Industrial Automation Software: Using the Web as a Design Guide

Dirk van der Linden, Georg Neugschwandtner
Electromechanics Research Group
Artesis University College of Antwerp
Antwerp, Belgium
dirk.vanderlinden, georg.neugschwandtner@artesis.be

Herwig Mannaert
Department of Management Information Systems
University of Antwerp
Antwerp, Belgium
herwig.mannaert@ua.ac.be

***Abstract*—When looking the World Wide Web (and the Internet at large) as one giant application, we observe certain desirable properties that would also be welcome, but cannot be taken for granted, in industrial automation systems. In particular, subtasks that are unrelated from a functional point of view (such as Web sites) are usually well-separated from each other. Web services can play a key role in bringing separation of concerns to automation systems. The paper introduces a relevant standard (OPC UA). It also presents the Normalized Systems theory as a structured way of ensuring separation of concerns which is applicable to a wide range of application domains, from the Web to programs within automation controllers.**

Keywords-OPC UA; Normalized Systems; Evolvability; Industrial Automation.

I. INTRODUCTION

Already before the Internet was adopted worldwide, and, in parallel, IP-based networks obtained their dominant role in the office world (no matter in which sector of the economy) [1], some authors have tried to figure out whether the Internet could 'crash' or not. Ted Lewis answered the question 'Is it possible for the Internet to overload and blow a fuse' with 'probably' [2], following a (simplified) comparison of the Internet with an 'infinite bus', which can indeed become overloaded – at least partly. Consequently, the threat has to do with a potential lack of hardware resources, or even physical unavailability in case of, e.g., a fire in one or more buildings. However, a crash of the entire Internet is considered improbable [3]. Still, this only applies to the entire Internet as a system; parts of this system, in particular individual application servers, certainly can crash. To prevent services becoming unavailable due to such *object* crashes, authors for example consider ways to implement highly-available distributed World Wide Web (WWW) servers [4].

Considering this, the Internet appears to be a *stable* system. In the literature, the meaning of *stability* varies [5]. For a more formal specification of this assumption for our purposes, we shall consider the generic concept of stability as defined in the fields of signal processing, systems theory, and control theory, BIBO (Bounded Input Bounded Output) stability. In these fields, (BIBO) stability is considered one of the most fundamental properties of a system [6], [7]. It implies that a bounded input function should result in bounded output values, even as $t \to \infty$ (with t representing time).

When we call the Internet a *system* in this sense, we focus on the fact that objects, services, clients and servers are being continuously added to it, updated and removed from it. We consider these changes as inputs to the (Internet) system, and consider the impact of those changes as outputs. The Internet apparently copes well with these continuous, worldwide changes. Their consequences are bounded: An object crash does not affect the robustness of the system as a whole. Also, the effort for making changes does not rise as the system evolves: For example, creating a new web site is not harder today than last year (given that the new site should provide the same functionality and given the same tools and availability of other resources). Of course, adding a more complex web site requires more effort than adding a simple one; but the effort required does not depend on the size of the web at large. The effort only depends on the size of the change itself.

For software systems in general, however, this is not the case. Industrial automation control systems are no exception [8]. Stability or long-term maintainability is a challenge. It is a desired characteristic, which is hard to control [5]. The issue of evolvability is widely known, and was already specified in the form of a statement back in 1980 by Manny Lehman: 'As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it' [9].

Thus, the Internet appears to be stable, while the software systems, which make part of it, are not. This paper presents approaches towards improving this situation, focusing on separation of concerns in automation systems by applying web based technology and appropriate formal guidelines. Section II outlines the desirable properties we find in the Internet as a system and how automation systems are different in contrast. Section III introduces the OPC Unified Architecture standard as a way of applying web based technology to automation systems. Section IV summarizes the main theorems of the Normalized Systems theory, which provides formal guidelines to building stable software systems independent of the application domain, and Section V

 ISBN: 978-1-61208-200-4

discusses how to migrate industrial automation systems to normalized systems. In Section VI, we conclude and present suggestions for future research.

II. The Web as a model for separation between automation software components

When looking at the world wide web, we notice various desirable properties. We already mentioned the example of web sites being independent of other web sites (as long as they do not depend on each other for content) regarding the effect of crashes as well as the effort required for launching a new website. Also, a web browser is not going to crash when a server does not respond in time. The Internet is robust against changes, *without* the need for a specific maintenance effort as stated by Lehman's law [9]. We are seeing a huge, stable system where essential concerns are well separated. Broadly speaking, each web site could be considered such a concern.

While our examples for desirable properties found in the world wide web are probably so familiar that they do not seem to be a special achievement at first sight, it is worth remembering that these properties cannot simply be taken for granted. As a case in point, the situation is significantly different with industrial automation systems.

Industrial Programmable Logic Controllers (PLCs) are usually programmed in one of the languages of the IEC 61131-3 standard [10]. An IEC 61131-3 Program Organization Unit (POU) contains code, which can be a Function, a Function Block, or a Program. The code can be written in any one of these languages, depending on which of them is most appropriate for the specific application.

Often a PLC controlled factory plant starts out from a basic solution and is extended over time. For such a basic solution in a starting SME (Small or Medium-sized Enterprise) business, one single PLC might be enough to implement production control (or a self-contained subpart). If the business is successful and the production capacity has to be expanded, the resource limit of the single PLC will be reached at some point. Typically, a second one is then added, which will result in a number of couplings between these PLCs. In case engineers are focusing on functional requirements only and neglect non-functional quality properties (like evolvability and stability), there is a risk that these couplings could be so-called undesired couplings [11]. These will cause combinatorial effects, resulting in an increased impact of changes as the system size increases. Engineering an evolving system based on functional requirements only – and thus neglecting desirable maintenance activities with respect to the software – typically results in a system with a low amount of separable concerns. Without separation of concerns, it is likely necessary to shutdown the entire system if one of the many PLCs needs to be replaced; it could even be necessary to re-engineer the entire system. All this is due to the negative effect of coupling between POUs – in contrast to the loose coupling that we observe to be in place between web sites.

One of the very likely sources of undesired coupling is that popular communication systems between PLCs are based on global variables, shared memory, or shared I/O (Input/Output) addresses (e.g., fieldbus systems). The *existence* of these systems is not necessarily a disadvantage, but the *way of use* of these mechanisms can lead to invisible, hidden dependencies (also referred to as 'common coupling' in this case [11]), resulting in combinatorial effects.

Of course, it is possible to create IEC 61131-3 programs without causing common coupling (e.g., by carefully documenting the use of all global variables). However, instead of hoping that developers use these communication capabilities without causing common coupling (which would at least require thorough education), it would be preferable to have concepts in place that actually remove the possibility of common coupling [12].

Another relevant aspect in this context is the separation of states. A simple example where this is achieved on the world wide web would be the separation between web browser and server in the example at the beginning of this section. On the other hand, in industrial automation, some PLCs contain an integrated diagnostic system to handle fieldbus failures that can actually cause the PLC to shutdown if handling code is not properly implemented. In other words, instead of just notifying the PLC system of a fieldbus failure, a fieldbus system failure can be propagated to cause further system failures.

III. OPC Unified Architecture

Web services cleanly separate software components from each other. They enable self-describing, modular applications to be published, located, and invoked across the Web. Software modules collaborating via web services make parameters and arguments explicit through the module's interface only. They allow modules to be loosely coupled. Hence, combinatorial effects cannot propagate. In other words, web services isolate web based applications from each other. Therefore, we think that web services are a very helpful means toward avoiding common coupling as outlined in the previous section. We think that the structure they bring to systems by way of their design can contribute significantly to system stability.

The use of service oriented architecture and web services in manufacturing systems has been previously studied in depth by academic research groups [13], [14]. However, the lowest level of control (also called the shop floor layer) is characterized by a great heterogeneity of systems and special fieldbus communication solutions. Unless they support a common communication standard, the effort to integrate these shop floor systems into a web services based solution is prohibitive in industrial practice. Here, the OPC specifications could play a major enabling role.

 ISBN: 978-1-61208-200-4

The OPC Foundation set out to enable interoperability between automation equipment and software of various vendors. The first and still most successful specification, called OPC Data Access, was designed as an interface to communication drivers, allowing standardized read and write access to real-time data in automation devices. The major use case are HMI and SCADA systems accessing data from different types of automation devices. The first OPC specification family (released in the nineties) was based on the DCOM (Distributed Component Object Model) technology by Microsoft.

OPC has become the de facto standard for industrial integration and process information sharing [15]. By now, over 20,000 products are offered by more than 3,500 vendors. Kalogeras et al. share the view that OPC is highly accepted in industry [16]. Millions of installed OPC based products are used in production, process industry, building automation, and many other fields of application around the world [17]. However, when the Internet gained widespread adoption, the DCOM technology caused certain limitations.

To address this, the OPC UA specifications were released in 2006, aligning OPC with the principles of service oriented architecture. OPC UA is considered one of the most promising incarnations of WS technology for automation [18], [17], [19]. Its design takes into account that the field of application for industrial communication differs from regular IT communication: embedded automation devices provide another environment for Web-based communication than standard PCs.

OPC UA makes it feasible to provide a web services based OPC UA interface directly on automation controllers, with integrated protocol security. OPC UA can thus be used to encapsulate PLC programs much in the way that HTTP and HTML 'encapsulate' the implementation details of a web server. For this purpose, an OPC UA companion specification for representing IEC 61131-3 code using the information modeling mechanisms of OPC UA is available.

While separation of concerns can also be achieved by using proprietary, custom-made web services, using OPC UA instead has additional benefits. The most important one is that, being a standardized interface, it enables interoperability between automation systems by different vendors. Given the fact that web services interfaces will usually be provided by automation system vendors, the use of non-standard interfaces makes considerable integration efforts necessary in a multi-vendor system (Figure 1). Also, automation system integrators typically do not have the skills to develop custom-made webservices. Rather, they are trained to write IEC 61131-3 code and configure commercial SCADA products, often with a C-like scripting language. The OPC UA specifications are an excellent way to stimulate automation product vendors to provide standardized interfaces, which can be configured by automation system integrators in webbased or web-enabled automation software components.

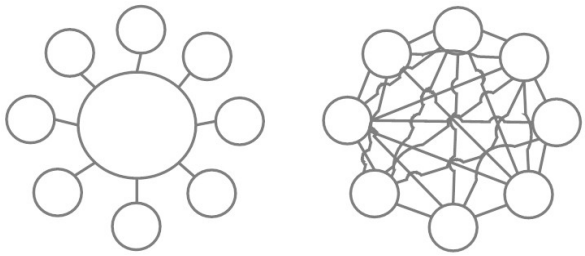

Figure 1. Left: OPC UA based communication; Right: Communication based on custom webservices

IV. Normalized Systems

The value of applying separation of concerns throughout a system (in our case, an industrial automation system) is apparent. Thanks to OPC UA, web services can be more easily applied to achieve loose coupling between PLCs, as well as between PLCs and SCADA systems. Still, unwanted couplings can also exist between POUs within a PLC. Moreover, applying web services technology is only one step towards achieving loose coupling; interfaces still have to be designed carefully to avoid unwanted functional dependencies. While comprehensive informal guidelines are available to educate software developers about good design, few formal contributions exist.

The Normalized Systems theory has recently introduced an approach to attain evolvable modularity in software, starting from a constructive point of view. Its authors state that probably all necessary knowledge is available to build stable software systems, but it seems to be hard to apply this knowledge [20]. The reason why it is very challenging to build stable software systems is not due to a lack of knowledge, but because this knowledge mainly takes the form of developers' individual experience. The Normalized Systems theory attempts to capture this knowledge in a succinct and formal way.

In Normalized Systems theory, instead of taking the functional requirements as the only starting point, elementary software constructs are defined as basic building blocks. Hence, the implementation of software can be seen as the transformation of functional requirements into software primitives. We can represent this implementation transformation γ as

$$\mathcal{S} = \gamma(\mathcal{R}) \quad [21].$$

In this formula, *S* represents the set of elementary software constructs, and *R* stands for the functional requirements.

In order to obtain evolvable modularity, the Normalized Systems theory states that this transformation should exhibit systems-theoretical stability. This means that a bounded input function (i.e., a bounded set of requirement changes) shall result in a bounded set of output values (i.e., a bounded

 ISBN: 978-1-61208-200-4

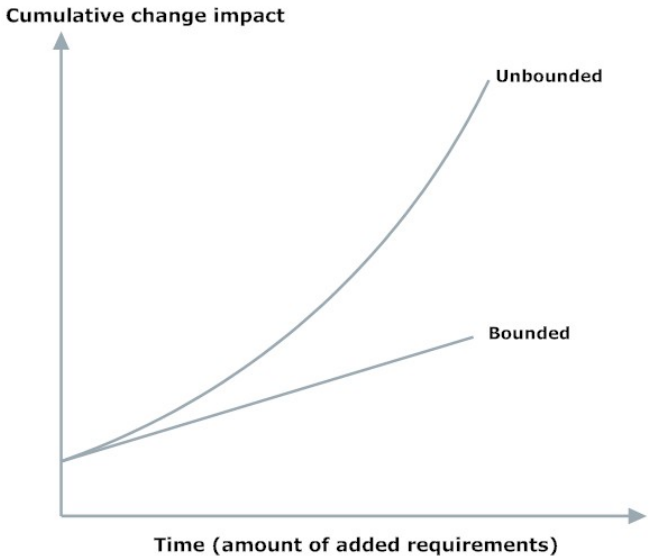

Figure 2. Cumulative impact of changes over time [24]

impact or effort), even if an unlimited systems evolution is assumed. The impact of a change shall only depend on the nature of the change itself. Conversely, changes causing impacts that are dependent on the size of the system are termed *combinatorial* effects; they must be eliminated from the system in order to attain stability. Stability in this context amounts to a linear relationship between the amount of requirements on the system (which is increasing as the system evolves over time) and the effort required to implement all these requirements. Systems that exhibit stability are defined as *Normalized Systems* [20]. In such a system, the effort required for adding a function with given complexity does not depend on the overall system complexity. This is in contrast to the situation typically observed in software projects, where combinatorial effects or instabilities cause this relationship to become exponential (Figure 2).

Mannaert et al. also proposed that, in Normalized Systems, modular structures should strictly adhere to the following principles [21]:

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

   Each task must be able to evolve independently. If it is expected that two or more aspects of a program function (i.e., tasks) will evolve independently, they must be separated. It is proven that if one module contains more than one task, an update of one of the tasks requires updating all the others, too. Therefore, Normalized Systems shall be constructed of Action Entities (independent code modules) dedicated to one core activity.

   Most discussions regarding the separation of concerns remain vague about what a concern is actually. In contrast, the Normalized Systems theory provides an explicit definition by introducing the concept of so-called change drivers: A module should not contain parts that can evolve separately; rather, these parts should be placed in separate modules.

2) Data Version Transparency: *Data Entities that are received as input or produced as output by action entities need to exhibit Version Transparency.*

   Data Version Transparency requires that Data Entities (variables, records) can exist in multiple versions without affecting the actions that consume or produce them. In other words, an update of a Data Entity shall not affect the interface of an Action Entity, i.e., it must be possible to use different versions of this Data Entity in the same way when exchanging parameters or arguments with an Action Entity.

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

   Action Version Transparency implies that an Action Entity can have multiple versions, without affecting the way this Action Entity is invoked. In other words, introducing a new version of an Action Entity or task shall not require changes to any other Action Entities calling (or called by) the Action Entity containing the task.

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

   This theorem focuses on the interaction of Action Entities with other Action Entities, more specifically, on the aggregation and mutual invocation of Action Entities in order to perform a function encompassing multiple tasks. State Keeping requires every Action Entity to be itself responsible for keeping track of its requests to other Action Entities. If results are not returned as expected, the calling action entity must not block indefinitely; rather, it shall handle the exceptional situation in an appropriate way.

V. MIGRATION AND REWRITES

Already decades ago, Doug McIlroy called for software building blocks which can be safely regarded as black boxes [25]. Such blocks should not contain any undesired, hidden dependencies. A truly Normalized System fulfils this requirement. However, such a system must comply with the Normalized Systems theorems from the ground up, down to the smallest building blocks or modules. Since each Action Entity may only contain one task, this leads to very fine-grained structure with an enormous amount of very small modules.

 ISBN: 978-1-61208-200-4

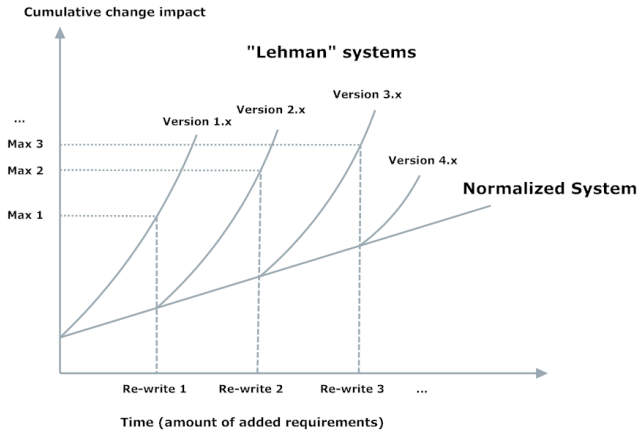

Figure 3. Reduction of cumulative effort by way of a rewrite

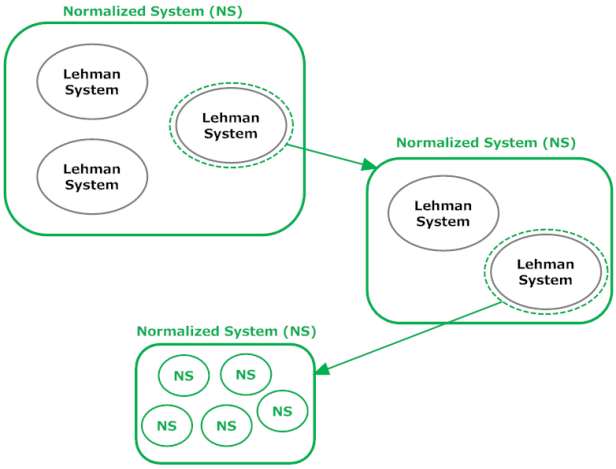

Figure 4. Migration from Lehman to Normalized subsystems

Restructuring a legacy ('Lehman') system into a Normalized System is, of course, a formidable task. A good start could be just separating the system into larger modules or subsystems with normalized interfaces. Each subsystem could then be considered to be a Normalized System, although it will likely still contain non-normalized subsystems (larger modules; see Figure 4). Such a subsystem module will not be a McIlroy-type safe black box due to its Lehman-type subsystems containing hidden dependencies. However, the combinatorial effects originating from its Lehman-type subsystems are stopped by the module interface, which complies with the Normalized Systems theorems.

In the case of automation components, web services (possibly making use of OPC UA) could be a very helpful mechanism to separate PLCs to transform them into subsystems with normalized interfaces. However, this is not enough. The internal modules of the subsystem, that is, all IEC 61131-3 code, must eventually be structured following the principles of Normalized Systems to make the subsystem a truly stable system.

When discussing the web as a design guide earlier in this paper, we observed that objects on the web, i.e., web sites, still can 'crash'. The same is true for a PLC which uses code that does not follow the principles of Normalized Systems. Both the web site and the PLC are still complex subsystems that, eventually, need to be rewritten, probably by again splitting them into different subsystems.

In general, the first step in a migration scenario to let a non-normalized system evolve into a normalized one would therefore be to identify parts that can be readily isolated from the remaining parts. After adding a normalized interface, each of the isolated parts can be replaced by a normalized software re-write. Such a well-performed maintenance activity or 're-write' will reduce the combinatorial effects within the system or subsystem (visible in Figure 3 as discontinuities along the y-axis). If full normalization is reached with the re-write, combinatorial effects will be removed entirely and permanently. Since the original part was already isolated via a normalized interface, the 'version change' brought about by the re-write will not cause combinatorial effects. Thus, once the parts of the overall integrated system have been isolated from each other (e.g., by way of web services), every Lehman-type subsystem can be transparently replaced by a Normalized one.

VI. Conclusion and Outlook

The timeline proves that the designers of the technologies behind the Internet applications we know today could not be aware of the Normalized Systems theory. However, the Internet as a system complies with the principles of this theory surprisingly well. Of course, the Normalized Systems theorems are not completely new, but have been available for a long time, albeit in the form of tacit knowledge. Those designers did a remarkable job following their intuition, and realised a world wide system that is stable even if the application objects within this system are not – thanks to loose coupling and excellent separation of concerns.

We are convinced that the Normalized Systems theory aids in achieving loosely coupled systems. It promotes the development of extremely fine-grained subsystems. As a first step on this way, isolating combinatorial effects in automation systems can be achieved by introducing web services based interfaces. These interfaces separate technologies, platforms, and vendor-dependent products; more generally spoken, they apply the Separation of Concerns principle to automation systems. OPC UA has a promising role in this regard thanks to the widespread adoption of OPC in industry.

Web-based or Web-enabled automation components can be regarded as a black box or isolated subsystem through an OPC UA interface [19]. If we manage to find concepts to restructure IEC 61131-3 code into very many small components, the IEC 61131-3 information model OPC UA companion standard could be applied to make this structure

 ISBN: 978-1-61208-200-4

transparent. We do not see a reason why the amount of such subsystems, interconnected via web services, would be limited, except for limited hardware resources. Having a large region system, comparable with the Internet, that interconnects automation control subsystems, might become very valuable in the future smart grid. Indeed, the Internet, which is mainly used for interconnecting information sources, might be extended with OPC UA based capabilities for interconnecting production control resources.

Certainly, it will be essential to further investigate which mechanisms the web and web services are built on that are responsible for the desirable system properties mentioned and translate them into the industrial automation domain.

REFERENCES

[1] T. Sauter and M. Lobashov, "How to Access Factory Floor Information Using Internet Technologies and Gateways," IEEE Trans. Ind. Inform., vol. 7, no. 4, Nov. 2011, pp. 699–712.

[2] T. Lewis, "Netstorms: the crash of '96," Computer, IEEE Computer Society , vol. 29, no. 11, Nov. 1996, pp. 12–14.

[3] R. Cohen, K. Erez, D. ben-Avraham, and S. Havlin (2000), "Resilience of the Internet to random breakdowns," Physical Review Letters, vol. 85, no. 21, Nov. 2000, pp. 4626–4628.

[4] R. Baldoni, S. Bonamoneta, and C. Marchetti, "Implementing Highly-Available WWW Servers based on Passive Object Replication," 2^{nd} IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99), Saint-Malo, 1999, pp. 259–262.

[5] D. Kelly, "A study of design characteristics in evolving software using stability as a criterion," IEEE Transactions on Software Engineering, vol. 32, no. 5, pp. 315–329, May 2006.

[6] L.B. Jackson, "Digital Filters and Signal Processing," 2^{nd} ed., Kluwer Academic Publishers, Norwell, MA, 1988.

[7] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, "Signals & Systems," 2^{nd} ed., Prentice-Hall, Upper Saddle River, NJ, 1996.

[8] H. P. Breivold, I. Crnkovic, R. Land, and M. Larsson, "Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study," The Third International Conference on Software Engineering Advances, Oct. 2008 (ICSEA '08), pp. 205–213, 2008.

[9] M. M. Lehman, "Programs, life cycles, and laws of software evolution," Proceedings of the IEEE, vol. 68, pp. 1060–1076, 1980.

[10] International Electrotechnical Commission, "IEC 61131-3, Programmable controllers – Part 3: Programming languages," 2003.

[11] D. van der Linden, H. Mannaert, and P. De Bruyn, "Towards the Explicitation of Hidden Dependencies in the Module Interface," ICONS 2012, 7^{th} International Conference on Systems, accepted for publication, 2012.

[12] D. van der Linden, H. Mannaert, and J. de Laet, "Towards evolvable Control Modules in an industrial production process", ICIW 2011, 6^{th} International Conference on Internet and Web Applications and Services, pp. 112–117, 2011.

[13] F. Jammes, H. Smit, "Service-Oriented Paradigms in Industrial Automation," IEEE Transactions on Industrial Informatics, vol. 1, no. 1, 2005.

[14] P. Spiess et al., "SOA-based integration of the Internet of Things in Enterprise services," IEEE International Conference on Web Services, Los Angeles, USA, July 6–10, 2009.

[15] T. Hannelius, M. Salmenpera, and S. Kuikka, "Roadmap to adopting OPC UA," 6^{th} IEEE International Conference on Industrial Informatics, pp. 756–761, 2008.

[16] A. P. Kalogeras, J. V. Gialelis, C. E. Alexakos, M. J. Georgoudakis, S. A. Koubias,"Vertical Integration of Enterprise Industrial Systems Utilizing Web Services," IEEE Transactions on Industrial Informatics, vol. 2, no. 2, 2006.

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

[18] W. Mahnke, S. Leitner, and M. Damm, "OPC Unified Architecture", Springer, 2009.

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

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

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

[22] OPC Foundation, "OPC Unified Architecture," www.opcfoundation.org.

[23] International Electrotechnical Commission, IEC 62541-1, "OPC unified architecture – Part 1: Overview and Concepts," 2010.

[24] D. van der Linden and H. Mannaert, "In Search of Rules for Evolvable and Stateful run-time Deployment of Controllers in Industrial Automation Systems," ICONS 2012, 7^{th} International Conference on Systems, Reunion, pp. 67-72, 2012.

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

 ISBN: 978-1-61208-200-4