TECHNISCHE
UNIVERSITÄT
WIEN

AUTOMATION & CONTROL INSTITUTE
INSTITUT FÜR AUTOMATISIERUNGS-
& REGELUNGSTECHNIK

Hierarchical Component-Based Programming of Control Systems

DIPLOMARBEIT

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

unter der Leitung von

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

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

von

Iñigo Alvarez, BSc.
Matrikelnummer: 11740451

Wien, im September 2018

Advanced Mechatronic Systems Group
Gußhausstrasse 27-29, A-1040 Wien, Internet: http://www.acin.tuwien.ac.at

Acknowledgements

First of all, I want to thank both my advisor Univ.-Prof. Dr.sc.techn. Georg Schitter and my co-advisor Dipl.-Ing. Martin Melik-Merkumians for giving me the opportunity to work for my master thesis in the Advanced Mechatronic Systems group at ACIN. Without them I could not have worked in a field that I am passionate about in the beautiful city of Vienna. I am very grateful for their patience and understanding with an exchange student, with all the extra administration issues that this condition entails. On the technical aspect, they did a great job in guiding me through my thesis and always helped me with their expertise. I also want to express my gratitude to Peter Gsellmann from the AMS group for his technical advice during the thesis. Many thanks as well to all the staff at the Mechatronics and Embedded Control Systems research group at KTH in Stockholm, my home university, for giving me the opportunity to do my master thesis at TU Wien. In this aspect, I am really grateful as well to all the staff at the ITM Schools Office of Students Affairs in KTH for their great help regarding the administration of my exchange period at TU Wien. Last but not least, I want to thank my girlfriend Kathrin, who shared these incredible months in Vienna with me, and my parents Alberto and Iciar for their great support during all this time.

Vienna, September 2018 Iñigo Alvarez

Abstract

Current programming techniques in industrial automation tend to mix control logic with hardware implementation, making it difficult to reuse the control logic during the plant life cycle if a certain hardware component has to be replaced for a different one. Hierarchically structured and component-based design approaches have been implemented in IEC 61131 and IEC 61499 with the goal of decoupling control logic from hardware implementation, thus enabling the reuse of the components which encapsulate control logic. In this work we show how these state of the art approaches still do not succeed in creating reusable components at every level in the component hierarchy, and therefore we propose a new component-based design approach that further decouples control logic from hardware implementation. This new approach is implemented in IEC 61499 in order to compare it against the state of the art approach and show the improvements in component reuse and plant reconfiguration effort.

Kurzfassung

Aktuelle Programmierpraktiken in der industriellen Automation tendieren zu einer Kombination aus Logik- und Hardwareimplementierung, die die Wiederverwendbarkeit der Steuerungslogik bei Austausch gewisser Teile während dem Lebenszyklus einer Anlage erschweren. Hierarchisch strukturierte und komponentenbasierte Entwurfsansätze sind in IEC 61131 und IEC 61499 implementiert worden, mit dem Ziel der Entkopplung von Steuerungslogik und Hardwareimplementierung. Diese Arbeit zeigt, dass diese State of the Art Ansätze noch immer keine Erfolge in der Erstellung von wiederverwendbaren Komponenten in jeder Ebene der Komponentenhierarchie verzeichnen können. Als Lösungskonzept wird ein neuer komponentenbasierter Entwurfsansatz mit weitgehender Entkopplung von Steuerungslogik und Hardwareimplementierung vorgeschlagen. Die Implementierung dieses neuen Ansatzes wird in IEC 61499 vorgenommen um einen Vergleich mit der State of the Art Methode zu ermöglichen. Hierbei können die Verbesserungen in der Wiederverwendbarkeit von Komponenten und der Aufwand der Anlagenrekonfiguration gezeigt werden.

Contents

Acronyms

4diac	Framework for Industrial Automation & Control.
BFB	Basic Function Block.
CBSE	Component-Based Software Engineering.
CFB	Composite Function Block.
DSL	Domain-Specific Language.
ECC	Execution Control Chart.
FB	Function Block.
FBD	Function Block Diagram.
FORTE	4DIAC RunTime Environment.
GA	Generalized Actuator.
GD	Generalized Device.
HSIFB	Hardware-Specific Implementation Function Block.
ICP	Instrumentation and Control-Point.
IDE	Integrated Development Environment.
IEC	International Electrotechnical Commission.
IL	Instruction List.

IO	Input/Output.
LD	Ladder Diagram.
LED	Light-Emitting Diode.
LSFB	Logical Service Function Block.
MDA	Model-Driven Architecture.
OOP	Object-Oriented Programming.
PDM	Platform-Dependent Model.
PIM	Platform-Independent Model.
POU	Program Organisation Unit.
PSM	Platform-Specific Model.
PTP	Point-to-Point.
ROOM	Real-Time Object-Oriented Modeling.
SFC	Sequential Function Chart.
SIFB	Service-Interface Function Block.
ST	Structured Text.

List of Figures

List of Listings

List of Tables

CHAPTER 1

Introduction

Today's rapidly changing industry demands flexible production plants. The manufacturing industry requires production systems that are able to quickly adapt to new requirements. The reconfiguration of production systems implies the change of mechanical structures of the plant. Consequently, changes in both the electrical setup and the control software application are required [2].

In the development of control applications for industrial automation, the rapid changes of the industry demand fast reconfiguration of the control applications. During the development of control applications, one the biggest challenges faced is the intrinsic hardware dependability of the developed control applications. Traditional control application programming techniques tend to mix logical functionality with hardware access methods [3], increasing the complexity of the software applications. This issue translates into significant system development times and increased costs in the automation and control engineering of production plants. For example, in the automotive sector's production plants, the development tasks related to software make up 55% of the total costs [4]. However, recent studies show that by optimizing the overall engineering process using the proper methods, architectures and tools the engineering effort can be reduced up to a 70% [5].

In the industrial automation field, current control systems are based on the IEC 61131-3 standard. A new standard for the development of control systems, IEC 61499 (first published in 2005), is said to improve some of the deficiencies of IEC 61131-3 by considering three important properties: *interoperability*, *portability* and *configurability* [6]. However, automation programs developed accordingly to these two standards are still quite platform dependent and software reuse is complicated. Most of the times, developers of automation programs "copy & paste" code from a previously developed application, or use this previously developed application as a template for the new control logic. In software engineering, modular organization aims at reuse of software components. Such component-based approach could be implemented in the design of control applications in the industrial automation field to create hierachically structured and modular applications that help to split the control logic from the hardware implementation and improve software reuse. The implementation of the component-based approach is, however, not straightforward in automation programs. Components may, for example, require some data or control signals from other components. In addition, nested structures of components may create hidden links between reusable modules [7].

1.1 Scope of the Thesis

The thesis work can be divided into three parts. The first part consists in researching the state of the art approaches that help to build modular and reusable control applications. The research focuses on approaches that are component-based and aim to decouple the logic and hardware-specific parts in industrial control applications developed under the IEC 61131 and IEC 61499 standards.

The second part of the thesis starts with a case study in which the state of the art approaches will be tested on a prototype in order to identify the current problems with such approaches. Then, a generic design concept will be presented, with the aim of solving the problems identified with the state of the art approaches. This design approach shall be component-based, hierarchically structured and enhance software reuse by further decoupling logic from hardware implementation in the control application. Since it is a generic design approach, specific IEC 61131 and IEC 61499 implementations can be derived from it.

The last part of the thesis consists in the implementation of the proposed design approach for IEC 61499. This implementation will be tested in a case study, over the same prototype as the one used to test the state of the art approaches. The goal is to then compare both approaches in order to evaluate the proposed concept.

1.2 Outline

The thesis starts with a state of the art research in Chapter 2, which focuses on component-based and hierarchically structured design approaches for IEC 61131 and IEC 61499 applications. In Chapter 3, a case study is conducted, in which the state of the art design approach is implemented on a real prototype in order to analyze the limitations of this design approach. At the end of the chapter, a generic design approach that enhances component reuse by overcoming the state of the art limitations is presented. This generic design approach is implemented in IEC 61499 in Chapter 4. The case study that was initially presented in Chapter

3 continues at the end of Chapter 4, by implementing the new design approach over the same prototype. The results of the case study are presented in Chapter 5, comparing the state of the art approach with the proposed design approach. The thesis ends in Chapter 6, presenting the conclusions of the research and work performed.

CHAPTER 2

State of the Art

In this section the state of the art research starts with a look into the software engineering field, in order to get an idea of how this field approaches modular and reusable component-based systems. Then the IEC 61131 and IEC 61499 standards are presented, in order to understand the following sections which review the state of the art component-based hierarchical design approaches for both standards.

2.1 Approaches and Design Methodologies from Software Engineering

A lot of interesting concepts from software engineering can be applied in the design of control applications in order to achieve more modular and reusable control programs. Perhaps the most interesting concept from the control application development point of view is the programming paradigm called Object-oriented Programming (OOP). Some of the programming languages that support OOP are C++, Java, and Python. Object-oriented programming is an approach for designing modular reusable software systems. Thus, some of the characteristics of OOP can be very useful in the design of modular and reusable control applications.

The main concept in OOP is known as *object*. The object concept can be seen in different ways. The most interesting view for a control application is an object as a logical machine, that is, an active component that can be implemented as software, hardware or non-electronic. This view of an object in particular, and the OOP paradigm in general is the base of Real-Time Object-Oriented Modeling (ROOM), a modelling language developed in the early 1990s for modeling real-time systems [8]. In ROOM, an object has an encapsulation shell, and the communication between objects is based on a message-passing model. The models are represented in terms of class definitions. This class definitions can then be implemented as objects. Also, inheritance can be used when defining a new class that shares some properties in common with a previously created class. If applying inheritance recursively, and ordered structure of class definitions is achieved, known as *inheritance hierarchy*. To sum up, an object in ROOM is defined as an independently active logical machine with an encapsulation shell.

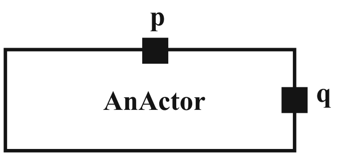


Figure 2.1 – Graphical notation of an actor reference in ROOM.

A key element in ROOM is the *actor*. An actor is the interpretation of the object concept from the OOP in ROOM. The basic interface of an actor is depicted in Figure 2.1, where p and q represent ports. Ports in ROOM are used for communication between actors. Actors must be capable of having a state. The internal operation of an actor over time is referred to as its behaviour. The high-level behaviour of an actor over time is represented by an extended state machine called *ROOMchart*. The state machines in ROOM can perform actions when a transition is taken, a state is entered or a state is exited. Also, ROOM provides for hierarchical state machines to any desired depths. An example of a *ROOMchart* is depicted in Figure 2.2. An actor in ROOM as both a structure, as seen in Figure 2.1, and behaviour, as depicted in Figure 2.2.

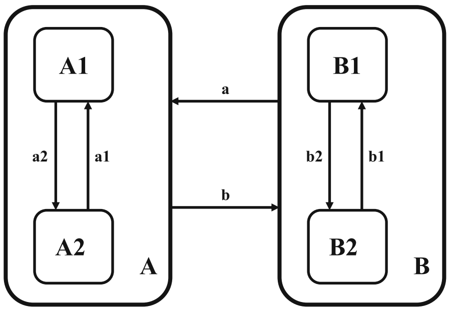


Figure 2.2 – A *ROOMchart* with 6 states and 6 transitions.

In ROOM a model is a collection of three class definitions: actor class definitions, protocol class definitions and data class definitions. An actor class definition has a structure and a behaviour, as previously mentioned. The ports in the actor structure are references to protocol class definitions. The data classes in ROOM are used to declare data objects that are encapsulated within actors. A ROOM model is, in effect, a source program in a very high-level language and can be executed.

A single actor class definition in ROOM can have multiple references (also called instances in OOP). An actor class definition can contain other actors. In the example of Figure 2.3, an actor of class *DyeingSystem* contains the following actors: *Valve*, *DyeingRunController*, *DyeingSolution* and *DyeingSpecifications*. Also, there are two actors of class *Valve*, one that plays the role of *drainValve* and another one that plays the role of *dyeValve*.

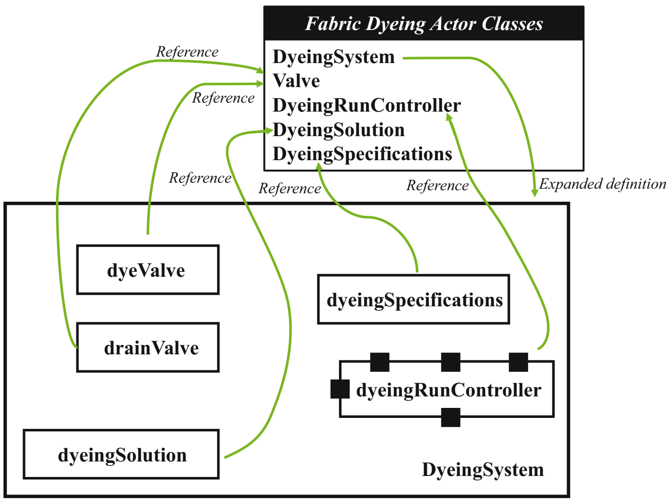


Figure 2.3 – Example of a hierarchical control application modelled in ROOM. Adapted from [8].

The main interest of ROOM for this thesis is the use of OOP concepts in order to achieve hierarchically structured and modular control programs. The actor concept in ROOM enables the construction of hierarchical structures by composition and layering. The authors of the ROOM methodology show how to create hierarchically structured and reusable control applications by applying some of the OOP paradigm concepts.

Another interesting concept from software engineering that can be applied in the development of control applications is the concept of *software components*, which are the key element in Component-based Software Engineering (CBSE). The main idea in CBSE is that software components enable practical reuse of software parts. To be more specific, software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system [9].

In the OOP, an object encapsulates state and behaviour and the mechanisms that this paradigm provide in order to achieve a hierarchical construction and reusable software units are polymorphism and inheritance. However, OOP lacks a view on independence and late composition. This is where the component-based paradigm complements the OOP. Components are units of independent deployment and third-party composition.

2.2 An Overview of IEC 61131

Traditional PLC programs have been developed according to the IEC 61131 programming languages, and this standard is still nowadays the preferred choice in the industry. Part 3 of the standard, known as IEC 61131-3, was first published in 1993.

This standard was developed to overcome the limitations of the ladder programming previously used for PLCs, as well as to improve software quality (capability, availability, usability and adaptability) [1]. Ladder programming evolved from the electrical wiring diagrams used in the car industry for describing relay control schemes. This programming technique became popular among maintenance engineers, since it allowed faults to be quickly traced. It was as well easy to understand for people who are familiar with simple electronic or electrical circuits. Therefore, it was well accepted by electricians and plant technicians. However, this programming technique had a weak software and data structure and made reuse of control applications nearly impossible [1]. In order to overcome these limitations the IEC 61131-3 standard introduced a new program development approach for PLCs, including five programming languages and the use of function blocks.

2.2.1 Function Blocks

Function blocks should be regarded as the basic building blocks of a control system, since they allow to pack a part of a control program so that it can reused in different parts of the same or a different program. The standard provides facilities so that well defined algorithms or control strategies written in any of the IEC languages can be packaged as reusable software elements [1]. The IEC 61131-3 standard defines a few rudimentary function blocks but there is not an available set of standard industrial FBs. The introduction of FBs into PLC programming introduces as well some concepts from OOP, since a FB encapsulates data and its associated methods. Since the third edition of the standard, OOP mechanisms are available in IEC 61131 and FBs allow for inheritance or polymorphism. The use of FBs allows the design of hierarchical and structured programs.

In IEC 61131, a function block has a set of input and output parameters that define data. Encapsulated in the FB there is an algorithm that runs every time that the block is executed, processing the input parameters and internal variables and producing a new set of output parameters. A FB can have a defined state and is able to store values. Figure 2.4 shows how a function block looks like in IEC 61131.

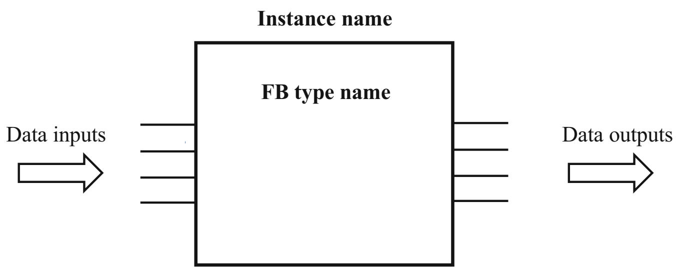


Figure 2.4 – A function block graphical representation as defined in IEC 61131.

2.2.2 Architecture and Software Model

At the highest level the standard defines the so-called *configuration*, which can be regarded as the required software for a specific control application. Within each configuration there can be one or more *resources*. A PLC is in this context a resource, and can contain more than one resource if it has multiple processor cards.

A *program* is executed in a resource and can be written in any of the languages described in Section 2.2.3. Typically a program contains multiple interconnected FBs. A program is able to read and write IOs and communciate with other programs. *Tasks* are used to configured and control a program or a FB. In IEC 61131, a program or FB has to be assigned to a task and this task has to be configured in order to execute periodically or when triggered by a changing state.

The standard defines the concept of *program organisation unit* or POU. Programs, functions and function blocks are POUs in IEC 61131. The main characteristic of a POU is that it can be used multiple times in an application. This characteristic encourages software reuse. A single FB type can have multiple instances, as well as a program type, or a function type. Another great advantage of this characteristic is that it allows to build FBs out of instances of other function block types. By building FBs out of networks of instances of other FB types, hierarchical structures can be achieved within IEC 61131 [1, p. 46].

In IEC 61131, there are both *local* and *global* variables. Local variables can only be accessed within the software elements in which they were declared. On the other hand, global variables can be accessed from any software element, allowing data transfer between different programs. There is another type of variables in IEC 61131 known as *directly represented variables*, which are used to access specific memory locations in a PLC. These variables shall not be used directly in FBs in order to facilitate software reuse, since they are hardware-specific parameters. Therefore, *directly represented variables* can only be declared and accessed within programs. For a whole view of the software model, see Figure 2.5.

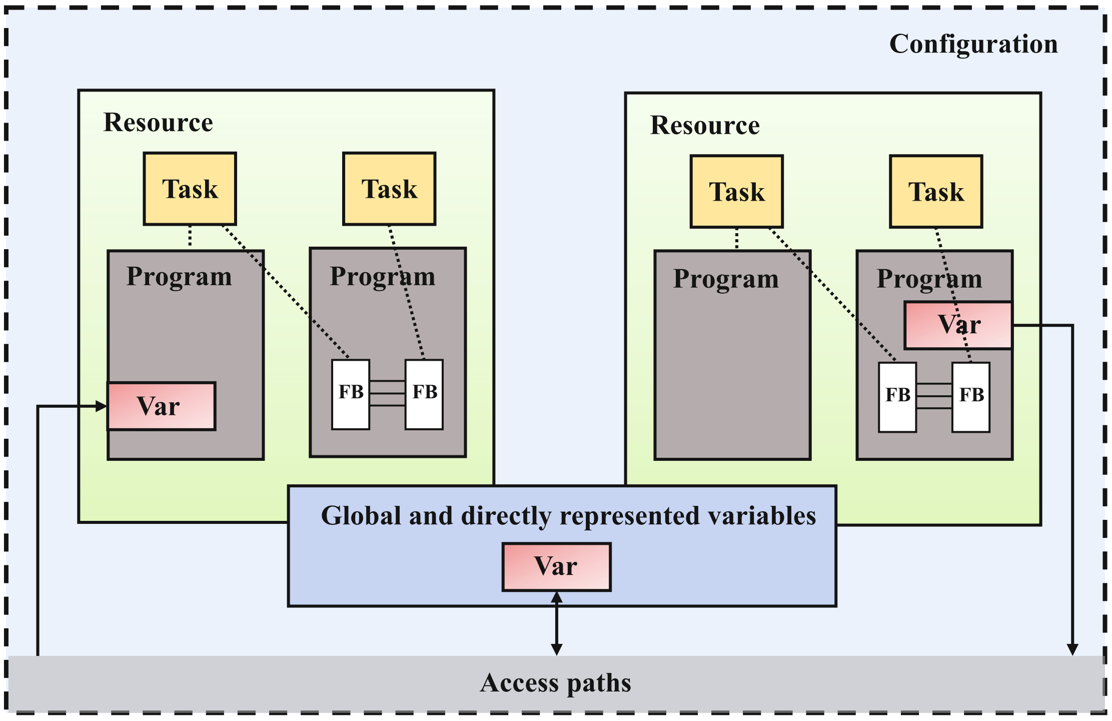


Figure 2.5 – The software model in IEC 61131-3.

IEC 61131 does not describe mechanisms for managing distributed configurations. As detailed in Section 2.3, IEC 61499 was designed with distributed systems in mind, overcoming the limitations of IEC 61131 in this area.

2.2.3 Programming Languages in IEC 61131-3

IEC 61131-3 defines five programming languages. Two of them are textual languages: Structured Text (ST) and Instruction List (IL). The others are graphical: Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC). Programs, functions and function blocks can be defined using any of the languages. No matter the language chosen, the variables and data types in POUs are describe using the the same common programming elements.

2.2.3.1 Structured Text

ST is a high level language with a syntax similar to PASCAL, developed specifically for industrial control applications. For an example of ST code, see Listing 2.1.

```
TYPE Alarm
    STRUCT
        TimeOn    : DATE_AND_TIME ;
        Duration  : TIME;
    END_STRUCT;
END_TYPE

VAR
    Rate,A1      : REAL;
    Count        : INT,
    Alarm        : Alarm;
    Alarm2       : Alarm;
    Profile      : ARRAY[1..0] OF REAL;
    RTC1         : RTC; (* Real time clock *)
END_VAR

Rate := 13.1;          (* Literal value i.e. constant *)
Count := Count + 1;    (* Simple expression *)
Al := LOG(Rate);       (* Value from a function *)
Alarml.TimeOn:= RTC1.CDT; (* Value from a function *)
                          (* block output parameter*)
Alarm2 := Alarml;      (* Multi-element variable *)

(* Value from a complex expression assigned to *)
(* a single element of an array *)
Profile[3] := 10.3 + SQRT(( Rate + 2.0 ) *
                    ( A1 / 2.3 ));
```

Listing 2.1 – An example of ST code in IEC 61131-3. Adapted from [1].

2.2.3.2 Function Block Diagram

A Function Block Diagram can express the behaviour of a POU (a program, a function or a function block) through a set of interconnected graphical blocks. These blocks can be standard FBs included in IEC 61131-3 (for example, an SR bistable as depicted in Figure 2.6), FBs from the library of a specific IDE, or functions (for example, an AND function) encapsulated in a block. An example of a program in the FBD language is depicted in Figure 2.7.

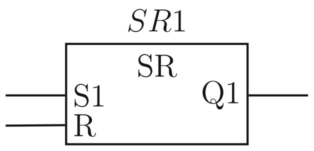


Figure 2.6 – A SR bistable FB as defined in IEC 61131-3.

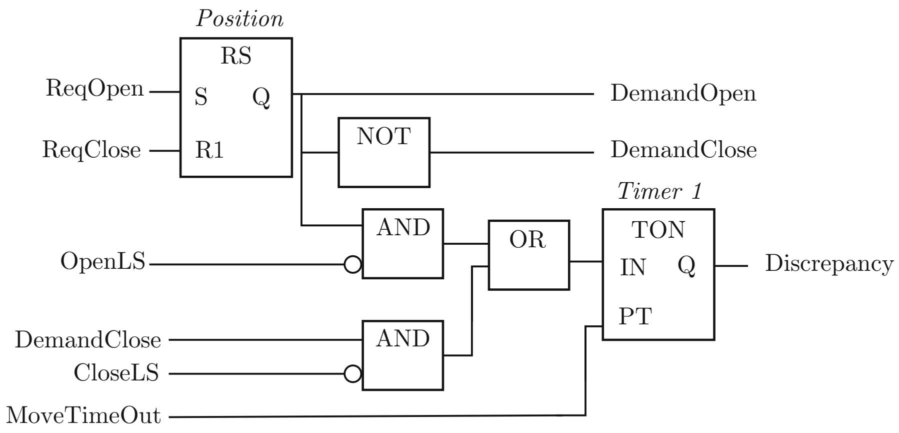


Figure 2.7 – A program in FBD containing FBs and functions in IEC 61131-3. Adapted from [1].

2.2.3.3 Ladder Diagram

This graphical language has been adapted from the traditional programming approach used in relay logic, which is based on the use of relays to describe a certain logic. Extending this traditional programming technique, in IEC 61131-3 both function blocks and functions can be added to the ladder diagrams. For an example of a program in the LD language, see Figure 2.8.

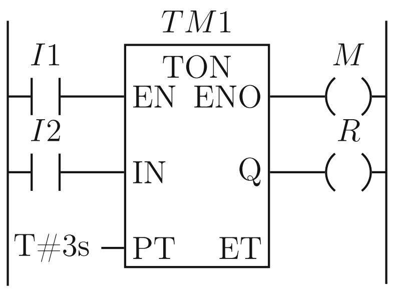


Figure 2.8 – An example of a Ladder Diagram in IEC 61131-3.

2.2.3.4 Instruction List

The second textual programming language included in the standard is a low level language with a similar structure to that found in a simple machine assembler. For an example of code written in IL, see Figure 2.2.

```
            LD      Speed       (* Load Speed and *)
            GT      1000        (* Test if > 1000 *)
            JMPCN   VOLTS_OK    (* Jump if not *)
            LD      Volts       (* Load Volts and *)
            SUB     10          (* Reduce by 10 *)
            ST      Volts       (* Store Volts *)
VOLTS_OK:   LD      1           (* Load 1 and store *)
            ST      \%Q75       (* in output 75 *)
```

Listing 2.2 – An example of IL code in IEC 61131-3. Adapted from [1].

2.2.3.5 Sequential Function Chart

The last language defined in IEC 61131-3 is a graphical language for describing the sequential behaviour of a control program. The behaviour of a system is based on states and transitions. It is based on Petri-net, with actions associated to each state. For an example of a program in the SFC language, see Figure 2.9.

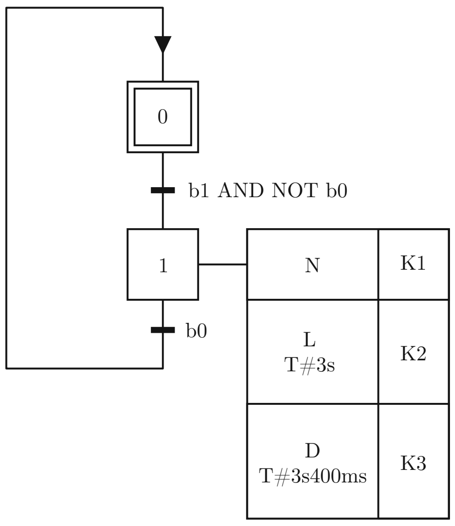


Figure 2.9 – An example of a Sequential Function Chart in IEC 61131-3.

2.3 An Overview of IEC 61499

IEC 61499 defines a general model and methodology for describing function blocks in a format that is independent of implementation. The methodology can be used by system designers to construct distributed control systems. It allows a system to be defined in terms of logically connected function blocks that run on different processing resources [6]. The standard was initially published in 2005, with the second edition being published in 2012.

The IEC 61499 standard builds on the function block concept defined in the PLC language standard IEC 61131-3. One of the main differences between the IEC 61131-3 and IEC 61499 is that the former one is based on scan-based execution while IEC 61499 is based on event-based execution, although this topic has been discussed [10]. IEC 61499 was also developed with distributed control systems in mind, in contrast to IEC 61131 which focuses on centralized computing platforms.

2.3.1 Function Blocks

In IEC 61499, a function block (FB) is a functional unit of software that encapsulates a certain behaviour. There are three types of function blocks: basic function blocks (BFBs), composite function blocks (CFBs) and service-interface function blocks (SIFBs). The first two will be described in this section and the third one will be described in the next section.

The IEC 61499 standard introduces a new approach to FBs, defining two types of inputs and outputs for FBs: event inputs/outputs and data inputs/outputs. Events and data can be linked together by using the `WITH` qualifier (represented as a square connector). By linking an event with some data input or output, the linked data will be sampled each time that the corresponding event arrives. A basic function block is depicted in Figure 2.10.

As mentioned before, FBs encapsulate a certain functionality. An Execution Control Chart (ECC) represents the behaviour of a BFB. Furthermore, it maps the events of the FB on to algorithms and describes the relationships between the possible states of the FB. In order to go from one state to another one in an ECC, a certain transition condition has to be evaluated to true. Also, each state has an associated action that can contain an algorithm, an output event or both of them. The algorithm associated to a certain action can be defined using a suitable language such as Java, C, or Structured Text (as defined in IEC 61131-3, see Section 2.2.3.1). For an example of an ECC as defined in IEC 61499, see Figure 2.11. The standard also defines a way to describe how to use the interface of a FB through the use of the service sequence diagrams. For a FB definition zero or more service sequence diagrams can be used to describe the timing and sequential relationships between various interactions with the function block [6]. An example

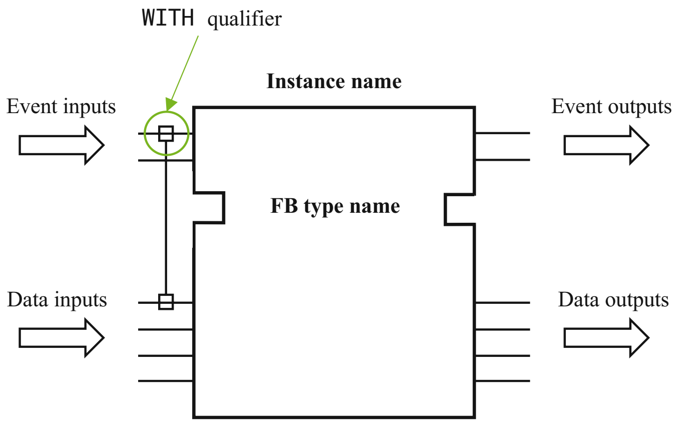


Figure 2.10 – A function block graphical representation as defined in IEC 61499.

of a service sequence diagram is depicted in Figure 2.12.

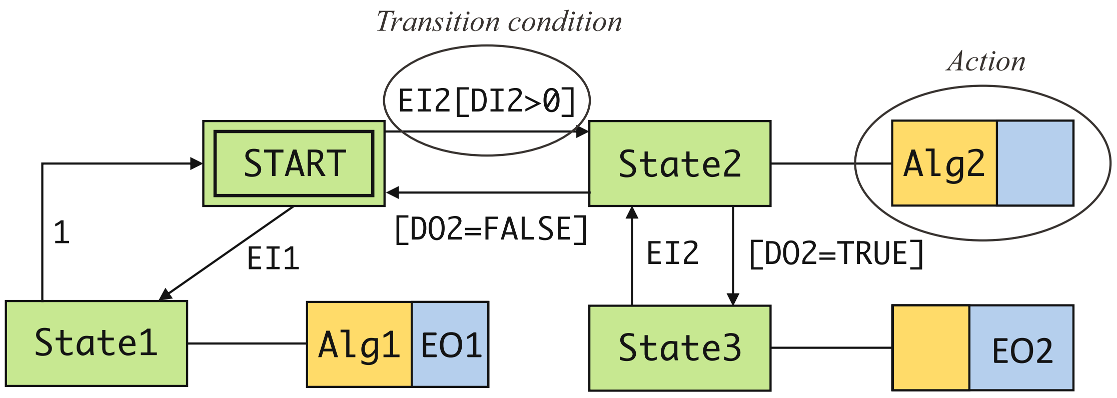


Figure 2.11 – Example of an ECC in IEC 61499. Adapted from [6].

In IEC 61499, a CFB is a different type of FB than the BFB. These CFBs are constructed from a network of FBs (these can be either BFBs, lower level CFBs, or SIFBs).

2.3.2 Service Interface Function Blocks

A service-interface function block (SIFB) is a special type of function block that provides an interface between FBs located in one resource and services provided by or outside the resource. The standard presents two SIFBs as examples and reference, the *REQUESTER* and *RESPONDER* SFIBs, the later one being the only type of FB in IEC 61499 that is active instead of passive (it does not need an input event coming from another FB to actuate, since an underlying service can

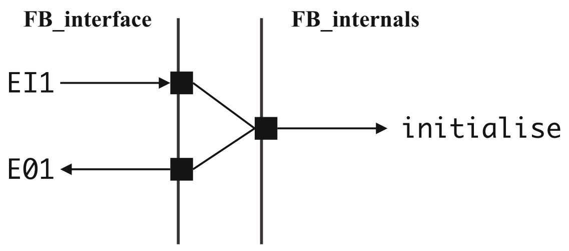


Figure 2.12 – Example of a service sequence diagram in IEC 61499. Adapted from [6].

trigger its execution). The standard stipulates that these forms of function blocks should be defined using a standard set of input and output variables and input and output events [6]. The behaviour of SIFBs is defined by a service sequence diagram.

Two common applications of SIFBs are network communication and IO reading/writing. For example, in a distributed application *CLIENT* and *SERVER* SIFBs can be used to send and request data between different resources. To interact with the hardware, *IO_ WRITER* and *IO_ READER* SIFBs can be used to write to physical outputs and read from physical inputs, respectively.

2.3.3 SubApplications

SubApplications (SubApps for short) encapsulate a network of BFBs, CFBs and/or SIFBs. They are, therefore, similar to Composite Function Blocks. However, unlike CFBs, they can be distributed between multiple resources. Each BFB and CFB can only be allocated to one resource, while a single SubApp can run over multiple resources. Another difference is that the `WITH` qualifier is not used in type definitions. To sum up, SubApps provide a way to structure IEC 61499 applications and can be distributed over many resources. The difference between CFBs and SubApps is illustrated in Figure 2.13.

2.3.4 Adapters

As mentioned in Section 2.3.1, in IEC 61499 the event and data interface is separated in FBs. This can lead to cluttered design spaces with lots of connections between FBs. Moreover, the interaction between FBs is confusing and the user may omit to connect all the required interface elements or wrongly connect interface elements [6]. To overcome these problems the adapter concept was introduced in IEC 61499.

Adapters allow to group multiple data an events, in order to pass them between

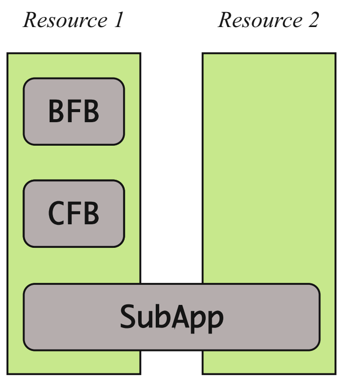


Figure 2.13 – A SubApp, unlike a FB, can be distributed over multiple resources in IEC 61499.

FBs within a single connection. An adapter contains no algorithms or state, it is just an extension of the interface of a FB. An adapter in IEC 61499 can be compared to an electrical cable and takes its terminology from this element (plugs and sockets, as following explained). An adapter provides a *Plug* for the FB that provides the data and events (represented as ">>" on the output side of a FB) and a *Socket* for the FB that requests the data and event (represented as ">>" on the input side of a FB). Therefore, a Plug is a providing interface and a Socket is a requesting interface. The interface of adapters is defined in the form of a FB interface. For an example of an application where the adapter concept is used, see Figure 2.14.

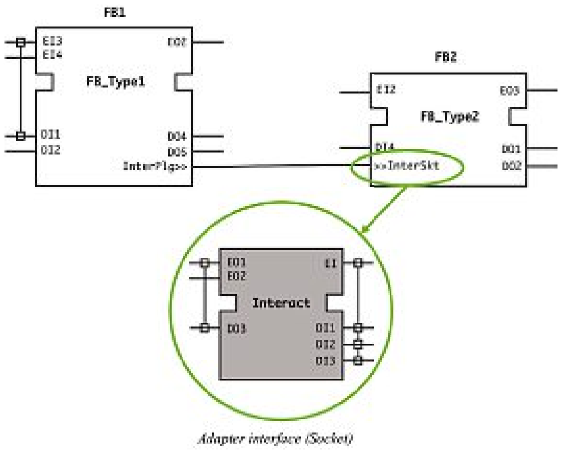


Figure 2.14 – Example of an adapter in IEC 61499. Adapted from [6].

According to the standard, the adapter interface (defined as a FB) is declared and stored as the requesting side of the interface (the Socket). However, the standard also states that IEC 61499 compliant IDEs could declare the adapter interface either as a Socket or as a Plug. The dynamic behaviour of adapters is described through service sequence diagrams. In contrast to the service sequence diagrams used for BFBs, which define the interaction between the FB interface and the FB internals, the service sequence diagrams used for adapters define the interactions between the Plug and the Socket.

2.3.5 Architecture and Software Model

The standard was developed with a focus on distributed control applications. With this idea in mind, the architecture of IEC 61499 can be divided into different views or models: the application model, the system model, and the distribution model.

The *application model* contains the function block network, linked by event and data. All the behaviour is defined by the function block network. The design of the application model is the first stage in the control application design and does not consider any particular hardware. This is why IEC 61499 is application-centered, while IEC 61131 on the other hand is very resource focused.

The *system model* considers the available devices and the physical communication network between them. In contrast to IEC 61131, IEC 61499 introduces the device concept. A device model in IEC 61499, with a single or multiple processing units, can contain one or more resources. Therefore, in IEC 61499 a resource is just a logical separation within a device that provides an independent execution environment. In IEC 61131, a single processing unit can only contain one resource. A device as defined in IEC 61499 is depicted in Figure 2.15.

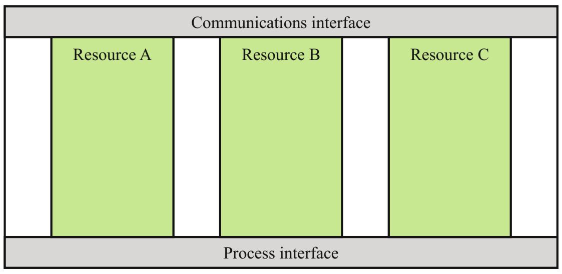


Figure 2.15 – A device model in IEC 61499.

The *distribution model* links the application model to the system model. An application can be distributed between multiple devices. Each application contains function blocks (BFBs, CFBs and SIFBs) and subapplications. However, only subapplications can be distributed between multiple resources. A global view of the three models is depicted in Figure 2.16.

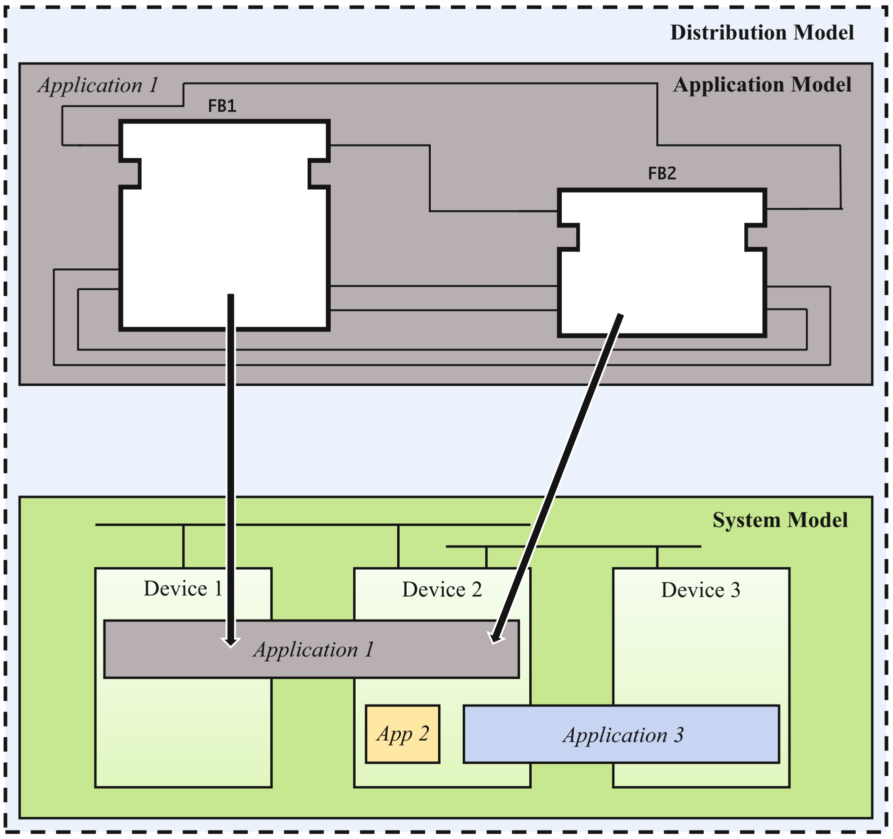
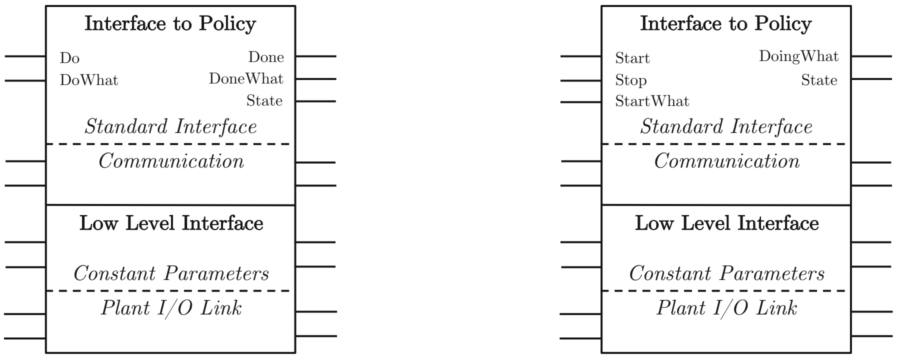


Figure 2.16 – The distributed architecture in IEC 61499.

2.4 Design Approaches Based on IEC 61131

Authors in [11] propose a hierarchical modelling procedure that can be applied in IEC 61131, with the goal of achieving a reusable and modular control logic in industrial automation. By modeling an example control logic in the SFC language from IEC 61131-3, the authors identify two types of modifications that can be needed in the plant during its life cycle: modifications in action mechanisms (related to sensor or actuators) and policy changes (related to a change in functional requirements). In traditional control logic design approaches there is a mix between mechanisms and policies which minimizes the modularity and reusability of the application.

To achieve a reusable and modular control logic, the authors propose a procedure that separates mechanisms and policies. The concept first splits the control logic into two groups: a set of basic actions (the mechanisms of functionality implementation) and sequences to coordinate actions execution (the control policy). These two have to be completely independent in order to have a reusable control



(a) Do-Done GA interface. (b) Start-Stop GA interface.

Figure 2.17 – Interfaces for the two types of Generalized Actuators FBs. Adapted from [11].

logic. An entity called *Generalized Actuator* (GA) is introduced for the presented approach. The steps to define the GAs in a control application are the following:

1. Identify the actions that cannot be further decomposed.
2. Associate each of the previously identified actions to a sensor or an actuator.
3. Define the GAs. Each GA groups a set of actions and sensors/actuators. It is represented as a FB in IEC 61131-3, which runs continuously even if no specific action is required from it.

The Generalized Actuators (GAs) can be divided into two groups: the *Do-Done* GAs and the *Start-Stop* GAs. The first group is associated to actions that terminate after a finite amount of time. The second group is associated to actions that could continue for an infinite time and whose termination has to be decided "externally". The interface of these two types of GAs is depicted in Figure 2.17. Authors suggest defining the behaviour of the GA function blocks in the SFC language. By encapsulation into GAs reusable components are achieved. A change in an action mechanism just implies changing the implementation in a certain GA, without affecting other mechanisms or the overall policy. This approach was applied to another case study in [12], where authors also conclude that GAs serve to create highly reusable mechatronic components.

The hierarchy in the approach presented in [11] has (at least) two levels. In the lower layer the GAs are located. The GAs are managed by a *policy manager* in the top level. This overall control policy represented by a policy manager in 2.18 can be defined in the SFC language as well.

In [13], the GA concept is extended by introducing a new concept: the Generalized Device (GD). The author points out a limitation of the GAs: the components,

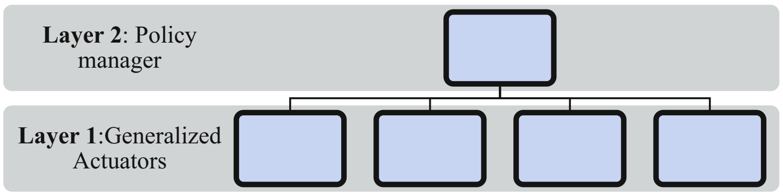

Layer 2: Policy manager
Layer 1:Generalized Actuators

Figure 2.18 – The hierarchical structure of the approach presented in [11]. The layer 1 includes the components that encapsulate hardware control operations.

encapsulated as FBs, can be reused as long as the hardware component used in the plant is the same. The idea of the GDs is to add a new layer in the hierarchy which is independent from hardware, since different field devices often require the same control logic. A classification of pneumatic cylinders, according to the type of sensors and actuators that can be used, is presented. Each of these pneumatic cylinder hardware components corresponds to a single Generalized Device, encapsulated as a FB (see Figure 2.20). What was previously encapsulated as a GA in [11] is now divided into a GA and a GD, in order to split the logic from the hardware implementation. The new hierarchy is shown in Figure 2.19.

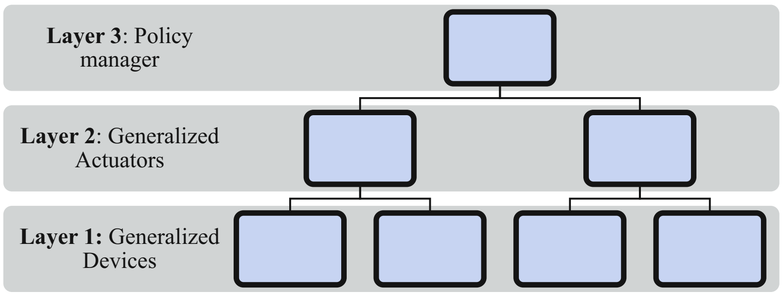

Layer 3: Policy manager
Layer 2: Generalized Actuators
Layer 1: Generalized Devices

Figure 2.19 – The hierarchical structure of the approach presented in [13]. This approach takes the hierarchy presented in [11] and adds a new layer, the layer 2, to further decouple hardware and control logic.

2.5 Design Approaches Based on IEC 61499

All the design approaches reviewed for IEC 61499 can be divided into two groups. The first group, presented in Section 2.5.1, contains approaches that focus on hardware abstraction, without providing details of how the application components would be hierarchically structured (or only showing part of a hierarchy). The second group, presented in Section 2.5.2, contains design approaches that focus on and explicitly show a component hierarchy.

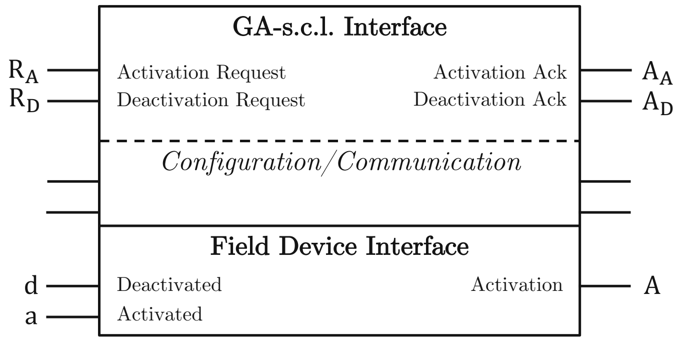


Figure 2.20 – Interface of a GD for a single-actuating pneumatic cylinder with two sensors. Adapted from [13].

2.5.1 Hardware Abstraction Approaches

Melik-Merkumians et al. [14] present an approach based on an adapted version of MDA (Model-Driven Architecture). The authors suggest to split the application in a Platform Independent Model (PIM), which contains the logic control part of the application, and a Plant Model (PDM) which describes the exact configuration of the actual plant to be programmed. This two parts can be mapped together generating the hardware-specific control application, which in a MDA approach is the Platform Specific Model (PSM). This concept is illustrated in Figure 2.21. Authors suggest that IEC 61499 is suited as a logical control application (PIM) metamodel, and also state that in the case of IEC 61499 it is possible to directly link the logical application blocks to the specific hardware through the use of hardware-specific adapters for the task function blocks. They provide an example of use for this concept. However, the example does not contain any implementation details for IEC 61499 or any other language, nor shows how to avoid hardware-specific parameters in the PIM.

Wenger et al. [3] propose also a MDA approach to design 61499 applications, with the goal of improving reusability by separation of logical functionality and hardware access methods. In their work the authors compare the different models specified by the MDA to the models in IEC 61499. By doing so they show the capabilities of the standard to decouple the logical part of the application from the hardware implementation. However, the SIFB defined in the standard renders hardware access methods at very high level in the application architecture. To overcome this issue the authors propose replacing the SIFB by two new FBs: Logical Service Function Blocks (LSFB) and Hardware-Specific Implementation Function Blocks (HSIFB). The LSFB specify a minimal logic interface where hardware-dependent parameters are fully avoided. Then, at the deployment phase, the appropriate HSIFB is selected and connected to the LSFB via the adapter con-

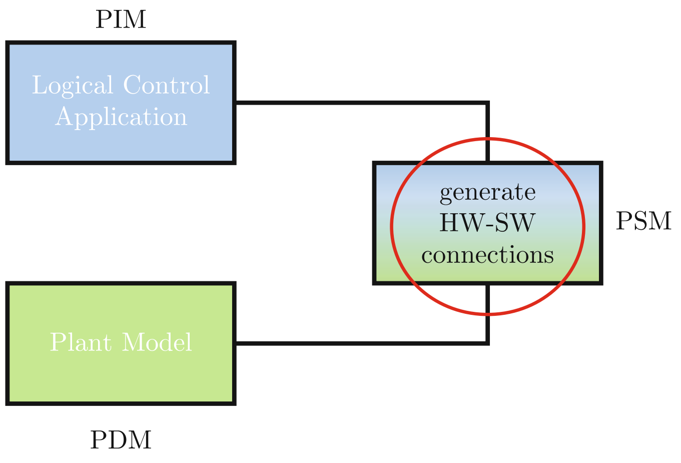


Figure 2.21 – A MDA approach for control application design in industrial automation. The control application is divided into two parts: the PIM and the PDM. These two parts are then coupled together. Adapted from [14].

cept defined in the standard. The idea of the authors is that LSFB are identified and standardized and that the vendors provide the HSIFBs. An example of this approach is reflected in Figure 2.22, where a LSFB of a handling unit which pics and places components represents a minimal logic. This LSFB could then be connected via an adapter to different HSIFBs provided by different vendors.

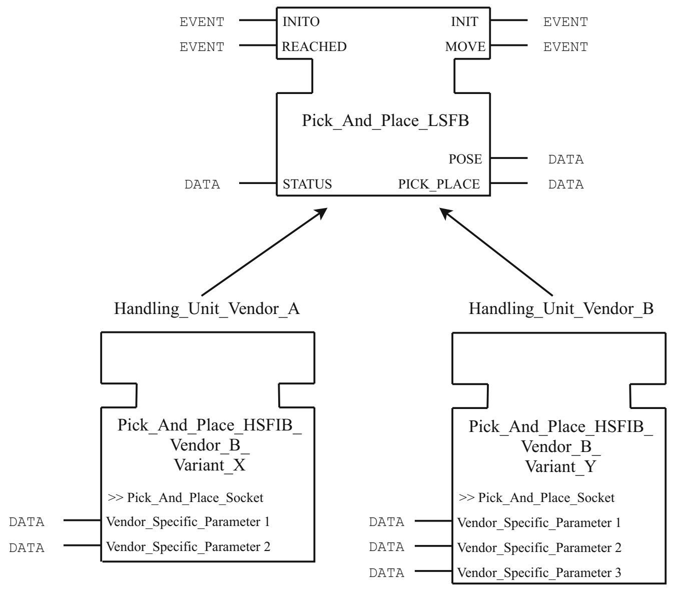

EVENT
EVENT
INITO
REACHED
INIT
MOVE
EVENT
EVENT
Pick_And_Place_LSFB
POSE
DATA
DATA
STATUS
PICK_PLACE
DATA
Handling_Unit_Vendor_A
Handling_Unit_Vendor_B
Pick_And_Place_HSFIB_ Vendor_B_ Variant_X
>> Pick_And_Place_Socket
DATA
Vendor_Specific_Parameter 1
DATA
Vendor_Specific_Parameter 2
Pick_And_Place_HSFIB_ Vendor_B_ Variant_Y
>> Pick_And_Place_Socket
DATA
Vendor_Specific_Parameter 1
DATA
Vendor_Specific_Parameter 2
DATA
Vendor_Specific_Parameter 3

Figure 2.22 – Example of a LSFB and two possible HSIFBs provided by the vendors. Adapted from [3].

Hegny et al. [15] base their approach on the concept of Instrumentation and Control-Points (ICPs) introduced in [16]. The aim is to provide easily identifiable interfaces, since hidden interfaces impose a higher effort in grasping the functionality. The ICPs provide a unified interface to well-defined parts of the plant, and are introduced in IEC 61499 applications via the adapter concept. Therefore, the access to all relevant sensors and actuators (basic services of the components) is encapsulated within the appropriate ICP. During the development of the hardware independent control application, only the well defined interface is available, the plug representation of the adapter (representing the ICP concept). The interaction of the control application with the controlled process can be defined through service sequence diagrams within the adapter. For example, an ICP adapter for a gripper only defines two possible states (gripped or released) and dictates via output events to the hardware-specific parts if the gripper shall grip or release (see Figure 2.23). An adapter representing an ICP has no hardware parameters, it only represents a minimal interface and can therefore be used for any type of gripper (it doesn't matter if its a vacuum or a 2-finger gripper, for example). This way hardware-specific parameters are pushed to the very lowest layer of the control application, enhancing reusability. The ICP adapter concept in [15] represents a similar approach as the use of the adapter between LSFB and HSIFB described in [3].

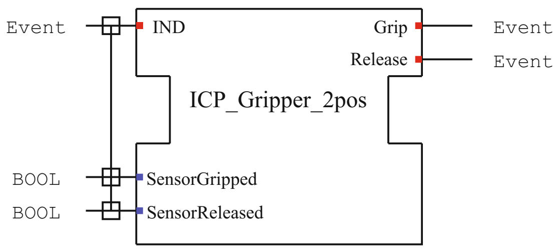


Figure 2.23 – Example of an ICP adapter for a gripper. The adapter describes a minimal logic interface without any hardware parameters. Adapted from [15].

Authors in [15] also state the importance of a well-structured hierarchical control application in order to improve reusability. Separation into manageable components increases comprehensiveness in control applications. This approach for hierarchical control application structuring is based on the use of the SubApps and adapter concepts from the IEC 61499 standard, and it is the same as described in Section 2.5.2. The concept presented in [15] uses the adapter concept to both structure the application hierarchically and decouple the hardware implementation from the control logic through the use of ICPs. The basic structure of this approach is depicted in Figure 2.24.

Authors in [17] propose a methodology to increase reusability by hardware

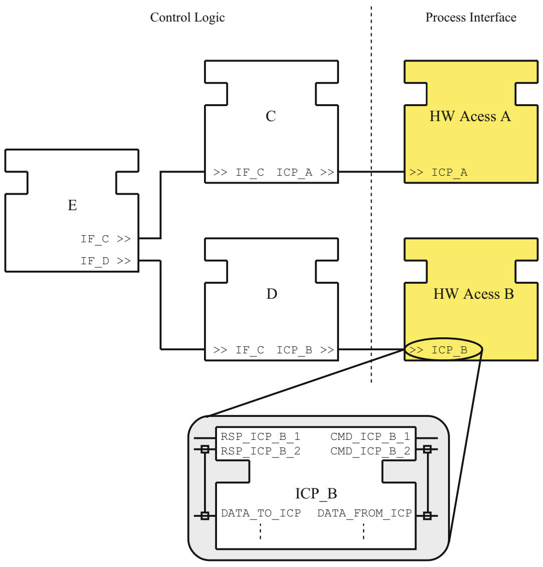


Figure 2.24 – A hierarchical structure implementing the ICP concept. Adapted from [15].

separation that can be implemented for the IEC 61499 standard in Eclipse 4diac. This methodology is based on a metamodel for describing a model generic device configuration. The proposed concept can be applied to both a standalone device or a modular device. The root element of this metamodel is the *Device*, which contents *Module* elements. These *Module* elements can be of type *master* or *slave*, and contain *IO* elements which represent the interface for interacting with the actuators and sensors hardware. *Module* elements also contain *ConfigParameter* elements. Another important element is the *BusInterface*, which represents the interface for the communication between *module* elements and other device with the same type of *BusInterface*. From the *BusInterface* two elements are inherited: *Socket* and *Plug*.

The idea is that this generic device model is then translated into SIFBs in a IEC 61499 application. Each hardware device is represented by a SIFB. In case of a distributed device configuration, the bus is implemented through the *adapter* concept. A control application developed according to this methodology will have at least two resources: one containing the hardware independent control logic and another one containing the device configuration FBs. The hardware-independent control logic is achieved by removing the *PARAMS* data input from the generic IO FBs. Now, all the hardware configuration parameters are in the resource which contains the device configuration model. In the device configuration model, each pin takes as a data input the instance name of the IO FBs used in the harware-independent control logic. That is, the instance names are used as identifiers.

To implement this functionality additional features have to be added to 4diac and FORTE. In Figure 2.25 an example of this methodology in 4diac is depicted, where a simple program turns a LED on and off. As can be seen, there are no hardware specific parameters on the control logic side.

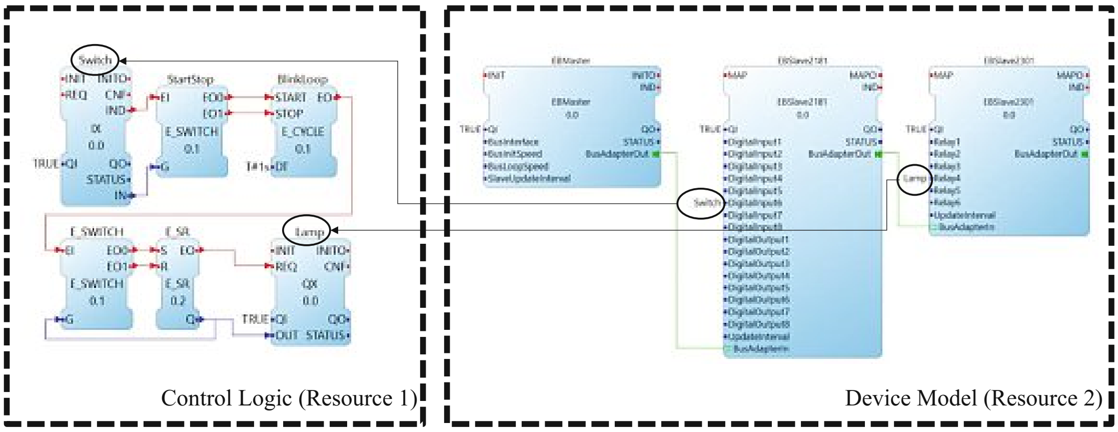


Figure 2.25 – On the left, a simple logic that generates a cyclic boolean signal. On the right, a device model with one master and two slaves managing the IOs. Adapted from [17].

To sum up, authors in [17] developed a methodology that can be applied in IEC 61499 to develop hardware-independent control applications. This methodology splits the application development into a device configuration model, located in one resource, and a control logic model, located in another resource. The device configuration model is a generic model that can be translated into FBs and used in an IDE like 4diac. The control logic located in a different resource has now no hardware specific parameters. However, this approach still introduces hardware interfaces (function blocks *IX* and *QX* in Figure 2.25) very early into the development process, does not specify how to get rid of other type of parameters that could couple the logic to the hardware and overall it just implements an approach that is very similar to how IO mapping is done in IEC 61131.

2.5.2 Hierarchical Control Architecture Approaches

Zoitl and Prähofer describe in [18] an approach to build hierarchical applications in IEC 61499 with FBs. The authors suggest to use the *adapter* concept defined in the standard in order to define components' interfaces. This has the advantage of both making the design space less cluttered as well as clearly separating higher and lower level components. The plug (the providing interface) is implemented in the higher-level side and the socket (the required interface) is implemented on the lower-level side of a hierarchical component connection. This implies that the lowest components in the hierarchy have no plugs. Therefore, the use of the adapter increases the decoupling of application parts. The behaviour of the interface can be documented with *service sequences*. If only the *adapter* is used, the component hierarchy is still flat. The authors suggest structuring the components with *sub-applications*, also defined in the standard. By using SubApps all the components of each hierarchy level can be encapsulated.

As mentioned in [18], in a hierarchical control architecture the components at different hierarchy levels usually serve different functions. At the lowest level in the hierarchy (layer 0) we usually find the components which implement hardware interfaces. In the upper level (layer 1) the basic control operation components are located. At higher levels the components are in charge of the coordination of the sub-components in the lower levels. This structure implies that there is a directed control flow, where events are flowing down to initiate control operations in the lower-level components and feedback on the progress of the control operations is flowing back up. This architecture is depicted in Figure 2.26.

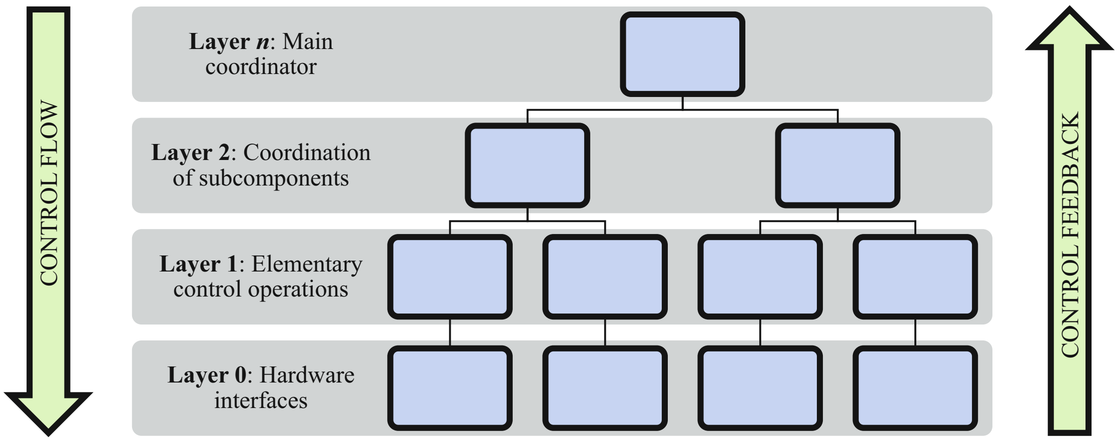

CONTROL FLOW
Layer *n*: Main coordinator
Layer 2: Coordination of subcomponents
Layer 1: Elementary control operations
Layer 0: Hardware interfaces
CONTROL FEEDBACK

Figure 2.26 – A hierarchical control architecture as defined in [18]. Layers 1 and 0 shall encapsulate hardware-related operations, while higher layers implement the control logic.

Authors in [19, Chapter 19] apply the concepts in [18] to a pick and place station. This approach implies that the control software hierarchy follows the mechatronical hierarchy of the plant. This means that each mechatronical component has a counterpart in the software structure. Therefore, the first step in the control application design process is to analyze the mechanical structure and identify the different components at different levels in the hierarchy. In the lowest level of the hierarchy (layer 0) we find the atomic components. These atomic components interact directly with the hardware, and are implement through SIFBs.

Within the 4diac IDE, these can be implemented with *IX* and *QX* SIFBs. Components at these level are hardware-dependent since a *PARAMS* input has to be specified with the desired input and output numbers. The upper layer (layer 1) contains the elementary components. These components connect atomic components and provide an independent function. In these layer the components are reusable for other applications of the same domain, since these components implement a very basic functional logic. An example of a component at this layer could be a gripper component that implements two outputs for the SIFBs in the lower layer (*Grip* and *Release*) and provides four states that are fed back to the upper layer components with information about the current state of the gripper (*Gripped*, *Released*, *GripFailed* and *ReleaseFailed*). An adapter provides the interface for the gripper component services to the higher level components. This approach renders components in the layer 1 reusable. Following the gripper example, it does not matter the gripper type (vacuum, two-finger, ...), all of the them have the same basic logic and functionality and therefore by encapsulating their basic behaviour and providing an interface via adapters reusable automation components can be developed.

The authors in [19, Chapter 19], following the approach in [18], make use of SubApps to group components of different hierarchy layers. For example, each hardware accessing SIFB in layer 0 is encapsulated with its corresponding component in layer 1 into a SubApp. This SubApp is connected to the components in layer 2, referred to as coordinating components. These components in layer 2 coordinate the lower layer components. Common interface elements of both layers are combined within adapters. Each component in layer 2 is encapsulated into a SubApp with its underlying components. In the pick and place example given by the authors, the highest level in the hierarchy is the layer 3, which contains a single component known as the process control component. Its role is to coordinate all the underlying SubApps. This hierachical structure is reflected in Figure 2.27. As shown, components in the lowest level of the hierarchy (layer 0) are implemented as SIFBs. The components at the other levels of the hierarchy are implemented with BFBs. Each component in layer 1 is encapsulated into SubApp with its corresponding component in layer 0. Furthermore, each component in layer 2 is encapsulated into a SubApp with its underlying SubApps in layer 1.

To sum up, in [19, Chapter 19] a hierarchical control application is designed and implemented in an IEC 61499 compliant IDE (Eclipse 4diac). The hierarchical structure is achieved by layering mechatronic components. The SubApp and adapter concepts in IEC 61499 are used to structure the application and provide logic interfaces to other layers, respectively. The authors claim that the components used up to the top layer are reusable, including the hardware access components in layer 0, since the *IX* and *QX* SIFBs are independent of the applied

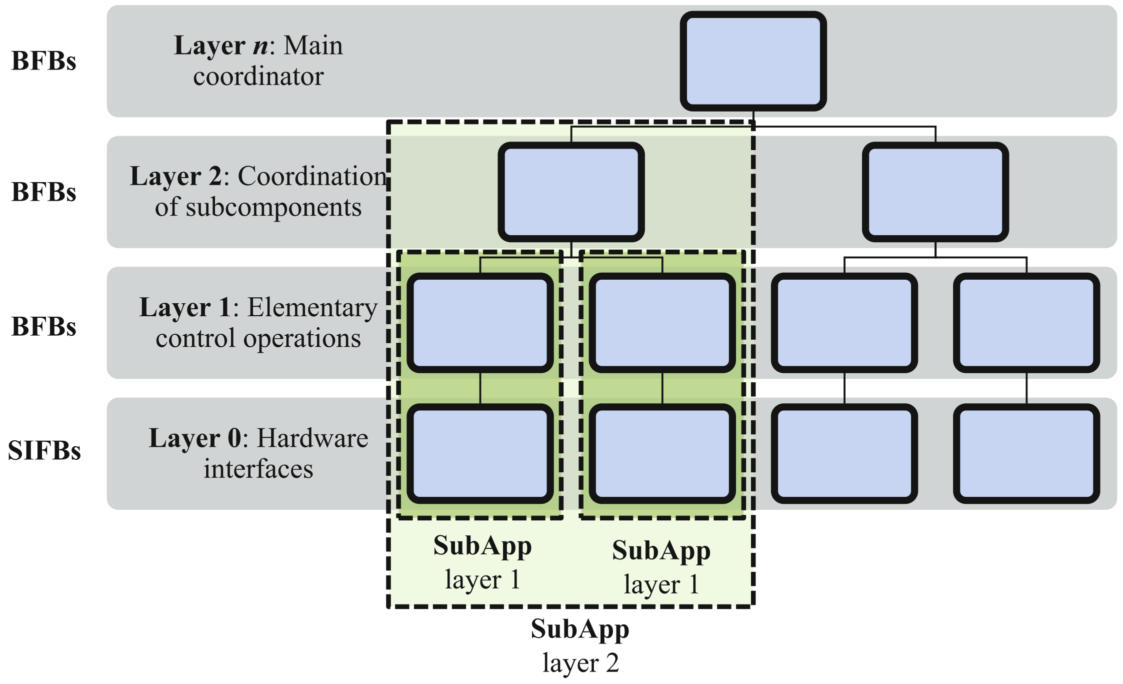

BFBs
Layer *n*: Main coordinator
BFBs
Layer 2: Coordination of subcomponents
BFBs
Layer 1: Elementary control operations
SIFBs
Layer 0: Hardware interfaces
SubApp layer 1
SubApp layer 1
SubApp layer 2

Figure 2.27 – A hierarchical control architecture as defined in [19, Chapter 19]. Authors propose the same hierarchy as in [18], and suggest to group components by using SubApps in order to enhance modularity.

hardware and make the application hardware independent. However, authors do not proof if the top layer components are reusable against hardware changes.

2.6 Research Questions

The design approaches reviewed in Section 2.4 and Section 2.5 claim to achieve reusable control applications for industrial automation, by decoupling the control logic from the hardware. Most design approaches suggest to structure the control application in a hierarchical way, where the components in the bottom of the hierarchy encapsulate hardware-control operations and the components in the higher layers implement basic control logic operations. However, none of the reviewed works explain how these components in higher layers can be completely hardware-independent. The initial hypothesis is that with the current design approaches, components in the higher layers of the hierarchy can still be coupled to hardware and therefore they cannot be reused when a hardware component is replaced, even if the control logic remains the same. From this initial hypothesis the first research question is derived:

Research Question 1

What are the limitations of the current hierarchical and component-based control design approaches in industrial automation?

An easy experiment to answer this research question and verify the initial hypothesis consists in implementing a state of the art design approach on a prototype with a certain hardware configuration. Then, a specific hardware component can be replaced by another hardware component which implements the same control logic, but has a different working principle. This experiment can verify the initial hypothesis if components in the higher layers of the hierarchy can not be reused after the hardware change. In case these components can not be reused, the experiment can also identify what is limiting the component reuse in the state of the art design approaches. With a better understanding of these limitations, the goal of this thesis is to come up with a design approach that improves component reuse, which leads to the second research question:

Research Question 2

How can component reuse be improved in control applications design for industrial automation?

CHAPTER 3

Concept

Between all the state of the art approaches there are multiple elements in common. Some authors propose an MDA approach, with two different application parts that are independent of each other and are finally mapped or coupled to each other in order to obtain the final control application. However, implementation details are not sufficient to test these approaches, since it is not specified how the hardware-independent part is designed or how the mapping between the two independent application parts is done. The most recent approaches include implementation specific details for IEC 61131 and IEC 61499 and focus on hierarchically-structured applications. In these hierarchical and component-based approaches, the hardware-specific components are located at the bottom of the hierarchy while on the higher levels components responsible of the logic control flow of the application are found. Authors claim that in these approaches components are reusable at every level in the hierarchy.

In this chapter a case study is going to be carried out, in which the state of the art approaches are going to be tested. The experiment is going to be implemented in IEC 61499, since there are implementation-specific details for this standard (see [19, Chapter 19], for example). However, it is sufficient to extract a generic conclusion on the limitations of all the approaches reviewed, since regardless of the language used (IEC 61131 or IEC 61499) all the approaches share a very similar concept. Once that the limitations are identified in the case study, a new design approach is going to be proposed.

Table 3.1 – Types of SFIBs used in 4diac for the case study.

Service Interface Function Blocks	
RBC	Read Bool Cycle
WB	Write Bool
RR	Read Real
WR	Write Real
WLR	Write LReal

3.1 Case Study: Initial Setup

The case study is carried out in a resistor sorter plant (see Figure 3.1). The chosen IDE for the case study is Eclipse 4diac, an open source infrastructure for distributed industrial process measurement and control systems based on the IEC 61499 standard [20]. It is available for multiple platforms: Windows, Linux and Mac OS. The runtime environment for 4diac is called FORTE.

The resistor sorter prototype has a vibrating conveyor, which is filled with resistors. When the conveyor is on, the resistors move up in the conveyor until a resistor reaches the pick-up position in the conveyor (signaled by a position sensor). When a resistor is in the pick-up position, a 3-axis manipulator with a gripper in the bottom of its vertical axis picks up the resistor and transports it to a measuring station. In this initial setup, the three axis of the manipulator are driven by electric motors. The resistor is positioned by the manipulator in a two-position turning table. This turning table rotates in order to position the resistor under some measuring clamps. Then the measuring clamps go down to measure the resistor. This measured value is compared against a reference value. Once the measurement is done, the measuring clamps retract and the turning table moves back to its initial position. Then the manipulator collects the resistor and positions it in one of the two available storage trays, depending if the resistor is between tolerances or not.

According to the guidelines in [18], a hierarchical composition of the plant can be created based on mechatronic components, as depicted in Figure 3.2 (the SIFB components on layer 0 have been omitted in this diagram).

In 4diac, the control application has been structured as depicted in Figure 3.3. The green blocks represent SIFBs in 4diac (layer 0 components). The blue blocks represent layer 1 FBs. Layer 0 and 1 components are grouped into SubApps (dark grey blocks). The orange blocks represent layer 2 FBs. Layer 1 SubApps and their corresponding layer 2 components are grouped into SubApps (light grey blocks). Finally, the top component FB in layer 3 is represented as a yellow block. The different SIFBs used are listed in Table 3.1.

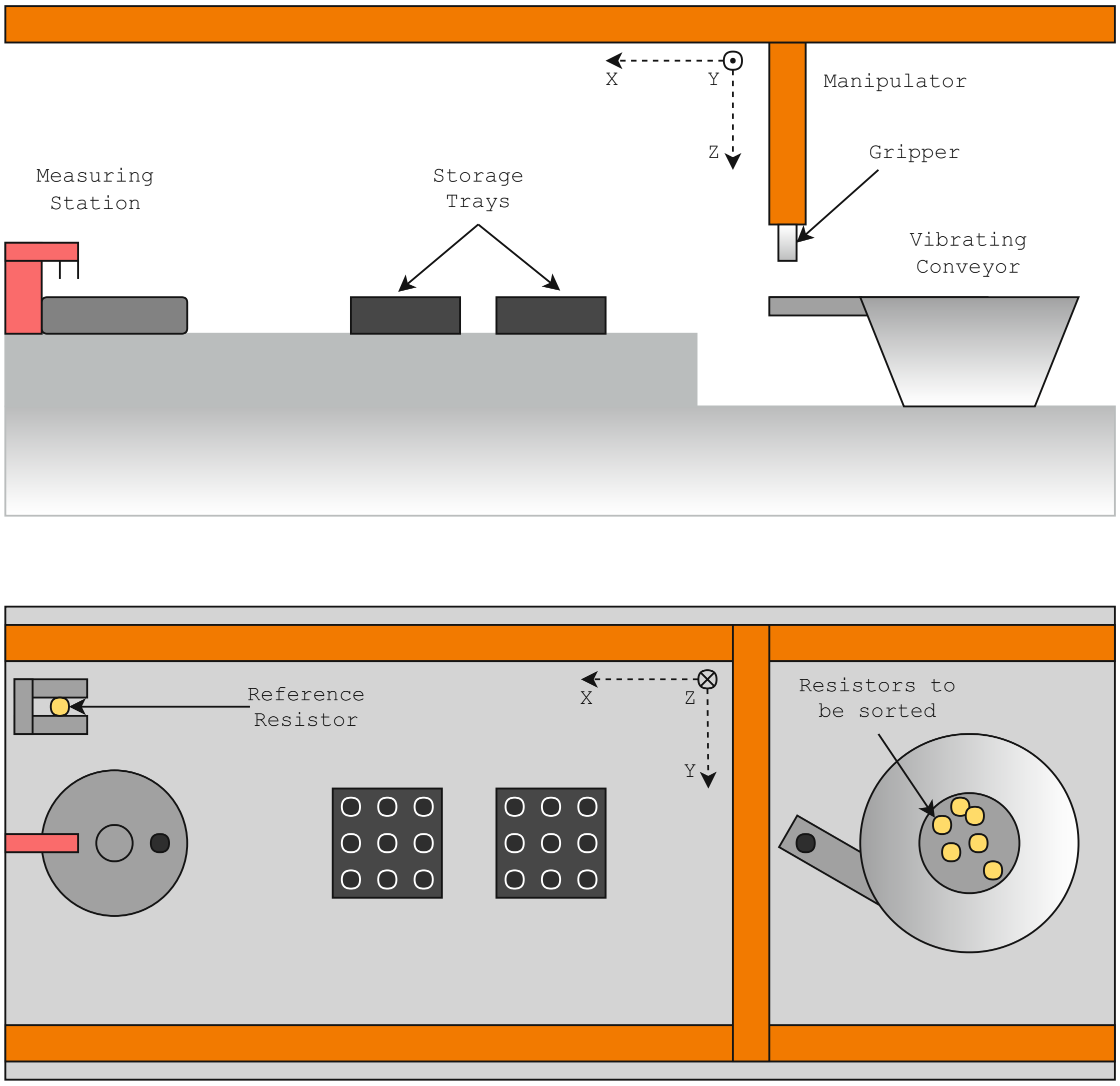


Figure 3.1 – Overview of the initial setup of the resistor sorter.

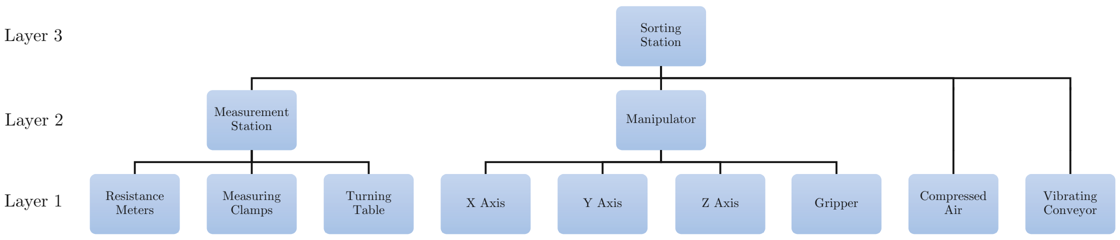


Figure 3.2 – A mechatronic component-based hierarchy for the resistor sorter.

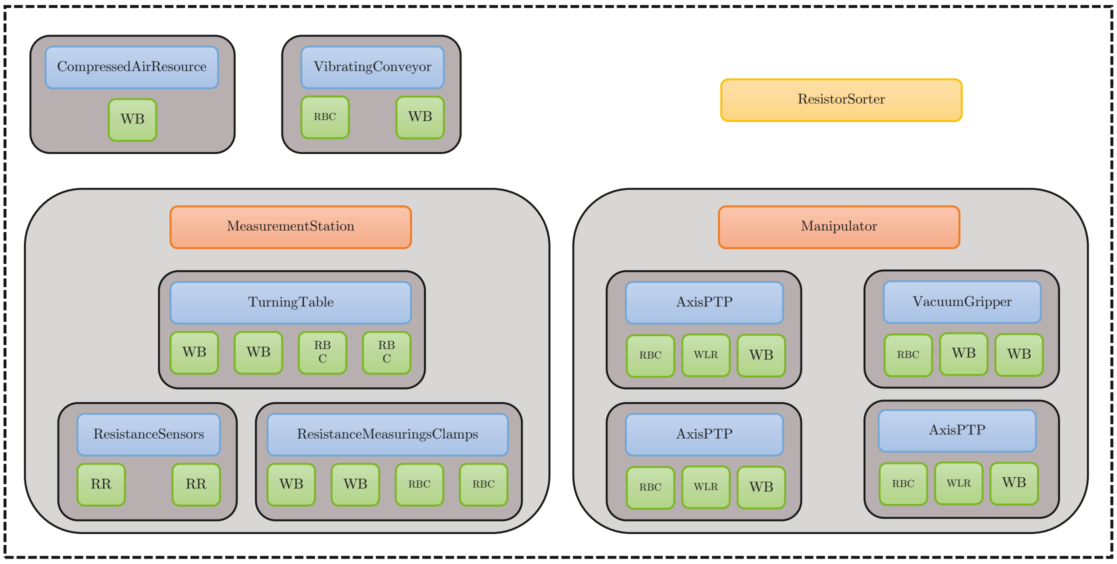


Figure 3.3 – Hierarchical structuring of the control application in 4diac.

3.1.1 Layer 1

The layer 1 includes the components that encapsulate basic hardware control operations (motor or gripper controllers, for example). In IEC 61499, there is one more layer below the layer 1, the layer 0, which includes the SIFBs that each component in layer 1 requires in order to communicate with the real hardware. The state of the art design guidelines suggest to encapsulate each layer 1 component with its corresponding layer 0 SIFBs into a SubApp. Therefore, layer 0 and layer 1 constitute the hardware-dependent part of the control application.

3.1.1.1 Resistance Sensors

The function block `ResistanceSensor` in layer 1 controls the two resistance sensors located in the resistor sorter: one sensor that measures the reference resistor and another sensor that measures the resistor that has to be sorted. Through the adapter interface `AMeasurements` the service offered for higher level components is an event `Measurements_Done` to signal that the measurements are ready. This event also samples the data corresponding to the two resistors (`Reference_Value` and `ToBeSorted_Value`), which is also fed back to the higher level component. The adapter interface also defines the requests that the component can take from upper level components: in this case, an event `Read_Values` from a higher level component commands the component to read the resistor values. The SubApp is depicted in Figure 3.4.

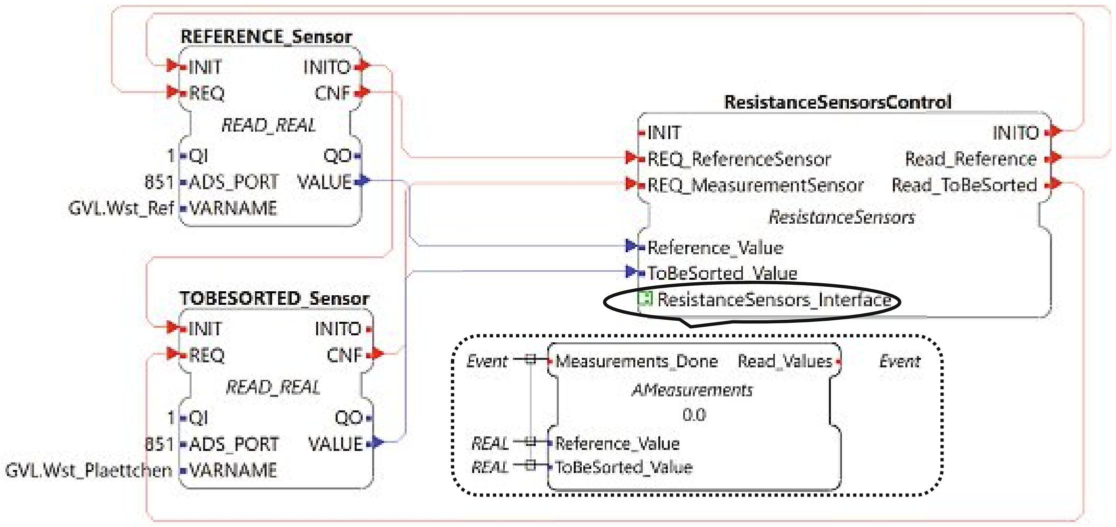


Figure 3.4 – SubApp containing the `ResistanceSensors` FB, a layer 1 component. The `READ_REAL` SIFBs (layer 0 components) read the resistor values coming from the sensors and send them to the `ResistanceSensors` FB. This FB can then send this values on command to higher level components, and its services are depicted in the adapter interface `AMeasurements`.

3.1.1.2 Resistance Measuring Clamps

In the measuring station, there is a pair of measuring clamps to measure the resistor which is in the turning table. These clamps have a pneumatic actuator and move along the vertical axis. When a resistor in the turning table is in the measuring position, underneath the measuring clamps, the pneumatic actuator should make the clamps descent until contact with the resistor is made. After the measurement is taken, the pneumatic actuator retracts the clamps back to their initial position.

The function block `ResistanceMeasuringClamps` (see Figure 3.5), as a layer 1 component, only implements a very simple functionality: extending and retracting the clamps that measure the resistor. The adapter `AMeasuringClamps` provides the communication interface with higher level components. Higher level components can request the actual position of the clamps as well as command the clamps to retract (`GoTo_InitialPosition`) or extend (`GoTo_Measurement`). Through the adapter the component also feeds back to upper layer components the current position of the clamps when the position is requested, by triggering events `Initial_Position` and `Measurement_Position`.

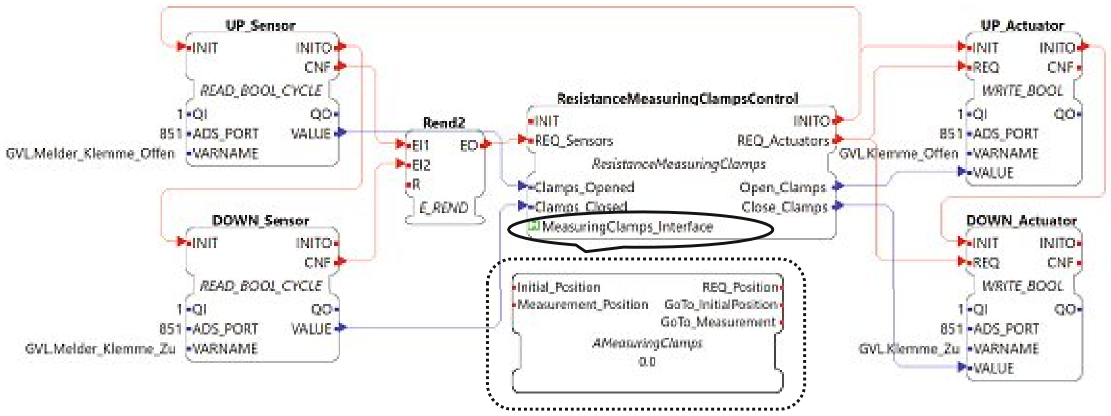


Figure 3.5 – SubApp containing the `ResistanceMeasuringClamps` FB, a layer 1 component. The `READ_BOOL_CYCLE` SIFBs (layer 0 components) read the clamp sensors, and the `WRITE_BOOL` SIFBs (layer 0 components) update the actuator values. The services that the `ResistanceMeasuringClamps` FB offers for higher level components are depicted in the adapter interface `AMeasuringClamps`.

3.1.1.3 Turning Table

The turning table in the measurement station has two possible positions. By rotating, it moves the resistor from a pick-up position (where the manipulator can drop it or pick it up) to a position just underneath the clamps which measure the resistance.

The function block `TurningTable` works in a similar way as the FB `Resis-tanceMeasuringClamps`. Through an adapter of type `ATurningTable`, a higher level component can request the FB `TurningTable` to send back the current position of the turning table or command the turning table to move clockwise or counter-clockwise. The SubApp with the layer 1 component and its corresponding layer 0 SIFBs is depicted in Figure 3.6.

3.1.1.4 Manipulator Axis

The manipulator has three axes (X,Y,Z), and in the initial configuration each axis is driven by electric motors. A function block called `AxisPTP` in layer 1 commands a motor to drive its corresponding axis until a certain position is reached. For each axis, the function is the same: go to point A and stop when point A is reached. Therefore, the same component (`AxisPTP`) is used for each axis. In Figure 3.7, the SubApp that encapsulate the control of the X axis is represented, but for the Y and Z axis the design is exactly the same.

This layer 1 component is responsible of sending the controller of the electric motor the coordinate of the desired position and a command to drive the motor.

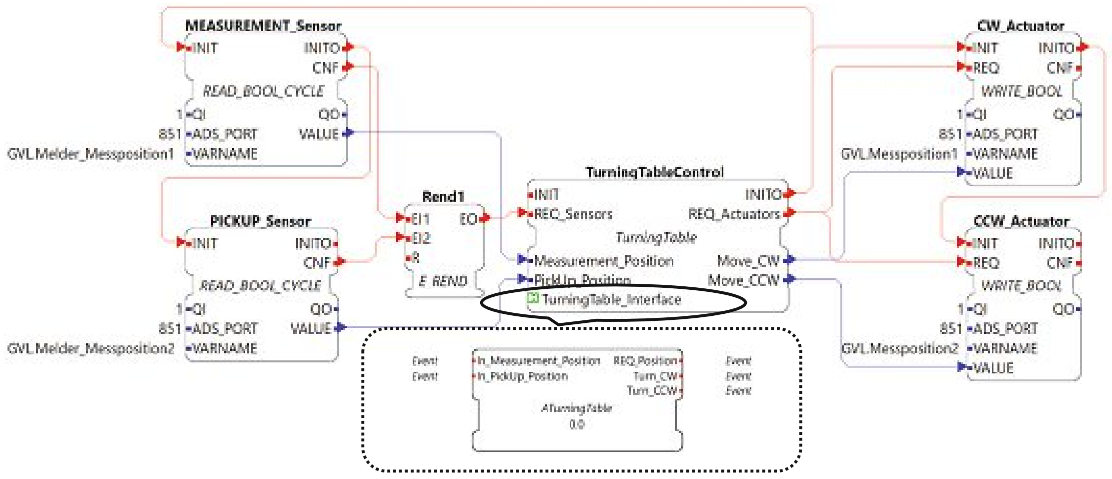


Figure 3.6 – SubApp containing the `TurningTable` FB, a layer 1 component. The `READ_BOOL_CYCLE` SIFBs (layer 0 components) read the position sensors of the turning table, and the `WRITE_BOOL` SIFBs (layer 0 components) update the actuator values of the component. The services that the `TurningTable` FB offers for higher level components are depicted in the adapter interface `ATurningTable`.

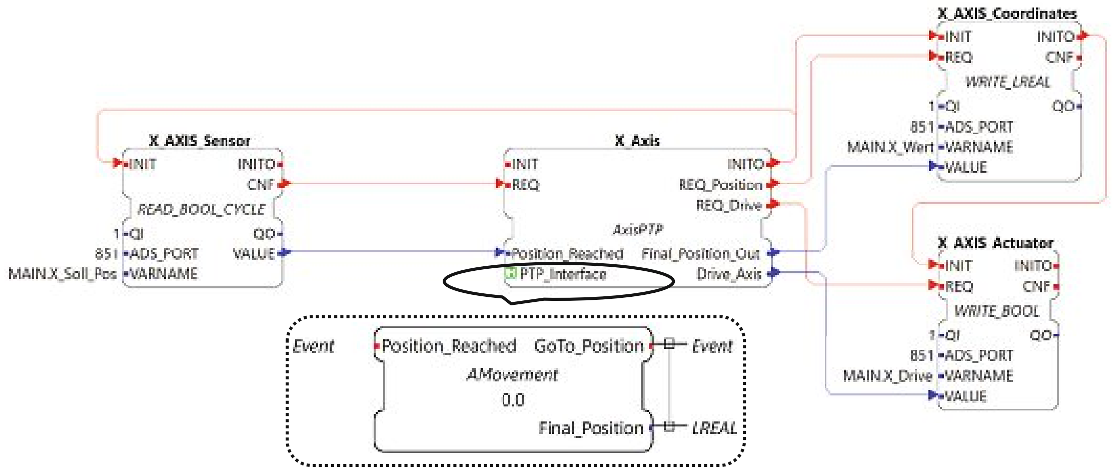


Figure 3.7 – SubApp containing the `AxisPTP` FB, a layer 1 component. The `READ_-BOOL_CYCLE` SIFB (layer 0 component) reads the position of the manipulator in one axis. The `WRITE_LREAL` SIFB (layer 0 component) communicates with the motor controller in order to specify the final position coordinates, while the `WRITE_BOOL` SIFB (layer 0 component) commands the motor controller to start the movement. The services that the `AxisPTP` FB offers for higher level components are depicted in the adapter interface `AMovement`.

As feedback from the motor controller, the component can sense that the final position has been reached. Through the adapter `AMovement` the component receives a command to move (and the corresponding coordinate) from the upper layer. As feedback for the upper layer, it triggers an event when the desired position is reached.

3.1.1.5 Gripper

The resistors are picked up with a vacuum gripper that is attached to the vertical axis (Z axis) of the manipulator. The component responsible for controlling the basic functionality of the vacuum gripper is represented as the function block `VacuumGripper` in Figure 3.8. The communication interface with the upper layer is represented in the adapter `AGripper`. A higher level component can command the FB `VacuumGripper` to either grip or release a component, and the component `VacuumGripper` can signal to the upper layer that the component has been gripped or released.

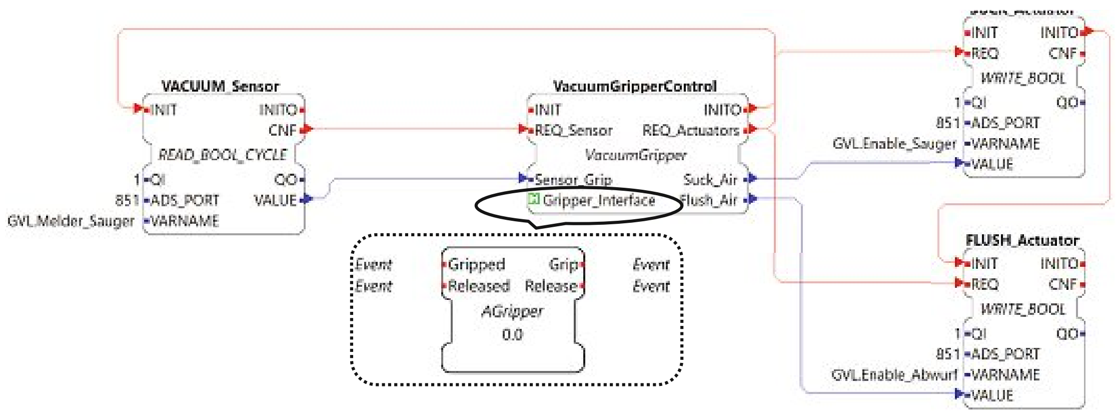


Figure 3.8 – SubApp containing the `VacuumGripper` FB, a layer 1 component. The `READ_BOOL_CYCLE` SIFB (layer 0 component) reads the state of the gripper (gripped or not gripped), and the `WRITE_BOOL` SIFBs (layer 0 components) command the actuators of the gripper (suck air or flush air). The services that the `VacuumGripper` FB offers for higher level components are depicted in the adapter interface `AGripper`.

3.1.1.6 Compressed Air Resource

Compressed air has to be provided to the different pneumatic actuators. To control the resource that provides compressed air a simple layer 1 component is needed. The function block `CompressedAirResource`, depicted in Figure 3.9, provides a simple interface to activate or deactivate the compressed air resource.

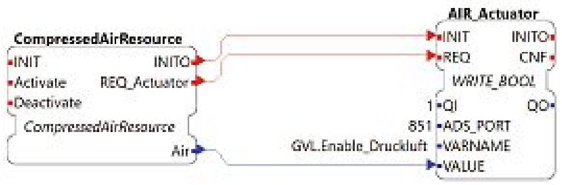


Figure 3.9 – SubApp containing the `CompressedAirResource` FB, a layer 1 component. This FB is only used to activate or deactivate the compressed air resource that feeds all the pneumatic components. The SIFB `WRITE_BOOL` communicates with the actuator.

3.1.1.7 Vibrating Conveyor

At the top of the conveyor, there is an extension where the resistors are driven to by vibrations. At the end of this extension, there is a position sensor. When a resistor arrives at the end of the extension, the position sensor signals a "high" value. This indicates that there is a resistor ready to be picked up and therefore, the conveyor should stop vibrating. When the resistor is collected, the position sensor value goes back to "low" and the conveyor will vibrate again until another resistor reaches the desired position.

This functionality is encapsulated by the FB `VibratingConveyor` in Figure 3.10. The adapter `AConveyor` provides the interface of the communication with upper layer components. An upper layer component can request to get feedback about the availability of a component through the event `REQ_State` and the `Vibrating-Conveyor` component can then signal an event `PartFeedback` back to the upper component with the corresponding status (available/not available) represented in `PartStatus`.

3.1.2 Layer 2

All the layers in the hierarchy from layer 2 and up include components that are responsible for logic control operations. Therefore, all these layers constitute the logic part of the application, according to the state of the art guidelines. The guidelines also suggest to encapsulate each component in layer 2 with its corresponding layer 1 SubApps into a single SubApp.

3.1.2.1 Measurement Station

A layer 2 component, represented as the FB `MeasurementStation` in Figure 3.11 controls three sub-components (each one encapsulated as a SubApp) through the corresponding adapter interfaces: the resistor sensors, the pneumatic clamp actu-

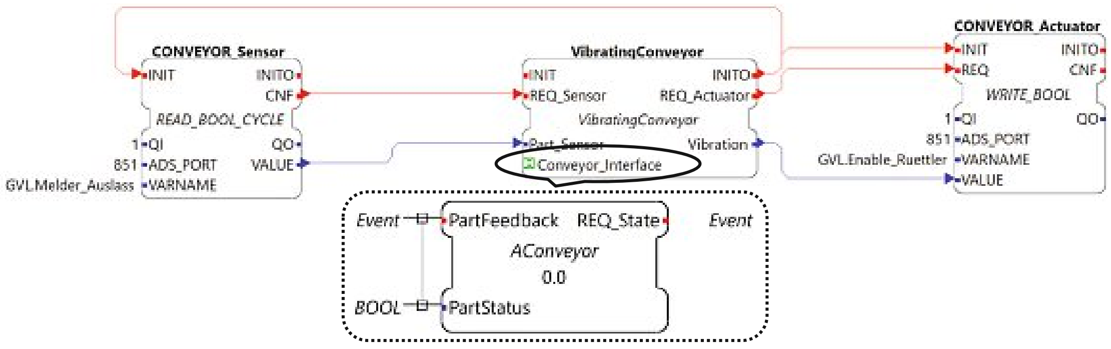


Figure 3.10 – SubApp containing the `VibratingConveyor` FB, a layer 1 component. The `READ_BOOL_CYCLE` SIFB (layer 0 component) reads the state of the position sensor at the end of the conveyor (part in place or not), and the `WRITE_BOOL` SIFB (layer 0 component) commands the actuator of the conveyor to vibrate. The services that the `VibratingConveyor` FB offers for higher level components are depicted in the adapter interface `AConveyor`.

ator and the turning table. A layer 3 component can request the `MeasurementSta-tion` component to take a measurement by triggering the event `Measure_Resistor`. On the arrival of this event, the `MeasurementStation` component manages its sub-components in order to:

1. Turn the turning table when a resistor is placed by a manipulator in order to position the resistor that is going to be sorted underneath the clamps.

2. Move the measuring clamps down to measure the resistor that has to be sorted.

3. Measure as well the reference resistor (which is located in a static position all the time, next to the turning table).

4. Retract the measuring clamps and turn the table back to its initial position.

When this process is finished, the `MeasurementStation` triggers the event `Re-sistorMeasurement_Done` to signal that the measurement is done and the resistor back at a pick-up position. This event also samples the values of the two measured resistors. The combination of this event and the two measured values serve as feedback for an upper layer component.

3.1.2.2 Manipulator

The function of the manipulator in the current control application, from a logic perspective, is to either perform a simple point-to-point (PTP) movement, or to

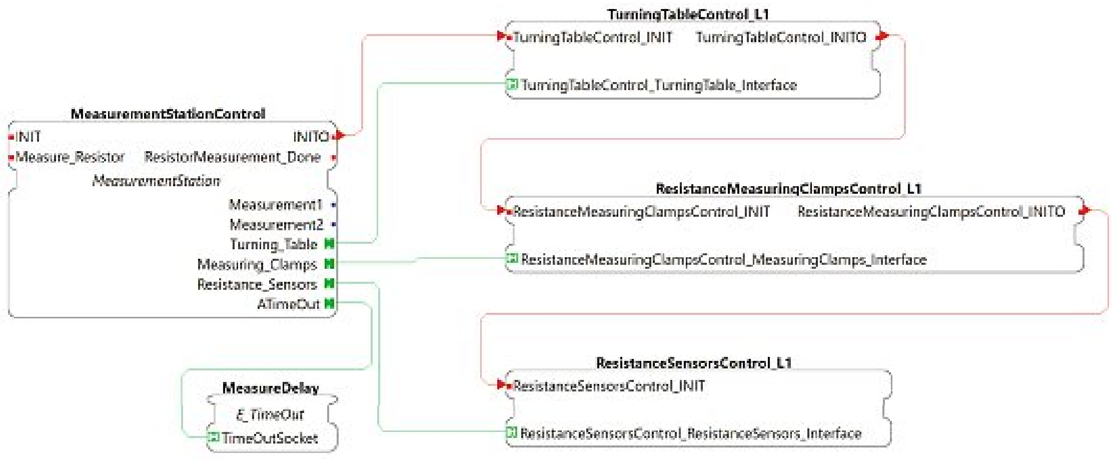

TurningTableControl_L1
TurningTableControl_INIT
TurningTableControl_INITO
TurningTableControl_TurningTable_Interface
MeasurementStationControl
INIT
INITO
Measure_Resistor
ResistorMeasurement_Done
MeasurementStation
Measurement1
Measurement2
Turning_Table
Measuring_Clamps
Resistance_Sensors
ATimeOut
ResistanceMeasuringClampsControl_L1
ResistanceMeasuringClampsControl_INIT
ResistanceMeasuringClampsControl_INITO
ResistanceMeasuringClampsControl_MeasuringClamps_Interface
ResistanceSensorsControl_L1
ResistanceSensorsControl_INIT
ResistanceSensorsControl_ResistanceSensors_Interface
MeasureDelay
E_TimeOut
TimeOutSocket

Figure 3.11 – SubApp containing the `MeasurementStation` FB (layer 2 component) and the layer 1 SubApps that it controls (the measurement station controls the turning table, the measuring clamps and the resistance sensors).

perform a PTP pick-and-place (extend vertical axis, suck component with gripper, retract vertical axis, move to final x-y point, extend vertical axis, flush component with gripper and retract vertical axis). These two functionalities are implemented in the Function Block `Manipulator` (see Figure 3.12) , and can be requested by an upper layer component through the events `Move_PTP` and `PickAndPlace_PTP`. The upper layer component must also pass the coordinates of the final position of the requested action. When one of these two actions is done, the `Manipulator` triggers the corresponding event `Move_Done` and `PickAndPlace_Done` as feedback for the upper layer component.

3.1.3 Layer 3

In this application, the top layer is the layer 3 and it includes only one component which acts as the main coordinator. This component is encapsulated in the function block `ResistorSorter` and directly controls four SubApps: one to control the measurement station, one to control the manipulator, one to control the compressed air resource and one to control the vibrating conveyor, as seen on Figure 3.13.

The functionality that this FB encapsulates is the following:

1. At initialization, it activates the compressed air resource and the vibrating conveyor.

2. When the component is commanded to start the sorting process by triggering

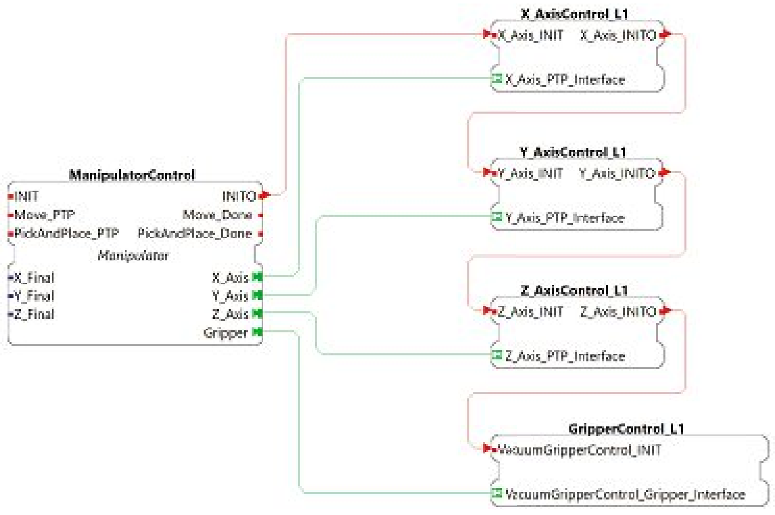


Figure 3.12 – SubApp containing the `Manipulator` FB (layer 2 component) and the layer 1 SubApps that it controls (the manipulator controls three manipulator axes and the gripper).

the event `Sort`, it waits until a resistor is available at the pick-up position in the conveyor.

3. When a resistor is available, it commands to the manipulator (which should be located on top of the pick-up position in the conveyor at initialization) to perform a pick-and-place action in order to carry the resistor to the measurement station.

4. When a resistor arrives at the measurement station, it commands the measurement station to perform the measuring action, and when the values are received it calculates if the resistor is between tolerances.

5. Commands the manipulator to perform a pick-and-place action in order to carry the resistor from the measurement station into the corresponding storage tray (and its corresponding position in the storage tray, with nine positions in total each).

6. Commands the manipulator to perform a PTP movement back to the initial position (above the pick-up position in the conveyor) after placing the resistor in a storage tray.

The component provides feedback to the operator, by triggering an event `Sorting_Done` every time that a resistor is sorted, an event `Tray_Full` when one storage tray is full (in this case also the sorting process is ended) and also displays the number of resistors that where under tolerances (`GoodResistors`) or outside tolerances (`BadResistors`).

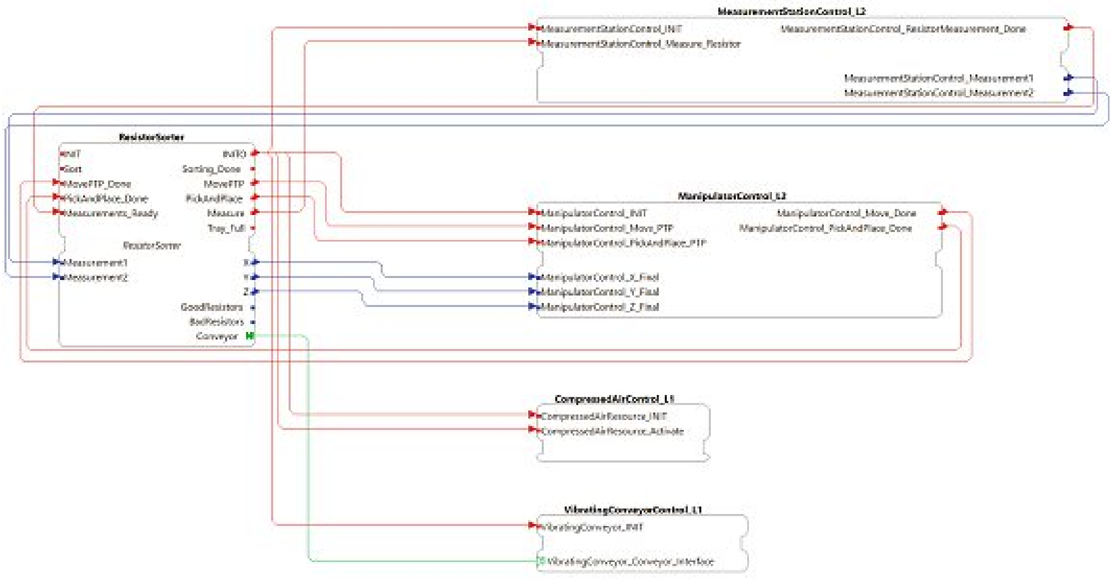

MeasurementStationControl_L2
MeasurementStationControl_INIT
MeasurementStationControl_Measure_Resistor
MeasurementStationControl_ResistorMeasurement_Done
MeasurementStationControl_Measurement1
MeasurementStationControl_Measurement2
ResistorSorter
INIT
Sort
MoveFTP_Done
PickAndPlace_Done
Measurements_Ready
INITO
Sorting_Done
MoveFTP
PickAndPlace
Measure
Tray_Full
ResistorSorter
Measurement1
Measurement2
X
Y
Z
GoodResistors
BadResistors
Conveyor
ManipulatorControl_L2
ManipulatorControl_INIT
ManipulatorControl_Move_PTP
ManipulatorControl_PickAndPlace_PTP
ManipulatorControl_Move_Done
ManipulatorControl_PickAndPlace_Done
ManipulatorControl_X_Final
ManipulatorControl_Y_Final
ManipulatorControl_Z_Final
CompressedAirControl_L1
CompressedAirResource_INIT
CompressedAirResource_Activate
VibratingConveyorControl_L1
VibratingConveyor_INIT
VibratingConveyor_Conveyor_Interface

Figure 3.13 – View of the complete control application. The `ResistorSorter` FB (layer 3 component) is the main coordinator of the control application.

3.2 Case Study: Change in the Setup

In this new setup for the case study, the electrical motor that drives the vertical axis (Z axis) of the manipulator is replaced by a pneumatic cylinder (see Figure 3.14). Since the logic of the control application remains exactly the same, the goal is to see how a change in hardware affects the components of the control application. The hierarchical composition in the application stays the same as well. After rebuilding the control application, the components that needed to be changed are listed in the following subsections. The rest of the components remain the same as in the initial setup.

3.2.1 Layer 1

The FB `AxisPTP` is no longer valid for the Z axis. Even though the logic keeps being the same, a different FB interface is need due to how each hardware component works. In Figure 3.15, we can clearly see how the interfaces are not compatible (different number of sensors/actuators and different data types), and therefore, we

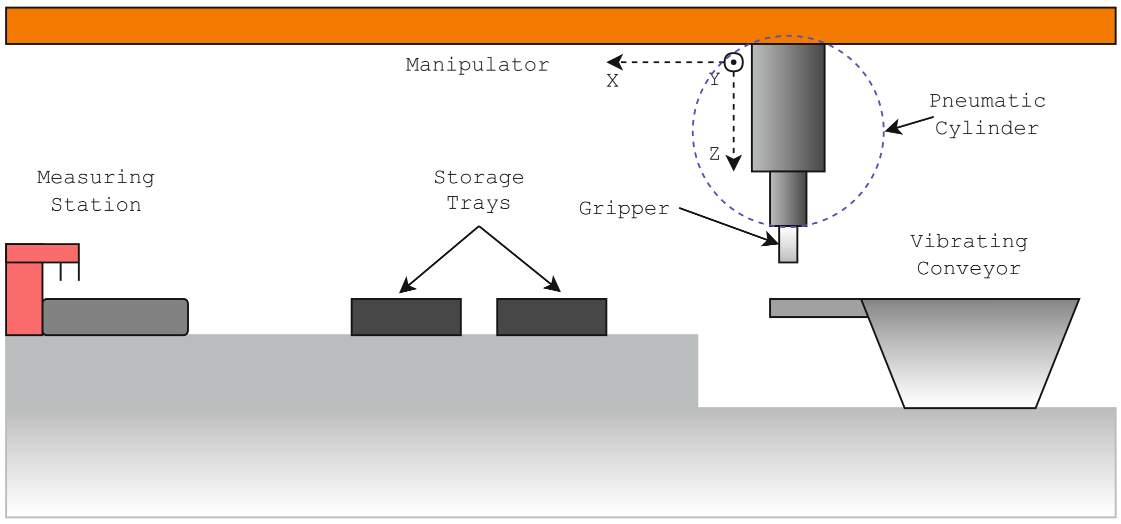


Figure 3.14 – Overview of the new setup, where the vertical axis electric motor has been swapped by a pneumatic actuator.

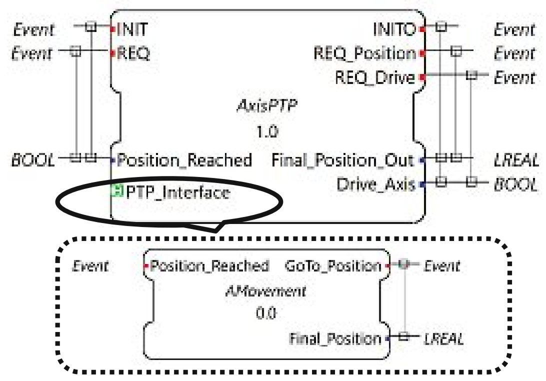
cannot reuse the component `AxisPTP`. A new FB had to be created, `AxisPneumatic`, since the used pneumatic cylinder cannot take coordinates to move. Also a new adapter `ACylinder` was needed, since the services offered by this component are also different. The different adapter interface implies a different service offered to the higher level components. As a consequence, the higher level components will have to adapt to this new services.

3.2.2 Layer 2

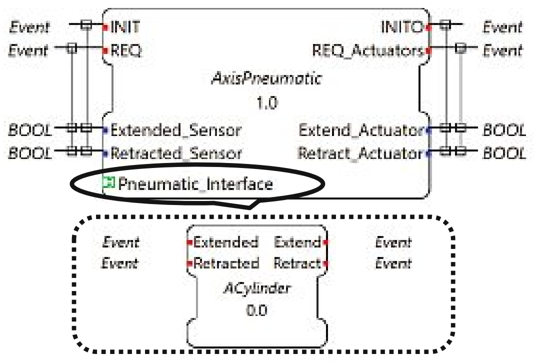
The FB `Manipulator` is also no longer valid. This time, the changes in the interface are not that severe (see Figure 3.16): we no longer need the coordinates for the Z axis as an input and the adapter type for the Z axis has also changed, which is now of type `ACylinder`. However, parts of the internal code of the component have to be modified (states, transitions, and algorithms), because the `Manipulator` FB was programmed to work with coordinates, but now in order to control the Z axis it needs to deal with "extend" and "retract" actions.

3.2.3 Layer 3

The FB `ResistorSorter` is, again, no longer valid. This time also, the changes in the interface are minimal since we are not longer passing coordinates for the Z Axis to the components in lower layers (see Figure 3.17). However, as with the `Manipulator` FB, parts of the internal code of the component have to be modified (states,



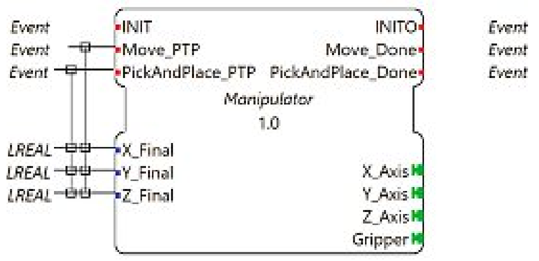
(a) `AxisPTP` FB.



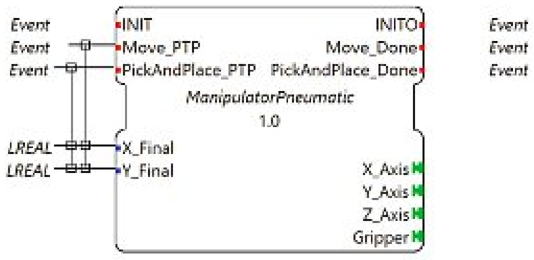
(b) `AxisPneumatic` FB.

Figure 3.15 – Incompatible layer 1 component interfaces.

transitions, and algorithms), because the `ResistorSorter` FB was programmed to work with coordinates, but now coordinates are no longer needed for the Z axis.

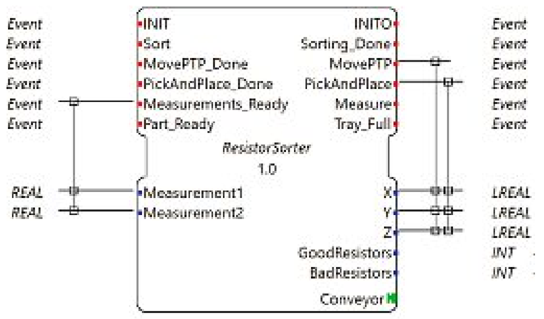


(a) `Manipulator` FB.

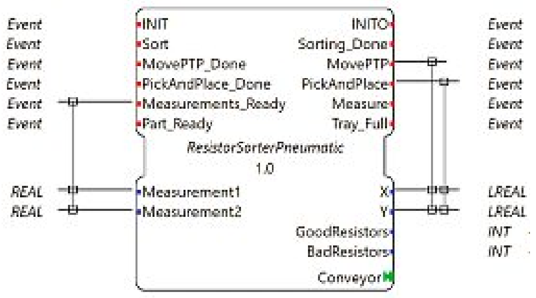


(b) `ManipulatorPneumatic` FB.

Figure 3.16 – Incompatible layer 2 component interfaces.



(a) `ResistorSorter` FB.



(b) `ResistorSorterPneumatic` FB.

Figure 3.17 – Incompatible layer 3 component interfaces.

3.3 Conclusions of the Case Study

The goal of the case study was to test if the current component-based design approaches for industrial control systems success in achieving truly reusable and modular applications, where hardware and control logic are decoupled. The conclusions of this case study can answer the Research Question 1.

As seen with this simple example, hardware and logic are still mixed together. The hardware implementation was changed (electric motor driven axis replaced by pneumatic cylinder) while the logic stayed the same. This change in hardware affected all the layers in the application, leaving the initial control application useless. The FBs that had to be modified (both the interface and the ECC) in order to work with the new pneumatic actuator were: `AxisPTP`, `Manipulator` and `ResistorSorter`. Basically, the fact that a pneumatic cylinder does not work with coordinates required to change the `Manipulator` and `ResistorSorter` FBs, since these two components where implemented with coordinates for positioning. As represented in Figure 3.18, all the layers in the hierarchy were affected by a hardware change, even though the control logic of the application remained the same.

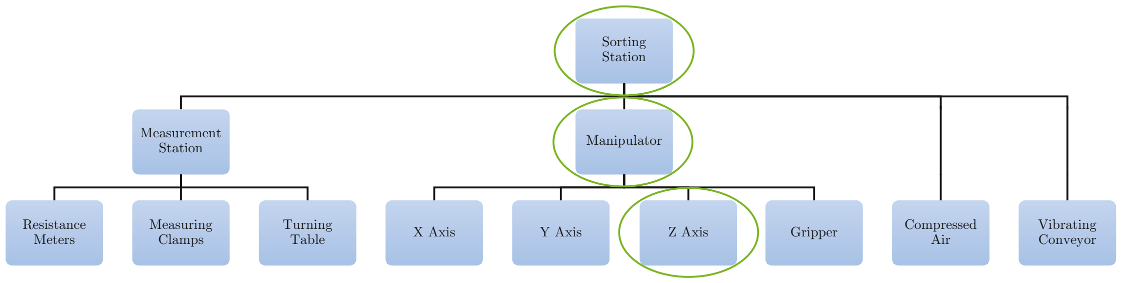


Figure 3.18 – The components marked with a circle could not be reused after the hardware change in the vertical axis of the manipulator.

It is clear that there is still something missing in order to achieve truly modular and reusable applications. The 'what' or logic is still coupled with the 'how' or implementation, at least in the higher layers of the component hierarchy. Something is missing to further decouple the 'what' and the 'how', so changes in the implementation only require minimal reconfiguration at software level, and do not require any modifications on logic-only components. Since different hardware often require the same logic, decoupling these hardware components from their logic control components would enable full component reuse.

3.4 Proposed Design Approach

The most important aspect for component reuse in industrial automation control applications is hardware and logic independence. This means that the whole control logic must be able to be defined in a complete hardware-independent way, in order to be able to reuse this components in other applications where the logic needed is the same, but the hardware or physical plant configuration has changed. So far, we have seen with the experiment in the resistor sorter (Sections 3.1, 3.2, and 3.3) that there are still hardware-specific parameters in higher level components in the hierarchy. Thus, in this thesis a methodology is presented with the aim of achieving two completely independent parts (a logic part and a hardware-dependent part) that can then be "glued" together under deployment depending on the plant configuration and hardware used.

This approach has a MDA-like architecture as other concepts that were reviewed in Section 2.5.1, since it divides the control application into two different parts: a logic part or PIM and a hardware-dependent part or PDM. These two parts are then mapped or coupled together in order to obtain the specific control application (see Figure 3.19). The corresponding hierarchical composition is depicted in Figure 3.20. In contrast to the state of the art approaches, a new intermediate layer has been introduced in the hierarchy, the *coupling* or *translation* layer. A control application designed following this approach can therefore be divided into three parts:

- **Logic Part**

  This part of the application includes all the components from layers 2 and up. Components in this part of the application should totally define the control logic of the application. As mentioned before, no hardware-related parameters are allowed in this part, these components must be completely decoupled from the specific hardware used. It is proposed to use **symbolic names**, as a *string* data type for example, to represent positions or other parameters that could be easily coupled to hardware in these higher levels in the hierarchy. The idea is to have a control flow that is defined in a very high level and simple language, similar to human language. For example, a command could be send to a manipulator to move to a certain position as "move to *positionA*", being *positionA* a *string*.

- **Hardware-Dependent Part**

  Components at the bottom of the hierarchy (layer 1) are hardware-dependant. These components implement hardware control operations. For example, a FB in this layer could be an electric motor controller.

- **Coupling or Translation Layer**

  Between the logic part and the hardware-dependent part there must be an intermediate layer that couples both parts, which are completely independent between them. Otherwise, the communication between the logic and hardware-dependent parts is not possible, since the FB interfaces of both parts will not match. In order to reduce reconfiguration times when there is a change in hardware or in the physical plant configuration (reallocation of different stations, dimension increase/decrease, etc.), it is also proposed that this components in the coupling layer are not defined following IEC 61131-3 or IEC 61499 coding standards. To reduce complexity, a very simple Domain-Specific Language (DSL) should allow the user to provide just the minimal information required for the coupling, such as type of logic and hardware to couple and a translation between symbolic names used in the logic part of the application and their corresponding hardware-dependent parameters. Then, code is generated out of this definition for the specific implementation (IEC 61131 or IEC 61499) in order to create the required FBs for this coupling.

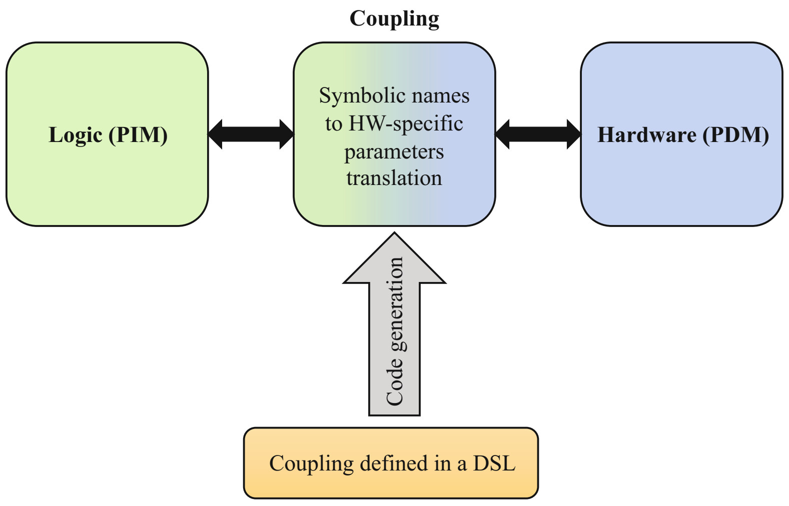

Coupling
Logic (PIM)
Symbolic names to HW-specific parameters translation
Hardware (PDM)
Code generation
Coupling defined in a DSL

Figure 3.19 – A MDA-like approach, with two independent application parts, the PIM and the PDM. These two parts can be coupled together in order to obtain the specific control application. The necessary information for the coupling is defined in a DSL, and code is automatically generated out of this definition in order to perform the coupling.

The design flow shall start with the design of the logic part of the application. Then at deployment, hardware components are added to the application and both

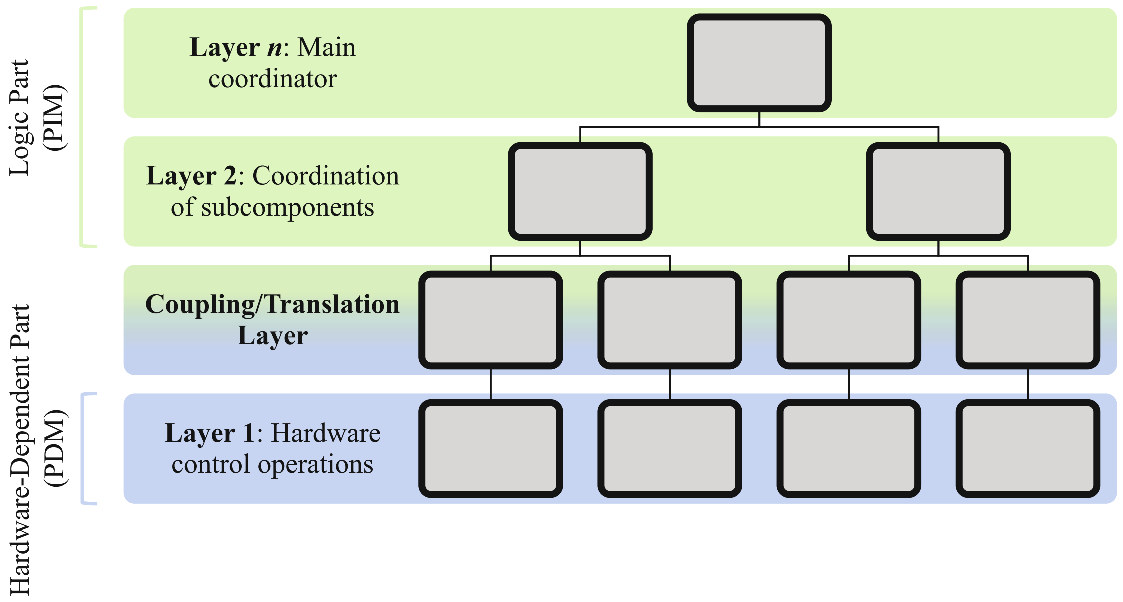

Logic Part (PIM)
Hardware-Dependent Part (PDM)
Layer *n*: Main coordinator
Layer 2: Coordination of subcomponents
Coupling/Translation Layer
Layer 1: Hardware control operations

Figure 3.20 – A view of the proposed hierarchy. The bottom layer components are hardware-dependent, while layer 2 and higher layers only contain control logic. A new intermediate layer between layer 2 and layer 1 performs the necessary translations in order to couple both parts of the application.

parts are coupled, providing the minimal information necessary for this coupling through the DSL.

CHAPTER 4

Implementation

The concept presented in Section 3.4 is going to be implemented in this chapter for IEC 61499. Furthermore, the creation of the function blocks for the new coupling layer, the domain-specific language and the code generation are detailed in this chapter for the two cases that were presented for the resistor sorter plant described in Chapter 3 (using an electric motor or a pneumatic cylinder to perform a linear movement in one axis). It is shown as well how the concept from Section 3.4 can be totally integrated in the Eclipse 4diac IDE for IEC 61499.

4.1 Implementation-Specific Details in IEC 61499

With respect to the generic concept presented in Section 3.4, the specific implementation for IEC 61499 implies the presence of an additional layer, the **layer 0**. Unlike in IEC 61131, in IEC 61499 IO writing and reading is performed in a special type of FB called SIFB. These SIFBs will be allocated to layer 0, while the components in the other layers will always be regular FBs (either BFBs or CFBs). The hierarchy for IEC 61499 is depicted in Figure 4.1.

Another important aspect of an IEC 61499 implementation is the use of the adapter concept for communication between components of different hierarchy levels. The goal of this new intermediate layer is to couple the logic and hardware-dependant parts of the application. For an IEC 61499 application, what has to be coupled are two different adapter interfaces. A component in this coupling layer shall be able to translate the commands coming from an adapter interface in layer 2 so that the adapter interface in layer 1 can understand these commands. For example, a component in layer 2 might issue a command to a motor controller component in layer 1 like this: "go to position *posX1*". Since the message is

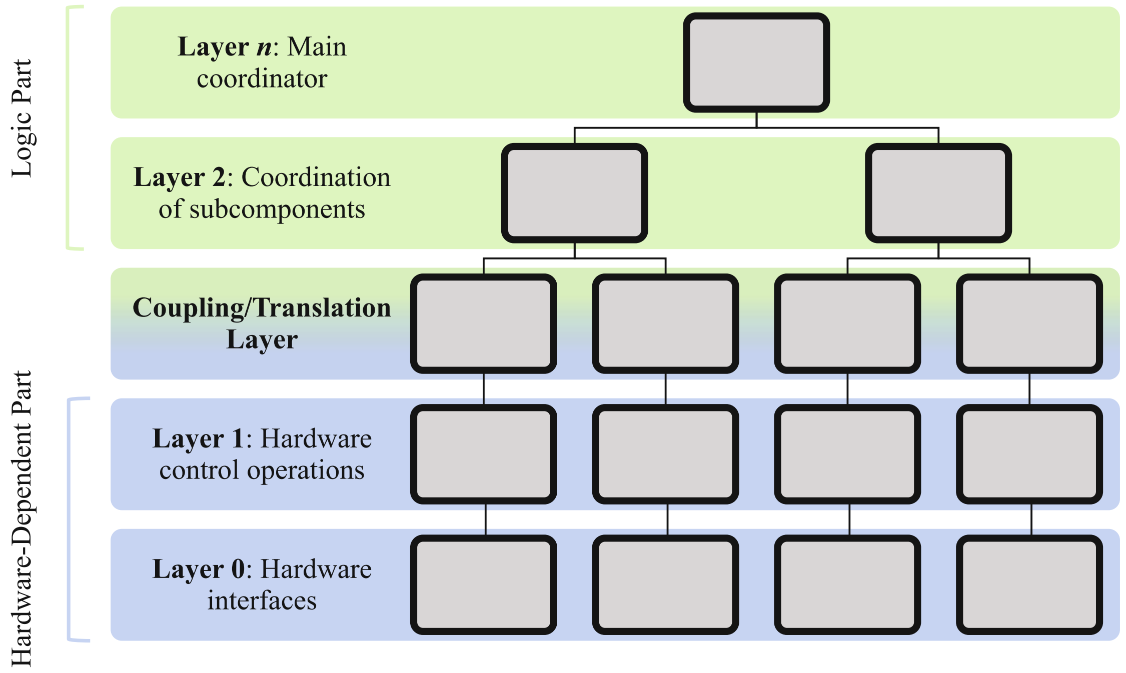

Logic Part
Hardware-Dependent Part
Layer *n*: Main coordinator
Layer 2: Coordination of subcomponents
Coupling/Translation Layer
Layer 1: Hardware control operations
Layer 0: Hardware interfaces

Figure 4.1 – A view of the proposed hierarchy for IEC 61499 applications.

coming from the logic part of the application, the adapter interface (seen as a socket) of the component in layer 2 will have at least two outputs, one to send an event that commands the movement and a data output containing the symbolic name of the position as a string. On the other hand, the adapter interface (seen as a socket) of the motor controller in layer 1 is not exactly the same. It will have as well an event to command the start of the movement but the data output will be of a different data type, for example a real number, since a motor controller would require some coordinates for the movement. The components in this new intermediate layer will therefore perform a translation between different adapters, so that a coordinate can be associated to a symbolic name in this example. The FB interface of these components is very simple, with just an adapter socket as input (with the adapter type used by the component in layer 2) and an adapter plug as output (with the adapter type used by the component in layer 1). This idea is depicted in Figure 4.2.

4.2 Developed Coupling/Translation FBs

For this thesis two FBs have been created for this new intermediate layer, which cover all the translations needed for the resistor sorter prototype available for tests. The first one performs a translation between an adapter interface for a linear

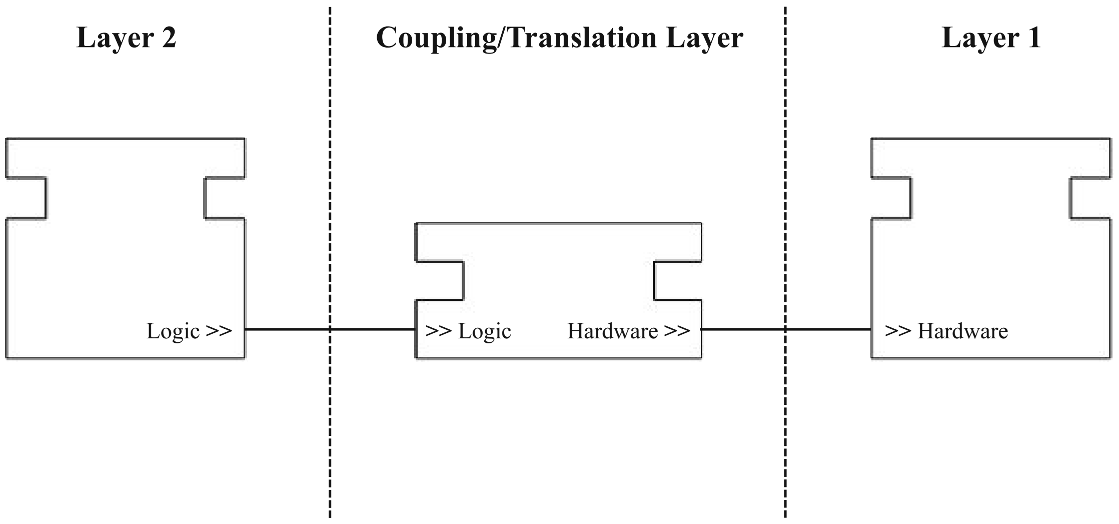


Figure 4.2 – The coupling between layers 2 and 1 in IEC 61499. The communication is performed via adapters. The coupling FBs only have one adapter input or socket and one adapter output or plug in their interfaces.

movement in one axis (or rotation around one axis) and an adapter interface for an electric motor controller. The second one performs a translation between the same previous logic (movement in one axis or rotation around one axis) and an adapter interface for a pneumatic cylinder controller. In this implementation, the adapter interface for a simple linear movement along one axis or rotation around one axis is depicted in Figure 4.3.

4.2.1 Coupling With an Electric Motor

An electric motor controller will provide an adapter interface similar to the one depicted in Figure 4.4. The FB able to couple a component with the adapter interfaces in Figure 4.3 to a component using the adapter in Figure 4.4 must be able to translate symbolic position names (*string* data type) into coordinates (*lreal* data type). The ECC of the FB created for this purpose is depicted in Figure 4.5. As seen, the ECC is very simple, with only one algorithm: `stringTOlreal`. This algorithm defines which coordinate corresponds to which symbolic name used for positioning in the logic part of the application. With this FB, an event `ToPosition` which is sent along with data of type *string* from a component in layer 2 will be translated into an event `GoTo_Position` along with data of type *lreal* for a component in layer 1.

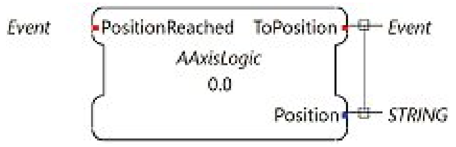


Figure 4.3 – The adapter interface for a logic movement in one axis or around one axis. A component implementing this interface can command a lower-level component to move to a certain position, and expects some feedback when the movement is done.

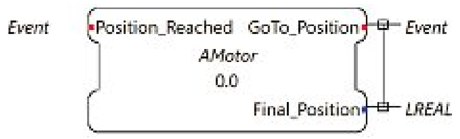


Figure 4.4 – An adapter interface for an electric motor controller. A motor controller component can take a command to drive the motor until a specific position is reached. It can also communicate higher-level components that the position has been reached.

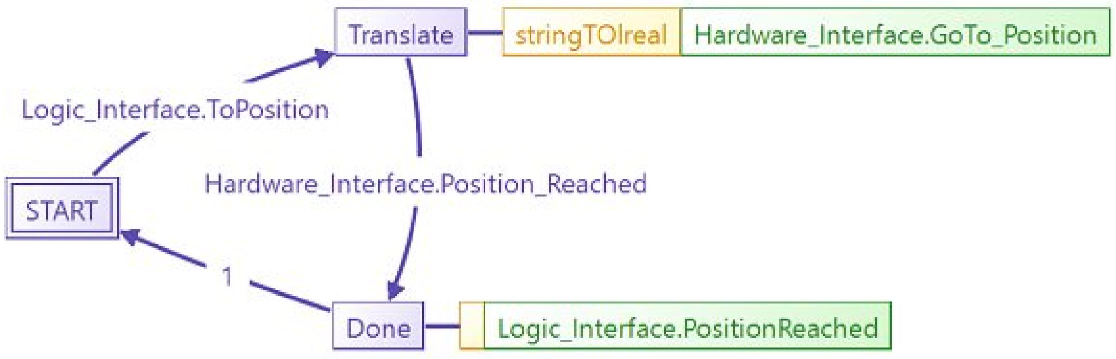

Translate
stringTOlreal
Hardware_Interface.GoTo_Position
Logic_Interface.ToPosition
START
Hardware_Interface.Position_Reached
1
Done
Logic_Interface.PositionReached

Figure 4.5 – ECC of the FB used to couple a logic movement with an electric motor controller. When a component in layer 2 commands a movement through the `AAxisLogic` adapter interface, the state *Translate* is reached, in which the necessary translation between symbolic names and coordinates is performed and communicated to the hardware component.

4.2.2 Coupling With a Pneumatic Cylinder

The adapter interface designed for a pneumatic cylinder controller is depicted in Figure 4.6. In this case, the ECC of the FB designed to translate the interface depicted in Figure 4.3 for a linear movement into the interface that a pneumatic cylinder would require is depicted in Figure 4.7. In this case the ECC does not contain any algorithm. It simply checks the symbolic name in a transition, to decide whether to command an `Extend` or `Retract` event. With this FB, an event `ToPosition` which is sent along with data of type *string* from a component in layer 2 will be translated into an event `Extend` or `Retract` for a component in layer 1.

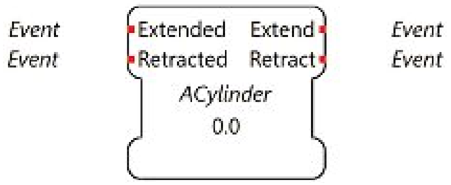


Figure 4.6 – An adapter interface for a pneumatic cylinder controller. A cylinder controller component can take a command to extend or retract the cylinder, and communicate to higher-level components when each action is done.

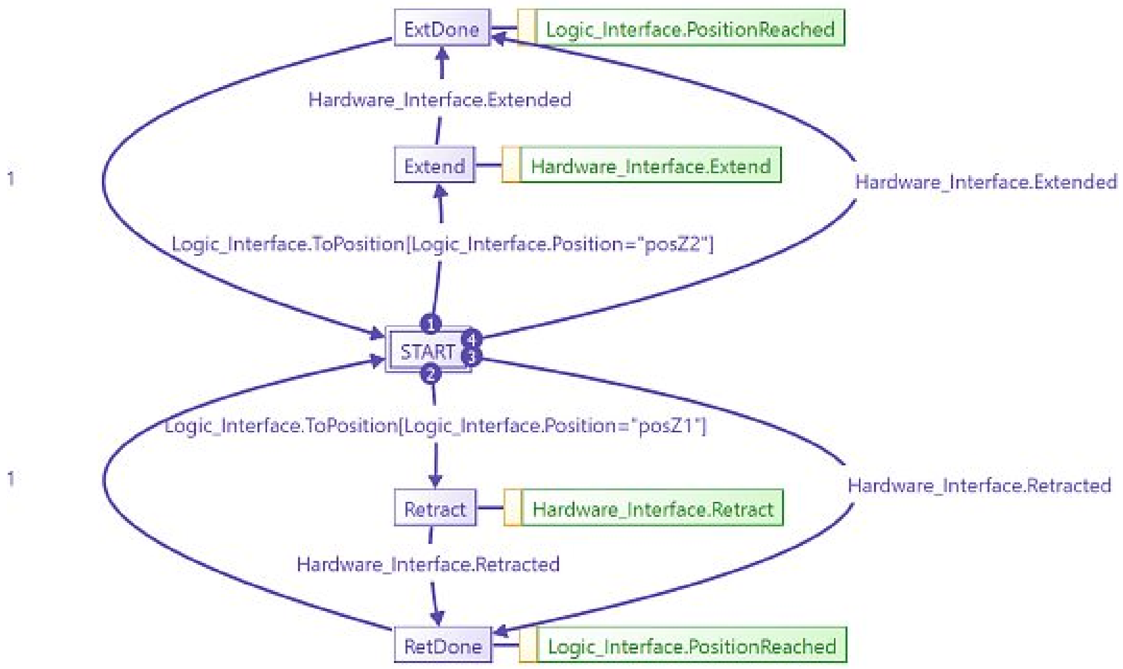

ExtDone
Logic_Interface.PositionReached
Hardware_Interface.Extended
Extend
Hardware_Interface.Extend
Hardware_Interface.Extended
1
Logic_Interface.ToPosition[Logic_Interface.Position="posZ2"]
START
1
2
3
4
Logic_Interface.ToPosition[Logic_Interface.Position="posZ1"]
1
Hardware_Interface.Retracted
Retract
Hardware_Interface.Retract
Hardware_Interface.Retracted
RetDone
Logic_Interface.PositionReached

Figure 4.7 – ECC of the FB used to couple a logic movement with a double acting pneumatic cylinder. Depending on the symbolic position name sent through the `AAxisLogic` adapter interface, either an *Extend* or *Retract* state is entered. The direct jump between *START* and the *ExtDone* or *RetDone* states on the right is a safety feature to avoid deadlocks.

As seen, adapter interfaces are in fact a way to define an interaction protocol between two entities. The FBs in this new intermediate layer provide the necessary

Table 4.1 – Functions and their corresponding big-O notation.

Notation	Name
O(1)	constant
O($log(n)$)	logarithmic
O(n)	linear
O(n^2)	quadratic
O(c^n)	exponential

information in order to translate one interaction protocol into a different one, so that the logic and hardware parts of the application can be coupled together and information can flow between each other.

4.2.3 Translation Algorithm Complexity

When evaluating an algorithm or piece of code, a differentiation has to be made between performance and complexity. Performance relates to the amount of time, memory, and disk used when a program runs, and therefore depends on the machine, compiler, and code. On the other hand, complexity refers to how a program or algorithm scale when the problem being solved gets larger. Complexity affects performance but not the other way around [21].

Big-O notation is a symbolism used in different fields like computer science to describe how fast a function grows or declines. Complexity can be expressed using big-O notation. A list of functions usually encountered when analyzing algorithms can be found on Table 4.1.

The FB responsible of coupling a one-axis movement and an electric motor controller might have to implement multiple translations between symbolic names and coordinates. The algorithm responsible of this translation is `stringTOreal` in Figure 4.5. For a simple view on how this `stringTOreal` algorithm is implemented in 4diac, let us consider the piece of code in Figure 4.8, where four translations are performed.

This implementation is not ideal, since an if-else chain is usually $O(n)$, because having n conditions means that for the worst case there are n comparisons being made. On the other hand, a switch statement could (depending on the compiler) reduce the complexity down to $O(1)$. Whether or not the complexity is reduced, a switch statement is easier to read and understand. However, the switch statement cannot be used with *strings* in some programming languages, including ST from IEC 61131-3 (which is the only language fully supported in 4diac for algorithm definitions at the moment), or C++. A way around this could be to use an *enum*

```
IF Logic_Interface.Position = "measurementX" THEN
  Hardware_Interface.Final_Position := 910.2;
ELSIF Logic_Interface.Position = "measurementY" THEN
  Hardware_Interface.Final_Position := 153.5;
ELSIF Logic_Interface.Position = "initialX" THEN
  Hardware_Interface.Final_Position := 222.9;
ELSIF Logic_Interface.Position = "initialY"  THEN
  Hardware_Interface.Final_Position := 178;
END_IF;
```

Figure 4.8 – Example of the `stringTOreal` algorithm written in the ST language, performing four translations.

and then map the *strings* to the *enum* values, so that the switch statement can be used with the numeric *enum* values. Unfortunately, *enums* are not supported in 4diac at the moment.

Then, it has to be considered that everytime a condition is checked, there is a string comparison. A string comparison has a complexity $O(n)$ for the worst case scenario:

- If the two objects are the same the result is immediate and only one operation is required. The complexity is therefore $O(1)$.
- If the two strings have different lengths, they cannot be equal, and the complexity is $O(1)$ as well.
- If the two strings have the same length, n characters have to be compared one by one until two characters are different, and the complexity is therefore $O(n)$ for the worst case scenario.

Considering the string comparison and the if-else chain, this algorithm `stringTO-real` has a quadratic complexity for the worst case scenario.

4.3 Development of a Domain-specific Language (DSL)

The Xtext framework [22] allows to define a custom language using a powerful grammar language. Furthermore it provides a full infrastructure, including parser, linker, typechecker, compiler, as well as editing support for Eclipse. Xtext is the chosen tool to develop a simple language for this project. For more information about Xtext and DSLs, please refer to [23].

The main reason for having a DSL in this component-based design approach is to simplify the creation of the FBs that are responsible for the translation of logic control commands into hardware-specific commands. Instead of having to hard-code the translation for each specific case where this coupling is required in our application, a DSL allows the programmer to just type in a very simple way the minimal information necessary for the translation. The structure of this language should be very simple: the user can introduce the type of logic and hardware components to be coupled. According to the type of hardware selected (pneumatic, hydraulic, electric, etc...) the user should introduce additional information in order to link the symbolic names used in the logic part of the application with the information that the specific hardware requires (coordinates in case of using an electric-motor based positioning system or "extend" and "retract" actions in case of using a pneumatic cylinder, for example).

```
logic movement

hardware electric axistranslation1 {
  posX1 = 123.0
  posX2 = 21
  posY1 = 421.2
  posY2 = 231.2
}
```

Listing 4.1 – Coupling with electric motor defined in the DSL.

```
logic movement

hardware pneumatic axistranslation2 {
  posZ1 = extend
  posZ2 = retract
}
```

Listing 4.2 – Coupling with pneumatic cylinder defined in the DSL.

Once that the grammar of the language is defined, Xtext provides an editor for the custom DSL with syntax highlighting and auto-completion. The code of the grammar definition for the DSL is listed in Appendix A. For this project, only a translation between one type of logic (a linear movement in one axis or rotation around one axis) and two types of hardware (an electric motor and a pneumatic cylinder) have been considered, as they represent all the different possible reconfiguration cases in the prototype available for tests during the thesis work. Therefore, the developed grammar only takes into account these cases. Examples of this language are depicted in Figures 4.1 and 4.2. As seen in Figure 4.1, the user has to first specify the type of logic and hardware to be coupled. First, the user can type the keyword *logic* followed by the type of logic, which for the grammar designed can only be *movement* meaning a linear movement in one axis or a rotation around one axis. Then the user has to introduce the keyword *hardware* followed by the type of hardware used, which for this project can be *electric*, meaning that an electric motor is the hardware used, or *pneumatic* as in Figure 4.2, meaning that a pneumatic cylinder is used. Following the type of hardware, the user has to introduce the name of the FB type that will perform the desired coupling or translation. Depending on the type of hardware selected, the grammar is defined to request different type of information to the user. If the *electric* type is selected, then the user shall introduce the symbolic name of the position used in the logic part of the application, followed by the = symbol and the corresponding coordinate needed by the motor controller, which must be an integer or a real number. The user can introduce as many associations between symbolic names and coordinates as needed. On the other hand, if the type of hardware used is *pneumatic*, the user shall introduce the symbolic name for the position towards which the pneumatic cylinder has to extend, followed by the = symbol and the keyword *extend*. The same has to be done for the position towards which the cylinder should retract (the order does not matter, the user can also define the retract action first).

4.4 Code Generation

The main goal of introducing a DSL to specify the minimum information required for the translation is to save the control application designer from manually coding the translation FB each time that a physical reconfiguration in the plant happens (while the control logic stays the same). Therefore, the information introduced in this DSL has to be translated into code to create the necessary FB.

The Xtext framework provides code generation capabilities. The code generator for the created DSL is a program written in the Xtend language. Xtend is a modernized Java, as defined by its authors [24]. Using Xtend, a code generator has been implemented for the defined DSL grammar. This means that, depending on the user input in the defined DSL, a FB will be automatically generated to couple or glue the selected logic and hardware. Since the implementation is carried out for IEC 61499, the generated FB will be a BFB according to the IEC 61499 standard, and will only have one input and one output (an adapter socket to connect with the logic part of the application and an adapter plug to connect with the hardware-dependant part of the application).

Basically, what has to be coded to define a FB in IEC 61499 is the ECC. A code written in the DSL such as that listed in Figure 4.1 will generate a FB with the ECC depicted in Figure 4.5. In this translation FB for an electric motor, the only thing in the ECC code which is dependent on the DSL input is the algorithm `stringTOlreal`. During code generation, the corresponding algorithm will be created depending on the information provided by the user (which symbolic name is equal to which coordinate). If working with a pneuamtic cylinder, A code written in the DSL such as that listed in Figure 4.2 will generate a FB with the ECC depicted in Figure 4.7. In this case, the only parameters in the ECC that are dependent on the user input are the symbolic names that are checked for transitions, which in this example are the names *posZ1* and *posZ2*.

It is clear that for such automatic generation of FBs to work, the types of logic and hardware interfaces have to be, in some way, standardized. The reason for this is that the adapter interfaces have to be known for the code generation program. In this project, the adapter interfaces used are depicted in Figures 4.3, 4.4 and 4.6.

Once that the code generator is done, everytime that a file that is created with the DSL extension is saved, a folder called *translation* is created under the same root with the generated FBs. The grammar and code generator are programmed in such away that they allow the user to define multiple FBs coupled with the same type of logic at once, and a single FB will be generated per hardware definition. The Xtend code for the code generator is listed in Appendix B.

4.5 Integration in Eclipse 4diac

Both the Xtext framework and the 4diac IDE are integrated into the Eclipse platform. This allows to generate a plugin out of a Xtext project created in an Eclipse version with the Xtext SDK installed that can then be installed into the Eclipse 4diac IDE, allowing us to create a DSL file inside a project in 4diac (under *Type Library*).

Once that the plugin is installed in Eclipse 4diac, a file created with the same file extension as the one defined for our DSL in Xtext will automatically generate the required coupling or translation FBs on save.

4.6 Case Study: Initial Setup

The prototype used for the implementation of the suggested design methodology is the same as in Chapter 3. Therefore, a hierarchical mechatronic component-based decomposition as the one presented in Figure 3.2 is still valid. The resistor sorter architecture was well defined in Figure 3.1, and a real image of the actual prototype is depicted in Figure 4.9. As in Chapter 3, first the case were the vertical axis of the manipualtor is driven by an electric motor is considered.

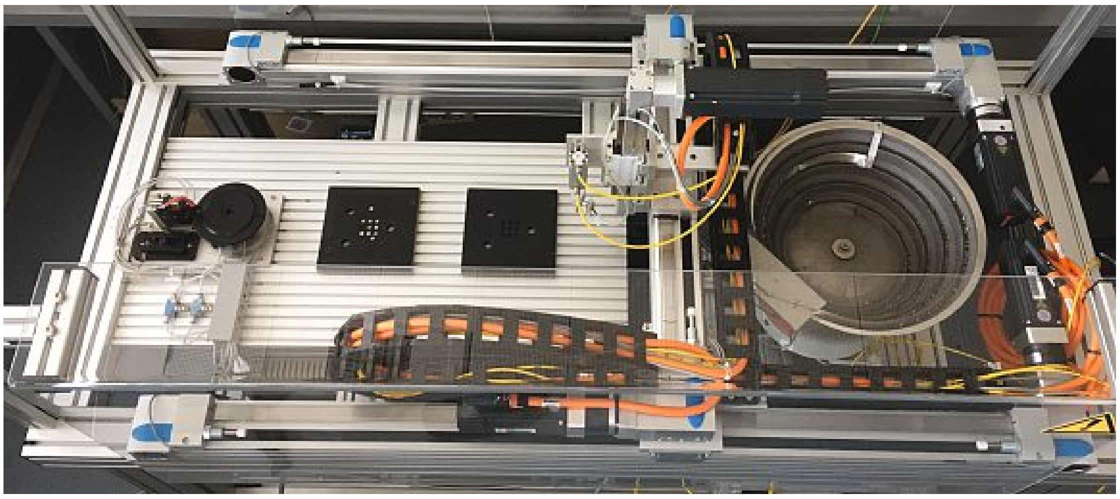


Figure 4.9 – View of the resistor sorter prototype.

For the actual hierarchical composition in Eclipse 4diac, see Figure 4.10. As in Figure 3.3, the yellow component represents a layer 3 FB, orange components represent layer 2 FBs, blue components represent layer 1 FBs and green components represent layer 0 SIFBs. Moreover, some components could be grouped into SubApps as it was shown in Chapter 3. However, the hierarchical structuring in 4diac (see Figure 4.10) is going to be different than that in Chapter 3 (see Figure 3.3). Applying all the guidelines from Chapter 3 and the previous sections in this chapter, the control application will have two completely independent parts, as seen on Figure 4.10 (note that in this figure the intermediate layer with the necessary FB to couple both parts is not represented). With respect to the approach in Chapter 3, some changes that were made in order to emphasize the hardware-dependant components can already be seen:

- Since the hardware technology for the measuring clamps and the turning table is in fact, the same, both components can use the same FB type. These two mechatronic elements are actuated by double-acting pneumatic cylinders, controlled by 2-solenoid 5/3 valves, and have two position sensors each (see Figure 4.11 for a schematic view of the type of cylinder and valve used). Therefore, both mechatronic elements will be now represented by the same FB type in the control application: `DA_53Valve_2Solenoid`.

- The FB block type that was previously called `AxisPTP` in Chapter 3 is now called `Servomotor`, to emphasize that it encapsulates a controller for a servomotor and therefore it is a completely hardware-dependant component.

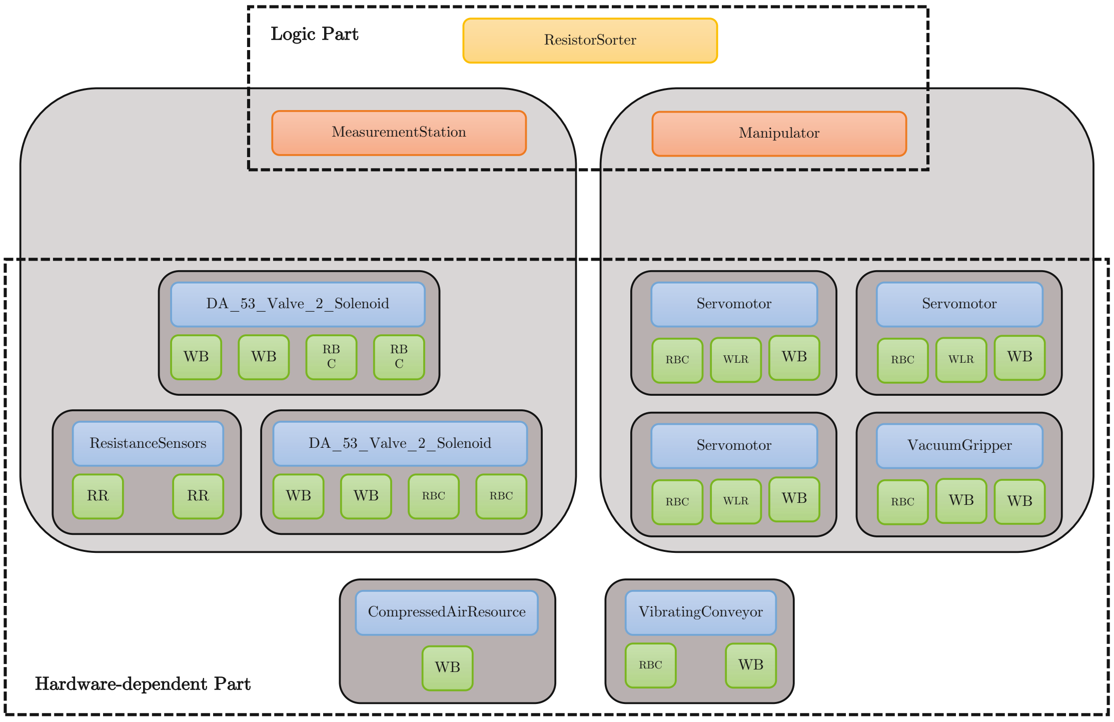
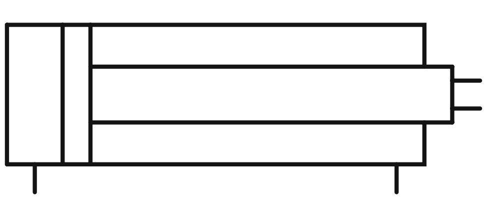
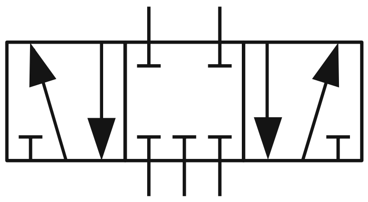


Figure 4.10 – Hierarchical structuring of the control application in 4Diac.



(a) A double-acting cylinder.



(b) A 5/3 pneumatic valve.

Figure 4.11 – Pneumatic hardware used for the vertical axis of the manipulator, the measuring clamps and the turning table.

4.6.1 Logic Part

The logic part of the control application for the resistor sorter includes the layer 2 and the layer 3 components. These components, represented as IEC 61499 FBs are: `ResistorSorter` (layer 3), `MeasurementStation` (layer 2) and `Manipulator` (layer 2).

4.6.1.1 Manipulator

This FB implements the same control logic that was defined Section 3.1.2.2. However, in this new approach the FB `Manipulator`, both its interface and ECC, have been modified (the interface is depicted in Figure 4.12). Now the `Manipulator` FB is completely hardware-independent. In order to achieve this independence from hardware, all the positions towards which the manipulator is commanded to move are represented by symbolic names as a *string* data type.

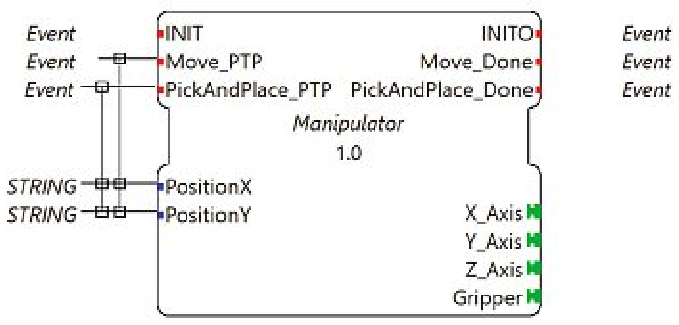


Figure 4.12 – Interface of the `Manipulator` FB.

The communication with lower level components is represented by the plugs `X_Axis`, `Y_Axis`, `Z_Axis` and `Gripper`. The first three plugs are of type `AAxisLogic`, whose interface was depicted in Figure 4.3. The communication between these three plugs and the three hardware-dependent components' sockets is not direct. A translation in between will be necessary via three translation/coupling FBs, as will be detailed in Section 5.1. For the last plug, the adapter type is `AGripper` and no translation will be necessary with the hardware-dependant component in layer 1 (the gripper controller).

4.6.1.2 Measurement Station

Again, this FB implements the same control logic that was defined Section 3.1.2.1. However, in this new approach the FB `MeasurementStation`, both its interface and ECC, have been modified (the interface is depicted in Figure 4.13), in order to decouple its control logic from the hardware used to implement it. The strategy is the same as with the `Manipulator` FB, all the positioning-related commands are represented as symbolic names, or more precisely, as a *string* data type. Positioning is needed for both the measuring clamps and the turning table control.

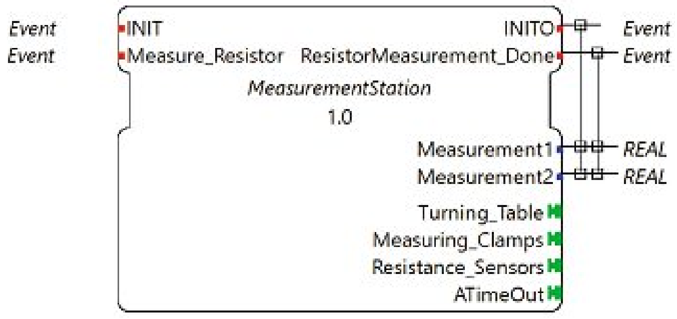


Figure 4.13 – Interface of the `MeasurementStation` FB.

The communication with lower components is represented by plugs, two of which are of type `AAxisLogic`, since the logic for the measuring clamp is a linear movement along the Z axis and the turning table's logic is a rotation around the Z axis.

4.6.1.3 Resistor Sorter

This FB is the top component in the hierarchy, the main coordinator. The logic that this `ResistorSorter` FB implements is the same as in Section 3.1.3. However, again the interface and ECC had to be modified. Previously, the component stored the coordinates of all the positions and passed them down in the hierarchy to other components. Since this obviously coupled the component with the hardware used, now all the positions are symbolic names stored as *strings*. The new FB interface is depicted in Figure 4.14. The ECC of this FB is shown in Figure 4.15. The goal of showing the ECC is to give an example of how the logic of this FB was decoupled from the hardware by using symbolic names. Therefore, there is no detailed explanation of the ECC, which is in fact very simple and it lacks functionalities that would be required on a real industrial application like fault protection.

To give an example about how the decoupling from hardware was implemented in this FB, see Figure 4.16, which contains the algorithm `passPosition` associated to the state `Pass` in the ECC. As seen, the positioning of each storage spot in a tray is not attached to coordinates in this FB, which would already couple the application to the type of hardware used. By using symbolic names like "passX1" the FB is kept hardware-independent and can therefore be reused in this control application no matter what kind of hardware is used for positioning, or even if

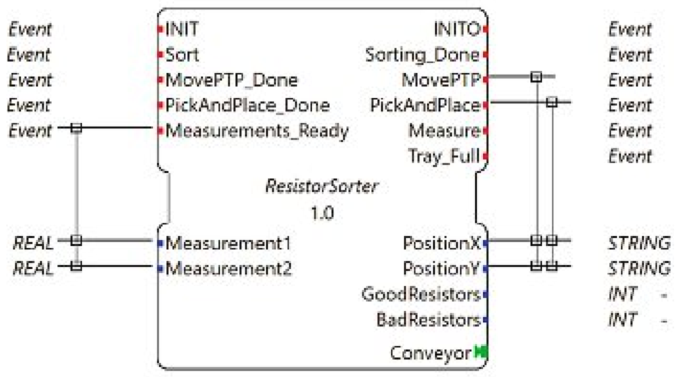


Figure 4.14 – Interface of the `ResistorSorter` FB.

the configuration of the plant changes and the coordinates of each element are no longer the same.

4.6.2 Hardware-Dependent Part

The hardware-dependent part of the control application for the resistor sorter includes the layer 0 and the layer 1 components. In total, there are six different FB types for the layer 1: `DA_53Valve_2Solenoid`, `Servomotor`, `ResistanceSensors`, `VacuumGripper`, `CompressedAirResource`, and `VibratingConveyor`. Of these six FB types, the last four are exactly the same as in Section 3.1.1 and therefore are not explained here again. The SIFB types are also the same as the ones used for the experiment in Chapter 3.

4.6.2.1 Double Acting Cylinder

The turning table and the measuring clamps in the measuring station are both driven by a double acting cylinder. This type of pneumatic actuator is controlled by a 5/3 valve with two solenoids in the prototype. For this reason, in this section it has been decided to use a single FB type for both mechatronic components, since the underlying hardware is the same. The ECC of this FB is essentially the same as in the FBs `ResistanceMeasuringClamps` and `TurningTable` of Section 3.1.1. Only the interface nomenclature and the adapter type for the socket, which is now of type `ACylinder` as in Figure 4.6, have been changed. In Figure 4.17 the SubApp for the turning table control is depicted, containing the `DA_53Valve_2Solenoid` type FB and its corresponding layer 0 SIFBs.

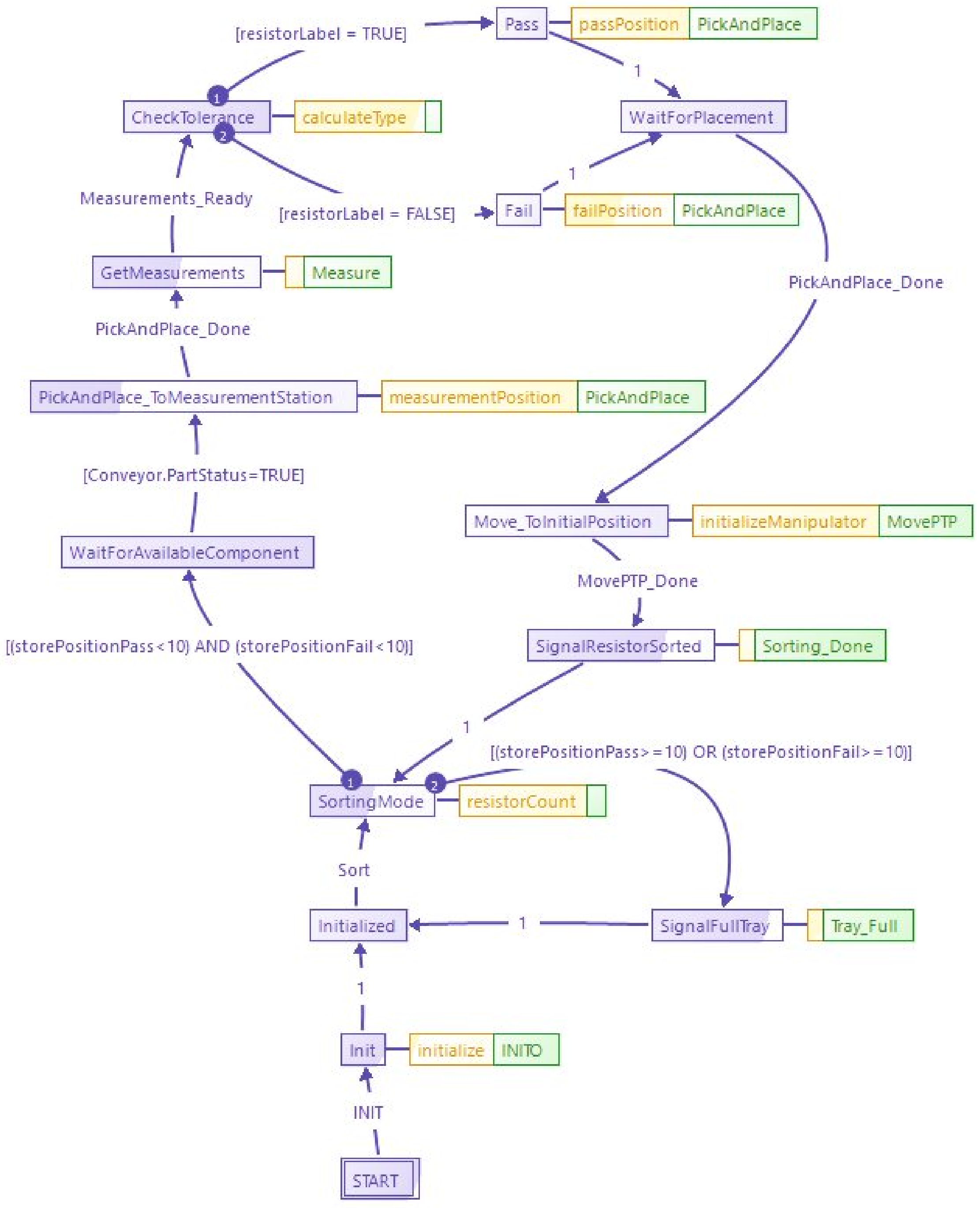
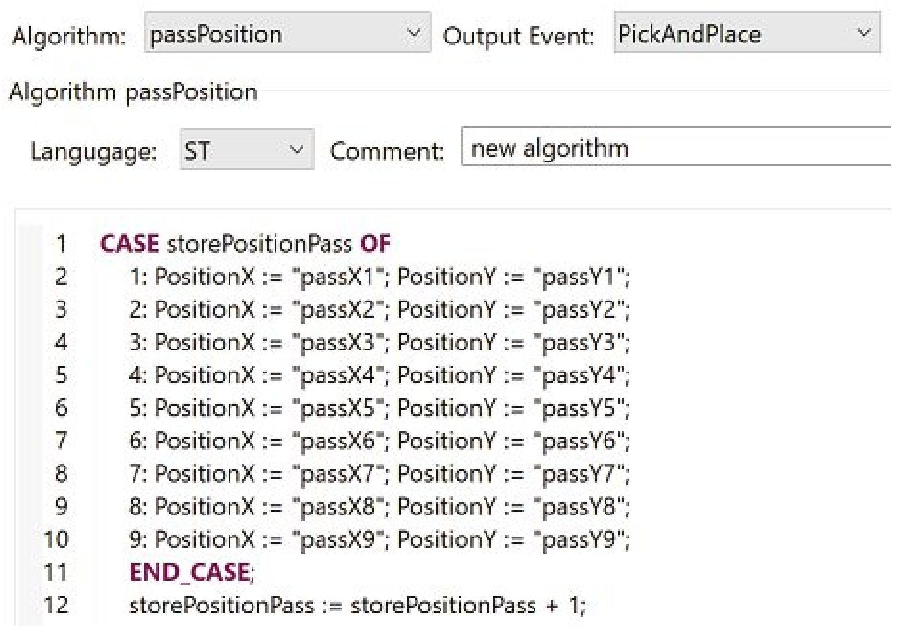


Figure 4.15 – ECC of the `ResistorSorter` FB, control flow from bottom to top.



Algorithm: passPosition Output Event: PickAndPlace

Algorithm passPosition

Langugage: ST Comment: new algorithm

```
CASE storePositionPass OF
   1: PositionX := "passX1"; PositionY := "passY1";
   2: PositionX := "passX2"; PositionY := "passY2";
   3: PositionX := "passX3"; PositionY := "passY3";
   4: PositionX := "passX4"; PositionY := "passY4";
   5: PositionX := "passX5"; PositionY := "passY5";
   6: PositionX := "passX6"; PositionY := "passY6";
   7: PositionX := "passX7"; PositionY := "passY7";
   8: PositionX := "passX8"; PositionY := "passY8";
   9: PositionX := "passX9"; PositionY := "passY9";
   END_CASE;
   storePositionPass := storePositionPass + 1;
```

Figure 4.16 – Algorithm `passPosition` of the `ResistorSorter` FB.

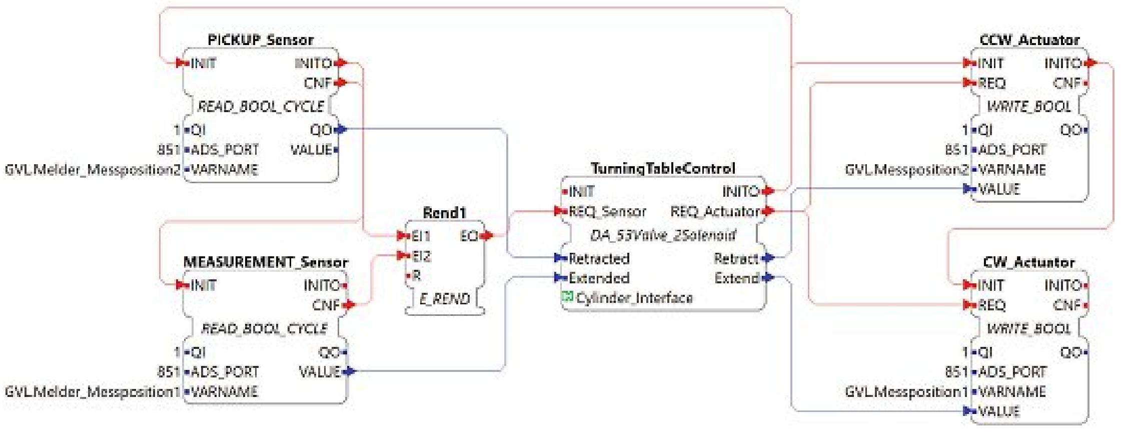


Figure 4.17 – SubApp containing the `DA_53Valve_2Solenoid` FB.

4.6.2.2 Servomotor

The `Servomotor` FB encapsulates a servomotor controller. It can be thought of as a "black box" that provides an interface for the rest of the control application through its socket `Controller_Interface`, of type `AMotor`, which was presented in Figure 4.4. In this FB only the interface nomenclature has been changed with respect to the FB `AxisPTP` in Section 3.1.1, in order to emphasize that this component is hardware-dependent and can be reused for the same hardware equipment.

Table 4.2 – List of components of the logic part.

Component	Layer	FB Type
Sorting Station	3	`ResistorSorter`
Measurement Station	2	`MeasurementStation`
Manipulator	2	`Manipulator`

In this case, all the three manipulator axes are driven by electric motors and therefore all of them require a `Servomotor` FB. In Figure 4.18 the SubApp for the control of the X axis is depicted.

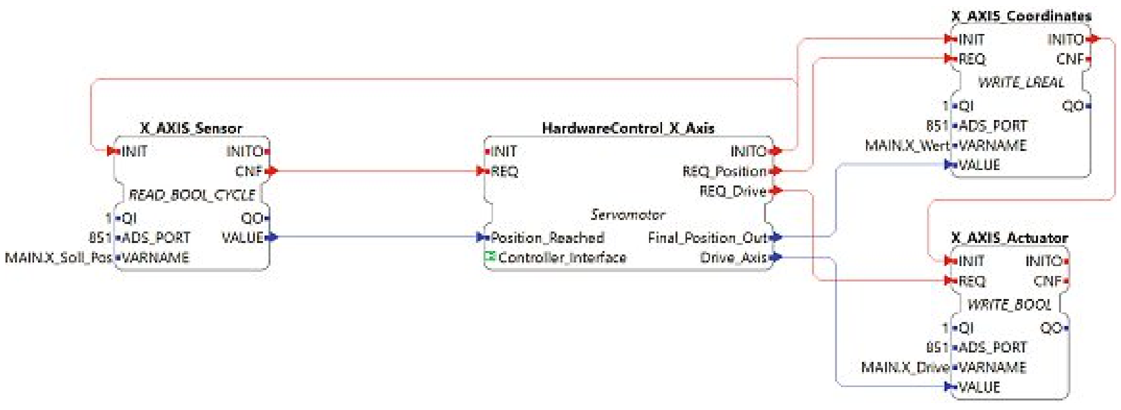


Figure 4.18 – SubApp containing the `Servomotor` FB.

4.6.3 Coupling

Once that the logic part of the control application is designed and tested, it can be coupled to the hardware-dependent part. The connection between these two independent application parts will be defined in the DSL that was created for this project.

4.6.3.1 The Logic Part

The components that belong to this part of the application are listed in Table 4.2 (the notation used for the components is the same as in Figure 3.2). The complete logic part of the control application in 4diac is depicted in Figure 4.19.

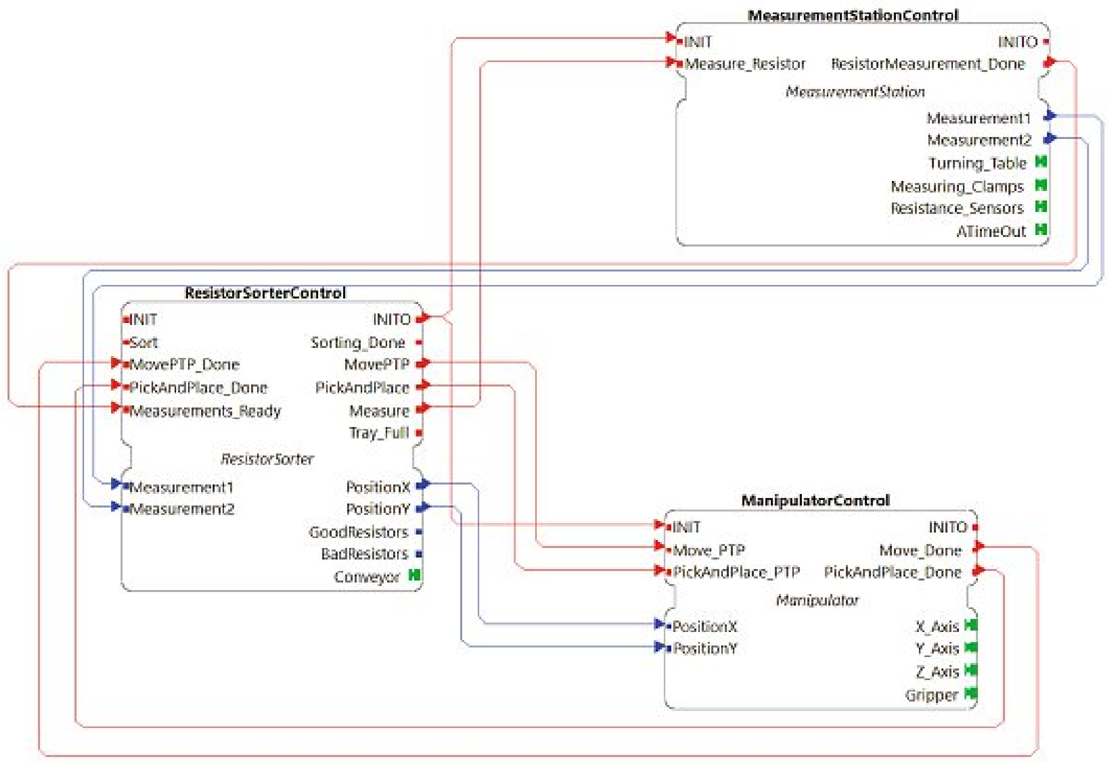


Figure 4.19 – The logic part of the control application in 4diac, with the two components from the layer 2 on the right and the main coordinator component in layer 3 on the left.

4.6.3.2 The Hardware-Dependent Part

In Table 4.3 all the components that belong to the hardware-dependent part of the application are listed (the notation used for the components is the same as in Figure 3.2). The table also provides information about the component which is right above in the hierarchy and whether or not a translation is needed in order to couple these components to the logic part of the application.

4.6.3.3 Coupling with the DSL

The last step of the control application is to couple the hardware-dependent and logic parts. As Table 4.3 reflects, for some components no intermediate layer is needed in order to couple them with the logic part. On the other hand, for other components this new intermediate layer is necessary in order to perform the required translation between logic and hardware-specific parameters. This translation is going to be performed in Eclipse 4diac through the DSL defined in

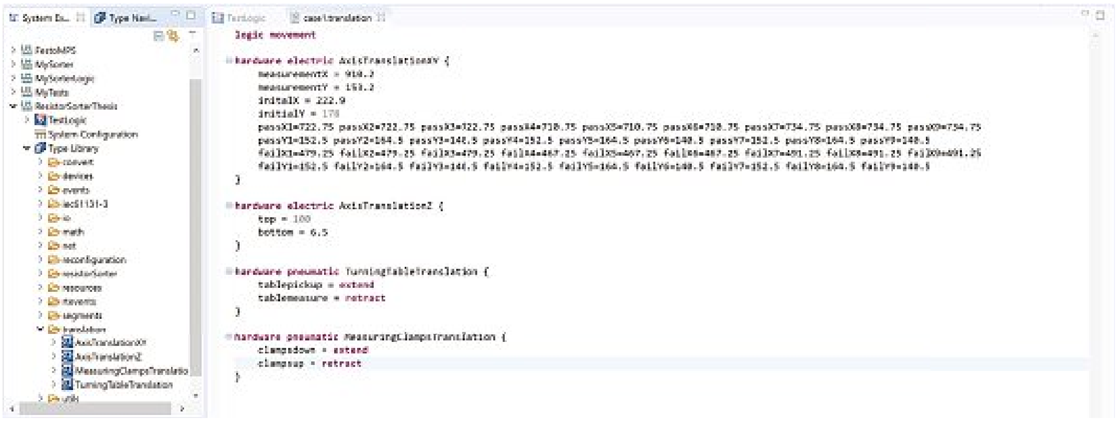
Table 4.3 – List of components of the hardware part.

Component	**FB Type**	**Component above**	**Translation needed?**
Resistance Meters	`ResistanceSensors`	Measurement Station	No
Measuring Clamps	`DA_53Valve_2Solenoid`	Measurement Station	Yes
Turning Table	`DA_53Valve_2Solenoid`	Measurement Station	Yes
X Axis	`Servomotor`	Manipulator	Yes
Y Axis	`Servomotor`	Manipulator	Yes
Z Axis	`Servomotor`	Manipulator	Yes
Gripper	`VacuumGripper`	Manipulator	No
Compressed Air	`CompressedAirResource`	Sorting Station	No
Vibrating Conveyor	`VibratingConveyor`	Sorting Station	No

Section 4.3.

As the extension for this DSL was chosen to be ".translation", a file created in 4diac with this extension allows the user to type the necessary information for each translation. In Figure 4.20, the necessary translations for this specific control application are represented. After saving the ".translation" file, four FB types are automatically generated under the folder "translation": `AxisTranslationXY` (used for the components *X Axis* and *Y Axis*), `AxisTranslationZ` (used for the component *Z Axis*), `TurningTableTranslation` (used for the component *Turning Table*) and `MeasuringClampsTranslation` (used for the component *Measuring Clamps*). As mentioned in Section 4.1, the interface of the FB generated for the coupling is very simple (see Figure 4.21 for example).

Once that the necessary FBs have been generated, the connection between the logic and hardware parts when a translation is necessary is very straightforward. An example of this connection is depicted in Figure 4.22, where the layer 2 FB `Manipulator` could be encapsulated in a SubApp along with its lower level components in the hierarchy. In Figure 4.22 the new intermediate layer is very clear, made up of three FBs (one per manipulator axis) since these are the three components that require a translation between symbolic names in the logic part of the application and hardware-specific parameters in the hardware-dependent



```
logic movement

hardware electric AxisTranslationXY {
    measurementX = 910.2
    measurementY = 153.2
    initalX = 222.9
    initialY = 178
    passX1=722.75 passX2=722.75 passX3=722.75 passX4=710.75 passX5=710.75 passX6=710.75 passX7=734.75 passX8=734.75 passX9=734.75
    passY1=152.5 passY2=164.5 passY3=140.5 passY4=152.5 passY5=164.5 passY6=140.5 passY7=152.5 passY8=164.5 passY9=140.5
    failX1=479.25 failX2=479.25 failX3=479.25 failX4=467.25 failX5=467.25 failX6=467.25 failX7=491.25 failX8=491.25 failX9=491.25
    failY1=152.5 failY2=164.5 failY3=140.5 failY4=152.5 failY5=164.5 failY6=140.5 failY7=152.5 failY8=164.5 failY9=140.5
}

hardware electric AxisTranslationZ {
    top = 100
    bottom = 6.5
}

hardware pneumatic TurningTableTranslation {
    tablepickup = extend
    tablemeasure = retract
}

hardware pneumatic MeasuringClampsTranslation {
    clampsdown = extend
    clampsup = retract
}
```

Figure 4.20 – Defining the translation with the DSL in 4diac. After saving the file, four coupling FBs are automatically generated under the folder *translation*.

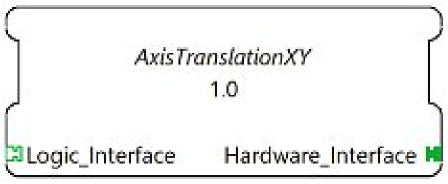


Figure 4.21 – Interface of a translation FB in 4diac.

part of the application. For the *Gripper* component no translation is necessary and it corresponding SubApp can be connected straight to the `Manipulator` FB through the corresponding adapter. In Figure 4.23, the connection between the `MeasurementStation` and its subcomponents is shown.

The whole control application, with the logic and hardware-dependant parts coupled together is depicted in Figure 4.24. From left to right, we have the layer 3, layer 2, coupling intermediate layer and the layer 1 components (encapsulated in a SubApp with their respective layer 0 SIFBs).

4.7 Case Study: Change in the Setup

As in Section 3.2, the vertical axis actuator in the manipulator is going to be replaced with a double-acting pneumatic cylinder. With the new design approach, no changes are needed now in the components `ResistorSorter` and `Manipulator`, since they are now decoupled from the hardware implementation.

In the hardware-dependent part, the change only affects the *Z axis* component. The new SubApp for this component is depicted in Figure 4.25. The new layer 1 FB needed is `DA_53Valve_2Solenoid_NS`. This FB is almost the same as `DA_-`

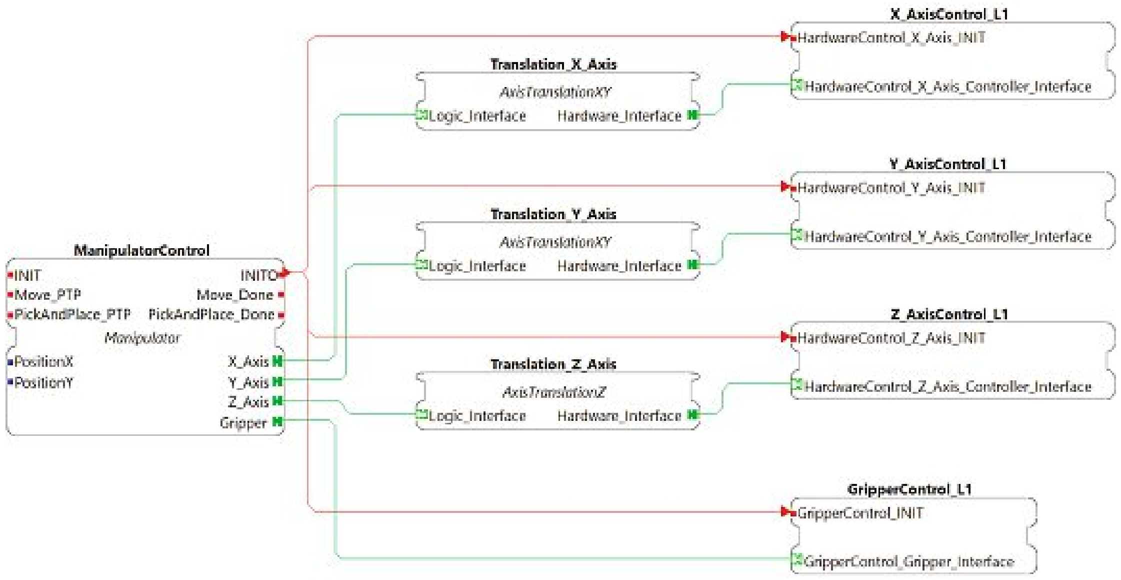


Figure 4.22 – SubApp containing the *Manipulator* component and its subcomponents.

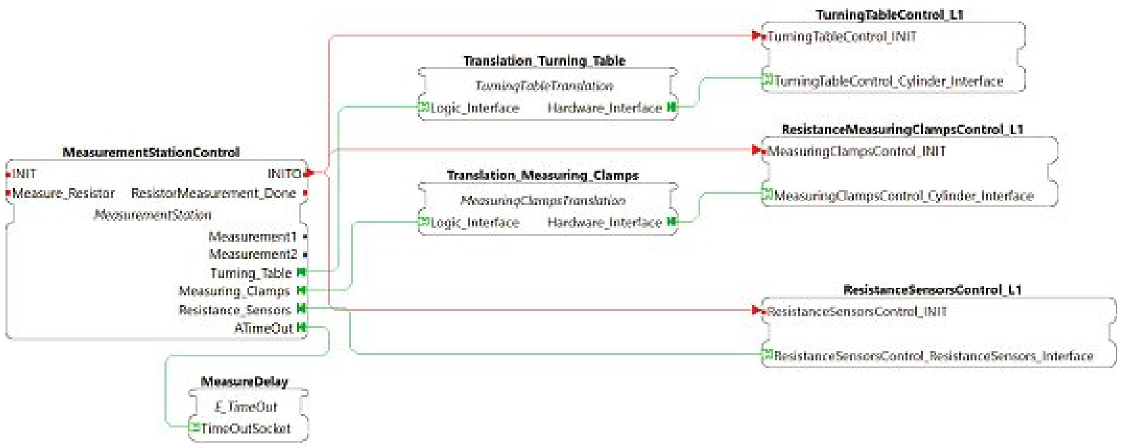

TurningTableControl_L1
TurningTableControl_INIT
TurningTableControl_Cylinder_Interface
Translation_Turning_Table
TurningTableTranslation
Logic_Interface
Hardware_Interface
ResistanceMeasuringClampsControl_L1
MeasuringClampsControl_INIT
MeasuringClampsControl_Cylinder_Interface
MeasurementStationControl
INIT
INITO
Measure_Resistor
ResistorMeasurement_Done
MeasurementStation
Translation_Measuring_Clamps
MeasuringClampsTranslation
Logic_Interface
Hardware_Interface
Measurement1
Measurement2
Turning_Table
Measuring_Clamps
Resistance_Sensors
ATimeOut
ResistanceSensorsControl_L1
ResistanceSensorsControl_INIT
ResistanceSensorsControl_ResistanceSensors_Interface
MeasureDelay
E_TimeOut
TimeOutSocket

Figure 4.23 – SubApp containing the *Measurement Station* component and its subcomponents.

`53Valve_2Solenoid`. The only difference is that this one controls a pneumatic cylinder without position sensors, and therefore uses a delay to estimate when the cylinder has reached the desired position.

Since the hardware used for the Z axis has changed, a new translation is needed. Following the same procedure as in Figure 4.20, the code listed in Figure 4.3 is used in order to generate the FB `AxisTranslationZPneumatic`. This is the

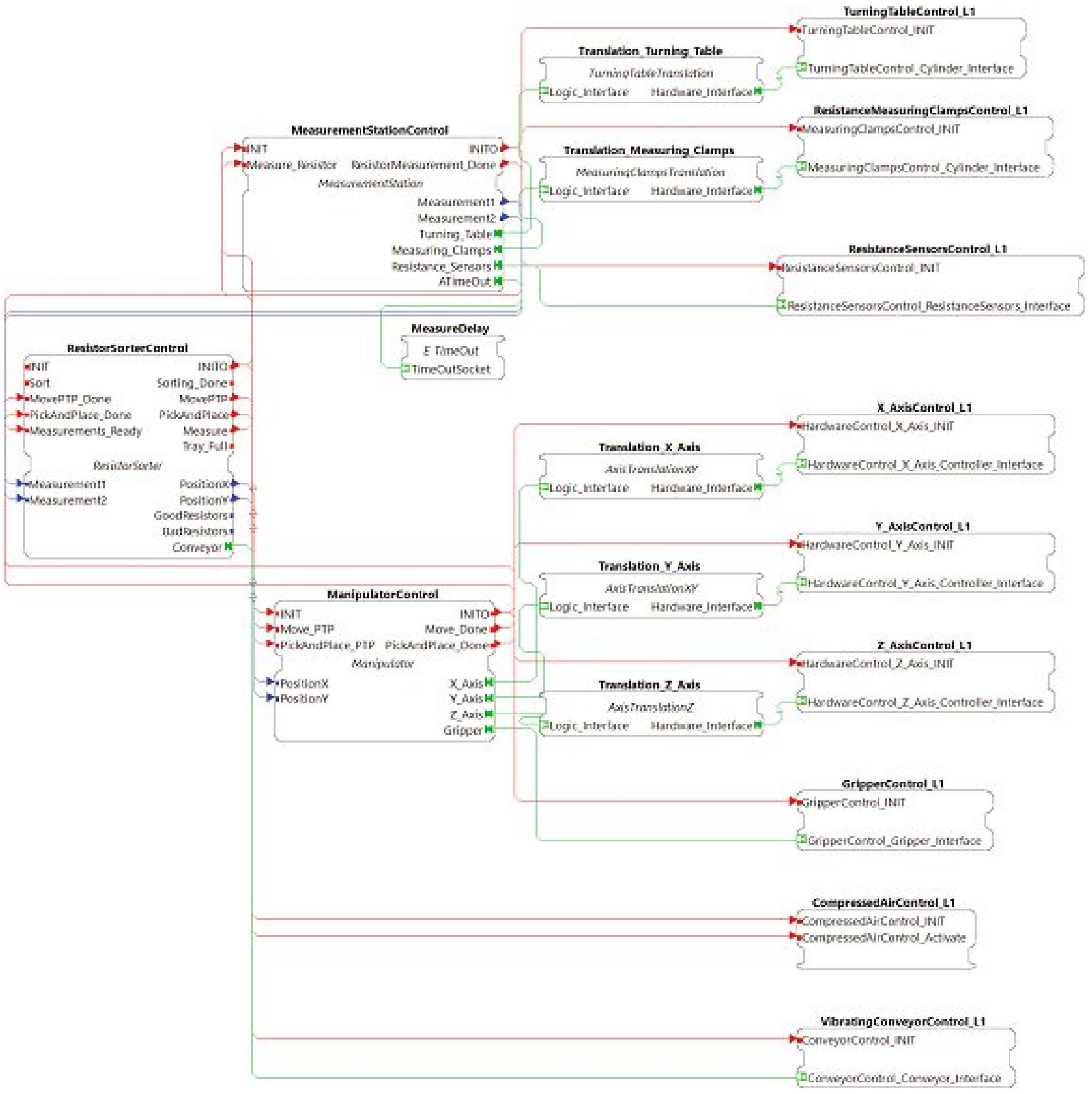


Figure 4.24 – View of the complete control application. From left to right: layer 3, layer 2, coupling layer and layer 1.

FB that will be used for this translation instead of the FB `AxisTranslationZ` used previously, when the Z Axis was driven by a servomotor.

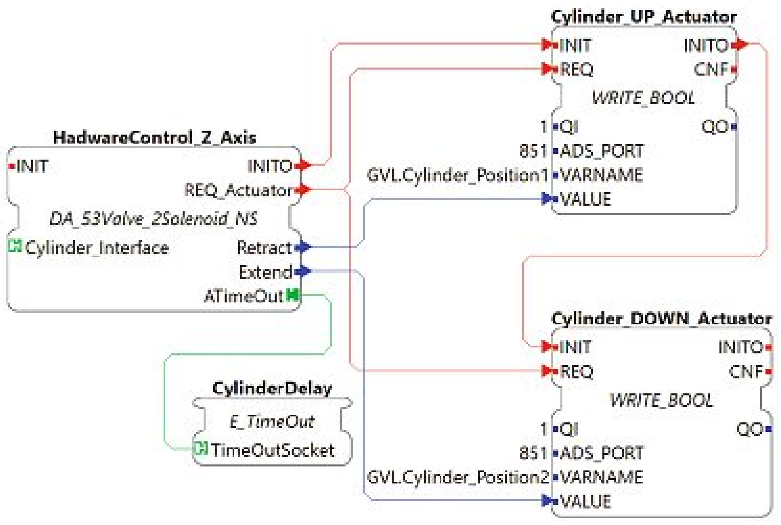


Figure 4.25 – SubApp containing the `DA_53Valve_2Solenoid_NS` FB.

```
logic movement

hardware pneumatic AxisTranslationZPneumatic {
  top = retract
  bottom = extend
}
```

Listing 4.3 – Translation for the new hardware in the Z axis defined in the DSL.

CHAPTER 5

Results

During Chapter 3 and 4 a case study has been presented. A resistor sorter prototype was used in order to test two different control application design approaches implemented in IEC 61499. The goal of this case study was to understand how component reuse works with two different design approaches. In Chapter 3 a state of the art approach was tested, while in Chapter 4 a new design approach which was presented is Section 3.4 was implemented. In this chapter, some results about the new concept presented in Section 3.4 are going to be extracted from the case study.

5.1 Coupling and Component Reuse

In OOP a key concept is coupling. Coupling can be defined as the extend to which the various subcomponents interact. If they are highly interdependent then changes to one are likely to have significant effects on the behaviour of others [25]. CBSE has an important focus on reusing software components. Therefore, it is interesting to give a quantitative measurement about coupling, since reusability and maintainability are the advantages of low coupling [26]. In this section, both a qualitative and a quantitative comparison (with static measurements) are going to be presented between the two different design approaches that were applied to the resistor sorter, in order to see up to which extend component reuse has been improved.

5.1.1 Qualitative Comparison

The state of the art approaches for hierarchical and component-based designs in industrial automation proposed a hierarchical structure where components in layer 1 are hardware-dependent and components in the layers above make up the logic part of the application. An approach of this type produces reusable components according to the authors of the publications reviewed in Chapter 2. During the case study in Chapter 3, it was shown how the current state of the art approaches do not guarantee component reuse on the logic part of the control application, since these components still contain hardware-specific information that couple them to the hardware implementation.

The case study continued in Chapter 4, where the new proposed approach was implement on the resistor sorter. The results of this implementation achieved a lower coupling between the logic and hardware-dependent parts of the application. With this new approach, the components in layer 2 (*Manipulator* and *Measurement Station*) and layer 3 (*Resistor Sorter*) are free of hardware-specific parameters for positioning. This allows to reuse these components independently of what is the hardware type chosen to implement the required movements for the X, Y and Z axis of the manipulator, as well as for the turning table and measuring clamps in the measurement station. This does not imply that there is no coupling at all between the logic and hardware-dependent parts of the application. However, the coupling with hardware has been moved to a single point in the new intermediate "coupling" or "translation" layer, enabling a higher degree of reusability in the higher layers of the hierarchy, which are now less tightly coupled to the hardware implementation. This coupling point is easily reconfigurable according to the selected hardware. Since positioning via a manipulator or other mechatronic devices is highly present in production lines, the presented approach in this thesis could be very useful for achieving modular and flexible control applications in this field.

At this point the Research Question 2 can be answered. By using symbolic names for position-related parameters in components which are part of the high levels of the component hierarchy, these components can be decoupled from the hardware implementation and then coupled to the hardware-specific components with the addition of a new layer in the hierarchy that performs the necessary translations between the two independent parts of the application. This allows higher-level components to be reused in cases where there is a hardware reconfiguration, or cases where there is a physical change in the layout meaning that the coordinates of certain positions have changed.

5.1.2 Quantitative Comparison

Authors in [27] propose a way to measure the coupling level for FBs in IEC 61499 based on the *fan-out* concept, which for a FB A is the number of FBs whose inputs are connected to the outputs of FB A. The higher the *fan-out*, the higher the coupling level. However, this quantitative measurement does not take into account the type of connection between two FBs. Since this thesis focuses in decoupling higher level components from the hardware implementation, a metric that weights in some way the type of parameters that are being send from one FB to another one gives a better understanding on how coupled the logic side of the application is to the hardware implementation. For this reason the Fenton and Melton [28] coupling metric has been selected, since it distinguishes between different coupling types. Felton and Melton define the the coupling between two components x and y as:

$$C(x, y) = i + \frac{n}{n+1}, \tag{5.1}$$

where n is the number of interconnections between x and y and i is the level of the worst coupling type found between x and y and its based on the Myers classification [29], as shown in Table 5.1.

In our case study, the number of interconnections n between one component x in layer 2 and one component y in layer 1 is the total number of IO parameters of the adapter interface that defines the communication between the two components.

The worst coupling level or type i between components in layer 3 and 2, and between components in layers 2 and below is $i = 5$ or *content* type for the design approach in Section 3.1 and Section 3.2. This is because the components on the logic part of the application (*Manipulator*, *Measurement Station* and *Resistor Sorter*) internally have hardware-specific parameters (coordinates, for example) that are being passed down in the hierarchy. If we analyze the connection between the component for the *X Axis* and the *Manipulator* component, the *X Axis* component branches into the *Manipulator* component via the coordinates, which ties the *Manipulator* component content to the type of content the *X Axis* component requires (coordinates if it is an electric motor controller).

When implementing the new design approach in Section 4.6 and Section 4.7 the coupling level between components in the logic part of the application is reduced to $i = 2$ or *stamp* type. In the new approach, what is being passed down the hierarchy are symbolic names that represent positions. Now the *Manipulator* component is not directly connected to the *X Axis* component, but to the translation component in the new intermediate layer instead. A coupling level $i = 1$ or *data coupling* for this approach is too low, since passing these symbolic position names creates some sort of interdependency between the modules, because it is

Table 5.1 – Fenton and Melton modified definition for Myers coupling levels.

Coupling Type	**Coupling Level**	**Definition**
Content	5	Component x refers to the inside of y, i.e., it branches into, changes data, or alters a statement in y.
Common	4	Components x and y refer to the same global data.
Control	3	Component x passes a parameter to y with the intention of controlling its behavior, i.e., the parameter is a flag.
Stamp	2	Components x and y accept the same record type as a parameter. This type of coupling may manufacture an interdependency between otherwise unrelated modules.
Data	1	Components x and y communicate by parameters, each one being either a single data element or a homogeneous set of data items that do not incorporate any control element.
No Coupling	0	Components x and y have no communication, i.e., are totally independent.

implicit that these modules are related together by implementing a positioning system. However, a *control* type ($i = 3$) is too high, since position names do not really affect control behaviour. Control behaviour parameters in a positioning application could include proportional–integral–derivative (PID) parameters, for example. Coupling type of $i = 4$ is not possible in IEC 61499 since this standard forbids the use of global data. And in the new approach the hardware components do not branch into the logic part of the application, they do not share the same content so a coupling type of $i = 5$ or *content* type can be easily disregarded for this case.

In Table 5.2 the coupling values according to the Felton and Melton metrics are listed, choosing the *Manipulator* component as component x and three of its subcomponents below (one per axis) as components y, in order to compare the different approaches from the case study. Table 5.2a and 5.2b showcase the coupling for the state of the art approach, for the initial setup with the electric motor driving the Z axis and for the setup change with the pneumatic cylinder driving the vertical axis, respectively. Table 5.2c shows the coupling values for the new approach, where the *Manipulator* component is not connected straight to the hardware, but to the translation or coupling FBs. In this case the FB `Manipulator`

Table 5.2 – Coupling between the *Manipulator* component (component x) and lower-level components using the Felton and Melton metrics.

(a) Old approach, electric motor.

Component y	i	n	$C(x,y)$
X Axis	3	5	5.75
Y Axis	3	5	5.75
Z Axis (M)	3	5	5.75

(b) Old approach, cylinder.

Component y	i	n	$C(x,y)$
X Axis	3	5	5.75
Y Axis	3	5	5.75
Z Axis (C)	4	5	5.8

(c) New design approach.

Component y	i	n	$C(x,y)$
X Axis Translation	3	2	2.75
Y Axis Translation	3	2	2.75
Z Axis Translation	3	2	2.75

does not depend on the hardware so the number of connections n is the same for every axis no matter the hardware used to drive the Z axis, since all the adapter interfaces used for every axis in the manipulator are of type `AAxisLogic`.

5.2 Complexity

The approach presented in Section 3.4 adds a new layer in the component hierarchy and consequently, more complexity in the application design. Therefore, in this section the complexity or cost of this new layer is going to be evaluated. Also, the complexity of the DSL is going to be measured in order to justify its use in this design approach.

5.2.1 Coupling Layer Complexity: Manual vs DSL Implementation

In this section, a comparison between manually creating a coupling FB and automatically generating these FBs out of a DSL definition is presented. The goal is to show that the introduction of a DSL reduces the complexity when creating coupling FBs.

5.2.1.1 Qualitative Comparison

A new intermediate layer with new components means more design complexity, since more FBs have to be created. The idea behind the use of a DSL and code generation is to make the implementation of these FBs more agile and simple for the application designer. Instead of manually creating or editing one FB that performs some sort of translation in the coupling layer, the DSL allows the user to just specify the minimum information required for the coupling in a very simple and high level language. For example, the manual and DSL implementations can be compared for the FB in Figure 4.21. This FB translated the positions coming from the manipulator as symbolic names into the coordinates for the servomotor controller. Taking just the first four positions (from a total of forty translations made in this FB) we can see the difference in complexity between the two approaches. In Figure 5.1, the DSL code necessary to create the whole FB is listed. On the other hand, if the FB is created manually, the ECC in Figure 4.5 has to be created and the code of the algorithm `stringTOlreal` is listed in Figure 5.2.

```
logic movement

hardware electric axistranslation1 {
  measurementX = 910.2
  measurementY = 153.5
  initialX = 222.9
  initialY = 178
}
```

Listing 5.1 – Implementation of a translation FB with the DSL.

```
IF Logic_Interface.Position = "measurementX" THEN
  Hardware_Interface.Final_Position := 910.2;
ELSIF Logic_Interface.Position = "measurementY" THEN
  Hardware_Interface.Final_Position := 153.5;
ELSIF Logic_Interface.Position = "initialX" THEN
  Hardware_Interface.Final_Position := 222.9;
ELSIF Logic_Interface.Position = "initialY"  THEN
  Hardware_Interface.Final_Position := 178;
END_IF;
```

Listing 5.2 – Manual implementation of a translation FB.

It can be argued that user-wise it is more simple in case of a reconfiguration in the plant to generate the new coupling FBs via the DSL than to create the FBs manually, saving time during reconfiguration.

5.2.1.2 Quantitative Comparison

In order to make a quantitative comparison between a manual implementation of the translation FBs and their definition through a DSL, the Halstead metrics have been chosen. The reason to choose these metrics is that there is a methodology for applying Halstead metrics to IEC 61499 FBs in [27]. Therefore, a fair comparison can be made between a FB coded in IEC 61499 and a textual program coded in the developed DSL.

With Halstead's metrics a static code analysis can be derived by taking into account the number of distinct operators n_1 and operands n_2, as well as the total number of operators N_1 and operands N_2. From these parameters, several complexity measures can be derived:

- Program length: $N = N_1 + N_2$
- Program vocabulary: $n = n_1 + n_2$
- Estimated length: $\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- Purity ratio: $PR = \frac{\hat{N}}{N}$
- Program volume: $V = N \log_2 n$
- Difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
- Program effort: $E = D \times V$

For the complexity comparison, two FBs have been created in Eclipse 4diac, one for a hardware of type electric and the other one for hardware of type pneumatic. Each FB has been created in two different ways: first manually and then through the DSL. For the FB which performs the coupling with an electric motor controller, only fours translations have been implemented, as in Figure 5.1. In the DSL, keywords *logic* and *hardware*, the hardware type and the "=" symbol have all been considered as operators. The rest of the elements in the DSL code are considered as operands. The complexity results are listed in Table 5.3. From the results listed it can be observed that both the difficulty D and the effort E are lower when the FBs are generated from the code written in the DSL. The difficulty measure is related to the difficulty of the program to write or understand, while the program effort measure translates into the actual coding time.

Table 5.3 – Halstead complexity measures for two different ways of creating the coupling FBs: manual vs DSL implementation.

HW/Implementation	N	n	$\hat{N}$	PR	V	D	E
Electric/Manual	35	21	72.95	2.08	153.73	5	768.66
Electric/DSL	18	15	44.04	2.45	70.32	3	210.97
Pneumatic/Manual	16	13	35.61	2.23	59.21	6.4	378.93
Pneumatic/DSL	12	11	27.12	2.26	41.51	3	124.54

Table 5.4 – Halstead complexity measures of the two design approaches implemented on the resistor sorter. For the approach proposed in this thesis, two cases are considered: manually implementing the coupling layer and defining the coupling layer with the DSL.

Approach	N	n	$\hat{N}$	PR	V	D	E
State of the art	297	484	2236.49	4.62	3975.73	32.67	129873.86
Proposed approach	627	1043	5525	5.30	9691.89	38.28	371033.89
Proposed approach w/DSL	366	616	2908.86	4.72	5245.67	29.52	154871.40

5.2.2 Overall Application Complexity

The approach presented in Section 3.4 introduces a new layer in the component hierarchy. This means more FBs in the control application and hence more code. In this section a quantitative comparison is going to be presented between the two design approaches implemented on the resistor sorter, using Halstead metrics. Two designs are going to be compared: the whole control application designed following the state of the art guidelines, as depicted in Figure 3.13, and the whole control application designed following the proposed methodology in Section 3.4, as depicted in Figure 4.24. For both approaches, the case where an electric motor is used to drive the Z axis is going to be compared. The results of the Halstead metrics for these two approaches are listed in Table 5.4.

As expected, the new layer increases the difficulty and program effort. However, a third measurement was taken, removing the complexity of the translation FBs and substituting it by the complexity of the DSL required to generate these FBs, since from the programmer perspective what matters is the code in the DSL,

Table 5.5 – Reconfiguration effort, measured in LOC.

Approach	*LOC* Difference
state of the art	7
Proposed approach	4

which will automatically generate the FBs. This third case is labelled as "Proposed approach w/DSL" on the table. It can be seen that when using the DSL to generate the coupling or translation FBs not only the program effort increase with respect to the state of the art approach is very small, but the program difficulty is smaller than that of the state of the art approach. This lower program difficulty can be linked with the higher modularity of the proposed approach, which facilitates code maintainability and readability.

5.3 Reconfiguration Effort

In Section 3.1 and Section 3.2 the control application for the resistor sorter following the state of the art approach was presented, first by using an electric motor to actuate the vertical axis of the manipulator and then by using a double acting pneumatic cylinder instead. In Section 4.6 and Section 4.7 the same experiment was performed on the resistor sorter, but in this case following the design approach presented in Section 3.4.

A quantitative measurement of the reconfiguration effort that this hardware change introduced in the control application can be presented in the form of lines of code (LOC). The number of lines of code that had to be written or deleted give an idea of the effort that the programmer needed in order to adapt the control application for the new hardware. In [27] a methodology to count the LOC in IEC 61499 was presented. The results are presented in Table 5.5.

For both approaches, it has been considered that the component that implements the hardware control in layer 1 is retrieved from a component library and therefore the LOC change when replacing the FB for an electric motor controller with a FB for a double acting pneumatic cylinder controller has not been taken into account. For the state of the art approach changes were needed in the *Manipulator* component in layer 2 as well as the *Resistor Sorter* component in layer 3. The LOC for these components have been calculated following the guidelines in [27] for IEC 61499 FBs, and are listed in Table 5.6. For the proposed approach, the change in the setup from Section 4.6 to Section 4.7 only required to define a new translation in the DSL, which was depicted in Figure 4.3. Blank lines and indentation have been omitted for the LOC count.

Table 5.6 – LOC of different components in the state of the art approach (total number of LOC per component).

Component	$LOC(ALG)$	$LOC(ECC)$	$LOC(FB)$
Manipulator (Electric)	5	45	50
Manipulator (Pneumatic)	2	45	47
Resistor Sorter (Electric)	39	30	69
Resistor Sorter (Pneumatic)	35	30	65

As depicted on Table 5.5 The difference in LOC is smaller for the proposed approach. It has to be mentioned that this measurement does not fully represent the difference in reconfiguration effort. First of all, because the LOC metrics for IEC 61499 FBs proposed in [27] do not take into account the changes on FB interfaces, which are changes that the programmer would also have to make and would add extra effort in the state of the art approach. Second, the difference in LOC between using an electric motor or a pneumatic cylinder do not reflect the LOC that had to be modified, only the number of LOC that were added or removed. If not only the lines that do not disappear or have to be added are taken into consideration, but also those that have to be anyways modified are counted, the total LOC change for the state of the art approach would also increase. And last, in the state of the art approach two FBs have to be reconfigured, while on the proposed approach only one FB is reconfigured. Moreover, reconfiguring FBs in the logic part of the application as in the state of the art approach adds an extra difficulty, since the programmer has to make an effort in order to understand the code and identify the LOC that have to be modified. In contrast, on the proposed approach only a new configuration has to be defined with the DSL.

CHAPTER 6

Conclusions

In the last years there has been an interest in the industrial automation world in new design approaches that enhance code reusabilty. Looking into the software engineering field, multiple authors have proposed component-based designs. The goal of these proposed approaches was to create reusable automation components, and many authors suggested to encapsulate hardware-control operations in the bottom layer components, while the higher levels of the component hierarchy include only components that encapsulate logic control operations. Since the reviewed works do not explain how these higher level components can be hardware-independent, and the use cases that they presented did not show how changes in hardware components affect the rest of the component hierarchy, the initial hypothesis was that the state of the art approaches still have some limitations regarding component reuse, leading to the Research Question 1:

Research Question 1

What are the limitations of the current hierarchical and component-based control design approaches in industrial automation?

The results of the case study in Section 3.1, Section 3.2, and Section 3.3 identified the following limitations in the state of the art approaches: the type of hardware used still couples higher levels of the component hierarchy with the hardware implementation, and the state of the art approaches do not propose any measures to avoid this. Therefore, when the hardware type is changed (for example, an electric motor is replaced by a pneumatic actuator) the components in higher layers of the hierarchy can not be reused, even if the control logic is the same.

In order to overcome the state of the art limitations, logic and hardware have to be further decoupled. With this goal in mind, a new design approach has been proposed in Section 3.4. In Chapter 4, this new design approach was implemented in IEC 61499 and tested on the same prototype (Section 4.6 and Section 4.7) as the state of the art approach.

Comparing the two different design approaches implemented on the same platform over the same prototype delivered some promising results in Chapter 5. The proposed design approach provides a looser coupling between the hardware implementation and the control logic, thus improving component reuse on the higher layers of the control application (Section 5.1). The complexity of the presented approach requires a slightly higher programming effort (i.e. total coding time) but delivers a lower programming difficulty, which translates into better code modularity, readability and maintainability (Section 5.2). Finally, the reconfiguration effort of the proposed approach turned out to be smaller, by reducing the number of total changes in the code and shifting multiple variation points in the control application into a single variation point in the new coupling or translation layer (Section 5.3). From the results in Chapter 5 it can be derived that the proposed design approach outperforms the state of the art approach in terms of component reuse and ease of plant reconfiguration against hardware changes in a sequential control application with multiple positioning-related operations, like the resistor sorter prototype used for the experiments. Thus, in the light of the results obtained, the Research Question 2 can be answered:

Research Question 2

How can component reuse be improved in control applications design for industrial automation?

Component reuse can be improved in industrial automation applications by replacing hardware-dependent parameters in higher level components with symbolic names. These symbolic names are then translated in a new intermediate layer into the necessary parameters that a hardware-specific component needs. With this approach even if the hardware technology in the plant is changed, the higher level components of the control applications can still be reused. When compared to the state of the art approaches, hardware and control logic in the high layers of the component hierarchy have been further decoupled with this methodology .

The great results obtained open the possibility of new research areas in the field of modular and reusable control applications in industrial automation. It would be very interesting to study what is the best way to implement this new design approach in IEC 61131 applications. Also, it would be useful to investigate the possibility of implementing *enum* types into 4diac, in order to improve the translation algorithm in IEC 61499 applications. Finding other use cases for this design approach would be very beneficial in order to collect more data and show the industry the advantages of implementing this new design approach in their control applications. Moreover, while analysing the results in Chapter 5 it was clear that component reuse and reconfiguration effort are currently difficult to measure in industrial automation control applications, due to the lack of specific metrics for this field. Therefore, a new research possibility opens in the metrics field for IEC 61131 and IEC 61499, as to provide better ways to compare different modular design approaches.

APPENDIX A

DSL Grammar Definition

```
grammar org.xtext.CouplingLayer with org.eclipse.xtext.common.Terminals

generate couplingLayer "http://www.xtext.org/CouplingLayer"

Model:
  'logic' name=LOGIC_TYPE

    (elements += Hardware)*;

Hardware:
  'hardware' HardwareType
  ;

HardwareType:
    type='electric' name=ID '{'
        (electricfeats += FeatureElectric)*
        //symbolic_position_name = coordinate (as REAL or INT)
    '}' |
    type='pneumatic' name=ID '{'
        pneumaticfeats += FeaturePneumatic
        pneumaticfeats += FeaturePneumatic
        //symbolic_position_name = "extend" or "retract"
    '}'
;

```

```
FeatureElectric:
  name=ID '=' coordinate=COORDINATE
;

FeaturePneumatic:
  name=ID '=' action=ACTION
;

ACTION:
  'extend' | 'retract'
;

COORDINATE:
  REAL | INT
;

terminal REAL:
  INT '.' (EXT_INT | INT)
;

terminal EXT_INT:
  INT ('e'|'E')('-'|'+') INT
;

LOGIC_TYPE:
  'movement' | 'grip'
;
```

APPENDIX B

DSL Code generation

```
package org.xtext.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.xtext.couplingLayer.HardwareType
import org.xtext.couplingLayer.FeatureElectric
import org.eclipse.emf.ecore.EObject
import org.xtext.couplingLayer.Model

class CouplingLayerGenerator extends AbstractGenerator {

  override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {
     resource.allContents.filter(typeof(HardwareType)).forEach[it.
    generateFunctionBlock(fsa)]
  }

  def void generateFunctionBlock(HardwareType hw, IFileSystemAccess2 fsa
    ) {
     fsa.generateFile(hw.name.toFirstUpper+".fbt", hw.generate)
  }

  def CharSequence generate(HardwareType hw) '''
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd
    ">
<FBType Comment="Tamplate for a simple Basic Function Block Type" Name="
    «hw.name.toFirstUpper»">
  <Identification Standard="61499-2"/>
  <VersionInfo Author="4DIAC-IDE" Date="2018-06-06" Organization="4DIAC-
    Consortium" Version="0.0"/>
  <VersionInfo Author="AZ" Date="2016-05-26" Organization="fortiss GmbH"
      Version="1.0"/>
  <InterfaceList>
    <EventInputs/>
    <EventOutputs/>
    <InputVars/>
    <OutputVars/>
    <Plugs>
      <AdapterDeclaration Comment="" Name="Hardware_Interface" Type="«IF
    hw.type=='electric'»AMotor"/>«ELSEIF
    hw.type=='pneumatic'»ACylinder"/>«ENDIF»
    </Plugs>
    <Sockets>
      <AdapterDeclaration Comment="" Name="Logic_Interface"«IF
    hw.modelOf.name=='movement'» Type="AAxisLogic"/>«ENDIF»
    </Sockets>
  </InterfaceList>
  «IF hw.type=='electric'»
  <BasicFB>
    <ECC>
      <ECState Comment="Initial State" Name="START" x="475.0" y
    ="1125.0"/>
      <ECState Comment="" Name="Translate" x="1205.0" y="750.0">
        <ECAction Algorithm="stringTOlreal" Output="Hardware_Interface.
    GoTo_Position"/>
      </ECState>
      <ECState Comment="" Name="Done" x="1200.0" y="1400.0">
        <ECAction Output="Logic_Interface.PositionReached"/>
      </ECState>
      <ECTransition Comment="" Condition="Logic_Interface.ToPosition"
    Destination="Translate" Source="START" x="815.0" y="965.0"/>
      <ECTransition Comment="" Condition="1" Destination="START" Source
```

```
="Done" x="965.0" y="1345.0"/>
      <ECTransition Comment="" Condition="Hardware_Interface.
Position_Reached" Destination="Done" Source="Translate" x="1435.0" y
="1140.0"/>
    </ECC>
    <Algorithm Comment="new algorithm" Name="stringTOlreal">
        <ST Text="«FOR elfeat:hw.getElectricfeats()»«IF
hw.getElectricfeats().indexOf(elfeat)==0»IF Logic_Interface.Position
= &quot;«hw.getElectricfeats().get(0).name»&quot; THEN
Hardware_Interface.Final_Position :=
«hw.getElectricfeats().get(0).coordinate»;
«ELSE»«elfeat.nameToCoordinate»«ENDIF»«ENDFOR» END_IF;"/>
    </Algorithm>
  </BasicFB>
</FBType>
«ELSEIF hw.type=='pneumatic'»
  <BasicFB>
    <ECC>
      <ECState Comment="Initial State" Name="START" x="475.0" y
="1125.0"/>
      <ECState Comment="" Name="Extend" x="500.0" y="600.0">
        <ECAction Output="Hardware_Interface.Extend"/>
      </ECState>
      <ECState Comment="" Name="ExtDone" x="500.0" y="200.0">
        <ECAction Output="Logic_Interface.PositionReached"/>
      </ECState>
      <ECState Comment="" Name="Retract" x="500.0" y="1600.0">
        <ECAction Output="Hardware_Interface.Retract"/>
      </ECState>
      <ECState Comment="" Name="RetDone" x="500.0" y="2000.0">
        <ECAction Output="Logic_Interface.PositionReached"/>
      </ECState>
      <ECTransition Comment="" Condition="Logic_Interface.ToPosition[
Logic_Interface.Position=&quot;«IF hw.getPneumatic
feats().get(0).action=="extend"»«hw.getPneumaticfeats().get(0)
.name»«ELSE»«hw.getPneumaticfeats().get(1).name»«ENDIF»&quot;]"
Destination="Extend" Source="START" x="585.0" y="900.0"/>
      <ECTransition Comment="" Condition="Hardware_Interface.Extended"
Destination="ExtDone" Source="Extend" x="645.0" y="465.0"/>
      <ECTransition Comment="" Condition="1" Destination="START" Source
    ="ExtDone" x="-135.0" y="655.0"/>
      <ECTransition Comment="" Condition="Logic_Interface.ToPosition[
    Logic_Interface.Position=&quot;«IF hw.getPneumatic
    feats().get(1).action=="retract"»«hw.getPneumaticfeats().get(1)
    .name»«ELSE»«hw.getPneumaticfeats().get(0).name»«ENDIF»&quot;]"
    Destination="Retract" Source="START" x="630.0" y="1405.0"/>
      <ECTransition Comment="" Condition="Hardware_Interface.Retracted"
    Destination="RetDone" Source="Retract" x="595.0" y="1800.0"/>
      <ECTransition Comment="" Condition="1" Destination="START" Source
    ="RetDone" x="-200.0" y="1760.0"/>
      <ECTransition Comment="Not part of translation. Signal to the
    upper component that the movement is done if we were already at the
    requested position." Condition="Hardware_Interface.Retracted"
    Destination="RetDone" Source="START" x="2940.0" y="1575.0"/>
      <ECTransition Comment="Not part of translation. Signal to the
    upper component that the movement is done if we were already at the
    requested position." Condition="Hardware_Interface.Extended"
    Destination="ExtDone" Source="START" x="2980.0" y="635.0"/>
    </ECC>
  </BasicFB>
</FBType>
«ENDIF»
    '''

   def nameToCoordinate(FeatureElectric electricfeat)'''
   ELSIF Logic_Interface.Position = &quot;«electricfeat.name»&quot; THEN
     Hardware_Interface.Final_Position := «electricfeat.coordinate»;
   '''

}
```

Bibliography

[1] R. Lewis, *Programming Industrial Control Systems Using IEC 1131-3 (Control, Robotics and Sensors)*. The Institution of Engineering and Technology, 1998.

[2] H. A. Elmaraghy, “Flexible and reconfigurable manufacturing systems paradigms”, *International Journal of Flexible Manufacturing Systems*, vol. 17, no. 4, pp. 261–276, Oct. 2005.

[3] M. Wenger, M. Melik-Merkumians, I. Hegny, R. Hametner, and A. Zoitl, “Utilizing IEC 61499 in an MDA control application development approach”, in *2011 IEEE International Conference on Automation Science and Engineering (CASE)*, IEEE, Aug. 2011.

[4] A. Hirzle, AutomaionML^{TM}, Press Conference, Hannover Messe – HMI, DaimlerChrysler AG, 2007.

[5] M. Buchwitz, “Neue wege in der software entwicklung”, *SPS Magazin*, 2012.

[6] A. Zoitl and R. Lewis, *Modelling Control Systems Using IEC 61499*, 2nd ed. The Institution of Engineering and Technology, 2014.

[7] W. Dai and V. Vyatkin, “A component-based design pattern for improving reusability of automation programs”, in *39th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, IEEE, Nov. 2013.

[8] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[9] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd. Addison-Wesley Longman Publishing Co., Inc., 2002.

[10] K. Thramboulidis, “IEC 61499 vs. 61131: A comparison based on misperceptions”, *Journal of Software Engineering and Applications*, vol. 6, no. 8, pp. 405–415, Jan. 2013.

[11] E. Faldella, A. Paoli, M. Sartini, and A. Tilli, “Hierarchical control architectures in industrial automation: a design approach based on the generalized actuator concept”, in *17th International Federation of Automatic Control World Congress*, IFAC, vol. 41, Jul. 2008, pp. 69–76.

[12] A. Tilli, A. Paoli, M. Sartini, C. Bonivento, and D. Guidi, “Hierarchical and cooperative approaches to logic control design in industrial automation”, in *2009 IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, IEEE, Sep. 2009.

[13] M. Sartini, “Architectures and design patterns for functional design of logic control and diagnostics in industrial automation.”, PhD thesis, University of Bologna, 2010.

[14] M. Melik-Merkumians, M. Wenger, R. Hametner, and A. Zoitl, “Increasing portability and reuseability of distributed control programs by i/o access abstraction”, in *15th Conference on Emerging Technologies Factory Automation (ETFA)*, IEEE, Sep. 2010.

[15] I. Hegny, T. Strasser, M. Melik-Merkumians, M. Wenger, and A. Zoitl, “Towards an increased reusability of distributed control applications modeled in IEC 61499”, in *2012 IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, IEEE, Sep. 2012.

[16] I. Hegny and A. Zoitl, “Component-based simulation framework for production systems”, in *IEEE International Conference on Industrial Technology (ICIT)*, IEEE, Mar. 2010.

[17] W. Eisenmenger, J. Meßmer, M. Wenger, and A. Zoitl, “Increasing control application reusability through generic device configuration model”, in *2011 IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, Sep. 2017.

[18] A. Zoitl and H. Prähofer, “Guidelines and patterns for building hierarchical automation solutions in the IEC 61499 modeling language”, *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2387–2396, Nov. 2013.

[19] A. Zoitl and T. Strasser, *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*. CRC Press, 2016.

[20] *Eclipse 4diac*, `https://www.eclipse.org/4diac/index.php`, Accessed: 10-04-2018.

[21] *16.070 Introduction to Computers and Programming, MIT*, `http://web.mit.edu/16.070/www/lecture/big_o.pdf`, Accessed: 06-08-2018.

[22] *Xtext - language engineering made easy*, `https://www.eclipse.org/Xtext/`, Accessed: 08-08-2018.

[23] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, 2nd. Packt Publishing, 2016.

[24] *Xtend - modernized java*, `https://www.eclipse.org/xtend/`, Accessed: 08-08-2018.

[25] G. Gui and P. D. Scott, "Measuring software component reusability by coupling and cohesion metrics", *Journal of Computers*, vol. 4, Sep. 2009.

[26] R. S. Pressman, *Software Engineering - A Practitioner's Approach*, 4th. Mcgraw-Hill Companies, 1997.

[27] G. Zhabelova and V. Vyatkin, "Towards software metrics for evaluating quality of iec 61499 automation software", in *IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, IEEE, Sep. 2015.

[28] N. Fenton and A. Melton, "Deriving structurally based software measures", *Journal of Systems and Software*, vol. 12, Jul. 1990.

[29] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, 2nd. PWS Publishing Co., 1997.