We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

2

Practice of Industrial Control Logic Programming using Library Components

Oscar Ljungkrantz, Knut Åkesson and Martin Fabian
Department of Signals and Systems
Chalmers University of Technology
Sweden

1. Introduction

This chapter discusses *Programmable Logic Controller* (*PLC*) programming practice, particularly the use of library components, in the automotive industry. A study of program structure and use of library components at two European car manufacturers is presented. The main purpose of the study is to provide understanding of current PLC programming in industry.

PLCs are commonly used in mass-production for instance to coordinate robots and machines. The life-cycles of many mass-produced products, including automotive products, have decreased significantly during the last years, due to changing market demands and increased competition. This has put new requirements on PLC programs, which must be easily modifiable and quickly made fully operational, to decrease down-time and ramp-up-time of the production system (Mehrabi et al., 2000).

PLCs are traditionally manually programmed in any of the languages of the *IEC 61131-3* standard (IEC, 2003; Lewis, 1998). Especially *Ladder Diagrams* (*LDs*), derived from the time when physical relays where used to control the machines, are common (Johnson, 2002). To gain reusability and modifiability, PLC code can be encapsulated and reused as *function blocks* (*FBs*). Nonetheless, the traditional PLC programs tend to be difficult to modify and extend and not flexible enough to meet the new requirements (Lewis, 2001).

A solution to the problems might be to use frameworks that facilitate the development of flexible and operational control programs. Hence, many researchers have developed new frameworks and tools to develop or automatically generate PLC code to meet the new requirements. Overview of such frameworks can be seen in (Lee et al., 2006; Ljungkrantz & Åkesson, 2007). In spite of the potential benefits of these academic frameworks, they have not been reported to be used in full scale industrial projects. One obstacle is that the generated code in practice often has to be modified by hand and integrated with working code already existing in industry.

For any code generating framework to be industrially successful, it certainly has to fulfil the requirements of industry. Moreover, successful integration of the generated code with already existing code requires understanding of PLC programming practice. This chapter aims at providing this knowledge. The chapter focuses on FB usage since reusing FBs created at the manufacturing companies is a promising approach for performing the code

Source: Programmable Logic Controller, Book edited by: Luiz Affonso Guedes,
ISBN 978-953-7619-63-3, pp. 170, January 2010, INTECH, Croatia, downloaded from SCIYO.COM

integration. Most results and findings are based on a study performed 2007 at two Swedish car factories, which is reported in (Ljungkrantz & Åkesson, 2007) and is restated with some additional comments and findings in Section 2–5 of this chapter. A comparable study was performed at Lamb Technion in USA (Lucas & Tilbury, 2003). That study was however focused on the development process and not on library components. Furthermore, only LD programming was used in that study, while this chapter presents the use of other languages and programming constructs as well.
This chapter describes three major observations:

- The PLC programs in the studied companies were written mainly in Ladder Diagrams and Sequential Function Charts. These programs frequently reused function blocks.
- The PLC programs handled, besides automatic control, also safety and supervision, human machine interface, product data, communication etc. The code for automatic control was a minor part of the total code.
- Although the function blocks were frequently reused, their behaviours were only informally described.

To improve the efficiency and reliability when reusing FBs, we think it's crucial that the FBs are unambiguously specified and verified. The end of this chapter therefore shows how FBs can be formally specified and then verified using *model checking*. Model checking means to automatically check whether or not a model fulfils a specification (Clarke et al., 2000). Thus, model checking complements the traditional methods of testing and simulation. FBs can be augmented with formal specifications to form components we call *Reusable Automation Components* (*RACs*) (Ljungkrantz et al., 2008), which can be verified using model checking. An example FB is specified and verified as a RAC; an error is detected, the implementation is corrected and the final RAC is successfully verified. This shows the potential of using formal methods in function block development.

1.1 Chapter organization

This chapter is organized as follows: Section 2 describes the scope and methods of the study. In Section 3, the control program development at the studied companies is explained. In Section 4, the most frequently used library components are presented and discussed and in Section 5 a classification and statistics of the library components FBs, are presented. Section 6 discusses formal specification and verification of FBs and applies these techniques on an example FB. Conclusions are given in Section 7.

2. Study of control code and library components

The program structure and the library components in PLCs used at two Swedish car companies were studied in (Ljungkrantz and Åkesson, 2007). Mainly the code used in the car body assembly factories located in Sweden was investigated, since the PLCs in those factories control many robots, conveyors and other machines and have quite standardized layout. Other PLC programs at the two companies may be different from those studied. Still, "Company 1" and "Company 2" will from now on be used to refer to the respective studied factories and "the studied companies" will be used when referring to both. The investigation was performed by 1) manually reading the code in the PLC program development tool, 2) discussing with PLC engineers and programmers at the studied companies and studying a master thesis performed at the companies (Bergqvist and Öberg,

2007) and 3) writing a program that searches through PLC code and libraries and extracts FB usage statistics. At the time of the study, the studied companies used the same PLC program development tool, see Section 3.2.
The PLC code investigated was structured as different *projects*, each representing the code that runs on one PLC. In many cases one PLC controls one manufacturing cell, but in some cases two or more PLCs are used for one cell. Normally a cell is divided into several stations and a PLC often controls more than one station. For a fair comparison between the studied companies nine similar projects were chosen at each company: two *underbody* projects, three *respot* projects, one *side line* project, one *framing* project and two *transportation* projects. In the underbody cells, robots weld/bolt parts together to form the floor of a car. In the respot line the car floor or body are transported between the cells by a conveyor system and in each cell robots perform extra welding/bolting to increase the strength and to add extra parts. In the side line cells the sides of the car are built. In the framing cells the car body is built by welding together the car floor with sides etc. In the transportation cells conveyors, lifts etc. transport the car floor or body.
The projects and the libraries were exported to text files. The developed program reads those files and detects all instances of each FB. It detects both FB instances that are used directly in the projects and FB instances that are used indirectly. FBs are considered to be used indirectly if they are used inside an FB, which has instances directly used in a project or in turn is used indirectly. The program presents usage statistics of the FBs.
All used FBs were also classified into nine different categories, see Section 5. The program presents the number of instances and proportion of each category.

3. Control program development

This section describes the development of PLC programs at the studied companies, by describing the development actors, programming environment and general program structure.

3.1 Important actors of the development process

The PLC programs are usually developed by firms contracted by the studied companies. These firms program in a certain structure by following guidelines at the studied companies. Company 1 has a written specification for control programming and a standard project to start from. Company 2 has a stricter standard and structure of the code for the developers to follow. Both of the studied companies also provide libraries with components to reuse. The consultants may add components into the library, but these components are reviewed by the studied companies. At the time of the study, Company 1 had one person responsible for the library but a team of people that could review the code. Most of the components had no documentation apart from comments within the components. Hence, to understand the behaviour of a component, its internal code and comments had to be examined. At Company 2 a single person reviewed and also documented all library components. The documentation at Company 2 was done using pictures and natural language and was connected to the library components (as help files in the PLC program development tool). In addition to these internally developed and maintained libraries, suppliers of certain equipment also provide libraries with components to use with that

equipment. Finally, the supplier of the PLC hardware and the development tool also provided a number of libraries to use.

3.2 Development environment and programming languages

At the time of the study, PLCs from the same vendor were used at both of the studied companies. The development tool used to program these PLCs supports programming in the IEC 61131 standard (IEC, 2003). Hence the programs can be written in five languages: *Sequential Function Chart* (*SFC*), LD, *Function Block Diagram*, *Instruction List* and *Structured Text* (to be precise, SFC is not considered a language in the standard, merely a graphical technique or program structure). The standard defines components called *POUs*, Program Organisation Units, to be reused and stored into component libraries. POUs can be of three different types: *functions*, FBs and *programs*. Functions may have many inputs but only one output. They have no memory and are typically used for mathematical operations. FBs allow an algorithm or set of actions to be applied to a given set of data, including inputs and internal variables, to produce a new set of output data. The behaviour of the FBs can be implemented in any of the five IEC 61131 languages and FB instances can be used in code written in any of the five IEC 61131 languages. Note that although IEC 61131 allows it, the used PLC program development tool did not permit the behaviour of FBs to be implemented using SFC.

3.3 General program structure

Both Company 1's and Company 2's projects consisted of several programs. Typically most programs were written in LD, one in Instruction List and up to two programs per station in SFC. A main sequence SFC of each station normally controlled the main order in which the operations of robots, clamps, transportation systems etc. should be performed. At Company 1 the robots also allocated resources (machines or virtual zones), before they for instance started welding, to avoid collision. This was done by having a separate LD program for each robot that handled interlocks and allocation of resources needed by the robot. At Company 2 this resource allocation was not needed since the sequence itself guaranteed that no collisions and variations occurred.

At Company 2 only nine types of programs were identified: two general programs ("Always" and "PLC_General", both LDs), one program for the Profibus communication (Instruction List), four programs for each station *X* (two SFCs, "Stn*X*_Auto" and "Stn*X*_Homerun", and two LDs, "Stn*X*_Manual" and "Stn*X*_General") and finally two built-in supervision programs provided by the PLC supplier. Company 1's projects were split into more types of programs: five to ten general programs (for communication, finishing the line, indication, communication with the safety PLC etc., all LDs), one program for the Profibus communication (Instruction List), many programs for each station *X* and robot *Y* (up to two SFCs, "S*X*Main" and "S*X*HomeRun", and many LDs, for instance "S*X*Movement", "S*X*Transport", "S*X*SumMemories", "S*X*BodyId", "S*X*Alarms", "S*X*Robots*Y* " and "S*X*Indications)". While Company 2's projects for instance had alarm handling code inside the actions of the SFC, at Company 1 the actions of the SFC were mainly used to set variables that in turn were used in the LD programs. For simple transportation cells neither Company 1 nor Company 2 used SFC.

Although most of the programs were implemented in LD many FB instances were used and called from the LD code. Some programs almost resembled Function Block Diagrams. The behaviour of almost all FBs was implemented in LD.

The studied companies had few levels of hierarchy in the sense that FBs, apart from basic FBs, seldom were used inside other FBs. Company 2 argued that blocks inside other blocks make it hard to read and understand the code. Both of the studied companies put emphasis on the importance of having code that can be understood and used in trouble-shooting by the operators; this affected both the structure and naming of the code and the comments. Ideally, the alarms and indications of a PLC project are sufficient for the operators to solve problems but since this is not always the case, the code must be readable by the operators.

4. Frequent library components

At both of the studied companies programs were reused indirectly by starting from copies of standard projects or programs. Only the built-in supervision programs at Company 2 were reused as is. Two built-in basic libraries provided with the PLC program development tool consisted of both functions and FBs for mathematical operations, bit-manipulating etc. The rest of the libraries almost exclusively consisted of FBs. Therefore the investigations focused on FB reuse.

To illustrate the use of FBs at the studied companies, the five most frequently used FBs, according to the projects investigated, are briefly described here. Then some of the FBs are further described in an example of controlling two parallel clamps and the approaches chosen at Company 1 and Company 2 are compared. The most frequent FBs are presented in Table 1. They represented approximately 50 % of the total number of FB instances.

	Company 1		Company 2	
	FB Name	FB Instances Dir. / Indir.	FB Name	FB Instances Dir. / Indir.
1	FB_Event	893 / 19	OUT_SVx	380 / 13
2	FB_Move	301 / 8	ManAuto	78 / 0
3	FB_Alarm_Clamps	269 / 0	Valve_ctrl$x$	74 / 4
4	FB_Event_Clamps	266 / 0	CycleTime	70 / 0
5	FB_AllocateZon	226 / 0	EM_Status	54 / 0

Table 1. Most used FBs at the studied companies in the investigated PLCs.

4.1 Company 1

FB_Event

FB_Event uses a counter to assure that its binary output signal is held high for a minimum time, when its binary input goes high. The purpose is to assure that signals sent to other systems keep their values long enough to be detected. It should be used for all signals sent via TCP/IP. Furthermore, in the study FB_Event was used at almost all binary status signals sent to actuators, supervision and HMI systems.

FB_Move

FB_Move, see Figure 1, controls the movement of actuators like clamps, fixation pins and lifts. It can be used for moving the actuator both backwards and forwards in either automatic or manual mode. If for instance a forward movement in automatic mode is ordered by signal *AutoFwd*, some conditions are checked and if those are fulfilled the output signal *OutputFwd* goes high. At the same time a timer is started and when the timer has reached the value of *TimeValue* the *TimeOutFwd* output goes high.

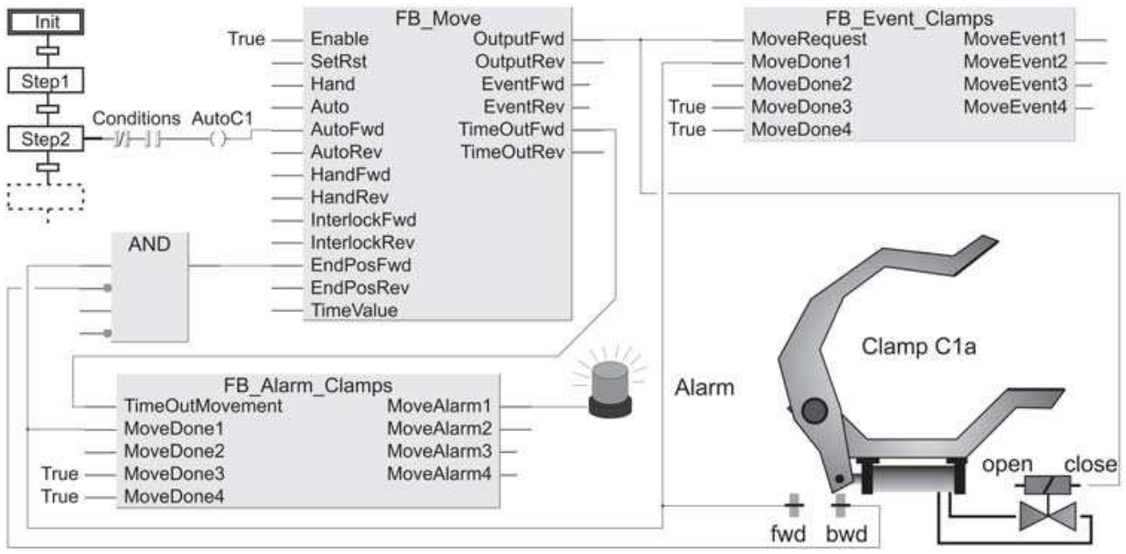

Fig. 1. The principle of controlling two parallel clamps at Company 1. To increase readability, the components for one of the clamps and for backward movement are omitted.

FB_Alarm_Clamps
FB_Alarm_Clamps is used to send an alarm if the movement of any of up to four parallel clamps is not performed within a specified time, see Figure 1. The input signal *TimeOutMovement* is activated by an external timer and when this signal goes high FB_Alarm_Clamps sends alarms for all clamps that have not yet reached the end position.

FB_Event_Clamps
FB_Event_Clamps FB has the same purpose as FB_Event but can be used for up to four parallel clamps. It has five input signals: a move request for the whole clamp group and four signals telling whether the connected clamps have reached their end positions or not, see Figure 1. Inside FB_Event_Clamps the four signals each goes through an FB_Event.

FB_AllocateZon
FB_AllocateZon is used to handle interlocks between a robot and a machine or between different robots, to avoid collision. A robot FB sends a unique number, representing the resource that the robot wants to use, to the FB_AllocateZon. The FB_AllocateZon checks that the conditions are met and when so it sends back the number representing the resource.

4.2 Company 2

OUT_SV*x*
OUT_SV and OUT_SV*x* (*x* = 2,..,6 is the number of parallel movements to supervise) are FBs included in the built-in supervision library provided with the PLC program development tool, see OUT_SV2 in Figure 2. These FBs are used to supervise movements by checking that the movement has stopped within a specified time after the *Run* signal is given. Otherwise alarms are given for the movements that are not finished. The output signal *Out* shall be connected to the component that shall be moved.

ManAuto
The ManAuto FB was in the study used once for every station at Company 2. The FB handles the choice for running in automatic or manual mode and has input signals for the desired mode and for emergency stop, acknowledge signals for Profibus communication etc.

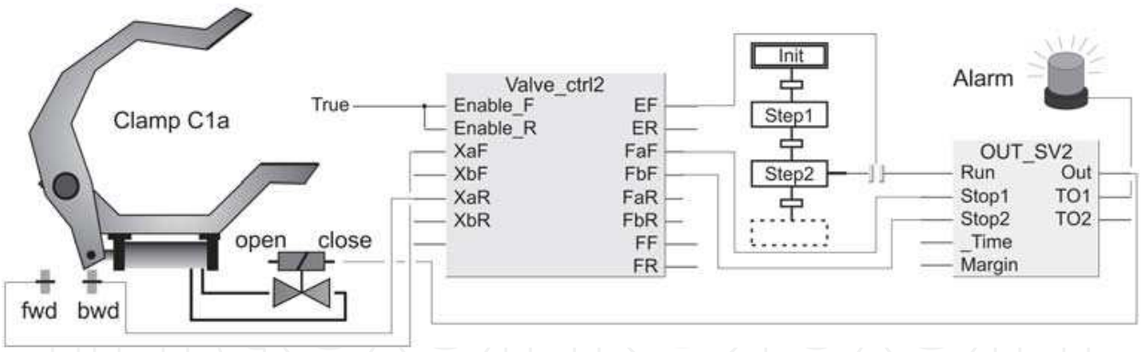

Fig. 2. The principle of controlling two parallel clamps at Company 2. To increase readability the components for one of the clamps and for backward movement are omitted.

If all conditions are fulfilled, the desired mode is chosen. The *Au* and *Ma* output signals were used as conditions for conveyors, robots and actuators, either as a logic condition for a specific movement or as a condition for a whole program. The latter was used for the programs that handled only automatic or manual control of a station. When the ManAuto output FB*X*.Ma was *true* it activated a *task*, see (IEC, 2003), so that program Stn*X*_Manual ran. When instead FB*X*.Au was *true* it activated a built-in function SFC_CTRL so that the SFC Stn*X*_Auto ran.

Valve_ctrl*x*

Valve_ctrl and Valve_ctrl*x* (*x* = 2,..,13 is the number of parallel actuators) are FBs to control one or many actuators connected to one valve, see Valve_ctrl2 in Figure 2. The end position sensors, backwards and forwards, for each of the connected actuators are input signals to the block. If the *Enable_F* input signal is *true* the forward output *EF* is set if all actuators are in backward position. The FB also has output signals for each of the actuators stating if the actuator is in forward and not backward position, and vice versa.

CycleTime

The FB CycleTime calculates the cycle time for a station by increasing a counter each second when the station is not paused, and resetting every new cycle.

EM_Status

EM_Status identifies and sends an error message from an electric monorail conveyor.

4.3 Example and comparison

In this section the main control approach at the studied companies will be explained using a simple example, in which many of the above FBs will be used. The task is to close a clamp group, consisting of two clamps that are moved in parallel via one pneumatic valve connected to the cylinders of both clamps. Each clamp has sensors in both end positions.

The components for controlling the clamps at Company 1 are shown in Figure 1. The movement is started when the main sequence of the station, implemented as an SFC, is in the position where the clamps should be closed. If some basic conditions are satisfied (for instance that the station cycle has not already been performed in manual mode) the FB_Move is told to start the movement of the clamps. If the station is in Auto, the clamp group is not already closed etc., FB_Move starts a timer and sends a signal to the valve to close the clamp group. This signal is also sent to the FB_Event_Clamps. If any of the two clamps is not closed within the maximum time allowed for the clamp group, the FB_Alarm_Clamps sends an alarm for that clamp. The AND operator is used to assure that

both clamps are closed and not open. In the figure all FBs for one clamp and for closing the clamp are shown. Components for the second clamp (C1b) should be added in the same way, components for opening the clamp should also be added in a similar way. The FBs FB_Event_Clamps and FB_Alarm_Clamps can be used for up to four clamps. As seen, fewer clamps can be controlled by setting the unused input sensor signals to *true* and letting the corresponding output signals be unconnected. Four of the five most used FBs at Company 1 in the study are used in the example (FB_Event is used inside FB_Event_Clamps). The signal out of the AND operator is a typical signal that can be used as an interlock for the fifth most used FB, FB_AllocateZon, for instance guaranteeing that the clamps are closed before the robot welds the part held by the clamps.

The components for controlling the clamps at Company 2 are shown in Figure 2. When the clamp group is open this is known by the Valve_ctrl2, since all four end position sensors are connected to this FB, and the EF (enable forward) output is high and the FaF and FbF outputs are low since the clamps are not in closed position. The real movement is started first when the auto sequence of the station, implemented as SFC also at Company 2, is in the position where the clamps should be closed. Now the OUT_SV2 FB tells the clamps to close. If a clamp is not in forward position before the time *_Time* has passed, an alarm is raised.

The approaches at the two companies in the study were quite similar, as exemplified above, letting an SFC start the movement and reusing common FBs, with LD to describe the logic. Nevertheless, there were also small but interesting differences. The function of the AND operator in the Company 1 example was instead included in the Valve_ctrl*x* FB using LD, at Company 2. At Company 2 different FBs were needed when different numbers of clamps were to be controlled, as indicated by the number succeeding the FB name (for instance Valve_ctrl2 and OUT_SV2). This means that Company 2, in this case, had to keep and maintain more FBs in the library, but on the other hand did not have to set unconnected inputs to *true* or *false*. The OUT_SV*x* FBs that were used at Company 2 are very similar to Company 1's FB_Alarm_Clamps, but are included in the built-in supervision library provided with the PLC program development tool. A benefit of using OUT_SV*x* FBs is that they can be given a teach mode in which the supervision program detects the actual time before the stop signals are detected and updates the _Time parameter with the measured time plus the *Margin*, given in %. Finally, the FB_Event was very common at Company 1 but not used at Company 2.

5. Classification and statistics of function blocks

In 2007, Company 1 had recently started classifying their in-house libraries into function based categories. Company 2 had chosen a more equipment-based classification. To be able to compare the libraries we divided the FBs into nine categories. All FBs in frequent libraries and all used FBs have been classified. The categories are listed below.

- *Robot Control:* Control of, and resource allocation for, robots.
- *Machine Control:* Control of other machines than robots, e.g. actuators and conveyors.
- *HMI:* FBs for indication, mode-choice and manual control. Interaction with the operators.
- *Safety and Supervision:* FBs for alarm handling, communication with the safety PLC and automatic safety operations like emergency stop.

- *Product and Production Data:* FBs for communicating with identification systems like barcodes and RFID, and for controlling the production by for instance choosing next product type.
- *Statistics:* Data collection and calculations for analysis, for instance cycle time, product counters and mean time between failures.
- *Ethernet & Profibus Communication:* Communication protocols, drivers etc. for Profibus and Ethernet.
- *General Functions:* FBs like timers, clock settings and bit-manipulating, maintained by the studied companies.
- *Basic:* The FBs in the two built-in libraries *Manufacturer_Lib* and *Standard_ Lib*, provided with the PLC program development tool. FBs for basic mathematical operations, bit-manipulating etc.

At Company 1, 249 FBs have been classified and 141 of those were used in the investigated projects, including basic FBs. At Company 2, 200 FBs have been classified and 80 were used in the investigated projects, including basic FBs.
In the investigated projects Company 2 had 1338 FB instances and Company 1 had 4514 FB instances. Ignoring the Basic FBs they still used 1115 and 4128 FB instances respectively. At Company 1 FB_Event and FB_Event_Clamps accounted for almost 30% of all FB instances. Besides, they were not used at Company 2 and used in many different circumstances at Company 1, so placing them in a single category would be inaccurate. Hence, they have been excluded when counting how many FB instances that are used within each category. Even with FB_Event and FB_Event_Clamps excluded Company 1 used 2950 FBs which is significantly more than Company 2's 1115. Although it was the intention to choose similar projects from Company 1 and Company 2, a reason for the difference may be more extensive PLC projects at Company 1. The difference might also be due to different structure and usage of FBs within the projects, at Company 1 and Company 2. This explanation is indicated by the clamp control example depicted in Figure 1 and 2, showing three directly used FB instances and one indirectly used FB instance (FB_Event inside FB_Event_Clamps) at Company 1 but only two FB instances at Company 2. The FB instances divided into the different categories can be seen in Table 2.

Category	Used FB instances [%]	
	Company 1	Company 2
Robot Control	22	6
Machine Control	17	15
HMI	19	7
Safety and Supervision	25	50
Product and Production Data	6	5
Statistics	5	8
Ethernet & Profibus Com.	4	7
General Functions	3	2

Table 2. Percentages of used FB instances divided into different categories.

The FB instances do not represent the complete code, neither do they directly correspond to the work done by the developers. For instance the Ethernet & Profibus communication instances were quite few in the study but each FB was often complex. Still, the FB instances

do represent a rough estimation of how the PLC code was divided. For instance the code handling HMI, safety, supervision, communication etc. undoubtedly represents a great part of the code. In (Lucas and Tilbury, 2003; Richardsson and Fabian, 2006) it is reported that according to their experience at Lamb Technion and Volvo Car Corporation respectively, the part of the code representing automatic control is about 10 % of the total. However, no data supporting this was shown in the two papers. In the investigation reported here the code for automatic control was *part of* the categories robot control and machine control, accounting for in total 39 % at Company 1 and 21 % at Company 2. For instance, Company 1's FB_Move, classified as machine control, was directly called from the SFC handling the automatic control. At Company 2 the EF output of Valve_ctrl*x*, classified as machine control, was directly used in the action logic of the SFC for automatic control. Nevertheless, some FBs classified as machine or robot control, especially at Company 1, handle low level control of the machines and robots and should not be considered code for the automatic control itself, rather help FBs for the code which handles the operation order for automatic control. Therefore we can not claim that the code representing automatic control is exactly 10 % of the total, but it is indeed fair to state that: *the code for automatic control is a minor part of the total code*.

It is also interesting to compare the category distribution at the two companies. Robot control is a greater part at Company 1 than at Company 2, which could be explained by the fact that Company 1 assumes that the operations can be executed in different orders and therefore uses zones to allocate resources. Company 1 also uses FBs for lamp indication (HMI) more frequently. The proportion of FB instances for alarm handling (safety and supervision) is significantly greater at Company 2. This can be explained by considering an SFC with parallel branches. At Company 2, the alarm handling FBs were included in the SFC and thus two instances of the involved FB existed in an SFC with two parallel branches and so on. At Company 1, the SFC branches set variables that in turn were used in separate LD programs, containing only one instance of the involved FB. In this particular case, the choice at Company 1 resulted in more compact code, while the code at Company 2 may be considered easier to read.

6. Formal specification and verification of function blocks

With the above findings as starting point, it is the authors' belief that the code reuse can be made more efficient and less error prone. Efficient code reuse indeed requires components with known behaviour. This can be achieved by developing clear and unambiguous *specifications* and by verifying that the specifications are fulfilled by the *implementation* (the code). The specification can be seen as an abstraction of the implementation, capturing important properties.

As explained in Section 3.1, most FBs at Company 1 had no external documentation. The internal comments of the FBs are in principle insufficient as specification, since these comments are too strongly connected to the implementation (possibly violating the principle of abstraction) and reading the comments require access to the implementation (violating the principle of information hiding, see (Parnas, 1972)). The external documentation of the FBs at Company 2 does not have these disadvantages. Nonetheless, being based on natural language, both the comments of Company 1 and the documentation of Company 2 might be ambiguous and not suitable as a basis for verification. In particular, this natural language documentation is not suitable when using *formal verification*.

Formal verification uses math-based models and algorithms to perform the verification and thus requires a formal and unambiguous specification. *Model checking* is an important set of formal verification methods that can perform the verification automatically and produce counterexamples if the specification is not fulfilled (Clarke et al., 2000). Model checking is promising in FB development, since compared to common field testing, model checking can be performed earlier in the development process. Model checking has also advantages compared to simulation, since in many situations it is too time consuming to simulate and test all different scenarios in which a component can be used. Model checking however typically performs exhaustive search of the models.
Model checking PLC code can be done using many different methods and tools, see (Bérard et al., 2001; Frey and Litz, 2000). The *Reusable Automation Component* (*RAC*) method developed by the authors of this chapter is tailored for specifying and verifying PLC program components, such as FBs (Ljungkrantz et al., 2008). The RAC specification structure and language is intended to be understandable by PLC program engineers without prior knowledge on formal languages. A RAC prototype tool has also been developed with which the RACs can be specified and then automatically translated into inputs to the model checking tool Cadence SMV (McMillan, 1993, 1999). The RAC method and tool is used here to demonstrate the usefulness of formal specification and verification in FB development. Next, the basics of the RAC are explained, followed by an example component that controls actuators similarly to the examples seen in Figure 1 and 2. This example component is not very complicated but stills shows the advantages of using formal verification. Formal specification and verification of the more complex component *FB_Move* used by Company 1 can be seen in (Ljungkrantz et al., 2008).

6.1 Reusable Automation Components (RACs)

The RACs were introduced in (Ljungkrantz et al., 2008). A RAC has an *interface* that includes inputs and outputs and a *body* that includes the implementation and internal variables. The main difference compared to FBs is that the RAC interface includes a formal specification. As help when developing and structuring the specification, five types of properties can be used, briefly described below:

- *Operation preconditions* are requirements that the user of the component must satisfy in order to obtain certain functionality, expressed by the *operation behaviours*.
- *Operation behaviours* are requirements, ensured by the developer of the component, that must be fulfilled when all operation preconditions are satisfied.
- *Exception conditions* are prioritized inputs or combinations of inputs that lead to exceptions. When an exception condition is *true* none of the operation behaviours can be guaranteed. Instead the exception condition must always guarantee certain behaviour, which must be described as *exception behaviours*.
- *Exception behaviours* are requirements, ensured by the developer of the component regardless of the operation preconditions. Each exception behaviour includes one or more exception conditions.
- *Invariants* are requirements, ensured by the developer of the component regardless of operation preconditions.

The operation preconditions and operation behaviours are grouped as *operation specification* and the exception conditions and exception behaviours are grouped as *exception specification*.

The *specification language* is based on IEC 61131-3 (all four languages but not SFC) and *Linear Temporal Logic* (*LTL*), see for instance (Clarke et al., 2000). The reason for basing the specification language on IEC 61131-3 is that most PLC engineers are familiar with the IEC 61131-3 languages but might not know other programming or specification languages. Augmenting the language with constructs for LTL is done to express relations over time. Temporal logic contains constructs to reason about the order in time without explicitly mentioning time; for instance it can state that something will *always* or *eventually* be *true*. LTL is a type of temporal logic that suits the input-output based relations of FBs well and is also supported by model checking tools. The specification language contains spelled out versions of the temporal operators but also short-hand notations for some basic constructs, like rising and falling edges of variables. For instance the rising edge of a boolean variable *v* can be expressed as *v_risingEdge* which is equivalent to (*NOT v_previous*) & *v*, using the Structured Text based variant of the specification language.

6.2 Example

As an example, the development of a RAC *Control_BinaryActuator*, implemented as a function block in LD, will be demonstrated. The RAC should control a binary actuator and should signal alarms if the movements are not performed within a maximum time. Hence this RAC will contain most parts of the components Valve_ctrl and OUT_SV from Company 2, see Section 4.2 and many parts of the components FB_Move and FB_Alarm_Clamps, excluding interlocks and mode handling, from Company 1, see Section 4.1.

Assume that the interface of the example component has already been determined. The inputs and outputs of Control_BinaryActuator can be seen in Figure 3, which also shows how the component can be used to control a cylinder. When the *Move* input is *true*, the actuator will move forwards by setting *ActuatorFwds* to *true* if the *DesiredState* is "*Forward*" and move backwards by setting *ActuatorBwds* to *true* if *DesiredState* is "*Backward*". Move must be held *true* throughout the complete movement. When the movement has been performed, as indicated by the sensor inputs *SensorFwd* and *SensorBwd*, the *State* output will be set to the new state. The component also has checks to see if the actuator performs accurately and outputs alarm signals if not. If the movement is not performed within the maximum time allowed, *MaxMoveTime*, the corresponding alarm, *TimeOutActFwds* or *TimeOutActBwds*, will be set *true*. The *AlarmUnauthMove* alarm is set if the actuator moves when it is not supposed to. Finally, the alarms can be reset by the user, by setting *ResetAlarms* to *true*.

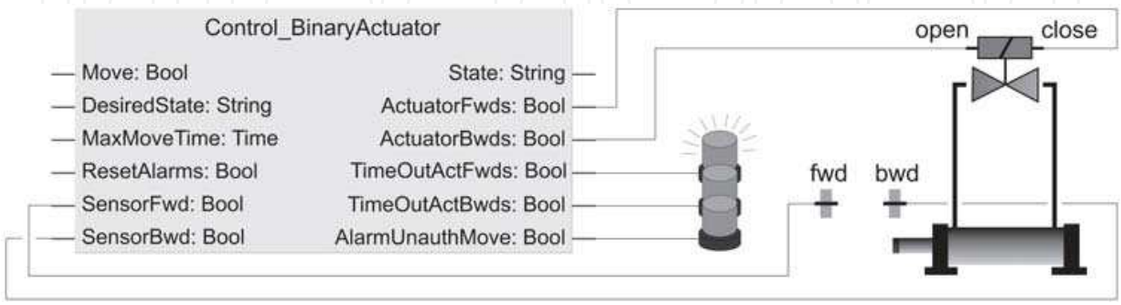

Fig. 3. The inputs and outputs of the *Control_BinaryActuator* RAC.

The specification of Control_BinaryActuator can be seen in Figure 4, using the Structured Text based variant of the specification language.

```
Operation specification
 Operation preconditions
 OpPre1 := ALWAYS( (MaxMoveTime > 0) &
                   ((DesiredState = Forward) OR (DesiredState = Backward)) );
 OpPre2 := ALWAYS( Move & Move_previous ->
                                    (DesiredState = DesiredState_previous) );
 Operation behaviours
 MoveOrAlarm := ALWAYS( (ALWAYS Move) -> EVENTUALLY((State = DesiredState) OR
                                        TimeOutActFwds OR TimeOutActBwds) );
Exception Specification
 Exception conditions
 Reset := ResetAlarms;
 Exception behaviours
 ResetBhvr := ALWAYS( ResetAlarms ->
           (NOT TimeOutActFwds & NOT TimeOutActBwds & NOT AlarmUnauthMove) );
 Invariants
 NotIllegalMove := NEVER( ActuatorFwds & ActuatorBwds );
 Stop := ALWAYS( (NOT Move) -> (NOT ActuatorFwds & NOT ActuatorBwds) );
```

Fig. 4. Specification of the *Control_BinaryActuator* RAC.

The first operation precondition states the allowed input values for MaxMoveTime and DesiredState. The second operation precondition states that the user must not change the direction of the movement while moving. The operation behaviour *MoveOrAlarm* summarizes the main functionality of the RAC by stating that when the user of the RAC is trying to move the actuator, the actuator will eventually reach the desired state or an alarm is raised. More operation behaviours could be added, for instance to specify under what circumstances the operating outputs ActuatorFwds and ActuatorBwds are actually *true*, but for brevity only MoveOrAlarm is shown. The exception condition *Reset* states that none of the operation behaviours can be guaranteed if the ResetAlarms input is *true*. The corresponding exception behaviour *ResetBhvr* declares that the ResetAlarms input will always reset all three alarms. The invariant *NotIllegalMove* states that the RAC will never try to move the actuator in both directions simultaneously. Finally, *Stop* declares that the outputs that move the actuator will never be *true* when Move is *false*. Note that *NOT Move* could as well have been specified as an exception condition, depending on how "normal" operation of the component is viewed. If so, *Stop* would have been specified as an exception behaviour instead.

Implementation and Verification

A rather straightforward attempt of implementing the example component can be seen in Figure 5. The implementation makes use of the standard functions *AND*, *EQ* (tests equality) and *MOVE_E* and of the function block *TON*. TON is a standard timer that sets the output *Q* to *true* if *IN* is *true* at least as long as the time *PT*. The function MOVE_E that is used in the position control at the top of Figure 5, copies the string on the *IN* input to the output to which *State* is connected, when the *EN* input is *true*. The positive (P) and negative (N) transition-sensing contacts are used to detect rising and falling edges of the signals, respectively.

The RAC can now be formally verified to check whether the implementation of Figure 5 fulfils the specification of Figure 4 or not. The RAC can be translated into inputs to Cadence

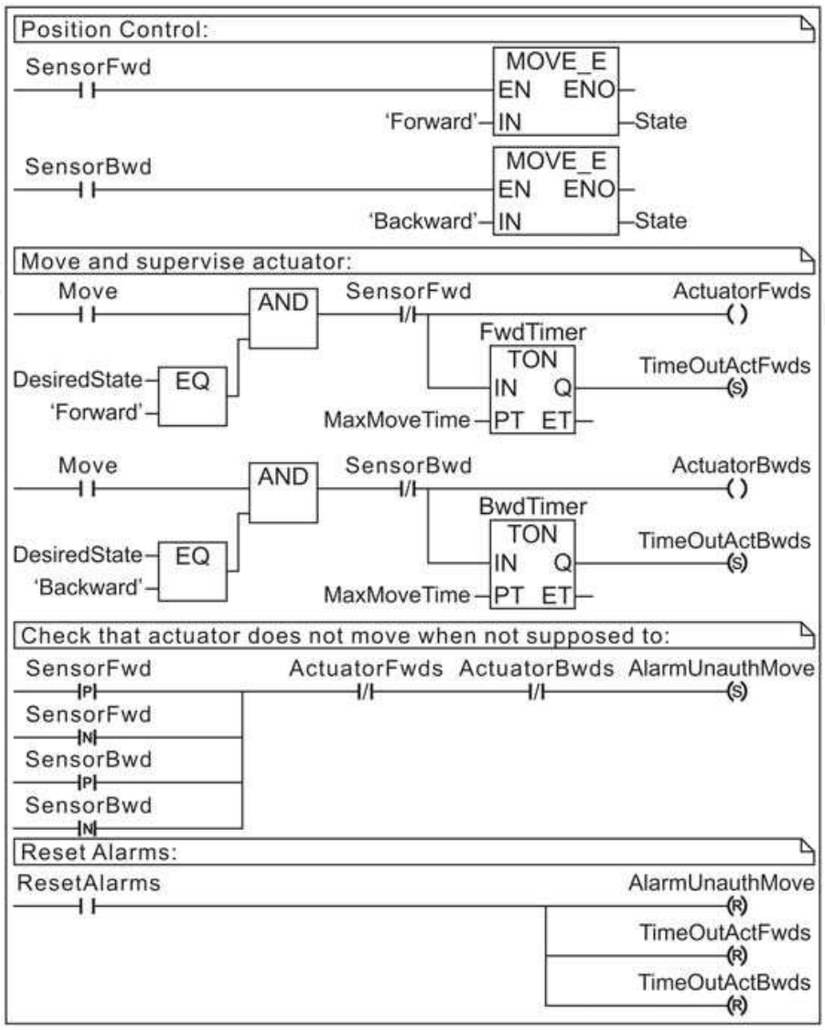

Fig. 5. An implementation approach of the *Control_BinaryActuator* RAC in Ladder Diagrams using standard functions and the timer FB *TON*.

SMV by the RAC prototype development tool, and then Cadence SMV can be used to perform the verification. Doing this, the result is that the RAC is *not valid*, that is the specification is *not fulfilled* by the implementation. Both invariants are fulfilled, but not the operation behaviour MoveOrAlarm. SMV gives a counterexample to why the operation behaviour is not fulfilled to help understand and solve the problem. If the actuator should be moved forward but the sensors are broken so that both sensor inputs are *true* at the same time, the actuator will not be moved and unfortunately the TimeOutActFwds will not be set. The FwdTimer will not be started since the SensorFwd is already *true*, but the state will be reported as *Backward* (from the second MOVE_E function) and hence MoveOrAlarm is not fulfilled. The RAC could certainly be made valid by adding a precondition saying that the actuators and sensors may never be broken, but a much better alternative is to change the implementation so the alarms will actually work when the sensors are broken.

To solve the problem, two internal variables *InFwdPosition* and *InBwdPosition* are used that are *true* only when SensorFwd and not SensorBwd are *true* and vice versa. Those internal variables are used as conditions to start the timers, as shown in Figure 6. Using this

implementation the complete specification is fulfilled and the RAC is valid. Even for such a small and elementary component as Control_BinaryActuator, the error of the first implementation attempt might be hard to foresee. By studying the counterexample of the model checking tool though, the error can be easily solved. This demonstrates the potential of using formal specification and verification in the FB development process.

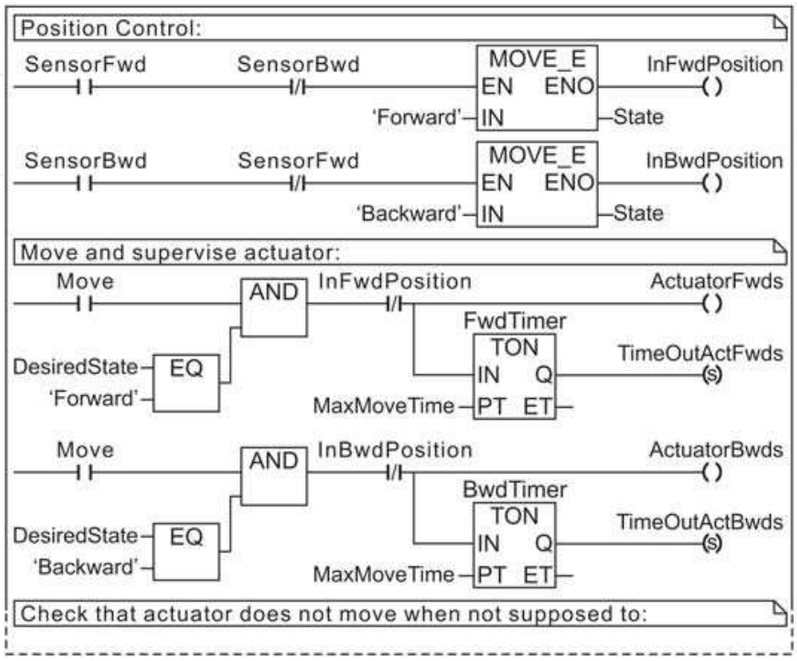

Fig. 6. A *valid* implementation of the *Control_BinaryActuator* RAC in Ladder Diagrams. The part not shown is exactly the same as in Figure 5.

7. Conclusions

In this chapter a study of PLC programming and use of library components at two Swedish car manufacturers is presented. Both companies used several programs for each PLC, implemented mainly in LD and SFC. These programs included lots of instances of reusable function blocks, FBs. Some of the most frequent FBs were used for automatic control of actuators and conveyors but in total only a minor part of the used FB instances was for automatic control; the majority was for HMI, safety, supervision, production data, communication etc. This is important to consider when developing or modifying frameworks for control program generation, to cope with the new requirements of flexible manufacturing systems. Integrating industrial FBs with new frameworks for generating control sequences is an interesting direction for future research.

It is also interesting to consider that although the FBs were frequently reused, their behaviours were only informally specified. In our opinion the FB reuse can be made more efficient by also using tools and methods for formal specification and verification. This is demonstrated by an example component, in which an error of the first implementation attempt is discovered and solved.

For formal specifications to be used in industry it is important that the development of relevant specifications is not too troublesome or time consuming. We therefore currently research into developing guidelines for formal specification of PLC program components.

8. Acknowledgment

This research is financed by the ProViking research programme. Thanks also to all concerned staff at the studied companies for sharing their knowledge and code. Thanks to Isak Öberg and Olof Bergqvist for performing an interesting master thesis.

9. References

Olof Bergqvist and Isak Öberg. PLC function block survey of Swedish automotive industry. Master's thesis, Dept. Signals and Systems, Chalmers Univ. Technol., Göteborg, Sweden, 2007.

B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification – Model-Checking Techniques and Tools*. Springer, 2001.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

Georg Frey and Lothar Litz. Formal methods in PLC programming. In *Proc. Int. Conf. Syst., Man, Cybern.*, pages 2431–2436, Nashville, TN, USA, 2000.

IEC. Programmable Controllers—Part 3: Programming languages. International standard IEC 61131-3. International Electrotechnical Commission, second edition, 2003.

Dick Johnson. Nano devices lead assault on traditional PLC applications. *Control Engineering*, 49(8):43 44, 2002.

Seungjoo Lee, Mark Adam Ang, and Jason Lee. Automatic generation of logic control. Technical report, Ford Motor Co., Univ. of Michigan and Loughborough Univ., 2006.

Robert W. Lewis. Programming industrial control systems using IEC 1131-3 Revised edition. The Institution of Electrical Engineers, 1998.

Robert W. Lewis. *Modelling Control Systems Using IEC 61499*. The Institution of Electrical Engineers, 2001.

Oscar Ljungkrantz and Knut Åkesson. A study of industrial logic control programming using library components. In *Proceedings of the 3rd Annual IEEE Conference on Automation Science and Engineering*, pages 117–122, Scottsdale, AZ, USA, 2007.

Oscar Ljungkrantz, Knut Åkesson, and Martin Fabian. Formal specification and verification of components for industrial logic control programming. In *Proceedings of the 4th IEEE Conference on Automation Science and Engineering*, pages 935– 940, Washington DC, USA, 2008.

M.R. Lucas and D.M. Tilbury. A study of current logic design practices in the automotive manufacturing industry. *Int. J. Human-Computer Studies*, 59(5):725–753, 2003.

Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

Kenneth L. McMillan. *The SMV language*. Cadence Berkeley Labs, 1999. URL http://www.kenmcmil.com/language.ps.

Mostafa G. Mehrabi, A. Galip Ulsoy, and Y. Koren. Reconfigurable manufacturing systems: Key to future manufacturing. *J. Intelligent Manufacturing*, 11(4):403–419, 2000.

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

Johan Richardsson and Martin Fabian. Modeling the control of a flexible manufacturing cell for automatic verification and control program generation. *J. of Flexible Service and Manufacturing*, 18(3):191–208, 2006.

Programmable Logic Controller

Edited by Luiz Affonso Guedes

ISBN 978-953-7619-63-3
Hard cover, 170 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

Despite the great technological advancement experienced in recent years, Programmable Logic Controllers (PLC) are still used in many applications from the real world and still play a central role in infrastructure of industrial automation. PLC operate in the factory-floor level and are responsible typically for implementing logical control, regulatory control strategies, such as PID and fuzzy-based algorithms, and safety logics. Usually PLC are interconnected with the supervision level through communication network, such as Ethernet networks, in order to work in an integrated form. In this context, this book was written by professionals that work and research in automation area and it has two major objectives. The first objective is present some advances in methodologies and techniques for development of industrial programs based on PLC. The second objective is present some PLC-based real applications from various areas such as manufacturing system, robotics, power system, communication system, and education.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Oscar Ljungkrantz, Knut Åkesson and Martin Fabian (2010). Practice of Industrial Control Logic Programming using Library Components, Programmable Logic Controller, Luiz Affonso Guedes (Ed.), ISBN: 978-953-7619-63-3, InTech, Available from: http://www.intechopen.com/books/programmable-logic-controller/practice-of-industrial-control-logic-programming-using-library-components

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen