Aalto University
School of Electrical
Engineering

Master's Programme in Electronics and Electrical Engineering

Accelerating the commissioning of PLCs using 3D simulations

Sampo Vänninen

Master's Thesis
2024

Aalto University
School of Electrical
Engineering

Author Sampo Vänninen

Title Accelerating the commissioning of PLCs using 3D simulations

Degree programme Electronics and Electrical Engineering

Major Control, Robotics and Autonomous Systems

Supervisor Prof. Valeriy Vyatkin

Advisor Fernando Ubis

Collaborative partner Visual Components Oy

Date 17 September 2024 **Number of pages** 60 **Language** English

Abstract

This thesis investigates the feasibility of integrating an IEC 61499 runtime into a simulation environment. The goal was to improve the support for virtual commissioning of PLCs in Visual Components simulation software. A literature review of PLC programming using virtual commissioning was conducted, which showed a lack of research on the overall implementation details of IEC 61499 runtimes. To accomplish the research goals, a new architecture is introduced which details a runtime capable of running in single-threaded constrained environments. The architecture specification is provided in a language-agnostic way, allowing the implementation in any object-oriented programming language, while being a lightweight implementation of the core features of the IEC 61499 standard. To test the architecture, a proof of concept was developed using Python and tested using an existing IEC 61499 digital twin. Capability to control the digital twin without external software using the IEC 61499 application developed in a commercial IDE was demonstrated, validating the design of the architecture. Benefits and downsides of the runtime integration approach were discovered and are discussed.

Keywords IEC 61499, Virtual Commissioning, 3D Simulation Software, PLC Runtime

Aalto-yliopisto
Sähkötekniikan
korkeakoulu

Tekijä Sampo Vänninen

Työn nimi PLC-ohjelmoinnin käyttöönoton nopeuttaminen 3D-simulaatioiden avulla

Koulutusohjelma Elektroniikka ja sähkötekniikka

Pääaine Control, Robotics and Autonomous Systems

Työn valvoja Prof. Valeriy Vyatkin

Työn ohjaaja Fernando Ubis

Yhteistyötaho Visual Components Oy

Päivämäärä 17.9.2024 **Sivumäärä** 60 **Kieli** englanti

Tiivistelmä

Tämä maisterityö tutkii IEC 61499 yhteensopivan ajonaikaisen ympäristön sisällyttämistä simulaatio-ohjelmistoon. Tavoitteena on parantaa PLC:eiden virtuaalista käyttöönottoa hyödyntäen Visual Components -simulaatiohjelmistoa. Tavoitteen saavuttamiseksi työ kuvaa uuden arkkitehtuurin joka mallintaa ajonaikaisen ympäristön mikä on suunniteltu toimimaan vähällä määrällä resursseja yksiytimisissä ympäristöissä. Arkkitehtuuri on kuvattu yleisellä tasolla, mahdollistaen toteutuksen useimmilla oliopohjaisilla ohjelmointikielillä. Arkkitehtuuri testattiin toteuttamalla se Python-kielellä ja valmiilla simulaatiomallilla, joka käytti IEC 61499-standardilla toteutettua ohjelmaa ohjaukseen. Arkkitehtuuri mahdollisti saman ohjausohjelman ajamisen simulaation sisällä, kopioiden simulaatiomallin toiminnan ilman erillistä PLC-ympäristöä. Ajoympäristön sisällyttämisestä simulaatioympäristöön löytyi sekä hyötyjä että haittoja.

Kirjallisuuskatsaus PLC-ohjelmoinnista hyödyntäen virtuaalista käyttöönottoa oli osa maisterityötä, joka toi ilmi vähäisen määrän tutkimustuloksia liittyen IEC 61499 ajoympäristöjen toteutusmenetelmiin. Tässä työssä kuvattu arkkitehtuuri toimii viitteenä toteuttamaan IEC 61499 standardin perusominaisuuksia yksinkertaisella arkkitehtuurilla, josta voi olla hyötyä ajoympäristöjen toiminnan opettamisessa sekä tutkimisessa.

Avainsanat IEC 61499, Virtuaalinen Käyttöönotto, 3D Simulaatio, Runtime

Preface

This master's thesis was written while working at Visual Components Oy, and was partially funded through the European Commission's Horizon Europe Innovation Actions programs Zero-SWARM (GA 101057083) and MODUL4R (GA 101091859). Academic examination of the thesis was carried out in the Department of Electric Engineering at Aalto University.

I would like to thank my thesis advisor, M.Sc Fernando Ubis, for taking time out of his busy schedule to guide the thesis topic and direction and providing me with insight into the world of IEC 61499.

I would addittionally like to thank Visual Components for the opportunity to work on this project and everyone at Visual Components who have made feel welcome and a valuable member of the company. I am also grateful to my family and friends for supporting me during the writing process, providing a fresh set of eyes for proofreading of this thesis and general support.

Espoo, Finland, 17. September 2024

Sampo Vänninen

Contents

Glossary

AFB Adapter Function Block.

API Application Programming Interface.

BFB Basic Function Block.

CAT Composite Automation Type.

CFB Composite Function Block.

DMS Discrete Manufacturing System.

DT Digital Twin.

ECC Execution Control Chart.

FB Function Block.

FBN Function Block Network.

FIFO First-In-First-Out.

HIL Hardware-in-the-loop.

IDE Integrated Development Environment.

IEC International Electrotechnical Commission.

Industrial PC Industrial Computer.

IT Information Technology.

MIL Model-in-the-loop.

OT Operational Technology.

PLC Programmable Logic Controller.

PoC Proof of Concept.

RIL Reality-in-the-loop.

RT Runtime.

S2S Source to Source.

SIFB Service Interface Function Block.

SIL Software-in-the-loop.

SIMFB Simulated-connected Function Block.

SimPLE Simulated Programmable Logic Environment.

ST Structured Text.

VC Virtual Commissioning.

VC 4.0 Visual Components 4.0.

VC™ Visual Components.

XML Extensible Markup Language.

1 Introduction

When commissioning new manufacturing hardware, various strategies are used to achieve a smooth transition into production. One of these strategies is Virtual Commissioning (VC), which is used within the automation industry to achieve faster time to deployment, reduce engineering costs, as well as increase the maintainability and flexibility of machines [1]. To enable virtual commissioning, simulation software is utilized to create a digital representation of a process, which can be used to develop and test a system before any real hardware is acquired. However, simulations are never perfect, and current workflows have problems when attempting full-stack virtual commissioning without any real hardware [2], particularly when engineering the control programs [3].

1.1 State of PLC programming

The main standard within the automation industry for Programmable Logic Controller (PLC) programming is the IEC 61131. Its widespread adoption in the industry makes it the de facto standard for programming industrial systems. Part 3 of the standard (IEC 61131-3) defines four programming languages which can be used to create control program [4]. Control applications developed with the standard operate in a cyclic fashion, intended for centralized control tasks within realtime environments, running on a single PLC. The increased digitalization and adoption of Information Technology (IT) practices within the automation industry, commonly known as Industry 4.0, has shown a need for more dynamic manufacturing systems [5], with distributed systems as one proposed solution. In distributed control systems, multiple PLCs cooperate to form the application, a task the IEC 61131 standard was not designed for. To support distributed control system development, the IEC 61499 standard for distributed automation was proposed, defining a programming language suited for developing these distributed control applications. However, the industry adoption of the standard has been slow since its introduction due to perceived overpromises of the capabilities [6], combined with the existing investments in incompatible hardware and software introducing an economic barrier to change. Nevertheless, the IEC 61499 standard has seen increased academic and industry interest in the last few years [7], with large consortiums adopting the standard [8].

The IEC 61499 standard is promoted as open, providing interoperability, portability, modularity and extensibility within the ecosystem. Unfortunately, research has found that these promises are yet to materialize within the offered tools [9] [6]. IEC 61499 applications developed using a specific Integrated Development Environment (IDE), with variants provided by various vendors, are generally not directly transferable to other IDEs. Additionally, the standard has documented ambiguities [10], causing differences in the implementations, which cause issues when attempting interoperability between differing software. These differences have been shown to be either minor or major, where minor incompatibilities can be overcome using external tools, while major incompatibilities make interoperability infeasible [11]. The same issues arise from different implementations of the execution platform of the applications, commonly

referred to as Runtimes (RTs). The interoperability between different RTs is possible, but seldom supported within IDEs, leading to vendor-lock as is common within the automation field.

1.2 Visual Components, Research Question and Goals

This thesis was conducted for Visual Components (VC™) Oy, a Finnish-based 3D Simulation software vendor, and focuses on virtual commissioning using the namesake product of the company, Visual Components Premium. The rest of the thesis refers to this simulation software with Visual Components 4.0 (VC 4.0) The feasibility of the software for virtual commissioning has been investigated before [12], and this thesis can be considered as a follow-up to the company's research in the area of interest. The aim of this thesis is to improve the support for IEC 61499 within VC™, using commercial IDEs alongside VC 4.0. The current PLC support primarily focuses on the older 61131-3 standard, and improving support for IEC 61499 can thus provide new opportunities for growth in the future.

In particular, this thesis attemps to answer the following research question: How can Visual Components improve its simulation software to better support the control engineering phase of virtual commissioning, and by extension, how can the simulation be used to improve PLC programming?

The simulation in this thesis refers to a complete virtual layout of a system being commissioned that has parity with the real world, including mechanical, electrical, kinematic and process information. Although such perfect simulation is impossible to create due to time and computing resource constraints, it will be used as a reference on how virtual representation of the system can be utilized to improve the programming of PLCs.

To answer the research question, this thesis seeks to identify the issues within the currently used virtual commissioning workflow, particularily in the control engineering phase. Afterwards, a solution to solve these issues is presented, with a proof of concept implementation of the solution to validate the proposed approach. This work is limited to supporting the most common IEC 61499 features found in various IDEs, as vendor-specific features would increase the complexity and reduce the applicability of the work. Some advanced features, such as Composite Automation Types (CATs) found in IDEs developed by nxtControl and used to simulate the controlled process, can naturally be replaced by the simulation inside VC 4.0. The work also focuses on the features described in the standard, and does not discuss details left ambiguous, such as networking implementation.

With the complete implementation of the proposed solution into VC 4.0, full-stack virtual commissioning using the simulation becomes more feasible. Depending on the implemented level of comprehensiveness following the IEC 61499 standard, it could fully replace the need for a third-party soft PLCs when developing control code during the virtual commissioning phase.

1.3 Thesis Structure

The rest of the thesis is divided into six chapters. Chapters 2 and 3 introduce the theoretical background for the topic, with Chapter 2 introducing Virtual Commissioning, 3D Simulation Software and programming technicalities for PLCs, and Chapter 3 providing an overview of the IEC 61499 standard, along with its applicability to the research goals. Chapter 4 discusses the requirements and challenges for integrating a IEC 61499 runtime into a simulation, with Chapter 5 introducing a new architecture for implementing a IEC 61499 runtime for constrained environments. Chapter 6 details a proof of concept implementation to test the validity of the proposed architecture, with Chapter 7 consolidating the results from Chapters 5 and 6. Finally, Chapter 7 concludes the thesis with discussion on results and proposals for follow-up research.

2 Theoretical background

2.1 Virtual commissioning

Virtual Commissioning refers to the validation of software and hardware without the real system. In [13] and [14], the authors show that up to 90% of the commissioning time is spent on the electric and control system configuration, 70% of which is spent correcting software errors. Utilizing VC can thus provide large reductions in time and capital costs by reducing the time spent by making errors easier to fix and also allowing the configuration of the control software to begin earlier in the commissioning process. The practice is well known in the industry, but is lacking widespread adoption due to technical challenges such as lack of accurate models [15]. Nevertheless, VC is an important tool for enhancing the manufacturing industry, providing large advantages both in time and quality when implemented [14].

According to [14], VC can be divided into three main types, depending on the relation with the real system [2]. Additionally, Model-in-the-loop is sometimes used as a fourth type. Each of these types is outlined below, alongside how VC 4.0 currently supports each stage. The types often follow each other, complementing the implementation of the next step.

- **Model-in-the-loop (MIL):** Integrates the control logic on a high level into the simulation, allowing the detection of logic errors within the system. The logic controls the simulation, which provides its state to the logic. MIL requires the least amount of fidelity from the components of the simulation. VC 4.0 supports this natively, allowing the logic to be programmed directly as a part of the simulation.

- **Software-in-the-loop (SIL):** The logic is transformed into real control code and ran in a simulated environment, such as a soft-PLC. SIL allows validating the control code implementation against the designed logic, but requires a more accurate simulation, as the logic is no longer abstract, instead operating on the signal level. VC 4.0 partially supports SIL with the Connectivity plugin, which allows connecting external soft- and hard PLCs into the simulation. However, many of the parts available are not modeled down to the signal level, requiring extra engineering effort for full compatibility.

- **Hardware-in-the-loop (HIL):** The control code is ran on real, physical PLCs that control the simulation. VC 4.0 has similar support for HIL as with SIL, requiring the use of a separate Connectivity plugin and parts that support signal based control. HIL can also be called hybrid commissioning if some parts of system use the simulation and other parts use real hardware.

- **Reality-in-the-loop (RIL):** A less common VC type, where the simulation is used to directly control the real system. RIL places the highest requirements for the simulation fidelity, as any inaccuracies can lead to erratic behaviour, which can have catastrophic consequences. VC 4.0 technically supports this by

connecting signals and values within the simulation into the controller of real machinery, but is not officially supported.

2.1.1 Benefits

The time and cost savings of VC are realized through several differences when compared to physical commissioning. Changes in the hardware can be made in minutes instead of a slow shop floor configuration process, allowing the testing and optimization of a large variety of layouts, machines, and controllers within a small amount of time [1]. Similarly, the lack of physical hardware allows testing potentially destructive processes without risk of damage, such as multi-robot systems which could collide with each other.

A large time advantage in the commissioning process can also be gained from allowing control engineers to start working on the control software in an earlier phase of the commissioning process. Traditionally, control code can only start being developed and tested after the hardware has been installed, leading to a sequential commissioning process. With VC, the control software can be developed and tested in parallel alongside the simulation design phase [2]. The authors in [14] note that up to 70% of the commissioning time is spent on fixing software errors, which leads to large time savings if the process can be started earlier. Additionally, time savings can be gained from making development more efficient by not requiring access to the physical shop floor.

VC also brings benefits to the complete system after the commissioning phase. Modifications can be applied without bringing the production line offline and operators can be trained with the simulation model. Changes to the system can be tested and adapted to the market situation, allowing informed decisions of required changes to stay competetive. Having access to a virtual replica of the system also introduces benefits for the maintaining and operating phases. By introducing bi-directional data flow between the real system and the simulated environment, a Digital Twin (DT) can be realized [16]. These DTs can then be used to monitor the state of the real system, enabling predictive maintenance and tracking the performance of the system.

2.1.2 Drawbacks

However, creating DTs is expensive, as collaboration between all included parties of the commissioning process is required, spanning multiple fields from hardware suppliers to IT specialists [2]. Creating a model that is sufficiently accurate can take a considerable amount of engineering time, and validating the models against the physical hardware is a complex task. With existing libraries, such as the eCatalog provided by VC 4.0, existing models can be used between projects and the engineering time required for the model creation is not included as a part of the commissioning process. Unfortunately, models from a component library are generally not accurate enough for use in VC directly. The models can be split into two parts [15]:

- Kinematic Model: 3D geometry, kinematic links and other physical properties

- Behaviour Model: Control logic, signal inputs and outputs

Most simulation software vendors provide a catalog of components for users, but generally focus on providing an accurate kinematic model with only a basic behaviour model. These components generally lack accurate signal-level modeling, requiring extensions if SIL or HIL is desired. To the authors knowledge, no component catalog provides accurate behaviour models for a large variety of vendors, leaving the implementation of accurate models to the end user, which increases the modeling costs and engineering effort.

2.2 3D simulation software

Simulation software is used to model systems in many domains, utilized both in the engineering phase and during operation. For Discrete Manufacturing Systems (DMS), 3D Simulation is often used due to the strong spatial nature of the systems. The simulations are often discrete-time with a variable timescale, include aspects of kinematic and behaviour logic, and allow gathering statistical data from long-running simulations. An overview of the most commonly used tools for 3D industrial simulation can be found in [17], introducing 16 different 3D tools, both open-source and proprietary. The capabilities of 3D simulation software will be primarily discussed from the context of VC 4.0 for familiarity and relevancy with the topic, as many of the features are shared between different vendors.

Common features

- Model library: Most simulation software provide a prepackaged library of ready-to-use components, including robots, conveyors, and machines. These allow rapid prototyping of ideas with minimal effort, but often require additional modeling for further development, such as for use in virtual commissioning. VC 4.0 provides an extensive model library from multiple vendors called the eCatalog.

- 3D layout configuration: All tools allow configuring the locations of components within the system, providing a way to easily modify the layout. VC 4.0 allows plugging parts together in the form of interfaces to provide functionality to the simulation, requiring minimal to no code expertise for creating functional simulations.

- Simulation time adjustment: Simulations can be run in discrete time steps, which allows pausing and resuming the simulation at will. The time step duration can generally be configured, which enables running the simulation slower than real time for detailed analysis, or faster than real time to simulate large amounts of system operation within a small amount of time.

- Statistics: To gather data from long simulation runs, most tools provide a way to gather values from within the simulation for statistics. VC 4.0 has a built-in statistics viewer that can display this data live while the simulation is running.

- Saving & loading: All available software support persistence of models by saving them in specific file formats. Unfortunately, these formats are often proprietary, making interoperability of models between different vendors impossible.

Advantages

3D simulation software acts as the backbone for Virtual Commissioning, providing the platform for digitally modeling production systems. They are also often used to explore new ideas and to create proof of concepts before committing to a project. A simulated prototype of a proposal can provide key information on the feasibility, and also act as the starting point for engineering if further development is desired. For this purpose, the model libraries of most tools are sufficient, as in-depth behaviour models are not yet required. The 3D nature of the tools provides a visual representation of the system, which is easier to conceptualize than tables of numbers and diagrams [18, Chapter 2.4]. This visualization of large amounts of data allows working on complex systems efficiently, especially for non-engineers. The visual layouts can also be used to present ideas to stakeholders, who might not fully understand a system from numbers and textual descriptions alone. 3D simulation software is often used to realize DTs within the DMS industry, which can be used to produce value over the entire system lifecycle. The term Digital Twin has various different meanings depending on the industry [19], with this work using it as a term to refer to virtual systems that match all kinematic and behaviour aspects of a real system, and include bidirectional data access between the two. Such "pure" DT has access to all relevant data of the physical object, and can be utilized for commissioning, predictive maintenance, reconfigurability and decommissioning, among others [20].

Issues with simulation

The provided model libraries are generally sufficient for the early stages of development, but require large amounts of extra engineering effort when full virtual commissioning is attempted. This effort, often in the form of modeling new components or extending existing components, requires considerable expertise which is either expensive or not available. This barrier can prevent the usage of simulation tools in following engineering phases, making virtual commissioning ineffective or impossible. The technical realities also hinder the use of VC. Simulations necessarily use computational resources to run, which limit the complexity of systems to the available capacity. Very large systems might be infeasible or require trade-offs in detail, scale, or time, limiting the possible applications of simulation software. Recent advances in computing performance have been primarily through the increase in core counts, but most simulation software, including VC 4.0, only utilize a single core, reducing the advantages gained from modern hardware.

Another major problem within the simulation tools available is the lack of interoperability. Different vendors use different standards and storage formats, causing models created in one program to be incompatible in another. Some research has been done on intermediary modeling languages, such as COLLADA [21], that would allow using

the same model in multiple simulation tools, but no universal standard encompassing all the functionality of a simulation has emerged. This can be seen as a drawback to using simulations, as any intellectual knowledge is currently locked into the ecosystem of a single vendor, limiting the choices available in later phases of the commissioning process. Co-simulation is a powerful tool to help with the lack of shared data formats. By utilizing an intermediary connector, such as OPC UA, different simulation software can be utilized in parallel, allowing the selection of best-of-breed software for each task. In VC 4.0, the Connectivity module allows connecting various PLC simulators, robot controllers and other simulation environments into the simulation. Due to the differences in software, tight time synchronization is generally infeasible, making the use of "real time" simulation mandatory, removing one advantage of a fully virtual system.

Another time-related issue of 3D simulation software for VC purposes is the lack of true realtime simulation. All commercially available simulation tools run on non-realtime operating systems, such as Windows or Linux, and controlling the simulation using PLC code that is intended to run on hard realtime operating systems necessarily causes inaccuracies when compared to the real system. An in-depth analysis of these timing issues is provided in [19]. The authors provide a time synchronization scheme to mitigate these issues, but this still forces the simulations to run at realtime. An introduction to how PLC runtimes work and how they are programmed is provided in the next section, along with approaches to solve the issues with integration into simulations, based on the findings in [12].

2.3 PLC IDEs and runtimes

Industrial control software running on PLCs is commonly developed in an IEC 61131-3 compliant language, with increased adoption for the IEC 61499 standard in recent years. As IEC 61499 and three out of the four defined languages in IEC 61131 are graphical programming languages, dedicated software is required for efficient development. These software packages allowing the editing, debugging, testing, and deployment of control programs are referred to as Integrated Development Environments (IDEs), with various vendors providing their own. There are both free and commercial offerings, with the main feature sets being comparable.

All IDEs support the basic functionalities of PLC development, with the choice of vendor primarily affecting the supported hardware and extra functionality. Most IDEs only support either IEC 61131 or IEC 61499 applications, with some notable exceptions, such as the ISaGRAF workbench and nxtControl supporting both [22]. The focus in this work will be on the IEC 61499 ecosystem. The standard itself will be discussed in depth in Chapter 3. The IEC 61499 standard promises interoperability as one of its core strengths [23], although studies have found interoperability between tools lacking [11]. Due to the differing amount of features within IDEs, the produced engineering artifacts do not fully transfer to other software, leading to issues between vendors. Similarly, commercial IDEs target a specific runtime for deploying the developed applications, producing similar vendor lock-in as is seen with the IEC 61131 ecosystem. Promisingly, hardware from different vendors utilizing a shared

runtime can be used together, showing the possibility of IEC 61499 as an enabler of distributed, hardware-agnostic control programs [8].

Runtimes

IEC 61499 runtimes are an implementation of the standard into a program that allows the execution of control applications developed within IDEs. Multiple runtimes with various levels of openness are available, from free open source, shared source and closed source [24]. The ambiguous semantics of the standard cause issues when attempting interoperability between different runtimes [10], causing most applications to only work when distributed on a homogenous runtime implementation. Such fragmentation causes similar vendor lock-in as can be seen within the existing 61131-compatible PLC vendors and their proprietary software. This lack of coordination complicates realizing the advantages of the IEC 61499 standard, where the programmer should not need to worry about the underlying hardware when designing an application.

On a technical level, a runtime is a computer program developed in a general programming language such as C++ or Java which allows the execution of other programs that provide the desired functionality. The runtime handles the interfacing with the underlying operating system and by extension the hardware. IEC 61499 control programs are compiled using IDEs to target specific runtimes, producing code that can be executed on the central processing unit of a computer or a PLC.

PLC Runtime are, as a rule, designed to run on hard realtime systems, which most commonly refers to physical PLCs. Such systems can guarantee the execution of code in a specified time window and form the backbone of industrial automation. During development and testing, the code can be ran on a simulated PLC, generally called a soft-PLC. Such simulators are usually non-realtime, primarily used to test the validity of the logic but often used in SIL virtual commissioning. However, this disparity in the timing requirements between soft and hard PLCs is a source of inaccuracy for VC, generally requiring the testing of control code on physical PLCs to guarantee the correctness of any developed control application.

Within the industry

The use of VC and simulation for automation engineering projects is a well-established field, but the issues presented prevent a wider utilization of the practices within the industry. Many issues arise from the technical limitations of existing software, partly caused by the relative infancy of feasible full-stack virtual commissioning. The academic interest on Digital Twins and VC is high with a large amount of various proposals and prototypes [19]. However, the lack of good reference implementations within the industry lead to system integrators having to develop a large part of the process themselves when attempting to realize the benefits to their fullest. This leads to even more engineering complexity on top of the difficulty of implementing VC in the first place.

The develoment of control software using the IEC 61499 standard is in a similar situation, with many software tools facing slow adoption. The full potential of the IEC

61499 standard is difficult to realize without the cooperation of all involved parties, a monumental task in the automation industry with legacy hardware, existing skillsets, and a large amount of small to medium sized companies weary of uncertainty. The demand for know-how on IEC 61131 programming to maintain existing equipment also acts as a barrier for change, noted by researchers [25]. To help with the transition, multiple approaches to integrate IEC 61499 within legacy projects have been proposed [26] [27]. The next chapter will discuss the IEC 61499 standard in detail.

3 The IEC 61499 standard

The observed need for more flexible manufacturing systems in the late 1990s led the International Electrotechnical Commission (IEC) to start working on a standard that would be designed to facilitate distributed control applications. This work culminated in the IEC 61499 standard, building upon the widely adopted IEC 61131-3 function block model. The first edition was released in 2005, with a second edition being published in 2013, providing clarifications to the ambiguous concepts of the initial document [28]. Applications, called Function Block Networks (FBNs), are decoupled from the hardware, allowing engineers to design the control logic without being limited by specific hardware implementations. In the deployment phase, applications are distributed to hardware resources such as PLCs. With a single application capable of running seamlessly over multiple resources, a distributed control system is achieved. The IEC 61499 standard has, in contrast to the cyclical nature of IEC 61131, an event-driven asynchronous execution model that facilitates distributed control by not requiring strict time coupling between nodes for functioning.

3.1 Technical implementation

The basic building block of IEC 61499 applications is the Function Block (FB), shown in Figure 1, familiar to control engineers from the 61131-3 standard. The data inputs and outputs, also present in the IEC 61131 FBs, have been extended with the addition of event inputs. The FBs are no longer executed cyclically in a deterministic order as in IEC 61131, but are invoked whenever an event input arrives [25]. Utilizing timed event generators, a cyclical execution similar to IEC 61131 can be achieved [29]. IEC 61499 applications can also be arranged in a centralized control fashion, allowing the standard to be used in both centralized and distributed control [30]. Only the main components of the IEC 61499 standard are presented here. For a more in-depth analysis, see [23].

3.1.1 Events and data

Event inputs and outputs are paired with data inputs and outputs, respectively, with the value of the data being updated only if an event connected to the data port is received. This leads to data value changes being unable to be used as a trigger for FBs, consequently allowing treating the system as pull-based instead of push-based. The standard defines a strict one-event-per-instant rule for FBs [32], necessating a rule to handle simultaneous events, such as an event buffer. However, such a mechanism is not provided by the standard, leading to variable behaviour between implementations. The data types used by IEC 61499 are the same as in 61131, providing a basis for intercommunication between the two standards [26].

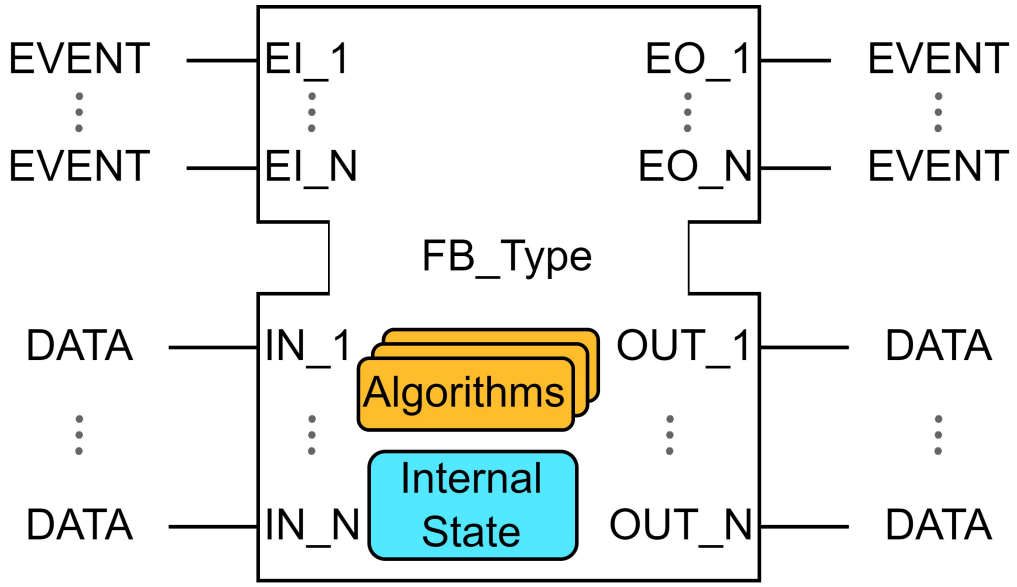

Figure 1: IEC 61499 Function Block. Adapted from [31].

3.1.2 Function block types

The IEC 61499 standard defines three main types of FBs, which all share the event and data port interfaces specification, but differ in their internal functionality.

- **Basic Function Block (BFB):** The simplest FB type, consisting of data and event ports, an Execution Control Chart (ECC) and one or more algorithms. The implementation language for the algorithms is not specified, but is often written in 61131-compliant Structured Text (ST) due to familiarity with control engineers.
- **Composite Function Block (CFB):** A hierarchical FB type that doesn't implement any functionality on its own. Contains a sub-FBN, acting as a logical grouping of functionality.
- **Service Interface Function Block (SIFB):** A general grouping of FBs that are implemented separate of the IEC 61499 standard. SIFBs are used to implement hardware-specific functionality, communication, basic operations such as timers and everything else that might not be feasible with the two other types.

Resources and system configuration

Resources act as an abstraction between the application and the hardware executing it. They contain the IEC 61499 runtimes, which translate the execution semantics defined in the standard into executable code. Usually, one resource corresponds to one physical controller, such as an Industrial PC or a soft- or hard PLC. Every FB within the application must be bound to a resource, with event and data connections between different resources requiring a set of SIFBs to define the communication protocol

between them. Some IDEs can automatically implement these communication links, reducing the engineering effort, especially in large distributed applications [25].

3.2 Advantages of IEC 61499

There are other advantages with the IEC 61499 standard than just enabling distributed control systems, namely modularity, interoperability, and reusability. Most of the advantages derive from the function block architecture, a trait partially shared by the 61131-3 standard. However, IEC 61131 is generally used in closed development loops with hardware tightly coupled within the applications, which leads to a lesser amount of possibilities to take advantage of the modular function block design. As the IEC 61499 standard is designed to separate the software and hardware, object-oriented benefits can be realized more readily [33], mirroring the developments from the IT world.

Modularity

FBs encapsulate functionality within them, allowing access through data and event connections. This allows complex behaviour to be accessed with simple connections, and permits swapping out blocks of the application at once, only requiring the interfaces to match up. This modularity is similar to IT counterpart of a *class*, where implementation details of software objects and methods are abstracted away with public facing interfaces, simplifying code reuse.

Interoperability

Separating the hardware and software implementation and handling data connections between them with SIFBs provides a way to connect systems together without having to explicitly design the application around it. Combined with modularity, providing expandability to applications in the form of pre-defined communication FBs allows rapid development of distributed applications. Unfortunately, the creation and managing of the connection-providing SIFBs is a major engineering bottleneck due to the lack of a standardized function block library, somewhat minimizing the actual realized interoperability.

Reusability

With modular, abstracted FBs, control applications can be developed by plugging in tried-and-tested existing FBs that provide the required functionality, providing more reliable code with minimal effort. The portable nature allows the creation of FB marketplaces [34], which enables achieving the desired functionality by connecting together FBs from various vendors, distributed through a digital storefront while keeping the implementation details proprietary. Similarly, any existing applications developed by a organization can be reused in the future, as IEC 61499 programs are not tied to specific hardware, unlike most 61131-3 applications. This decouples

the cost of developing control software within a commissioning project from the underlying hardware, allowing cheaper operational costs as the software can be reused with minimal changes in the future.

However, all of the advantages so far are only theoretized. In almost every implementation of the 6149 standard available as of writing, design choices can be identified that prevent the full benefits from being realized [6]. Many of the advantages can also apply to 61131-3 applications with various levels of implementation effort, further reducing the perceived value of switching to the less-supported standard.

3.3 Identified problems

The primary chunk of the identified problems come from the lax definition of the standard, providing room for different interpretations between implementations, rather than technical flaws [35]. As the IEC 61499 standard can be viewed as an extension of the accepted 61131-3 standard, increased focus is directed on the differences from IEC 61131 instead of the inherited features. An in-depth analysis of the ambiguities within the standard can be found in [36] and [37].

3.3.1 Execution semantics

The IEC 61499 standard has been noted to have multiple ways of implementing the execution of FBs [36] [35], with many different existing implementation within the runtimes [24]. This causes problems for interoperability between different implementations, as program execution differs between runtimes, leading to non-deterministic behaviour. Similarly, different execution semantics between a simulator used for virtual commissioning and physical hardware prevents the validation of control code without testing on real controllers. Different implementations have varying strengths, particularily when considering multi-threaded, multi-controller and embedded systems. In low-performance environments, a sequential, single threaded execution can be preferable over the more common one thread per FB model [38].

As long as the standard lacks an unambiguous definition of execution semantics, three outcomes are possible:

- The status quo is kept, with islands of interoperability arising around incompatible execution semantics.
- A consensus is formed on execution implementation, either through standardization effort or from some implementation becoming a de facto standard naturally.
- A way to circumvent the issue is found. A promising research field is Formal Validation [10], although other solutions to solve the lack of interoperability have seen higher research interest.

3.3.2 Extensions to the standard

All vendors providing IEC 61499-related software provide extra functionality not defined in the standard, such as the 4diac IDE providing function block colours or nxtControl introducing the Composite Automation Type (CAT) for representing real hardware within the application. These additions do not necessarily have a counterpart in other vendors' IDEs, causing issues with source code level portability [9]. IEC provides a solution to these by providing the IEC 61499 Compliance Profiles [39] that allow vendors to define the compliance level and implementation of the standard within their software. However, without a higher level compliance profile, islands of compatibility are necessarily created from various software vendors choosing different compliance profiles to support, a situation not alike of the existing 61131-3 ecosystem.

3.3.3 Evolutionary vs. revolutionary

Yet another reason for the slow adoption of the IEC 61499 standard can be attributed to its evolutionary nature from the 61131-3 standard. While the familiar nature of function block-derived IEC 61499 can be seen as a benefit for adoption, it can also be felt as an evolutionary change instead of a revolutionary one. Companies with existing 61131-3 systems, who already have the know-how and expertise with the older standard, can see the new standard providing little benefit over their existing methods. To solve this conundrum, multiple approaches to combine the two standards to ease the transition have been researched, outlined in the next section. Additionally, Industry 4.0 concepts are increasingly bringing Operational Technology (OT) and IT together. Combined with the new generation of control engineers being increasingly receptive to the concepts of object oriented programming within the field, the industry could see a transitioning into new technologies, including IEC 61499.

3.4 Related research

Nevertheless, the IEC 61499 standard has seen a large amount of academic interest, with research on how to utilize the standard [40] [41], how to solve specific issues within distributed system development with IEC 61499 [42, 43, 44, 45], proposals to interoperate IEC 61131-3 applications with IEC 61499 applications [26] [27], and many others. A particular topic of interest relating to this work is the use of the standard for simulation [46] [47] [48], particularily in 3D discrete manufacturing [49]. Previous research detailing the development of IEC 61499 runtimes is also of particular interest.

In [40], the authors approached virtual commissioning of IEC 61499 applications by using a simulator with soft-PLCs to tune the parameters of the control program before transferring them to hardware. Successful optimization of a control application was demonstrated with virtual commissioning, with the optimized paramters being transferable to a real-world system directly.

A lightweight IEC 61499 compiler called goFB was introduced in [50], which could convert FBNs into C code. The paper discusses the timing benefits and drawbacks of

various runtime execution strategies, and also discusses the implementation of BFBs and CFBs in a generic programming language.

Various papers have described the usage of IEC 61499 tightly coupled with the simulation. A common approach is implementing the simulation of the system directly within IEC 61499 itself, removing the need for co-simulation, with an example found in [46]. The proposed approach in the paper is similar to the CAT used in some IEC 61499 IDEs. A similar approach is also taken in [51], which implements a new Hybrid Function Block type which can model Ordinary Differential Equations with the base semantics of the standard. These Hybrid FBs can be used to simulate continuous processes directly in IEC 61499 applications. Such approaches both provide a tight coupling between the simulation and the control logic, but are limited in complexity due to the abstraction of the system into FBs, and do not transfer to 3D DMS.

Papers detailing the implementation of IEC 61499 runtimes are scarce, likely due to the existence of good open source implementations which are commonly used as a base for research instead of developing a runtime from the ground up. An early example of runtime developement can be found in [52], which details the creation of a custom runtime for a Java-based microcontroller, providing insight into the development process to convert the IEC 61499 standard into executable programs on the Java runtime. Some implementation details in this work are influenced by the ideas presented in their work.

3.5 61499 vs 61131-3

Despite the large adoption of the IEC 61131-3 for developing control applications, IEC 61499 was chosen as the focus of this thesis as it provides better interoperability characteristics for virtual commissioning. Additionally, reducing the engineering complexity of centralized control using the well-understood IEC 61131-3 standard has lower potential efficiency gains compared to IEC 61499, which has its complexity as one of the identified reasons for low industry adoption. Furthermore, applications developed using the IEC 61131 standard can already be used with VC 4.0 through the connectivity plugin, albeit with some previously noted drawbacks, such as forced real time simulation. Support for IEC 61499 is very minimal, leading to a more integrated solution providing new ways of using the software.

4 Integration into the simulation

One approach for improving PLC support inside the simulation software is to integrate an existing IEC 61499 runtime into VC 4.0. However, this approach has some major implementation problems. VC 4.0 is built on top of a C++ core, with a C# interface layer and a Python scripting layer, leading to the runtime having to be implemented in one of these languages for integration to be feasible. Additionally, the source of the runtime would have to be modifiable. There are runtimes that fit this criteria, specifically the 4diac FORTE [53] and Universal Automation runtime [8], both being written in C++ and having open and shared source, respectively. Choosing an existing runtime would provide good support for IDEs that target that specific runtime, but would not work with other runtimes, making this approach undesirable, as being vendor-agnostic is preferable. As such, a different approach based on the well-defined system interchange format found in the IEC 61499 standard has been chosen for this work. A different approach not chosen for this thesis would be integrating a IEC 61499 IDE into VC 4.0 directly to create the control logic and applications within the simulation software. However, various powerful editors dedicated to producing IEC 61499 applications already exist, and competing against these is not in the primary interest of Visual Components. As such, focus is on how to use these external tools alongside VC 4.0.

4.1 The system interchange format

The format specified in IEC 61499-2 is a schema for Extensible Markup Language (XML) that describes the entire function block network, called a system file and marked with the .sys extension by many IDEs. The files allow the transfer of applications between software, and are commonly used as the save format of IDEs. Figure 2 shows an example of a system file and the corresponding graphical representation. The primary importance in the files are the SubAppNetwork specifications, which include all the FBs and connections between them that make up the function block network. As the FBs in the system file are only referred to using a type, a separate schema for defining FBs is also provided in the standard, often denoted with the .fbt extension.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE System SYSTEM "../LibraryElement.dtd">
<System Name="System" Comment="System Configuration">
  <Attribute Name="System.Device.FolderPath" Value="EcoRT_0" />
  <Identification Standard="61499-1" />
  <VersionInfo Organization="Schneider Electric" Version="0.0" Author="SampoVänninen" Date="5/29/2024" />
  <Application ID="56A939F684496BA6" Name="APP1">
    <SubAppNetwork>
      <FB ID="E3D5243C754B3590" Name="RollingCounter" Type="E_CTU" x="1060" y="580" Namespace="IEC61499.Standard">
        <Parameter Name="PV" Value="3" />
      </FB>
      <FB ID="BF60367FD99F7ED6" Name="Reset" Type="E_PERMIT" x="1540" y="580" Namespace="IEC61499.Standard" />
      <FB ID="20986EF18BBC2969" Name="Conveyor" Type="CONVEYOR_CONTROLLER" x="2060" y="900" Namespace="VisualComponents.SimFB">
        <Parameter Name="ID" Value="1" />
      </FB>
      <FB ID="7157D0136B99898F" Name="PLC" Type="PLC_IO" x="340" y="580" Namespace="VisualComponents.SimFB">
        <Parameter Name="ID" Value="2" />
      </FB>
      <FB ID="51DC588789C3875B" Name="OneAdder" Type="ADD_1990CFD1468AAE4A6" x="1480" y="1060" Namespace="Main">
        <Attribute Name="Configuration.GenericFBType.InterfaceParams" Value="Runtime.Standard#CNT:=2;IN${CNT}:LREAL" />
        <Parameter Name="IN2" Value="1" />
      </FB>
      <EventConnections>
        <Connection Source="RollingCounter.CUO" Destination="Reset.EI" />
        <Connection Source="Reset.EO" Destination="RollingCounter.R" dx1="47.16675" dx2="70" dy="-110" />
        <Connection Source="PLC.EO1" Destination="RollingCounter.CU" />
        <Connection Source="RollingCounter.CUO" Destination="OneAdder.REQ" dx1="60" />
        <Connection Source="OneAdder.CNF" Destination="Conveyor.ROUTE" dx1="47.47925" />
      </EventConnections>
      <DataConnections>
        <Connection Source="RollingCounter.Q" Destination="Reset.PERMIT" dx1="123.5625">
          <AvoidsNodes>false</AvoidsNodes>
        </Connection>
        <Connection Source="RollingCounter.CV" Destination="OneAdder.IN1" dx1="40" />
        <Connection Source="OneAdder.OUT" Destination="Conveyor.DIR" dx1="70" />
      </DataConnections>
    </SubAppNetwork>
  </Application>
  <Device ID="3535682F-8D51-4D16-8237-70EF3832F2DF" Name="EcoRT_0" Type="Soft_dPAC" x="700" y="700" Namespace="SE.DPAC">
    <Resource ID="CB056D06C3F801CA" Name="RES0" Type="EMB_RES_ECO" x="800" y="800" Namespace="Runtime.Management">
      <FBNetwork />
    </Resource>
    <FBNetwork />
  </Device>
</System>
```

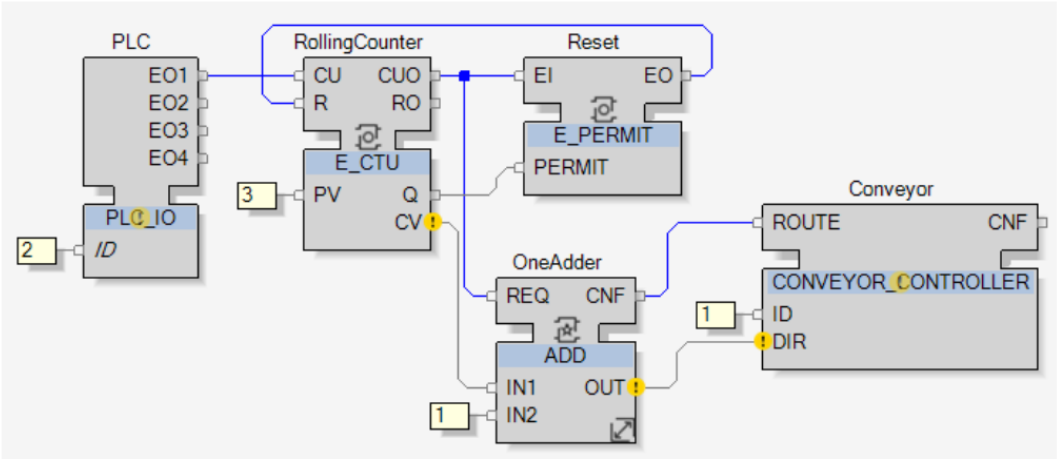

Figure 2: Example .sys file and the corresponding Funtion Block Network made with EcoStruxure Automation Expert.

State of interoperability

While all available IDEs follow the XML schema for their base functionality, due to differing technical implementations, they often have extensions not present in the standard. Some observed additions are function block colour information, namespace information and identification data. When attempting to load these system files between programs, information is often lost, leading to partial transfer of applications between IDEs.

4.2 Design guidelines

To provide an approachable and functional tool, five high-level design goals are described, guiding the development of the tool and acting as a benchmark for evaluation in Chapter 6.

1. **Minimal:** The implementation shall not require any extensions to the format provided by the standard to stay compatible with all variants using it as the base.
2. **Extendable:** It shall be easy to implement more functionality to interface with the simulation in the form of function blocks.
3. **Observable:** The function block networks shall be observable during their execution.
4. **Interoperable:** The functionality to access the simulation shall be easy to implement within IDEs used to create the applications.
5. **Integrated:** The tool shall use features already present in VC 4.0 to integrate it into the simulation and minimize the knowledge required to use it.

A technical view on how to approach these goals is discussed in the following section.

Minimal implementation

To stay compatible with as many IDEs as possible, creating a function block network within the simulation needs to only require information described in the base standard. Generally, differences in the formatting of IEC 61499 XML files are due to technical implementation of the IDE itself, such as namespace declarations, identification schemes and other organization features. As we are primarily interested in replicating the functionality of the function block network which is fully described by the IEC 61499 XML definition, discarding these IDE specific additions does not alter the execution logic of the program.

Extendability

VC 4.0 supports creating custom components in the software, which need to be controllable using the tool to be useful for testing control code. Furthermore, the existing catalog of components does not have support for controlling them using function blocks. As such, the creation of new FBs that allow connecting to components with the logic needs to be both easy and accessible to the end user.

Observing the programs

Troubleshooting and debugging the control applications is the primary purpose for running simulated PLC code, which requires access to the state of the network. Many IDEs allow observing the execution of the function block network directly from the graphical representation, usually denoted the Watch functionality. Watching FBs or variables allows seeing the variable values and FB states during execution, and replicating similar functionality within VC 4.0 is important for validating the correctness of control programs.

Connecting to other tools

Importing function block networks from IDEs needs to be simple and preferably realtime. Additionally, being able to edit the network while the application is running and have the changes propagate immediately is a feature to be considered. Going to the other direction, importing FBs to be used within the IDEs from the simulation needs to be easy, both in defining new FBs and providing them in a format the IDEs can use. For this purpose, the IEC 61499 standard defines a XML format for specifying the interface and behaviour of FBs which is utilized by IDEs to import external function block types.

Integration into the simulation

To stay familiar for the end user, the tool has to leverage these existing features for connecting the simulation with the control application. VC 4.0 already provides ways to control components using events in the form of signals, which allows using the graphical user interface for modifying connections. Additionally, any data required by the tool has to be provided through mechanisms found within the software, such as component properties for textual and numeric data. In essence, the developed tool attempts to be as transparent as possible to the end user, requiring no external software apart from the IDE and VC 4.0.

4.3 Choice of programming language

With these design goals in mind, Python is chosen for the implementation language of the tool. While C++ and C# would provide improved integration with the simulation engine, they are cumbersome to work with for the end user. Python still allows accessing the simulation directly through Application Programming Interfaces (APIs)

included in VC 4.0, providing sufficient functionality for the tool, while being easy for end users to modify and extend.

4.3.1 Technical limitations

The primary issue for implementing IEC 61499 function block network execution with VC 4.0 is the single-threaded nature of the software, which applies to any of the programming languages discussed. The event-based nature of the standard is suited for asynchronous execution where FBs are scheduled for execution based on events. In a single-threaded application, all processing is necessarily sequential, which prevents fully accurate replication of distributed application logic. However, the constraints imposed by single-threadedness allows simplifying the architecture of the tool. Additionally, any network related variance in applications cannot be tested using the tool. The simulation engine, which would also run the function block network, has perfect information of the state of the entire application, even if it is meant to be distributed. Any network delays, dropped messages or downtime must be artificially created. These limitations make the tool ideally suited for single PLC applications, which would be more suited for the 61131-3 standard. However, it is possible to convert IEC 61499 applications into 61131-3 applications if required [27]. Future versions of VC 4.0 could see multi-threading implemented, which would allow better implementation of distributed control application simulations. The next chapter introduces the architecture of the tool following the given design guidelines.

5 Technical Implementation

This chapter documents an architecture that allows the execution of IEC 61499 compliant system files inside VC 4.0 or other similar environments. A language agnostic high level overview is given, with an implementation in Python discussed in Chapter 6.

5.1 Architecture overview

The architecture proposed by this thesis will be referred to as the Simulated Programmable Logic Environment (SimPLE), which comprises three submodules: processor, runtime and exporter. The relations between the modules are illustrated in Figure 3. The concepts apply to IEC 61499 runtimes written in any programming language, and the following overview of the implementation, data flow, and datastructures can be implemented in most object-oriented languages for use in similar constrained environments. The architecture utilizes functions as variables, requiring the language to support functions as first-class citizens. SimPLE operates on system XML files (.sys) and FB definitions (.fbt) produced using IDEs that describe a complete IEC 61499 application using the format specified within IEC 61499-2. The Processor module parses the .sys file and creates a FBN with equivalent functionality. These FBNs are created within the runtime module, which is executed on a special PLC component inside a simulation environment that runs the FBN and handles signals and events. The Exporter is ran separately, creating FB XML files (.fbt) of all the SimPLE FBs specifications for use in IDEs. Depending on the targeted IDE, the Exporter also packages the .fbt files in a library format for importing into the IDE directly. As the architecture requires an object-oriented programming language where functionality is encapsulated within classes, functions will be referred to as *methods* and they will be *invoked* instead of called in the architecture.

5.2 Runtime

The runtime is split into the application and FB definitions. The application is a minimal implementation of a IEC 61499-compatible execution platform. Because we are constrained by the single-thread nature of the underlying simulation environment, the application can be implemented using only the FBs and all event and data connections between them to model the entire network.

The FBNs are built through instances of the SimPLE Application. The application is executed using a component inside the simulation which has access to

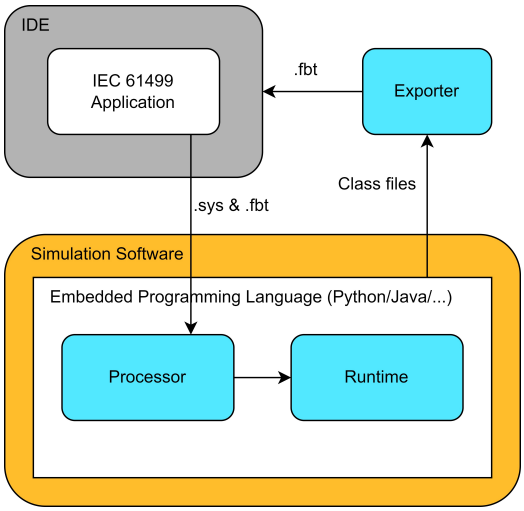

Figure 3: SimPLE architecture.

a single thread for both processing then simulation and the components themselves. This single-threadedness causes execution of the FBN to pause the execution of the rest of the simulation. Because the controlled system is also contained within the simulation, this is equivalent to events being processed in zero time from the perspective of the simulation. This behaviour can be seen as a PLC with infinite execution speed, which causes differing behaviour compared to real world execution of the FBN. In a physical PLC, execution of code necessarily takes CPU cycles and therefore time. However, events can arrive at any time instant and faster than they can be processed. This necessitates a strategy to handle events that cannot be immediately processed, often in the form of event queues. As the SimPLE application processes all events instantaneously, the need for an event queue for this purpose is removed. However, a First-In-First-Out (FIFO) queue is still used to sequence the emission of event outputs according to the Sequential Hypothesis [54].

Application architecture

As the majority of functionality of the network is within the FBs, the SimPLE application is kept lightweight. In other runtimes this part of the application is often called the scheduler. However, due to the instantaneous execution of events, no scheduling is required and therefore the class that runs the FBNs is instead referred to as the Application. The execution of the FBs is realized through two data structures: A map of event sources to event outputs, and a map of data outputs into their inputs. As the IEC 61499 standard does not allow a data input to have multiple sources, mapping each data input to its data output source is sufficient to track all data connections. However, event inputs allow multiple event output sources, requiring a mapping of event outputs into all directly connected event inputs.

As a consequence of this, the event connections can be seen to work as a push system, with every event invocation propagating the output event to all connected input events. A FIFO queue is used to invoke events in the order the output events were scheduled in, as otherwise the architecture would process the entire event chain until no more output events are fired and only then return to earlier events. Like the event connections, data connections can similarly be seen as a pull system, with data inputs querying the network for the data output that provides the value.

Indeed, this push-pull system makes intuitive sense when a FB network is analyzed, assuming progression of the network has directionality. FBNs are commonly developed to visually flow from left to right, and with this directionality in mind, events can be seen to propagate rightwards, triggering FBs which consequently pull the required data. High-level code structure of the application can be seen in Algorithm 1. The Event and Data inputs, as well as outputs, are functions which allow the application to directly invoke them. Additionally, the keys to the two maps are functions, enabling FB classes to directly pass their member methods to the application. To allow accessing FBs through the application, a map of FB names to FB instances is required.

Algorithm 1 SimPLE Application

```
class SimPLEApplication
EventConnections: Map(Function → List(Function))
DataConnections: Map(Function → Function)
FunctionBlocks: Map(String → FunctionBlock)
Queue: FIFOQueue(Function)
function EVENT(EOut: List(Function)):
    for Event in EOut do
        for EIn in EventConnections[Event] do
            Queue.PUT(EIn)
    while Queue is not empty do                ▷ Consume the entire queue
        Queue.POP()()

function DATA(DIn: Function):
    return DataConnections[DIn]()

function UPDATEINPUT(DIn: Function):
    DIn(DATA(DIn))

function ADDEVENTCONNECTION(ESrc: Function, EDst: Function):
    add EDst to EventConnections[ESrc]

function ADDDATACONNECTION(DSrc: Function, DDst: Function):
    DataConnections[DDst] = DSrc

function ADDFUNCTIONBLOCK(name: Function, FB: Function):
    FunctionBlocks[name] = FB
```

5.3 Function blocks in SimPLE

Funtion blocks are modeled as classes with a specific format, mirroring the specification of FBs in the IEC 61499 standard. The classes are fully executable code, providing a way to naturally add the simulation-affecting algorithms within them. Five different categories of FBs are included in the SimPLE architecture: Service Interface Function Block (SIFB), Simulated-connected Function Block (SIMFB), Basic Function Block (BFB), Composite Function Block (CFB) and Adapter Function Block (AFB). The functionality matches the functionality described in the standard, with Simulated-connected Function Blocks (SIMFBs) being a subcategory of SIFBs that have direct access to the simulation components.

5.3.1 Technical details

The FB classes have internal variables for all input and output data ports, defined in lowercase, with matching methods in uppercase. To handle the conversion between IEC 61131-3 data types and the implementation language data types, a helper library is used. In the prototype, the Types library has a method for every IEC 61131 data type, which are also used by the exporter to match data ports with their type in the produced

.fbt file. Data input methods have a single *Value* parameter that is assigned to the input variable when a matching method is invoked by converting *Value* through Types. All other methods take no parameters. Data output methods simply return their matching output variable, while event output methods pass themselves to the application for propagation to the connected input event ports. The primary functionality of FBs resides in the event input methods, which implement the WITH-relations of FBs, algorithms and event output invocations. High-level overview of the FB class format is provided in Algorithm 2.

Algorithm 2 SimPLE FB

```
class FB
App ← SimPLEApplication
in1: Any
out1: Any

//Event Inputs
function EI:
    //WITH Relations
    App.updateInput(IN1)
    //Algorithms
    ...
    out1 = in1
    ...
    //Input to Output Event Relations
    EO()

//Event Outputs
function EO:
    App.Event(EO)

//Data Inputs
function IN1(Value: Any):
    in1 = Types.DataType(Value)

//Data Outputs
function OUT1:
    return out1
```

5.3.2 Limitations

Due to the choice of using fully lowercase internal variable names, SimPLE does not support FB definitions that have fully lowercase event or data ports names. For a commercial application, removing this limitation would be a high priority to not introduce restrictions on how to use IDEs. Another quirk emerging from SimPLE lacking a scheduler is the lack of safeguards against infinite loops. IEC 61499 allows chaining the event output of a FB back to a FB that triggers the initial FB. If this

is attempted with SimPLE, the host program will stall indefinitely. However, such constructs can easily be avoided by the control engineer developing the application, and fixing these edge cases is not a high priority.

5.4 SIFBs

IEC 61499 describes a set of generic FBs that implement the most common operations required within PLC programs. These base FBs such as mathematical operations, timers, and event manipulation are implemented as SIFBs in many runtimes, including SimPLE. The SIFB classes match the functionality of equivalent FBs within IDEs and can be used in applications by default. They cannot directly influence the simulation components or state. Most IDEs do not support the creation of new SIFBs directly, instead providing them with libraries that can be imported by the user. SIFBs are the simplest of FBs, following the class definition shown in Algorithm 2 without any modifications.

5.5 SIMFBs

A functionally equivalent subset of SIFBs, with the addition of a reference to their underlying component within the simulation, which allows modifying the component state whenever the SIMFB is executed. Simulation components need to have *FBType* and *ID* properties that match the type and ID of a SIMFB within the IDE, which are used to supply the component reference to the class. SIMFBs are the primary means of integrating application logic into the simulation, and are used in a similar way as hardware-interfacing SIFBs within the IDE. New SIMFBs can be created by end users to provide compatibility with SimPLE for custom components, or provided alongside existing components.

As no other part of the SimPLE architecture has access to the simulation, SIMFBs act as the input and output of the FB network, and can be used to implement human-machine interfaces, sensors or actuators. They are similar in nature to the Composite Automation Type found in some IDEs, and can directly replace CATs or other similar FBs in applications. Additionally, as the SIMFB only requires the interface definition to be used in SimPLE, the interfaces of existing CATs or SIFBs can be used, enabling the use of a control program both on real hardware and inside a simulation without modifications.

5.6 Basic FBs

Basic FBs include all of their functionality within their Execution Control Chart (ECC) and along with CFBs can be created within most IDEs directly. Their specifications are provided in .fbt files in a specific XML format defined by the standard. Each .fbt file is parsed and a FB class is created using the template specified in Algorithm 2 as a base. As all functionality is contained within the ECC, event inputs do not contain algorithms, instead invoking the ECC with the event.

The primary difficulty of implementing BFBs comes from converting the ECC and algorithms into the target language. While IEC 61499 does not specify how the algorithms need to be implemented, the architecture of SimPLE only supports ST for algorithms. In practice, most IDEs support ST for algorithm creation.

5.6.1 ST-to-source compiler

As runtimes are developed with a general-purpose programming languages, conversion from ST to the target language must be done to execute ST code within the runtime. These converters are called Source to Source (S2S) compilers, and have similarities with normal compilers that convert code into machine-executable instructions. The SimPLE architecture does not include a S2S compiler specification. Any implementation that can convert ST source code in text form, parsed from a .fbt file, into executable code in the target language can be used.

5.6.2 Execution Control Chart

ECCs implement a state machine that contain states and transitions describing the operation of a FB. SimPLE implements the ECC as its own class that tracks the states, the current state, transitions between states and any algorithms and event outputs assigned to states. The ECC class is shown in Algorithm 3. As the transitions and algorithms are provided in the FB definition files as text corresponding to ST code, implementing the ECC requires a S2S Compiler. The IEC 61499 standard does not define the language of algorithms, allowing them to be implemented in various programming languages such as ST, Java and C. If the implementation language of SimPLE is the same as the algorithm language, the need for a S2S compiler is reduced. In this thesis, the use of ST for both transitions and algorithms is assumed.

The ECC class itself is lightweight, similar to the FBN Application. Every ECC is linked to an underlying FB, providing access to the data variables. The ECC has a map of all the states and any actions that should happen when the state is entered in a list of methods to invoke, along with the current state. Output events are modeled as a function in the underlying application, allowing both algorithm invocations and output events to be treated equivalently.

An ECC implemented with this structure within the SimPLE framework follows the six postulates proposed in [54], providing a guideline to the behaviour of the ECC. The ECC is always executed by some event, which can be used in the transition statements as a variable. The ECC is recursively executed until no transition clears, with the event being "consumed" after the initial execution.

The second and fifth postulates warrant extra consideration for the constrained environment SimPLE operates in. They state that "Execution cannot be pre-empted by execution of another FB" and "Output events are issued immediately after the corresponding action is completed". In an instant-execution runtime with no scheduler, the two postulates contradict each other, as any input event that would be emitted directly after an action would immediately execute other FBs. The intent of the postulates is to avoid multiple FBs running at the same time, leading to the conclusion

Algorithm 3 ECC

```
class ECC
FB ← FunctionBlock

States: Map(String → List(Function))
State: String //Current state
//Source state to 0..n of (Destination, Guard condition)
Transitions: Map(String → List((String, Function)))
Algorithms: Map(String → Function)
Queue: List(Function) //FIFO Queue

function addState(name: String, actions: List(Function)):
    States[name] = actions
function addTransition(src: String, dst: String, cond: Function):
    add (dst, cond) to Transitions[src]
function addAlgorithm(name: String, algorithm: Function):
    Algorithms[name] = algorithm
function evaluate(cond: Function):
    return cond(FB) //Variable access through FB

function execute(event: String):
    FB → event = True
    for destination, condition in Transitions[State] do
        if condition then
            State ← destination
            for action in States[State] do
                if action in Algorithms then
                    action(FB)
                else
                    Queue.append(action)
            FB → event = False //Event is consumed
            execute() //Run ECC until no transition clears
            break
    FB → event = False
    //Clear the queue by passing output events to the Application
    FB → App.Event(Queue)
```

that the ECC of the initial FB should be fully executed until no transition clears before processing the events in the order they would have been issued in. To implement this behaviour, algorithms of each state are immediately executed on entering the state, while events are added to a queue to be processed in a first-in-first-out manner after all transitions have cleared. Basic FBs in SimPLE can be considered to be atomic. They will always execute their ECC until no transition clears, without allowing any other FBs to execute in the meanwhile.

5.7 Composite FBs

CFBs implement a separate SimPLE application inside them, with the FB providing data and event passthrough in and out of the subapplication, requiring three modifications to the base FB template. The modifications are shown in Algorithm 4.

- Event inputs pass themselves into the subapp after updating data inputs.
- Data inputs become data outputs from the subapplication perspective. When a CFB data input is called without a value, it returns the data value instead of setting it.
- Data outputs query the return value from the subapplication instead of the FB itself.

Additionally, CFBs that contain SIMFBs require access to the simulation to allow linking them to their respective components.

CFBs in SimPLE follow the idea of being a *transparent container for events* as described in [55]. Events can leave the currently executing CFB, even if execution will return to another component FB inside the CFB at a later date without additional CFB input events. This means CFBs are not atomic in their execution. Additionally, the direct passthrough of data connections within the CFB layer is invisible to the FB network, reinforcing the transparent nature of CFBs.

Algorithm 4 Composite function block.

```
App ← SimPLEApplication
Subapp = new SimPLEApplication
//Subapplication initalization
...
Subapp.addFunctionBlock(...)
Subapp.addEventConnection(...)
Subapp.addDataConnection(...)
...
in1: Any
out1: Any

//Event Inputs
function EI:
    App.updateInput(IN1)
    //Event passing to subapplication
    Subapp.event(EI)

//Event Outputs
function EO:
    App.Event(EO)

//Data Inputs with passthrough
function IN1(Value: Any):
    if Value then
        in1 = Types.DataType(Value)
    else
        return in1

//Data Outputs with passthrough
function OUT1:
    return Subapp.Data(out1)
```

5.8 Adapter FBs

Adapters are a bidirectional FB type that can be used to provide connection of multiple event and data inputs easily. As the standard limits adapters to being connected to at most a single other adapter, SimPLE implements them as pairs of mirrored FBs that have access to each other. The Adapter FB class can be either a Plug or a Socket. In socket configuration, inputs and outputs are treated normally, while plugs mirror the distinction, with outputs acting as inputs and inputs being the outputs. As each input and output can be both an input and an output depending on the adapter being a plug or a socket, a similar approach to the CFBs is taken with the functionality of the method changing depending on the value given. Adapters do not have algorithms with event invocations, instead propagating any input event into the mirrored output event. Additionally, the output methods return the input value of the linked adapter instead

of the internal variables. The Adapter class template is detailed in Algorithm 5.

The mirrored nature of adapters can be seen in Figure 4. Any event inputs are propagated to the corresponding event output on the linked adapter, and any data output values are instead queried from the data inputs of the corresponding pair.

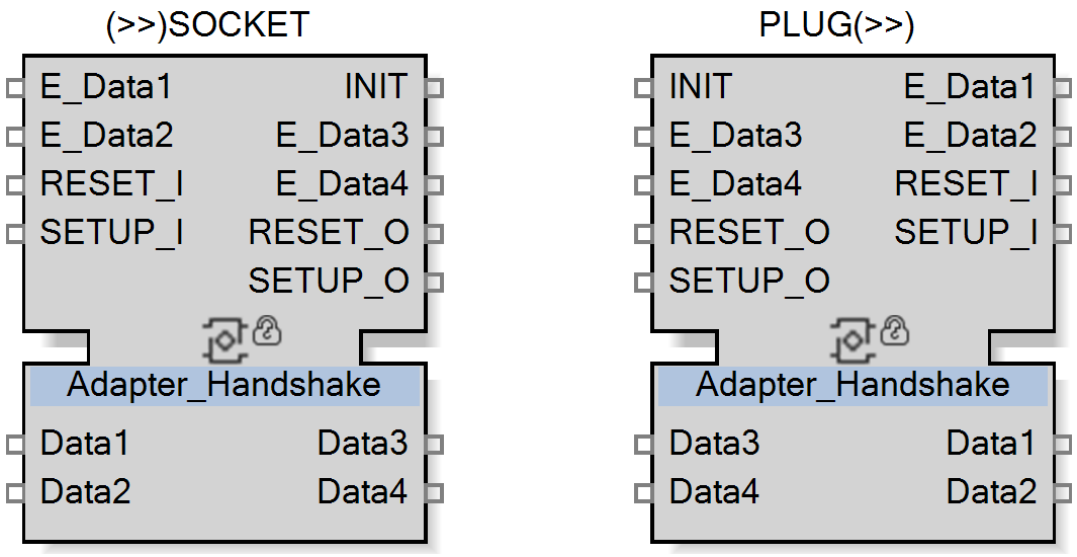

Figure 4: Example adapter pair with mirrored inputs and outputs.

Algorithm 5 Adapter FB

```
App ← SimPLEApplication
Connection ← AdapterFB
Plug ← Boolean //not Plug == socket
in1 = Types.DataType
out1 = Types.DataType
//Event Inputs
function EI:
    if Plug then
        App.Event(EI) //Plug inputs acts as outputs
    else
        App.updateInput(IN1)
        Connection.EI() //Socket inputs invoke the connection output
//Event Outputs
function EO:
    if Plug then
        App.updateInput(OUT1)
        Connection.EO() //Plug outputs act as inputs
    else
        App.Event(EO)
//Data Inputs
function IN1(Value: Any):
    if Plug then
        return Connection.IN1() //Plug inputs act as linked outputs
    else
        if Value then
            in1 = Types.DataType(Value)
        else
            return in1
//Data Outputs
function OUT1(Value: Any):
    if Plug then
        if Value then
            out1 = Types.DataType(Value)
        else
            return out1
    else
        return Connection.OUT1() //Outputs return the linked input
```

5.9 Processor

The Processor is the main interface between the IEC 61499 standard and the SimPLE runtime. It converts the system specification into executable code utilizing the architecture described in this chapter. It is given an IDE specification and a project folder produced by that IDE. The high-level requirements required to create a FBN from the system files can be broken down into three steps:

1. **FB Creation:** FB definitions created with an IDE are processed and turned into classes following the templates covered in this chapter.
2. **System Parsing:** The system file is processed and all FBs, data and event connections are created.
3. **Component Linking:** Simulation components are linked to their matching SIMFBs.

Depending on the implementation language, these steps can be done simultaneously or sequentially. Intepreted languages allow the creation of new class definitions dynamically during execution, while compiled languages will require the preprocessing of FB definitions before compiling the FB network creator. The following sections assume an intepreted language.

5.9.1 FB Builder

The FBBuilder class processes FB definitions and creates concrete implementations of the various FB template classes. It can be seen as a code generator. The detailed architecture of FBBuilder is heavily dependant on the implementation language, generally consisting of XML parsing and file writing. The high-level structure and tasks are detailed in Figure 5, showing the similarities and differences between the FB class templates. The subapplication creation within CFB classes is identical to the initalization of the main FB network, while the ECC creation in basic FBs can be implemented ahead of time or during initialization depending on the language.

The ECC requires the conversion of ST code in the form of algorithms and transitions into functions that can be executed. Compiled languages will require this conversion before the compilation of the FB class. This can be implemented by converting the ST code while writing the class definition, by including the converted source code as methods of the class. Intepreted languages can bypass this step and use the text form of ST in the class source code, which is converted during initalization of the FB. However, this is slow and needs to be done for every instantiation of the FB, even if the resulting code is identical. Due to this, processing all ST code before passing the classes to the runtime is preferable even with the added complexity, even in intepreted languages.

As the builder is completely separate from the runtime, it can be ran independently ahead of loading an application on the runtime. Only running the builder when the .fbt files have changed, especially when processing ST code in this step, drastically speeds up the testing cycle.

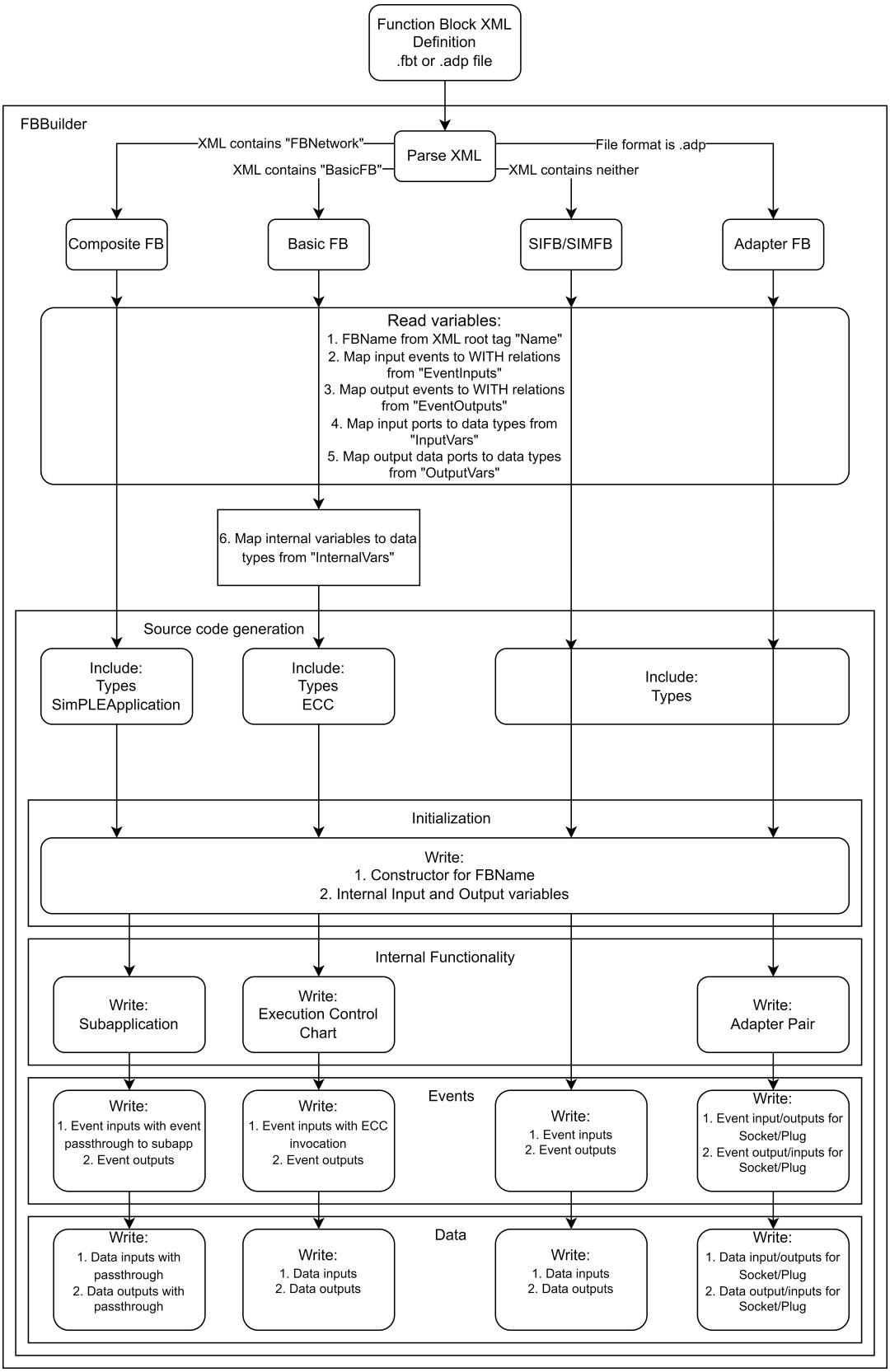

Figure 5: Function Block Builder structure.

5.9.2 Function block network creation

The primary function of the Processor is the creation of a FBN from a .sys file, which contains the FBs of the network and connections between them. Before the FBN creation, required FB types need to be made available to the processor using the FBBuilder. Additionally, as the FB types are in textual form in the system file, a way to convert them into the class objects is required.

The system file has the **Application.SubAppNetwork** element which contains all the required fields for creation of the FB network. SubAppNetwork has four types of elements that need to be processed: **FB**, **EventConnections**, **DataConnections** and **AdapterConnections**. The SimPLEApplication class has matching adder methods for FBs, events and data. The FB definitions can also include Parameter elements, which define constant values in a FBN. The Processor needs to set the internal values of the FB classes to the parameter values after adding a new FB to the FBN. Adapters are implemented by linking the two mirrored adapters contained within the FB definitions into their respective *connection* attribute.

The files also contain device mapping and resource information for use in distributed systems, however it is assumed the environment SimPLE is running in is constrained to a single thread, rendering any attempt at distribution redundant. A further improvement could seek a way to simulate a distributed system in software, however the focus of SimPLE is on running IEC 61499 application logic instead of emulating the possible distributed nature of the applications.

5.9.3 Component links

To allow interaction with the simulation, SIMFB instances need to be bound to the corresponding components. To facilitate this, the processor module requires access to the simulation and all the components within a layout. For VC 4.0, any components that have a matching FB include a property for the FB type name and a numeric ID to allow multiple instances of the same component. Constant values for FB data inputs are available in the .sys files following the standard, and can be used to match components without having to build the entire FB network, allowing the binding of components directly during class instantiation.

5.10 Exporter

To enable the usage of SIMFBs within IDEs, a way to import the FB interfaces is required. The exporter module handles this task, transforming SIMFB classes into FB interface definitions that can be read by IDEs, a reverse of the Processor module. Depending on the IDE, the Exporter can be optional if the user is allowed to create SIFBs directly, such as CATs. The interface specifications contain a matching type name and a data input for an identification number which enable the linking of simulation components with the corresponding FBs.

5.11 Alternative design choices

The architecture described does not use a scheduler that dispatches events, instead processing all event chains continuously until no more events are emitted. The implementation detail that achieves this behaviour is the choice to consume the entire event queue whenever a new output event is invoked, but modifying when the queue is consumed allows adapting the architecture for different execution strategies. One variant would be decoupling the event queue insertion from the consumption by, for example, consuming the entire event queue on every simulation step without allowing the consumption of events that were not present at the start of the step. This would mimic a PLC that operates in a cyclic fashion with scan rate set to be the same as the simulation update rate. The advantages and disadvantages of the various execution strategies in the context of a simulated environment will not be considered in this work. The next chapter will introduce a proof of concept implementation of the architecture described in this chapter, using the instantaneous execution strategy.

6 Prototype

This chapter introduces an existing digital twin and an accompanying IEC 61499 control program that will be used as a benchmark to validate the applicability of the described SimPLE architecture. The architecture is implemented as a Proof of Concept (PoC) runtime in Python, which allows us to execute it directly within Visual Components. The performance and usability of the architecture is compared against a commercial runtime.

6.1 Festo CP Lab

The Festo CP Lab is an Industry 4.0 testbench that provides a modular set of parts for learning and demonstration of Indusry 4.0 concepts [56]. CP Lab is used in multiple university labs [57] [58] [59], including Aalto Factory of the Future [60], to teach and conduct research. It provides a well-documented and flexible framework for exploring concepts such as reconfigurable manufacturing, IIoT and distributed control, providing a small-scale factory with conveyors, assembly stations and controllers. Additionally, Festo provides digital models of the parts, enabling the use of the CPLab in simulation environments.

6.1.1 Visual Components digital twin

A Digital Twin of a CPLab test setup was developed in Aalto as part of their Factory of the Future laboratory, which can be seen in Figure 6. The digital twin consists of a square arrangement of four modules: Two part magazines, one quality control camera and one muscle press to press parts together, along with conveyors connecting the modules. The system receives a part carrier as input, which traverses the four modules and completes the assembly of a mock mobile phone. The assembly can be divided into 7 processing steps:

1. **Carrier insertion:** A carrier pallet is inserted to the CPLab, which can be carried by the conveyors. In the digital twin, the pallet is automatically created on the first conveyor section when simulation is started.

2. **Back cover attachment:** The back cover is attached to the pallet using the first part magazine.

3. **Motherboard insertion:** A human is required to insert a motherboard on the back cover, demonstrating human-machine interaction.

4. **Quality control:** A camera module is used to check for the quality of the back cover and the motherboard. In the digital twin, this step only checks if the back cover and motherboard are found.

5. **Front cover attachment:** The front cover is attached to the pallet using the second part magazine.

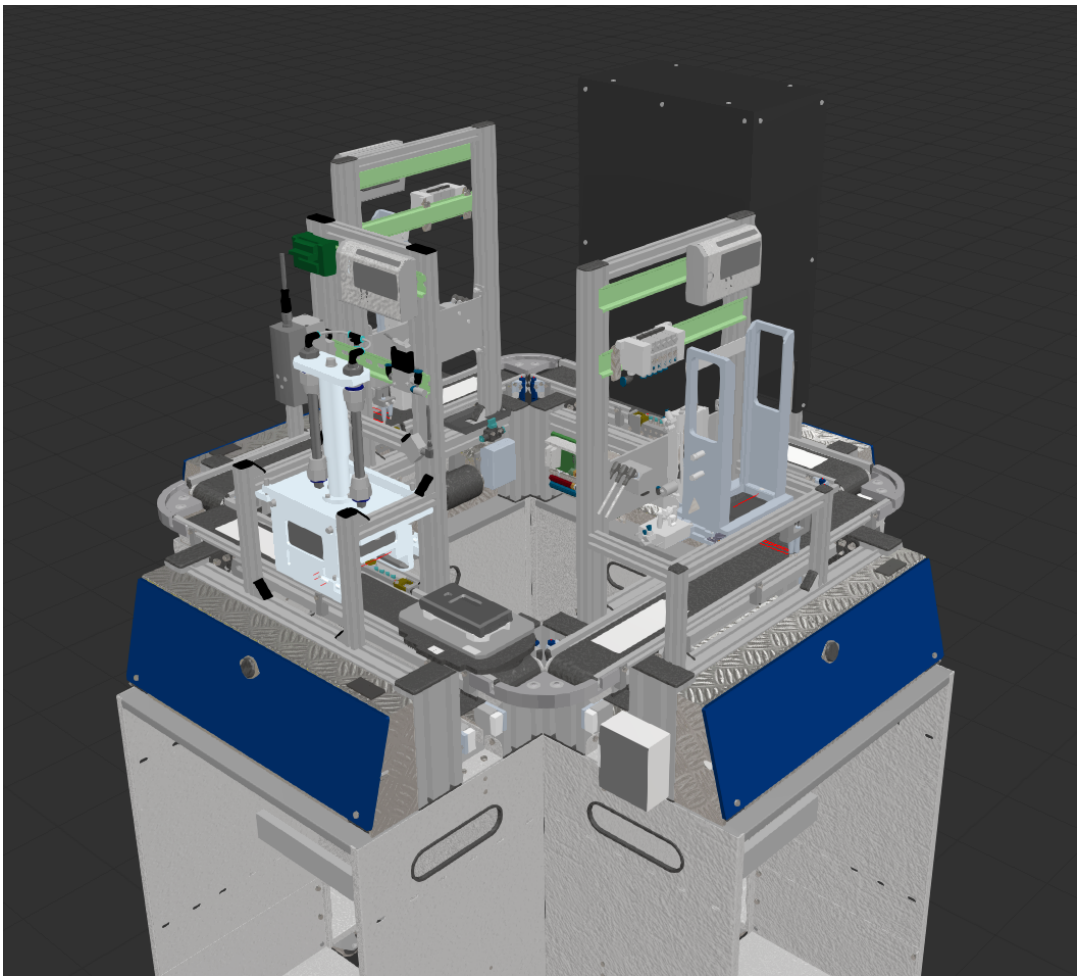

Figure 6: Festo CP Lab in Visual Components

6. **Stamping the assembly:** The two covers are joined together using the muscle press module.

7. **Carrier extraction:** The carrier is removed from the CPLab system, and a new carrier is inserted.

In the digital twin, the insertion and extraction of the carriers and parts is not simulated, with the parts appearing and disappearing as required. Additionally, the DT was modified so a human does not have to insert the motherboard manually.

The control of the digital twin is implemented through boolean signals, which are a feature of VC 4.0 that invoke a method in a component every time any signal of that component changes states. Changes in these signals correspond to, for example, conveyors moving a part or a part feeder attaching a part to a carrier.

6.1.2 Control application

As part of the digital twin, a control application using IEC 61499 was also developed. It was originally designed to be used with a soft-PLC and the Connectivity feature in

VC 4.0 , using an OPC UA server as a bridge, to control the digital twin inside the simulation. The system follows a Service-Oriented Architecture (SOA), implementing the desired functionality using FBs which communicate by requesting services from each other. A high-level view of the system can be seen in Figure 7.

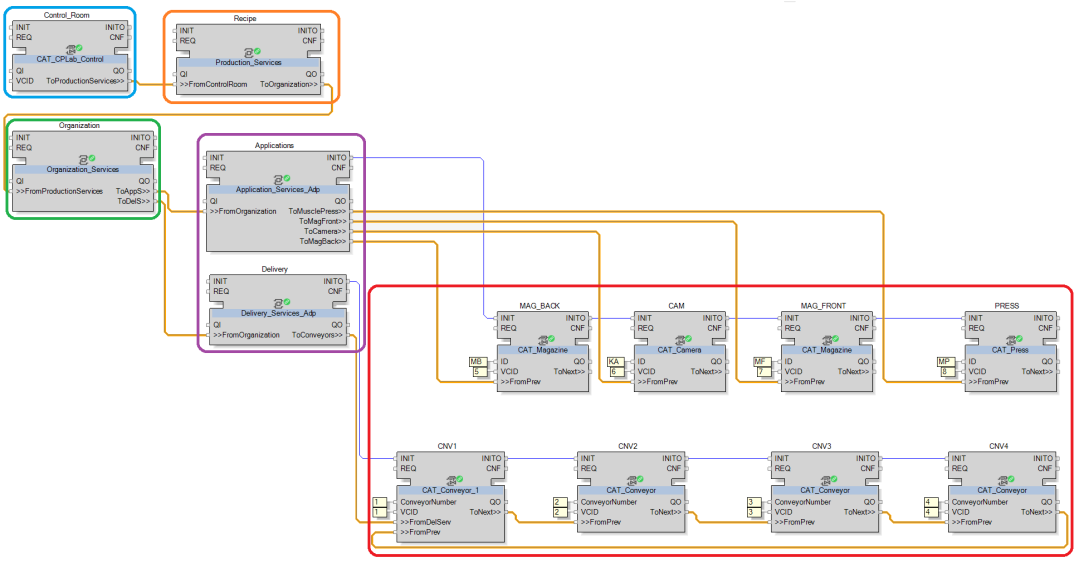

Figure 7: Digital Twin IEC 61499 Application.

The application is split into 5 distinct services. The control room, outlined in red in Figure 7, provides a human-machine interface and tracks the progress of production. The recipe service (Orange) contains the processing steps that need to be done and provides the current task for subsequent services. The organization service (Green) checks if the current task requires moving the pallet or operating one of the modules and routes the task to the appropriate service. The Application and Delivery services (Purple) parse the current task and convert it into commands for modules and conveyors, respectively. Finally, the individual module and conveyor services (Red) receive service requests in the form of command strings, operating the modules.

The recipe and tasks are implemented as short operation codes which describe the current recipe step. A task to move the pallet from one conveyor to another is in the form of "Cn_to_Cm", where n and m are the conveyor numbers, while the task to operate a module is just the unique identifier of the module, for example, "MP" for the Muscle Press module. All communication between the different FBs implementing the services is done using Adapters, which provide a two-way connection. Every module service returns a confirmation signal which is propagated through the services back to the recipe service, which is used to advance the current recipe to the next step when a confirmation signal for the module corresponding to the current task is received.

The control room is used to send signals to the system, such as start, stop and reset commands. It also interfaces with a virtual control room that displays the current step, status of the system and provides buttons to send the signals.

The control application uses the majority of features described in the IEC 61499 standard, providing good coverage for testing the implementation of the proof of

concept. A notable missing feature is the lack of distribution, as the entire application is intended to be ran on a single soft-PLC. However, this is not a problem for the PoC, as we only have access to a single thread to execute the application on, causing proper resource distribution to be unsupported in all cases.

6.2 SimPLE in Python

To validate the design of the architecture and implement it in VC 4.0 , a proof of concept implementation was developed using Python. To the author's knowledge, no IEC 61499 compatible runtime utilizing Python exists. However, IEC 61131 runtimes using Python do exist, such as Beremiz [61]. This is likely due to the relative obscurity of IEC 61499 compared to the widely used 61131-3 standard, combined with the notion of Python being regarded as slow and non-realtime, being ill-suited for PLCs. However, SimPLE implementations are not full runtimes, as the constrained environment they are intended to be executed in allow the omission of multiple important features, such as interfacing with communication standards and handling system resources. Additionally, due to the apparent infinite execution speed from the simulation perspective, the slowness is not a concern.

6.2.1 Software versions

The following software was used for development and testing of the Proof of Concept.

- **Visual Components Premium 4.9** The latest version of Visual Components Premium as of writing, providing the full suite of features. Most importantly, Premium provides access to the Modeling tab for modifying and creating components, which is required to add the parameters required for function block linking. Visual Components 4.9 is still using Python 2.7, consequently leading to the proof of concept using Python 2.7.
- **EcoStruxure Automation Expert 23.0** EcoStruxure Automation Expert (EAE) is a IEC 61499-compatible integrated development environment from Schneider Electric, providing all the features needed to compose and run IEC 61499 applications. The IDE contains a bundled Universal Automation runtime for testing applications on a soft-PLC, providing a reference to compare the PoC with.

6.2.2 Implementation details

The class structure matches the described architecture of SimPLE, with an added class used to match function block type names into Function Block class intances. A full class structure and relations is shown in Figure 8. The application is contained within a PLC component inside VC 4.0 , which implements the coupling with the simulation, discussed in detail in section 6.2.4. The component is given a project folder produced by an IDE, which is processed for any .fbt and .adp file extensions. The FBBuilder is used to convert these files into Python classes following the class structures described

in Chapter 5. Afterwards, the .sys file is passed to the Processor class (Blue in figure), which instantiates a new FBApp which represents the function block network. The network is created by parsing the system file for function blocks, connections and constants. Any additional information contained in the file, such as device mapping, is not required by the architecture. The FBApp additionally receives a *DelayExecutor* instance from the PLC component which can be used by FBs to implement delays and timers. Due to the implementation of the Python environment in Visual Components, the *DelayExecutor* class needs to be a part of a script within a component.

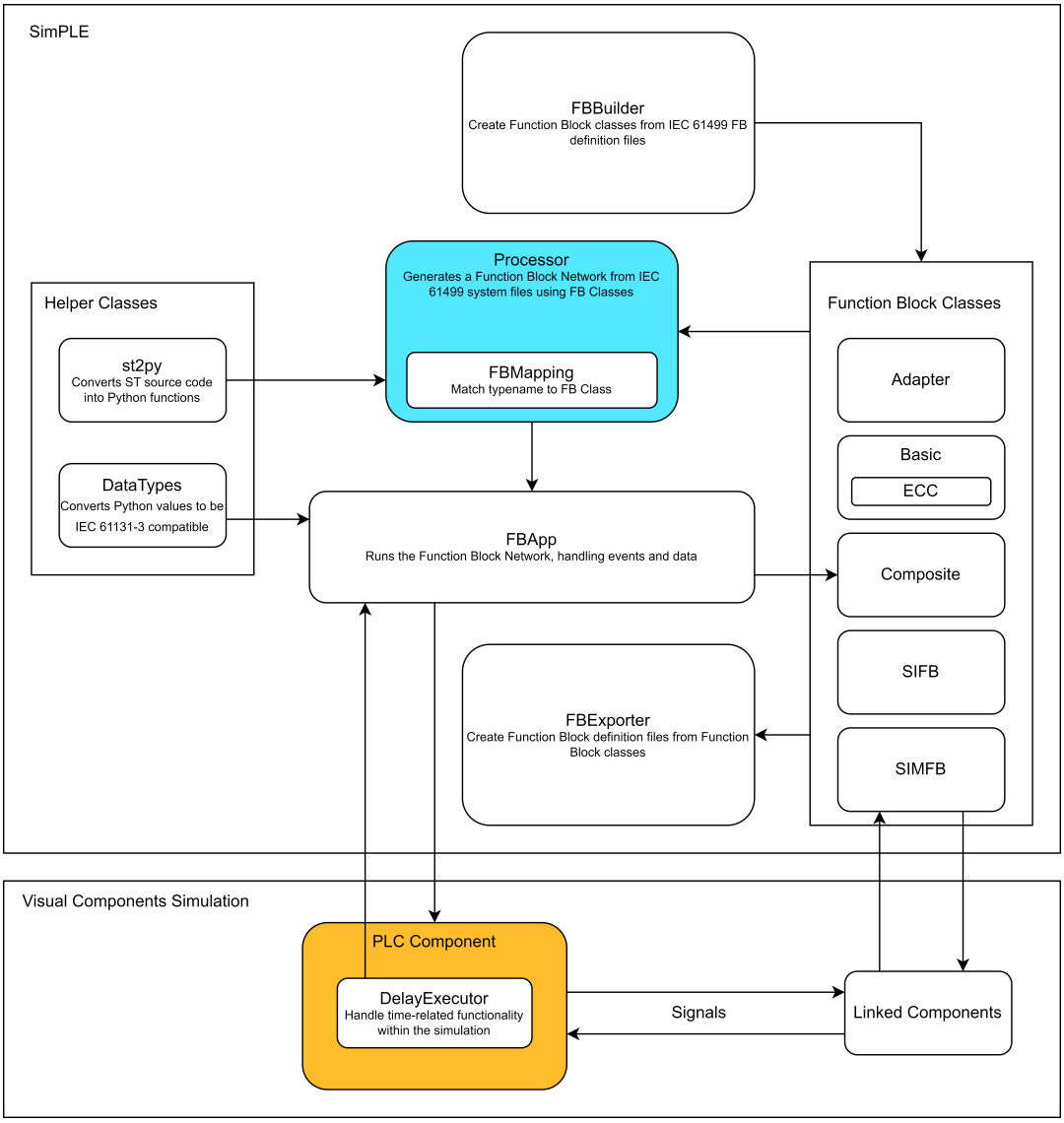

Figure 8: Structure of SimPLE implemented in Python.

6.2.3 ST to Python Compiler

The S2S compiler class required to execute ST code was custom-made for the proof of concept. It provides a single method which takes code as an input and returns a Python Function object which implements the ST source code. The produced functions have a single input parameter, a Function Block object used to access variables. As Python is an intepreted language, the S2S compiler can be used while creating the function block network during runtime. This enables the definitions of Function Block classes to contain ST source code which is converted into Python functions on class instantiation.

Compilers use syntax trees to represent source code in an abstract form, which are used to represent the structure of code. As they generally contain only the necessary (non-language-specific) parts of the code, the term Abstract Syntax Tree (AST) is commonly used. Code in AST form is language-agnostic, and is often used as an intermediary step in compilers.

The operation principle of developed the S2S compiler can be divided into four distinct steps. Initially, the source code is parsed and an AST of the code is generated. Using the AST, a temporary Python function is written in a file by converting the syntax of the tree into Python source code. Next, this temporary Python file is imported, which inteprets the source code and converts it into a Function object, which is loaded in memory. With the function stored within a function block class, the temporary file can be deleted.

One additional consideration is required for the temporary files. Python internally caches all imported classes by name, and uses the cache when a new import matches an existing class. This necessitates the S2S compiler to keep track of how many temporary files have been created and appends the counter to the name of the temporary file to avoid reusing the cached class.

The developed compiler is not fully compliant with all features of Structured Text, missing some of the more modern additions, such as object-oriented improvements added in the third revision of the standard. However, in IEC 61499 applications, ST is primarily used for ECC algorithms and transition conditions, which are generally simple and functional in nature. This observation is reinforced by the ability of the developed compiler to compile all ST code found in the CPLab application.

6.2.4 Implementation in Visual Components 4.0

Using the implemented architecture inside a simulation layout within VC 4.0 required minor changes to the provided Digital Twin. The primary concern was interfacing the signals of the components with the FBN. VC does not provide a way to globally access the Python script of another component, which means the entire FBN has to run inside a single component. For this, a new dummy PLC component was used, which contained the function block network and handled all signal inputs and outputs. This required the duplication and pairing of all signals found within the DT into a single component. The signals were named based on the connected component, and any signal change of a specific component sent an output event of the matching SIMFB within the network.

The described way to connect signals into the FBN is functional, but unintuitive and cumbersome. Further development of the proof of concept into a commercial offering would require the simplifcation of connecting control applications with the simulation, essentially requiring changes within the VC 4.0 software itself which was out of scope for the proof of concept.

Additional VC 4.0 specific detail was the implementation of the delay and timer function blocks. The Python API included with VC does not include a way to delay the execution of a function, and the single-threaded nature of the software prevented the use of a separate thread to implement this. The solution was to include a separate *DelayExecutor* class within the PLC component, which was accessible to the function block network. Any delays or timers could be registered into a queue within the *DelayExecutor* with a time offset, the current simulation time and the event to invoke after the delay. The *DelayExecutor* is continuously executed, and whenever the delta time between the current simulation time and the start time differ by more than the time offset, the registered event is invoked. If the event was registered as a delay, the queue entry is cleared, while a timer registration would refresh the start time to the current simulation time.

The next chapter discusses some limitations of the implemented prototype and compares the performance against a soft-PLC.

7 Results

7.1 Performance of prototype

7.1.1 Missing features

As noted earlier, the SimPLE architecture does not describe a full IEC 61499 runtime, primarily being a way to execute the applications conforming to the standard.

The proof of concept follows the design guidelines given in Chapter 4, with observability being minimally implemented. The PoC has no graphical layer, which makes observing the function block network state difficult. It allows monitoring events and ECC states through terminal messages, which additionally works as an execution trace, but does not provide a complete overview of the current network. Additional development efforts is required with a focus on implementing a graphical way to display the FBN.

As the targeted environment of the application is fully single-threaded due to the underlying simulation, any distribution of the application would necessarily be simulated by software. The IEC 61499 standard already makes a clear distinction between the application logic and its distribution, where the application functionality does not depend on the underlying hardware. Due to this reason, implementing distribution-supporting features was omitted in the proof of concept.

7.1.2 Performance comparison

The functionality of a FBN was able to be replicated using the SimPLE architecture, and the constrained execution environment is indifferent to the processing speed and realtimeness of the implementation due to the discussed behaviours. As such, comparing the raw speed of an integrated runtime implementation to an external hard- or soft-PLC is not immediately useful.

A downside with the current implementation of the PoC is the excessive processing time every time a simulation that contains a moderaly complex function block network is started. Due to the way the S2S compiler was implemented, it converts all algorithms and guard conditions from ST to Python during the creation of the FBN, which can take upwards of half a minute depending on the hardware. A better implementation would be to cache the output of the compiler for later use or process the ST code only when changes are detected instead of every time the network is loaded.

Computationally, the architecture of IEC 61499 is well suited for being ran inside a simulation, as the event-based, asynchronous nature means processing time is not wasted on executing the function block network unless required, preventing excessive slowdown of the simulation even with very large FBNs. Additionally, having the control logic directly integrated with the simulation itself means the controller can never go out of sync with the simulation, which allows us to run control logic at a variable speed. This is a feature absent from all commercial runtimes, which can only run the applications in "real time". The ability to vary the simulation speed, even down to individual simulation ticks, can be useful for testing and developing the application control logic. On the other extreme end, being able to run the simulation with the real

control logic at faster-than-realtime speeds for long periods of time allows testing the robustness of the control application during long production runs.

7.2 Observations from prototype

The proof of concept successfully demonstrates the ability to run IEC 61499 applications with the SimPLE architecture. The functionality of the introduced digital twin was replicated without a need for any external software, showing that execution of PLC programs within a simulation is feasible. The transferability of control programs between real and simulated environments without changes was not tested, and is an important topic for future work. Problems can arise from hardware constraints, such as I/O and processing speeds, which do not exist in the simulation environment and were not considered in this work. However, these constraints could be simulated, possibly increasing the transferability of PLC programs tested in a purely virtual environment.

The constrained environment allowed the runtime to be very simple, with the core FBN logic only requiring less than 80 lines of code to implement in Python. The primary difficulty in implementing the SimPLE architecture comes from the conversion of IEC 61499 compliant system definitions into the targeted language. This simplicity in the core logic makes SimPLE useful for experimenting with various execution strategies and allows adapting the architecture to be closer in execution characteristics with various runtimes.

However, the implementation is not without flaws, primarily due to having no graphical interface of any kind. The lack of a visualization for the function block network makes observing and debugging the application harder compared to runtimes that integrate with an IDE, while the creation of new FBs is currently only possible by directly writing the classes. Implementing a graphical frontend for SimPLE would fix both of these issues and make it much more approachable for the end user.

Some architectural choices introduce artificial limits on the design of function blocks. The decision to use fully lowercase internal variable names and fully uppercase names for event and data ports means a function block cannot have any members with the same name but different capitalization. The IEC standard and most IDEs do allow this, which can cause issues when using SimPLE. Additionally, the standard defines WITH-relations for output ports, a trait that the architecture does not utilize in any way. For stricter conformance with the standard, implementing the relations is a priority.

Additionally, the instant execution trait of SimPLE makes infinite loops, such as two FBs triggering each other in succession, stall the entire program due to the assumed single-threaded nature. In many other runtimes which work in a cyclical fashion, the same infinite loops are possible but do not prevent the execution of everything else. Avoiding such infinite loops is generally the responsibility of the control engineer implementing the application, but alternative execution strategies, such as the tick-based solution, might be more desirable if this behaviour proves problematic.

7.2.1 Applicability

To validate the applicability of the architecture against other runtimes would require comparing the execution trace of both runtimes in identical applications, as discussed in [47]. However, the IEC 61499 ecosystem is lacking a standard suite of applications which would enable the in-depth testing of the differences in execution semantics. As conforming to any specific runtime was not the goal of the architecture, having a large amount of differences due to the environment in the first place, no analytical comparisons were made for this work. The correctness of the architecture was primarily observed using the tested Digital Twin. While the processing steps and end result were identical to the original reference implementation, the execution trace of the function block network itself could be completely different. Testing on a larger set of various applications would be required to identify differences between the SimPLE implementation and the used reference runtime.

8 Conclusions

The goal of this master's thesis was to investigate ways to improve IEC 61499 support of Visual Components for virtual commissioning purposes. Implementing a runtime to execute IEC 61499 applications was chosen, as it was already identified as a promising option in another master's thesis done at Visual Components [12].

To achieve this goal, a new architecture was introduced for executing IEC 61499 applications within constrained environments, primarily intended for simulation applications. The capability to run a function block network with the architecture was successfully demonstrated using an implementation of the architecture written in Python, which was used to control a digital twin using a pre-existing control program. The entirety of the logic was able to be replicated within the simulation, showing promise for tightly coupling the simulation and its control logic. Due to the unorthodox environment compared to normal PLCs, quirks and limitations of the architecture were discovered. These included the lack of physical constraints present in real-world PLCs, such as signal transmission delays or non-instantaneous execution of programs.

These differences from the actual hardware makes coupling the control application with the simulation useful for certain tasks, such as validating the correctness of control logic and developing the logic alongside the simulation layout, but cannot reliably be used to guarantee the applicability of the control applications in real-world situations due to the different timing characteristics. Such behaviour is still useful for virtual commissioning, allowing control engineers to contribute from the start of the layout creation process instead of having to wait for a finished simulation before developing the control code, but introduces a need for additional verification without the simulation, either on physical hardware or a separate soft-PLC that more accurately mimics the realities of the hardware. Nevertheless, the integration of a IEC 61499 runtime to the simulation allows for a smoother workflow compared to using a soft-PLC with co-simulation, enabling faster-than-realtime simulation, direct integration with the simulation environment and not requiring a third-party runtime.

To the author's knowledge, this work is also the first IEC 61499 runtime environment using Python. Additionally, this work illustrates the simplicity of the core of the IEC 61499, showing that a compatible architecture capable of executing applications specified by the standard can be implemented very compactly. This lightweight nature of the architecture could have value in providing an accessible way of teaching IEC 61499 runtimes, or as a base for studying different implementations of the execution mechanics.

Some drawbacks in the implementation were also identified, which will need to be improved in further work. The lack of a graphical user interface for observing the FBN make the usage of SimPLE cumbersome. The custom nature of it and the incomplete implementation of all features of IEC 61499 can cause issues with compatibility when working with commercial runtimes, and solving these issues would require further research and developement of the architecture. Additionally, SimPLE was only tested with a single programming language using a specific simulation software. More research on how well the architecture generalizes to other languages and environments is required.

References

[1] R. Drath *et al.*, "An evolutionary approach for the industrial introduction of virtual commissioning", in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, Sep. 2008, pp. 5–8. DOI: 10.1109/ETFA.2008.4638359.

[2] J. B. Sim *et al.*, "Full Stack Virtual Commissioning: Requirements Framework to Bridge Gaps in Current Virtual Commissioning Process", in *International Manufacturing Science and Engineering Conference*, vol. 87240, American Society of Mechanical Engineers, 2023, V002T07A016.

[3] H. Carlsson *et al.*, "Methods for reliable simulation-based PLC code verification", *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 267–278, 2012.

[4] K.-H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, ISBN: 978-3-662-07849-5 978-3-662-07847-1. DOI: 10.1007/978-3-662-07847-1.

[5] J. Morgan *et al.*, "Industry 4.0 smart reconfigurable manufacturing machines", *Journal of Manufacturing Systems*, vol. 59, pp. 481–506, Apr. 1, 2021, ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2021.03.001.

[6] K. Thramboulidis, "IEC 61499 vs. 61131: A Comparison Based on Misperceptions", *Journal of Software Engineering and Applications*, vol. 06, no. 08, pp. 405–415, 2013, ISSN: 1945-3116, 1945-3124. DOI: 10.4236/jsea.2013.68050. arXiv: 1303.4761 [cs].

[7] G. Lyu and R. W. Brennan, "Towards IEC 61499-based distributed intelligent automation: A literature review", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2295–2306, 2020.

[8] "Universal Automation - A call for change | Schneider Electric", [Online]. Available: https://www.se.com/ww/en/download/document/998-21066447/.

[9] A. Hopsu, "Portability of IEC 61499 compliant software", M.S. thesis, 2019.

[10] P. Lindgren *et al.*, "A formal perspective on IEC 61499 execution control chart semantics", in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, IEEE, 2015, pp. 293–300.

[11] A. Hopsu *et al.*, "On portability of IEC 61499 compliant structures and systems", in *2019 IEEE 28th International Symposium on Industrial Electronics (ISIE)*, IEEE, 2019, pp. 1306–1311.

[12] T. Syväjärvi, "Requirements of Simulation Software for Virtual Commissioning of Discrete Manufacturing Systems", M.S. thesis, Tampere University of Technology, 2016.

[13] Z. Liu *et al.*, "Virtual Commissioning of Automated Systems", in *Automation*, F. Kongoli, Ed., InTech, Jul. 25, 2012, ISBN: 978-953-51-0685-2. DOI: 10.5772/45730.

[14] M. Schamp *et al.*, "Impact of a virtual twin on commissioning time and quality.", *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1047–1052, 2018.

[15] N. Striffler and T. Voigt, "Concepts and trends of virtual commissioning–A comprehensive review", *Journal of Manufacturing Systems*, vol. 71, pp. 664–680, 2023.

[16] F. Tao *et al.*, "Digital Twin in Industry: State-of-the-Art", *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, Apr. 2019, ISSN: 1941-0050. DOI: 10.1109/TII.2018.2873186.

[17] L. Naciri *et al.*, "Modeling and simulation: A comparative and systematic statistical review", *Procedia Computer Science*, 5th International Conference on Industry 4.0 and Smart Manufacturing (ISM 2023), vol. 232, pp. 242–253, Jan. 1, 2024, ISSN: 1877-0509. DOI: 10.1016/j.procs.2024.01.024.

[18] R. Spence, *Information Visualization: An Introduction*. Cham: Springer International Publishing, 2014, ISBN: 978-3-319-07340-8 978-3-319-07341-5. DOI: 10.1007/978-3-319-07341-5.

[19] M. Soori *et al.*, "Digital twin for smart manufacturing, A review", *Sustainable Manufacturing and Service Economics*, p. 100 017, 2023.

[20] M. Macchi *et al.*, "Exploring the role of Digital Twin for Asset Lifecycle Management", *IFAC-PapersOnLine*, 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018, vol. 51, no. 11, pp. 790–795, Jan. 1, 2018, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2018.08.415.

[21] "COLLADA - 3D Asset Exchange Schema". (Jul. 19, 2011), [Online]. Available: https://www.khronos.org//.

[22] J. H. Christensen *et al.*, "The IEC 61499 function block standard: Software tools and runtime platforms", *ISA Automation Week*, vol. 2012, 2012.

[23] V. Vyatkin, *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. ISA, 2007.

[24] L. Prenzel *et al.*, "IEC 61499 Runtime Environments: A State of the Art Comparison", in *Computer Aided Systems Theory – EUROCAST 2019*, R. Moreno-Díaz *et al.*, Eds., vol. 12014, Cham: Springer International Publishing, 2020, pp. 453–460, ISBN: 978-3-030-45095-3 978-3-030-45096-0. DOI: 10.1007/978-3-030-45096-0_55.

[25] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review", *IEEE transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[26] S. Campanelli *et al.*, "An architecture to integrate IEC 61131-3 systems in an IEC 61499 distributed solution", *Computers in Industry*, vol. 72, pp. 47–67, Sep. 2015, ISSN: 01663615. DOI: 10.1016/j.compind.2015.04.002.

[27] P. Gsellmann *et al.*, "A Novel Approach for Integrating IEC 61131-3 Engineering and Execution Into IEC 61499", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5411–5418, Aug. 2021, ISSN: 1551-3203, 1941-0050. DOI: 10.1109/TII.2020.3033330.

[28] J. H. Christensen *et al.*, "The IEC 61499 function block standard: Overview of the second edition", *ISA autom. Week*, vol. 6, pp. 6–7, 2012.

[29] P. Tata and V. Vyatkin, "Proposing a novel IEC61499 runtime framework implementing the cyclic execution semantics", in *2009 7th IEEE International Conference on Industrial Informatics*, IEEE, 2009, pp. 416–421.

[30] K. Thramboulidis, "Design Alternatives in the IEC 61499 Function Block Model", in *2006 IEEE Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic: IEEE, Sep. 2006, pp. 1309–1316, ISBN: 978-0-7803-9758-3. DOI: 10.1109/ETFA.2006.355411.

[31] A. Zoitl and V. Vyatkin, "Different perspectives [Face to face; "IEC 61499 architecture for distributed automation: The '"glass half full" view", *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 7–23, 2009, ISSN: 1932-4529. DOI: 10.1109/MIE.2009.934789.

[32] V. Vyatkin and V. Dubinin, "Execution Model of IEC61499 Function Block",

[33] A. Zoitl and V. Vyatkin, "IEC 61499 architecture for distributed automation: The '"glass half full" view", *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 7–23, 2009, ISSN: 1932-4529. DOI: 10.1109/MIE.2009.934789.

[34] A. Barni *et al.*, "Building an Automation Software Ecosystem on the Top of IEC 61499", in *The Digital Shopfloor-Industrial Automation in the Industry 4.0 Era*, River Publishers, 2022, pp. 339–364.

[35] T. Strasser *et al.*, "Design and execution issues in IEC 61499 distributed automation and control systems", *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 1, pp. 41–51, 2010.

[36] V. Vyatkin *et al.*, "Alternatives for execution semantics of IEC61499", in *2007 5th IEEE International Conference on Industrial Informatics*, vol. 2, IEEE, 2007, pp. 1151–1156.

[37] C. Sunder *et al.*, "Usability and Interoperability of IEC 61499 based distributed automation systems", in *2006 4th IEEE International Conference on Industrial Informatics*, Aug. 2006, pp. 31–37. DOI: 10.1109/INDIN.2006.275713.

[38] W. E. Rumpl *et al.*, "Platforms for Scalable Flexible Automation Considering the Concepts of IEC 61499", in *Knowledge and Technology Integration in Production and Services*, V. Mařík *et al.*, Eds., Boston, MA: Springer US, 2002, pp. 237–246, ISBN: 978-1-4757-5632-6 978-0-387-35613-6.

[39] "IEC 61499-4:2013", IEC 61499-4 Function blocks - Part 4: Rules for compliance profiles, [Online]. Available: https://webstore.iec.ch/en/publication/5508.

[40] T. Lyu *et al.*, "Mechatronic Swarm and its Virtual Commissioning", in *2023 IEEE International Conference on Mechatronics (ICM)*, IEEE, 2023, pp. 1–6.

[41] F. Basile *et al.*, "On the implementation of industrial automation systems based on PLC", *IEEE Transactions on Automation Science and Engineering*, vol. 10, no. 4, pp. 990–1003, 2012.

[42] T. Lyu, "Evaluation of Containerized Simulation Software in Docker Swarm and Kubernetes", M.S. thesis, 2020.

[43] T. Lyu *et al.*, "Methods of data streaming from IEC 61499 applications to Cloud storages", in *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)*, IEEE, 2023, pp. 1–6.

[44] T. Lyu *et al.*, "On automatic generation of OPC UA connections in IEC 61499 automation systems", in *2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)*, IEEE, 2022, pp. 1–6.

[45] T. Lyu *et al.*, "On enhancing reconfigurability of I/O connection and access in lEC 61499", in *2022 IEEE 31st International Symposium on Industrial Electronics (ISIE)*, IEEE, 2022, pp. 818–823.

[46] I. Hegny *et al.*, "IEC 61499 based simulation framework for model-driven production systems development", in *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, Bilbao: IEEE, Sep. 2010, pp. 1–8, ISBN: 978-1-4244-6848-5. DOI: 10.1109/ETFA.2010.5641364.

[47] B. Wiesmayr *et al.*, "Close Enough? Criteria for Sufficient Simulations of IEC 61499 Models", in *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, Auckland, New Zealand: IEEE, Aug. 26, 2023, pp. 1–7, ISBN: 9798350320695. DOI: 10.1109/CASE56687.2023.10260555.

[48] B. Dowdeswell *et al.*, "simbIoTe: A Simulator for Building Cyber-Physical System and Internet of Things Environments", in *2022 IEEE 1st Industrial Electronics Society Annual On-Line Conference (ONCON)*, kharagpur, India: IEEE, Dec. 9, 2022, pp. 1–6, ISBN: 9798350398069. DOI: 10.1109/ONCON56984.2022.10126940.

[49] M. Colla *et al.*, "Applying the IEC-61499 model to the shoe manufacturing sector", in *2006 IEEE Conference on Emerging Technologies and Factory Automation*, IEEE, 2006, pp. 1301–1308.

[50] H. Pearce *et al.*, "Faster Function Blocks for Precision Timed Industrial Automation", in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, Singapore: IEEE, May 2018, pp. 67–74, ISBN: 978-1-5386-5847-5. DOI: 10.1109/ISORC.2018.00017.

[51] H. A. Pearce *et al.*, "Simulation of cyber-physical systems using IEC61499", in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, Vienna Austria: ACM, Sep. 29, 2017, pp. 136–145, ISBN: 978-1-4503-5093-8. DOI: 10.1145/3127041.3127052.

[52] J. L. M. Lastra *et al.*, *Function blocks for industrial-process measurement and control systems*. 2004, ISBN: 978-952-15-1244-5.

[53] "4diac FORTE - The 4diac runtime environment", [Online]. Available: https://eclipse.dev/4diac/en_rte.php.

[54] V. Vyatkin and V. Dubinin, "Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499", in *2007 5th IEEE International Conference on Industrial Informatics*, vol. 2, Jun. 2007, pp. 1183–1188. DOI: 10.1109/INDIN.2007.4384943.

[55] C. Sunder *et al.*, "Execution Models for the IEC 61499 elements Composite Function Block and Subapplication", in *2007 5th IEEE International Conference on Industrial Informatics*, vol. 2, Jun. 2007, pp. 1169–1175. DOI: 10.1109/INDIN.2007.4384941.

[56] "CP Lab | Festo", [Online]. Available: https://www.festo.com/fi/en/e/technical-education/training-concepts/highlights/training-factories/cp-systems-all-round-i4-0-training-factories/cp-lab-id_36133/.

[57] S. Szafraniec, "Designing and implementing a carrying device for the Smart Factory in the Kosmos Lab", 2023.

[58] M. Nardello *et al.*, "The smart production laboratory: A learning factory for industry 4.0 concepts", in *CEUR Workshop Proceedings*, vol. 1898, CEUR Workshop Proceedings, 2017.

[59] O. Madsen and C. Møller, "The AAU smart production laboratory for teaching and research in emerging digital manufacturing technologies", *Procedia Manufacturing*, vol. 9, pp. 106–112, 2017.

[60] "Aalto Factory of the Future", [Online]. Available: https://www.aalto.fi/en/futurefactory.

[61] "Beremiz", [Online]. Available: https://beremiz.org/.