat plant concept with all its functions and processes that the material undergoes from start to finish.

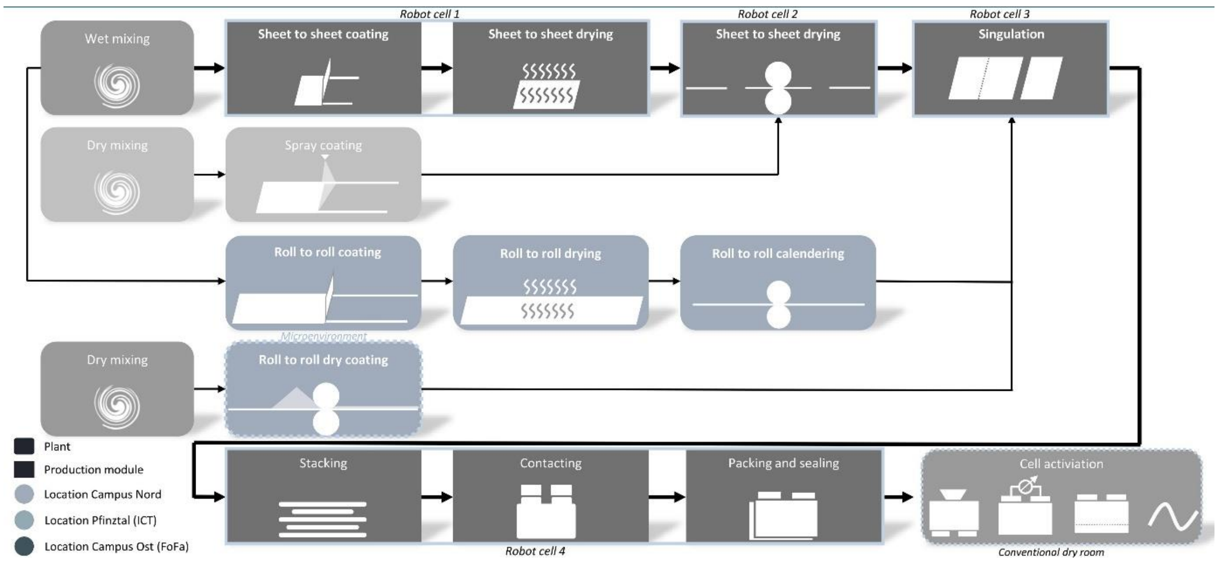


Figure 18: Overview AgiloBat plant concept (luetgering)

2.7.1.1 Cell 1: Coating and Drying (Beschichtung)

In this first production step the slurry is either continuously or intermittently coated on one or both sides. The next step of this phase is the drying process. Here the aluminium-copper sheet is fed directly into the dryer and if it was previously applied a simultaneous, double-sided coating, a flotation dryer must be used. The solvent is removed from the substrate by supplying heat and recovered or sent to thermal recycling. After passing through the dryer, the foils are cooled to room temperature.

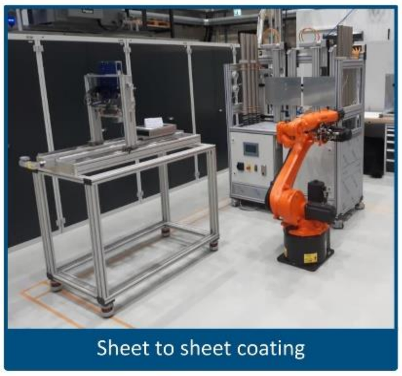


Figure19: Coating Cell (luetgering)

2.7.1.2 Cell 2: Calendering (Kalandrieren)

In this phase, the copper or aluminum foil coated on both sides is compressed by one or more rotating rollers. The pair of rollers are designed to create a precisely defined pressure to be applied to the sheets. It’s vital to correctly set this value as well as cleaning the rollers in order not to damage the foils and the substrate material.

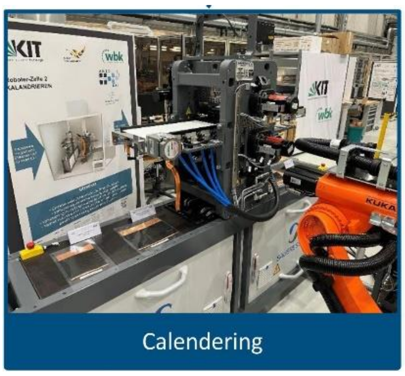


Figure 20: Calendering machine (luetgering)

2.7.1.3 Cell 3: Separating (Vereinzeln)

Separation is necessary for manufacturing the pouch cell and refers to separating the anode, cathode, and separator sheets from the rolled goods. It can be performed with a shear cut (punch tool) or thermally (laser cut).

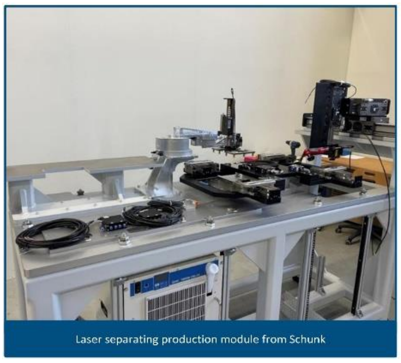


Figure 21: Singulation (luetgering)

2.7.1.4 Cell 4: Cell Assembly (Assemblieren)

To produce a pouch cell, a stacking process is usually carried out, and a winding process is carried out for the round and prismatic cells. In the stacking process, the electrode sheets are stacked in a repeating cycle of anode, separator, cathode, separator, etc. The anode and cathode strips are cut to length directly from the daughter coils produced for the winding process. First, the conductor foils (anode-copper and cathode-aluminum) are contacted with the cell conductors (pouch cell) or with the contact terminals (round cell and prismatic cell) using an ultrasonic or laser welding process. When placed in the packaging, the electrode stack of the flat cell or the jelly roll of the round cell and prismatic cell are placed in the packaging material of the cell. The pouch cell is closed with an impulse or contact seal, while the round and prismatic cells are usually closed with a laser welding process.

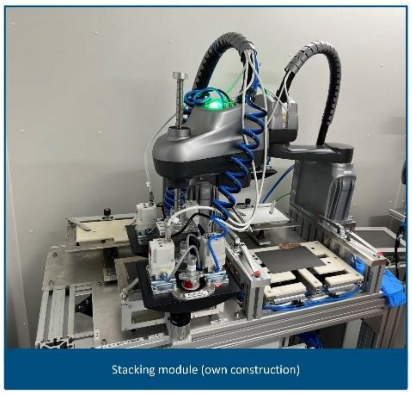


Figure 22: Cell assembly (luetgering)

2.7.2 Follow-up Project

Future updates regarding the AgiloBat project concern about the following main areas: *agile solid-state battery production*, *agile sodium-ion battery production*, *cell compound production* and *agile disassembly of cell compounds* (luetgering). In the following the developing topics for each area will be list:

- <u>Agile solid-state battery production:</u>
  - Automated production for largescale cell tests instead of manual cell construction in glovebox
  - Microenvironments as a protective barrier against H2S
  - Scalable throughput to accelerate market entry of solid-state batteries

- <u>Agile sodium-ion battery production:</u>
  - Conversion of LiB production to sodium battery (drop-in technology)
  - Transfer of LiB production know-how
  - Preparation of guidelines as a basis for future switch to sodium battery in gigafactories

- <u>Cell compound production</u>
  - Agile cell compound assembly
  - BMS development
  - Thermal management
  - Cell compounds as structural elements as enablers for lightweight construction

- Agile disassembly of cell compounds
  - Automated disassembly of individual cell clusters based on robot cells
  - Reuse of functional cells
  - Extension of AgiloBat with regard to circular economy

3 Graphical Interfaces and System Operations

As previously mentioned, a Graphical User Interface (GUI) serves as the optimal means for facilitating user interaction and communication with a production system, encompassing its various components and parameters. The integration of a graphical interface empowers the operator to effortlessly initiate the overall process, focus on individual segments, or even modify specific operational data through designated interface elements.

The app-based control system is implemented to oversee and manage an industrial process responsible for battery cell production across four autonomous microenvironments. This system employs OPC UA and HTTP protocols to establish communication with the PLCs, OPC UA Servers, and the database. All the PLCs are already programmed to receive and accept information from the GUI and to provide process states the app can easily read.

Within each microenvironment, there exists a central PLC that communicates with its dedicated OPC UA Server. This OPC UA Server is responsible for data collection from the machinery within the respective microenvironment. The control application utilizes an OPC UA Client to communicate with the OPC UA Server on the PLC, retrieving essential information concerning the microenvironment.

Furthermore, the control application employs an HTTP Client to facilitate data exchange with the database. The Database Infrastructure serves as a repository for critical data related to the industrial process, including target values for various parameters and real-time parameter values. The control application relies on this data to monitor the process and make necessary adjustments.

In summary, the application-based control system establishes a centralized mechanism for monitoring and governing the industrial process in battery cell production. The system is characterized by its scalability and reliability, allowing for seamless adaptation to diverse industrial processes' requirements. Everything is shown in details in *Figure* 23.

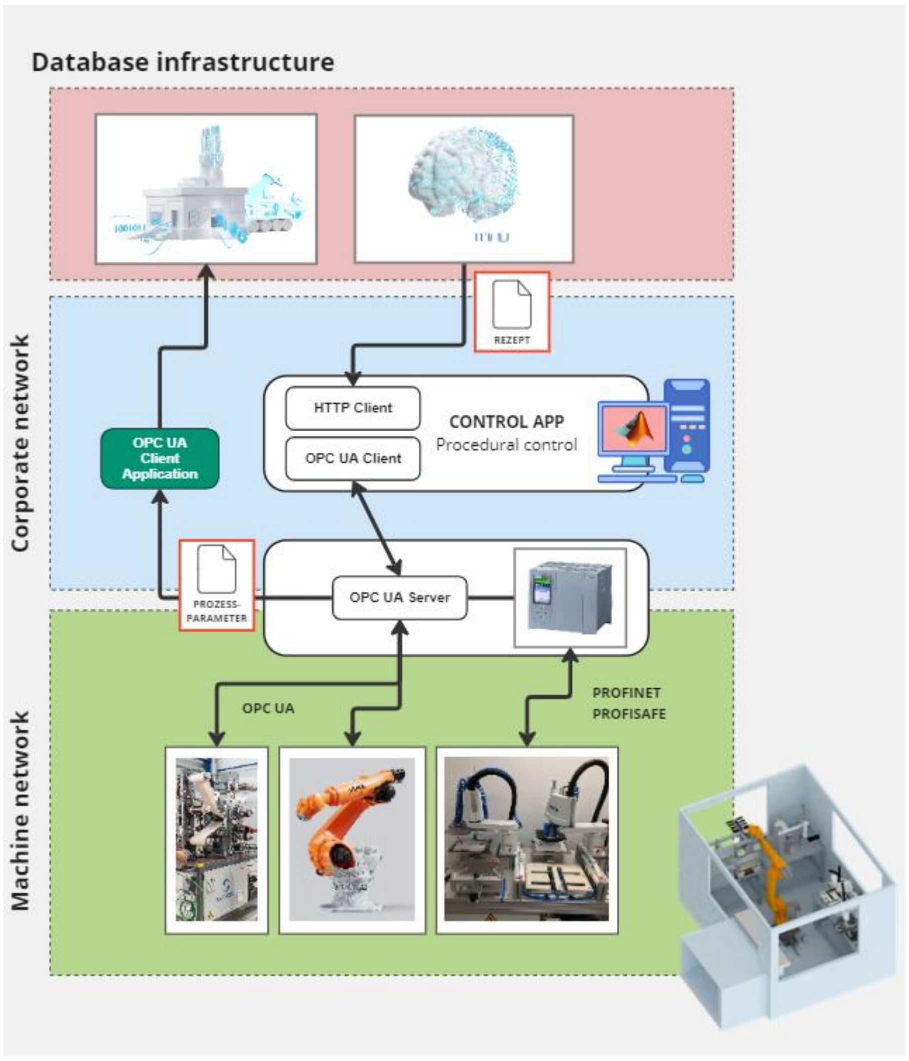


Figure 23: OPC UA-Based Monitoring and Control System

The developed GUI offers several essential capabilities to the operator, including the ability to input pertinent information such as:

- The URL of the Database HTTP Endpoint
- A Bearer Token, essential for establishing a secure connection with the database
- Selection of the cell type
- Access to the four distinct OPC Endpoints corresponding to the four Programmable Logic Controllers (PLCs), each associated with a distinct working cell

Furthermore, the GUI permits navigation through the complete array of processes and sub-processes within the production system. This empowers the operator to select specific parameters associated with a particular process, thereby retrieving its corresponding value from the database. If deemed necessary, the operator can also make localized changes to these values without impacting the corresponding data stored within the database.

An additional pivotal feature of the application lies in its ability to visually present the ongoing states of various sub-processes. This visual representation offers clarity regarding the progression of the overarching process and the current operation in execution. The application as a whole comprises seven distinct windows, enumerated below:

- Main Window: This serves as the initial interface upon program initiation. Here, users can opt to access process parameters, configuration settings, or enter one of the different cell to visualize the processes.
- Process Parameter Window: Within this interface, users can selectively target specific parameters associated with individual sub-processes, enabling parameter reading and potential value modification.
- Configuration Parameter Window: In this window, users can input essential data pertaining to the HTTP Endpoint URL for database communication, as well as the four OPC Endpoints corresponding to the distinct PLCs.
- Four distinct windows, each dedicated to a specific cell's sub-processes, along with the monitoring mode.

Subsequent chapters of this document will expound upon the comprehensive functionalities encapsulated within each of these windows.

3.1 Color-Coded System

We have four distinct microenvironments, each of which is designated to handle specific responsibilities. These primary responsibilities within each microenvironment are referred to as "level 1 processes." For instance, let's examine the Kalandrierzelle cell as one of these microenvironments. This cell is primarily responsible for the calendaring task. Consequently, this calendaring task becomes one of the "level 1 processes" within the battery cell production system. To accomplish this task, a set of 3 essential main tasks is required:

- Provide the transport box.
- Perform calendaring of sheets.
- Pick up the transport box.

These main tasks, often referred to as "level 2 processes," represent the broader objectives that need to be accomplished within the Kalandrierzelle. To achieve each of these main tasks, a set of distinct actions must be executed. For example, for the first task, providing the transport box, the following sub-processes, referred to as "level 3 processes," are required:

- Unload the Automated Guided Vehicle (AGV).
- Perform the insertion action.
- Open the transport box.

The hierarchical structure extends further to encompass "level 4 processes”, which would be sub-processes of level 3 processes.

This hierarchical approach allows for a well-structured breakdown of tasks and actions, with each level representing a different layer of detail and specificity in the execution of the overall process. It ensures that every action is clearly defined and contributes to the successful completion of the broader tasks within the microenvironment, like the Kalandrierzelle.

Regarding the Main Window and the Kalandrierzelle interface, they will be equipped with buttons, each corresponding to a specific sub-process, and their color will reflect the state of that particular sub-process. This section will introduce and explain the criteria for assigning colors to these buttons. The objective is to furnish an intuitive and efficient interface for the users within the production plant. Below, the various colors and their respective meanings are presented:

- Grey: "OFF" indicates that all the machines within the microenvironment or a specific sub-process are currently switched off and await operator initiation by pressing the start button under it.
- Yellow: "ACTIVATING" signifies that the machines are in the process of warming up for the selected task.
- Green: "IDLE" indicates that all systems are activated and ready to commence the specific job. This state occurs when the start button is pressed or when a task has just concluded, and the machines are prepared for the next task.
- Blue: "ONGOING" denotes that the process has initiated, and all machines are actively engaged in their tasks. This state occurs after pressing the execute button.
- Dark green: This color indicates that the individual process has “CONCLUDED” and been successfully executed. Upon completion, the button reverts to the "IDLE" state, represented by green.

This color-coded system simplifies the understanding of the processes and their states, streamlining operations within the production environment and it is immensely beneficial for the plant as it provides a clear and intuitive visual representation of the various processes and their states, taken right from the OPC UA Server on the four different PLCs. It enables plant operators and personnel to easily and rapidly assess the status of each sub-process at a glance. This instant visibility enhances operational efficiency and minimizes the chances of errors, as operators can quickly identify machines that need attention or processes that are ready to proceed.

By employing such a user-friendly interface, the plant can optimize its production workflows, reduce downtime, and improve overall productivity. Operators can respond promptly to changes in the production process, ensuring that machines are utilized efficiently and effectively. In summary, this system not only streamlines operations but also contributes to a more agile and responsive production environment.

3.2 Main Window

The Main Window, as previously mentioned, serves to display the current status of each cell, making it evident which one is currently operational. From here, we can navigate to other windows and explore the sub-processes within each microenvironment: *Beschichtung*, *Kalandrierung*, *Vereinzelung*, and *Assemblierung*. For the purposes of this thesis, the focus has been primarily on developing the environment for the *Kalandrierzelle*.

Figure 23 presents the layout of the initial window, which serves as a central control hub for the system. In the center of the window, four distinct buttons are prominently displayed, each meticulously associated with a specific cell. These buttons dynamically adapt their colors in real-time, in a way that will be describer in Chapter 3.2.3, conveying valuable information about the status of the corresponding processes, with the aid of the Color-Coded system detailed earlier. Below this array of cell-specific buttons, you'll find two additional buttons, namely the "Start" and "Execute" buttons.

The "Start" button plays a pivotal role in triggering the machinery warm-up process, a critical step in preparation for the actual execution of tasks. It sets the stage for all the machines, ensuring they are adequately prepared for their respective roles. On the other hand, the "Execute" button takes on the responsibility of commencing process execution once the warm-up is completed.

In the specific scenario showed below, we delve into the details of cell number 2. In the first figure, this particular cell is depicted in an "OFF" state, indicating that all the associated equipment remains dormant, awaiting a crucial initiation.

In the subsequent figure, a notable transition occurs. The "Start" button has been pressed, setting into motion a sequence of events. As a result, all the equipment within the cell comes to life, marking a significant shift in status. The entire cell now resides in an "IDLE" mode. It stands prepared and fully activated, standing by for the forthcoming execution phase.

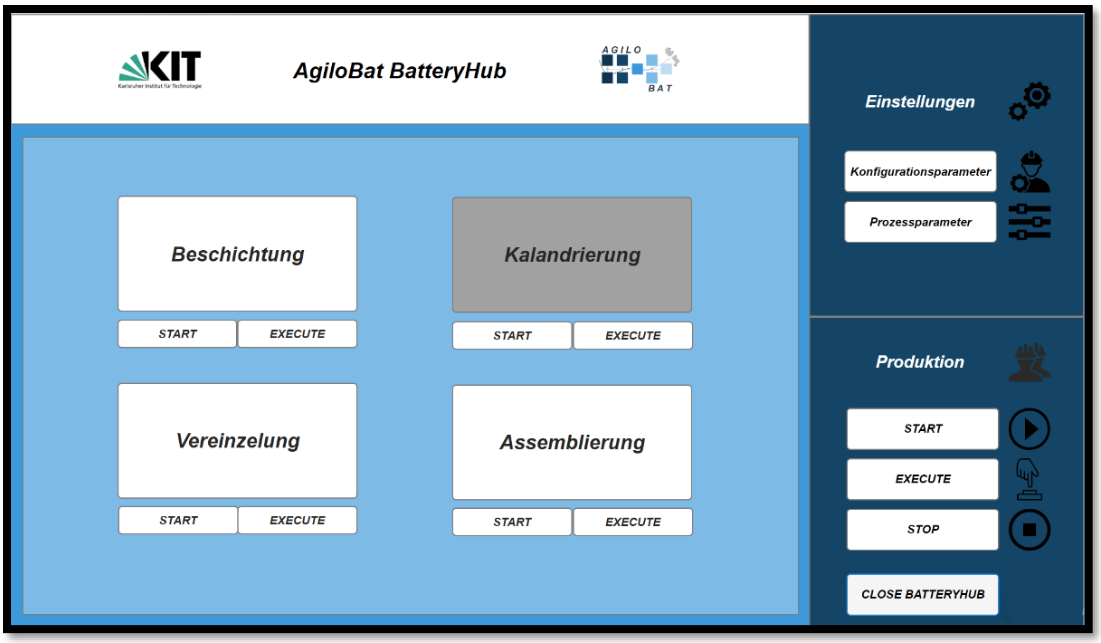


Figure 24: Main Window, ‘off’ state

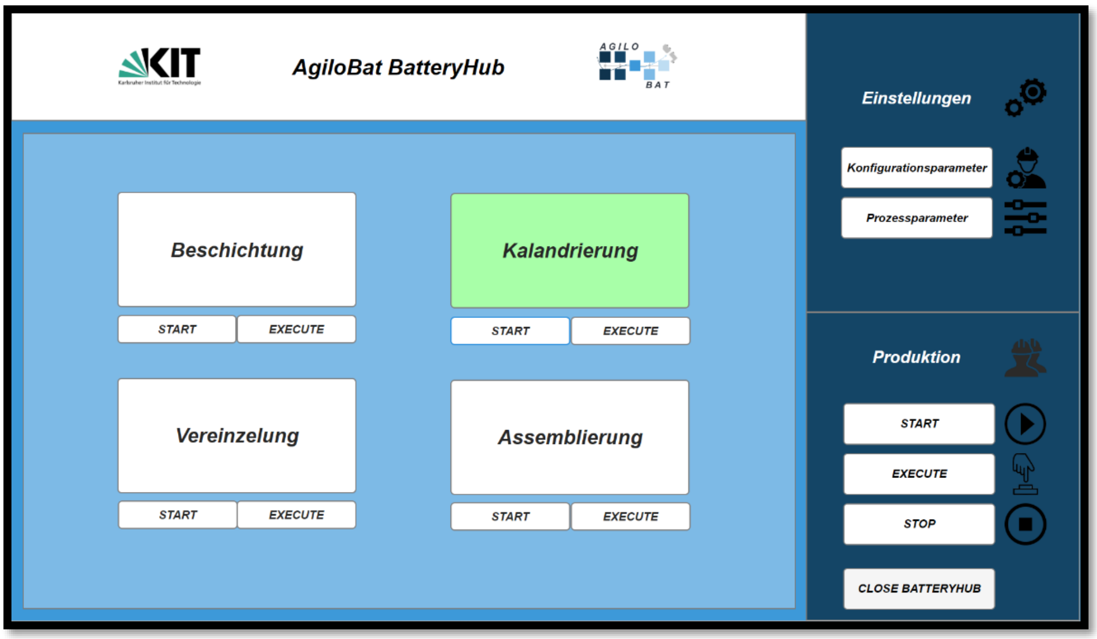


Figure 25: Main Window, ‘idle’ state

On the right side of the window we have the setting (*Einstellungen*) and the production (*Produktion*) sections:

- Setting: we can, in this panel, easily control the configuration parameters (*Konfigurationsparameter*), such as the OPC Endpoints of the cells and other data regarding the access to the database, as well as the process parameters (*Prozessparameter*). We'll see everything explained better and in depth in the next chapters.
- Production: in this other panel we have the possibility to activate the whole process, from *Beschichtung*, to *Assemblierung*, with a START, an EXECUTE and a STOP button. For the purposes of this thesis these functionalities have not been developed yet, because of the limited time available

Eventually, at the right bottom we have the CLOSE button, with which the application is closed.

3.2.1 Integration of OPC UA Communication for Real-time Data Exchange

As an initial step, it is crucial to establish a connection with the OPC UA Server on the PLC by creating a client. This is because the Server accepts commands from the Graphical User Interface, and displays states the GUI can read. This is the reason of this connection: it is essential for working with real-time data from the ongoing process. This initialization is performed within the startup function of the Main Window and is replicated across all other cell interfaces, which will be discussed later. The code for this process remains consistent, is shown in the Appendix and functions as follows:

- *HTTP Communication Setup*:
  - HTTP Endpoint and Authentication: The code begins by specifying the HTTP endpoint for the external web service (httpEndpoint) and the corresponding bearer token (bearerToken). This token is used for authentication and authorization purposes, ensuring secure access to the web service.
  - UA Specs Configuration: The OPC UA specifications are set up next. The uaEndpoint variable defines the OPC UA server's endpoint, which is essential for establishing communication with the local automation system. This part of the code prepares the connection to the OPC UA server.
  - HTTP Headers: HTTP headers, such as the bearer token and accepted media types, are configured. These headers ensure that the HTTP requests to the web service are properly formatted and include necessary authorization details.

- *HTTP Request and Response*:
  - HTTP Request: A POST request is created using MATLAB's HTTP toolbox. It includes the HTTP method, headers (with the bearer token), and the message body. The message body is constructed as a JSON object representing specific data needed for the web service interaction.
  - URI Configuration: The URI (Uniform Resource Identifier) is configured with the httpEndpoint, defining the target endpoint for the HTTP request.

- Sending the Request: The HTTP request is sent to the web service using the specified URI. This initiates communication with the external service, triggering the desired action.
- HTTP Response Handling: The HTTP server's response, including its status code, is captured and processed. This status code indicates the success or failure of the HTTP request.

▪ *OPC UA Client Configuration*:
- OPC UA Client Setup: A global OPC UA client (uaClient) is established, connected to the OPC UA server specified by uaEndpoint. This client serves as the communication bridge between the user interface and the underlying automation system, enabling real-time data exchange via OPC UA protocols.

This section of the chapter lays the foundation for integrating web services into the automation environment, allowing for real-time data exchange between the local system and external services. The code presented here demonstrates the configuration of HTTP requests and the setup of an OPC UA client, pivotal components in achieving efficient data communication and automation control. It’s important also to point out that by declaring uaClient as a global variable, we ensure that we don't need to repeat the code every time we need to establish a connection with the OPC UA server:

```
global uaClient;
uaClient = opcua(uaEndpoint);
connect(uaClient);
```

3.2.2 Starting and Executing

In order to initiate and carry out the production process, it is necessary to implement a code segment capable of triggering the appropriate variable on the OPC UA server. As an illustrative case, we will delve into the functionalities of the start and execute buttons associated with the Kalandrierzelle, which represents the second phase of the process. It's worth noting that for all other analogous buttons encountered later in this thesis, the underlying code follows a similar structure, as demonstrated below:

```
% Button pushed function: STARTButton_3
function STARTButton_3Pushed(app, event)
    global uaClient;
    string = ['"Prozess_DB_ML".', '2_0_0_0', '."Start"'];
    plcNode = opcuanode(3, string);
    writeValue(uaClient, plcNode, 1);
end

% Button pushed function: EXECUTEButton_2
function EXECUTEButton_2Pushed(app, event)
    global uaClient;
    string = ['"Prozess_DB_ML".', '2_0_0_0', '."Execute"'];
    plcNode = opcuanode(3, string);
    writeValue(uaClient, plcNode, 1);
end
```

The global variable uaClient is declared as global in advance, which is essential because it establishes a connection with the OPC UA server, permitting the exchange of real-time data with the ongoing industrial process. This declaration ensures that we do not have to replicate the code for setting up the OPC UA client every time we need to establish a connection. This reusability simplifies the code structure, making it more efficient and easier to manage. Let's break down the functions' actions:

- *STARTButton_3Pushed*: This function is associated with a button press event, which is typically used to initiate the start of a specific process. It constructs a string representing the path to the "Start" node within the OPC UA server, related to the process identified by the ID '2_0_0_0'. Subsequently, it uses the global uaClient to write a value of 1 to this node, effectively instructing the system to commence the process.
- *EXECUTEButton_2Pushed*: Similar to the previous function, this one is linked to a button press event, but it is responsible for executing a specific action. It constructs a string specifying the path to the "Execute" node within the OPC UA server, related to the same process as in the previous function. When activated, it writes a value to this node, triggering the execution of the desired action within the process.

The importance of these functions lies in their role as the user interface's interactive elements, allowing operators or users to actively control and interact with the industrial process. By writing values to specific nodes within the OPC UA server, these functions enable the application to send commands to the underlying automation system, thereby influencing the real-time operation of the process.

3.2.3 Real-time PLC State Monitoring and Button Color Updating

In this section, we delve into the development of a crucial feature within the Human-Machine Interface (HMI) for the industrial automation system. The presented code addresses the dynamic updating of button colors, representing different processes, based on the real-time state information obtained from Programmable Logic Controllers (PLCs). This capability provides operators and users with immediate visual feedback regarding the status of various processes, enhancing situational awareness and facilitating efficient process monitoring.

```
            buttons = [];
            tagArray = [];

            buttons = [app.ID_2_0_0_0];
            tagArray = {'2_0_0_0'};

            % Create a timer object with fixed-rate execution and a period of 1 second.
            timerObj  =  timer('ExecutionMode',  'fixedRate',  'Period',  1,  'TimerFcn',
@updateButtonColor);

            % Timer callback function to update button colors based on PLC states.
            function updateButtonColor(~, ~)
                % Iterate through the buttons and update their background colors.
                for i = 1:length(buttons)
                    % Construct the string to access the PLC node for the current button.
                    string = strcat('"State_DB".', '"',tagArray(i),'"', '."State"');
                    disp(string)
                    % Create an OPC UA node and read the state value.
                    plcNode = opcuanode(3, string);
                    state = readValue(uaClient, plcNode);
                    % Update the button background color based on the PLC state.
                    if state == 0
                        buttons(i).BackgroundColor = [0.63,0.63,0.63];
                    elseif state == 1
                        buttons(i).BackgroundColor = [1.00,1.00,0.00];
                    elseif state == 2
                        buttons(i).BackgroundColor = [0.66,1.00,0.66];
                    elseif state == 3
                        buttons(i).BackgroundColor = [0.07,0.62,1.00];
                    elseif state == 4
                        buttons(i).BackgroundColor = [0.19,0.59,0.04];
                    elseif state == 5
                        buttons(i).BackgroundColor = [0.76,0.39,1.00];
                    elseif state == 6
                        buttons(i).BackgroundColor = [0.95,0.33,0.33];
                    end
                end
            end

            % Start the timer to periodically update button colors.
            start(timerObj);
```

Code Description:

- *Initialization*: The code begins by initializing two arrays, buttons and tagArray, which will respectively store references to the relevant buttons and their associated PLC tags.
- *Button and Tag Configuration*: In this section, we populate the buttons array with references to specific buttons representing processes within the interface. Additionally, we populate the tagArray with corresponding PLC tags. In the example provided, we are focusing on the second phase of the process, represented as '2_0_0_0,' but similar configurations can be made for the other buttons and tags.
- *Timer Creation*: A timer object, timerObj, is created to facilitate periodic updates of button colors. This timer is configured to execute at a fixed rate, with a period of 1 second. The TimerFcn property is set to call the updateButtonColor function at each timer interval.
- *Button Color Updating*: The core functionality is encapsulated within the updateButtonColor callback function. This function iterates through the buttons in the buttons array and dynamically updates their background colors based on the real-time state information retrieved from the OPC UA server. It constructs the appropriate string to access the PLC node for each button's state, retrieves the state value, and maps it to a corresponding background color as explained before. Different colors are assigned to different states, ensuring clarity and intuitiveness in the HMI.
- *Timer Activation*: Finally, the timer is started, initiating the periodic update of button colors according to the state changes in the PLCs. This feature enhances the user's ability to monitor ongoing processes, detect issues, and take timely actions within the industrial automation environment.

This functionality is pivotal in providing a user-friendly and informative interface, ultimately contributing to the efficiency and safety of industrial processes. It serves as a prime example of how real-time data integration and visualization can significantly improve the Human-Machine Interface in the context of industrial automation.

3.2.4 Initialization of Specific Process Interface

In this unit, we explore a critical aspect of the Human-Machine Interface (HMI) within the industrial automation system. The presented code focuses on the initialization of a specific process interface represented by the button labeled ID_2_0_0_0. This button, when pushed, triggers the creation of an instance of the Kalandrierzelle_FINAL interface, which is dedicated to monitoring and controlling the detailed aspects of the Kalandrierzelle process, the second phase of the overall industrial process.

```
% Button pushed function: ID_2_0_0_0
        function ID_2_0_0_0Pushed(app, event)
            app.Kalandrierzelle_FINAL=Kalandrierzelle_FINAL(app);
        end
```

Code Description:

- *Button Push Event*: The code is associated with the button's push event, specifically ID_2_0_0_0Pushed. When the user interacts with this button by clicking it, an event is generated, and this function is called.
- *Initialization of Kalandrierzelle_FINAL*: The primary action performed within this function is the creation of an instance of the Kalandrierzelle_FINAL class, referred to as app.Kalandrierzelle_FINAL. This instance represents a dedicated interface for monitoring and controlling the Kalandrierzelle process's various parameters and functions.
- *Interface Design and Functionality*: The Kalandrierzelle_FINAL interface is designed to provide detailed insights and control over the specific process phase, enabling operators and users to interact with and supervise the Kalandrierzelle process effectively. It likely includes various buttons, displays, and controls tailored to the unique requirements of this phase.
- *Modular and Extensible Approach*: This extract code showcases a modular and extensible approach to interface design. By creating dedicated interface classes for specific process phases, such as Kalandrierzelle_FINAL, the HMI can adapt to the complexity and specificity of each phase while maintaining a structured and organized design.
- *User Interaction*: The initiation of the Kalandrierzelle_FINAL interface demonstrates the importance of user interaction within the HMI. It allows operators to delve into the intricacies of individual process phases, enabling them to make informed decisions and adjustments as needed.

This code signifies the critical role of dedicated interfaces in industrial automation, where each phase of the process demands specialized attention and control. It serves as a testament to the versatility and adaptability of the HMI in accommodating the diverse needs of complex industrial processes.

3.3 Kalandrierzelle

Within each of the four microenvironments, there's a hierarchical breakdown of processes into three levels: Level 2 contains processes, under which are Level 3 processes, and finally Level 4 processes. In this window, there are as many panels as there are levels, each containing buttons for Level 2, Level 3, and Level 4 processes.

At the top, you'll find the Level 2 buttons, which remain fixed and positioned uniformly. All other buttons are generated dynamically based on the list of processes stored in a JSON file. This dynamic generation allows for easy window editing by simply modifying this file. As mentioned earlier, all buttons representing processes and sub-processes are color-coded to reflect their respective statuses.

The following image illustrates the Kalandrierzelle's window, where everything is in the "off" state, providing a visual reference for the initial status.

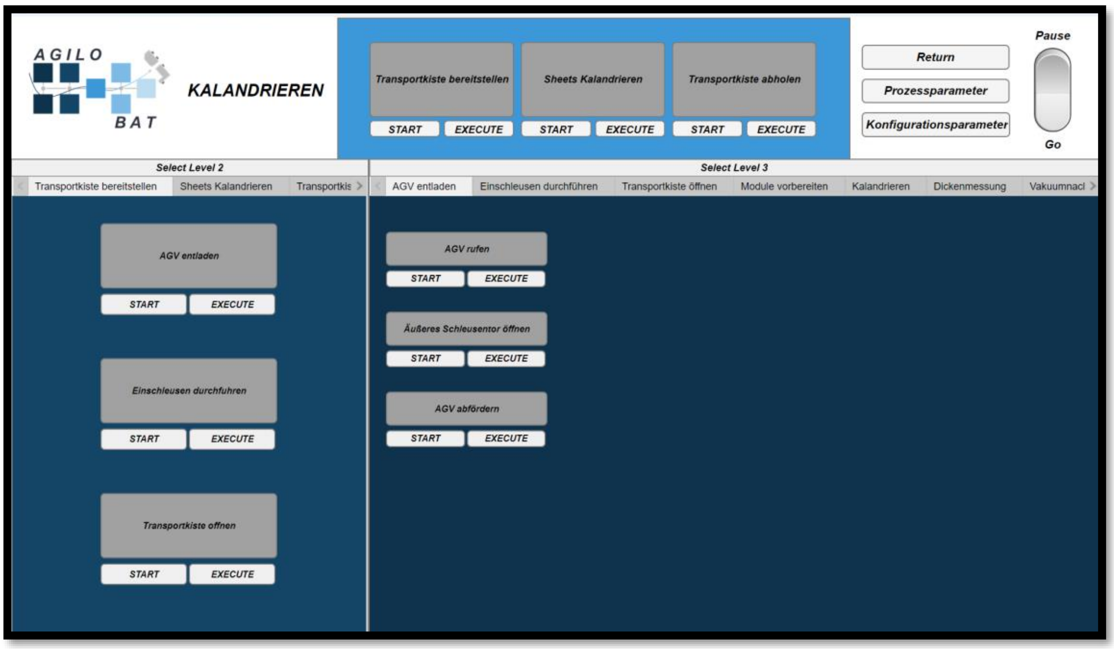


Figure 26: Kalandrierzelle, off state

As previously mentioned, at the top center of the window, you'll find the Level 2 processes. On the left-hand side, there's a panel where you can select the desired Level 2 process. Below this selection, the corresponding Level 3 buttons will be generated dynamically. Similarly, the large panel in the center functions in a similar manner. You select the desired Level 3 process, and beneath it, the Level 4 buttons associated with that process will appear dynamically. This hierarchical organization allows for a structured and intuitive interface to access different levels of processes and sub-processes within the system.

The additional buttons in the figure serve various functions:

- *Konfigurationsparameter* (Configuration Parameters): This button is used to access and configure the parameters related to the process setup.
- *Prozessparameter* (Process Parameters): Clicking this button allows users to access and adjust parameters associated with the ongoing process.

- *Rocker Switch* (Pause/Go): This switch serves as a control for pausing and resuming the process. Initially set to "Go" when the window opens, it can be toggled to "Pause" mode to halt the current process, which can be continued later.
- *Return Button*: Clicking this button closes the window and returns you to the Main Window.

3.3.1 UID Management and Uniqueness Verification in Hierarchical Structures

In this window we need to define a set of functions designed to manage and ensure the uniqueness of Unique Identifiers (UIDs) within a hierarchical structure. These UIDs serve as essential markers for distinct elements within the structure, facilitating their identification and reference. The functions perform the following key tasks:

- *updateUID Function*: this function iterates through the structure and calls the createUID function when it encounters a field named "UID." It ensures that UIDs are generated or modified to adhere to a specific format, enhancing their utility as unique identifiers. The recursive nature of this function enables it to traverse nested structures within the hierarchy.
- *createUID Function*: the createUID function is responsible for generating or adjusting UIDs to conform to a predefined format. It checks if the provided UID complies with the format, and if not, it generates a new UID that adheres to the specified structure. The format includes three random digits, followed by a lowercase alphabet character, another digit, and a second lowercase alphabet character.
- *checkUniqueUIDValues Function*: this function verifies the uniqueness of UIDs within the entire structure. It maintains a list of encountered UIDs and checks each encountered UID for uniqueness. In the event of a duplicate UID, the function raises an error, ensuring that all UIDs remain distinct.

These functions collectively contribute to the effective management and enforcement of unique identifiers, enhancing the structure's reliability and integrity. Additionally, they offer a valuable mechanism for ensuring the uniqueness of UIDs, crucial for precise element identification and referencing within complex hierarchical data structures.

3.3.2 Button Generation Function

An important aspect of this window's functionality is the automation of generating Level 3 and Level 4 buttons. This automation is achieved through two functions: "createButtonsLvl_3" for Level 3 processes and "createButtonsLvl_4" for Level 4 processes. Below, you'll find the first of these two functions along with detailed comments that provide insights into each step of the process, making it easier to understand how the buttons are dynamically created for these levels.

```
% This function creates level 3 buttons and associated functionality.
        function [tag, buttonArray] = createButtonsLvl_3(app, position, myStruct, spacing,
startY, buttonLength, buttonHeight)
            % Get the field names of the input structure.
            fields = fieldnames(myStruct);
            % Initialize arrays to store button objects and their tags.
            buttonArray = [];
            tag = [];
            % Initialize the position index for level 3 buttons.
            posLvl3 = 1;

            % Iterate through the fields in the input structure.
            for i = 1:numel(fields)
                fieldName = fields{i};
                value = myStruct.(fieldName);

                % Check if the value is a structure with a 'ButtonText' field.
                if isstruct(value) && isfield(value, 'ButtonText')
                    % Extract button text and PLC variable name.
                    buttonText = value.ButtonText;
                    ID = value.PlcVarName;

                    % Calculate button position within the panel.
                    panelWidth_Lvl3 = position.Position(3);
                    positionX = (panelWidth_Lvl3 - buttonLength) / 2;
                    positionY = startY - (buttonHeight + spacing) * (posLvl3 - 1);

                    % Create the level 3 button.
                    btns_lvl3 = uicontrol(position, 'Style', 'pushbutton', 'String',
buttonText, ...
                        'Position', [positionX, positionY, buttonLength, buttonHeight], ...
                        'FontName', 'Arial', 'FontWeight', 'bold', 'FontAngle', 'italic',
...
                        'Callback', {@lvl3ButtonCallback, app, value});

                    % Add the button object to the buttonArray.
                    buttonArray = [buttonArray, btns_lvl3];
                    % Add the tag (PLC variable name) to the tag array.
                    tag{end+1} = ID;
                     % Increment the position index for level 3 buttons.
                    posLvl3 = posLvl3 + 1;

                    % Calculate positions and create Start and Execute buttons.
                    buttonLengthAUX = (buttonLength / 2) - 2;
                    buttonHeightAUX = buttonHeight / 3;
                    positionXAUX = positionX;
                    positionYAUX = positionY - buttonHeight / 2 + 7;

                    positionXStart = positionXAUX;
                    positionXExecute = positionXAUX + buttonLengthAUX + 2;

                    % Create the Start button with a callback.
                    btnStart = uicontrol(position, 'Style', 'pushbutton', 'String',
'START', ...
                        'Position', [positionXAUX, positionYAUX, buttonLengthAUX,
buttonHeightAUX], ...
                        'FontName', 'Arial', 'FontWeight', 'bold', 'FontAngle', 'italic',
...
                        'Callback', {@startProcess, app, ID});

                    % Create the Execute button with a callback.
                    btnExecute = uicontrol(position, 'Style', 'pushbutton', 'String',
'EXECUTE', ...
                        'Position', [positionXExecute, positionYAUX, buttonLengthAUX,
buttonHeightAUX], ...
                        'FontName', 'Arial', 'FontWeight', 'bold', 'FontAngle', 'italic',
...
                        'Callback', {@executeProcess, app, ID});
                end
            end
            % Nested function to handle the Start button callback.
            function startProcess(~, ~, app, ID)
                    startFunction(app, ID);
                end

            % Nested function to perform the Start action.
            function startFunction(app, ID)
                % Construct the string to access the PLC node and write a value.
                string = ['"Prozess_DB_ML".', '"',ID,'"', '."Start"'];

                % Access the global UA client and create an OPC UA node.
                global uaClient;
                    plcNode = opcuanode(3, string);
                    writeValue(uaClient, plcNode, 1);
            end
            % Nested function to handle the Execute button callback.
            function executeProcess(~, ~, app, ID)
                executeFunction(app, ID);
            end

            % Nested function to perform the Execute action.
            function executeFunction(app, ID)
                % Construct the string to access the PLC node and write a value.
                string = ['"Prozess_DB_ML".', '"',ID,'"', '."Execute"'];

                % Access the global UA client and create an OPC UA node.
                global uaClient;
                    plcNode = opcuanode(3, string);
                    writeValue(uaClient, plcNode, 1);
            end
        end
```

The above code, operates as follows:

- *Input Parameters*: The function takes several input parameters, including the application object (app), the position information, the data structure (myStruct, obtained by the JSON file) spacing parameters, and button dimensions.
- *Initialization*: Arrays (buttonArray and tag) are initialized to store button objects and their associated tags.
- *Iterating Through Fields*: The code iterates through the fields within the input structure myStruct.
- *Button Generation*: For each field, the code checks if the value is a structure that contains a 'ButtonText' field. If so, it extracts the button text and the value PlcVarName, used to fill the tag array.
- *Button Position Calculation*: It calculates the position of the button within the panel, considering factors like panel width and spacing.
- *Button Creation*: Using the calculated position and other parameters, the function creates a Level 3 button (btns_lvl3) with specified attributes like text, position, font style, and callback functions. Here, btns_lvl3 represents the Level 3 button created in each iteration. By appending each newly created button to buttonArray, the vector effectively holds references to all the Level 3 buttons generated. Subsequent, ID represents the value used to create the PLC variable name. The code appends each ID to the tag vector, creating a correspondence between each button and its associated PLC variable.
- *Callback Functions*: Callback functions are defined within the main function. These callbacks are associated with the Start and Execute buttons of the Level 3 process. When these buttons are clicked, they trigger the corresponding PLC actions.

- *Nested Functions*: Two nested functions, startFunction and executeFunction, handle the Start and Execute button actions, respectively. They construct strings to access PLC nodes and write values using the OPC UA client.

These vectors, buttonArray and tag, are essential for tracking and managing the generated buttons and their respective PLC variable names, enabling efficient interaction with the PLC system based on user input

In the case of generating Level 4 buttons, the process is analogous, although with a few notable distinctions:

- *Column-Based Layout*: createButtonsLvl_4 incorporates a more complex layout strategy. It arranges buttons in columns with customizable spacing between columns (columnSpacing), allowing for better organization when a large number of buttons are involved.
- *Maximum Buttons per Column*: createButtonsLvl_4 includes logic to handle situations where the number of buttons in a column exceeds a specified maximum (maxButtonsPerColumn). In such cases, it moves to the next column, enhancing the layout's flexibility.
- *Position Calculation*: The X and Y positions of Level 4 buttons in createButtonsLvl_4 are calculated differently, taking into account the column-based layout and spacing between buttons.

The function statement results as follows:

```
function [tag, buttonArray] = createButtonsLvl_4(app, position, myStruct, spacing, startY, buttonLength, buttonHeight, maxButtonsPerColumn)
```

Subsequently, in the startup function (*startupFcn*) of the Kalandrierzelle's window, we need to invoke the functions to generate all the buttons. This is accomplished as follows:

```
            buttons = [];
            tagArray = [];

            [tag, btn]=createButtonsLvl_3(app,app.TransportkistebereitstellenTab,newStructData.x2_0_0_0_Kalandrieren.x2_1_0_0_Transportkiste_bereitstellen,100,464,208,77);
            for i = 1:numel(btn)
                buttons = [buttons, btn(i)];
                tagArray = [tagArray, tag(i)];
            end
```

```
[tag, btn]=createButtonsLvl_4(app,app.AGVentladenTab,newStructData.x2_0_0_0_Kalandrieren.x2_1_0_0_Transportkiste_bereitstellen.x2_1_1_0_AGV_entladen,60,494,190,40,5);
            for i = 1:numel(btn)
                buttons = [buttons, btn(i)];
                tagArray = [tagArray, tag(i)];
            end
```

The steps carried out by this code are as follows:

- *Initialization of Arrays*: At the beginning of the code, two empty arrays are initialized: buttons and tagArray. These arrays will be used to store the created buttons and their corresponding tags.
- *Calling the createButtonsLvl_3 Function*: A call to the createButtonsLvl_3 function is made with various arguments. This function creates buttons for the level 3 of the user interface and returns two outputs: an array of tags (tag) and an array of button objects (btn).
- *Iteration Through Buttons*: A for loop is initiated to iterate through the button objects in the btn array returned by the createButtonsLvl_3 function.
- *Appending to Arrays*: Within the loop, each button object is appended to the buttons array, and its corresponding tag is appended to the tagArray array.

It's crucial to highlight that in the application the function responsible for generating level 3 buttons is invoked three times. This repetition is due to the existence of precisely three level 2 processes, under each of which we must create the subsequent levels of buttons. Correspondingly, a similar iterative process occurs for the subsequent level, level 4, as well. This approach ensures that the user interface dynamically adapts to the structure of the underlying processes, generating the required buttons for each level.

After successfully initializing the interface, the arrays, namely *'buttons'* and *'tagArray*,' should be populated with references to the buttons and their corresponding tags, which directly align with the process IDs. These arrays will later play a pivotal role within the *'updateButtonColor'* function, responsible for upgrading button color. This function is responsible for dynamically enhancing button colors to reflect specific process states. After the correct initialization of the application and the generation of the buttons, if the connection to the PLCs has been properly initialized, the process should initially be in 'off' state and the visualization of the window should appear as in *Figure* 25.

3.3.3 Pause Switch

As previously noted, positioned in the upper-right corner of the Kalandrierzelle's interface, there is a toggle switch designed to halt or resume the active process. The associated function, called 'PauseSwitchValueChanged,' operates as follows:

```
% Value changed function: PauseSwitch
        function PauseSwitchValueChanged(app, event)
            value = app.PauseSwitch.Value;

            global uaClient;

            if strcmp(value, 'Go')
                string = ['"Prozess_DB_ML"','."Paused"'];
                plcNode = opcuanode(3, string);
                writeValue(uaClient, plcNode, 0);

        elseif strcmp(value, 'Pause')
                string = ['"Prozess_DB_ML"','."Paused"'];
                plcNode = opcuanode(3, string);
                writeValue(uaClient, plcNode, 1);
            end
        end
```

- *Value Retrieval*: The function first retrieves the current value of the 'PauseSwitch' control element. This value can be one of two states: 'Go' or 'Pause.'
- *Global OPC UA Client*: The code interfaces with the global OPC UA Client declared in the startup function, which serves as the communication bridge between the user interface and the underlying automation system.
- *Resuming the Process ('Go' State)*: In the beginning, the OPC UA server features a variable named "Prozess_DB_ML.Paused," which is initially set to 0, representing the 'false' state, as it's a Boolean variable. This configuration allows the process to commence whenever an 'Execute' button is activated. From a technical standpoint, when we are in the 'Go' state, the function generates a string to access the 'Paused' node on the server and changes its value to 0. This action will effectively allow the process to continue once interrupted.
- *Pausing the Process ('Pause' State)*: Conversely, when the 'PauseSwitch' value is 'Pause,' the function again constructs the same string to access the 'Paused' node. However, this time, it writes a value of '1.' This action signals the system to pause the ongoing process.

In essence, this code extract permits operators by providing a straightforward user interface component to control the pausing and resuming of processes within the automation system. This enhances the flexibility and real-time control capabilities of the production environment, contributing to efficient and responsive operations.

3.3.4 Process Simulation

To commence the process or simulate it with the assistance of PLCSIM Advance, the initial step involves pressing the start button to initiate the warming up of all the machinery within the designated process. When the start button associated with a level two process is pressed, it triggers the activation of all machinery, including level three and four apparatuses linked to the selected level two process. For instance, pressing the start button under "*Sheets Kalandrieren*" activates all the relevant machinery within "*Module Vorbereiten*" and "*Module Nachbereiten*," along with their corresponding level four processes. This action results in a transition of all associated buttons to the '*ready'* ('*idle'*) state, as illustrated in *Figure* 26 below.

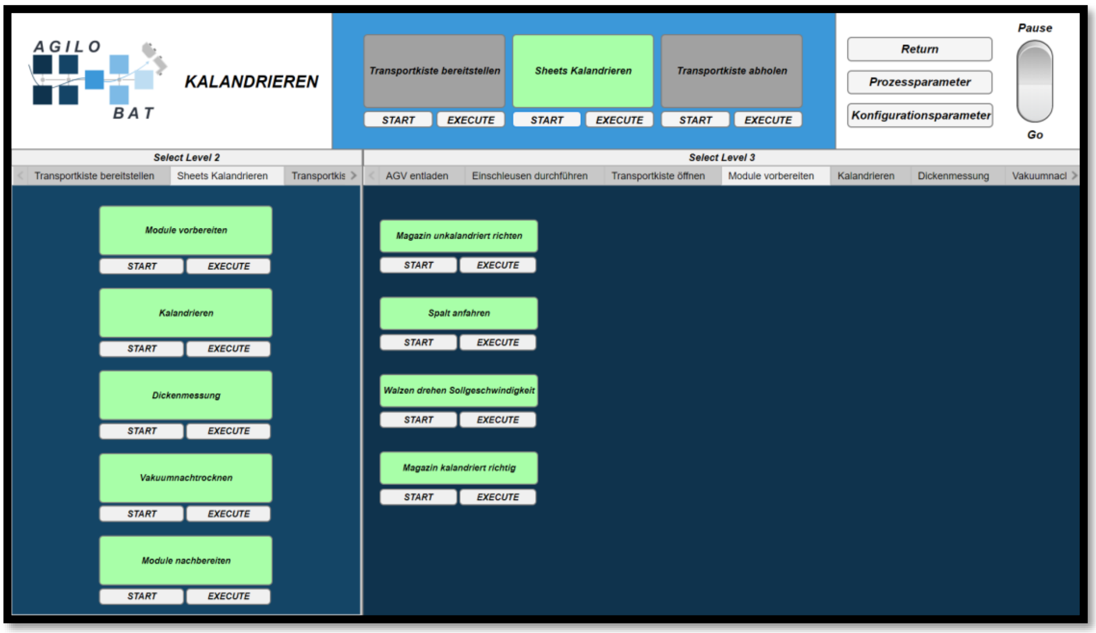


Figure 27: Kalandirerzelle, 'ready' state

Likewise, a similar process unfolds when you interact with buttons corresponding to levels three and four. For instance, when you press a button linked to a level three process, it not only initiates the machinery associated with that level but also activates all the relevant machinery within the corresponding level four processes. This cascade effect ensures that all relevant buttons turn green, indicating the activation status, just as demonstrated earlier. The process is similarly replicated when interacting with level four buttons, ensuring that the entire hierarchical structure of the manufacturing process is synchronized and properly initiated.

After all the necessary components have been activated, the user can proceed with the execution of the manufacturing process itself. This action is initiated by pressing the execute button corresponding to one of the activated processes. When a specific execute button is pressed, it signifies the commencement of the selected process, including all components and processes at the lower hierarchical levels, mirroring the behavior observed when the start button is pressed. This synchronization ensures that the entire manufacturing process, along with its sub-processes, is set in motion seamlessly.

In the two following figures, we can observe two distinct stages of the ongoing process. In both cases, the execution button beneath the *'Sheets Kalandrieren'* process has been pressed. In the first figure, we witness an active level four process under the *'Vakuumnachtrocken'* process, indicated by a blue color. In the second figure, a level four process under *'Module Nachbereiten'* has recently concluded, and the subsequent process is about to commence.

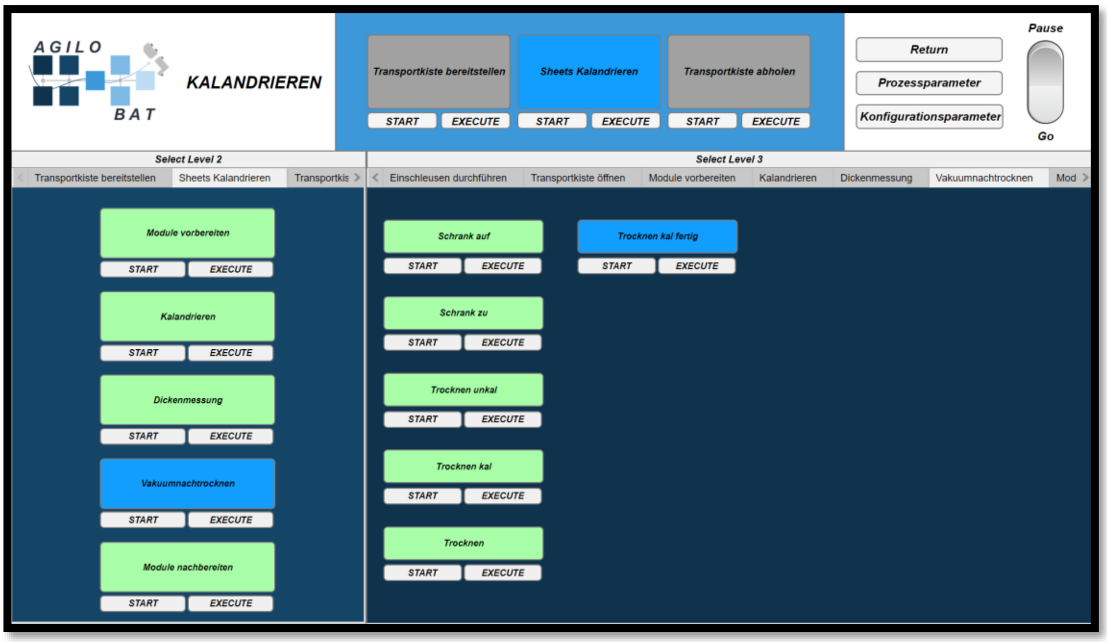


Figure 28: Kalandrierzelle, 'ongoing' state

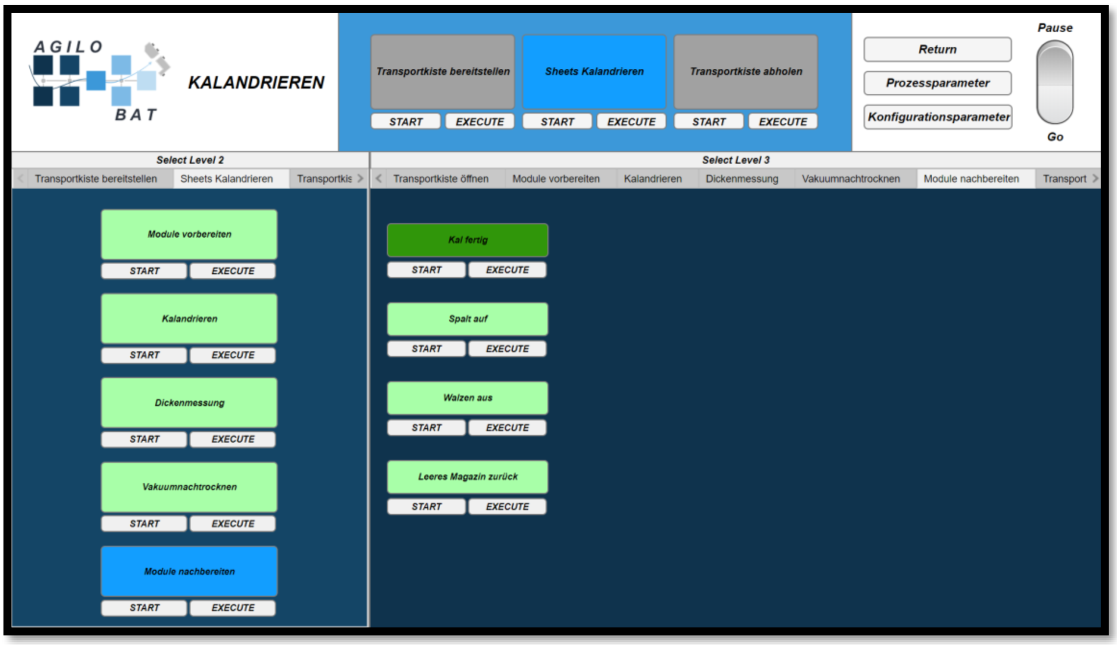


Figure 29: Kalandrierzelle, 'ongoing' state

3.4 Konfigurationsparameter (Configuration Parameter Window)

As previously discussed, users can conveniently access the configuration parameters window from both the Main Window and the Calendering interface. Here, they can navigate through various essential data points, including:

- *HTTP – Endpoint URL*: This denotes the HTTP endpoint linked to the external web service, which in turn connects to the central database.
- *Bearer Token*: This serves as a critical element for authentication and authorization, guaranteeing secure access to the web service.
- *Cell type*: This relates to the specific battery cell under development, providing crucial information about the ongoing process.
- *Four distinct OPC Endpoints*: There are four of these, with each one corresponding to a central PLC within its respective microenvironment.

In this section we focus on the initialization of these critical configuration parameters and user interface (UI) elements within the application. These parameters play a pivotal role in defining the application's behavior and functionality, while the UI elements enable users to interact with and configure the application. Let's investigate into how this code accomplishes these tasks:

```
function startupFcn(app, Main_Window_finalVersion, Kalandrierzelle_FINAL)
            % Get the directory path of the current script.
            appDirectory = fileparts(mfilename('fullpath'));

            % Get the parent directory of the current script.
            parentDirectory = fileparts(appDirectory);

            % Construct the full file path for the JSON configuration file.
            filePath        =        fullfile(parentDirectory,        'configuration        files',
'CellWizard_config_params.json');

            % Read the contents of the JSON file into a string.
            fileContent = fileread(filePath);

            % Decode the JSON content into a MATLAB structure.
            jsonData = jsondecode(fileContent);

            % Populate app EditFields with data from the JSON structure.
            app.EditField.Value = jsonData.Database_Config_Parameters.HTTP_Endpoint_URL;
            app.EditField_2.Value = jsonData.Database_Config_Parameters.Bearer_Token;
            app.EditField_5.Value = jsonData.PLC_Parameters.Endpoint_URLs.Beschichtung;
            app.EditField_6.Value = jsonData.PLC_Parameters.Endpoint_URLs.Kalandrierung;
            app.EditField_3.Value = jsonData.PLC_Parameters.Endpoint_URLs.Vereinzelung;
            app.EditField_4.Value = jsonData.PLC_Parameters.Endpoint_URLs.Assemblierung;

            % Iterate through the Cell_Type_IDs in the JSON data and add them to a uitree.
            for i = 1:numel(jsonData.Database_Config_Parameters.Cell_Type_IDs)
                uitreenode(app.ZelltypIDNode,                                        'Text',
char(jsonData.Database_Config_Parameters.Cell_Type_IDs(i)));
            end
        end

% Button pushed function: RETURNButton
function RETURNButtonPushed(app, event)
    delete(app);
```

```
end
```

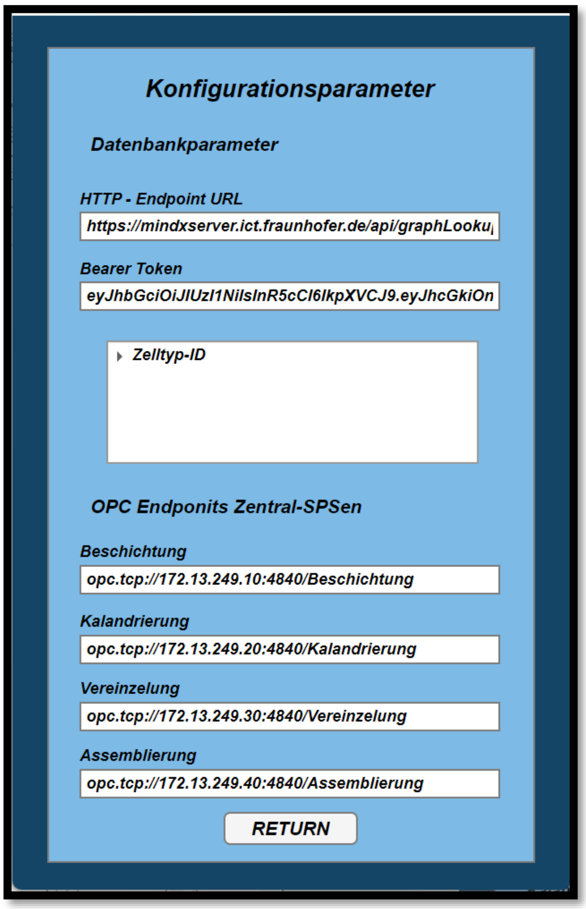
Code Description:

This code serves as the foundation for setting up essential aspects of the application:

- *File Path Retrieval*: It begins by determining the directory paths of the current script and its parent directory. These paths are essential for locating a JSON configuration file within the application's file structure.
- *JSON Configuration Parsing*: The code proceeds to read the contents of the designated JSON configuration file and parses them into a structured MATLAB format using the jsondecode function. This parsed data encapsulates critical parameters, including HTTP endpoint URLs, bearer tokens, and PLC (Programmable Logic Controller) endpoint URLs, associated with different microenvironments.
- *UI Element Population*: With the configuration data now accessible, the code dynamically populates various UI elements, particularly EditField components, with the pertinent values retrieved from the parsed JSON dataset. These EditField elements offer a clear view of the configuration parameters and allow users to make adjustments as needed.
- *Cell Type Selection*: The code also enables users to select specific battery cell types through the uitree UI component named ZelltypIDNode. This functionality is invaluable as it empowers users to tailor the application's operation to suit the characteristics of different battery cell types.
- *User-Friendly Return Button*: Lastly, the code defines the behavior of the 'RETURNButton.' When users engage with this button, it initiates the graceful closure of the application, facilitating a smooth exit.

In essence, this code plays a pivotal role in establishing the groundwork for the application's configuration and interaction, guaranteeing that it starts with the correct settings and provides a user-friendly interface for users to configure and operate the application effectively.

The very first of this code segment is essentially telling MATLAB to execute the "startupFcn" function and pass it three arguments: the application object ("app") and references to two other components or windows within the application ("Main_Window_finalVersion" and "Kalandrierzelle_FINAL"). The following figure shows the layout of this window.



HTTP - Endpoint URL: https://mindxserver.ict.fraunhofer.de/api/graphLooku
Bearer Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhcGkiOn
Beschichtung: opc.tcp://172.13.249.10:4840/Beschichtung
Kalandrierung: opc.tcp://172.13.249.20:4840/Kalandrierung
Vereinzelung: opc.tcp://172.13.249.30:4840/Vereinzelung
Assemblierung: opc.tcp://172.13.249.40:4840/Assemblierung

Figure 30: Konfigurationsparameter window

3.5 Prozessparameter (Process Parameter Window)

Through this window, users gain easy access to an interface for visualizing and modifying the parameters governing the entire process. This functionality plays a pivotal role in granting operators the freedom and autonomy to tailor the process according to their specific requirements. The interface is divided into two halves: on the left-hand side, there's a drop-down menu for selecting the desired cell. Upon selection, a cascading menu appears, enabling users to choose the specific process whose parameters they wish to inspect.

Once a process is chosen, the right side of the interface activates, revealing a panel containing various fields displaying essential information such as the process name, target values, and other crucial variables, like Upper and Lower Limit, for instance, which are also important information to display, as they help the user to ensure that the values they enter are within the acceptable range. This window is depicted in *Figure* 30 below.

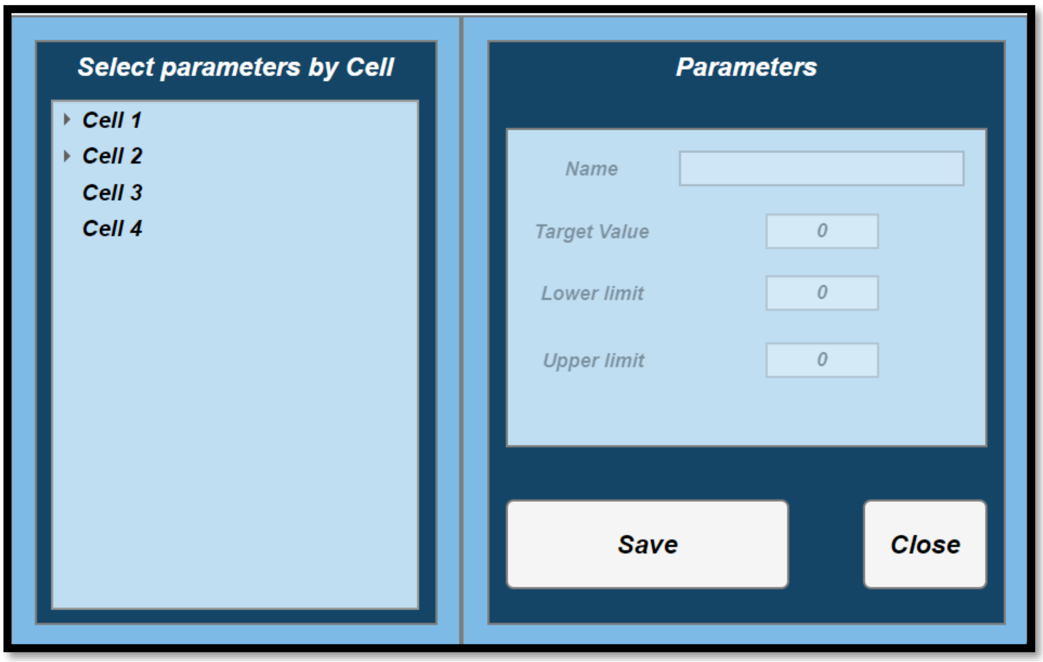


Figure 31: Prozessparameter Window, initial view

3.5.1 Dynamic Tree Node Creation

To establish the dynamic tree node menu, the implementation of a crucial function becomes necessary. This function's role is to extract information from a predefined data structure that contains details about parameters and their associations with specific processes. Hence, the createTreeNode function has been utilized for this purpose:

```
function createTreeNodes(app, parentNode, name, parameters)
        % Function to create tree nodes based on specific criteria

        % Create a child node under the parentNode with the text corresponding to the name
        childTree = uitreenode(parentNode, 'Text', name);

        % Check if the parameters are a cell or a struct
        if iscell(parameters)
            for k = 1:numel(parameters)
                if isstruct(parameters{k})
                    % Create a child node with the text corresponding to the parameter name
                    uitreenode(childTree, 'Text', parameters{k}.name);
                end
            end
        elseif isstruct(parameters)
            % Create a child node with the text corresponding to the parameter name
            uitreenode(childTree, 'Text', parameters.name);
        end
end
```

The purpose of this function is to dynamically populate a tree-like structure with nodes based on specific criteria. Let's break down the code's functionality in detail:

Function Parameters:

- *app*: This parameter represents the GUI object where the tree nodes will be created.
- *parentNode*: It denotes the parent node under which child nodes will be added.
- *name*: This parameter is a string representing the text label for the newly created child node.
- *parameters*: It is the criteria or data based on which child nodes will be generated. It can be either a cell array or a struct.

Creating Child Nodes:

- The function starts by creating a child node (‘*childTree*’) under the specified ‘*parentNode*’. The text label for this child node is set to the value of the ‘name’ parameter.

Checking Data Type:

- The code then checks the data type of the ‘parameters’ variable to determine how to proceed with creating child nodes.
- If ‘parameters’ is a cell array, the code enters a loop to process each element of the array.
- Within the loop, it checks if the single element is a struct.
    - If it's a struct, a child node is created under ‘*childTree*’ with the text label corresponding to the ‘name’ field within that struct.
- If ‘parameters’ is a struct (not within a cell array), the code directly creates a child node under ‘*childTree*’ using the text label specified by the ‘name’ field within that struct.

Within the domain of graphical user interface (GUI) development, the *createTreeNodes* function assumes a pivotal role in dynamically constructing the interface's tree structure. This function is invoked as part of this window’s startup function, which springs into action during the interface's initialization phase. When the GUI is launched, the *createTreeNodes* function is summoned to orchestrate the creation and population of tree nodes within the interface. These tree nodes hold significant importance as they serve as the backbone for organizing and presenting crucial information to users. They facilitate seamless navigation and enable users to tailor various parameters and processes to their specific needs.

In the startup function, similar to the setup observed in both the Kalandrierzelle's interface and the Main Window, we must ensure the proper integration of OPC UA communication. This involves establishing an OPC UA client, which connects to the specified OPC UA server, a concept thoroughly clarified in Chapter 3.2.1. The only divergence lies in the fact that, at this moment, the data structure obtained from the database assumes the role of a *global* variable. Below, we present the code extract from the startup function, showcasing the invocation of the *createTreeNodes* function:

```
for i = 1:numel(response.processChainB)
                name = response.processChainB(i).name;
                parameters = response.processChainB(i).parameters;

                if name(1) == '1'
                    createTreeNodes(app, app.Cell1Node, name, parameters);
                elseif name(1) == '2'
                    createTreeNodes(app, app.Cell2Node, name, parameters);
                elseif name(1) == '3'
                    createTreeNodes(app, app.Cell3Node, name, parameters);
                elseif name(1) == '4'
                    createTreeNodes(app, app.Cell4Node, name, parameters);
                end
end
```

The code iterates through the elements in ‘*response.processChainB*’, the data structure containing information about different processes and parameters, obtained by the database. For each element in this array, it extracts the ‘*name*’ (process name) and ‘*parameters*’ variable associated with the process.

Subsequently, based on the extracted process ‘*name*’, the code categorizes processes into different groups. It creates dynamic tree nodes within the interface, organizing processes by category. For instance, processes with names starting with '1' are associated with ‘app.Cell1Node’, '2' with ‘app.Cell2Node’, and so on, which are the four main nodes, one for each microenvironment. The ‘*createTreeNodes*’ function is called to populate the relevant tree nodes with the process information.

In summary, this code segment, in collaboration with the ‘*createTreeNodes*’ function gets data, processes it, and dynamically populates tree nodes within the interface based on the received data. This dynamic organization of processes enhances user interaction and customization within the application.

After the window has been properly initialized, the phase of generating dynamic tree nodes should be completed, granting the user the freedom to navigate through all the processes along with their associated parameters. The view should be as in the figure below:

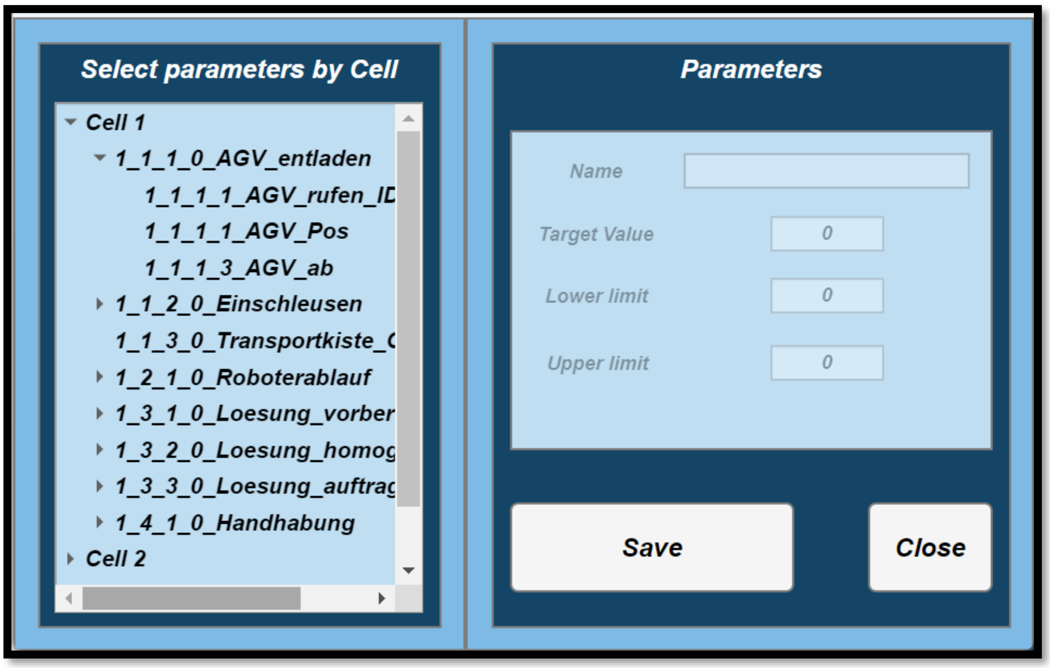


Figure 32: Prozessparameter, initialized view

3.5.2 Interactive User Interface and Parameter Selection

In this section it will be describe what happens when the operator select the desired process whose value he wants to investigate. For the purpose, a function called TreeSelectionChanged has been implemented as shown below:

```
% Selection changed function: Tree
        function TreeSelectionChanged(app, event)
            % Enable form only when a Process with parameters is selected
            % Cell array containing name fo the cells
            Cell = {'Cell 1','Cell 2','Cell 3','Cell 4'};

            % Declaration of the struct "response" as a global variable
            global response;

            % Initialize a cell array 'process' with empty cells based on
            % the number of elements in the struct 'response.processChainB'
            process = cell(1, numel(response.processChainB));

            % Loop through each element in 'response.processChainB'
            for i = 1:numel(response.processChainB)
                % List of processes, in this case level 3
                process{i} = response.processChainB(i).name;
            end

            % Check if the selected node's text is not found in 'process' and 'Cell'
            if isempty(find(strcmp(app.Tree.SelectedNodes.Text, process), 1)) &&
isempty(find(strcmp(app.Tree.SelectedNodes.Text, Cell), 1))
                % Enable the app form
                enableForm(app);

                % Set the value of 'NameEditField' to the selected node's text
                app.NameEditField.Value = char(extractText(app, app.Tree.SelectedNodes.Text));

                % Set the value of 'TargetValueEditField' using the 'findValues' function
                app.TargetValueEditField.Value = findValues(app, response.processChainB,
app.Tree.SelectedNodes.Text, 'targetValue');
            end
        end
```

Code description:

- *Form Activation Condition*: When a user interacts with the tree component in the MATLAB app, the code is triggered as a result of a change in the selection. Its primary purpose is to determine whether the application form should be enabled based on the user's selection.
- *Cell Array Initialization*: A cell array named ‘Cell’ is defined. This array contains the names of different cells, namely 'Cell 1', 'Cell 2', 'Cell 3', and 'Cell 4'. These are the names of the four main parent node on the Tree Node Menu.
- *Global Variable*: The code declares the global variable named ‘response’. This variable holds important data and has already been declared as a global variable in the startup function, as previously explained.
- *Initializing a Cell Array*: A cell array named ‘process’ is initialized. The size of this array is determined by the number of elements in ‘*response.processChainB*’. This because it's meant to store data related to processes.

- *Loop through Process Data*: The code iterates through each element in the '*response.processChainB'*. For each element, the code extracts the name of a process (indicated by 'name') and stores it in the 'process' cell array. This indicates that process will contain the names of level 3 processes.
- *Selection Check*: The code checks whether the text of the selected node (within the tree component) is not found in either the 'process' cell array or the predefined 'Cell' array. In other words, it's ensuring that the selected node represents a process with parameters, not one of the predefined cell names.
- *Form Activation*: If the selected node represents a process with parameters, the application form is enabled using the '*enableForm'* function.
- *Field Value Assignment*: The values of two specific fields in the app, namely '*NameEditField'* and '*TargetValueEditField'*, are set based on the selected node. The '*NameEditField'* is set to the name of the selected node, and the '*TargetValueEditField'* is populated with data retrieved from the '*response.processChainB'* based on the selected node's text.

In summary, this code dynamically enables or disables the application form and updates certain fields within the app based on the user's selection of a node in the tree structure. It ensures that the user can interact with the form only when a process with parameters is selected.

3.5.3 Data Extraction and Value Retrieval Technique

The contents of '*NameEditField'* and '*TargetValueEditField'*, discussed above, are populated using two dedicated functions: '*extractText'* and '*findValues'*. While the latter function is currently employed solely for the Target Value, it's designed for potential application to other types of values within the data structure in the future. Here's an overview of how these two functions operate:

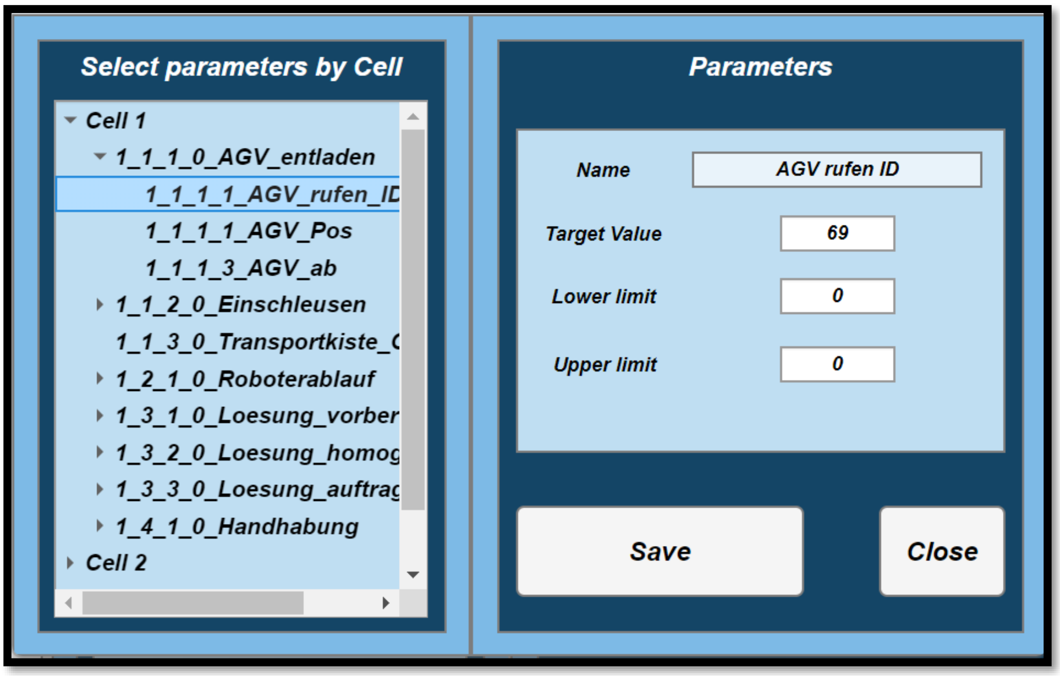
- *'extractText':*
  - *Pattern Definition*: It starts by defining a regular expression pattern. In this case, the pattern is set to '\d+_\d+_\d+_\d+_(\D+)'. This pattern looks for sequences of digits separated by underscores, followed by a sequence of non-digits (indicated by \D+). The parentheses (\D+) are used to capture the non-digit sequence. This functionality has been implemented to provide the interface a more readable aspect since in the data structure the names of the parameter were written using underscores.
  - *Matching*: The '*regexp'* function is then used to search for matches of the pattern within the input string. The *'tokens'* option indicates that we want to extract the captured portions of the text, and 'once' specifies that only the first match should be considered.
  - *Check for Matches*: The code checks if any matches were found. If matches exist, it proceeds with further processing. If no matches are found, the string will be left empty.
  - *Extraction and Formatting*: If there is a match, it extracts the captured text. This text may contain underscores, so it uses '*strrep'* to replace underscores with spaces to make the text more readable.

In summary, the '*extractText'* function searches for a specific pattern in the input string and extracts a portion of text that matches this pattern. It then formats the extracted text for readability by replacing underscores with spaces.

- *'findValues':*
  - Initialization: It initializes the 'outputValue' variable as an empty array.
  - Iteration: The function iterates through the elements inside the data structure. For each element, it checks if the parameters field is either a cell or a struct. This is done to handle different possible structures within the parameters.
  - Nested Iteration: If the parameters is a cell, it further iterates through its elements. For each element (referred to as param), it checks if it's a struct and if the name field of the struct matches the inputName provided as an argument.
  - Field Check: If the name of the selected node on the Tree Menu matches and the target field desired, Target Value in this case, is present within the structure, it retrieves the value of the field. The value is assumed to be a string, so it's converted to a numeric value using str2num.
  - Break Loop: If the desired value is found, the loop is broken early to improve efficiency since there's no need to continue searching.
  - Final Result: After the loop, the edit field will either contain the desired value if it was found, or it will remain empty if no match was found in the data structure.

The '*findValues'* function allows you to search and extract values from the complex nested structure by specifying the parameter name and the target field to retrieve. It handles different data structures within the parameters field, such as cells and structs, to find and return the requested value.

After the user has chosen the desired parameter on the interface, and the 'TreeSelectionChanged' function has been successfully executed, the interface will resemble the layout depicted in *Figure* 32.

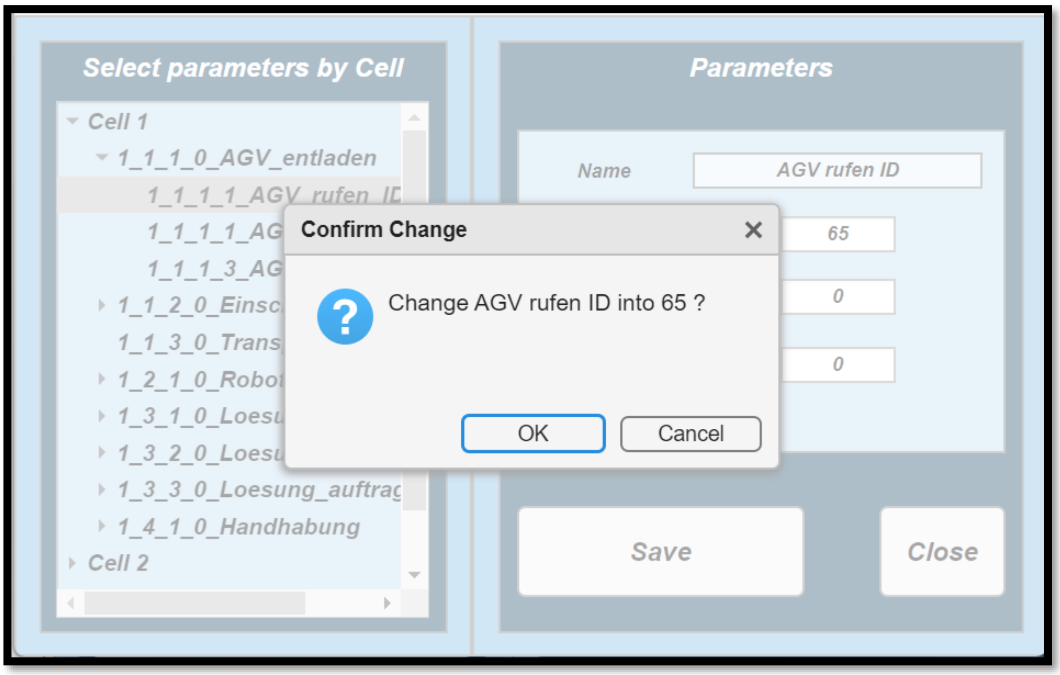


Name	AGV rufen ID
Target Value	69
Lower limit	0
Upper limit	0

Figure 33: Prozessparameter, parameter selection

3.5.4 Real-time Parameter Modification

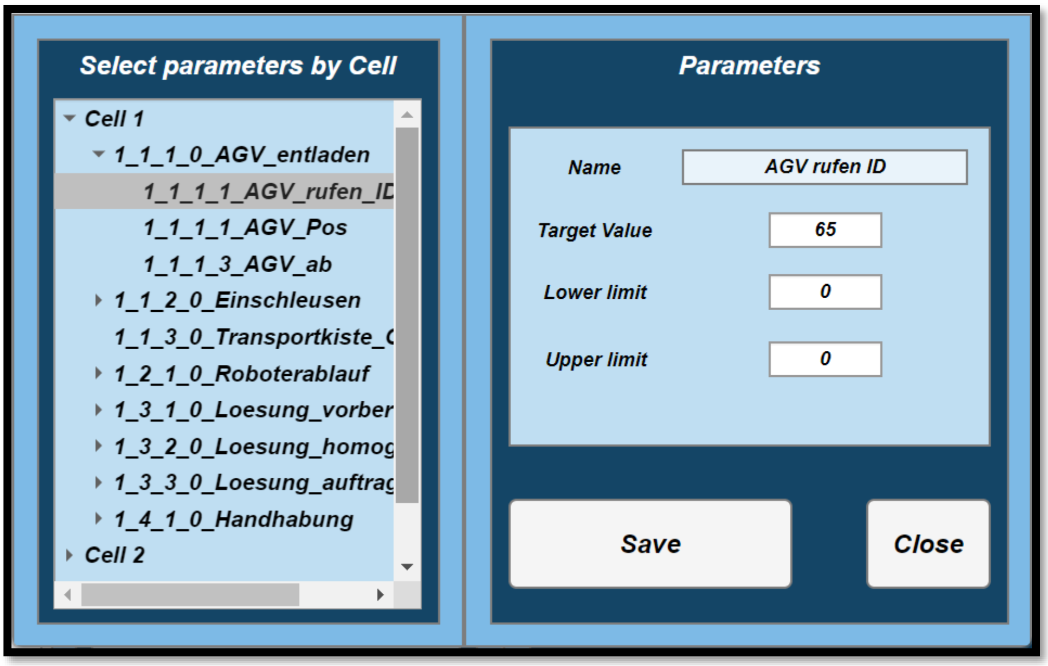
At this juncture, the user possesses the ability to modify the 'Target Value' for the selected parameter by directly inputting an alternative value into the designated edit field. This feature facilitates adjustments to the final outcome of the ongoing process, ensuring it aligns with customer requirements and development an agile production method. Once the new value has been entered into the respective field, upon pressing the 'Save' button located at the bottom left, an alert message will be triggered, as illustrated in *Figure* 33. This alert serves as a confirmation step, inquiring if the proposed value for the corresponding parameter is accurate. If confirmed, the newly input value is updated within a fresh version of the data structure, and the interface promptly displays the revised value within the edit field. This responsive and user-centric approach empowers the operator with real-time control over key process parameters.



Confirm Change

Change AGV rufen ID into 65 ?

Figure 34: Prozessparameter, Alert Message



Name	AGV rufen ID
Target Value	65
Lower limit	0
Upper limit	0

Figure 35: Prozessparameter, updated parameter

In *Figure* 34, the modified value is displayed within the designated edit field.

As previously mentioned, it's essential to make updates to the data structure without affecting the primary database. To address this requirement, a dedicated piece of code has been developed. This code possesses the capability to modify the structure solely within the interface's workspace and provides the user with the ability to view its contents whenever they request information about a specific altered parameter. This function has been aptly named "*updateParameters*."

In its core functionality, the "*updateParameters*" function empowers users to make precise adjustments to particular parameters within the 'response' data structure. It proficiently crosses the data structure, effectively locates the pertinent parameter, and proceeds to adjust its *'targetValue'* with the freshly provided value as an input to the function.

This function exhibits a high degree of adaptability, accommodating parameters organized as either cell arrays or structs within the data structure. It plays a pivotal role in enabling users to tailor and fine-tune the parameters governing various aspects of the ongoing processes directly within the interface.

Now, let's delve deeper into the intricacies of how this function operates.

```
        % Define a function to update parameters in a data structure
        % based on input name and target field
        function updateParameters(app, inputName, targetField, value)
            % Declaration of the struct "response" as global variable
            global response;
            % Iterate through the elements of 'response.processChainB'
            for i = 1:numel(response.processChainB)
                % Extract the 'parameters' field for the current element
                parameters = response.processChainB(i).parameters;

                % Check if the 'parameters' field is a cell or a struct
                if iscell(parameters)
                    % If it's a cell, iterate through its elements
                    for j = 1:numel(parameters)
                        param = parameters{j};

                        % Check if the parameter is a struct and its 'name' matches
'inputName'
                        if isstruct(param) && strcmp(param.name, inputName)
                            % Check if the specified 'targetField' exists in the structure
                            if isfield(param, targetField)
                                % Update the 'targetValue' field with the new 'value'
                                param.targetValue = num2str(value);
                                % Update the 'parameters' cell with the modified parameter
                                response.processChainB(i).parameters{j} = param;
                                % Exit the loop since we've found and updated the parameter
                                break;
                            end
                        end
                    end
                elseif isstruct(parameters)
                    % If it's a struct, directly check the parameter's 'name'
                    param = parameters;

                    % Check if the parameter's 'name' matches 'inputName'
                    if strcmp(param.name, inputName)
                        % Check if the specified 'targetField' exists in the structure
                        if isfield(param, targetField)
                            % Update the 'targetValue' field with the new 'value'
                            param.targetValue = num2str(value);
                            % Update the 'parameters' struct with the modified parameter
                            response.processChainB(i).parameters = param;
                            % Exit the loop since we've found and updated the parameter
                            break;
                        end
                end
            end
        end
    end
```

- It begins by calling the global variable ‘*response*’. This variable represents the data structure that contains information about parameters and processes.
- The function then iterates through the elements of ‘*response.processChainB’* appears to be a field within the response structure that holds an array of elements. Each element represents a process.
- Within the loop, it extracts the *'parameters'* field for the current element. The *'parameters'* field contains information about the parameters associated with the process.
- It checks if the *'parameters'* field is a cell or a struct. This suggests that parameters can be organized in different ways: either as a cell array or a struct, depending on the process.
- If 'parameters' is a cell, it further iterates through its elements, which represent individual parameters. For each parameter, it checks if the parameter is a struct and if its *'name'* matches the *'inputName'* provided as a parameter to the function; this variable represents name of the parameter designated.
- If it finds a matching parameter, it checks if the specified *'targetField'* exists within the parameter's structure.
- If *'targetField'* exists, the function updates the *'targetValue'* field of the parameter with the new 'value'. The *'targetValue'* is modified with the new value as a string representation.
- The updated parameter is then placed back into the 'parameters' cell in its original position, effectively updating the parameter within the data structure.
- The loop exits since the parameter has been found and updated.
- If 'parameters' is a struct, the function directly checks the parameter's *'name'* and follows the same procedure as described above, with the modification taking place in the *'parameters'* struct.

This function is called every time the user press the “*OK*” button on the Alert Message discussed above.

Conversely, when the operator opts to press the "*Cancel*" button upon detecting an incorrect value, the system refrains from inserting the parameter, thereby preventing the invocation of the “*update Parameters*” function. Consequently, the data structure remains unaltered, ensuring data integrity and operational consistency. In such cases, the system's responsiveness to erroneous inputs safeguards against unintended modifications, reinforcing the robustness and reliability of the control system.

4 Results

The implemented user interface encompasses two primary components: the "Kalandrierzelle" interface and the Main Window. Each interface provides unique functionalities for monitoring and controlling manufacturing processes.

Kalandrierzelle Interface:

- The Kalandrierzelle interface is the centerpiece of our project, designed to empower users with real-time process monitoring and parameter customization.
- The "updateButtonColor" function has been successfully integrated into this interface. It dynamically updates button colors, providing users with a clear visual representation of the ongoing processes. This is a fundamental feature for process monitoring.

Main Window:

- The Main Window acts as the primary control center, providing access to various parameters and configuration settings.

Other two window have been developed and successfully implemented in the application. These two functionalities concern the parameter customization and the configuration parameter in a robust and efficient way.

Konfigurationsparameter (Configuration Parameter Window):

- In which the "*updateParameters*" function further enhances the flexibility of our system by allowing users to modify specific parameters directly. This feature was tested extensively during user evaluations.

Prozessparameter (Process Parameter Window):

- Through the "createTreeNodes" function, this interface presents a dynamic tree structure to navigate through the processes and their associated parameters, offering a user-friendly approach to customization.

The results of our tests were positive and revealed the following key findings:

- User-Friendly Interface: key aspect of our application, with intuitive controls for process monitoring and parameter customization.
- Clear Process Visualization: especially important the "updateButtonColor" function, as it offered a quick and efficient way to assess the progress of manufacturing processes.
- Parameter Customization: The "updateParameters" function was deemed highly beneficial, allowing users to adapt parameters in real-time, thereby enhancing process agility.

While the overall results were positive, we encountered several challenges during the development process. These challenges primarily revolved around the integration of the OPC UA Communication for Real-time data exchange and providing the users a visual interface for the process monitoring.

5 Assessment

Our project's primary objective was to design and implement a dynamic interface for monitoring and controlling processes within a manufacturing environment. In this regard, we can confidently conclude that our goals have been met. The "Kalandrierzelle" and Main Window interfaces provide users with a real-time visualization of the manufacturing processes, enhancing their understanding and control. In addition an interface to provide easy and rapid modification of process parameter has been developed and implemented for the AgiloBat project.

From a technical standpoint, the implemented interface has demonstrated robustness and reliability. The "updateButtonColor" function effectively updates the button colors, providing a clear visual indication of the progress of each process. The flexibility to handle parameters organized as cell arrays or structs adds a layer of adaptability to our system. We met different technical challenges during the development, such as interfacing with external databases and maintaining real-time communication with PLCs. However, through careful design and effective coding, these challenges were successfully overcome.

Our project holds significant importance in the field of process automation. By providing an adaptable and flexible interface, it contributes to improving the efficiency and agility of production plants. In particular, the "Kalandrierzelle" interface's role in parameter customization has the potential to transform how manufacturing processes are customized and adjusted to meet specific production requirements.

6 Summary and Outlook

6.1 Summary

In the course of this thesis, we embarked on a journey to develop a sophisticated Graphical User Interface that empowers users to seamlessly customize and control critical process parameters within an industrial context. The central focus of our exploration revolved around the Kalandrierzelle (more specifically, on the hardware available during the writing of this thesis), an integral component of our manufacturing process, and the Main Window that served as the gateway to our GUI. Below, we summarize our key findings

The inception of our journey involved establishing secure connections with the OPC UA server, paving the way for real-time data access. We harnessed the potential of the '*uaClient*' variable as a global entity, ensuring a one-time establishment of this connection, thereby eliminating the need for repetitive setup.

Our initial challenge revolved around enhancing the user's comprehension of the ongoing process by providing a clear indication of the specific phase within the process. To address this challenge effectively, we conceived and successfully incorporated the 'updateButtonColor' function into the application. This function adeptly manages the dynamic color changes of process-named buttons, as delineated in Chapter 3.2.3.

Subsequently, our focus shifted towards augmenting the agility of our production plant by enabling the modification of process parameters. To this end, we meticulously designed an intuitive interface, empowering users to navigate through diverse values and make necessary adjustments.

Moreover, in recognition of the importance of verifying configuration parameters for the distinct cells, we introduced a dedicated window. This interface accommodates essential data, including the Database HTTP Endpoint URL, the imperative Bearer Token for secure database connectivity, the specific cell type, and access to four unique OPC Endpoints, each associated with an individual Programmable Logic Controller (PLC) corresponding to a specific working cell.

The Kalandrierzelle interface took center stage as we concentrated on facilitating parameter customization. This specialized interface incorporates buttons pertinent to process parameters, with the 'updateButtonColor' function thoughtfully integrated. Notably, the same set of buttons for both process and configuration parameters was thoughtfully integrated into both the Kalandrierzelle's interface and the Main Window.

6.2 Outlook

As we contemplate the future development of our GUI, specific areas of focus emerge, particularly in the context of the Kalandrierzelle and the Main Window. These crucial components serve as the digital nerve centers of our agile manufacturing system. Here, we envision a profound transformation with the integration of advanced control algorithms, which will not only enhance operational efficiency but also drive a holistic approach to manufacturing.

The integration of advanced control algorithms within the Kalandrierzelle interface opens a gateway to a new era of optimized and automated control. This technological advancement promises to uphold our commitment to ensuring consistent high-quality production. Through the intelligent utilization of these algorithms, the manufacturing process can seamlessly adapt to evolving needs, fostering a higher level of agility and flexibility. This adaptability ensures that we remain responsive to dynamic customer requests, thus strengthening our position in a competitive market.

List of Figures

List of Tables

References

Alphonsus, E. R., & Abdullah, M. O. (2016). A review on the applications of programmable logic controllers (PLCs). *Renewable and Sustainable Energy Reviews*, *60*, 1185–1205. https://doi.org/10.1016/j.rser.2016.01.025

Degele, N. (2007). Neue Kompetenzen Im Internet Kommunikation Abwehren, Information Vermeiden. In *Die Google-Gesellschaft* (pp. 61–74). transcript Verlag. https://doi.org/10.14361/9783839407806-007

Diaconescu, E., & Spirleanu, C. (2012). Communication solution for industrial control applications with multi-agents using OPC servers. In *2012 International Conference on Applied and Theoretical Electricity (ICATE)* (pp. 1–6). IEEE. https://doi.org/10.1109/ICATE.2012.6403431

Energy.gov. (2023, September 4). *How Lithium-ion Batteries Work*. https://www.energy.gov/energysaver/articles/how-lithium-ion-batteries-work

Felser, M. (2001). Ethernet TCP/IP in automation: a short introduction to real-time requirements. In *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.01TH8597)* (pp. 501–504). IEEE. https://doi.org/10.1109/ETFA.2001.997724

Fleischer, J., Fraider, F., Kößler, F., Mayer, D., & Wirth, F. (2022). Agile Production Systems for Electric Mobility. *Procedia CIRP*, *107*, 1251–1256. https://doi.org/10.1016/j.procir.2022.05.140

Fourth Industrial Revolution - Wikipedia. (2023, August 1). https://en.wikipedia.org/wiki/Fourth_Industrial_Revolution.

Frauenhofer ISI. (April 2022). *Solid-State Battery Roadmap 2035+*.

González, I., Calderón, A. J., Figueiredo, J., & Sousa, J. M. C. (2019). A Literature Survey on Open Platform Communications (OPC) Applied to Advanced Industrial Environments. *Electronics*, *8*(5), 510. https://doi.org/10.3390/electronics8050510

IEA. (2023, August 31). *Trends in electric light-duty vehicles – Global EV Outlook 2023 – Analysis - IEA*. https://www.iea.org/reports/global-ev-outlook-2023/trends-in-electric-light-duty-vehicles

Is it really the end of internal combustion engines and petroleum in transport? (2023, August 2).

Karlsruher Institut fuer Technologie. (2023, August 27). *KIT - KIT - Media - Press Releases - Archive Press Releases - AgiloBat: Flexible Production of Battery Cells*. https://www.kit.edu/kit/pi_2020_012_agilobat-batteriezellen-flexibel-produzieren.php

Lydia Dorrmann, Dr. Kerstin Sann-Ferro, Patrick Heininger & Dr. Jochen Mähliß. *Kompendium: Li-Ionen-Batterien: Grundlagen, Merkmale, Gesetze und Normen*.

luetgering, a. 20170601_Agenda Simulationsworkshop.

Microsoft Word - Seminararbeit_SimonGese.docx.

Mustermann, M. (2020). *Toller Titel*. Karlsruhe.

Netto, R., (2013), Programmable Logic Controllers.

Networking. (2023, September 14). *What is Transmission Control Protocol (TCP)? Definition from SearchNetworking*. https://www.techtarget.com/searchnetworking/definition/TCP

Noack, J., Roznyatovskaya, N., Herr, T., & Fischer, P. (2015). The Chemistry of Redox-Flow Batteries. *Angewandte Chemie (International Ed. In English)*, *54*(34), 9776–9809. https://doi.org/10.1002/anie.201410823

OPC Blog. (2018). *What is OPC? Learn about the most used technology in automation*. https://integrationobjects.com/blog/what-is-opc-most-used-technology-automation-world/

OPC Foundation. (2017, June 15). *What is OPC? - OPC Foundation*. https://opcfoundation.org/about/what-is-opc/

Rinke, A. (2022). *What is OPC UA? A practical introduction*. https://www.opc-router.com/what-is-opc-ua/

Sanchez, L. M., & Nagi, R. (2001). A review of agile manufacturing systems. *International Journal of Production Research*, *39*(16), 3561–3600. https://doi.org/10.1080/00207540110068790

Schleipen, M., Gilani, S.-S., Bischoff, T., & Pfrommer, J. (2016). OPC UA & Industrie 4.0 - Enabling Technology with High Diversity and Variability. *Procedia CIRP*, *57*, 315–320. https://doi.org/10.1016/j.procir.2016.11.055

Schmidt-Rohr, K. (2018). How Batteries Store and Release Energy: Explaining Basic Electrochemistry. *Journal of Chemical Education*, *95*(10), 1801–1810. https://doi.org/10.1021/acs.jchemed.8b00479

Science review of internal combustion engines. (2023, August 2).

Siemens. (05/2021). *STEP 7 und WinCC Engineering V17: Systemhandbuch*.

Simon Gese. Agile Produktionssysteme – Grundlagen, Konzeption und Diskussion, *2021*.

VDMA. *Batterieproduktion Roadmap 2023*.

What is Industry 4.0 and how does it work? | IBM. (2023, August 3). https://www.ibm.com/topics/industry-4-0

Appendix

In this appendix, we include a sample section of code used to perform an HTTP request and process the response data, code segment used in the Main Window and in the "*Kalandrierzelle*" interface. This code was employed as a part of the data retrieval process for our research project. It demonstrates how we interacted with a remote server and obtained data that is integral to our study. While the specific details and variables in the code may vary, this example offers insight into the approach we adopted for accessing external data sources.

```
        % Code that executes after component creation
        function startupFcn(app, Main_Window_finalVersion, Kalandrierzelle_FINAL)
            import matlab.net.*
            import matlab.net.http.*
            httpEndpoint = 'https://mindxserver.ict.fraunhofer.de/api/graphLookup';
            bearerToken                                                              =
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhcGkiOnRydWUsInVzZXJJZCI6IjYyZmIyZTM4ZmQ4ODI4NDY2N
mVhYjA0ZiIsIm9yZ2FuaXphdGlvbiI6ImljdEV4dCIsImlhdCI6MTY2MDYyODUzNiwiZXhwIjoxNzIzNzQzNzM2fQ.o
QahcHqievqV4OHIU3SWiRRkvl0_dOt_-XLlmzCP-P8';
            % UA specs
            uaEndpoint = 'opc.tcp://192.168.0.1:4840';
            f1 = matlab.net.http.HeaderField('Authorization', "Bearer " + bearerToken);
            type1 = matlab.net.http.MediaType('text/*');
            type2 = matlab.net.http.MediaType('text/plain','q','.5');
            acceptField = matlab.net.http.field.AcceptField([type1 type2]);
            f2 = matlab.net.http.HeaderField('acceptField', acceptField);
            header = [f1 f2];
            msgBody = '{"isKIT": "64a567770678361c6b52e8d9"}'; % new
            msgBody = jsondecode(msgBody);
            msgBody = matlab.net.http.MessageBody(msgBody);
            body = msgBody;
            method = matlab.net.http.RequestMethod.POST;


            request  =  matlab.net.http.RequestMessage(method,  header,  body);  %  create  a
request message
            uri = URI(httpEndpoint);
            resp = send(request, uri);
            status = resp.StatusCode;
            disp("HTTP Server Status Code: " + status);

            response = resp.Body.Data;
```

The code is explained in details in Chapter 3.2.1.