Towards evolvable Control Modules in an industrial production process

Production control software based on Normalized Systems Theory

Dirk van der Linden[1], Herwig Mannaert[2], Jan De Laet[1]

[1]*Electro Mechanics Research Group*
Artesis University College of Antwerp
Antwerp, Belgium
dirk.vanderlinden, jan.delaet@artesis.be

[2]*Department of Management Information Systems*
University of Antwerp
Antwerp, Belgium
herwig.mannaert@ua.ac.be

***Abstract*—Normalized Systems theory has recently been proposed to engineer evolvable information systems. This theory includes also a potential of improvement in control software for the automation of production systems. In production control systems, the end user has always the right to have a copy of the source code. However, it is seldom manageable to fluently add changes to these systems, due to the same problems as information systems: couplings, side-effects, combinatorial effects, etc. Finding solutions for these problems include several aspects. Some standards like ISA 88 suggest the use of building blocks on macro level. The OPC UA standard enables these building blocks to communicate and interoperate over the borders of the hosting controllers via local networks or the internet. Consequently, production data collection, manual interfaces and recipe driven production control systems become web service based. Finally the Normalized Systems' theory suggests how these building blocks should be coded on micro level. This paper introduces a control module, based on a design pattern for flexible manufacturing and the principles of Normalized Systems for evolvable software.**

***Keywords*-Normalized Systems, Automation control software, PLC, ISA 88, IEC 61131-3, OPC UA.**

I. INTRODUCTION

Industrial communication has in the last 10 years become a key point in modern industry. A continually growing number of manufacturing companies desire, even require, totally integrated systems. This integration extends from electronic automation field devices (PLC: Programmable Logical Controller, PAC: Programmable Automation Controller, DCS: Distributed Control System) to Human Machine Interfaces (HMI) culminating into supervision, trending, and alarm software applications (SCADA: Supervisory Control And Data Acquisition and MES: Manufacturing Execution System). Industrial communication is implemented from field management via process management to Enterprise Resource Planning (ERP) applications (business management).

Just like transaction support software and decision support software systems, production automation systems have also a tendency to evolve to integrated systems. Tracking and tracing production data is not only improving the business, in some cases it is also required by law (e.g., sectors like food and pharmacy). Because of the scope of totally integrated systems (combination of information systems and production systems) the amount of suitable single vendor systems is low or even non-existing. Large vendor companies may offer total integrated solutions, but mostly these solutions are assembled with products with another history (merged companies or SMEs - Small and Medium Enterprises - bought by larger groups). For the engineer, this situation is very similar to a multi-vendor environment.

Globalisation is bringing opportunities for companies who are focussing their target market on small niches, which make part of a totally integrated system. These products can expand single-vendor systems, or can become part of a multi-vendor system. Moreover, strictly single-vendor systems are rather rare in modern industry. Sometimes they are built from scratch, but once improvements or expansions are needed, products of multiple vendors might bring solutions. Over time, single vendor systems often evolve to multi vendor systems. Minor changes, often optimizations or improvements of the original concept, occur short after taking-in-service. Major changes occur when new economical or technological requirements are introduced over time. As a consequence, software projects should not only satisfy the current requirements, but should also support future requirements [1].

The scope of changes in production control systems, or the impact of changes to related modules in a multi-vendor environment is typically smaller than in ERP systems and large supply chain systems. However, there is a similarity of the problem of evolvability [2]. Since the possibilities of industrial communication increases, we anticipate to encounter similar problems like in business information systems. The more the tendency of vertical integration (field devices up to ERP systems) increases, the more the impact of changes on production level can increase. Since OPC UA (interOperability Productivity and Collaboration - Unified Architecture) enables web based communication between field controllers

 ISBN: 978-1-61208-124-3

and all types of software platforms, over local networks or the internet, the amount of combinatorial effects after a change can rise significantly (change propagation). Based on the systems theoretic concept of stability, a software engineering theory is proposed to engineer evolvable information systems [1]. Although the theory was developed towards business information systems, it has an abstract and generic fundament. Consequently, it should be applicable for production automation control systems too.

This paper introduces a proof of principle on how the software of a production control module can be developed following the principles of Normalized Systems. Some developers could recognize parts of this approach, because it needs to be emphasized that each of the Normalized Systems theorems is not completely new, and even relates to the heuristic knowledge of developers. However, formulating this knowledge as theorems that cause combinatorial effects, supports systematic identification of these combinatorial effects so that systems can be built with minimal combinatorial effects [1]. Normalized Systems allows the handling of a business flow of entities like orders, parts or products. For these process-oriented solutions 5 patterns for evolvable software elements are defined [2]. In this paper however, we focus on the control of a piece of physical equipment in an automated production system. The code of an ISA 88 based control module is not process-oriented but equipment-oriented. The focus of this code is not about how a product has to be made, but about how the equipment has to be controlled. Consequently, we need another type of design patterns. Moreover, we need another type of programming languages because of the nature of industrial controllers. In Section II, we will give an overview of industrial standards on which industrial production control modules can be based. These standards include software modelling and design patterns, communication capabilities, and programming languages. In Section III, an evolvable control module is introduced, including a discussion of change drivers. In Section IV, some changes are implemented. We tested in our lab industrial automation the robustness of the control module against these changes. In Section V we evaluated the proof of principle against the principles of Normalized Systems. During this evaluation, the Design Theorems for Software Stability [2] are used as criteria.

II. INDUSTRIAL STANDARDS

Manufacturing operations can be generally classified into one of three different processes: discrete, continuous, and batch. On October 23, 1995, the SP88 committee released the ANSI/ISA-S88.01-1995 standard [3] to guideline the design, control and operation of batch manufacturing plants. The demand of the users for production systems with a high flexibility and a high potential of making product variants, became important. Process engineers focus on how to handle the material flow to meet the specs of the end-product. Control system experts focus on how to control equipment. To improve the cooperation of both groups, the SP88 committee had isolated equipment from recipes. This provides the possibility of process engineers to make process changes directly, without the help of a control system expert (reducing the setup-costs). This provides also the ability of producing many product-variants with the same installation (increasing the target market). Expensive equipment can be shared by different production units (enough reducing the production costs). This approach opened the way to what has been called ”agile manufacturing”. The utilization of ISA 88 data models simplify the design process considerably [7].

Despite the useful ISA 88 terminology and models to structure flexible manufacturing, different interpretations are possible. The standard does not specify how the abstract models should be applied in real applications. Implementers sometimes develop recipes and procedures, which are far more complex than necessary. Since 1995 there have been many applications and a commonly accepted method for implementing the standard has emerged. The S88 design patterns [5] of Dennis Brandl (2007) address this. These patterns might decrease the tension of implementers to make their recipes and procedures more complex than necessary. Unfortunately the part, which describes the connection between computer network systems and control network systems, is limited.

This is where the OPC interfaces come into play. OPC UA is considered one of the most promising incarnations of WS technology for automation. From the very beginning, OPC UA was intended as system interface, aggregating and propagating data through different application domains. Its design, thus, takes into account that the field of application for industrial communication differs from regular IT communication: embedded automation devices such as PLCs, PACs or DCSs provide another environment for web-based communication than standard PCs.

The fundamental components of OPC UA are different transport mechanisms and a unified data modelling [4]. The transport mechanisms tackle platform independent communication with the possibility of optimization with regard to the involved systems. While communication between industrial controllers or embedded systems may require high speed, business management applications may need high data volumes and firewall friendly transport. As a consequence, two data encoding schemes are defined, named OPC UA Binary and OPC UA XML [9].

Data modelling defines the rules and base building blocks necessary to expose an information model with OPC UA. Rather than support data communication, it facilitates the conversion of data to information. The OPC Foundation avoids the introduction of unnecessary new formalisms. Instead, definitions of complex data based on related industrial standards are encouraged. Examples are FDI (Field Device Integration), EDDL (Electronic Device Description

 ISBN: 978-1-61208-124-3

Language), IEC 61131-3 (PLC programming languages) and ISA 88 (batch control). Basically, an OPC UA information model has nodes and references between nodes. Nodes can contain both online data (instances) and meta data (classes). OPC UA clients can browse through the nodes of an OPC UA server via the references, and gather semantic information about the underlying industrial standards. For clients, it is very powerful to program against these complex data types, it brings a potential of code-reuse.

The lowest level of the ISA 88 control hierarchy is the control module. Control modules perform two primary functions: they provide an interface with the physical devices, and they contain basic control algorithms. Control modules encapsulate basic control algorithms and the I/O interface to the actual physical devices. The most common method of programming control modules is any of the IEC 61131-3 programming languages [6]. This standard specifies the syntax and semantics of a unified suite of programming languages for programmable controllers. These consist of two textual languages, IL (Instruction List, has some similarity with assembler) and ST (Structured Text, has some similarity with C or pascal), and two graphical languages, LD (Ladder Diagram, has some similarity with electrical schemes) and FBD (Function Block Diagram, is based on boolean algebra). Industrial programmable controllers are based on divers, often dedicated, operating systems and vendor-dependent programming environments. Besides, earlier days every controller had its own programming language. The release of IEC 61131-3 addresses the problem of too many different programming languages for similar solutions with controllers of different brands. The code of the proof of principle of this paper is written in these languages. However, we emphasize that using the IEC 61131-3 languages is not enough for a 'best practice' implementation. One of the available modelling concepts to analyse the problem and structuring the solution will considerable improve an implementation [11].

III. EVOLVABLE CONTROL MODULES

An invalid function call because the function meanwhile has an updated parameter set is an issue what happens in PLC programming as well as in IT software. Other problems on evolving software occur as well. One of the most annoying problems an automation service engineer confronts is the fear to cause side-effects with an intervention. They have often no clear view on how many places they have to adapt code to be consistent with the consequences of a change. Some development environments provide tools like cross references to address this, but the behaviour of a development environment is vendor-dependent, although the languages are typically based on IEC 61131-3.

In this section we introduce a control module for a motor. We aim to make this motor control software module as generic as possible. In stead of introducing new formalisms, we based our proof of principle on existing standards. For the modelling, we used concepts of ISA 88 (IEC 61512), for interfacing, we used OPC UA (IEC 62541), and for coding we used IEC 61131-3. More specific, we used the S88 design patterns [5] (derived from ISA 88) because these patterns can be used not only in batch control, but also for discrete and continue manufacturing. None of these standards contain suggestions on how the internal code of a control module should be structured. We introduce a granular structure following the theorems of Normalized Systems. Since the process-oriented approach of evolvable elements for business software [2] is not applicable in our equipment-oriented controller code, we neglect these patterns and use design patterns based on ISA 88 [5]. Every task (action), which must be done by the control module, is coded in a separated POU (Program Organization Unit, sort of subroutine [6]).

In the most elementary form control modules are device drivers, but they provide extra functions like manual/automatic mode, interlocking (permissions), alarming, simulation, etc. [8]. We used the design pattern of Figure 1. This state machine is very simple, when the control systems powers on, the motor comes in the 'off' state. It can be started and stopped via the 'on' and 'off' commands. Hardware failures can cause the motor to go to the 'failed' state, from where a 'reset' command is needed to return to the 'off' state. The concept of this 'failed' state brings us a very important benefit: process safety. Besides, it forms the base for failure notification [10]. This functionality is implemented in a function block we called 'StateAction'. This function block has only one parameter we called 'Device'. The datatype of this parameter is called 'DeviceDataType'. Only a part of this complex datatype is used in the function block 'StateAction'. We called this part 'StateType' (Figure 2). For every other action like controlling the hardware or handling the modes (see further), we have a similar datatype. All action related datatypes are merged into one overall datatype. Exchange of data between the actions can be done via this DeviceDataType (stamp coupling), however without crossing the borders of the control module.

It is obvious that both commands (arrows) and states of Figure 1 are represented as a boolean value in this datatype. One parameter is passing all the necessary data for performing one task: the state action of the control module.

The primary function of our control module is not fulfilled yet: controlling the physical device, in our example the motor, or more general the device hardware. Controlling the hardware is another task in another function block. This function block is receiving the same single parameter 'Device', but it uses another part. The content of the code and the datatype is again very limited (Figure 3). This is one of the key-points of normalized systems: building the application starting from very small modules, performing only one task.

 ISBN: 978-1-61208-124-3

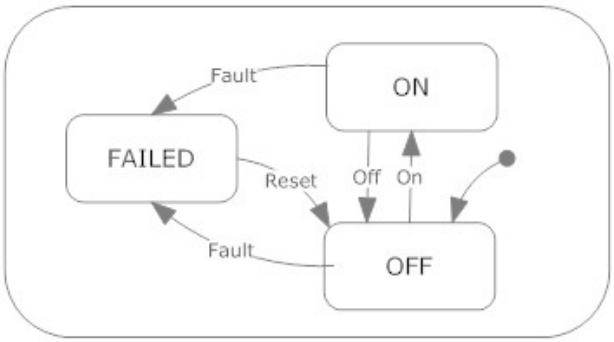

Figure 1: Example of a motor state model [5]

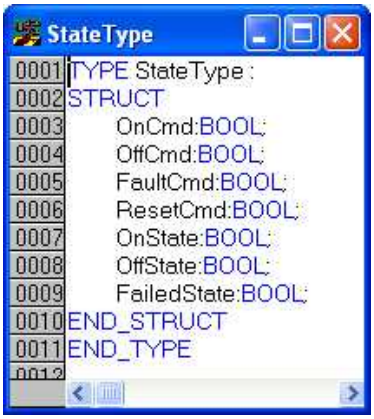

Figure 2: Structure of the datatype 'StateType'

The complex datatype 'DeviceDataType' encapsulates the datatypes 'StateType' and 'HardwareType'. The resulting building block, the Control Module, is connected to only one parameter: an instance of 'DeviceDataType'. This block contains only one action: DeviceAction. This complex action contains two actions: StateAction and HardwareAction, each performing one task. But what is a task? We propose that the definition of tasks can be derived from change drivers. Every interface to our control module can cause or have influence of a change. In our case the change drivers are exposed in Figure 4.

The change driver 'physical equipment' forms the base for the task 'HardwareAction'. The state commands and states forms the base for the task 'StateAction'. Figure 4 suggest another change driver. If we allow low level HMI, the operator becomes the incarnation of a change driver. This means that we add a state machine (ModeAction) to deal with manual/automatic modes, and a subroutine (CommandAction) to separate the commands of the operator (manual commands) and the commands of a higher entity in the automation control project (automatic commands). Following ISA 88 this should be an Equipment Module.

Change drivers have influence on the data structure. To add the functionality above, we need a part 'ModeType' and 'CommandType' in the complex datatype 'DeviceDataType'. So we end up in a Control Module with 4 datatypes (StateType, HardwareType, ModeType, CommandType) encapsulated by the datatype DeviceDataType (Figure 5).

These datatypes are passing all actions, but have all one corresponding action: StateAction, HardwareAction, ModeAction and CommandAction.

IV. ADDING CHANGES

A way to test evolvability is just adding changes and evaluating the impact of these changes. We have 3 actors on our control module: the operator (manual mode), the hardware and the equipment module (automatic mode). Every actor can change his behaviour or can have new expectations or can do new requests. Moreover, one should be able to debug without causing side-effects on other or older features. In general, we start with a first version. Then we maintain one or more running instances of the control module with initial expected behaviour. Second, we consider the addition of a change, and consequently a possible update of the datatype, existing actions or introduction of a new action. Finally, we make a new instance, check the new functionality and the initial expected behaviour of the older instances as well.

We considered the situation that manual operations could harm automatic procedures. For instance, stopping our motor manually could confuse an algorithm if it is happening during a dosing action. To prevent this, we add the feature manual lock. This means, we still support manual mode, but we disable manual mode during the period a software entity like an equipment module requires this.

Without removing the calls of instances, which dont need this feature, we added a command 'ManLockcmd' to the datatype 'CommandType'. Consequently, this new command becomes part of the overall 'DeviceDataType', so it is passing all actions, but only ModeAction is doing something with this new command.

We considered the situation of a motor instance, which must be able to run in two directions. Again, without removing the calls of instances of single-direction motors, we added a hardware tag 'reverse' to the HardwareType and the commands 'ManReverseCmd' and 'AutoReverseCmd' to the CommandType. In the StateType the tags 'ReverseCmd' and 'ReverseState' were added. As expected, we had to adapt some code in the function blocks 'HardwareAction', 'CommandAction' and 'StateAction'.

On a similar way, we performed other changes like the use of another fieldbus, which required mappings to new hardware addresses. We also introduced a new action SimAction, made (for software testing purposes) to neglect the Fault command (FaultCmd) if no hardware is connected.

 ISBN: 978-1-61208-124-3

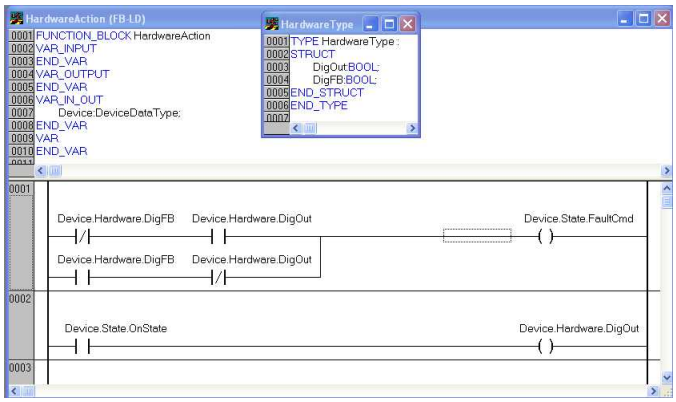

Figure 3: HardwareAction and HardwareType

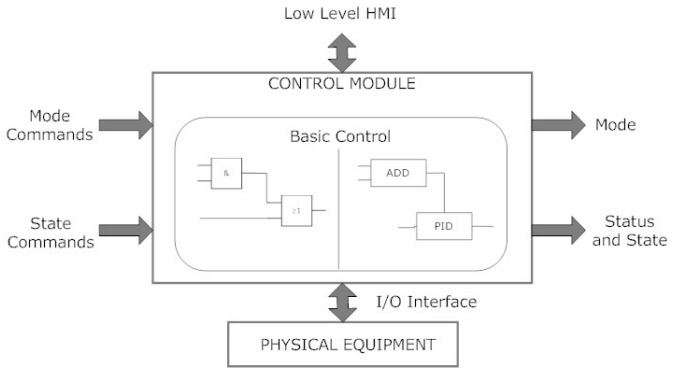

Figure 4: Change drivers of a control module [5]

V. EVALUATION ACCORDING TO THE PRINCIPLES OF NORMALIZED SYSTEMS

Since it is not possible to anticipate on all changes, we cannot test all future change cases. Besides, we are aware that our proof of principle is performed on lab scale, and other, real life situations can occur in a real industrial automation project. In our evaluation we made the assumption that code, following the principles of normalized systems will evolve better than systems, which are not respecting these principles. As a consequence, we checked whether the code is respecting these rules.

First, we consider the separation of concerns. An action entity can only contain a single task. In contrast with industrial usage, where a control module is often just one POU, we made 4 (or more) separate function blocks who are encapsulated by one overall module. For the definition of a task, we based the primary actions on change drivers. Later on, we added the simulation action, which was not directly related to a change driver and thus could be added.

Second, we looked at data version transparency. Data entities that are received as input or produced as output by action entities, need to exhibit version transparency. Only one complex parameter is passed to the control module. Obviously deleting or changing the name of the parameter would destroy existing running connections. Whether adding a parameter would destroy a running instance is vendor-dependent. We stick to one parameter during all changes. Four (later on five) structs are nested. All actions can see the data passing, but every action just picks the data needed for the specific task. We never changed tags, we only added tags. Because of this, earlier instance calls were not affected by data type conflicts.

Third, following the theorem of action version transparency: action entities that are called by other action entities, need to exhibit version transparency. When we changed code, we always bewared for not harming the original functionality. For example, we never erased a state or transition in a state machine, we only added new states and/or transitions. More specific, besides the new functionality, we checked the initial behaviour of existing instances as well.

Fourth, separation of states: the calling of an action entity by another entity needs to exhibit state keeping. It is obvious that the StateAction is managing his state, and keeping it in the related StateType instance. The CommandAction is resulting in a command for the StateAction. The HardwareAction is resulting in a command for the physical hardware. Finally, similar with the StateAction it is obvious that also the ModeAction is managing his state, and keeping it in the related ModeType instance. Besides, the later on added SimAction results in the tags 'SimOnState' and 'SimOffState'.

VI. CONCLUSION AND FUTURE WORK

Evolvability of software systems is important for IT systems, but also a relevant quality value for industrial automation systems. Function blocks of automation systems are programmed close to the processor capabilities. For example, there is a similarity with the IEC 61131-3 language Instruction List (IL) and assembler. The key point of Normalized Systems is a large granularity of software modules, with a structure, which is strictly disciplined to the related theorems. As a consequence, making a proof of principle close to the processor is a very informative exercise to concretize the principles of normalized systems. Besides, this approach can be of great value for improving the quality of industrial automation software projects.

It must be stated that implementing these concepts were highly facilitated by the use of existing industrial standards. They provide us methods to develop the macro-design of software modules, while Normalized Systems provide us guidelines for the micro-design of the actions and data structures encapsulated in these modules. Adding functionality or even adding an action to a (macro) building block, in our case the Control Module, can be done with a limited impact (micro manageable) towards other (macro) entities (bounded impact). To define the most basic actions (tasks)

 ISBN: 978-1-61208-124-3

Figure 5: Structure DeviceDatatype

and data structures, the identification of the change drivers of the concerned entity, in our case represented by the different interfaces to external entities, is essential. This confirms the first theorem for software stability, separation of concerns.

Our future work will be focused on other (macro) elements of ISA 88, which contain other types of control. A Control Module contains mainly basic control, together with a limited coordination control (the mode). We will study on elements with more advanced coordination control code and procedural control, again developed and tested following the principles of Normalized Systems.

Moreover, future work will also be focused on interfaces. Since OPC UA is very generic, we wonder if constraints should be added to the standard to let data communication be compliant to the second theorem of software stability, data version transparency. We wonder whether both currently existing OPC UA transport types, UA binary and UA XML, can be done in a data transparent way.

ACKNOWLEDGMENT

The authors thank Marc Martens (Artesis lab industrial automation), for building the hardware mini-processes needed for performing the testing of this paper, and the good collaboration.

REFERENCES

[1] van Nuffel Dieter, Mannaert Herwig, de Backer Carlos, Verelst Jan. ”Towards a deterministic business process modelling method based on normalized theory” International journal on advances in software - ISSN 1942-2628 -3:1/2(2010), p. 54-69

[2] Mannaert Herwig, Verelst Jan. ”Normalized Systems Recreating Information Technology Based on Laws for Software Evolvability” Koppa, 2009.

[3] ANSI/ISA-88.01-1995, Batch Control Part 1: ”Models and Terminology.”

[4] Mahnke Wolfgang, Leitner Stefan-Helmut, Damm Matthias. ”OPC Unified Architecture”, Springer, 2009.

[5] Brandl Dennis. ”Design patterns for flexible manufacturing”, ISA, 2007.

[6] International Electrotechnical Commission (IEC). ”IEC 61131-3, Programmable controllers- part 3: Programming languages”, Edition 2.0, 2003-01.

[7] Juoku Virta, Ilkz Seilonen, Antti Tuomi, Kari Koskinen. ”SOA-Based Integration for Batch Process Management with OPC UA and ISA-88/95”, 15th IEEE International Conference on Emerging Technologies and Factory Automation, september 13-16, 2010, Bilboa, Spain.

[8] Larry Lamb, Jim Parshall. ”Applying S88 - Batch Control from User’s Perspective”, ISA, 2000

[9] OPC Foundation. ”OPC Unified Architecture, Part1: Overview and Concepts”, Release 1.01, february 2009.

[10] Clark Case, Rockwell Automation. ”Applying ISA S88 to Small, Simple Processes”, World Batch Forum conference 13-15 nov 2006, Zemst, Belgium

[11] David Friedrich, Birgit Vogel-heuser. ”Benefit of system modeling in automation and control education”, American Control Conference, 2007, New York City, USA.

 ISBN: 978-1-61208-124-3