Towards Normalized Connection Elements in Industrial Automation

Dirk van der Linden[1], Herwig Mannaert[2], Wolfgang Kastner[3], Herbert Peremans[2]

[1]Artesis University College of Antwerp, Electromechanics Research Group, Belgium
dirk.vanderlinden@artesis.be
[2]University of Antwerp, Belgium
{herwig.mannaert, herbert.peremans}@ua.ac.be
[3]Vienna University of Technology, Automation Systems Group, Austria
k@auto.tuwien.ac.at

Abstract—**There is a tendency to Web-enable automation control systems, with consequently the challenge to propagate and aggregate data and control over the Internet. While classical industrial controller systems are limited to a local network, Web-enabled systems can be coupled in a new dimension. However, this also introduces larger impacts of changes and combinatorial effects. The Normalized Systems theory was recently proposed with the explicit goal of keeping these impacts bounded. It can be applied from the production control level up to the Web-enabled interface. One of the key principles of the Normalized Systems theory is to enforce Separation of Concerns in a multi-technology environment. To this end, this paper introduces Normalized Connection Elements as a stable interface between PLC software and field devices. As a case in point, the IEC 61131-3 code design of an ISA88 Control Module following these principles is discussed.**

Keywords-**Normalized Systems; Automation control software; IEC 61131-3; ISA88; OPC UA.**

I. Introduction

Meeting the requirements of a software project has always been one of the top priorities of software engineering. However, not rarely, after taking in service, or even during the development, customers come up with new requirements. Project managers try to satisfy these additional requirements accompanied with an extra cost to the customer. The estimation of these additional efforts, depending on the development progress of the project, is often not straightforward. Managers tend to focus on functional requirements, while experienced engineers know that non-functional requirements can sometimes cause more efforts and costs. Evolvability became one of the most desirable non-functional requirements in software development, but is hard to control. One of the most annoying problems automation engineers are confronted with is the fear to cause side-effects with an intervention [1]. They have often no clear view in 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 behavior of a development environment is vendor-dependent, although the programming languages are typically based on IEC 61131-3 [2].

The Normalized Systems theory has recently been proposed for engineering evolvable information systems [3]. This theory also has the potential to improve control software for the automation of production systems. In production control systems, the end user always has the right to a copy of the source code. However, it is seldom manageable to incrementally add changes to these systems, due to the same problems as we see in business information systems, such as undesired couplings, side-effects, combinatorial effects. Finding solutions for these problems includes several aspects. Some standards like ISA88 suggest the use of building blocks on the macro level. The Normalized Systems theory suggests how these building blocks should be coded on the micro level. Interfacing between modules can be supported with the OPC UA standard.

Just like transaction support software and decision support software systems, production automation systems also have 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 (in particular in the food and pharmacy sectors). Due to 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 totally integrated solutions, but mostly these solutions are assembled from products with different history. For the engineer, this situation is very similar to a multi-vendor environment. Also, assembling software instead of programming is a challenge Doug McIlroy called for already decades ago [4]. Several guidelines, approaches, tools and techniques have been proposed that aim at assisting in achieving this goal. Unfortunately, none of these approaches have proven to be truly able to meet this challenge.

Globalization is bringing opportunities for companies who are focusing their target market on small niches, which are 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. Hence, over time single-vendor systems often evolve to multi-vendor systems. Each of these systems can be considered as a different technology. Isolating these technologies to prevent them exporting the impact of their internal changes into other technologies is the key contribution of this paper.

Minor changes, often optimizations or improvements of the original concept, occur shortly 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 [5].

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 (Enterprise Resource Planning) systems and large supply chain systems. However, there is a similarity in the problem of evolvability [3]. Since the possibilities of industrial communication increase, we anticipate to encounter similar problems to the ones 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 the production level can increase. Since OPC UA (Open Product Connectivity - Unified Architecture) [6] enables Web-based communication between field controllers 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).

This paper introduces a proof of principle on how the software of an ISA88 Control Module [7] can be developed following the Normalized Systems theory. Some developers could recognize parts of this approach, because (as should be emphasized) each of the Normalized Systems theorems is not completely new, and some even relate to the heuristic knowledge of developers. However, formulating this knowledge as theorems that prevent combinatorial effects supports systematic identification of these combinatorial effects so that systems can be built to exhibit a minimum of these combinatorial effects [3]. The Normalized Systems theory allows the handling of a business flow of entities like orders, parts or products. For these process-oriented solutions, five patterns for evolvable software elements are defined [8]. In this paper, however, we focus on the control of a piece of physical equipment in an automated production system. The code of an ISA88 based Control Module is not process-oriented but equipment-oriented [1]. 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 programming language because of the nature of industrial controllers. Since the patterns for evolvable software elements are fundamental, we can use them as a base for IEC 61131-3 code. For this code, we concentrate on 3 patterns: Data Elements, Action Elements and Connection Elements. The Connection Elements can connect software entities in two directions: first, towards the physical process hardware, and second, towards higher level, non-IEC 61131-3 software modules. The first concerns IEC 61131-3 code, the second typically Web-enabled platform independent systems via an OPC UA interface. In this paper, we focus on the connection with process physical hardware. The possibility of OPC UA-based Connection Elements is crucial to enable upcoming larger automation systems, whose parts are connected via the Internet, and will be worked out in detail in future work. Such automation software entities should be able to evolve over time. This is a key requirement in the beginning age of decentralized energy generators and consumers prominently known as smart grid [9].

The remainder of this paper is structured as follows. In Section II, we discuss the Normalized Systems theory. In Section III, we give an overview of industrial standards on which industrial production Control Modules can be based. These standards include software modeling and design patterns, communication capabilities, and programming languages. In Section IV, an evolvable Control Module is introduced. In Section V, we discuss some changes and their evaluation. We tested the robustness of the Control Module against these changes in our industrial automation laboratory. In Section VI, we conclude and introduce suggestions for future research.

II. Normalized Systems

Adding small changes or extending an existing software system with new functionality often leads to an increase in architectural complexity and a decrease in software quality [10]. This is regarded as more expensive than developing the same functionality from scratch. This phenomenon is known as Lehman's law of increasing complexity [11], expressing the degradation of information systems' structure over time:

> *"As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."*

To challenge this law, the Normalized Systems theory has recently been established [12]. Since the Normalized Systems theory takes *modularity* as basis, the principles are independent of any specific programming language, development environment or framework. Modularity implies that every module hides its internal details from its environment: another module does not need a white box view (i.e., analysis of the internal data and code) of the first module in order to be able to call and use this module. Hiding internal details is referred to as the *black box* principle. The user or caller of a black box module only needs to know the

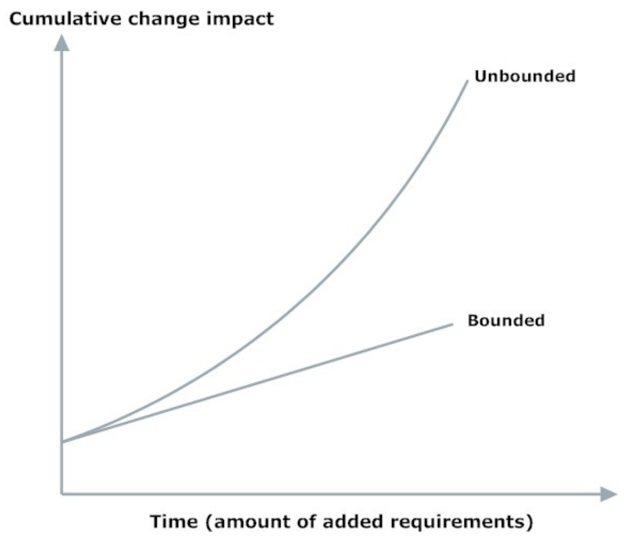


Figure 1: Cumulative impact over time [13]

interface of the module, i.e., the name of the module, the input and output parameters [3].

A. Stability

The starting point of the theory is system stability. In systems theory, one of the most fundamental properties of a system is its stability: in a stable system, a bounded input function results in bounded output values, even for t $\rightarrow \infty$ (with t representing time) [5]. This means that a limited set of changes, needed for maintenance or extension of the system, results in a limited amount of code changes or impacts to the system, even for t $\rightarrow \infty$. This includes the absence of side-effects in modules which are not changed, independent of the size of the system.

Stability demands that the impact of a change only depends on the nature of the change itself. Conversely, changes causing impacts that increase with the size of the system can be termed *combinatorial* effects and should be eliminated from the system in order to attain stability. Stability can be seen as the requirement of a linear relation between the cumulative changes and the growing size of the system over time. Combinatorial effects or instabilities cause this relation to become exponential (Figure 1). Systems that exhibit stability are defined as *Normalized Systems* [3].

> *”Normalized systems are systems that are stable with respect to a defined set of anticipated changes, which requires that a bounded set of those changes results in a bounded amount of impacts to system primitives.”*

The challenge to control the impact of changes starts with identifying the changes systematically. Here, it is interesting to know the source or cause of a change. In terms of modularity, it is useful to know which parts of a module are changing independently. We should limit the size of a module to a cohesive part of content, which is changing independently of every other part. A cause or source of a change, which can be considered independently, to a software primitive can be called a ‘change driver’.

B. Design theorems for Normalized Systems

Derived from the postulate that *a system needs to be stable with respect to a defined set of anticipated changes*, four design theorems or principles for the development of Normalized Systems are defined. They are briefly summarized in the following. A more detailed discussion can be found in the paper by Mannaert et al. [12].

1) Separation of Concerns: *An Action Entity shall only contain a single task.*

   The identification of a task is to some extent arbitrary. The concept of change drivers brings clarity here, because every Action Entity should only evolve because of a single change driver. Every task can evolve independently. If two or more aspects of a functionality are considered to evolve independently, they should be separated. It is proven that if one action contains more than one task, an update of one of the tasks requires updating all the others, too.

2) Data Version Transparency: *Data Entities that are received as input or produced as output by Action Entities shall exhibit Version Transparency.*

   We now concentrate on the interaction between Data Entities and Action Entities, more precisely, whether the passing of parameters or arguments affects the functionality of a module. Data Version Transparency implies that Data Entities can have multiple versions without affecting the actions that consume or produce them. In more practical terms, merely adding a field to a set of parameters that is not used in a specific Action Entity should not affect that Action Entity.

3) Action Version Transparency: *Action Entities that are called by other Action Entities shall exhibit Version Transparency.*

   In this theorem we concentrate on the interaction of Actions Entities with other Action Entities. Action Version Transparency implies that an Action Entity can have multiple versions, without affecting the actions that call the Action Entity. In other words, the mere introduction of a new version of an Action Entity or task should not affect the Action Entities calling the Action Entity containing the task.

4) Separation of States: *The calling of an Action Entity by another Action Entity shall exhibit State Keeping.*

   We continue to concentrate on the interaction of Action Entities with other Action Entities, more specifically on the aggregation or propagation of Action Entities. Every Action Entity itself is responsible for remembering the calling of other Action Entities, and consequently the corresponding state. To comply with this theorem, a chain of actions calling other actions should always be asynchronous. Besides, asynchronous processing is usually associated with high reliability, and even performance. The latter is a result of avoiding locking resources related to one task during the execution of (an)other task(s).

C. Encapsulation of Software Entities

On the level of individual Action Entities, Theorems 2 and 3 (Data and Action Version Transparency) avoid instabilities caused by different versions of data and tasks. On the level of aggregations and propagations, Theorems 1 and 4 (Separation of Concerns and States) avoid unstable interactions between software constructs. Since software entities complying with Theorem 1 are very small, their application results in a highly granular structure. On the application oriented level, there is a need for larger building blocks, which are not focused on actions with only one small task, but on higher-level elements. The Normalized Systems Elements are manifestations of encapsulations, which represent each a typical building component in a software system:

- Data Encapsulation: This is a composition of software constructs to encapsulate Data Entities into a Data Element. Such a Data Element can also contain methods to access the data in a Version Transparent way, or can contain cross-cutting concerns – in separate constructs.
- Action Encapsulation: This is a composition of software constructs to encapsulate Action Entities into an Action Element. There can be only one construct for the core task (core Action Entity), which is typically surrounded by supporting tasks (supporting Action Entities). Arguments or parameters of the individual Action Entities need to be encapsulated as a Data Element for use in the entire Action Element.
- Connection Encapsulation: This is a composition of software constructs to encapsulate Connection Entities into a Connection Element. Connection Elements can ensure that external systems can interact with Data Elements, but can never call an Action Element in a stateless way. The concept of Connection Encapsulation allows the representation of external systems in several co-existing versions, or even alternative technologies, without affecting the Normalized System.
- Flow Encapsulation: This is a composition of software constructs to create an encapsulated Flow Element. Flow Elements cannot contain other functional tasks but the flow control itself, and they have to be stateful.
- Trigger Encapsulation: This is a composition of software constructs to create an encapsulated Trigger Element. Trigger Elements control the separated – both error and non-error – states, and decide whether an Action Element has to be triggered.

III. INDUSTRIAL STANDARDS

A. PLC coding with IEC 61131-3

Since their introduction in the late 1960s, PLCs (Programmable Logic Controllers) have found broad acceptance across the industry. Because they are programmable, they provided a higher flexibility than the previous control equipment based on hardwired relay circuits. PLCs were produced and sold all over the world with a large diversity of vendors. The programming languages used to program PLCs of various brands were more or less similar, but due to a lot of implementation details, intensive trainings were needed if an engineer wanted to move from one vendor's system to another.

To unify the way PLCs are programmed, the IEC (International Electrotechnical Commission) introduced the IEC 61131 standard, which is a general framework that establishes rules all PLCs should adhere to, encompassing mechanical, electrical, and logical aspects, and consist of several parts. The third part (IEC 61131-3) deals with programming of industrial controllers and defines the programming model. It defines data types, variables, POUs (Program Organization Units), and programming languages. A POU contains code; it can be a Function, Function Block, or a Program.

Functions have similar semantics to those in traditional procedural languages and directly return a single output value. However, besides one or more input values, the Function may also have parameters used as outputs, or as input and output simultaneously. They cannot contain internal state information. Consequently, they can call other Functions, but no Function Blocks.

Function Blocks are similar to classes in object oriented languages, with the limitation of having a single, public, member function. Function Blocks are instantiated as variables, each with their own copy of the Function Block state. The unique member function of a Function Block does not directly return any value, but has parameters to pass data as input, output or bidirectionally. Since Function Blocks have internal memory, they can call both Functions and other Function Blocks.

Programs can contain all the programming language elements and constructs and are instantiated by the PLC system. They are cyclically triggered by the PLC system based on a configurable cycle time, or triggered by a system event. Typically they organize the progress of the PLC functionality during runtime, by calling Functions and Function Blocks.

All three types of POUs may be programmed in one of two textual languages (IL: Instruction Language; ST: Structured Text), or two graphical languages (LD: Ladder Diagram; FBD: Function Block Diagram). The standard also defines a graphical language for specifying state machines (SFC: Sequential Function Chart), that may also be used in Function Blocks or Programs. It should be noted that typically one of the other languages are used to code the SFC transition conditions and steps.

Programming in LD is similar to designing a relay based electrical circuit. It can be said that LD is a historical artifact. The very first PLCs were competing with existing control equipment based on hardwired relay circuits and therefore adopted a language similar to the design schematics of these electrical circuits in order to ease platform acceptance by the existing technicians.

The FBD language may be considered as a graphical incarnation of boolean algebra, where boolean OR, AND and more complex boxes are simply placed in the GUI. The inputs and outputs of the boxes are connected by drawing lines between them.

The IL language is similar to assembly. It is definitively a low level programming language, because it contains a *jump instruction*, which should be abolished from all “higher level” programming languages [14].

The ST language has a syntax similar to Pascal, and can be considered as a higher level language. Indeed, as proven by Dijkstra [14], there is no need for a *jump instruction* in ST because all processing algorithms can be implemented through three primitive types of control: *selection, sequencing, and iteration*.

B. Modeling with ISA88 (IEC 61512)

Manufacturing operations can be generally classified as one of three different processes: discrete, continuous, or batch. In October 1995, the SP88 committee released the ANSI/ISA-S88.01-1995 standard [7] (its international equivalent is IEC 61512) to guideline the design, control and operation of batch manufacturing plants.

The demand for production systems with a high flexibility, with regard to setting up the system for making product variants, became important. Process engineers focus on how to handle the material flow to meet the specifications of the end-product. Control system experts focus on how to control equipment. To optimize the cooperation of both groups, the SP88 committee wanted to separate product definition information from production equipment capabilities. Product definition information is contained in recipes, and the production equipment capability is described using a hierarchical equipment model. This provides the possibility for process engineers to make process changes directly,

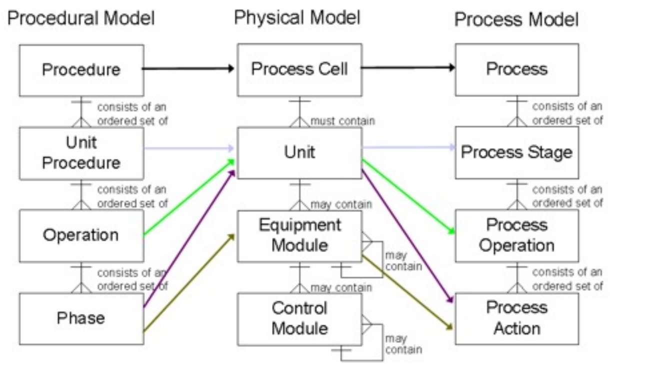


Figure 2: The relation between procedural, physical and process models [16]

without the help of a control system expert (reducing the setup costs). Moreover, the ability of producing many product variants with the same installation is achieved, increasing the target market. Expensive equipment can be shared by different production units (reducing the production costs). The utilization of ISA88 data models simplifies the design process considerably [15].

1) Challenges:

Despite the usefulness of ISA88 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 [15] of Dennis Brandl (2007) address this. These patterns can reduce the tendency of implementers to make their recipes and procedures more complex than necessary.

When automated control was introduced to manufacturing, it was accompanied by the problem that control system programming became a critical activity in both initial startup and upgrades. Often the physical equipment can be reconfigured in days, if not minutes, if manually controlled and maintained. In contrast, the automatic control system, unless it was designed for reconfiguration, may take weeks or months to reconfigure and reprogram [15].

Turning the ISA88 models into well structured code is not straightforward [1]. Again, different interpretations are possible. On the macro level they provide a clear structure, but there is a need for prescriptive specifications to convert these models into code, or even to divide them into smaller sub-modules at the micro level.

2) Important ISA88 models for automation control:

During the development of ISA88, the SP88 committee was focusing on batch control, but made the models universal enough to make them suitable for other process types [16]. However, to implement these models, different design patterns are recommended for the different process types [15].

The most important key-point of ISA88 is the separation of (end) product definition information from production equipment capability. This separation allows the same equipment to be used in different ways to make multiple products, or different equipment to be used to produce the same product. Recipes are used to describe the product definition information, and a hierarchical equipment model is introduced to describe the production equipment capability. A consequence of this approach is a separation of expertise. Experts in different domains, who have to cooperate to achieve the production goals, are educated differently and think in different ways. Process engineers focus on how to handle the material flow to meet the specifications of the end-product. Control system experts focus on how to control equipment. Recipes are to be developed by process engineers, and control system experts will have to make the equipment run, based on information contained in the parameters and procedures of the recipes. ISA88 provides a physical model hierarchy to deal with equipment oriented control, and a procedural model hierarchy to deal with process oriented control. For researchers, the process model is provided (Figure 2). In this paper, we focus on the lowest level equipment oriented element: the ISA88 Control Module.

3) ISA88 Control Modules:

The lowest level of the ISA88 physical model is the Control Module, but not all parts are necessarily physical. In an automated system, Control Modules are partly (PLC) software. In their simplest form, Control Modules are device drivers, but they can provide robust methods of device control too, including functions such as automatic and manual modes, simulation mode, Interlocks and Permissives (ISA88 terminology), alarming. Control Modules execute basic control and minimal coordination control. They perform no procedural control functions. The most common method of programming basic control are any of the IEC 61131-3 programming languages, such as LD, FBD, IL, ST. Control Modules usually make up the majority of control system code, but they are also the mechanism for defining significant amounts of reusable code [15].

Typically, at least two state machines are introduced for a Control Module, one for the state of the device itself (e.g., on, off, fail), and a second for the mode (e.g., manual and automatic). Normal operation of the Control Module should be commanded from an equipment module (the equipment oriented element right above Control Module in the ISA88 physical model); however, a Control Module may also be controlled manually. Thus, the Control Module may be in one of two modes – automatic or manual.

The ISA88 standards provide the models, but do not prescribe how these models should be coded. After the standard was released, many engineers applied the standards in ways that the original authors had not considered. To address this problem, the ISA88 models have been extended with the so-called S88 design patterns [15]. These patterns are not normative, but they are effectively applied in multiple industries. The design patterns have been applied in almost every kind of batch, discrete, and continuous manufacturing applications.

C. OPC Unified Architecture (IEC 62541)

Reusable software components made their entry in automation technology and replaced monolithic, customized software applications. These components are preferably connected by standardized interfaces. In the mid-1990s, the OPC Foundation was established with the goal to develop a standard for accessing real-time data under Windows operating systems, based on Microsoft's DCOM technology.

OPC has become the de facto standard for industrial integration and process information sharing [17]. By now, over 20,000 products are offered by more than 3,500 vendors. Millions of OPC based products are used in production and process industry, in building automation, and many other industries around the world [18]. However, in the period when Internet based systems were introduced, the DCOM technology resulted in limitations. To challenge these limitations, and truly support Web-enabled automation systems, a new standard family has recently been released: The OPC Unified Architecture.

Web-based technology is the key to taking interoperability to a new level. Web Services (WS), are totally platform independent – they can be implemented using any programming language and run on any hardware platform or operating system. Components can be flexibly arranged into applications and collaborate over the Internet as well as corporate intranets. OPC UA is considered one of the most promising incarnations of WS technology for automation. Its design takes into account that industrial communication differs from regular IT communication: embedded automation devices such as PLCs provide another environment for Web-based communication than standard PCs.

The concepts of OPC UA include enabling Version Transparency in a system with a high diversity of components. OPC UA complements the existing OPC industrial standard by adding two fundamental components: different transport mechanisms and unified data modeling [19]. Scalability, high availability, and Internet capability open up many possibilities for new cost-saving automation concepts. Alternative platforms, including typical embedded (systems) operating systems, can be accessed directly, eliminating the need for an intermediate Windows PC to run the OPC Server.

For this paper, the most important aspects of OPC UA are the parts Address Space and Information Modeling

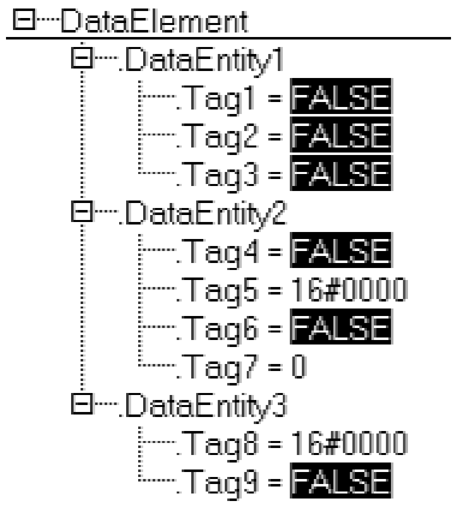


Figure 3: Data Encapsulation

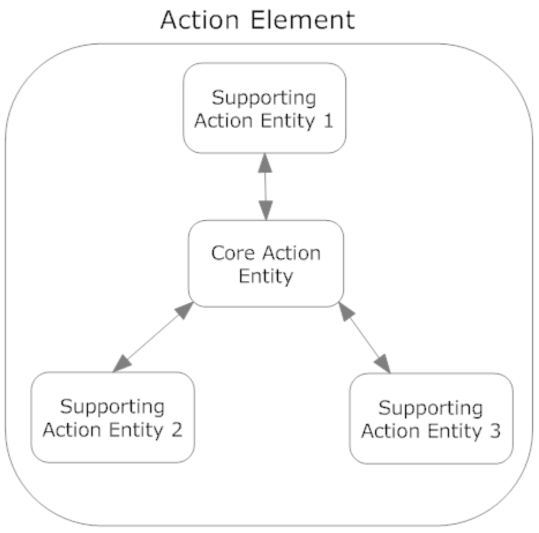

Action Element
Supporting Action Entity 1
Core Action Entity
Supporting Action Entity 2
Supporting Action Entity 3

Figure 4: Action Encapsulation

of the OPC UA specification [6]. Indeed, these models include interesting concepts on converting our IEC 61131-3 based ISA88 Control Module to an evolvable black box module, accessible by an OPC UA-based interface [20], complying with the principles of Normalized Systems. This enables the use of automation control building blocks over a standardized network, even the Internet. As a result, PLC projects, which are typically intra-process, can become inter-process with a higher potential of production system integration.

IV. Evolvable Control Modules

A. Design concept

The Normalized Systems theory defines five encapsulations of software entities. These encapsulations are defined in a fundamental way, and further worked out in the form of design patterns. These design patterns are exemplified in the background technology of the Java programming language [3]. We propose an interpretation of the fundamental encapsulations for the IEC 61131-3 programming environment. When we base the design of a Control Module on the Normalized Systems theory, we propose 3 building components of a Control Module. In this paper, we focus on Data Encapsulation, Action Encapsulation, and Connection Encapsulation. The latter is a special case of Action Encapsulation:

- Data Encapsulation: The composition of software entities, i.e., encapsulating all tags of a larger building block into a single data element, implies that only one (complex) parameter shall be passed to and returned from this building block. Note that in an IEC 61131-3 program, the use of *structs* can tackle the problem of adding extra parameters. By extending the struct, all parts of the struct, old and new fields, remain visible and accessible by every entity. We apply this concept to the core module of this paper, an ISA88 Control Module. Inside the borders of an IEC 61131-3 project, this leads to a “struct” for the whole production control device, containing smaller “substructs” for every action or task in that device. There is no straightforward concept of data hiding available in IEC 61131-3, so we cannot hide the new fields in such a struct for older entities. However, this is not causing data type conflicts, because in IEC 61131-3 no runtime construction of instances is supported, so all data instances of this complex parameter have the same type structure. On the inter-process level, different type instances are possible, but we can use data hiding based on OPC UA, where an interface is made for, e.g., SCADA or MES software, which can run on several technologies or platforms (inter-process). We define the entire parameter, main struct and substructs together, as a *Data Element* (Figure 3).
- Action Encapsulation: the composition of software entities to encapsulate all Action Entities into a single ISA88 Control Module implies that the core action (state machine of the Control Module) shall only contain a single functional task, not multiple tasks. In concept, we consider one core task, surrounded by supporting tasks (Figure 4). Arguments and parameters of the larger building block (ISA88 Control Module) should be an aggregation of all encapsulated Data Entities: the single complex datastruct or Data Element (Figure 3). We define the larger building block,

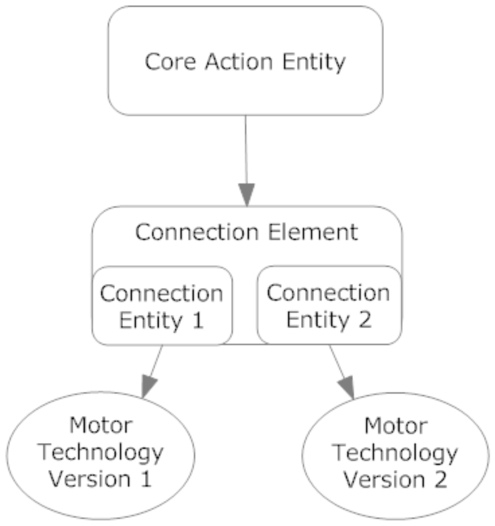


Figure 5: Connection Encapsulation

encapsulating the core Action Entity together with supporting Action Entities as an *Action Element*.

- Connection Encapsulation: Typically, three actors are interacting with this Action Element as an implementation of an ISA88 Control Module: An Equipment Module (automatic mode, recipe based control interface), the operator (manual mode, low level HMI: Human Machine Interface) and the process equipment hardware (e.g., a motor, valve or instrumentation device). Each of these actors is considered to represent an external technology, in possible new versions or even alternative technologies over time. The (supporting) Action Entities of the Control Modules, which are handling the connection of these actors with the core action, are defined as *Connection Entities*. In case of multiple versions of this special kind of Action Entities, every connection is an encapsulation of several versions or alternative technologies. We define such a Connection Encapsulation as a *Connection Element* (Figure 5). This encapsulation implies that the corresponding Data Element has a subsubstruct for each Connection Entity related to the substruct of the Connection Element.

The study of Flow Elements and Trigger Elements is outside the scope of this paper. Remember that IEC 61131-3 Programs are triggered by the PLC system. Consequently, Trigger Elements are integrated in the configuration part of the IEC 61131-3 environment. Following the ISA88 modeling rules, Control Modules should not contain procedures, so Flow Elements are not applicable in an ISA88 Control Module.

B. Anticipated Changes

In the design of evolvable Control Modules, we want to create a module which is stable with respect to a defined set of anticipated changes. We distinguish high-level changes and elementary changes. A lot of engineers know only the high-level changes, which are either real-life changes or changes with respect to implementation related aspects. These changes reflect additional functional requirements, which can typically be found in requirements documents or new customer requests. The elementary changes are related to software primitives, and formulated in terms of software constructs. One additional functional requirement corresponds to at least one elementary change, but, more probably, several elementary changes. First, we discuss the elementary anticipated changes of software primitives, and second, we discuss how real-life changes can be translated into these elementary changes. The elementary anticipated changes are:

- A new version of a data entity
- An additional data entity
- A new version of an action entity
- An additional action entity

Remember we make an aggregation of several data tags into an IEC 61131-3 struct, with a substruct for every Data Entity corresponding with an Action Entity. Extending a substruct with one or more tags corresponds with the change “A new version of a data entity”; adding a new substruct corresponds to the change “An additional data entity”. The core task of a Control Module is a hardware device driver. This core task is surrounded by supporting tasks, like manual/automatic control, simulation, Permissives, alarm. The introduction of a new surrounding task corresponds with the change “An additional Action Entity”. A change of the functionality of a task corresponds with the change “A new version of an Action Entity”.

An experienced programmer should be able to transform real-life changes into changes of software primitives. However, in a team where inexperienced engineers do the coding, a “change architect” should fulfill this task. The systematic translation of high-level requirements into the more elementary form is outside the scope of this paper, but we do provide some examples in Section V.

C. Managing action versions

To comply with the third theorem of Normalized Systems, Action Version Transparency, we distinguish three cases: Transparent Coding, Wrapping Functionality and Wrapping External Technologies. Each of them is another approach, but provides a similar result to ‘the outside’.

1) Transparent Coding: Since Normalized Systems require a high granularity, it is not unexpected that the individual (small and straightforward) modules or subroutines end up to be a simple piece of code, on which the programmer has a clear overview. In such cases, the programmer can preview the effect of a functional change on the previous version(s). If the change is not contradictory with one of the previous versions, it might be possible to apply Transparent Coding. This means the new functionality can just remain in the module without affecting the original code, even if a calling entity is not aware of the new functionality. We provide some examples in Section V.

2) Wrapping Functionality: There will be lots of cases where Transparent Coding is not possible, because the code is too complex for the programmer to have a reliable overview, or if the new functionality is contradictory with one or more of the previous versions. To exhibit Action Version Transparency, the different versions can be wrapped. The calling action has to inform the called action which version should be used, by way of a version tag. In addition, following the 'Separation of States' principle, the called action has to inform the calling action whether the (type version of the) instance of the called action is recent enough to perform the requested action version.

An Action Entity which is designed according to the concept of Wrapping Functionality is aggregating all the versions as separate Action Entities, and is therefore called an Action Element. Each of the nested Action Entities contains a version of the core functionality. Following the 'Separation of Concerns' principle, the wrapping Action Entity (Action Element) should not contain any core functionality, but is limited to wrapping the versions as a kind of supporting task.

3) Wrapping External Technologies: It is very unlikely that an external technology complies with the Normalized Systems theorems. On the contrary, Lehman's Law of increasing complexity probably applies in this external technology. Consequently, we assume that lots of combinatorial effects and unbounded impacts are generated in case of any change in this external technology. We do not want to allow these effects and impacts to penetrate into our stable system. To isolate these effects, we use the concept of Connection Encapsulation. A Connection Entity is an Action Entity dedicated to doing nothing but mapping requests from an internal action to an external technology, and mapping the responses of the external technology to the calling internal action.

When there is a change in the external technology, this change might have effect on our Connection Entity. If it is a small update or hot fix, the Connection Entity could handle this change by way of Transparent Coding, but in general, the Connection Entity will remain dedicated to the version of the external technology that it was developed for. A new version of the external technology leads to the introduction of a new Connection Entity. An Action Entity, which is representing the core functionality of the external technology, has the task to wrap the different Connection Entities. This wrapping Action Entity is defined as a Connection Element. Again, following the 'Separation of Concerns' principle, the functionality of the wrapping entity should be limited to mapping requests from an internal action to a specific Connection Entity, and mapping the responses of the specific Connection Entity to the calling internal action. Every Connection Entity, which is part of the wrapped versions, should have a version tag of the instance of the external technology it is connected to. The calling internal action should inform the Connection Element (wrapping action) about which Connection Entity or external technology version is desired by way of a version tag. The Connection Element should inform the calling action whether the requested version is available.

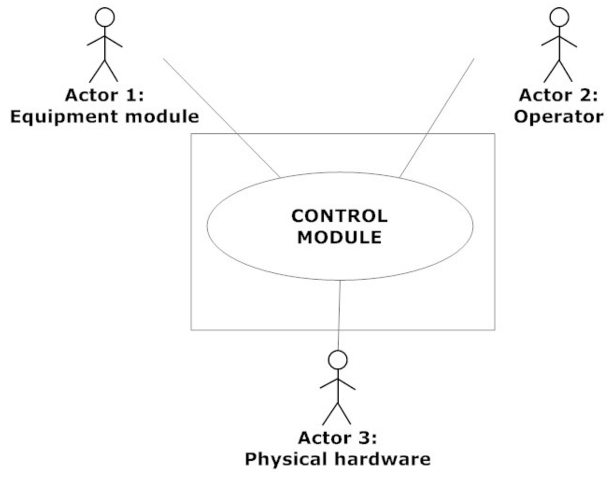


Figure 6: Actors on the Control Module [20]

With the concept of Wrapping External Technologies, the separation of versions is done by wrapping Connection Entities, each representing a version of the external technology. This concept can be easily extended with the introduction of Connection Entities, representing alternative external technologies. Every new version of an alternative external technology leads to the introduction of a new Connection Entity.

D. Evolvable Control Module

In this section we introduce a Control Module for a motor. We aim at making this motor control software module as generic as possible. Instead of introducing new formalisms, we based our proof of principle on existing standards. For the modeling, we used concepts of ISA88 (IEC 61512), for interfacing, we used OPC UA (IEC 62541), and for coding we used IEC 61131-3. More specifically, we rely on the S88 design patterns [15] (derived from ISA88) because

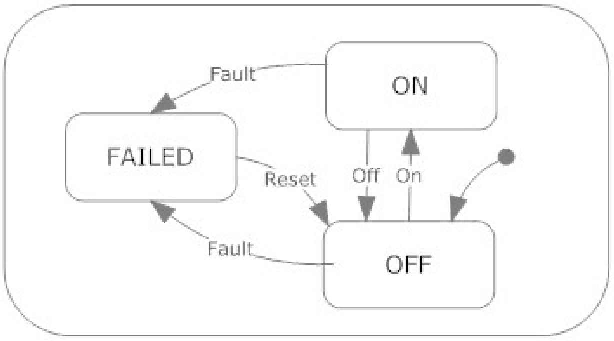
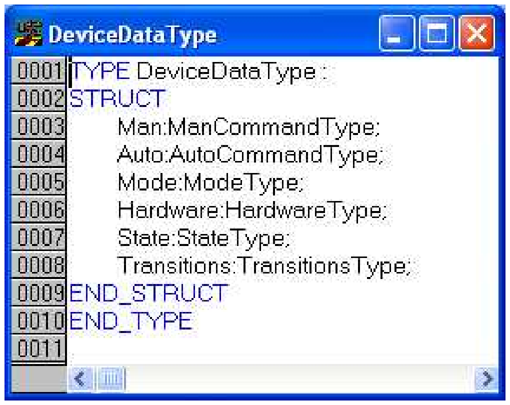


Figure 7: Example of a motor state model [1]



```
TYPE DeviceDataType :
STRUCT
    Man:ManCommandType;
    Auto:AutoCommandType;
    Mode:ModeType;
    Hardware:HardwareType;
    State:StateType;
    Transitions:TransitionsType;
END_STRUCT
END_TYPE
```

Figure 8: The Data Element

these patterns can be used not only in batch control, but also for discrete and continuous manufacturing. None of these standards contains prescriptive suggestions on how the internal code of a Control Module should be structured. We introduce a granular structure following the theorems of Normalized Systems. Every task (action), which must be executed by the Control Module, is coded in a separated Function Block.

In the most elementary form Control Modules are device drivers, but they provide extra functions. In our proof of principle we integrated the functionality manual/automatic mode and alarming. We kept the functionalities 'Interlocking' and 'simulation' as possible 'future changes', since it should be able to add them without causing combinatorial effects.

The proposed design of an evolvable Control Module contains *one* Data Element and *one* Action Element, which can include several Connection Elements. These Elements are implementations of Data, Action and Connection Encapsulation.

1) (One) Data Element: To make sure the interface of an action will not be affected in case of adding an additional tag or Data Entity, we work with one single struct and define this struct as a Data Element (Figure 3). The Data Element is a struct, which contains a substruct for every Data Entity.

2) (One) Action Element: The Action Element is a Function Block, which contains other Function Blocks, one for each Action Entity. The Action Element contains one core Action Entity, surrounded by supporting Action Entities. The tags controlled by each Action Entity belong to the corresponding substruct of the Data Element (Figure 4). An Action Entity can read all tags of the other substructs, but can only write in its own substruct (Data Entity).

3) Connection Elements: A Connection Element corresponds with a special kind of Action Entity in the sense that the change driver is an external technology, or, more generally, the change driver is coming from the outside of the Control Module. Typically, we have three actors on the Control Modules: the operator (low level HMI), the Equipment Module, which owns the Control Module, and a Process Hardware Device (Figure 6). Following the 'Separation of Concerns' principle, each connection has to be handled with a separate software module. If the device hardware has several versions, a Connection Entity is needed for every version (Figure 5).

In the following, we specify the software entities which represent the content of the Elements. We used the design pattern shown in Figure 7. This state machine is very simple. When the control system powers on, the motor enters 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 required 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 [15]. This functionality is implemented in a Function Block we call 'CoreStateAction'. This Function Block has only one parameter we call 'Device'. The datatype of this parameter is called 'DeviceDataType'.

4) Data Entities:

The 'DeviceDataType' is a struct containing 6 substructs (Figure 8). Four of them include information coming from 'the outside':

- Man: Manual commands. Contains the tags 'On' and 'Off', which allow the operator to start and stop the motor. Additionally, this substruct contains the tag 'Reset' to allow the operator to bring the status of the Control Module back to the initial state after a failure.
- Auto: Automatic commands. Contains the tags 'On' and 'Off', which allow an ISA88 Equipment Phase to start and stop the motor in automatic mode. In this

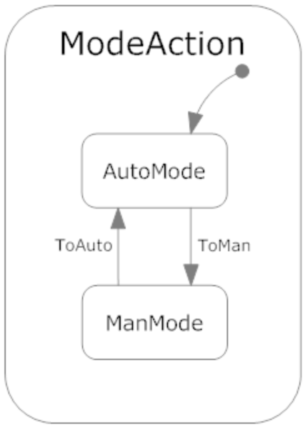


Figure 9: The Action Entity "ModeAction"

version, we choose that in Automatic Mode the reset functionality can not be accessed.
- Mode: The mode of the Control Module. Contains the tags 'ManMode' and 'AutoMode', which indicate the mode of the Control Module. Additionally, this substruct contains the tags 'ToMan' and 'ToAuto' to allow an entity from 'the outside' to switch the Mode between Manual and Automatic.
- Hardware: The Hardware tags. First, this substruct contains the tag 'Qout', which can be linked with a PLC output address to electrically control starting and stopping of the motor. Second, this substruct contains the tag 'FeedBack', which can be linked with a PLC input address to check whether the motor is physically running or not. Third, this substruct contains a tag 'Fault', which assumes the value 'TRUE' if the output is not corresponding with the feedback of the motor.

A fifth substruct 'State' contains the state data of the core state machine: 'On', 'Off' and 'Failed'. The transition tags 'ToOn', 'ToOff' and 'Reset' are placed in the sixth substruct 'Transitions'. These tags contain the results (output) of an Action Entity, which decides whether the operator or the Equipment Module has control (based on the mode).

5) *Action Entities:*

Our evolvable Control Module contains four Action Entities:
- ModeAction: Mode action (Figure 9). This is a state machine, which maintains the mode. Mode commands switch between manual mode and automatic mode. The inputs of this action are the mode commands of the substruct 'Mode'. The outputs of this action are both mode states of the same substruct.
- TransAction: Transition action (Figure 10). The inputs of this action are all requests of both automatic control and manual control entities, available in the substructs 'Auto' and 'Man', respectively. The outputs of this action are the tags in the substruct 'Transitions'.
- StateAction: Core state machine action (Figure 7). This is the state machine, which maintains the state of the Control Module. The inputs of this action are the transitions tags of the substruct 'Transitions'. The outputs of this action are the state tags of the substruct 'State'.
- HardwareAction: The hardware action (Figure 11). The inputs of this action are the states of the substruct 'State', and the input 'FeedBack' of the substruct 'Hardware'. The outputs are the tags of the substruct 'Hardware'.

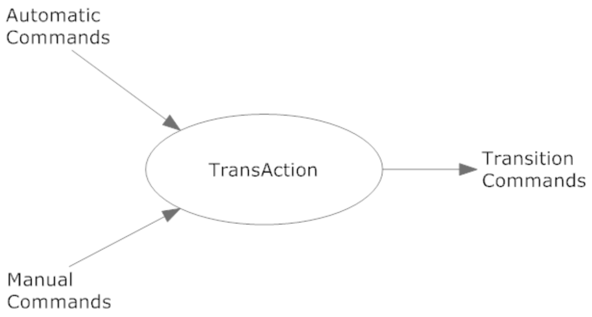


Figure 10: The Action Entity "TransAction"

Please note that the entity which is performing manual commands (typically the low level HMI), must check the tags of the mode state machine to check whether the manual commands will be accepted. In automatic mode, the manual commands will be ignored. Similarly, the ISA88 Equipment Module (automatic mode entity) must check that automatic mode is active before sending requests.

6) *Connection Entities:*

In fact, the action 'HardwareAction' is a Connection Entity, which connects the control software (with the core state machine as its central task) of the Control Module to physical production process hardware. Remember that a Connection Entity is a special case of an Action Entity. Indeed, in the first version of our proof of principle it maps the 'On' state of the core state machine to the hardware output 'Qout'. In addition, it checks if the value of 'Qout' corresponds with the input value 'FeedBack'. If there is a discrepancy, it sets the tag 'Fault' to inform the core state machine action that the hardware is not responding as expected.

In a second version, we connected a bidirectional motor to the Control Module. Consequently, we added a tag

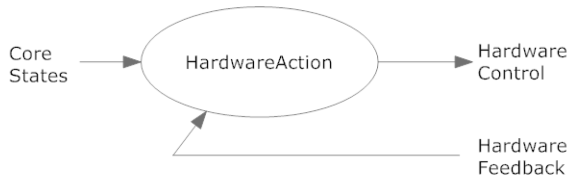


Figure 11: The Action Entity “HardwareAction”

‘Qreverse’ to the substruct ‘Hardware’. The Connection Entity sets this ‘Qreverse’ tag in case the core state machine action requests to run the motor in reverse direction. As a result, when the core state action requests the motor to run in the original direction, only the tag ‘Qout’ is set. When the core state action requests the motor to run in the reverse direction, both the tags ‘Qout’ and ‘Qreverse’ are set. Since the way of setting the tag ‘Qout’ is not changed, we could apply Transparent Coding. Instances of the Control Module which are connected to a unidirectional motor will just start and stop the motor and neglect the tag ‘Qreverse’ (which is initialized to the value ‘FALSE’).

In a third version, we have a bidirectional motor, but it is controlled differently. Instead of having a tag which controls whether the motor should run or not and another tag indicating the direction, we have two tags controlling a direction each. If one of them is TRUE, the other must be FALSE to provide an unambiguous command to the device. Obviously, since the interface to the device is changed, Transparent Coding is not possible. So for this third version, we used the wrapping concept. We added a tag ‘Version’ to the substruct ‘Hardware’. The code of the action ‘HardwareAction’ is moved to a new module called ‘HardwareActionV0’. The newly introduced code in the action ‘HardwareAction’ is a selection, associated with the version tag. If the version tag has the value ‘0’, the request to the action ‘HardwareAction’ is forwarded to the action ‘HardwareActionV0’. Besides, the Action Entity ‘HardwareAction’ could more appropriately be called ‘Connection Element’ now, because it is only selecting versions and mapping. Another new Function Block ‘HardwareActionV1’ contains the newly introduced functionality of the new motor. HardwareActionV0 and HardwareActionV1 are called Connection Entities.

To connect the automatic procedure (typically an ISA88 Equipment Phase) to our Control Module, no code is needed. Indeed, such a Phase (Figure 2) is typically coded in an ISA88 Equipment Module by way of the IEC 61131-3 language SFC. Since the Equipment Module, which has control over our Control Module, is coded into the same PLC as the Control Modules it owns, it only needs access to the instance of the Data Element ‘Device’.

Besides, even for manual control no IEC 61131-3 connection code is needed. Similarly, the low level HMI just needs access to the Data Element ‘Device’. However, since the low level HMI is located in an external technology, it would lead to the coding of a Connection Entity or Element. Thanks to the OPC UA IEC 61131-3 companion specification [21], it is expected that we will not need to code, but only configure the Connection Entity. Based on this OPC UA companion specification, software constructs of IEC 61131-3 can be mapped to OPC UA. Unfortunately, this companion specification is rather recent, and we could not find any commercial products supporting this standard at the moment of submitting this paper.

V. Changes and Evaluation

A way to test evolvability is adding changes and evaluating the impact of these changes. In general, we start with a first version. Then we maintain one or more running instances of the Control Module with the initially expected behavior. 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 behavior of the older instances as well.

We provide some examples for the transformation of high-level changes to the anticipated changes of software primitives.

- We consider the situation that manual operations could harm automatic procedures. For instance, stopping a motor manually could confuse an algorithm if it is happening during a dosing procedure. 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 such as an equipment module requires the non-interruptible (exclusive) use of the Control Module.
  In terms of elementary changes, this requires an additional version of a Data Entity, more specifically the addition of a tag ‘ManLock’ in the substruct (Data Entity) dedicated to receiving automatic commands. Additionally, a new version of an Action Entity is introduced. The action dedicated to select manual or automatic mode adds the ‘ManLock’ tag as a constraint to switch over to manual mode.
- We consider the situation of a motor instance where the motor must be able to run in two directions (while the functionality of earlier unidirectional motor instances should remain).
  In terms of elementary changes, this requires an additional version of three Data Entities. First, the addition of a tag ‘Reverse’ in the substruct (Data Entity) dedicated to hardware control. Second, the addition

of a tag 'ManReverseCmd' in the substruct dedicated to receiving manual commands. Third, the addition of a tag 'AutoReverseCmd' in the substruct (Data Entity) dedicated to receiving automatic commands. Moreover, the action which is processing the result (aggregation) of the requests of both manual and automatic commands needs a new version to provide a command 'ReverseCmd' for the core state machine action. Finally, the core state machine Action Entity needs a new version to add the new state 'ReverseState', accompanied by transitions from and to this new state.

- We consider the situation where one wants to introduce a simulation mode (for testing purposes), to neglect the Fault transition if no hardware is connected.
  In terms of elementary changes we need three changes. First, an additional Data Entity or substruct which can be used to store the state of the simulation mode. Second, an additional Action Entity to process the newly introduced state machine, and third, a new version of the core state machine Action Entity to neglect the Fault transition when in simulation mode.

Remember that manual operations could affect automatic procedures. In terms of elementary anticipated changes, this requires the addition of the tag 'ManLock' in the substruct 'Mode'. The action 'ModeAction' adds the 'ManLock' tag as a constraint to switch over to manual mode. We applied the concept of Transparent Coding. In the IEC 61131-3 data type declaration part of this additional tag in the substruct, we explicitly declared the initial value to be FALSE. We did not remove or change the calls of instances which do not need this feature. For older calls the behaviour did not change, and for the new instances we can indeed prevent the mode going to automatic.

We consider again the situation of a motor instance (as above) which must be able to run in two directions. In the previous section we discussed new versions of the Connection Entity 'HardwareAction', updated to the Connection Element 'HardwareAction', which is containing the two Connection Entities 'HardwareActionV0' and 'HardwareActionV1'. The elementary changes with regard to this Connection Element can be done without affecting the other actions or Data Entities. However, this does not mean that the related high-level changes or real-life requirements are met. Remember that *one* additional functional requirement corresponds typically to *more* elementary changes. For our two directions motor instance, a change of the core state machine was necessary, in addition to the elementary changes needed for the Connection Element. First, in the Data Entity (substruct) 'State', an additional state 'Reverse' was introduced. Second, in the data entity (substruct) 'Transitions', the tag

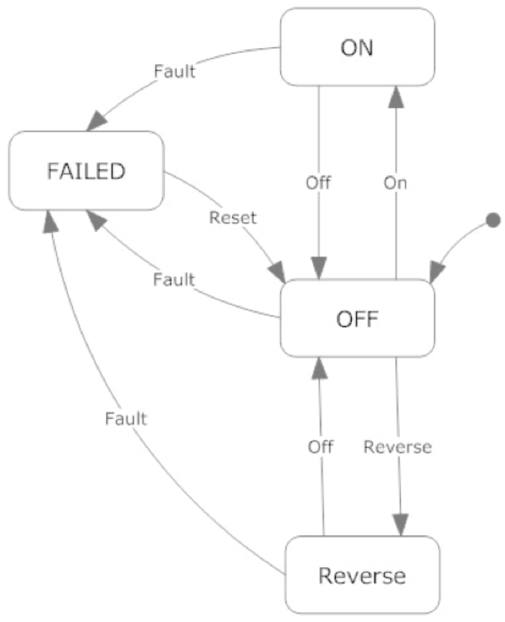


Figure 12: Core state machine bidirectional motor

'Reverse' was added. Third, the functionality of the Action Entity 'StateAction' was extended by way of Transparent Coding. Transparent Coding for a state machine means that no states can be removed, no states can be changed, and no transitions can be changed or removed. In other words, the allowed changes are only additions of states and transitions (Version Transparency principles). We end up with the new version of the state machine depicted in Figure 12.

We provide two more examples of Transparent Coding. Remember the manual lock feature. We can code the initialization of the new tag 'ManLock' to FALSE. By assuming this default value, the code of the change can be made in a way that this default value guarantees the behavior of the previous version. More precisely, if the value is FALSE, the manual lock functionality will not apply until a newer version instance is setting it to TRUE, which is not going to happen if an older version is used on this specific instance.

Remember the motor instance, where a new version can let the motor run in the reverse direction. Similarly, if the default version of the tag 'Reverse' is initialized to FALSE, the motor will run in the original direction until a newer version instance is setting the reverse tag to TRUE, which again is not going to happen if an older version is used on this specific instance.

VI. Conclusion and Future Work

Evolvability of software systems is important for IT systems, but also a relevant quality for industrial automation systems. IEC 61131 Function Blocks of automation systems are programmed close to the processor capabilities. For example, there is a similarity between the IEC 61131-3 language Instruction List (IL) and assembly. The key point of Normalized Systems is a high 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 Normalized Systems theory. In addition, this approach can be of great value for improving the quality of industrial automation software projects.

It must be stated that implementing these concepts was highly facilitated by the use of existing industrial standards. They provide us with methods to develop the macro-design of software modules, while the Normalized Systems theory provides 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) and data structures, the identification of the change drivers of the concerned entity is essential. This confirms the first theorem for software stability, Separation of Concerns.

Our future work will be focused on other (macro) elements of ISA88, which contain different types of control. A Control Module contains mainly basic control, together with limited coordination control (the mode). We will extend this study to elements with more advanced coordination control code and procedural control, again designed to comply with the Normalized Systems theory.

Moreover, future work will also be focused on interfaces. Since OPC UA is very generic, we wonder if constraints must be added to the standard to let data communication be compliant with the second theorem of software stability, Data Version Transparency. It will be interesting to investigate whether both currently existing OPC UA transport types, UA binary and UA XML, can be handled in a Data Transparent way.

REFERENCES

[1] van der Linden D., Mannaert H., and de Laet J., “Towards evolvable Control Modules in an industrial production process”, ICIW 2011, 6th International Conference on Internet and Web Applications and Services, pp. 112-117, 2011.

[2] International Electrotechnical Commission, “Programmable controllers - part 3: Programming languages”, IEC 61131-3, 2003.

[3] Mannaert H. and Verelst J., “Normalized Systems Re-creating Information Technology Based on Laws for Software Evolvability”, Koppa, 2009.

[4] McIlroy M.D., “Mass produced software components”, NATO Conference on Software Engineering, Scientific Affairs Division, 1968.

[5] van Nuffel D., Mannaert H., de Backer C., and Verelst J., “Towards a deterministic business process modelling method based on normalized theory”, International Journal on Advances in Software, 3:1/2, pp. 54-69, 2010.

[6] OPC Foundation. “OPC Unified Architecture”, www.opcfoundation.org.

[7] The Internation Society of Automation, “Batch Control Part 1: ”Models and Terminology”, ANSI/ISA-88.01, 1995.

[8] Mannaert H., Verelst J., and Ven K., “Towards evolvable software architectures based on systems theoretic stability”, Software, Practice and Experience, vol. 41, 2011.

[9] Kuhl I. and Fay A., “A Middleware for Software Evolution of Automation Software”, IEEE Conference on Emerging Technologies and Factory Automation, 2011.

[10] Eick S.G., Graves T.L., Karr A.F., Marron J., and Mockus A., “Does code decay? Assessing the evidence from change management data”, IEEE Transactions on Software Engineering, vol 32(5), pp. 315-329, 2006.

[11] Lehman M.M., “Programs, life cycles, and laws of software evolution”, Proceedings of the IEEE, vol 68, pp. 1060-1076, 1980.

[12] Mannaert H., Verelst J., and Ven K., “The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability”, Science of Computer Programming, 2010.

[13] van der Linden D. and Mannaert H., “In Search of Rules for Evolvable and Stateful run-time Deployment of Controllers in Industrial Automation Systems”, ICONS 2012, 7th International Conference on Systems, accepted for publication, 2012.

[14] Dijkstra E., “Go to statement considered harmful”, Communications of the ACM 11(3), pp 147-148, 1968.

[15] Brandl D., “Design patterns for flexible manufacturing”, ISA, 2007.

[16] van der Linden D., “Implementing ISA S88 for a discrete process with the Bottom-Up approach”, AGH - Automatyka 12/1, pp. 67-76, 2008.

[17] Hannelius T., Salmenpera M., and Kuikka S., “Roadmap to adopting OPC UA”, 6thIEEE International Conference on Industrial Informatics, pp. 756-761, 2008.

[18] Lange J., Iwanitz F., and Burke T.J., “OPC: von Data Access bis Unified Architecture”, VDE-Verlag, 2010.

[19] Mahnke W., Leitner S., and Damm M., “OPC Unified Architecture”, Springer, 2009.

[20] van der Linden D., Mannaert H., Kastner W., Vanderputten V., Peremans H., and Verelst J., “An OPC UA Interface for an Evolvable ISA88 Control Module”, IEEE Conference on Emerging Technologies and Factory Automation, 2011.

[21] PLCopen & OPC Foundation, “OPC UA Information Model for IEC 61131-3 1.00 Companion Specification”, 2010.