Development and Validation of Controllers for Automatic Machines with Object Oriented Approach

RACCHETTI Lorenzo

Department of Engineering
University of Modena and Reggio Emilia

International Doctorate School in
Information and Communication Technologies (i.e. ICT)
XXVIII Doctorate Cycle
Electronics and Telecommunications Curriculum

School Dean: Prof. VITETTA G. M.
Courses Coordinator: Prof. BORGARINO M.
Supervisor: BIAGIOTTI L.
Assistant Supervisor: FANTUZZI C.

February 2016

Declaration

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for any other degree or qualification in this, or any other university, except where specific reference is made to the work of others. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments.

RACCHETTI Lorenzo
February 2016

For my family,
who taught me to work hard;
and for my beloved Francesca,
who taught me to flow forward change.

Acknowledgements

Mundus, mutatio; vita, opinio.
The universe is change; our life is what our thoughts make it.

Marcus Aurelius

I would like to thank my supervisors Luigi Biagiotti, Cesare Fantuzzi, and Lorenzo Tacconi for making it possible for me to work on a research problem that I enjoyed. They also allowed me to operate in such a good environement, and organized my abroad period in such a nice place. Additionally, I would like to thank them and Marcello Bonfé for teaching me so much, and giving me the feedback that shaped the research, the scientific papers, and this dissertation.

Therefore, I would like to thank the entreprises that allowed me to do my doctorate in a business environment, where I met great colleagues.

I would like to thank my colleagues Alessandro, Daniel, Fabio, Felice, Francesco, Manuel, Mirko, Paolo, Roberto, Thomas, and Thorbjorn for your help, the great working and not-so-working time. This work was possible thanks to you all also.

Florian, Giacomo, Marco, Roberto, and Vincent, thank you for teaching me to be open to criticism both from others and myself. Moreover, thank you for the great free time experiences.

I would also like to thank Alessandro, Daniele, Emanuel, Hana, Henrik, Lucia, Serena, Theodor, Valeria, Zelda, and all the people (e.g. colleagues and not, from work and/or university) from Italy to Sweden that shared at least a part of this experience with me. You all are too many to be written here, but I will not forget you.

Finally, I would like to thank my family and Francesca, without whom I do not believe I could flow through this experience.

Related Publications

This thesis contains most the work performed during my PhD. Despite we already made five publications on part of this work [68, 66, 65, 67, 64], the thesis chapters resume and deepen the scientific articles already published by adding new content.

1. **Towards an Abstraction Layer for PLC Programming using Object-Oriented Features of IEC61131-3 applied to Motion Control**
   L. Racchetti, L. Tacconi, M. Bonfe, C. Fantuzzi, Industrial Electronics Society, IECON 2015 - 41st Annual Conference of the IEEE, DOI: 10.1109/IECON.2015.7392115;

2. **Verification and Validation based on the generation of Testing Sequences from Timing Diagram Specifications in Industrial Automation**
   L. Racchetti, L. Tacconi, C. Fantuzzi, Industrial Electronics Society, IECON 2015 - 41st Annual Conference of the IEEE, DOI: 10.1109/IECON.2015.7392529;

3. **Generating Automatically the Documentation from PLC Code by D4T3 to Improve the Usability and Life Cycle Management of Software in Automation**
   L. Racchetti, L. Tacconi, C. Fantuzzi, Conference on Automation Science and Engineering (CASE), 2015, DOI: 10.1109/CoASE.2015.7294057;

4. **The PLC UML State-chart design pattern**
   L. Racchetti, C. Fantuzzi, L. Tacconi, M. Bonfe, Emerging Technology & Factory Automation (ETFA), 2014, DOI: 10.1109/ETFA.2014.7005309;

5. **Hardware in the loop simulation and Machine Modular Development: Concepts and application**
   L. Racchetti, C. Fantuzzi, Emerging Technologies & Factory Automation (ETFA), 2013, DOI: 10.1109/ETFA.2013.6648075.

Abstract

We applied principles of object oriented software engineering in automation to develop methods for the development of software in automation. These methods allowed us to achieve the following: to develop PLC software with control logic independent from hardware platform specificity by an abstraction architecture; to increase reusability by object oriented features; to model reactive behaviors by UML State-charts directly into code; to guarantee software quality and lifecycle management by generating software test sequences for verification and validation from timing diagram specifications; to guarantee certain properties by verifying and validating software through modular Hardware-in-the-Loop; to improve usability and lifecycle management by automatically generating software documentation from source code. We published and presented our methods as 5 papers in 4 international conferences during the PhD period.

The cost and complexity of software development in automation is growing. Such growth creates critical software challenges in automation software development: usability, maintainability, management of lifecycle, and flexibility of software. To meet these challenges, often software engineers introduce new approaches in software development. However, often these approaches met only part of these challenges. For example, an approach that addresses to ease and improve usability of software development has to take into account also the management of documentation. In fact, as far as it is possible to ease software development, its documentation will always be required.

Due to that, we supported 10 PLC developers to learn, and use our methods to develop 11 software libraries which contained more than 1000 entities (e.g. Function Blocks, Structs, Interfaces, etc.). Then, these libraries and their entities were used to realize machine software. Developed software showed a degree of reusability greater than 70% by object oriented features, a greater independence from its automation hardware platform, an improved usability, and improved lifecycle management. Moreover, the software development showed a reduction in development time. The set of our methods allowed us to improve different phases of software development and to achieve 6 releases of our libraries. Moreover, developers' feedback helped us to improve our methods themselves.

We believe that these results give a good indication of the direction to be taken to address today's challenges in automation software development. Moreover, we believe that though these methods, software engineers will improve development efficiency and increase the quality of developed software.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Automation is the control of system through hardware (i.e. mechanical and electronic) and software (e.g. Fig. 1.1). Historically, hardware has had the lion's share in automation. However, due to the progress made and being made in software, this part is moving towards software. This trend is leading to a growing complexity and cost of software development in automation. The German Engineering Federation VDMA presented that the ratio of software development in machinery has doubled in one decade from 20% to 40% [81]. If this trend continues, the main activity of automation suppliers and developers will be software engineering [92].

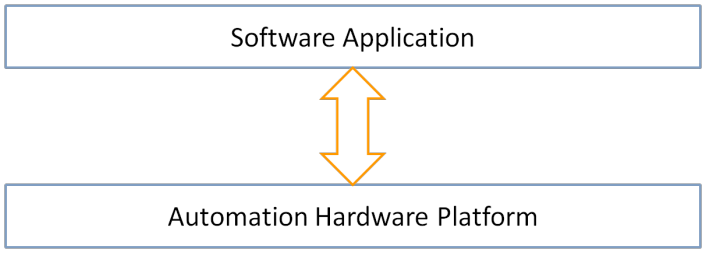

Fig. 1.1 shows the coupling of hardware and software in automation.

This trend is due partially to the role that automation is taking in our life. In fact, Fig. 1.2 shows how our everyday life is interconnected with automation systems. For example, the automobile is a highly automated consumer product produced as well in automated plants. Current automobiles are equipped with several microprocessors that operate a variety of functions, including engine control, the clock, the radio, and cruise control.

The growth in costs and complexity of software combined with the pervasive nature of automation systems in modern society are the conditions that created and still are creating new challenges of software development in automation.

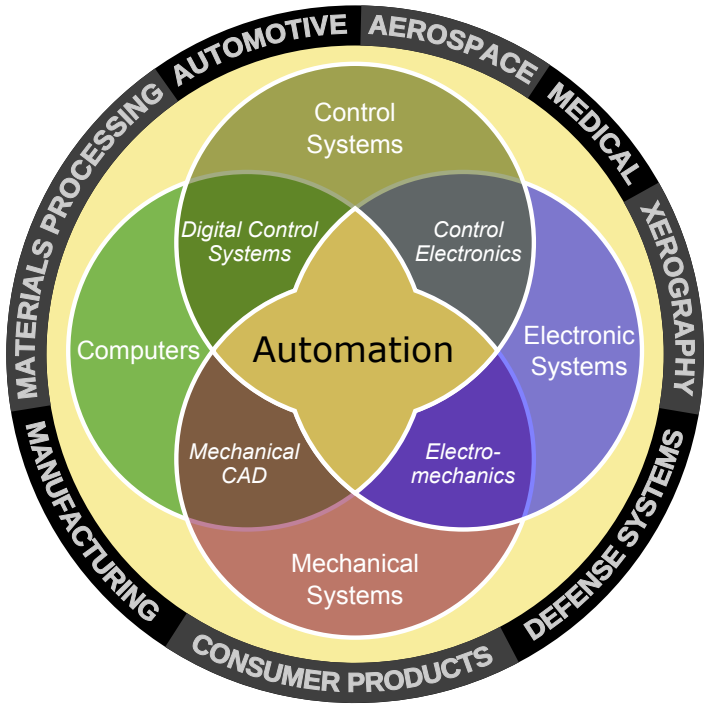

Fig. 1.2 is based on figure of [96] and shows the various fields touched by automation.

1.1 Challenges of software development in automation

Currently, these challenges impacts different phases of software development, and they may be summarized as the following five challenges:

Usability during the systems' development, utilization and maintenance various groups are involved. This requires software engineers to consider all different stakeholders and provide suitable integrated methods and processes to ease the activities of each group;

Reusability software engineering in automation contains several levels of software development. For example, generic basic modules are created independent from the specific projects by computer scientists or engineers of various groups. In order to reduce the development time, these modules should be developed and provided taking into account the reuse;

Maintainability software engineers should ease software maintenance and introduce continuous improvement on reused modules;

Flexibility software should be designed to be easily adaptable when external changes occur. For example, changes on hardware automation configuration should change software as

less as possible in order to reduce the commissioning and start-up time on construction sites;

Life cycle management it is crucial to keep the downtime caused by software changes as low as possible or to completely avoid it. For software products, this means that different versions have to be used and tracked in parallel. Therefore, the compatibility, maintainability, and documentation of software has to be taken into account.

To meet these challenges, many approaches from general software engineering (e.g. object orientation, model driven engineering, etc.) have already been incorporated into software for automation. However, as Vyatkin [92] concludes his summary of the current state of the art in automation software engineering, while many methods/approaches were introduced, the list of open issues is still long. For example, object oriented programming was introduced in automation since the third edition of IEC 61131-3 standard. However, while object oriented programming is massively used in computer science to improve software development (e.g. design patterns, libraries, etc.), it is still underused in automation.

Additionally, in 2014 Vogel-Heuser et al. [89] gave an introduction to the essential challenges of software engineering and requirements that software has to fulfill in automation. They state that current approaches and methods reach their limits by fulfilling part of the challenges, and by not taking into account the whole software development process. Therefore, they conclude that methods and modeling techniques from software engineering and new ones need to be studied and evaluated based on all the current discussed challenges.

1.2 Our aim

Supported by those reasons, during these three years of doctorate we applied principles of software engineering and object oriented programming in automation. With the aim of improving software development in automation, we developed methods to face the challenges of the whole development process.

These methods were published and presented as 5 papers in 4 international conferences during the PhD period. Moreover, together with 10 PLC developers, we used our methods to develop 11 software libraries which contained more than 1000 entities (e.g. Function Blocks, Structs, Interfaces, etc.). Then, we used these libraries and their entities to realize the machine software of a plant module.
As results, developed library entities showed a degree of reusability greater than 70% by object oriented features [35], a greater independence from its automation hardware platform,

an improved usability and lifecycle management. Additionally, the software development time was reduced.

We believe that our methods should allow developers to achieve the following:

- to develop a flexible PLC software by an architectural design pattern;
- to increase maintainability and reusability using object oriented features (i.e. encapsulation, composition, inheritage, delegation, and polymorphism);
- to improve usability and lifecycle management by automatically generating software documentation from source code;
- to model hardware reactive behaviors by UML state-charts implemented through a design pattern;
- to guarantee software quality by generating software test sequences from timing diagram specifications;
- to verify and validate software through modular hardware-in-the-loop;

The set of our methods allowed us to improve different phases and challenges of software development and to achieve 6 releases of our libraries. Furthermore, developers' feedback helped us to improve our methods themselves. We believe that these results give a good indication of the direction to be taken to address today's challenges in automation software development. Finally, we believe that through these methods, software engineers will improve development efficiency and increase the quality of developed software.

1.3 Outlines

This dissertation is organized to deal one of our methods in each chapter, except for the sixth chapter which deals with two methods. Next chapter (i.e. Chap. 2) describes our architectural design pattern known as Abstraction Layer Object-Oriented Architecture (i.e. ALOOA) and its application to Motion Control.

Chap. 3 presents PLC UML State-chart Design Pattern (i.e. PUSDP), which we used to model reactive behaviors in ALOOA and our libraries.

To reduce architecture complexity, automatic generation of documentation from code is described in Chap. 4 as D4T3 (i.e. Doxygen for TwinCAT3).

Testing method for components is introduced in Chap. 5, while chap. 6 introduces the method for system verification & validation by a modular hardware-in-the-loop approach.

Moreover, Chap. 6 puts every method together by presenting also the machine modular development.

Finally, conclusions and perspectives are introduced in the last chapter (i.e. Chap. 7).

Chapter 2

Flexibility through abstraction

Flexibility has different meanings in software. Each one of these meanings may have different goals and ways to achieve them. While each one of them is important, we focused on architectural flexibility. In fact, if a software architecture is described as flexible, usually it refers to the capability of software to adapt to possible and / or future changes. Of course, it is impossible to anticipate all the possible changes. However, when software is designed and developed steps should be taken at least to make it adaptable to future inevitable changes.

For example, often in automation there are several versions of the same machine. Each version may have changes in hardware configuration and consequently, also in software. These different versions may be due from customer personalization to hardware obsolescence. In such example, important decisions about the software architecture must be taken. Design and develop software for each version of the system as far as feasible, it may not be the best solution. Develop a flexible software that adapts easily to change the basic necessities for each version might be a better solution. During my doctorate we had to had the opportunity to handle the last case as one of our real cases with one of our enterprise partners.

In order to achieve such kind of flexiblility there are different ways. In our work we decided to achieve that by decoupling software from hardware, in order to limit software variations on hardware changes.

2.1 Related works

Model-Driven Engineering (i.e. MDE) is a widespread approach that uses domain models instead of pure programming concepts to achieve flexibility. In this approach, the models can be used both as a design support [76], and as a source for the automatic generation of code [83, 98, 99, 26]. The use of model ensures developers a good flexibility from the hardware automation platform. Howver, despite this approach increases also productivity by reusing

standardized models, it is rarely used in practice. In fact, often the code generated by a model is changed manually, making impossible to round-trip it to a model [92, 89].

Component-Based Software Engineering (i.e. CBSE) is an alternative approach which emphasizes the separation of concerns. Thus, its aim is producing independent software components that encapsulate a set of related functions and/or data. Those components are units that can be reused in different software applications without modification. Other authors proposed the use of CBSE for industrial automation systems design. For example, [46] addresses application level issues by modular planning of the software application. Implementation level issues are instead considered in [18, 19, 85] by masking the location components details (e.g. distributed systems). An used-in-practice CBSE approach for implementation-level issues is the use of PLCOpen libraries. PLCopen is an independent organization that aims to meet the specific-platforms-and-their-constraints and maintainability defining the functional and interface requirements for standard application libraries that are reusable for multiple hardware platforms [86]. However, PLCOpen libraries are provided by the different vendors which claim compliance with the PLCOpen specifications. Due to that, even the use of PLCOpen specification does not guarantee interoperability at all as the library is vendor-dependent (e.g. PLC technology, actuator technology, and fieldbus) [80, 11].

Design, develop a flexible architecture is a very complex task which require as well to design, develop and maintain components. These components should be designed for reuse, which is a very complex process that requires not only components with functionality and flexibility, but also a development well organized. CBSE may raise questions on the actual generality and efficiency of components, compatibility problems, maintainability and life-cycle-management [92].

An approach to face those challenges is the use of design patterns, and more specifically architectural design patterns. In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. Faldella et al. [28] proposed an architectural design pattern based to abstract control through the Generalized Device (i.e. GD). The GD is a new software component that abstractly model by means of a standardized FSM, the behavior of several field devices commonly used in automated systems, regardless of their nature, intrinsic features, and specific functional purposes. GD plays the fundamental role of keeping cleanly distinct higher-level control policies from the low-level mechanisms dealing with actuators and sensors. While this approach is promising and some concept could be found also in our work, it can model only a sub-set of field devices with basic boolean feedback and actuation (e.g. pistons, on-off valves). In fact, it can not model devices with more complex feedback and actuation (e.g. a servo-controlled motion axis).

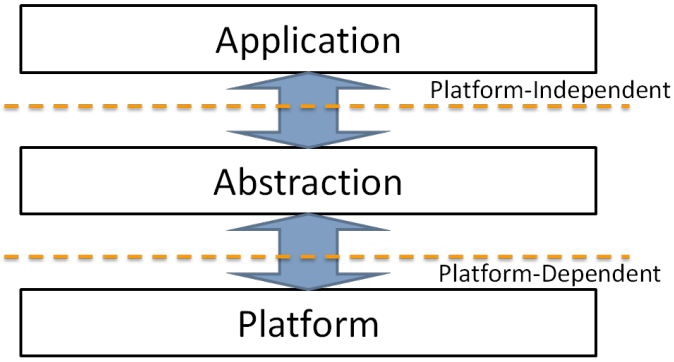

Fig. 2.1 shows a simple vision of an architecture as a series of abstraction layers: platform, abstraction, and application. Platform is the platform from which the application should be independent, while abstraction is the layer that hides the platform details. Application is the software part that should be platform independent.

At the best of our knowledge there is still not an approach that faces the flexibility, maintainability, life-cycle-management, and specific-platforms-and-their-constraints altogether. Due to that, we aim to improve design software in automation to face those challenges through an object oriented design pattern called Abstraction layer Object Oriented (i.e. ALOOA). ALOOA should provide a set of components layers that abstract control from implementation and platform through object oriented features. Moreove, object oriented feature should improve the maintainability of ALOOA components [58].

ALOOA is currently used in our development process to improve software design and development. While we currently used it in a set of libraries (i.e. Base Abstraction, Input Output, Motion, Variable Frequency Drive, Advanced Functions, and Fieldbus), we present only its base concept and its application to motion control. In fact, we consider motion control as the automation issue that most benefits from our proposed approach.

2.2 ALOOA

ALOOA is an object-oriented architectural design pattern. Its proposal is to be a reusable solution to the problem of hiding the implementation details of a particular set of functionality (i.e. Fig.2.1). Moreover, ALOOA also aims to allow the separation of concerns to facilitate interoperability and platform independence. Such features should allow application developers to create flexible and reusable control applications.

To reach these goals, we took into account all of the entities and actors involved in the control software (i.e. CS) of an automated system (i.e. AS) through a top-down approach. Using this approach, we identified the following actors:

Operator uses the human-machine interface (i.e. HMI) to monitor, make corrections, and manually operate on the AS if it is required;

Developers develop the software of the AS;

Assemblers assemble the AS.

Then from the interaction of those actors with AS software, we identified the following entities:

HMI supports the operator with indications about thye current status of the system. It can be developed on the same control platform or with a third-party one (e.g. C#);

Control Software represents the software part of the AS that handles the hardware to perform the finished product;

Hardware is the collections of all the devices that allows the CS to interact with the system and the operator.

Finally, from CS, we identified further entities relying on the relationship with the actors and/or the hardware:

Control Logic (i.e. CL) is the part of the software that control the operations of the system to perform the finished product (e.g. reactive behavior at system level);

Platform Related is the part of the software that performs the transformation needed to read, write and handle the hardware, and also configure it.

Once the entities and actors involved with the AS were identified, we organized the entities in an n-Tier architecture [34] of three levels. The resulting architecture is shown in Fig. 2.3, while the three levels are described as:

Presentation is the top-most level of the AS and it is divided into two parts: HMI and Application layer;

- the HMI uses the HMI Interfaces to manage the AS from the HMI panel. Those interfaces presents information and generic actions (e.g. configure) that are implemented in the FBs of the Abstraction layer.

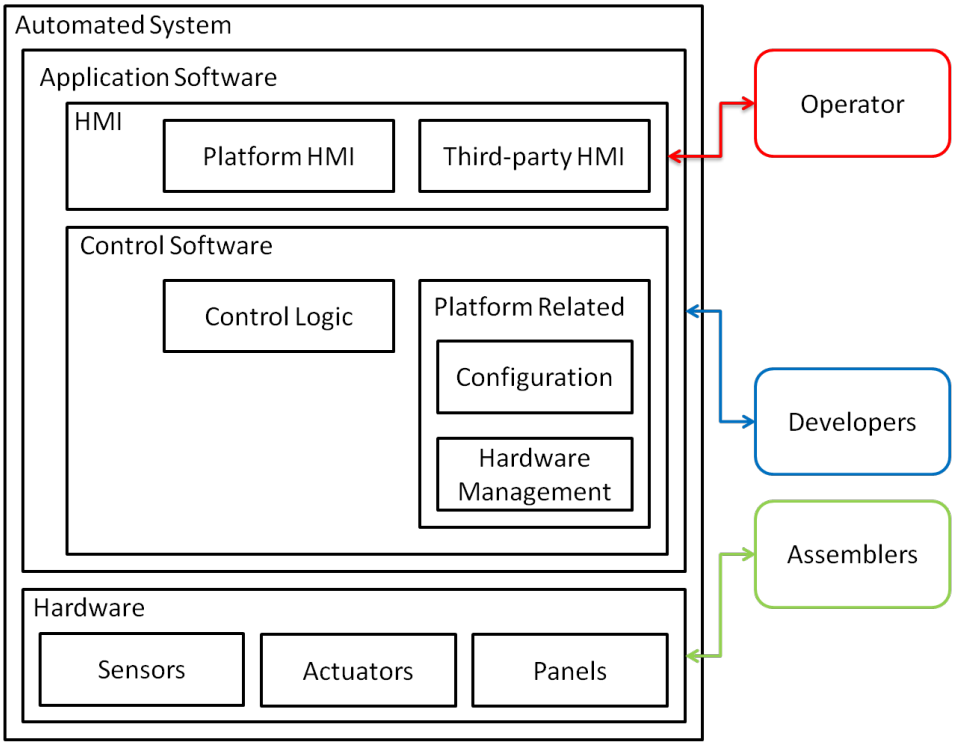

Fig. 2.2 shows the AS entities and actors identified during the Top-Down identification phase. The entities compose the AS, while the actors interacts/build the AS to perform its task. Fig. 2.3 the actors-entities arrows represent which entities are used by which actor.

- the Application layer uses the Application Interfaces to manage the AS behavior from the Control Logic. Those interfaces presents information and generic actions (e.g. move forward) that are implemented in the FBs of the Abstraction layer.

Abstraction is the middle level which uses the FBs implementing the reactive behavior of the devices (e.g. input, output, cylinder, etc.) and the methods and properties from the HMI and Abstraction Interfaces;

Platform is the lowest level implementing the hardware-dependent details which are wrapped into the Abstraction FBs methods and properties.

Furthermore, the entities of the Presentation and Abstraction are further organized by an Model-View-controller (MVC) pattern [51]: the HMI Interface separates the presentation of the data of the device from its control logic which is in the Application Interface; the access methods and the behavior of the device are in the FB of Abstraction. The Fig. 2.4 shows the

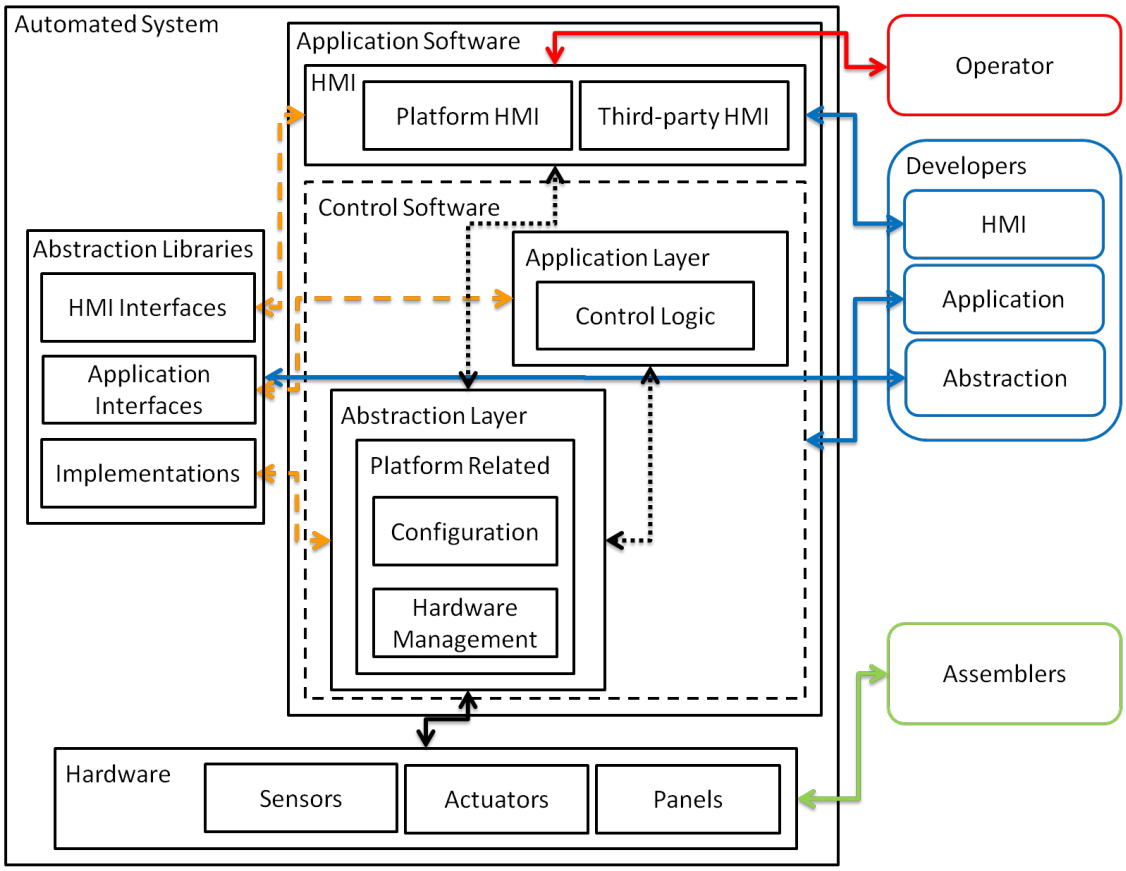

Fig. 2.3 shows Fig. 2.2 actors and entities reorganized as ALOOA. In the figure, the actors are divided relaying on their task: HMI development, CL development, and Platform Related Software development. The entities are divided relaying on the Fig. 2.1 layers. Moreover, those entities (i.e. FBs, Interfaces, and Data Types) are gathered in a set of libraries for reuse. The arrows from the libraries to the entities represents the "use" relation, while the arrows from the HMI and Application layer entities represents the "interface of" relation. Finally, the arrow from the Abstraction layer to the Hardware represents the "access" relation.

MVC architecture inside ALOOA, while List. 1 shows an example of the architecture in an IEC61131-3 3rd pseudo-code.

Finally, ALOOA uses also the OO hierarchy and polymorphism features of IEC61131-3 3rd [74]. In fact, it uses a set of base FBs which allows developers to define a generic device of a certain kind (e.g. input/output, axis, etc.) that is extended to implement the hardware-dependent details in more specific FBs. For this purpose, the entities of ALOOA are grouped according to their kind in a set of libraries (e.g. Kernel Abstraction, Input Output, Motion, etc.) as in Fig. 2.5. Each of those libraries collect the FBs and interfaces from the most generic one (e.g. an axis with unspecified control technology) to the most specific ones (e.g. Beckhoff AX5000 2^{nd} generation [1]).

Listing 1 shows an example of the MVC architecture of ALOOA. APL_Cylinder interface represents the Application layer interface, while HMI_Cylinder represents the HMI interface. Those interfaces are implemented by Cylinder FB, which implements the device reactive behavior and the generic methods with the hardware-dependent details.

```
INTERFACE APL_Cylinder
    METHOD MoveForward: BOOL
        // NO IMPLEMENTATION of the method.
    END_METHOD
    ...
END_INTERFACE

INTERFACE HMI_Cylinder
    METHOD Configure
        // NO IMPLEMENTATION of the method.
    END_METHOD
    ...
END_INTERFACE

FUNCTION_BLOCK Cylinder IMPLEMENTS HMI_Cylinder, APL_Cylinder
    METHOD MoveForward: BOOL
        // Method WRAPS IMPLEMENTATION for the cylinder.
    END_METHOD
    METHOD Configure
        // Method WRAPS IMPLEMENTATION for the cylinder.
    END_METHOD
    ...
        // Reactive behavior of the cylinder in FB BODY.
END_FUNCTION_BLOCK
```

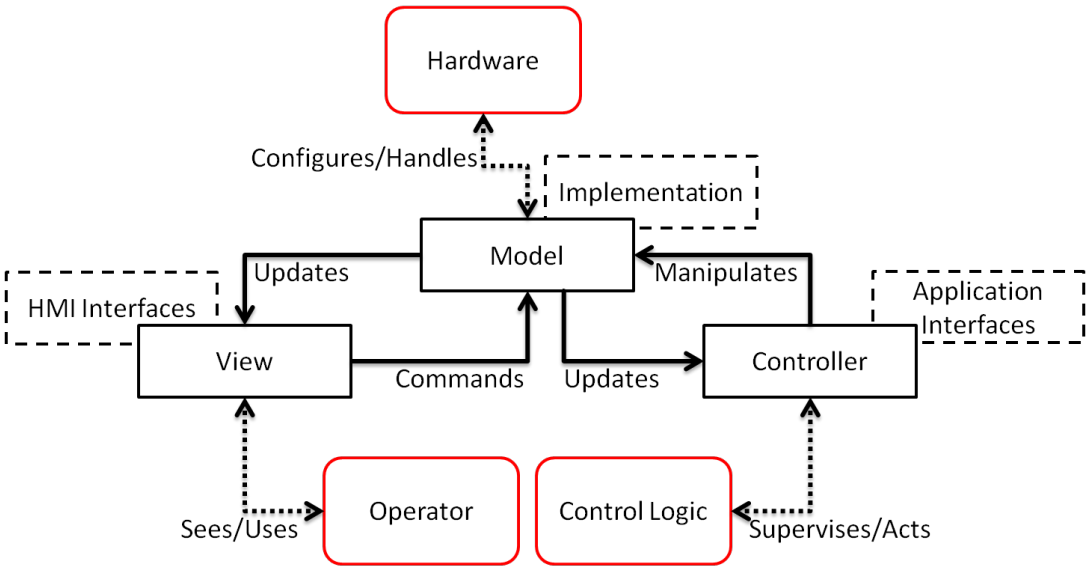

Fig. 2.4 shows the MVC architecture among HMI (i.e. View), Application layer (i.e. Controller), and Abstraction layer (i.e. Model). The model captures the behavior of the device independently of the View and Controller, but with hardware-dependent details. The Controller sends commands to the control logic device in a hardware-independent way. The View generates the viewing and allows the configuration of the model by HMI (e.g. panel).

2.3 Motion control by ALOOA

The ALOOA described until now is used to define an additional level of abstraction than the one supported by the PLCOpen. This additional level is provided through the use of inheritance and by generic devices (e.g. as suggested in [27]). However, the architecture described up to now is limited. In fact, this architecture can not represent certain actions and this limitation is highlighted if the architecture is applied to devices with such actions.

For example, there are actions in motion control that must be performed for a number of PLC cycles until completion (e.g. positioning, homing, etc.) with a certain behavior. Such actions can not be represented by a simple structure as a method, but require a more complex structure able to manage their status during several PLC cycles. For this purpose, ALOOA was extended to resolve this limitation. This extension represents such actions through FBs named “Method FBs”. Method FBs are characterized by a reactive behavior, as well as Abstraction layer FBs (Sec. 2.2). This reactive behavior allows Method FBs to track the status during PLC cycles. The coupling Abstraction Library FBs with Method FBs happens wrapping Method FBs in the methods of Abstraction layer FBs. Thus, the behavior of Method FBs is masked to Abstraction layer FBs.

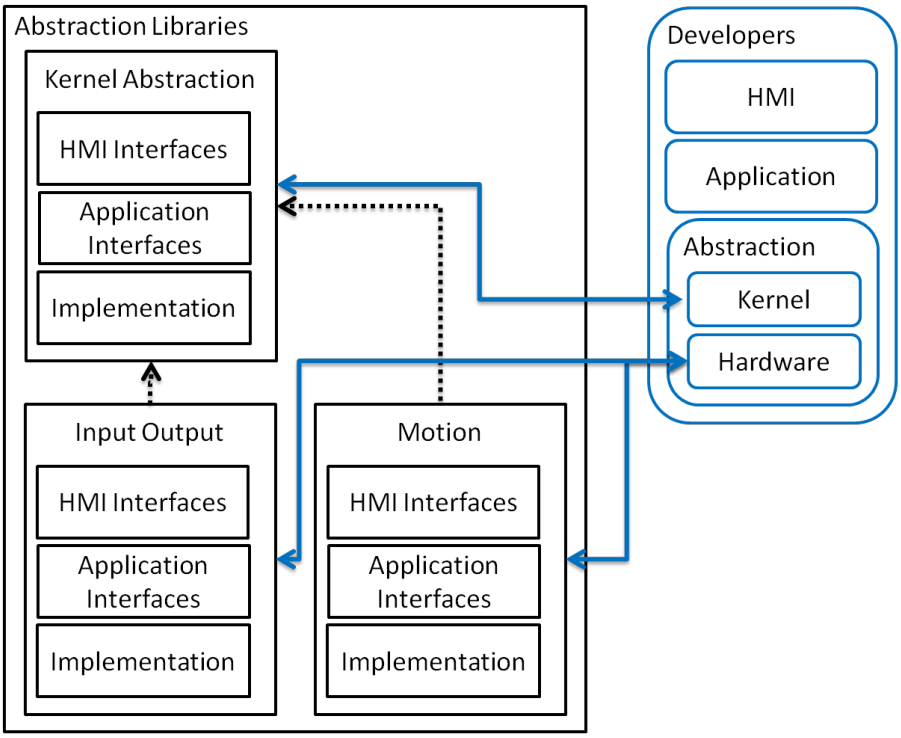

Fig. 2.5 shows how libraries are structured. The Kernel Abstraction library contains FBs and interfaces of base and common entities, which are used/extended in the other libraries (e.g. HMI_Device, APL_Device, FB_Device, etc.). The other libraries contain the base/generic FBs of the device kind, their interfaces, and the specific implementations.

Furthermore, the Method FBs observe the same architecture described in ALOOA 2.2. Hence, those FBs are structured in more layers through inheritance. The highest layer contains the generic method within the management of the generic and/or common reactive behavior (i.e. hardware-independent). This layer should be used and managed by the Abstraction layer FBs in order to mask the details of the lowest layer through the polymorphism. In fact, the lowest layer (i.e. final) contains the reactive behavior implementation with the hardware-dependent details of the specific action for the specific device.

The resulting ALOOA is a “two dimensional” n-tier architecture and is shown in Fig. 2.6. In this type of architecture, the devices are abstracted through the first “dimension’ (2.2), while the actions (i.e. methods) are abstracted by the second “dimension”.

2.4 Example and results

In this section we present the use of ALOOA through an example. The example gives an idea of how ALOOA can be applied to the motion control. The example is presented by

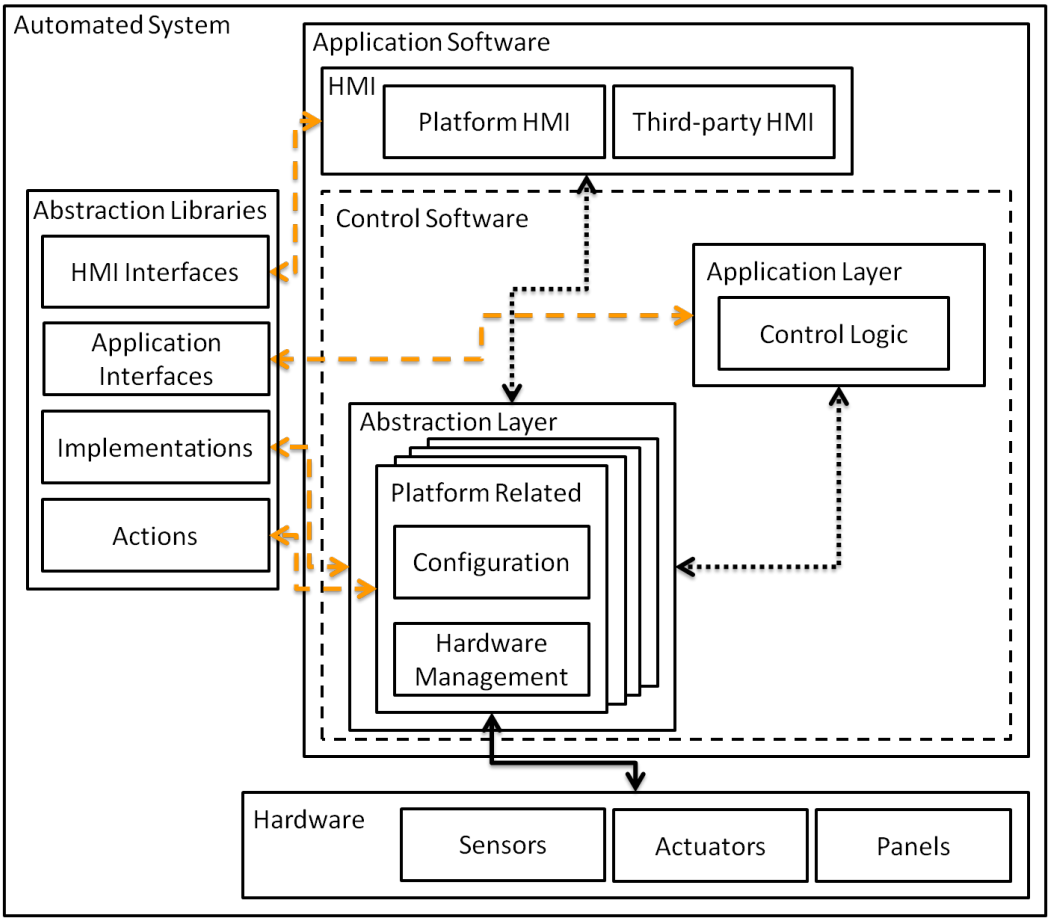

Fig. 2.6 shows the resulting ALOOA. The HMI uses the HMI Interfaces to handle the plant. The Application layer uses the Application Interfaces to control the plant from the Automated System of Modules point of view. The Abstraction layer uses the Implementation of HMI Interfaces and Application Interfaces to handle the devices of the Automated System. The Implementations use the Actions to handle the Platform Related actions.

pseudo-code which represents the implementations of an “analog” and a “CANOpen” axis in Abstraction layer.

List.2 represents a generic axis in the highest level of Abstraction layer. For this purpose, it implements the HMI and Application interfaces in order to be used by the HMI and Application layer.

AxisDevice implements the reactive behavior of a generic axis in its body. This reactive behavior allow the FB to handle the device and its actions. An example of a reactive behavior can be the one from PLCOpen Motion Control library [10], which is shown in Fig. 2.7. The actions (e.g. homing in List. 4) of the generic axis are represented by the instanced actions (i.e. Methods FB, for example List. 3) which are wrapped by its methods. Every of those actions has a reactive behavior which can be represented as in Fig. 2.8. Of course, actions

Listing 2 shows an example of the implementation of a generic axis in the Abstraction layer, with its actions and its reactive behavior. In this example the HMI does not support the use of interfaces (e.g. HMI_Axis). Hence, we used a Struct (i.e. ST_HMI) to access the Implementation. The Implementation methods can be called through booleans (i.e. the Implementation calls its methods on the rising edge of the action booleans).

```
FUNCTION_BLOCK AxisDevice IMPLEMENTS APL_Axis
VAR
   Identity : Device_Id;
   mConfigure : BaseMethod;
   mHome : BaseMethod;
   mMove : BaseMethod;
   ...
END_VAR
VAR_INOUT
   HMI_Interface : ST_HMI;
END_VAR
METHOD Configure : BOOL
   mConfigure(...);
END_METHOD
METHOD Home : BOOL
   mHome(...);
END_METHOD
// FB BODY implements the Axis State Machine.
```

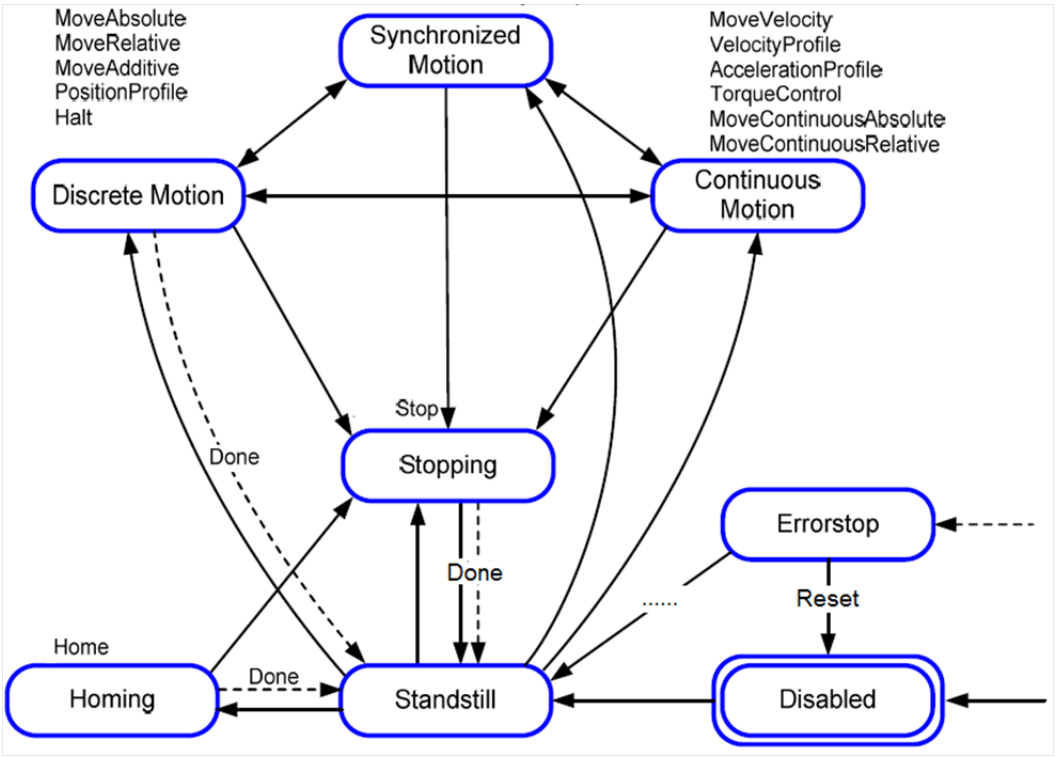

Fig. 2.7 shows the reactive behavior of a servo-controlled motion axis of the PLCOpen Motion Control Library.

have a number of levels as Abstraction layer implementations: from the most generic (i.e. List. 3) to the most specific (i.e. List. 4). Hence, if an axis has an action with a different behavior or hardware-implementation than the one defined by its base implementation, the developer can just do the override of the instantiated action.

List. 5 and List. 6 are two extensions of the generic axis implementation. AxisDevice_Analog is implementing an axis that has the same general behavior of Axis, while AxisDevice_CANOpen represents the implementation of a CANOpen axis. While AxisDevice_Analog shares the reactive behavior with AxisDevice, AxisDevice_CANOpen extend the state machine of the generic axis as it has a different behavior.

To give an idea of the results that we obtained in our project, we report the degree of code reuse during the development of the Abstraction layer implementations of two types of axis provided by Beckhoff: AX5000 [1] and XTS [2]. The AX5000 is a “traditional” series of servo drive in single or multi channel form, while the XTS is a “linear” independent carts servo drive system.

The generic axis (i.e. AxisDevice in List. 2) has:

- 23 methods (e.g. configure, homing, jogging, brake, positioning, etc.);

Listing 3 shows an example of base Method FB. BaseMethod is a FB with the reactive behavior of a generic action (e.g. as shown in Fig. 2.8). That FB should be extended by the specific Method FBs with the action details.

```
FUNCTION_BLOCK BaseMethod
VAR
   State : OpState;
END_VAR
METHOD Init : BOOL
   ...
END_METHOD
METHOD Active : BOOL
   ...
END_METHOD
METHOD Done : BOOL
   ...
END_METHOD
// FB BODY implements the Generic Method State Machine.
```

Listing 4 shows an example of an Method FB. AxisHome extends the BaseMethod with a reactive behavior that performs the homing action of an axis.

```
FUNCTION_BLOCK AxisHome EXTENDS BaseMethod
METHOD Init : BOOL
   ...
END_METHOD
METHOD Active : BOOL
   ...
END_METHOD
// FB BODY implements the Homing Method State Machine.
```

Listing 5 shows an example of the extension of the generic axis to an “analog axis”. Moreover, it shows the extension of mHome action from BaseMethod implementation to AxisHome.

```
FUNCTION_BLOCK AxisDevice_Analog EXTENDS AxisDevice
...
   mHome : AxisHome;
...
// FB BODY, reuse the basic State Machine.
SUPER();
END_FUNCTION_BLOCK
```

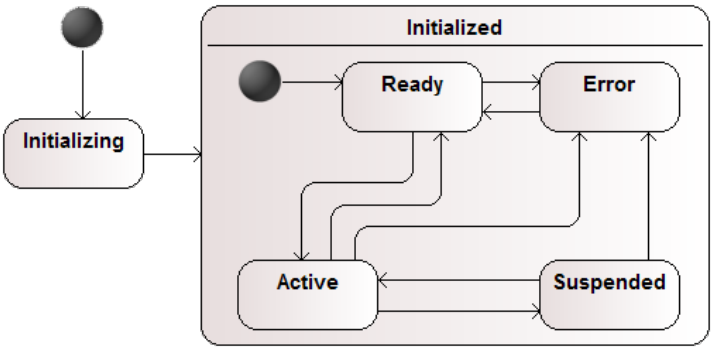

Fig. 2.8 shows an example of the reactive behavior of a generic action.

Listing 6 shows an example of an CANOpen axis. The CANOpen axis has a different behavior than the one from AxisDevice. Due to that it implements another state machine.

```
FUNCTION_BLOCK AxisDevice_CANOpen EXTENDS AxisDevice
...
   mHome : AxisHome;
...
// FB BODY, implements CANOPen DS402 State Machine.
CASE DS402_State OF
   Not_ready_to_Switch_ON : ...
       Switch_On_Disabled: ...
END_CASE
END_FUNCTION_BLOCK
```

- a state machine that shapes the generic reactive behavior.

During the development of the two specific implementations, we overrode:

- four methods (i.e. configure, brake, homing, and configureDev) which resulted in lesser than 30% of the generic axis code;
- about 20% of AX5000 state machine;
- about 30% of XTS state machine.

The ALOOA approach allowed us to obtain a degree of reuse of at least 70%.

2.5 Summary

We have presented ALOOA and an example of its application to motion control. ALOOA allowed us to have a degree of reuse in our project of more than 70%. Moreover, it improved

the flexibility, maintainability, and life-cycle-management. This approach also reduced the platforms-and-their-limitation difficulties by providing an abstraction layer higher than the one provided by the well-known PLCOpen.

We believe that ALOOA approach could be used improve the control software development. Moreover, we expect that it could help OO features to penetrate further into the PLC world.

Oriented Design Pattern), may require documentation (as more levels are developed in the Abstraction layer, more complex it becomes to know who does what), and requires the verification and validation of the libraries. However, it improves software development for automation. In addition, document, verify and validate the code is suggested (if not mandatory) in software development.

ALOOA has some limitation:

- ALOOA requires OO features, then is not applicable in functional programming;
- while OO features improve ALOOA components maintainability, it increases the complexity of the code [58]. Additionally, more levels are developed in the architecture, more complex it becomes to know who does what. Therefore, ALOOA libraries may require documentation;
- ALOOA requires to verify and validate entities of Abstraction libraries as they are the base of Application software.

While ALOOA has still some limitations, it improves software development for automation. Moreover, documenting and verifying and validating the code is suggested (if not mandatory) in software development.

In next chapters we introduce our methods which we believe overcome some of ALOOA limitations.

Chapter 3

Model reactive behaviors

In Chap. 2 we introduced ALOOA architecture which intensively use models to describe the reactive behavior of its components. This reactive behavior is usually defined as state machine, which are a set of states connected via transitions.

A state is a persistent (i.e. non-transient) system condition, in which its behavior is peculiar and only a sub-set of its features are of interest. For example, when the valves controlling an hydraulic linear actuator are in a certain configuration, the position of the piston is either increasing or decreasing with respect to its home position. If the position of the actuator is detected only with proximity switches at the two end-stroke and the actuator is in the "extending" state, from a control point of view we are interested only on the signal coming from one of the two proximity sensor, in order to stop the actuator's movement.

State machines are usually implemented in IEC61131-3 framework [44] using Sequential Functional Chart (i.e. SFC) language. SFC presents some important limitations with respect the application to machine control applications. In particular, SFC lacks of the state hierarchy structure featured by the State-chart language embedded in UML specification [78]. In fact, SFC permits that a nested state is active while its macro state is deactivated, which is in contrast with models used in ALOOA (i.e. State-charts). Moreover, SFC does not permit inter-level transitions.

State hierarchy provides an efficient solution to the problem of asynchronous conditions management in a state machine. A typical condition is the management of alarm that might occur in all the states of a complex state machine. Hence, using state hierarchy, these states can be grouped in a super-state from which is easy to manage a transition triggered by a common alarm condition toward the state sequence for alarm handling. Moreover, using the "history" pseudo-state, it is easy to return from alarm management sequence to the main state sequence.

Because of these characteristics, the implementation of state-chart in IEC61131 PLC code has been widely studied arriving to propose several solutions that present both pro and cons, as discussed in the next section. Recently IEC languages have been enriched by Object Oriented (i.e. OO) characteristics [95], which indeed are not yet deeply exploited in the current practice of industrial applications. Starting form this point, this chapter presents our approaches, which takes advantage from these features. Our approaches aim to propose new methodologies for the implementation of state-chart in IEC61131-3 languages with OO extensions, combining an effective implementation with a simple approach.

3.1 Related works

State-charts have been adopted widely by software community, from computer science to automation domain, since Harel's work [43]. In automation domain there are several works that aim to integrate State-charts with PLC software. Vogel-Heuser, et al. in [90] propose an approach to generate IEC61131-3 code from UML-Model.

In [97], Witsch, et al. define an adaptation of UML state-charts which can be used as a visual programming language for PLCs.

Krzysztof, in [71] describes a new kind of timed automata and ends with a complete program written in one of the IEC61131 languages.

Seidel, et al. in [77] introduce an approach to modeling the teal-time behavior of PLCs using UML state-charts. Vidanapathirana, et al. in [88] also use state-chart modeling to control an elevator on a PLC platform.

However, all these works do not take into account the OO possibilities in PLC software.

To the best of our knowledge, only Witsch, et al. in [98] started to explore the introduction of OO in PLC software (i.e. IEC61131) and UML state-charts. They propose a simple and effective design pattern to inherit a common behavior to all modules of the modular machine. However, as they stated, the pattern is for modules with a standardized model (i.e. same state-chart for all modules). In fact, the pattern focuses more on how to provide a common behavior to modules rather than on how to build state-charts with certain functionality. For example, the details on how to Implement the State-charts and their features (e.g. nest states, histories, etc.) are left to the developer.

The literature is rich of works that aim to integrate state-charts with software taking into account the OO possibilities (e.g. C++, Java, etc.). Unfortunately, they do not target the PLC software.

Gamma, et al. in [37] describe design patterns exploring the OO capabilities and pitfalls. One of the provided solution is the State Pattern, which is a design pattern to implement

Finite State Machines (i.e. FSMs). However, State Pattern treats only flat states, whereas State-charts exploits also hierarchical/nested states.

The State Table Pattern, by Douglass in [24], is a pattern for large and flat state machines, that does not handle nested states very easily.

Tamura, et al. in [84] propose the State-chart Pattern, and do a comparison of design patters for implementing FSMs. However, as stated in their work, the execution of State-chart Pattern lack of performance.

Samek, et al. in [72] propose an HSM design pattern to direct map UML-State-charts to C or C++. While, this design pattern covers the main UML state-chart specifications, it is tailored for C and C++ languages and not for PLC, automation, and IEC61131-3 as:

- it reacts to messages, but PLC works on inputs and outputs;
- it treats stimuli (i.e. messages) from the inner state to the outer, while in automation are treated from outer to inner;
- it uses macros, defines, and most of all function pointers, which are not supported by IEC61131-3.

Due to these characteristics, Samek state-chart design pattern could not be used as is in PLC software. However, Samek, et al. work has been one starting point of our work.

3.2 The PLC UML state-chart design pattern

The PLC UML state-chart design pattern (i.e. PUSDP) is a set of classes that allows developers to easily implement UML state-chart (i.e. hierarchical state machine, HSM) models. PUSDP allows developers to implements UML State-charts with the following features:

- nested states with proper handling of group transitions and group reactions;
- guaranteed execution of entry/exit actions upon entering/exiting states;
- guaranteed execution of do actions of current states;
- straightforward implementation of condition on signals;
- design that enables inheriting and specializing state models;
- respect of PLC UML state-chart transition scheduling;

- shallow/deep history.

Some of the more advanced features of UML state-charts have been omitted (e.g. orthogonal regions) to minimize implementation complexity. However, these features can be added as behavioral patterns built on top of the implementation presented here, or changed into equivalent state-chart models with PUSDP implemented features.

3.2.1 The design pattern

The PUSDP is based on the following core classes:

HSM class contains the whole engine of state-charts. Like Samek work [72], HSM methods allow state-charts to evolve and spin;

top points to the top state of the state-chart;

current points to the current state of the state-chart;

next points to the target state of a transition;

onStart initializes the state-chart triggering the outer start state;

toLCA calculates the number of step to reach the Last Common Ancestor (i.e. LCA) of the current state and the target one;

exit_ triggers the onExit methods from the current state to the LCA and update the history;

stateTransition triggers a transition from the current state to the target;

stateStart triggers a start transition from the current state to the target;

stateHistory triggers a transition from the current state to the history of the target state;

spin spins the state-chart;

State class represents a generic state;

handler describes the specific behavior of the state by an handler implementation;

superior is a pointer to the super-state, which defines the nesting of the state;

historyType indicates if there is a "history state" in the state and its type;

historyStates contains the state in the history of the state;

OnStart, OnEntry, onDo, and onExit wrap the handler implementation methods of the same name;

State is the constructor;

Spin is the engine that checks and does the actions and transitions;

Handler is and interface that is used to specify the behavior of a state by its methods in its implementations;

handle will contain the state transitions, guards, and actions;

onStart, onEntry, onDo, and onExit will contain the behavior of the state;

HistoryType is an enumerate which indicates state history type;

History is an alias of a "pointer to State";

Figure 3.1 shows the PUSDP core classes, while the next section introduces its use by an example.

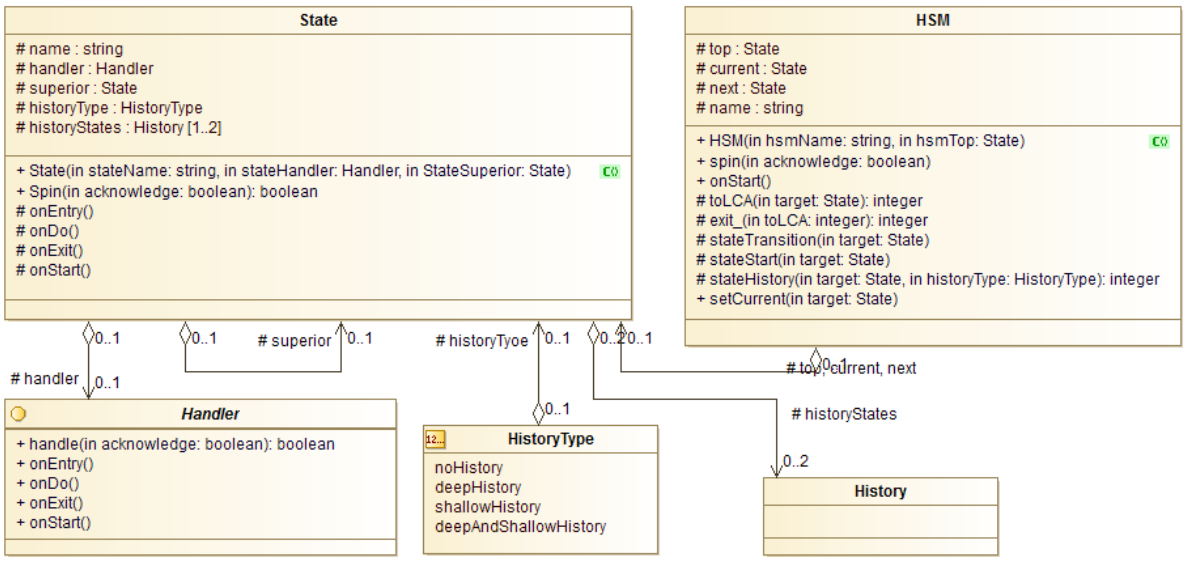

Fig. 3.1 The Class diagram shows the PUSDP core architecture to quickly implement State-charts. Classes properties to access the attributes are masked to minimize the figure space.

3.2.2 An application example

Consider a system like the one shown in Figure 3.2. The system behavior is associated to an internal state which depends by the actual state and the system input signals (e.g. Trx0, Trx1, etc.).

We apply the PUSDP to the example state-chart by these steps:

1. create a new class (the Example class), which extends the Hsm class and inherits the state-chart engine;

2. create one state handler class for each state in the UML state-chart (Figure 3.2) plus the Top state one. These state handlers implement the Handler interface (e.g. TopHandler, HighSpeedHandler, IdleHandler, etc.);
3. in these new classes, instantiate an HSM derived class (Example class) object as private attribute;
4. declare all states (State class instances) and all states handlers (implementation of Handler interface) in the new HSM derived class;
5. declare the states handling function methods (member functions like in List. 7) into the HSM derived class;
6. wrap the handling function methods in new HSM derived class (e.g. List. 8);
7. define the state-chart topology (i.e. nesting states) in the derived class (concrete HSM class) constructor (e.g. List. 9;
8. define the handling function methods, coding the entry, exit, do actions and the transition handlers;
    - return false if you handle the transition and true if you do not;
    - use stateStart() for transition from “start state”;
    - use stateTransition() to fire transitions;
    - use stateHistory() to fire target state history transition;
9. declare the concrete HSM class instance;
10. fire initial start transition by invoking HSM.onStart();
11. invoke Hsm.spin() each PLC cycle.

Figure 3.3 shows the UML class diagram of the application example. We implemented the classes in TwinCAT3 on a Beckhoff PLC, but they can be implemented with any programming language that respects IEC61131-3.

3.3 The inherit hierarchical state machines design pattern

PUSDP allowed us and developers to have and use a well developed, tested and validated engine to build the designed state-charts in ALOOA. However, we experienced some limitation of this design pattern during the doctorate. In fact, developers reported that models

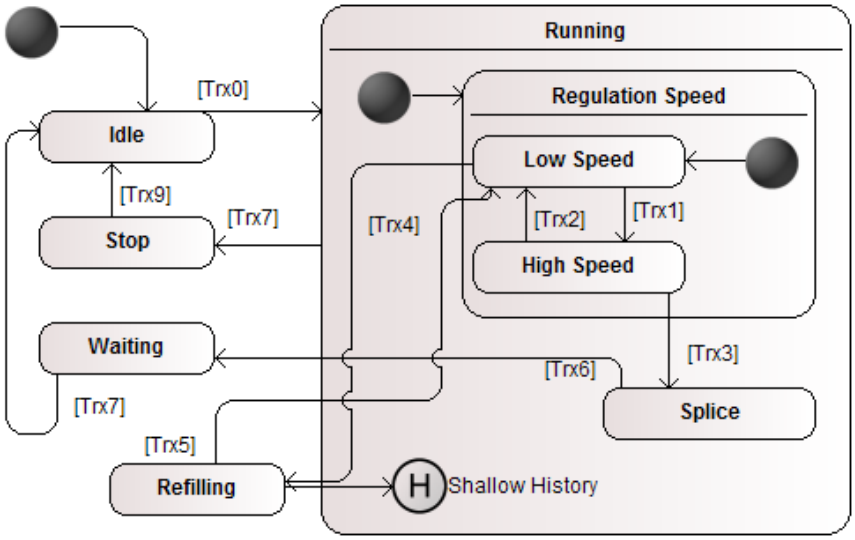

Fig. 3.2 shows the state-chart used as example in this chapter.

Listing 7 The listing shows the code of runningHandle method of the Example class (i.e. Function Block). The method checks if the guard of its transition (i.e. trx8 BOOL variable associated to an input signal) is true, and if fires the transition to the stop state.

```
runningHandle := acknowledge; //Init ack as un-done

IF trx8 THEN
   THIS^.stateTransition(ADR(THIS^.stop));
   runningHandle := FALSE; //Set ack as done
END_IF
```

Listing 8 The listing shows the code of handler method of the runningHandler class (i.e. Function Block). The method wraps the runningHandle method of Example class.

```
//wrap the runningHandle method of Example FB by the Handler
handle := THIS^.context^.runningHandle(acknowledge);
```

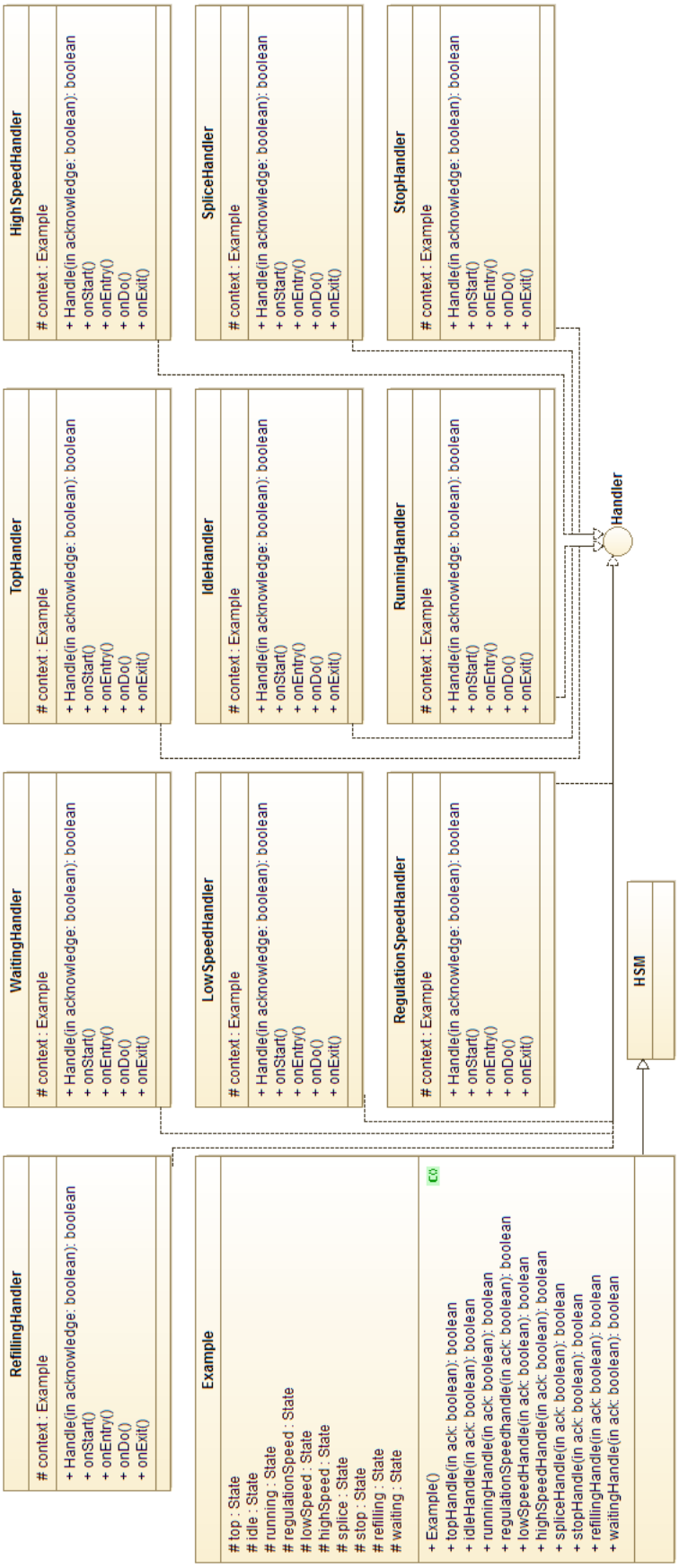

Fig. 3.3 shows the classes to implement the example state-chart (Fig. 3.2) using PUSDP. HSM class and Handler interface members are masked, to minimize the space. Refer to Fig. 3.1 for details.

Listing 9 The listing shows the code of Example constructor method of the Example class (i.e. Function Block). The method initializes the state-chart structure.

```
//Init Wrappers
THIS^.topHndlr.topHandler(ADR(THIS^));
THIS^.stopHndlr.stopHandler(THIS);
THIS^.idleHndlr.idleHandler(THIS);
THIS^.runningHndlr.runningHandler(THIS);
THIS^.regulationSpeedHndlr.regulationSpeedHandler(THIS);
THIS^.lowSpeedHndlr.lowSpeedHandler(THIS);
THIS^.highSpeedHndlr.highSpeedHandler(THIS);
THIS^.spliceHndlr.spliceHandler(THIS);
THIS^.waitingHndlr.waitingHandler(THIS);
THIS^.refillingHndlr.refillingHandler(THIS);

//Init States
topState.State('Top State', 0, topHndlr, 0);

THIS^.top := topState;
THIS^.stop.State('Stop State', THIS^.getTop(), stopHndlr, 0);
THIS^.idle.State('Idle State', THIS^.getTop(), idleHndlr, 0);
THIS^.running.State('Running State', THIS^.getTop(), runningHndlr,
    deepHistory);
THIS^.regulationSpeed.State('Regulation Speed State', ADR(running),
    regulationSpeedHndlr, 0);
THIS^.lowSpeed.State('Low Speed State', ADR(regulationSpeed), lowSpeedHndlr,
    0);
THIS^.highSpeed.State('High Speed State', ADR(regulationSpeed),
    highSpeedHndlr, 0);
THIS^.splice.State('Splice State', ADR(running), spliceHndlr, 0);
THIS^.waiting.State('Waiting State', THIS^.getTop(), waitingHndlr, 0);
THIS^.refilling.State('Refilling State', THIS^.getTop(), refillingHndlr, 0);

//Init HSM
THIS^.Hsm('SFC HSM', THIS^.getTop());
```

built with PUSDP were using several FBs which was confusing. Therefore, we designed and developed the Inherit Hierarchical State Machines design pattern (i.e. IHSM).

IHSM key idea is still the same: an engine to inherit. However, IHSM builds state-charts by inheritance instead of building them by aggregation as PUSDP.

3.3.1 The design pattern

The design pattern use an interface to define the public structure of the engine and the models that will be based on that. This interface publish four methods (i.e. on exit, on do, on entry, and transitions)in charge of handling the different actions. These methods are implemented in the engine, which implements the interface.

The engine contains implements as well the methods used by itself to handle state changes, hierarchy policies, and the different actions (e.g. on entry, on exit, etc.). This engine is extended by the implementations of the state-charts, which will implement all the details about the actions in the four interface method. The states and macro-states of this state-chart are implemented as variable within a structure.

When a state-chart implementation has to change the behavior of the actions, it has to be extended by another class which will overload these actions. If a state-chart implementation has to change its states and/or macro-states, then it has to use the extension of the structure which adds the additional states and/or macro-states. In case the state-charts has to be extended by new states and actions, it will be extended by a class which will overload and/or add new actions and will use the extension of its state structure. Fig. 3.4 shows a class diagrams which describes the relations among the design pattern entities, their methods, and structures.

3.3.2 An application example

Applying IHSM design patten requires these steps:

1. create a new class/FB (e.g. Example), which extends the engine class (e.g. FB_Statechart_Engine);

2. create one structure containing all the states, pseudo-states, and macro-states as boolean (e.g. Example_STRUCT);

3. implement each method from the interface by coding the state-chart topology. The topology may be implemented as each developer prefers. For example, in our implementation we used simply a set of nested if. The important thing is to use the methods

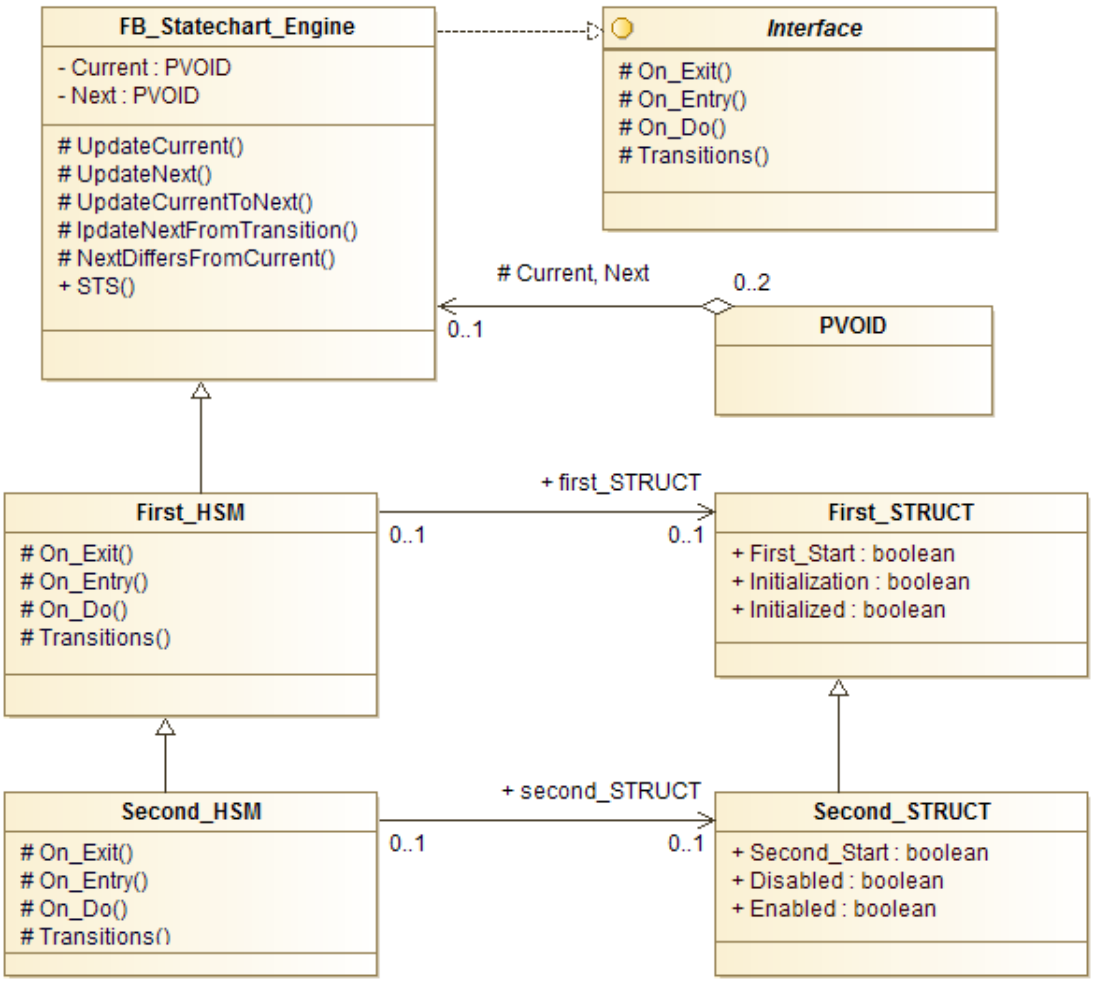

Fig. 3.4 shows the design pattern classes and the relations among them. Interface is used to publish the method which handle the actions. FB_Statechart_Engine implements the state-charts engine, which is in charge of providing all the methods to handle the actions, their policies, and the hierarchy. First_HSM is an example of a state-chart which contains the initialization state and initialized state. These states are provided by a structure called First_STRUCT. In this example, First_HSM is extended by Second_HSM, which adds Disabled and Enables states states by Second_Struct.

provided by the engine to handle the single actions (e.g. state transition method in List. 14);

4. declare the concrete HSM class instance;

5. declare a structure containing the HSM states (i.e. Example_STRUCT);

6. assign the structure instance to the HSM class;

7. invoke the body of the class for each PLC cycle.

The idea of IHSM is to implement models as a set of layers. Each layer is a class (i.e. FB as in List. 12). The first implementation contain the most generic part of the HSM. Then each extension implements more specific details.

Consider as example the model in Fig. 3.2 used in Sec. 3.2.2. This model could be implemented with IHSM in different ways. For example, it could be implemented by two layers:

- the first layer (e.g. List. 12) represents a generic device without speed regulation: idle, stop, waiting, refilling, and running states;

- the second layer (e.g. List. 13) represents a device with speed regulation which adds regulation speed, splice, low speed, and high speed states.

Following this implementation, second layer extend first layer actions by overloading the instances methods and using second layer state structure. For example, List. 14 overloads transitions actions of Transitions method in List. 15 while List. 10 shows the structure of first layer which is extended by the structure of second one in List. 11.

3.4 Synchronization among models

Both design patterns described in this chapter allow you to implement UML state-charts, but are limited when applied to ALOOA. In fact, ALOOA cobines layers of state-charts which are linked among them. These links make that the evolution of a state-chart could influence the evolution of other. When this influence occurs, there may be cases of non-synchronism among state-charts (i.e. models).

Listing 10 The listing shows the code of Example_STRUCT which contain the first layer of the HSM states.

```
TYPE Example_STRUCT :
STRUCT
   SYSTEM_Start : BOOL := TRUE; // If true, System macrostate is not active,
      so the IHSM is not active. Else, the macrostate is active.
   Idle    : BOOL;  // If true, it represents the Idle state as active. Else,
      the state is not active.
   Stop    : BOOL;
   Waiting  : BOOL;
   Refilling  : BOOL;
   Running  : BOOL;
END_STRUCT
END_TYPE
```

Listing 11 The listing shows the code of Regulated_STRUCT which contain the second layer of the HSM states.

```
TYPE Regulated_STRUCT TYPE Advanced_Struct EXTENDS Example_STRUCT :
STRUCT
   Running_Start : BOOL := TRUE; // If true, Running macrostate is not
      active, so the IHSM is not active. Else, the macrostate is active.
   Regulation_Start : BOOL := TRUE; // If true, Regulation Speed macrostate
      is not active, so the IHSM is not active. Else, the macrostate is
      active.
   Regulation  : BOOL;  // If true, it represents the Regulation state as
      active. Else, the state is not active.
   LowSpeed  : BOOL;
   HighSpeed  : BOOL;
   Splice    : BOOL;
   Running  : BOOL;
END_STRUCT
END_TYPE
```

Listing 12 The listing shows the body of Example FB which contain the first layer of Fig. 3.2 HSM. For simplicity, two instances of Example_STRUCTS may be aggregated, but this is not mandatory as the engine uses two void pointers to spin the state-chart. These two void pointer have to be provided by a setter during initialization.

```
FUNCTION_BLOCK Example EXTENDS FB_Statechart_Engine
VAR
   stCurrent : Example_STRUCT;
   stNext : Example_STRUCT;
END_VAR
```

Listing 13 The listing shows the body of Regulated FB which contain the second layer of Fig. 3.2 HSM. For simplicity, two instances of Regulated_STRUCTS were aggregated.

```
FUNCTION_BLOCK Regulated EXTENDS Example
VAR
   stCurrent : Regulated_STRUCT;
   stNext : Regulated_STRUCT;
END_VAR
```

Listing 14 The listing shows the code of Transition method of Example class which contain the first layer of the HSM states. At the beginning it is mandatory to call the Transitions method of the super class in order to maintain the HSM topology. Then, it is mandatory also to update current instance of state structure (i.e. Example_STRUCT) and update the instance of next state by their methods.

```
SUPER^.Transitions();

UpdateCurrent(ADR(current), SIZEOF(current)); // Update current to the
    current.
UpdateNext(ADR(next), SIZEOF(next));   // Update current to the current.

//---------------------------------------------------------------
IF current.SYSTEM_Start THEN
   // Starting transition.
   Transit(next.SYSTEM_Start, next.Disable);
ELSIF current.Idle THEN
   S_Idle(next, ePhase := E_State_Phases.on_transitions);
ELSIF current.Stop THEN
   S_Stop(next, ePhase := E_State_Phases.on_transitions);
ELSIF current.Waiting THEN
   S_Waiting(next, ePhase := E_State_Phases.on_transitions);
ELSIF current.Refilling THEN
   S_Stop(next, ePhase := E_State_Phases.on_transitions);
ELSIF current.Running THEN
   S_Running(next, ePhase := E_State_Phases.on_transitions);
END_IF
//---------------------------------------------------------------

UpdateNextFromTransition(ADR(next), SIZEOF(next));
```

Listing 15 The listing shows the code of Transition method of Example class which contain the first layer of the HSM states. At the beginning it is mandatory to call the Transitions method of the super class in order to maintain the HSM topology. Then, it is mandatory also to update current instance of state structure (i.e. Example_STRUCT) and update the instance of next state by their methods.

```
SUPER^.Transitions();

UpdateCurrent(ADR(current), SIZEOF(current)); // Update current to the
    current.
UpdateNext(ADR(next), SIZEOF(next));   // Update current to the current.

//-----------------------------------------------------------------
IF current.Running THEN
   S_Running(next, ePhase := E_State_Phases.on_transitions);
   // Macro
   IF next.Running THEN  // Check if we want to stay in the macro-state.
      //Macro Explosion!!
      IF current.Running_Start THEN
         Transit(next.Running_Start, next.Regulation); // No default state.
      ELSIF current.Splice THEN
         S_Splice(next, ePhase := E_State_Phases.on_transitions);
      ELSIF current.Regulation THEN
         S_Regulation(next, ePhase := E_State_Phases.on_transitions);
         // Macro
         IF next.Regulation THEN  // Check if we want to stay in the
             macro-state.
            //Macro Explosion!!
            IF current.Regulation_Start THEN
               Transit(next.Regulation_Start, next.LowSpeed); // Default
                   state.
            ELSIF current.HighSpeed THEN
               S_HighSpeed(next, ePhase := E_State_Phases.on_transitions);
            ELSIF current.LowSpeed THEN
               S_LowSpeed(next, ePhase := E_State_Phases.on_transitions);
            END_IF
         ELSE
            // Macro-state reset.
            Macro_Reset_Regulation(next);
         END_IF
      END_IF
   ELSE
      // Macro-state Reset.
      Macro_Reset_Running(next);
   END_IF
END_IF
//-----------------------------------------------------------------

UpdateNextFromTransition(ADR(next), SIZEOF(next));
```

3.4.1 Propagation delay

Whenever two models are out of sync in ALOOA, there is a critical and potentially unsafe situation. For example, an axis is prompted to perform homing in two contiguous PLC cycles. During the first PLC cycle axis model receives the first request for homing and it sets its next status as Homing;

During the second PLC cycle:

- axis model changes its current status to Homing which enables the condition to activate homing action;
- axis model evaluates its transitions while the homing condition is still active as second homing request. The transition which handles an homing request during an homing fires and sets as next status Error;
- homing model changes its next status to Active due to the condition enabled by Homing in axis model.

During the third PLC cycle:

- axis model changes its current status to Error which enables the condition to activate Error in homing model;
- homing model changes its current status to Active and execute exit, entry and do methods linked to this state. Axis model and homing model are out of sync;
- homing model evaluates Active transitions which set as next status Error due to conditions enabled by Error in axis model;
- the two models are out of sync.

During the fourth PLC cycle homing model changes its current status to Error and execute exit, entry and do methods linked to this state. Only in this PLC cycle the two models sync back again.

This example is shown in Fig. 3.5 and it introduces how it is possible to have critical situation due to the out of sync. We called this issue "propagation delay".

The propagation delay is related to the very nature of state machine. In fact, this is because state change occurs the next cycle after the transition which is propagated to having separated state machines which one influences other.

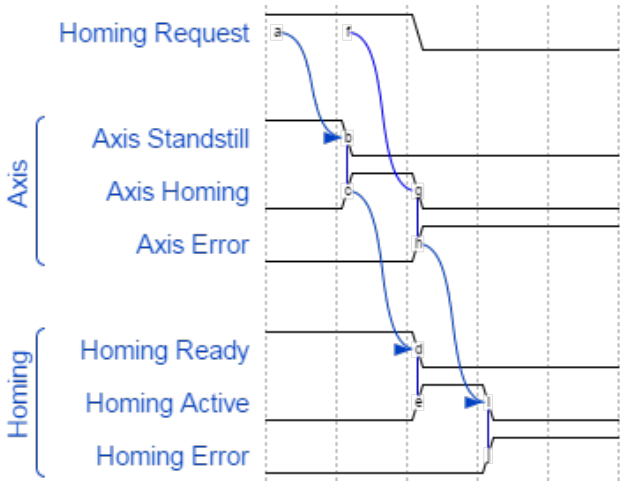

Fig. 3.5 shows a timing diagram which provides an example of propagation delay. The homing request in two contiguous PLC cycles create an out of sync situation in third cycle among axis and homing models.

3.4.2 Dealing with propagation delay

To address this issue, we have decided to allow state machines to share their future status among other and re-evaluate transitions after next status is reported by underlings. For this purpose, we changed our design patterns (i.e. PUSDP and IHSM) to share next status. Moreover, we modified design pattern engines to perform an additional evaluation of transitions after that exit, entry, and do actions and transitions of underlings.

In order to check if this current approach solves the propagation delay, we test it with the example of the homing action during two contiguous PLC cycles.

During the first cycle:

- axis model receives the first homing request and sets its next status to Homing;
- homing model sets its next status to Active due to the next status of axis model.

During the second cycle:

- axis model sets current status to Homing and executes the actions related to it;
- homing request stays enabled and axis model sets next status to Error while checking its transitions;
- homing model sets its current status to Active due to the next status of axis model and then executes the methods linked to current status actions;
- homing model sets its next status to Error due to the next state of axis model;

- axis model re-evaluates its transitions but none triggers.

Finally, during the third cycle:

- axis model sets current status to Error and executes the actions related to it;
- homing model sets its current status to Error and executes the actions related to it as well;

Both models are synchronized and the same would happen also if homing model would go in error before axis model. Fig. 3.6 shows the example which uses the new approach.

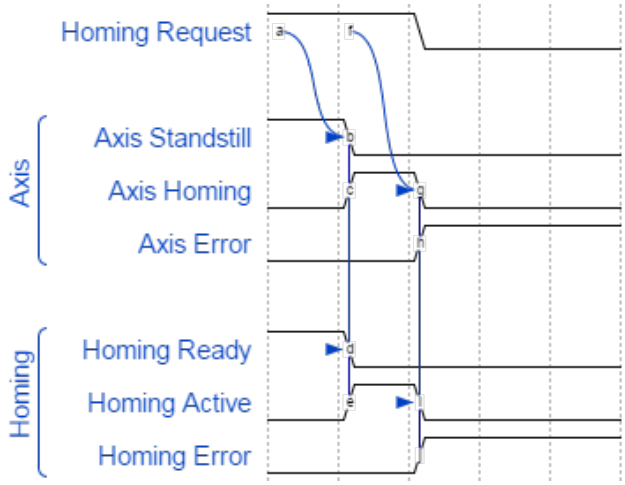

Fig. 3.6 shows a timing diagram which represents the example of Fig. 3.5, but keeping axis and homing models synchronized by the new design pattern engine. In fact, during the second PLC cycle the models stay aligned.

It is important to know that the re-evaluation of transitions may be critical if models are badly constructed. In fact, models with contradictory condition on transitions could introduce instability in models evolution.

For example, Fig. 3.7 shows two state machines with contradictory transitions. The axis state machine (i.e. standstill, synchronized motion, and stopping states) has "standstill" as initial state, while the discrete state machine (i.e. ready, error, and active states) has "ready" as initial state. During the first PLC cycle, axis model sets its next status to "synchronized motion", then discrete model sets its next status to "error" due to axis next status which is represented by "axis -> synchronized motion" guard. However, axis model re-evaluates its transitions and changes its next status to "stopping" as the contition "axis -> synchronized motion" is true. At this point, discrete model still has "error" as next status even if the condition for that state is no longer valid.

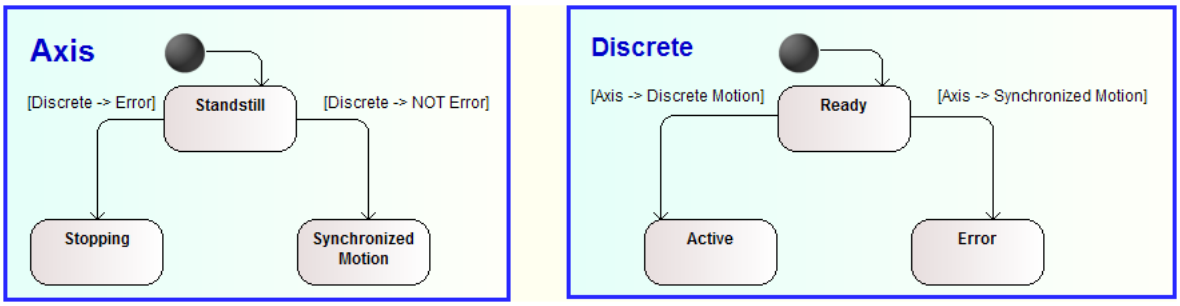

Fig. 3.7 represents two state machines that individually do not have issues, but with their interaction create a critical situation. In fact, they both have contradictory transitions one against the other model.

Fig. 3.7 example is also presented as timing diagram in Fig. 3.8. This timing diagram represents current and next model status at each evaluation istant, and expresses the evaluation istants as "cycles". To prevent such kind of situations, models should be verified before during design phase and before their implementation.

3.5 Summary

This chapter can be summarized as follows:

- we explored the current state of the art for state-chart integration with PLC software, and the current state of the art for State-chart Design Patterns;
- we proposed two design pattern to direct map UML State-charts to IEC61131 code;
- our design patterns use OO characteristics, providing an engine made of classes and mechanisms that allows developer to realize the static structure of State-charts;
- design patterns are compliant to ALOOA architecture;
- engine OO characteristic allow the engine classes and mechanisms to be used as library (e.g. provided by the enterprise, PLC producer, etc.);
- engine classes and mechanisms can be extended to introduce new features in all the state-charts;
- we provided application examples that uses our design patterns, their class diagrams, the steps to realize them in PLC code, and some code example.

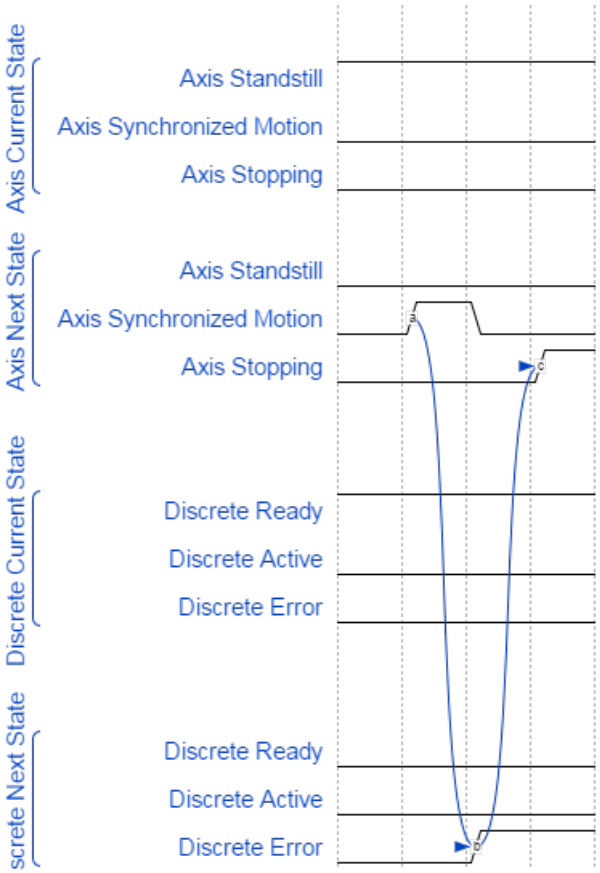

Fig. 3.8 represents two state machines transitions evaluation and re-evaluation. The first cycle represents the istant before the transition evaluation of the axis model. The second cycle represents the transition evaluation of the axis model. The third cycle represents the transition evaluation of the discrete model, while the fourth cycle represents the transition re-evaluation of the axis model.

These works improve ALOOA architecture by using OO features, which improve code maintainability and reusability. However, OO features limit the applications of such design patterns to PLC that are IEC61131-3 compliant and introduce complexity in code [58]. While it is not possible to overcome the first limitation, we believe to reduce the introduced complexity by generating documentation from code.

Chapter 4

Automatic generation of documentation

The new approaches introduced in the software development for automation [92] fulfilled a part of challenges due to the growing cost of software development in machinery [82]. However, those approaches highlighted the criticality of keeping the downtime caused by software usability and life cycle management as low as possible or to completely avoid it [89].

While it is still possible to improve those approaches to ease the use of the developed software as much as no further help or documentation is needed, this goal cannot always be met [60]. In fact, a documentation will always be needed, or at least provided.

Accordingly to this and to the growing complexity of software in automation, instead of improving the approaches, in this chapter we focus on improving and easing the documentation process to improve the usability and life cycle management of software.

4.1 Related works

It is a well-known fact that software documentation is a widely studied topic [23, 39, 55, 57]. However, software documentation, in practice, is often still poor and incomplete. This is mainly due to:

- the perception that software documentation is often too expensive and difficult to maintain [31];
- lack alignment between documentation and software versions (i.e. documentation updates [57]);
- lack of a defined standard and crucial information to make the documentation understandable and usable by developers, maintainers, and users [16, 55];

- different people, groups, and versions of software involved during software development [89], which makes it hard to gather all information for the documentation [33].

To deal with those challenges, the automatic generation of code documentation is a common practice in computer science. In fact, software engineers already use several documentation generator in support to their project. For example, the well known Doxygen [87] is the de facto documentation generator standard tool for C++ source code.

Moreover, these generators achieve the documentation through the use of documented and/or undocumented source files. These source files are documented if they contain descriptive comments within the code (e.g. data layout definition, use, etc.). This approach allows developers the integration of the documentation as part of the process of software development, and to improve the readability of the source files [32, 47, 21]. The use of generator is an elegant way to met also the documentation updates issue [57].

In order to standardize the documentation in source files [16, 32], guidelines that define what and how to document should be provided to the developers. Then, the comments that meet the guidelines are used by the documentation generator to automatically generate documentation, while the comments that do not meet the guidelines and the undocumented entities are reported by the tool. However, Doxygen and the others well known documentation generators do not support PLC languages (e.g. IEC61131-3, etc.).

Supported by that, we propose to improve the usability and life cycle management through a complete, updated, and coherent documentation generated from documented and/or undocumented PLC source code. This documentation is produced by a documentation generator which is developed to support IEC61131-3 languages ([9]) and their implementation in Beckhoff environment (i.e. TwinCAT3 [12]). We believe that this approach can met part of the challenges of industrial automation [89] and highlight the importance of documentation in software automation.

4.2 D4T3

D4T3 is a documentation generator [6]. Nowadays, there are already different documentation generators [3], but to the best of our knowledge none is oriented/designed for PLC source code (e.g. ladder logic, structured text, etc.). Due to this lack, we decided to design and develop a documentation generator for TwinCAT3, which is one of our PLC platforms.

In order to not reinvent the wheel, we decided to take advantage of the well known Doxygen [87] as base for D4T3. Doxygen supports C++ sources, and also other programming languages (e.g. C#, Java, SystemC, etc.). Unfortunately, Doxygen does not support natively

any PLC programming language (e.g. IEC 61131-3 languages, etc.). In spite of this, it has four features that made us decide to use it as the basis for our tool:

Open source Doxygen is open source and it can be extended to add new features (e.g. support new programming languages, new entities, etc.).

Input filter Input filters are external programs that Doxygen invokes to filter each input file. Then, the output from the input filter is used as input by Doxygen.

Markdown Doxygen supports Markdown tool, which allows users to extend and improve their documentation from the source code. It is discussed briefly in [7] and in greater detail in [5].

Graphviz Graphviz [8] is a way of representing structural information (e.g. diagrams, graph, etc.) through the DOT language. It is supported by Doxygen and it can be used to add and generate more advanced diagrams and graphs in the documentation.

First, we figured out if we could use the open source feature to extend Doxygen in order to support natively the TwinCAT3, but we did not. In fact, while this seems like the most natural choice, it involves a lot of effort in the long term as it requires to maintain the extensions through the life cycle of Doxygen. An alternative is to fork Doxygen and extend the child fork, but it would result in the loss of Doxygen updates and/or in wasting energy to merge the updates with the child fork.

Due to these reasons, we decided to use the input filter feature instead of the open source one. Moreover, we used the Markdown and Graphviz to add PLC-world features (e.g. inputs, outputs, task, inout diagram, etc.) to the documentation generated. The use of those features and the resulting information flow of D4T3 is presented in Sub. 4.2.3.

4.2.1 Support TwinCAT3

We introduced the support of TwinCAT3 languages in Doxygen through the design and development of an input filter. The idea is to use the input filter as a TwinCAT3 translator. The input filter is invoked by Doxygen for each input file, and it provides the meaning of the PLC source code by a Doxygen supported target language.

In this approach, it is critical to choose the target language that covers most of the features of the source code (i.e. IEC 61131-3 2nd edition and IEC 61131-3 3rd edition). Tab. 4.1 [61] shows the cross-check of the languages and the features. This table helped us for the choice of the target language.

Table 4.1 Tab. 4.1 shows a comparison of the PLC code features and the Doxygen natively supported languages. The + means that the feature is supported by the language, the − means that it is not, and the ∼ means that it is approximately supported.

Language features	IEC 61131-3 2nd edt.	IEC 61131-3 3rd edt.	C++	Java	C#
Multi language	+	+	−	−	−
Procedural	+	+	+	−	+
Functional	+	+	+	−	+
Classes	−	+ (i.e. Function Block)	+	+	+
Methods	∼ (i.e. Actions)	+	+	+	+
Properties	−	+	−	−	+
Interfaces	−	+	−	+	+
Abstract classes	−	−	+	+	+
Polymorphism	−	+	+	+	+
Inheritance	−	+	+	+	+
Visibility	∼ (i.e. Variables)	+	+	+	+
Dynamic allocation	−	− (i.e. in TwinCAT3)	+	+	+

As shown in Tab. 4.1, C# is the language that covers most of the PLC languages features, hence it was chosen as target language.

Once the target language was identified, we mapped the entities of the source language with the entities of the target language (e.g. Function Blocks with Classes, References with Usings, etc.). Then, we designed and developed the input filter as a C# application object oriented.

Our input filter is organized in two phases:

- the first phase reads the code, identifies the entities, and use the mapping to dynamically create a tree containing the corresponding C# objects;
- the second phase scans the tree of C# objects, identifies them, and provides out the corresponding C# pseudo-code.

Therefore, Doxygen receives in the standard-input the C# pseudo-code and generates the documentation through them.

A benefit of this approach is to use the declarative part of the source code to get the information. In fact, this is written in a language common to all languages of IEC61131. One disadvantage is that of not being able to generate the call graph. Moreover, thanks to this organization, it is possible to extend the first phase objects to read and identify other languages (e.g. PLCopenXML) and/or platforms (e.g. Rockwell Automation). However, this

approach has the limitation of using a language for the PC world (i.e. C#) to describe one of the world PLC. In fact, C# can not cover all the features of the PLC world.

4.2.2 Support PLC world features

D4T3 uses the Markdown language to cover the features (i.e. entities) that were not directly coverable through the C# mapping. In fact, the input filter reads and identifies those entities, then a Markdown plain text that describes those features is is generated on-the-fly. Then, this plain text that had the same meaning of the entities is embedded in the C# generated pseudo-code. For example, the input/output/inout variables of TwinCAT3 Function Blocks are supported through this mechanism. We generate and add to the Function Block documentation page one paragraph for the inputs, one for the outputs, and one for the inouts through the Markdown.

Other examples of PLC features not covered by C# are the TwinCAT3 POU/DUT/etc. groups, and the PLC tasks. For those, we did also another action to support them. In fact, Doxygen recognizes and includes in its documentation only the entities/features contained in C#. Therefore, we had to create a specific layout Doxygen file to allow us to organize and include in the generated documentation the new entities/features. For example, we added the POU grouping page that groups the Function Blocks, Programs, etc. .

Additionally, we enhanced Doxygen documentation with a new diagram through Graphviz. The new diagram is called Interface Diagram and it is a type of static structure diagram that describes the set of input/output/inout variables of Function Blocks. It is composed by two kinds of entities:

- the Function Block is the main entity; it represents the Function Block itself; it is represented as a box with inside its name and it is centred in the diagram;
- the parameters are represented as labels of the parameter names, and are divided in three kinds;

  Input the input parameters are located above the Function Block, and are linked to the Function Block by an arrow from the parameter to the Function Block;

  Output the output parameters are located below the Function Block, and are linked to the Function Block by an arrow from the Function Block to the parameter;

  Inout finally, the inout parameters are located below the Function Block, and are linked to the Function Block by an arrow from the parameter to the Function Block and one from the Function Block to the parameter.

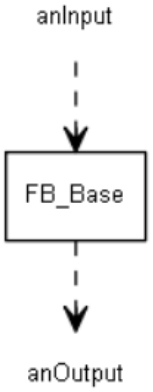

Fig. 4.1 shows an Interface Diagram of the Function Block FB_Base in Listing 16, which has an input variable called anInput and an output variable called anOutput.

This diagram serves as a brief overview of the Function Block interface. In fact, it allows users to understand the Function Block parameters interface without looking at the complete and detailed description in the documentation. Fig. 4.1 shows an example of an Interface Diagram representing a Function Block (i.e. FB_Base) with an input parameter (i.e. anInput) and an output parameter (i.e. anOutput).

In order to simplify the introduction and use of the new features, we created a specific configuration file. This file specifies to Doxygen the input filter, the new layout, and the file extensions of the PLC source code. Due to this file, the D4T3 users do not have to bother about remembering to link everything during the generation of the documentation.

4.2.3 The information flow

D4T3 information flow, its actors, their relationship, and the architecture of D4T3 is shown in Fig. 5.4, while the list of its actors is the following:

Input filter we designed and developed an input filter that allowed Doxygen to support the TwinCAT3 languages, the PLC world features, and a new diagram;

Layout we made a new documentation layout to add the PLC world features to the documentation provided by Doxygen;

Configuration file Doxygen is configured to D4T3 through a configuration file that we created. This configuration file sets the input filter, the layout, TwinCAT3 source files, and Graphviz to generate automatically the documentation. Additionally, this

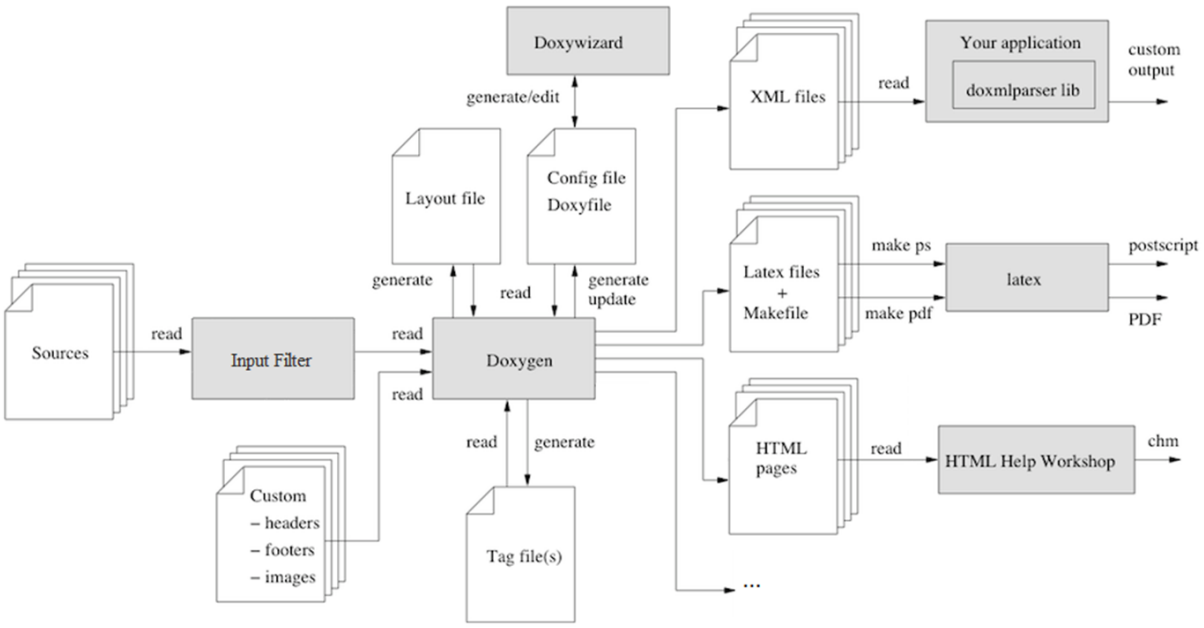

Fig. 4.2 shows D4T3 information flow [87] including the input filter, the configuration file, the layout, and their relationships.

configuration file can be edited by D4T3 users to provide additional Doxygen settings (e.g. entities visibility level in the documentation, output format, diagrams style, etc.).

4.2.4 Documenting the code

As documentation generator, D4T3 automatically generates the documentation from PLC source code in TwinCAT3. This source code can be:

Undocumented D4T3 extracts the project entities from the source code, and creates a documentation. The documentation is a reference manual of the entities, and it is very useful to find your way in large source projects. In fact, the documentation visualizes the relations among the project entities by means of dependency graphs, inheritance diagrams, collaborations diagrams, and function block interface diagram (i.e. a map of the inputs, outputs, and inouts).

Documented The project entities can be documented through comments. Those comments are used by D4T3 to describe the source code entities in the documentation. The resulting documentation is a reference manual which is much easier to keep consistent with the source code and also to provide to the users.

The possibility to have the undocumented entities in the documentation is very useful as overview of the source code. However, those entities should be considered incompletely documented as they do not provide any additional information than the ones that could not be found in the code. In order to simplify and solve the problem of incomplete documentation, D4T3 warns the user about the undocumented entities. Thanks to this, when the documentation is generated, Doxygen reports the entities that have to be documented.

Due to the code documentation, we introduced a set of guidelines to standardize the documentation in the code. While treat those guidelines in this chapter would be out of the scope, we briefly introduce them in order to simplify the understanding of D4T3 and the examples.

4.2.5 Guidelines

The guidelines are recommendations about how to document the code, but they become rules if D4T3 is used to generate the documentation.

Like Doxygen, the code is documented through comment blocks. However, for us a comment block is made up of two parts. The first part is called brief description, and it is mandatory. The second part is optional, and it is called detailed description. Having more than one brief and/or detailed descriptions is not allowed.

The brief description have to be placed as the first line of the comment block and, as the name suggests, it is a short one-lined description. The detailed description have to follow after an empty line from the brief description and provides a longer and more detailed documentation.

The comment blocks have to be used to describe entities (e.g. Function Block, Program, Variable, Struct, Method, Property, etc.). The entities are described in two ways, depending on the type of the entity;

- POUs , DUTs, GVLs, and IOs entities (e.g. Program, Function, Function Block, Interface, Struct, Property, Method, etc.) are described through a comment block before their declaration;

- VARs entities (i.e. instantiating entities) are described through a comment block after and in-line their declaration.

Next section presents a brief example of documenting the code following guidelines and a part of the resultant documentation.

Listing 16 shows another example of a whole entity should be described with comment block type.

```
(* FB_Base is the base FB that implements I_Interface.

FB_Base implements I_Interface and provides two methods: one protected and
    one public. When FB_Base is extended the Do_Public method must be
    overwrote. *)
FUNCTION_BLOCK FB_Base IMPLEMENTS I_Interface
VAR_INPUT
  anInput : BOOL; (* anInput is just an input.

  anInput is a FB_Base semaphore.*)
END_VAR
VAR_OUTPUT
  anOutput : BOOL; (* anOutput is just an output.

  anOutput is another FB_Base semaphore. *)
END_VAR
VAR //PUBLIC
  aMember : BOOL; (* aMember is just a member.

  aMember should be descripted in more details.*)
END_VAR
```

4.3 Examples

Listing 16 shows an example of a whole entity described through comment block and following the guidelines. While Fig. 4.5 shows how the Beckhoff ToolTip tool handles the comment block in TwinCAT3 environment, Fig.4.4 shows the corresponding documentation generated by the source code.

As stated in Sub. 4.2.5, if entities are not documented through comment blocks, then warnings are generated in the log during the generation as shown in Fig. 4.6.

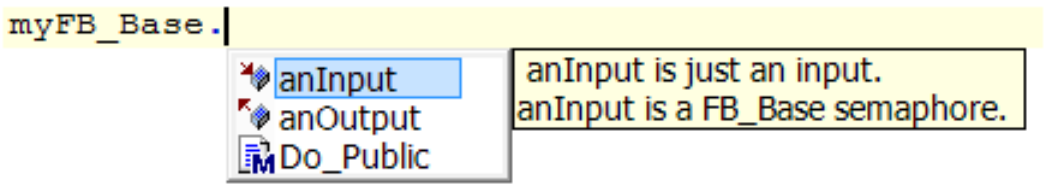

Fig. 4.3 shows Listing 16 result in Beckhoff TwinCAT3 IDE, which respecting the guidelines allow the users to generate the documentation and take advantage of the tool tip tool.

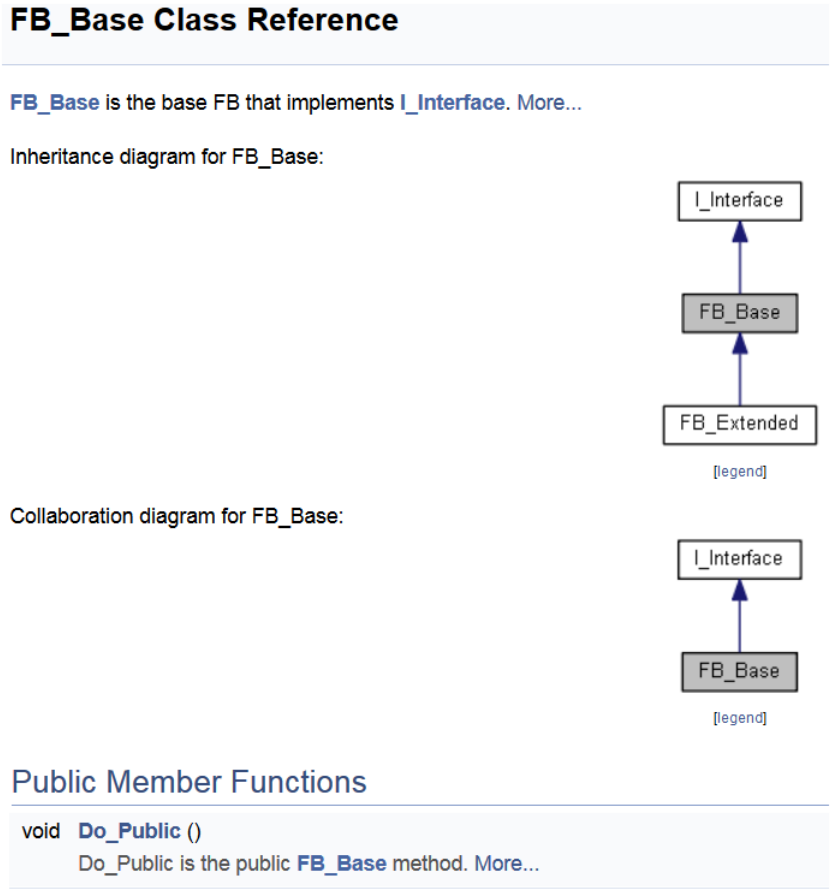

Fig. 4.4 shows part of the documentation obtained through D4T3 of Listing 16. While, its interface diagram is showed in Fig. 4.1.

This approach allows users to take advantage of two kind of documentation, the first one comes from the tips provided by the documented code from IDE. While, the second one comes from the reference manual generated from D4T3.

D4T3 example in [4] present a TwinCAT3 project and its documentation generated by D4T3. The project represents a “toy" example which gives a more complete view of D4T3 features in Sec. 4.2. This example presents the object oriented features through I_Interface which is implemented by FB_Base FB, FB_Base extended by FB_Extended Function Block, and entities visibility (i.e. private, protected, public, and internal). Additionally, this example presents some of the tradition PLC features of D4T3 by DoubleFB struct, Main program, and PlcTask task. Finally, the documentation presents also the collaboration, inheritance, and interface diagrams of example Function Blocks.

In order to access this D4T3 example, download the archive from [4]. Once the archive is uncompressed, the project is contained in the project folder and it may be opened and

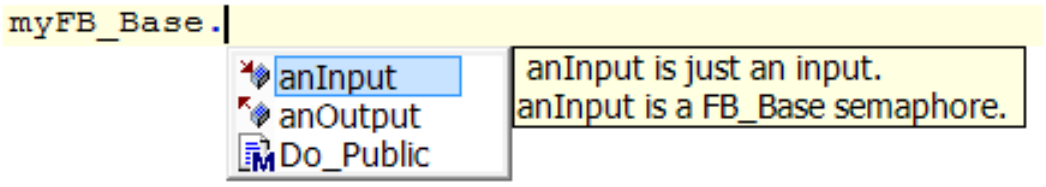

Fig. 4.5 shows Listing 16 result in Beckhoff TwinCAT3 IDE, which respecting the guidelines allow the users to generate the documentation and take advantage of the tool tip tool.

view from its solution by Beckhoff TwinCAT3. The project documentation is contained in the folder documentation. Due to handiness, the documentation was generated as Microsoft Compiled HTML Help (i.e. CHM).

4.4 Summary

We proposed to improve the usability and life cycle management through the use of documentation generators (i.e. D4T3). D4T3 documents the entities, their interface, their collaborations, and their inheritance from their IEC61131-3 and TwinCAT3 code.

D4T3 was used on ALOOA (i.e. Chap. 2) and other real-world projects of us. In one of the main ALOOA libraries, D4T3 generated the documentation of 406 entities (i.e. 233 Function Blocks, 6 interfaces, 144 structs, 2 programs, 2 GVLs, and 25 references). This documentation was used by the developer team of 10 people, and the verification and validation team of 3 people.

Basing on those experiences, we found that the resources dedicated to the documentation development were reduced, the big chunk of work usually done at the end of software development to document was avoided, the documentation was integrated in the software development process, and the documentation updates were aligned with the code versions. Moreover, the maintainability and intelligibility of the code was increased with the documentation availability.

It was remarked that the quality of the documentation was influenced by the quality of the comment in the code. Therefore, to partially fulfill part of this drawback a check on comments was introduced: the comments that met the guidelines are used by the documentation generator to automatically generate documentation, while the comments that do not meet the guidelines and the undocumented entities are reported by the tool in order to improve the software documentation.

We believe that this work may help groups of automation to reduce the resources given to the development of the documentation in order to use them elsewhere and that it can lead

to design a common and widespread standard to support system engineers and software engineers in automation.

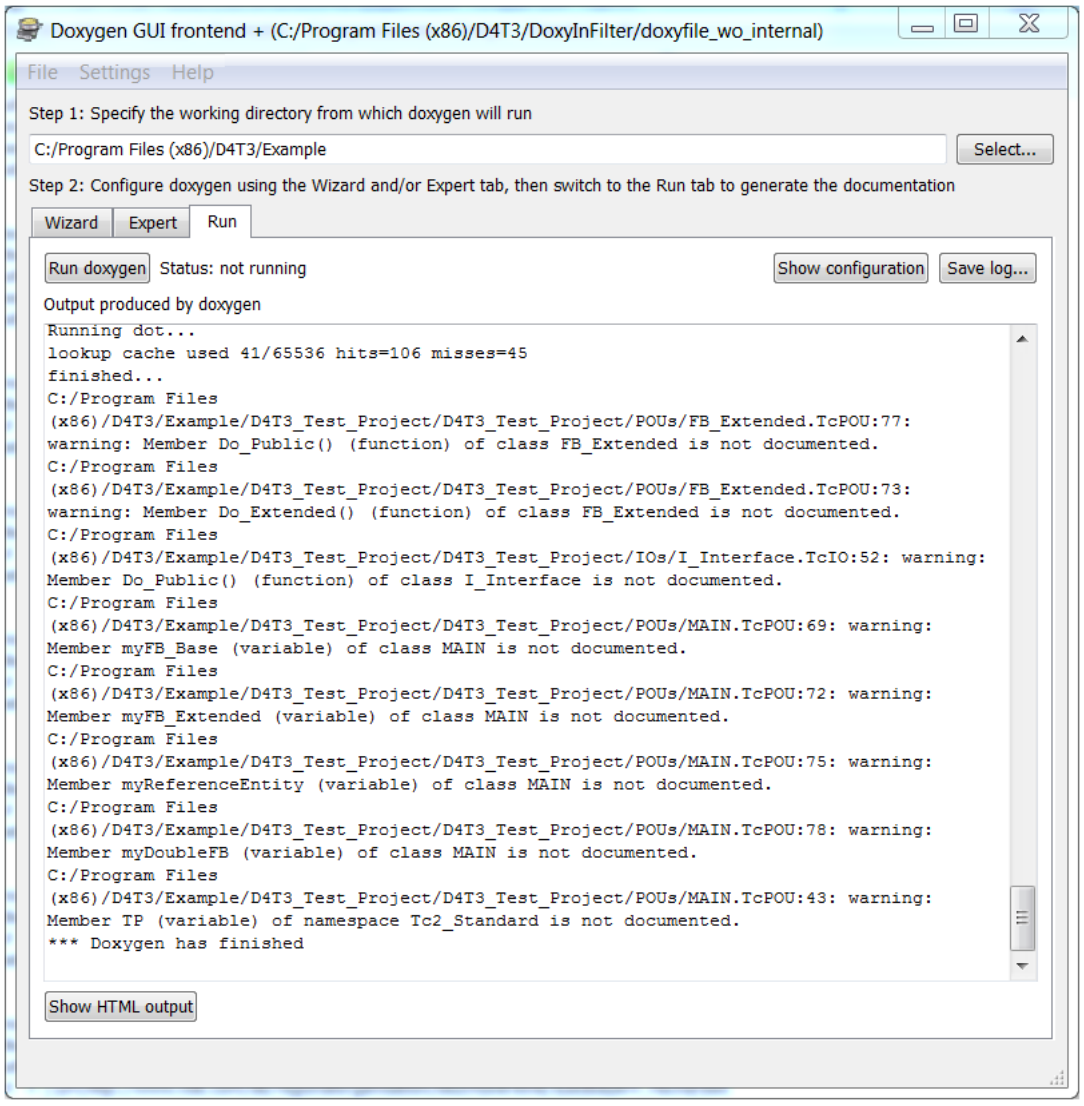

Fig. 4.6 shows the log obtained by the generation of a documentation. The log last sentences are the warning of the entities found in the code that were not documented.

Chapter 5

Components testing and formal verification

In this chapter we focus on the testing and Verification and Validation (i.e. V&V) aspects of software components. Specifically, we aim to support system engineers to improve the quality of the software by simplifying the testing, the V&V, and the method of specifying certain functionalities in the specifications. In fact, our goal is to provide the means to specify certain properties in the specification, then to generate the testing sequences to V&V those properties, and finally to evaluate the sequences results.

Our interest is due to the fact that testing and V&V are key factors of software development process [53] for industrial automation. The V&V consists of applying a set of experiments with several goals to a prototype system (e.g. from the correct operation to the measurement of performances).

The goals, the System Under Test (i.e. SUT), the means of specifications, the results evaluation, and the modalities are all factors that affect the difficulty of this task.

5.1 Related works

There are a number of practical approaches from software engineering, research projects, and publications oriented to deal with such aspects of the development process.

Leavens et al. [54] report ways that researchers might help further the goal of verified software. This report addresses researchers interested in program verification, specification languages, program generation, correctness by construction and programming languages.

Reflecting the goals of V&V and SUT, Krichen et al. [53, 52] propose a new framework for black-box (i.e. input and output signals) conformance testing of real-time systems

based on timed-automatas, a formal conformance relation (i.e. tioco), and two kinds of test (i.e.analog-clock and digital-clock tests). While the work opens a number of interesting perspective, authors state that there are still issues that have to be explored (i.e. real-time on-line test execution).

In industrial automation Younis et al. [101], Frey et al. [36], and Vyatkin [91] explore the modeling and verification done in this field. Also Roussel et al. [70] and Gourcuff et al. [41] explore verification methods of extrinsic properties (i.e. input and output signals), but focusing on binary signals.

Reflecting the means of specifications, Fisler et al. [30] and Chockler et al. [20] investigate on the use of timing diagrams (i.e. TDs) for capturing temporal and signal dependencies specifications.

In [93], Vyatkin et al. investigate on the visual specifications of input and output signals of the controller through timing diagrams, but focusing on binary signals.

Kormann et al. [50] proposed an approach for automated software testing of PLC application with a special focus on architectural concerns and semantics of UML sequences diagrams for test case specification. While the paper showed the possibility of test efficient test case description and execution for conformance verification purposes of PLC application, it does not take into account the dependencies among SUT signals value changes.

In [63], Preuße et al. present a framework to verify functional and non-functional properties of manufacturing control systems. The verification is done through a formal model of the plant based on Symbolic Timing Diagrams or Safety-Oriented Technical Language. The framework verifies that the whole sequence of states specified is completed or that the final state is reached.

Hametner et al. [42] present an agile and keyword-drive test approach for industrial automation. The authors define test scenarios and test cases from a basic set of keywords and test data, but the automated generation of test scenarios from specification documents was out of their scope.

To the best of our knowledge, we do not know works in automation tha deal with specifying dependencies of changes in value between signals and V&V them and generating from those specifications the testing sequence to V&V them in real-time. Moreover, by allowing system engineers to express these changes as events expressed no through time but by the dependencies from another event.

For our aims, we found particularly interesting the timing diagram logic of Fisler et al. in [30]. Fisler et al. use this logic to specify the behavior of a SUT using timing diagrams and checking algorithmically its behavior implementation. In our humble opinion, the most

important thing in this work is the chance to express dependencies between signals and their time constrains.

Based on this logic, we propose to use TDs to specify dependencies between signals of control software. These dependencies should allow system engineers to specify the changes in the value of the signals even without specifying the specific PLC cycle, but through other events and their time constraints.

We present Timing Diagram Testing for Verification & Validation (i.e. T4V), a testing system method based on TDs. This method generates test sequences to V&V industrial automation software from specifications. T4V is responsible of using the mathematical representation of target specifications to run V&V sequences in real-time on the PLC, providing to the SUT the right inputs at the right PLC cycle while monitoring, logging, and checking the outline. Then exporting the outline of the sequences to double-check the specifications on the PC.

We believe that T4V could help developers to simplify and meet the testing, V&V, and maintainability challenges and help PLC providers to understand the need of an environment for testing and V&V in the industrial automation.

5.2 T4V

Timing Diagram Testing for Verification & Validation (i.e. T4V) is an approach to V&V in industrial automation software. The main idea is to separate the task of V&V from the development of software in order to express the extrinsic properties of software in the software specification by TDs independent by the software implementation. Therefore, build and run in real-time test sequences to V&V the properties and finally evaluate the test results. Fig. 5.1 shows the overview of the whole approach.

In order to do that, T4V requires three basic elements: a language to express models of software; a language to assert the properties that software must satisfy; and a method to verify that the software meets the properties listed in the model.

5.2.1 Timing diagrams

TDs are a widespread mean to specify time constraints during design [30]. This is due to researchers who begun to formalize the notations of the timing diagram for practical use [30, 20, 15, 93] and information that allow us to express. In fact, the TD are used to express patterns of value changes on signals, precedence and synchronization relationships among changes, and timing constraints between changes.

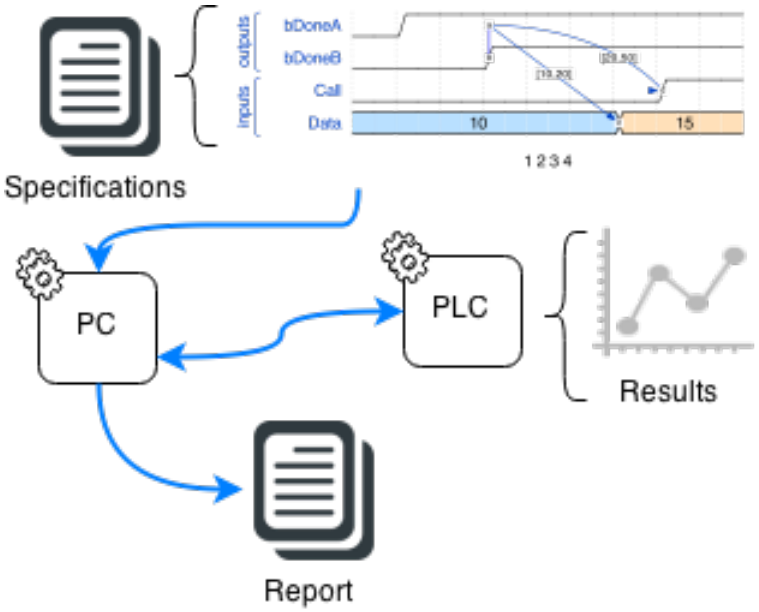

Fig. 5.1 shows the overview of T4V. The TD from specifications are downloaded into the PLC which runs the test and provides the results for the V&V report.

Fig. 5.2 shows a simple TD according to our notation, which derives from Fisler notation [30]. The TD shows variables bDoneA, bDoneB, Date, and Call as digital signals which value over time is depicted on the right of each variable. For example,TD shows bDoneA transaction from low to high and Data from 10 to 15. The terms high and low represent the logical value false and true, while the value 15 represents the digital value 15 (e.g. uint, int, etc.).

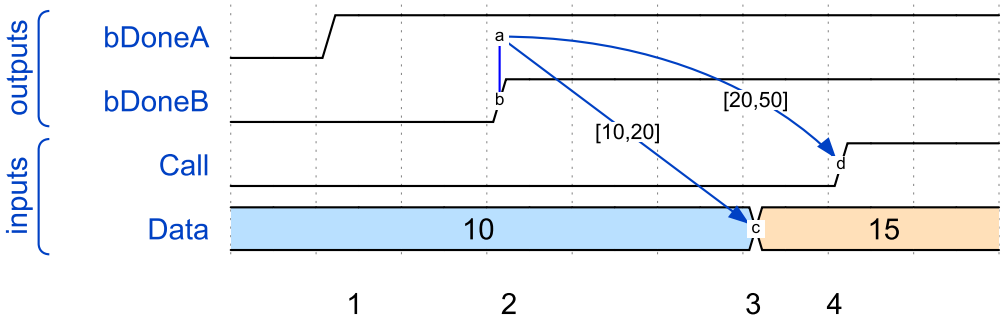

Fig. 5.2 shows a simple T4V timing diagram. The TD is composed by bDoneA and bDoneB as SUT output signals, while Data and Call are the SUT input signals. The labels below the signals waves represents the events of the value change transitions.

Vertical parallel line indicate synchronization of the levels or transitions crossed by the line (e.g. “a” is synchronized with “b”). A transition or a synchronization is defined by a

source event and at least a destination event (e.g. the synchronization between “a” and “b” events is the source of “c” and “d” events). The events and the arrows indicate temporal ordering between events, while the form $[l,u]$ on the arrows indicate the lower and upper bound of the transition time constraint. If the transition (i.e. arrow) does not indicate any time constraint, it indicates that the time constraint is $[0,\infty]$.

Valid bounds consist of natural numbers only and they represent the number of PLC cycle that the destination event must occur from the source event. Moreover, events are labeled with unique monotone increasing natural numbers. A well formed timing diagram has all time bounds expressions satisfiable.

5.2.2 Syntax

T4V TD syntax is derived from Fisler one [30], and is based on the concept that a timing diagram contains three components:

P is the ordered sequence of events. Each event can represent abstract moments of time at which events occur or specific PLC cycle;

N is a function from P and signal name to signal values, which is represented as a matrix with a column for each event and a row for each signal;

O is the ordered sequence that captures the temporal ordering, synchronization, and time bounds. Each element of O is composed by the coordinates (i.e. row, column) in N of the source event, the coordinates in N of the destination event, and its time constraint.

The tuple representation is unique up to isomorphism on the events [30].

For example, Fig. 5.2 can be described through the following $< P,N,O >$ tuple:

- $P = [1\ 2\ 3\ 4]$;
- $N = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 10 & 10 & 15 & 15 \end{bmatrix}$;
- $O = \begin{bmatrix} \begin{pmatrix} 1,2 \\ 2,2 \\ [0,0] \end{pmatrix} & \begin{pmatrix} 1,2 \\ 4,3 \\ [10,20] \end{pmatrix} & \begin{pmatrix} 1,2 \\ 3,4 \\ [20,50] \end{pmatrix} \end{bmatrix}$.

The first element of O represents “a” and “b” events synchronization. The second element and the third elements represent the dependency of specifically “c” and “d” events from “a” and “b” synchronization event.

5.2.3 Algorithmic verification

TDs specify the extrinsic properties of control software. The syntax converts TDs from the visual form to a tuple form. The tuple form keeps the information about properties specifications. These properties contain both input signals of the SUT and both output signals that are verified by the algorithm.

While the SUT generates output signals, input signals must be generated in the test environment. In our approach, the input signals can be generated by the algorithm verification. In fact, the algorithm exploits dependencies between events specified in the TD to generate a sequence of testing on-the-run that is within specifications. If the generated sequence does not meet the specifications, the test fails. We call this process “Generation”.

In Fig. 5.2 Generation generates the change of “Data” signal (i.e. event 3) after 10 cycles of PLC since “bDoneA” and “bDoneB” are both true (i.e. event 2). Instead, the value of “Call” is changed to true after 20 cycles from the event 2.

In order to perform Generation, we designed a verification algorithm that uses the TD information expressed in the tuples (i.e. $< P,N,O >$) to generate the testing sequence and also two queues to run it. These two queues are:

Queue (i.e. Q) is the queue containing the time value of events taken during the testing sequence. An important aspect of Generation is that if the temporal behavior of the SUT changes (e.g. different version), Q will contain different time values;

Watchlist (i.e. W) is the queue containing the O elements that are currently active in the processing. The active elements are composed of source events (i.e. guards) and an event destination (i.e. statement).

The algorithm is divided into three phases

Initialization is the starting phase in which Q is initialized to the copy of P and W to empty;

Processing is the phase performed each PLC cycle. Processing evaluates the elements of O to be added to W, checks whether active guards are met, checks the active time constraints, checks whether statements should be generated, and updates the elements of W and Q;

Finalization concludes the test sequence and if the sequence is not successful, it performs the actions associated with the failure of the test.

Fig. 5.3 summarize Generation Flowchart.

In addition to Generation, another verification algorithm can be provided. For example, it is possible to use the test results and the complete TD (i.e. not only signal dependencies) to validate the whole signals tracks of SUT. This process we call “Validation”.

In case inputs are not generated by the Generation, they must be generated by something included in the test. In this case, Generation checks value changes of all the signals and checks that they meet the specifications.

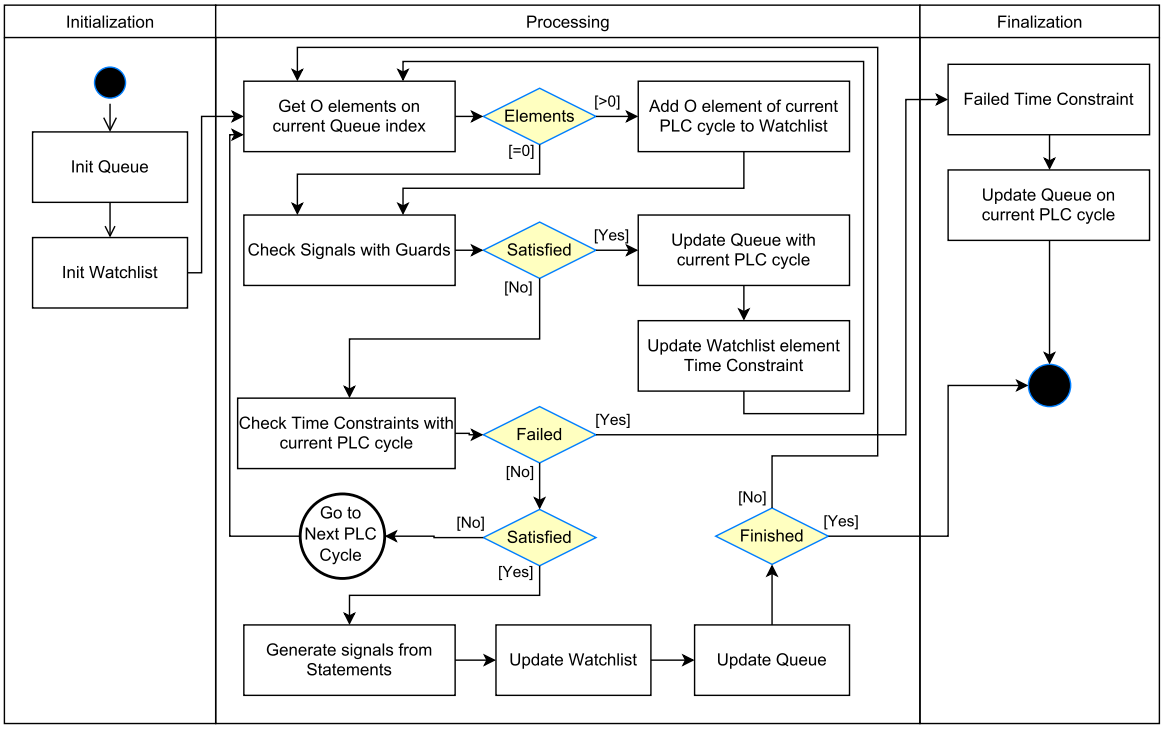

Fig. 5.3 shows the generation flowchart. The initialization initializes the Queue and the Watchlist; Processing run the SUT until all members of P are updated; Finalization finalizes the test sequence.

5.3 Implementation

This section introduces: T4V implementation issues, T4V Generation, and also presents the current implementation of T4V. For example, some of the issues that we encountered are “How should statement signals generated in real-time during testing?”, “How should N be

implemented to be independent from the type of signals treated?", and "How to keep track of changes in the value of the signals during the test? ".

In order to address these issues, we designed T4V with the architecture shown in Fig. 5.4. The architecture is implemented partly on PLC and partly on PC. We used this approach to

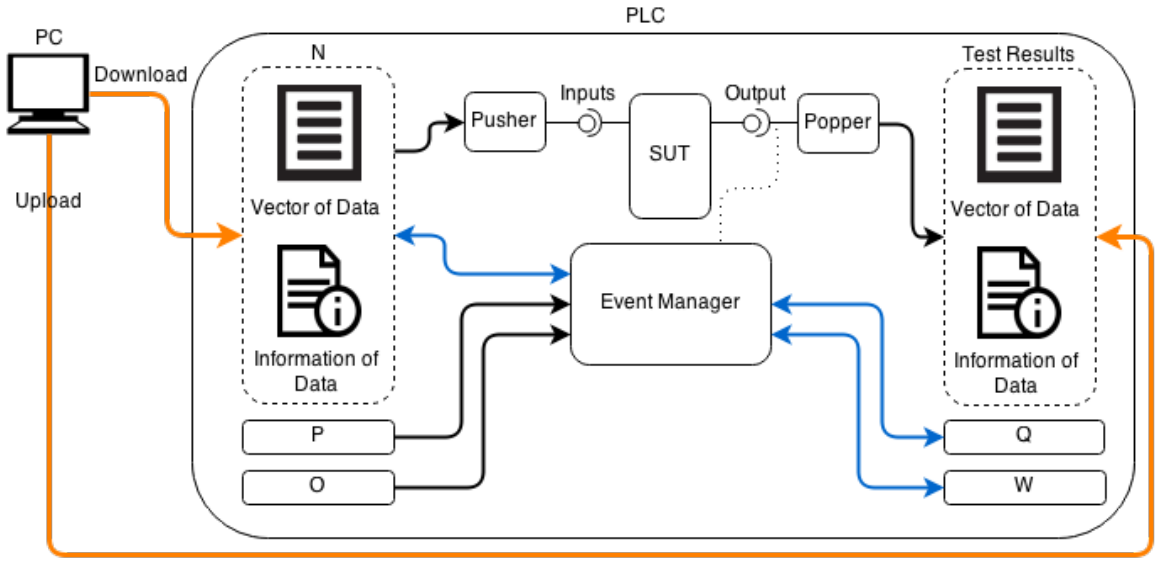

Fig. 5.4 shows the architecture of T4V. The black entities represents T4V entities: the PC that downloads the tuples, the tuples (i.e. N,P, and O) and the queues (i.e. Q and W), the pusher that pushes the input sequences at the right PLC cycle in the SUT input interface, the popper that places the test results from SUT output interface, and finally Event Manager that monitors the signals to updates the test sequence on-the-run.

allow the PLC to perform the SUT testing sequences in real-time, while the PC downloads the tuple before the test and uploads the results after testing. However, N has a form that depends on the type of the treated signals. In order to make T4V independent from the type of processed signals, we introduced the conversion of N to another format.

This conversion takes place when the PC downloads the tuple to the PLC and organize each element of N into two elements:

Data corresponds to the binary representation of the downloaded items. It is stored in a vector of bytes in PLC memory. However, this element is useless without the second element;

Index corresponds to information needed to recover the element of N from Data. It contains the name of the signal associated with the element, the size of the element, the address in the vector, and the event associated with the element.

Once N is converted, O and P are updated and downloaded to the PLC in order to be compatible with the new format of N. However, the interface of the SUT is not compatible with the tuple.

In order to associate the signal values contained in the new format to the interface of the SUT, we designed and developed two entities:

Pusher takes values from the tuple and push them into the SUT input interface. It takes charge of scrolling through the index to copy the binary of the proper element of N to SUT interface. For this purpose, Pusher requires the name of the signal, the current cycle of the PLC, and the target signal of SUT interface from where copy data.

Popper takes valued from the SUT output interface and log them. It takes charge of storing the value changes in SUT interface (e.g. SUT output). The storing uses the same principle used for N elements. For this purpose, Popper requires the name of the signal, the current PLC cycle, the value source in the SUT output interface, and the size of the source.

However, Pusher and Popper have the sole function of copying values from the interface to the memory and vice versa. Generation (Sec. 5.2.3) is performed by the Event Manager (Fig. 5.4). Event Manager performs Generation by reading the latest values tracked by Popper, updating the values in the tuple, and then updating W and ultimately Q.

5.4 Example

To clarify our approach of the verification and validation based on timing diagram of testing specification for industrial automation systems, we present an example based on the implementation for Beckhoff platform. This implementation is based on the third edition of IEC 61131-3. This allowed us to design and develop a library composed of the entities presented in Sec. Ref sec: implementation implemented as Function Blocks (FBs). These FBs (i.e. Event Manager, Pusher, and Popper) are generic elements which are independent from SUTs. Then, they are used in each test as they are. However, SUTs are the only part that may change in tests.

For example, let's consider the SUT in Fig. 5.5. The corresponding IEC61131-3 code is presented in List. 17. The example shows that the only entities that may change in PLC part are those related to the SUT (i.e. SUT and the variable assigned to its interface).

Fig. 5.6 shows the outcome obtained during out test. The testing sequence was generated from TD in Fig. 5.2, while as SUT we used a simple mock object.

Listing 17 shows an example of the T4V implementation for PLC Beckhoff with SUT is the SUT of this example. The declaration part declares the objects used by T4V (i.e. EventManager, Pusher, and Popper) to generate the testing sequences for the SUT. The implementation part implements the association testing framework. The test is enable from Flag, while Call, Data bDoneA, and bDoneB link SUT interface with T4V by Pusher and Popper. Finally, EventManager spins with the SUT monitoring, logging, and updating SUT signals.

```
PROGRAM MAIN
// DECLARATION PART
VAR
   myEM : EventManager; // Independent by the SUT.
   flag : BOOL;   // Semaphore for the Test run.
END_VAR
VAR // SUT Dependant.
   mySUT : SUT;  // SUT -> FB of the System Under Test.
   bDoneA : BOOL;  // Output1 from SUT.
   bDoneB : BOOL;  // Output2 from SUT.
   Call : BOOL;  // Input1 of SUT.
   Data : REAL;  // Input2 of SUT.
END_VAR
VAR
   myPusher : Pusher; // myPusher pushes the current cycle inputs into the
       SUT interface.
   myPopper : Popper; // myPopper pops the outputs of the current cycle into
       the SUT interface.
   plcCycle : UINT; // plcCycle represents the current plc cycle.
END_VAR

// IMPLEMENTATION PART
IF flag THEN
   // push inputs!
   myPusher(varName:='Call', currentCycle:=plcCycle, varTarget:=ADR(Call));
   myPusher(varName:='Data', currentCycle:=plcCycle, varTarget:=ADR(Data));
   // update interface and spin SUT.
   mySUT(input1:=Call, input2:=Data, output1=>bDoneA, output2=>bDoneB);
   // pop outputs!
   myPopper(varName:='bDoneA', currentCycle:=plcCycle,
       varTarget:=ADR(bDoneA), varTargetSize:= SIZEOF(bDoneA));
   myPopper(varName:='bDoneB', currentCycle:=plcCycle,
       varTarget:=ADR(bDoneB), varTargetSize:= SIZEOF(bDoneB));
   // Event Manager
   myEM( eventsUpdated=> myQ );
END_IF

plcCycle := plcCycle + 1; // Count!
```

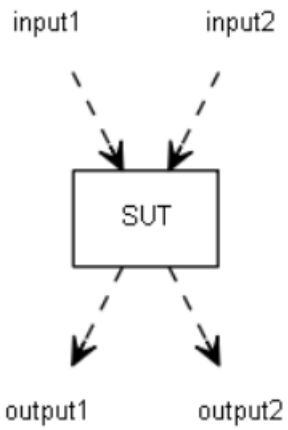

Fig. 5.5 shows the interface diagram of the SUT used as example. The SUT is composed by two inputs and two outputs. The system outputs are represented by bDoneA and bDoneBs, while Data and Call are the SUT input signals.

5.5 Summary

We presented T4V, its implementation, and an example based on IEC61131-3. T4V is our approach for V&V based on Timing Diagram specification for Industrial Automation Systems. Our approach allows users to specify the extrinsic properties of the control software and generate the testing sequence in real-time. The conversion from TD to tuple, its download, and the Validation are still manual, but the test execution and results verification are automated by the presented implementation. In fact, we were able to run testing sequences on a subset of ALOOA components as well.

We believe that T4V approach is a valuable testing method which could be used to improve the control software. Moreover, we believe that it could make V&V daily applicable in Industrial Automation in order to improve the development of the control software in quality.

However,T4V has still some limits:

- we are still looking for a TD editor tool that suits our needs;
- it is possible to add another feature to the whole process, for example verify the feasibility of the TDs before Generation;
- T4V could emulate signals generation, but it could not reproduce the dynamics of the system, which are critical to verify and validate a system instead of its components.

While we could not do anything for the first two limits during the doctorate, we worked on the third one. In fact, next chapter introduce our method based on Hardware-in-the-Loop to simulate part of and/ or the whole system for V&V.

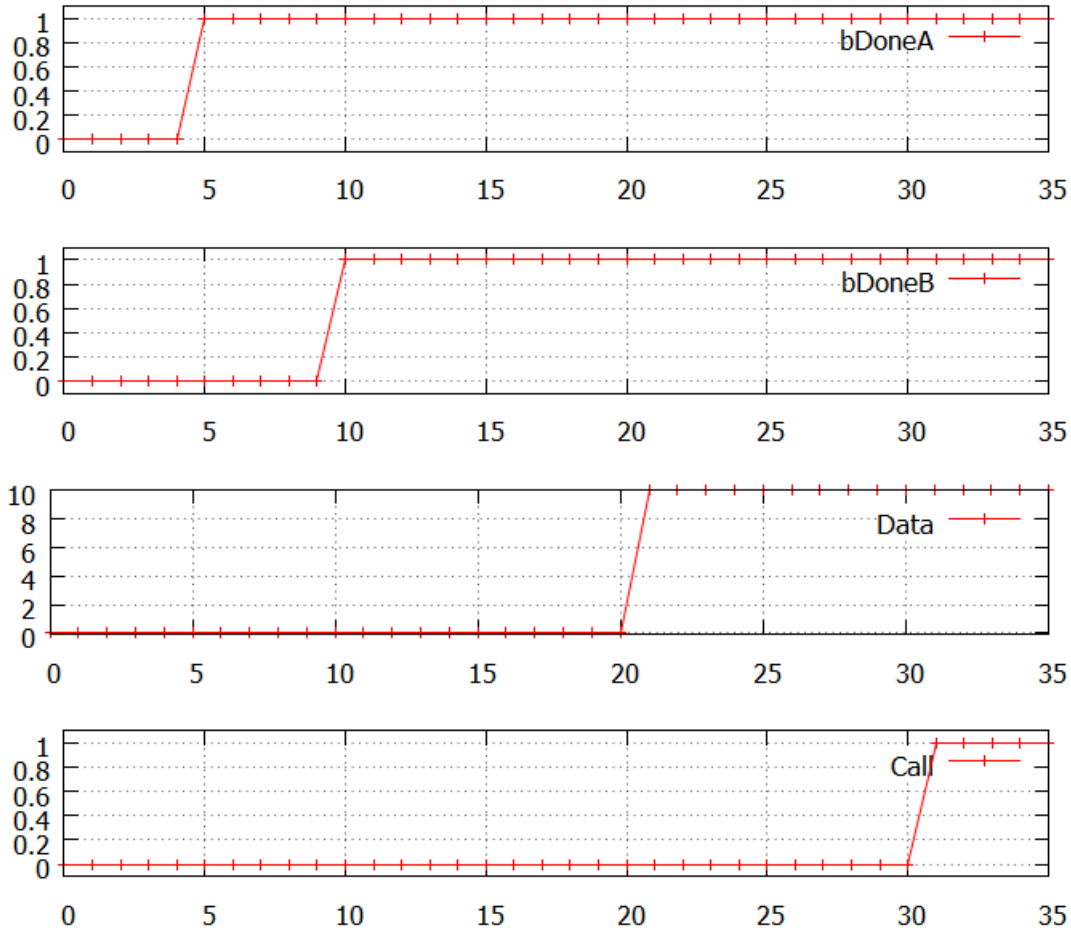

Fig. 5.6 shows the testing sequence results. Each graph represents one of the signals in the SUT interface, where The Y axis represents the PLC cycles, while the X axis represents the signal values.

Chapter 6

System verification & validation by modular development

The software verification and validation (i.e. V&V) of complex automatic machines has an important place in engineering. This is because software controls the systems, which makes it the most critical part. Therefore, software is responsible for the performance of the manufacturing production, the safety of system, and the human operators safety.

It is a common practice to develop, V&V the control system after the finalization of the mechanical part of the machine. In fact, the identification and treatment of particular working conditions (e.g. alarms management) is possible only with a working machine on which generate such conditions. This is also true when the control system is developed using engineering methodologies that are based on abstract modeling of the software (e.g. [75]).

Although this pratice is common, it wastes many resources (e.g. time, material, power, etc.). More efficient solutions are provided in the literature. For example, the Hardware-in-the-Loop (i.e. HIL) is a promising technique in this area which could be integrated in software development process.

HIL simulation [56] is a technique that connects the real controller with a simulated model of the machine for V&V. This model allows system engineers (i.e. SE) to save many resources. Additionally, the HIL simulation provides an effective V&V platform by adding the complexity of the plant under control to the test platform. This complexity is included in tests and developments by adding a mathematical representation of all related system dynamics. Even though HIL is an efficient V&V practice, it has some difficulties.

In fact, HIL difficulties concern the following:

- developing a validated model of the target real plant;

- developing an interface system between the controller and the simulation of the plant model;
- dealing with the computational power required by the simulation to respect real-time;
- the difficulties that concern in the complexity of the control.

In this chapter, we discuss the Modular Hardware-in-the-Loop (i.e. MHIL), which we believe could help to overcome such difficulties. Additionally, we introduce the Machine Modular Development (i.e. MMD), which could be used by developers and SEs to simplify the coupling among software and simulated plant.

6.1 Related works

The current literature is rich of application examples of HIL simulation to various fields within industry and personal services. Generally, the papers describe:

1. the model development of the real system;
2. the control hardware set-up.

The main applications are related to safety critical systems. Rankin and Jiang [69] described a HIL simulation platform to perform functionality tests for the control system of nuclear power. Another field of application to safety are in avionics [49, 73], unmanned aerial vehicles [17, 38], and space missions [13, 25]

Other applications are related to the development of large scale systems, such as smart-grids. Several authors [100, 14, 45] proposed simulation environments for “hardware-in-the-loop” or “software-in-the-loop” validation of distributed controls in Smart Grid; in [59] a HIL simulation of a wind power system is described.

A third important field of application is in the industry, in particular in the verification and validation of mechatronic machine and components. For example, papers [40, 62] deal with the automotive applications, while applications in mechatronics systems are presented in [22, 29, 79].

HIL is also an important branch within the framework of SEs. In fact, HIL technology can be considered as an important part of the Model Based System Engineering (i.e. MBSE) process. In this area, [48] describes an implementation with a model-based representation that encompasses real system HIL components to verify the effectiveness and capability of the development platform.

To the best of our knowledge there are works in the literature that deal with the effectiveness and capability verification of the development platform [48], but there is not any work that deals with the system complexity.

6.2 Modular hardware-in-the-loop

Hardware-in-the-loop approach models the target system with the purpose of testing the controller. Therefore, it's a matter of fact that the biggest issue of HIL is the availability of a reliable and tractable model of the real system.

Since our concern is to test the controller for the physical system, a reliable mathematical model should repeat the physical behavior from the controller point of view.

A second aspect which should be considered is the model complexity. In fact, an automatic machine is a very complex system, and it is unrealistic to pretend to dominate the complexity of such a model taken as a whole. Therefore, we can define a tractable model as system model with a level of complexity which can be successfully managed to obtain useful information from its conceptual analysis and simulation results.

Following a modular development approach, a machine is constructed using a functional decomposition in mechatronic parts. Each of these parts is formed by physical components (e.g. mechanical, chemical, etc.) tightly coupled with an electrical control system to implement a particular function (e.g. cutting device, pick and place gripper, etc.) [94, 75].

As discussed in [75], this division decreases the complexity of the whole system into simpler and more tractable subsystems. The division allows the subsystems to act only on their internal variables, since it hides the internal behavior of each subsystem to the others. Instead, the external behavior of each subsystem is visible to other subsystems through communications among them.

The resulting system is a set of mechatronic subsystems (i.e. modules) that can be simulated alone as single components.

The Communications among subsystems are described through the exchange of:

Information it represent the control signals between the control module of each mechatronic object.

Material it is the material that is exchanged between physical modules (e.g. a piece of product which should be printed with the current date and then packaged by two different modules).

energy it is the power that each module exchange (e.g. an mechanical power which is exchanged between an electrical motor and a mechanical linkage).

These exchanges are done through interfaces that are defined during the system designing (e.g. the module sensors used by the module control). The definition of these interfaces makes the exchange interfaces specific and well defined. Consequently, communication interfaces make every module usable by knowing its external behavior, and thus makes modules interchangeable and reusable. Furthermore, the use of communication interfaces allows the control software to prevent cross-relationship side-effects, and the physical model to simulate modules as stand-alone systems.

The modular hardware-in-the-loop (i.e. MHIL) allows each module of the model to describe the behavior of the physical subsystem of the system. Consequently, the degree of fineness of the module description depends on the specifications and capacity of the simulation. Several module models of the same subsystem can be made as library. This is done to describe different features, to adapt the model to the computational power of the simulation platform, to facilitate the reuse, and to simplify the creation of new models. This approach fits with the philosophy behind ALOOA (i.e. Chap. 2): using a set of single components which could be integrated together and/or extended to model something more. However, in order to correctly couple simulated modules with its control software it is, software must be organized in modules as well.

6.3 Machine modular development

ALOOA organizes software components to abstract control logic from hardware details, but it does not treat the organization of control logic into modules. Therefore, we introduce the Machine modular development (i.e. MMD). MMD takes advantage of ALOOA and our already described methods to allow developers and SEs to:

- divide the system under development into modules by a conceptual model typical of software engineering;
- use MHIL to V&V the machine from modules to the while plant.

The conceptual model of software development allows developers the identification of mechatronic parts during the software design by using object oriented software engineering tools. These tools will simplify the development of the control software. One of the tools of software engineering that can simplify further development is the Unified Modeling Language (i.e. UML).

UML is universally accepted as the reference for the development of a software model and allows:

Sketch quick and informal;

Blueprint they are useful to understand a project with no documentation good enough;

Programming language tools that allows developers to translate diagrams into executable code.

Extending the concepts of applying UML to the modular development of mechatronic objects (e.g. [94]) to the modular development of the physical system model, we identify the system modules as classes using UML and the OOP:

UML Class diagram describes the static structure of the machine and the control software of the system through the system classes and relations between them;

UML State-chart diagram describes the behavior of each module, defining the states, transactions, events, procedures that define the reactive behavior inside the class. Thus, the diagram describes the reactions of the form to the stimuli (e.g. sensors, timers, Human Machine Interface, alarms);

UML Sequence diagram describes the interactions of the module with other modules and their order. The definition of the interactions serves to describe the communication interfaces intra-modular;

Signal list describes the communication interfaces of each module (input / output variables).

The modules identification and description of the control software allows developers the creation of standard modules libraries as in Section 6.2. Additionally, software engineering methods and OOP allow developers to use and design design patterns to handle such modules. The complete process is shown in Fig. 6.1, and its steps are described below:

1. Machine division This first step is the development of a model of the machine. Usually, this step is started from a rough model of the complete machine. Starting from this, the models of the physical modules are developed. It is important to outline that is a model of the mechatronic object of the machine, which comprises both the model of the control system and the model of the physical part.

2. Model and Code libraries Models are described using a tool, which should allow developers to develop and simulate the whole system. Many tools can be used to support this modular development. For example, Simulink and SimScape toolboxes of MATLAB allow the development of such models. Once the models are developed, they could be

organized as set of libraries. Software code is designed and developed to encapsulate modules control logic. Therefore, code modules could be organized as set of libraries as well.

3. Module V&V Single modules are tested using the modular hardware-in-the-loop technique: each module control code (i.e. module code) is matched with the simulated module models.

4. Models integration Model and code Modules are collected all together to perform the machine V&V, which finalizes the tests on the complete machine.

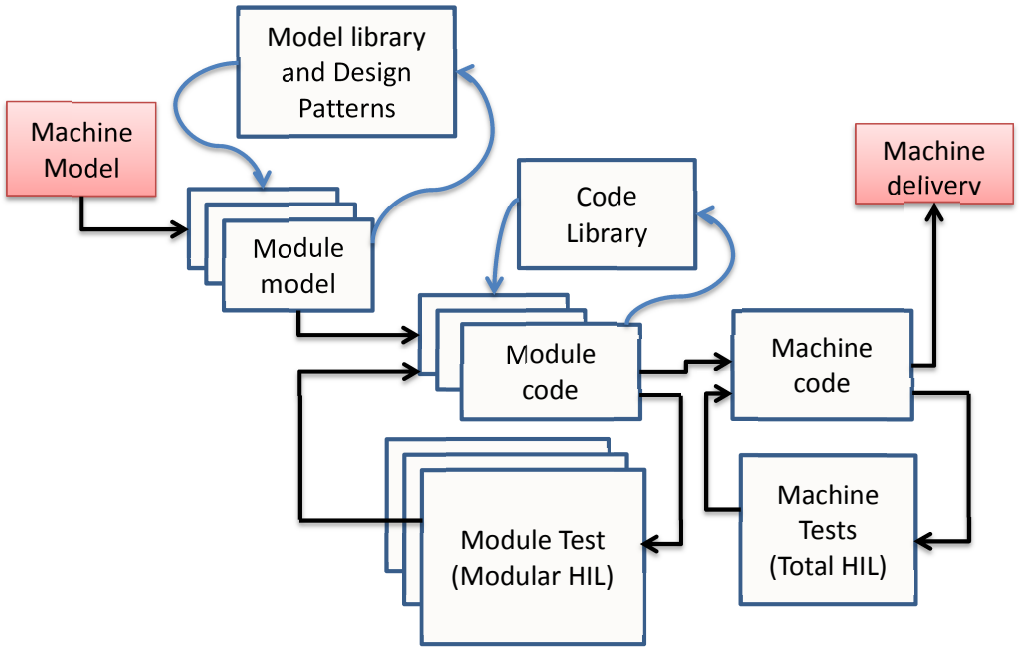

Fig. 6.1 shows the MMD HIL process. The machine is decomposed in modules models, and its control software in modules code which are aggregated as model library and code library. Therefore, each module could be tested with its control code, then it can be integrated with other modules and tested again with the related control code. This process could go on untill the final machine model is obtained, which can be tested with its control code.

Tthe first step of the modular development approach divides the whole system into modules. This allows developers to build models of the system designed to V&V certain features of the control software (see Section 6.2).

The V&V of such particular features is possible by integrating only the intended modules of the model and modules of the software. For example, if there are many modules, SEs can create module models with two degrees of fineness:fine and coarse. The fine is used to test in

detail the control system of the module. The coarse is used when modules are aggregated in portions of the system until you get the entire system. Consequently, the approach permits SEs to V&V individual modules, groups of modules, and finally the whole control system *incrementally*.

We think that these facts simplify the V&V procedures of the control system.

6.4 An example

During the beginning of the doctorate, we used MHIL approach with one of our industrial partner on a mixer plant. While we used MHIL and MMD in such experience, we did not use ALOOA due to platform limitations. Hence, we decided to present as example that experience on an high-level (i.e. conceptual example).

The mixer plant mixes three different liquids (Alpha, Beta, and Gamma). Each liquid is contained in a separate tank, and the mixing is performed in pairs of liquids (e.g. at first Alpha with Beta, then Alpha-Beta with Gamma). Additionally, the mixing is done respecting a note and specific recipe. The system has the components (i.e. sensors and actuators) to check the status of the mixing (e.g. the amount of liquid used, the amount of liquids available, pressures, and the amount of product obtained through sensors).

The system was divided into four modules: Alpha liquid module, Beta liquid module, Alpha-Beta and Gamma liquid module, and Alpha-Beta-Gamma liquid module.

Once the modules were identified, we identified the mechanical and electrical parts associated with each module. Therefore, we knew the components associated to each module of the control software. Finally, we developed models and code modules in order to test the functionality of each module of the system with the appropriate control module.

However, to test the software with the models we needed a framework which allowed to connect simulations to control software. Hence, we designed and implemented a framework to exploit MHIL approach, which is shown in Fig. 6.2

The framework contains a simulation part that was implemented using MATLAB and SimScape tools to:

- simulate the modular model;
- develop and build a library of object-oriented components, which are used to model the mechanical and electrical components identified (e.g. tank, valve, pump, flux sensor, etc.);

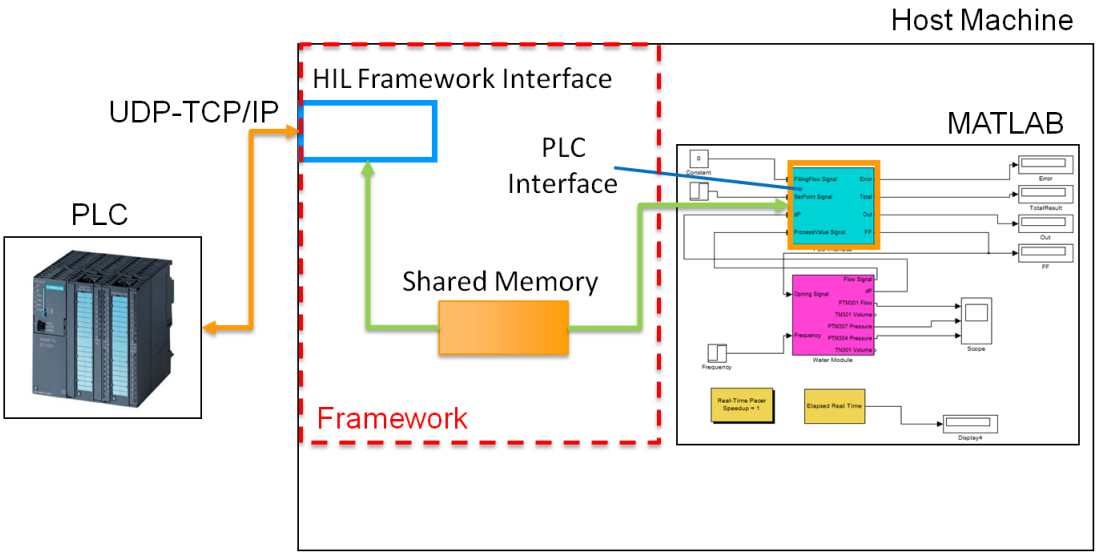

Fig. 6.2 shows the MHIL framework schema. ScimScape toolbox of MATLAB was used to develop modules and machine models. Additionally, an interface from the simulation to the PLC was developed. Such interface include a part in the simulation and another one on the host machine. The first part is in charge of reading and writing data from the simulation to outside and vice-versa. The second part is in charge of reading and writing data from the host machine to the PLC memory and vice versa.

- develop a set of components required for the HIL, which allows simulations to communicate with the world outside (i.e. a data reader and writer component from a simulation to a PLC in real-time);

- develop a set of synchronization components that allows the simulation to synchronize its clock to the controller clock.

This simulation part works together with the HIL part, which is a custom tool that we developed to exchange data among the PLC and the simulations.

During that experience, we used a Siemens PLC as a platform to run the software control of the example, and an Intel i7 Quad-Core as the framework platform.

You may feel that the framework platform had enough computational power to run a fine model of the whole system respecting the real-time constraints. Nevertheless, it was not. We had to V&V the control system on individual fine module models, and then aggregate the coarse models.

In order to V&V the model modules, we confronted experimental patterns ran on the real plant with the simulated module ones. Then, we tested the plant modules using the HIL framework.For example, figure 6.3 shows a real plant acquisition, while figure 6.4 shows a

simulation result. From the comparison between the two set of experimental and simulative tests, we concluded that the qualitative plant behavior is well captured by the model.

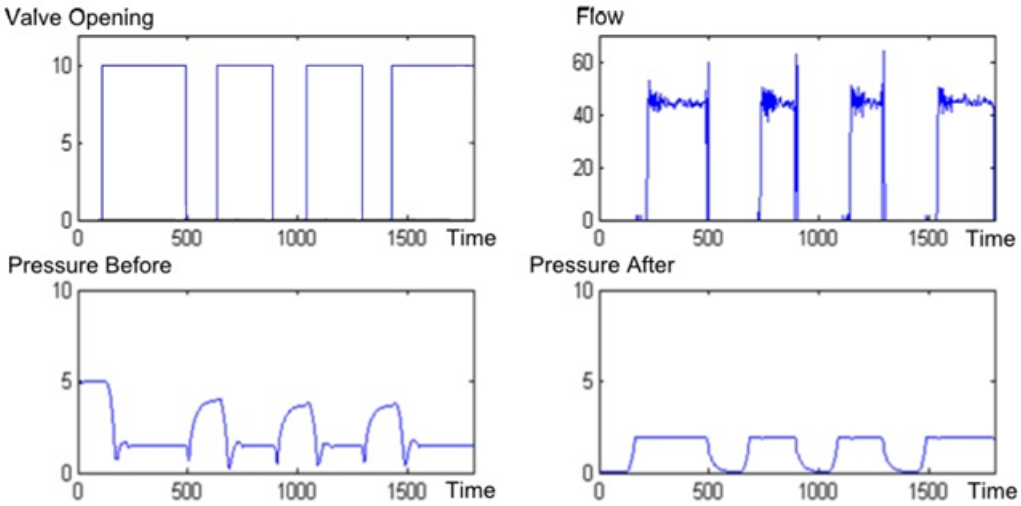

Fig. 6.3 A snapshot of the validation tests.

6.5 Summary

We proposed modular hardware-in-the-loop approach as verification & validation. Therefore, we introduced also our machine modular development method to handle the MHIL in machine software.

Our approaches allowed us to divide the system into modules, and thus dividing the complexity of the system to simplify the verification & validation process. Therefore, we developed the models of such modules and submodules which were simpler and more tractable subsystems than the whole systems. Finally, we implemented a framework to V&V the control system incrementally through HIL.

While this approach covers the handling of the simulations for the verification & validation, it is not always possible to divide a plant into single and independent modules. However, such cases are rare, and we believe that our approach could be used in most of real-world projects.

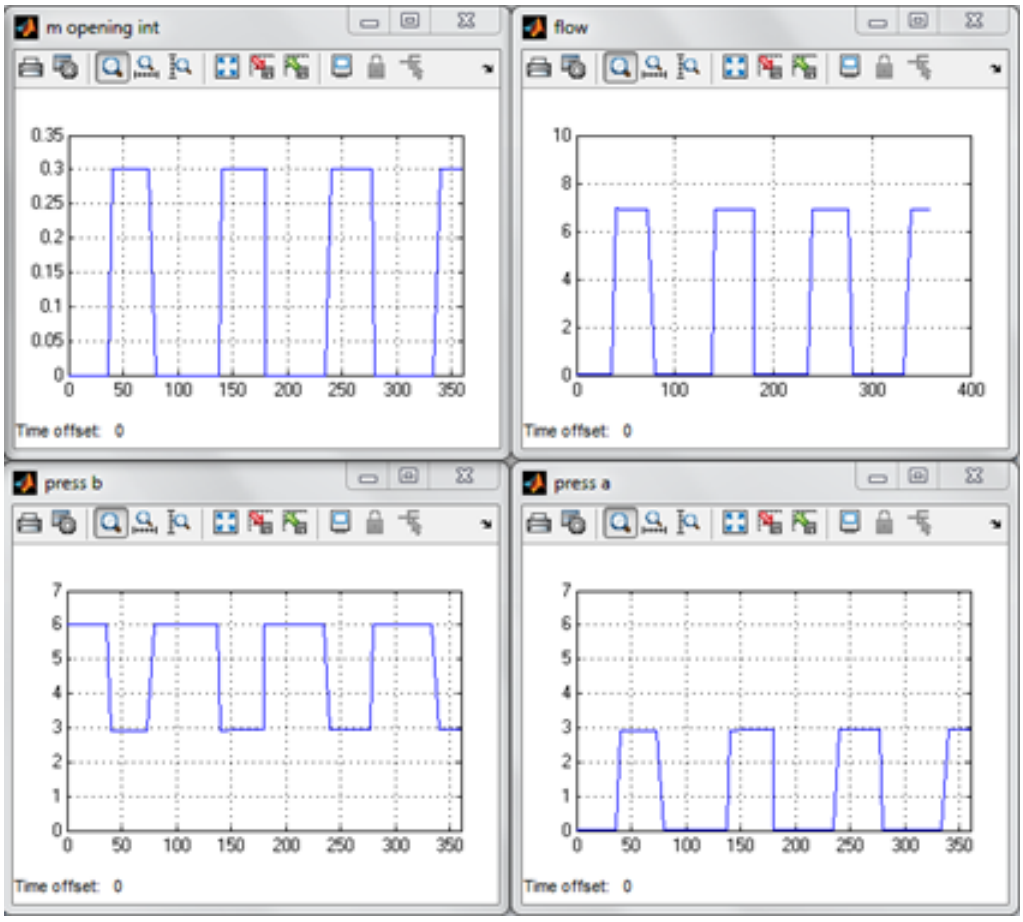

Fig. 6.4 Results from simulation.

Chapter 7

Conclusion

Current challanges in automation software development may be faced with the introduction of specific methods. These methods can be both new and/or adaptations from other domains (e.g. computer science), but the introduction of such methods is necessary to address these challenges. However, we believe that more than focusing on the introduction of individual methods, it is necessary to organize the development process in order to create a strategy of methods. This strategy and its methods could allow to face the challenges in a more complete way.

For this purpose, we believe that the methods we have developed and presented allow developers and software engineers to improve the development of software in automation:

- MMD defines a strategy that allows to decompose the system into modules and generate software specifications using the tools provided from engineering software. This decomposition allows developers to split the total complexity of the system through modules and sub-modules which can be developed as subsystems. In addition, the specifications of the system and its modules allow system engineers to specify hardware parts used by each module, its control logic, and its reactive behavior;
- ALOOA improves software flexibility by implementing control logics from software specification and managing devices in abstraction from hardware platform. These control logics utilize software components implemented using OOP concepts to increase reusability and maintainability. Finally, both logics and software components may be realized via IHSM and/or PUSDP the reactive behaviors described in specification state-charts;
- ALOOA components may be tested and validated using MMD specifications through T4V approach. In fact, T4V allows developers to use specification timing diagrams to generate test sequences and perform the verification & validation of the results.

- Control logic of modules and system may be verified & validated through MHIL. MHIL allows verification & validation of modules and systema through modules and system simulations made of physical plant models. These models should be based on MMD division as well.

- Finally, the software developed should be documented in all its versions through documentation generator tools like D4T3. In fact, this approach allows developers to increase the usability of the software developed and manage its life cycle.

While a lot has been done, there is still room for improvements.

7.1 Outstanding challenges

A number of challenges still exist, most notably the standardization of programming languages of PLC.

Currently a part of our methods is based on design patterns that use OOP introduced by the standard IEC61131-3. However, only some of the producers currently is compliant to the standard, although the number is increasing. We believe that it might be interesting to investigate changes to the current design patterns and/or to design new ones for platforms that are not IEC61131-3 compliant.

Moreover, our methods involve the use of tools to support developers and system engineers. Some of these tools were made during the doctorate, but some are still missing (e.g. tools to generate the timing diagrams). However, it would be too ambitious to think that we could make a set of comprehensive tools during one doctorate. This lack, however, allows room to a complete investigation of tools that could support developers and system engineers during software development process. For example, an investigation on a tool which should be able to generate the timing diagrams of specifications and which allows to generate tests or simply the test tuples.

References

[1] Beckhoff ax5000 user manual 2^{nd} generation.

[2] Beckhoff xts linear transport system.

[3] Comparison of documentation generators.

[4] D4T3 example archive - project and documentation.

[5] Daring fireball: Markdown.

[6] Documentation generator.

[7] Doxygen manual: Markdown support.

[8] Graphviz - graph visualization software.

[9] IEC61131-3 - third edition.

[10] Plcopen motion control library.

[11] Plcopen motion control: Reducing development time and cost with standardized motion programming.

[12] TwinCAT3 - BECKHOFF.

[13] K.S. Badaruddin, J.C. Hernandez, and J.M. Brown. The importance of hardware-in-the-loop testing to the cassini mission to saturn. In *Aerospace Conference, 2007 IEEE*, pages 1–9, 2007.

[14] A. Benigni, Junqi Liu, A. Helmedag, Weilin Li, C. Molitor, B. Schafer, and A. Monti. An advanced real-time simulation laboratory for future grid studies: Current state and future development. In *Complexity in Engineering (COMPENG), 2012*, pages 1–6, 2012.

[15] Gustavo Bouzon, Valeriy Vyatkin, and Hans Michael Hanisch. Timing diagram specifications in modular modelling of industrial automation systems. In *IFAC World Congress*, 2005.

[16] Lionel C Briand. Software documentation: how much is enough? In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 13–15. IEEE, 2003.

[17] Guowei Cai, Ben M. Chen, Tong H. Lee, and Miaobo Dong. Design and implementation of a hardware-in-the-loop simulation system for small-scale {UAV} helicopters. *Mechatronics*, 19(7):1057 – 1066, 2009. Special Issue on Hardware-in-the-loop simulation.

[18] Isidro Calvo, Federico Pérez, Ismael Etxeberria, and Guadalupe Morán. Control communications with dds using iec61499 service interface function blocks. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–4. IEEE, 2010.

[19] Jacques Camerini, Antonio Chauvet, and Michael Brill. Interface for distributed automation: Ida. In *Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on*, volume 2, pages 515–518. IEEE, 2001.

[20] Hana Chockler and Kathi Fisler. Temporal modalities for concisely capturing timing diagrams. In *Correct Hardware Design and Verification Methods*, pages 176–190. Springer, 2005.

[21] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75. ACM, 2005.

[22] M. Deppe, M. Robrecht, M. Zanella, and W. Hardt. Rapid prototyping of real-time control laws for complex mechatronic systems. In *Rapid System Prototyping, 12th International Workshop on, 2001.*, pages 188–193, 2001.

[23] Wei Ding, Peng Liang, Antony Tang, and Hans Van Vliet. Knowledge-based approaches in software documentation: A systematic literature review. *Information and Software Technology*, 56(6):545–567, 2014.

[24] Bruce Powel Douglass. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Newnes, Newton, MA, USA, 1st edition, 2010.

[25] Jens Eickhoff. *Simulating Spacecraft Systems*. Springer, 2010.

[26] Elisabet Estévez, Marga Marcos, and Darío Orive. Automatic generation of plc automation projects from component-based models. *The International Journal of Advanced Manufacturing Technology*, 35(5-6):527–540, 2007.

[27] E. Faldella, A. Paoli, A. Tilli, M. Sartini, and D. Guidi. Architectural design patterns for logic control of manufacturing systems: The generalized device. In *Information, Communication and Automation Technologies, 2009. ICAT 2009. XXII International Symposium on*, pages 1–7, Oct 2009.

[28] Eugenio Faldella, Andrea Paoli, Andrea Tilli, Matteo Sartini, and Daniele Guidi. Architectural design patterns for logic control of manufacturing systems: the generalized device. In *Information, Communication and Automation Technologies, 2009. ICAT 2009. XXII International Symposium on*, pages 1–7. IEEE, 2009.

[29] C. Fantuzzi, R. Panciroli, and M. Gargiulo. Hardware in the loop simulation for distributed automation systems. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–6, 2012.

[30] Kathi Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language and Information*, 8(3):323–361, 1999.

[31] Andrew Forward. *Software Documentation–Building and Maintaining Artefacts of Communication*. PhD thesis, University of Ottawa, 2002.

[32] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33. ACM, 2002.

[33] JA Forward and Timothy C Lethbridge. Qualities of relevant software documentation: an industrial study. *University of Ottawa, Ottawa, Ontario, Canada*, 2002.

[34] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[35] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

[36] Georg Frey and Lothar Litz. Formal methods in plc programming. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 2431–2436. IEEE, 2000.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[38] N.R. Gans, W.E. Dixon, R. Lind, and A. Kurdila. A hardware in the loop simulation platform for vision-based control of unmanned air vehicles. *Mechatronics*, 19(7):1043 – 1056, 2009. Special Issue on Hardware-in-the-loop simulation.

[39] Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682, 2015.

[40] O.J. Gietelink, J. Ploeg, B. De Schutter, and M. Verhaegen. Development of a driver information and warning system with vehicle hardware-in-the-loop simulations. *Mechatronics*, 19(7):1091 – 1104, 2009. Special Issue on Hardware-in-the-loop simulation.

[41] Vincent Gourcuff, Olivier De Smet, and J Faure. Efficient representation for formal verification of plc programs. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 182–187. IEEE, 2006.

[42] Reinhard Hametner, Dietmar Winkler, and Alois Zoitl. Agile testing concepts based on keyword-driven testing for industrial automation systems. In *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 3727–3732. IEEE, 2012.

[43] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[44] I.E.C. IEC 61131-3. Programmable Controllers - Part 3: Programming Languages (2nd Edition). Final Draft International Standard (FDIS), 2002.

[45] Z.R. Ivanovic, E.M. Adzdic, M.S. Vekic, S.U. Grabic, N.L. Celanovic, and V.A. Katic. Hil evaluation of power flow control strategies for energy storage connected to smart grid under unbalanced conditions. *Power Electronics, IEEE Transactions on*, 27(11):4699–4710, 2012.

[46] Nasser Jazdi, Camelia Maga, and Peter Göhner. Reusable models in industrial automation: experiences in defining appropriate levels of granularity. In *Proc. of the 18th IFAC World Congress*, pages 9145–9150, 2011.

[47] Mira Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 354. IEEE Computer Society, 2001.

[48] R. S. Kalawsky, J. O'Brien, S. Chong, C. Wong, H. Jia, H. Pan, and P. R. Moore. Bridging the gaps in a model-based system engineering workflow by encompassing hardware-in-the-loop simulation. *Systems Journal, IEEE*, PP(99):1–1, 2013.

[49] Mark Karpenko and Nariman Sepehri. Hardware-in-the-loop simulator for research on fault tolerant control of electrohydraulic actuators in a flight control application. *Mechatronics*, 19(7):1067 – 1077, 2009. Special Issue on Hardware-in-the-loop simulation.

[50] Benjamin Kormann, Dmitry Tikhonov, and Birgit Vogel-Heuser. Automated plc software testing using adapted uml sequence diagrams. In *Information Control Problems in Manufacturing*, volume 14, pages 1615–1621, 2012.

[51] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[52] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *Model Checking Software*, pages 109–126. Springer, 2004.

[53] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[54] Gary T Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, et al. Roadmap for enhanced languages and methods to aid verification. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236. ACM, 2006.

[55] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *Software, IEEE*, 20(6):35–39, 2003.

[56] D. Maclay. Simulation gets into the loop. *IEE Review*, 43(3):109–112, 1997.

[57] Ken Martin and Bill Hoffman. An open source approach to developing software in a small organization. *Ieee Software*, 24(1):46–53, 2007.

[58] Rick Miller and Raffi Kasparian. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, 2006.

[59] I. Munteanu, A.I. Bratcu, S. Bacha, D. Roye, and J. Guiraud. Hardware-in-the-loop-based simulator for a class of variable-speed wind energy conversion systems: Design and performance assessment. *Energy Conversion, IEEE Transactions on*, 25(2):564–576, 2010.

[60] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[61] Josef Papenfort. Object oriented programming in automation.

[62] Michael D. Petersheim and Sean N. Brennan. Scaling of hybrid-electric vehicle powertrain components for hardware-in-the-loop simulation. *Mechatronics*, 19(7):1078 – 1090, 2009. Special Issue on Hardware-in-the-loop simulation.

[63] Sebastian Preuße and H-M Hanisch. Verifying functional and non-functional properties of manufacturing control systems. In *Dependable Control of Discrete Systems (DCDS), 2011 3rd International Workshop on*, pages 41–46. IEEE, 2011.

[64] L. Racchetti and C. Fantuzzi. Hardware in the loop simulation and machine modular development: Concepts and application. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–4, Sept 2013.

[65] L. Racchetti, L. Tacconi, and C. Fantuzzi. Generating automatically the documentation from plc code by d4t3 to improve the usability and life cycle management of software in automation. In *Automation Science and Engineering (CASE), 2015 IEEE International Conference on*, pages 168–173, Aug 2015.

[66] Lorenzo Racchetti, Cesare Fantuzzi, and Lorenzo Tacconi. Verification and validation based on the generation of testing sequences from timing diagram specifications in industrial automation. In *Industrial Electronics Society, IECON 2015 - 41st Annual Conference of the IEEE*, pages 002816–002821, Nov 2015.

[67] Lorenzo Racchetti, Cesare Fantuzzi, Lorenzo Tacconi, and Marcello Bonfe. The plc uml state-chart design pattern. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–4. IEEE, 2014.

[68] Lorenzo Racchetti, Cesare Fantuzzi, Lorenzo Tacconi, and Marcello Bonfe. Towards an abstraction layer for plc programming using object-oriented features of iec61131-3 applied to motion control. In *Industrial Electronics Society, IECON 2015 - 41st Annual Conference of the IEEE*, pages 000298–000303, Nov 2015.

[69] D.J. Rankin and Jin Jiang. A hardware-in-the-loop simulation platform for the verification and validation of safety control systems. *Nuclear Science, IEEE Transactions on*, 58(2):468–478, 2011.

[70] J-M Roussel and J Faure. An algebraic approach for plc programs verification. In *Discrete Event Systems, 2002. Proceedings. Sixth International Workshop on*, pages 303–308. IEEE, 2002.

[71] Krzysztof Sacha. Translatable finite state time machine. In *SDL 2007: Design for Dependable Systems*, pages 117–132. Springer, 2007.

[72] Miro Samek. *Practical Statecharts in C/C+: Quantum Programming for Embedded SystemscMiro Samek*. Taylor & Francis US, 2002.

[73] M. Sato. Flight test of flight controller for arbitrary maneuverability and wind gust rejection. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 2534–2540, 2006.

[74] U Schunemann. Programming plcs with an object-oriented approach. *ATP International, Automation Technology In Practice*, (2):59–63, 2007.

[75] C. Secchi, M. Bonfe, C. Fantuzzi, R. Borsari, and D. Borghi. Object-oriented modeling of complex mechatronic components for the manufacturing industry. *Mechatronics, IEEE/ASME Transactions on*, 12(6):696–702, 2007.

[76] Christian Secchi, Marcello Bonfé, Cesare Fantuzzi, Roberto Borsari, and Davide Borghi. Object-oriented modeling of complex mechatronic components for the manufacturing industry. *Mechatronics, IEEE/ASME Transactions on*, 12(6):696–702, 2007.

[77] Stephan Seidel, Thomas Klotz, Ulrich Donath, and Jürgen Haufe. Modelling the real-time behaviour of machine controls using uml statecharts. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8. IEEE, 2010.

[78] B. Selic and J. Rumbaugh. Using UML for complex real-time systems. IBM Rational Software Ltd, white paper, 1998. www-106.ibm.com/developerworks/rational/library/-139.html.

[79] D. Shetty, R.A. Kolk, J. Kondo, and C. Campana. A new approach to mechatronics system design using hardware in the loop simulation. In *Advanced Intelligent Mechatronics, 2001. Proceedings. 2001 IEEE/ASME International Conference on*, volume 2, pages 1005–1010 vol.2, 2001.

[80] C. Sünder, A. Zoitl, F. Mehofer, and B. Favre-Bulle. Advanced use of plcopen motion control library for autonomous servo drives in iec 61499 based automation and control systems. *e & i Elektrotechnik und Informationstechnik*, 123(5):191–196, 2006.

[81] R Stetter. Software im maschinenbau–laestiges anhaengsel oder chance zur marktfuehrerschaft. Technical report, Technical report, VDMA, 2011.

[82] Rainer Stetter. Software im maschinenbau - lästiges anhängsel oder chance zur marktführerschaft?

[83] Kleanthis C Thramboulidis. Using uml in control and automation: a model driven approach. In *Industrial Informatics, 2004. INDIN'04. 2004 2nd IEEE International Conference on*, pages 587–593. IEEE, 2004.

[84] Toyoaki Tomura, Kiyoshi Uehiro, Satoshi Kanai, and Susumu Yamamoto. Developing simulation models of open distributed control system by using object-oriented structural and behavioral patterns. In *Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001. Proceedings. Fourth IEEE International Symposium on*, pages 428–437. IEEE, 2001.

[85] K Trkaj. Users introduce component based automation solutions. *Computing and Control Engineering*, 15(6):32–37, 2004.

[86] Eelco van der Wal. Plcopen. *IEEE Industrial Electronics Magazine*, 3(4):25, 2009.

[87] Dimitri Van Heesch. Doxygen.

[88] AC Vidanapathirana, SD Dewasurendra, and SG Abeyratne. Statechart based modeling and controller implementation of complex reactive systems. In *Industrial and Information Systems (ICIIS), 2011 6th IEEE International Conference on*, pages 493–498. IEEE, 2011.

[89] Birgit Vogel-Heuser, Christian Diedrich, Alexander Fay, Sabine Jeschke, Stefan Kowalewski, Martin Wollschlaeger, et al. Challenges for software engineering in automation. *Journal of Software Engineering and Applications*, 2014, 2014.

[90] Birgit Vogel-Heuser, Daniel Witsch, and Uwe Katzke. Automatic code generation from a uml model to iec 61131-3 and system configuration tools. In *Control and Automation, 2005. ICCA'05. International Conference on*, volume 2, pages 1034–1039. IEEE, 2005.

[91] Valeriy Vyatkin. Modelling and verification of discrete control systems.

[92] Valeriy Vyatkin. Software engineering in industrial automation: State-of-the-art review. *Industrial Informatics, IEEE Transactions on*, 9(3):1234–1249, 2013.

[93] Valeriy Vyatkin and H-M Hanisch. Application of visual specifications for verification of distributed controllers. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, volume 1, pages 646–651. IEEE, 2001.

[94] M. Wehrmeister, C. Pereira, and F. Rammig. Aspect-oriented model-driven engineering for embedded systems applied to automation systems. *Industrial Informatics, IEEE Transactions on*, PP(99):1–1, 2013.

[95] B. Werner. Object-Oriented extensions for IEC 61131-3. *IEEE Industrial Electronics Magazine*, 3(4):36–39, December 2009.

[96] Wikipedia. Mechatronics — wikipedia, the free encyclopedia, 2016. [Online; accessed 8-February-2016].

[97] D Witsch, M Ricken, B Kormann, and B Vogel-Heuser. Plc-statecharts: An approach to integrate umlstatecharts in open-loop control engineering. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 915–920. IEEE, 2010.

[98] Daniel Witsch and Birgit Vogel-Heuser. Close integration between uml and iec 61131-3: New possibilities through object-oriented extensions. In *Emerging Technologies and Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–6. IEEE, 2009.

[99] Maria Witsch and Birgit Vogel-Heuser. Towards a formal specification framework for manufacturing execution systems. *Industrial Informatics, IEEE Transactions on*, 8(2):311–320, 2012.

[100] C.H. Yang, G. Zhabelova, C.W. Yang, and V. Vyatkin. Co-simulation environment for distributed controls of smartgrid. *Industrial Informatics, IEEE Transactions on*, PP(99):1–1, 2013.

[101] M Bani Younis and Georg Frey. Formalization of existing plc programs: A survey. In *Proceedings of CESA*, pages 0234–0239, 2003.